UNIVERSITÉ JOSEPH FOURIER – GRENOBLE 1
SCIENCES & GEOGRAPHIE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Spécialité : «Systèmes et Communication »
École doctorale : «Mathématiques-Informatique »

Présentée et soutenue publiquement

par

Marc-Antoine LACOSTE

Le 22 octobre 2003

Une machine virtuelle répartie

pour la programmation de processus mobiles

Directeur de thèse

Sacha KRAKOWIAK

Composition du jury

Président : Roland BALTER
Rapporteurs : Jean-Jacques LEVY
Elie NAJM
Examinateurs : Sacha KRAKOWIAK (directeur)
Pierre CREGUT
Jean-Bernard STEFANI

Thèse préparée au département d'**Architecture de Systèmes Répartis**
Direction des Techniques Logicielles
France Télécom R&D

Remerciements

Je tiens à exprimer ma gratitude à M. Roland BALTER, professeur à l'Université Joseph Fourier et vice-président de la société ScalAgent Distributed Technologies pour avoir accepté de présider ce jury de thèse. Sa vision des plates-formes à objets répartis et du middleware m'a été précieuse pour l'approche de cet univers à l'aide de techniques formelles.

Je remercie vivement MM. Elie NAJM et Jean-Jacques LÉVY, respectivement professeurs à l'Ecole Nationale Supérieure des Télécommunications et à l'Ecole Polytechnique, d'avoir accepté d'être mes rapporteurs. Leurs discussions passionnantes à propos des calculs de processus et des technologies de l'information, et la justesse de leurs remarques ont constamment soutenu mes efforts au cours du développement de ce travail.

Je remercie chaleureusement M. Sacha KRAKOWIAK, professeur à l'Université Joseph Fourier, pour avoir encadré ce travail de thèse. J'ai particulièrement apprécié ses conseils judicieux et sa grande disponibilité à mon égard. Tout au long de ces années, il m'a soutenu, remotivé quelquefois, et remis sur la bonne voie tout en me laissant une grande liberté d'action. Il m'a guidé pas à pas dans le monde de la recherche avec une patience remarquable. Son regard neuf sur le travail en cours fut inestimable. Plus que tout, j'ai pris grand plaisir à écouter ses conseils et à partager son expérience.

Je souhaite également exprimer ma gratitude à M. Jean-Bernard STEFANI, responsable du laboratoire d'Architecture de Systèmes Répartis de France Télécom R&D, puis du projet SARDES à l'INRIA. Jean-Bernard est l'un des créateurs du M–calcul. Il m'a communiqué sa passion pour la recherche en me faisant partager sa vision large et profonde de la programmation répartie et mobile. A travers la notion extrêmement riche de domaine, il m'a fait découvrir l'univers passionnant des architectures de systèmes répartis. Je lui en suis particulièrement reconnaissant.

Je tiens à également exprimer ma vive reconnaissance à M. Pierre CRÉGUT, ingénieur de recherche et développement à France Télécom, pour avoir accepté de participer à ce jury, et pour les discussions que nous avons pu avoir ensemble concernant le λ–calcul, la programmation fonctionnelle et les architectures de sécurité.

Je remercie vivement Mme Kathleen MILSTED, responsable du laboratoire d'Architecture de Systèmes Répartis à France Télécom R& D, pour son aide tant technique que d'encadrement au cours de mes quatre années de thèse. J'ai eu beaucoup de plaisir à travailler avec Kathleen sur l'expérimentation DCP, dans la transcription des propriétés formelles d'un modèle vers une plate-forme à objets répartis.

Je tiens à remercier Mme Florence Germain avec qui j'ai eu plaisir à travailler sur la définition des machines abstraites et virtuelles du M–calcul. J'ai particulièrement apprécié les travaux que nous avons menés ensemble sur les modèles de programmation à base de domaines. Je la remercie également de m'avoir communiqué ses remarques à propos de ce manuscrit.

Cette thèse s'est déroulée dans le cadre des projets RNRT MARVEL et IST MIKADO. Les participants à ces projets avec qui j'ai eu la chance de collaborer m'ont permis de vivre une expérience enrichissante et stimulante. Je souhaite remercier ici Ilaria CASTELLANI, Gérard BOUDOL, Jean-François MONIN, Rocco DE NICOLA, Vasco VASCONCELOS, Matthew HENNESSY, Fabrice LE FESSANT, Alan SCHMITT, Lorenzo BETTINI, Luís LOPES, Andrea VALENTE, ainsi que Bruno DUMANT, François HORN et Fabien DELPIANO de l'équipe JONATHAN.

Je tiens à remercier spécialement Davide SANGIORGI, Cédric FOURNET et LUCA CARDELLI pour l'attention qu'ils ont accordé à mon travail. Ils m'ont prodigué de précieux conseils et fait partager leur vision des calculs de processus et leur implantation sous forme de machines abstraites.

Je remercie également l'ensemble des personnes du laboratoire d'Architecture de Systèmes Répartis à France Télécom R & D pour les nombreuses discussions techniques et amicales que nous avons eues, mais aussi pour leur esprit d'équipe et la bonne ambiance qu'ils y font régner. Merci en particulier à Jean-Philippe FASSINO, François-Gaël OTTOGALLI dont je partage le bureau, pour leur bonne humeur, leurs encouragements et les discussions animées concernant les systèmes d'exploitation. Merci à l'ensemble des « thésards » du laboratoire, en particulier Mikaël BEAUVOIS, Romain LENGLET, Jean-Charles TOURNIER et Mourad ALIA, mais aussi tous les autres, pour leurs contributions et les moments de détente que nous avons partagés.

Je tiens enfin à remercier mes amis et mes parents pour leur soutien sans faille et leurs encouragements sans lesquels ce travail n'aurait pas été possible.

Résumé

La construction d'infrastructures fiables de programmation répartie et mobile sur des réseaux à large échelle reste un défi majeur. Dans ce nouvel environnement d'exécution appelé « global computing », deux difficultés doivent être surmontées: un manque de cadre de référence pour les modèles et primitives de programmation sûre ; et une divergence entre théorie et pratique de la programmation répartie, se traduisant par une intégration insuffisante des méthodes formelles avec le génie logiciel et par une sémantique des infrastructures middleware déployées hautement implicite et donc peu fiable.

Dans cette thèse, nous tentons d'apporter une réponse à cette double problématique en explorant l'intérêt de la notion de *domaine* comme abstraction fondamentale possible pour le « global computing ». Unité de composition et de contrôle, ce concept capture de manière uniforme les partitions dans un système réparti de grande taille. Il est de plus reflété de façon implicite dans la plupart des modèles de programmation répartie et des plates-formes middleware.

Nous montrons que la notion de domaine est directement implantable, donc utilisable comme primitive de programmation, contribuant ainsi à réduire la divergence entre théorie et pratique identifiée précédemment. Nous utilisons une méthodologie de construction d'infrastructures dignes de confiance et à base de domaines où la formalisation du système s'effectue par raffinements successifs: (i) définition d'un modèle formel de programmation répartie; (ii) spécification formelle d'une implantation du modèle par une machine abstraite; (iii) réalisation de l'implantation par une machine virtuelle répartie s'appuyant sur les technologies middleware actuelles.

Cette démarche est appliquée au cas d'un modèle de programmation répartie explicitement fondé sur la notion de domaine, le M–calcul. Ce calcul de processus réalise une combinaison originale de localités programmables, d'ordre supérieur, de mobilité et de liaison dynamique. Une machine abstraite répartie nommée CLAM est définie pour le M–calcul, puis implantée par une machine virtuelle nommée OYSTERS. L'évaluation qualitative de la démarche montre la viabilité du concept de domaine, l'enrichissement du modèle de programmation choisi avec cette primitive ne se traduisant que par une augmentation de complexité raisonnable, comparable au niveau du modèle et de son implantation.

Mots clés: infrastructures réparties fiables; domaines; méthodes formelles; raffinement; calcul de processus; machine abstraite; machine virtuelle; système réparti.

Abstract

Building reliable distributed infrastructures for mobile programming on wide-area networks still remains a major challenge. In this new research area, called « global computing », two main issues must be tackled: first, a lack of reference point for models and safe programming constructs in a global computing environment; second, the outbreak of two separate and slowly diverging visions of distributed programming. The first relies on an implementation-based approach of today's middleware technologies, often without much concern for underlying theoretical foundations. Hence, execution semantics remains highly implicit and provides few guarantees of reliability. The second focuses on elaborate theoretical formal models of distributed computation but has remained insufficiently integrated with software engineering techniques.

This dissertation attempts to reconcile those two opposite views of reliable distributed computing. We investigate the notion of *domain* as a possible abstraction for the foundations of « global computing ». Both a composition and a control unit, a domain offers a uniform view of spatial and logical partitions occuring in large distributed systems. This abstraction is both reflected – often implicitly – in most formal models of distributed computation and in middleware technology.

We show the notion of domain is directly implementable and can be used as an effective programming primitive, thus contributing to reduce the gap between theory and pratice of distributed systems. We use a simple methodology to build reliable domain-based infrastructures, where formalization consists in refinement steps: (i) definition of a formally-defined distributed programming model; (ii) formal specification of an implementation of the model using an abstract machine; (iii) implementation by a distributed virtual machine relying on current middleware technology.

We apply this methodology to a model of distributed computation explicitly based on domains: the M–calculus. This process calculus realizes an original combination of explicitly programmable localities, higher-order processes, mobility, and dynaminc binding features. A distributed abstract machine called CLAM is defined for the M–calculus, and implemented by a virtual machine called OYSTERS. A qualitative evaluation of the methodology shows the abstraction of domain is effective: the extension of a programming model with a domain primitive results in reasonable additional complexity, comparable in the model and its implementation.

Keywords: Reliable Distributed Infrastructures; Domains; Formal Methods; Refinement; Distributed Process Calculi; Abstract Machine; Virtual Machine; Distributed Computing.

Table des matières

Table des figures

Chapitre 1

Introduction

Contexte de l'étude

Avec l'expansion des technologies de l'information, les infrastructures réparties augmentent en taille et en complexité: on assiste à la mise en réseau massive d'une grande diversité d'entités communiquant sur une échelle planétaire. En terme de programmation, on entre dans l'ère du « global computing », caractérisé par un fort degré de répartition des ressources d'exécution, des réseaux hétérogènes et de capacités variables, et un environnement hautement dynamique manifestant diverses formes de mobilité. Ce brusque changement d'échelle implique des besoins sans précédent de fiabilité pour les infrastructures logicielles.

La construction d'infrastructures fiables de programmation répartie et mobile dans des réseaux à large échelle reste toujours un défi majeur. Jusqu'ici, la principale voie d'approche a été d'utiliser des méthodes formelles pour améliorer la fiabilité des systèmes, au niveau de la conception et du développement. Ces méthodes permettent de concevoir clairement des infrastructures, de raisonner de manière rigoureuse sur leurs propriétés et de valider les systèmes construits. Durant ces dernières années, les méthodes formelles ont été utilisées pour fiabiliser des systèmes principalement centralisés ou de petite taille, mais n'ont été que peu appliquées à des environnements de type « global computing ». Le problème de développement d'infrastructures fiables reste donc largement ouvert.

Pourquoi le problème est-il non résolu?

On se heurte à un double obstacle: d'une part, une insuffisance d'abstractions, de modèles et de primitives de programmation pour le « global computing » qui constitue un domaine de recherche nouveau. En pratique, cette lacune se traduit par un manque de plate-formes intégrées, et une diversité impressionnante d'infrastructures explorant des aspects particuliers de la programmation répartie, mais difficilement comparables; d'autre part, une divergence entre théorie et pratique de la programmation répartie, les méthodes formelles étant encore insuffisamment intégrées au monde du génie logiciel, parfois jugées trop coûteuses ou difficiles d'accès. De ce fait, la sémantique des infrastructures logicielles actuellement déployées reste souvent empirique ou peu explicitée.

Où en est-on aujourd'hui?

Les travaux concernant les fondements du « global computing » s'appuient sur une longue tradition en théorie des langages et des systèmes. On a ainsi assisté à l'émergence de concepts successifs s'adaptant à des contextes de programmation toujours plus larges. On peut par exemple citer la programmation séquentielle et fonctionnelle (λ–calcul, LISP, ML), puis concurrente et parallèle (π–calcul, architectures multi-processeurs), puis enfin répartie. En parallèle, d'autres paradigmes se sont orientés vers une programmation plus fiable de systèmes de plus grande taille. Par exemple, la programmation à objets rend le développement plus sûr en rassemblant code et données au sein d'une unité d'encapsulation. De même, la programmation par composants permet de mieux maîtriser le passage à l'échelle en définissant des unités logicielles composables entre elles. Simultanément, les systèmes répartis ont dû prendre en compte la mobilité à la fois des utilisateurs et des composants logiciels ou matériels des infrastructures, les programmes franchissant les frontières administratives ou logiques. On a pu également réaliser des calculs répartis de plus en plus volumineux (« Grid Computing »). Le Web apparaît alors comme une infrastructure d'exécution unique et planétaire (« Programming the Web »), mais pour laquelle manquent encore les techniques et outils de programmation sûre. Ce constat est également valable pour les autres réseaux à large échelle: on assiste à une divergence entre des modèles d'exécution répartie sophistiqués, mais parfois éloignés des réalités de l'implantation, et des plates-formes largement diffusées, mais dont la sémantique reste informelle ou hautement implicite, et donc peu fiable.

Objectifs et contributions de la thèse

L'interrogation qui motive notre étude est la suivante: est-il possible de réduire en partie cet écart en disposant d'abstractions suffisamment puissantes mais implantables pour modéliser la large variété de situations du « global computing »?

En cas de réponse positive, l'intérêt est double: on dispose alors d'un cadre théorique fort permettant de raisonner formellement sur les systèmes. De plus, on obtient une abstraction reflétée dans une grande variété de plates-formes réparties, susceptible d'être directement implantable en pratique et de donner lieu à une nouvelle primitive de programmation, et donc de servir de servir de base à la construction de systèmes pour le « global computing ».

L'objet de cette thèse est précisément d'apporter des éléments de réponse à cette question à travers la notion de domaine (par exemple d'administration). Cette abstraction modélise à l'aide d'une seule primitive la composition et le contrôle et traite de manière uniforme les différents types de partitions rencontrées dans un système réparti à large échelle. Notre objectif est de montrer que les domaines peuvent constituer une abstraction effective pour le « global computing », contribuant à réduire la divergence entre théorie et pratique de la programmation répartie.

Pour y parvenir, nous mettons en oeuvre une méthodologie simple pour la construction d'infrastructures réparties de grande taille à base de domaines et dignes de confiance. La formalisation s'y effectue par raffinements successifs, où, à chaque étape, les domaines sont au coeur de la démarche: en partant d'un modèle formel de programmation répartie fondé sur la notion de domaine, on définit une spécification formelle du modèle par une machine abstraite. Puis, cette spécification est implantée par une machine virtuelle répartie raisonnablement efficace, utilisant des briques logicielles de base couramment utilisées dans les plates-formes middleware actuelles.

La première contribution de cette thèse est d'illustrer l'intérêt des domaines comme primitive de programmation pour le « global computing ». Nous étudions les différentes manifestations de ce concept dans un certain nombre de modèles de programmation répartie existants, dans les machines virtuelles qui les implantent, ou encore dans les plates-formes middleware les plus répandues. Nous montrons comment les domaines, vus comme primitives d'un calcul de processus, c'est-à-dire comme localités programmables, sont raffinés dans des domaines d'exécution rencontrés dans les plates-formes à objets répartis, c'est-à-dire comme composants éventuellement mobiles ou comme conteneurs de composants.

La seconde contribution de cette thèse est de préciser la méthodologie de formalisation par raffinements successifs, pour construire à base de domaines des infrastructures réparties dignes de confiance: à partir d'un modèle de programmation à base de domaines, le M–calcul, nous en spécifions une implantation par une machine abstraite, puis réalisons l'implantation sous forme de machine virtuelle. Ce cheminement a permis de démontrer que la notion de domaine est effective, et peut être utilisée comme primitive de programmation. Autrement dit, un modèle de programmation à base de domaines est implantable en pratique. En reliant théorie et pratique de la programmation répartie à large échelle par un continuum d'étapes de raffinement, cette méthodologie fournit aussi un guide pour la constructions d'infrastructures à base de domaines.

Plan de la thèse

Ce mémoire de thèse est organisé de la manière suivante:

Nous commençons par examiner dans le chapitre 2 le double obstacle lié à la construction d'infrastructures fiables pour la programmation répartie sur des réseaux à large échelle: nous analysons les causes possibles de la divergence entre théorie et pratique de la programmation répartie et identifions les exigences à satisfaire pour un modèle de programmation réparti adapté au contexte du « global computing ». Nous définissons alors de manière précise la notion de domaine, et présentons la méthodologie utilisée dans l'ensemble de la thèse, de construction de systèmes à base de domaines par raffinements successifs. Nous terminons ce chapitre par un tour d'horizon des différentes déclinaisons de domaines dans l'état de l'art des modèles et infrastructures pour la programmation répartie et mobile.

Nous cherchons ensuite dans le chapitre 3 à déterminer le bien-fondé et la faisabilité de cette méthodologie à l'aide d'une expérimentation préliminaire. En partant d'un modèle de programmation appelé DCP comprenant une forme simple de domaine, nous détaillons les étapes de formalisation depuis la conception du modèle jusqu'à son implantation, formellement spécifiée à l'aide d'une machine abstraite, et intégrée avec un middleware représentatif de la technologie à objets répartis.

Nous appliquons ensuite les trois étapes de la démarche à un véritable modèle de programmation répartie, fondé explicitement sur la notion de domaine, le M–calcul (cf. chapitre 4): nous définissons une machine abstraite répartie pour ce calcul appelée CLAM (cf. chapitre 5) puis implantons cette spécification formelle sous forme d'une machine virtuelle appelée OYSTERS, dans un cadre centralisé puis distribué (cf. chapitre 6).

Enfin, dans le dernier chapitre, nous dressons un bilan sur l'intérêt des domaines comme primitive de programmation répartie, et celui de la méthodologie proposée pour construire des systèmes fiables à base de domaines. Nous évaluons les résultats obtenus et leurs limites, ainsi que les perspectives de recherche qu'ouvre cette étude.

Chapitre 2

Modèles et infrastructures pour la programmation répartie

2.1 La programmation répartie fiable dans les systèmes à large échelle

Depuis quelques années, on assiste simultanément à une croissance sans précédent de la taille des infrastructures réparties comme l'Internet et au développement d'outils et de paradigmes de programmation comme les agents mobiles ou la programmation par composants permettant la réalisation d'applications intrinsèquement réparties sur de telles infrastructures. Le réseau et ses ressources y occupent une place prépondérante, à tel point que ce contexte de programmation a pu être qualifié de « network-centric computing », « pervasive computing », ou encore « global computing » [123]:

> *The term "global computing" is used to describe a projected scenario in which processors will be everywhere, e.g. in household devices, in cars, in clothing etc, and in which networks will be heterogeneous and highly diverse in their capabilities, e.g. high-speed networks, ad-hoc wireless networks etc. They will be able to communicate and also sense and interact with their environment. The result is a massive networked infrastructure composed of highly diverse interconnected objects that should support the design and use of systems with a predictable and desirable behaviour.*

Simultanément, les besoins en fiabilité et en sécurité pour ces infrastructures logicielles sont devenus immenses avec la complexité grandissante des applications réparties. Malheureusement, le développement de méthodes et d'outils de programmation pour l'écriture d'applications fiables dans un environnement de réseaux à large échelle exhibant à la fois répartition et mobilité apparaît toujours comme un problème largement ouvert. C'est dans ce contexte très général que s'inscrit ce travail de thèse.

Depuis les années 70, un travail de recherche considérable a été effectué sur les méthodes formelles pour garantir la fiabilité et la sécurité des systèmes informatiques – généralement centralisés et de taille réduite. Certains des résultats obtenus ont depuis été appliqués avec succès pour concevoir, implémenter et valider des systèmes critiques dans des domaines comme l'avionique, le contrôle aérien ou les télécommunications [130].

Ces méthodes rendent plus sûre l'écriture de programmes en utilisant, pour chaque étape du cycle de développement logiciel, des techniques rigoureuses comme des modèles mathématiques minimaux permettant de raisonner sur le programme en cours de développement et d'en expliciter la sémantique de manière précise. Elles peuvent être utilisées lors des phases: de spécification, en rationalisant l'écriture d'un cahier des charges ou en clarifiant la description d'une architecture logicielle; de programmation, en utilisant des langages de programmation sûrs dont la sémantique limite les erreurs d'exécution; ou de validation, en spécifiant et prouvant formellement des propriétés vérifiées par le système. Des classes intéressantes de propriétés à vérifier comprennent les *propriétés de sûreté* comme la « subject reduction » qui exprime la conservation du type d'un terme au cours d'un calcul et garantit qu'un programme bien typé ne provoquera pas d'erreur à l'exécution ou des *propriétés de vivacité* qui concernent la progression de l'exécution d'un programme.

L'application de ces méthodes à des systèmes répartis ouverts à large échelle comme le réseau d'information d'un opérateur de télécommunications reste à présent largement inexplorée. L'enjeu est important car il s'agit de poser les fondements des futurs systèmes répartis fiables et sûrs. Leur comportement doit pouvoir être prévisible et analysable, par exemple pour présenter des garanties de qualité de service – vues au sens large comme incluant certaines propriétés non-fonctionnelles traditionnelles: sécurité, temps de réponse, tolérance aux défaillances, etc. Seules les techniques formelles pour la spécification, programmation et vérification apparaissent en mesure de répondre à ce niveau élevé de besoins car pouvant fournir des garanties de systèmes « corrects par construction ».

Cependant, l'application des techniques formelles actuelles à ces systèmes se heurte à une double difficulté:

- Un fossé grandissant s'être creusé entre, d'une part, les techniques formelles de modélisation des systèmes répartis mobiles, qui offrent une vue de ces systèmes à un niveau d'abstraction trop élevé, et qui ont fait l'objet de peu d'implémentations – et encore moins d'implémentations rigoureusement conformes au modèle de départ – et, d'autre part, les plates-formes réparties couramment utilisées, dont la sémantique du modèle de programmation sous-jacent est peu explicitée ou difficile à formaliser.

- Les modèles, langages, machines virtuelles et middleware actuels apparaissent insuffisants pour relever les défis posés par un environnement de type « global computing »: les primitives d'interaction offertes sont limitées et peu flexibles, la manifestation des différents composants du système est souvent réduite, et le support pour la mobilité, la sécurité ou la tolérance aux défaillances généralement inexistant.

L'objet de cette thèse est précisément de montrer que ce fossé entre deux visions opposées de la programmation de systèmes distribués à large échelle peut être en partie réduit, d'une part, en définissant un modèle formel de programmation suffisamment expressif pour rendre compte de la plupart des situations rencontrées dans l'univers du « global computing », et, d'autre part, en spécifiant formellement et en réalisant une implémentation raisonnablement efficace de ce modèle, qui soit implémentable à l'aide de briques logicielles de base couramment utilisées dans les plates-formes réparties actuelles.

2.1.1 Divergence entre théorie et pratique

Deux visions de la programmation répartie

La construction d'infrastructures réparties fiables reste un défi majeur. Cependant, en termes de solutions, un écart significatif s'est creusé entre deux visions pourtant complémentaires de la programmation répartie, écart qu'il apparaît urgent de réduire. La première approche est d'inspiration théorique et a produit des modèles formels sophistiqués capturant de plus en plus finement la sémantique d'exécution des systèmes concurrents et répartis. Ces modèles, comme par exemple les algèbres de processus, visent à en étudier rigoureusement plusieurs dimensions comme la communication, la mobilité, les défaillances ou la sécurité. Cependant, les résultats théoriques obtenus font assez rarement l'objet de traductions directes en terme d'implémentations: il s'agit souvent de maquettes élémentaires écrites dans des langages fonctionnels, mais dont les liens avec les technologies à objets répartis couramment utilisées dans l'industrie sont assez ténus. La seconde approche est orientée implémentations et s'appuie sur de nouveaux langages de programmation répartie ou sur des technologies middleware devenues des standards de fait ou ayant acquis un certain degré de maturité comme les plates-formes à objets répartis JAVA RMI [160], CORBA [133] ou EJB [161]. Malheureusement, le savoir-faire concernant ces plates-formes est principalement détenu par les programmeurs, ces infrastructures s'étant souvent développées sans beaucoup se préoccuper des fondements théoriques sous-jacents (sémantique inexistante ou peu explicitée) et sont par conséquent assez difficiles à formaliser.

Un bref historique

Ces deux visions de la programmation répartie correspondent à deux communautés de recherche qui semblent parfois avoir du mal à communiquer entre elles car leurs centres d'intérêts respectifs – les méthodes formelles et la théorie des langages de programmation pour la première, les plates-formes d'exécution réparties et le middleware pour la seconde – sont étudiés séparément sans véritable synergie alors qu'ils sont pourtant complémentaires.

Ce décalage en soi n'est pas nouveau. Depuis les débuts de l'informatique, dans les années 60 et 70, de nombreux points de désaccord ont souvent opposé théoriciens et développeurs, à propos d'un langage de programmation, d'une méthode de développement en génie logiciel ou d'un système d'exploitation. Dans les pires cas, de telles divergences prirent l'allure de véritables guerres de religion, que Robert Glass qualifie dans [69] de « proof of correctness wars »:

> *The truth, as the case may have been, was either the possession of a particular branch of the academic world, or a particular collection of powerful practitioners, or a leading vendor, or someone else who tolerated no opposition. Woe be unto those who challenged such truth.*

Ainsi, quelques querelles de ce type opposèrent dans les années 70, les partisans de l'application des méthodes mathématiques au développement de l'industrie naissante du logiciel – avec notamment l'introduction de la notion de preuve de correction ouvrant la voie aux techniques formelles de vérification de programmes – aux ingénieurs et chercheurs en génie logiciel de l'époque qui l'ignorèrent, la considérant comme sans intérêt et sans avenir pour l'industrie du logiciel, allant même jusqu'à publier en 1977 un article sur ce sujet [52] dans une des conférences majeures sur les langages de programmation (POPL). Ce qui ne manqua pas de déclencher une querelle virulente, comme le montre, une réponse de Dikjstra (*ACM Software Engineering Notes*, avril 1978), qualifiant l'article de « political pamphlet from the Middle Ages ». Dans les années 80, une querelle similaire eut lieu, suite à l'article [56] déclarant son opposition farouche à l'utilisation des méthodes formelles à des fins de vérification, déclenchant également des protestations importantes de la partie adverse (*Communications of the ACM*, mars-avril 1989).

Actuellement, de telles querelles ouvertes se sont atténuées et des progrès notables ont été réalisés : par exemple, les techniques d'analyse statique sont appliquées à la sécurisation du code mobile (*proof-carrying code*) [132] ou à la construction de compilateurs et de machines virtuelles sûres [74], largement utilisés en pratique dans l'industrie. On peut citer aussi des premières tentatives de systèmes d'exploitation sécurisés, prouvés corrects par construction et certifiés au plus hauts niveaux de l'échelle des Critères Communs [43]. Cependant, une frontière invisible entre ces deux approches semble toujours être présente, les méthodes formelles continuant à être appelées de leurs vœux par la communauté universitaire, tandis qu'elles sont largement ignorées en pratique par l'industrie.

Quelques tentatives d'explication

Quelles sont les véritables causes de cette divergence profonde? S'agit-il, comme le souligne Robin Milner en conclusion de [128], d'un décalage temporel entre modélisation et technologie dont les rythmes de progression sont différents? « *Our concepts and models follow pioneering technology, not the reverse. But they will be essential to underpin a mature engineering practice.* »

Une autre raison possible concerne la difficulté de transposer les résultats formels obtenus dans ces modèles théoriques au niveau d'une implémentation. Ainsi, un formalisme élégant et extensivement étudié comme le calcul des ambients [40] modélise une exécution répartie à l'aide d'un nombre minimal de primitives de migration, mais se révèle en pratique très difficile à implémenter dans un contexte distribué [61, 149], car nécessitant la mise en oeuvre atomique d'algorithmes de synchronisation répartis. Un argument similaire concerne une forte tendance des modèles et techniques formelles à s'être orientés vers la spécification plutôt que vers la programmation, comme le montre le nombre important de méthodes formelles de spécification de systèmes concurrents existants (Z, B, LOTOS, ESTELLE, réseaux de Pétri, etc.), mais la gamme relativement réduite de langages de programmation dérivés de ces formalismes, ce qui éloigne encore ces modèles de l'aspect implémentation. Enfin, ces modèles formels se sont surtout concentrés sur des aspects particuliers de l'exécution concurrente et répartie comme la mobilité ou la sécurité, mais ne proposent pas de concepts ou de primitives uniformes, comme en témoigne le vaste ensemble de systèmes de types proposés pour contrôler la mobilité ou l'accès aux ressources [76, 150]. Cette hétérogénéité rend la comparaison difficile quand il s'agit de choisir un modèle à implanter parmi de nombreux formalismes.

Au niveau pratique, de nombreux langages de programmation répartie sont apparus ces dernières années. Mais leurs fondements théoriques ont encore été peu étudiés, contrairement aux paradigmes de programmation séquentielle, fonctionnelle ou orientée-objets qui ont fait l'objet de travaux extensifs (cf. par exemple [13] et [1]). Ces langages demeurent difficiles à formaliser car les primitives de communication ou de synchronisation introduites sont multiples, difficiles à comparer car souvent directement inspirées de fonctions système, ce qui rend l'abstraction ardue a posteriori. De même, au niveau des plates-formes (à objets répartis, à composants, à agents mobiles, etc.), on assiste à une diversité de spécifications, dont les modèles de programmation implicites sont divergents, avec des sémantiques formelles souvent inexistantes, et où, en terme de modélisation, il paraît difficile d'aller au delà d'un cadre très général de programmation répartie comme le modèle de référence ODP [23, 81, 82, 83]. Comme le remarque Cédric Fournet, « *overall, distributed programming is still more an art than a science* » [58].

Une raison importante de cette divergence entre modélisation et technologie pourrait provenir d'un dialogue encore insuffisant entre les deux communautés de recherche qui y sont associées: même si l'on trouve de plus en plus de travaux décrivant une approche d'utilisation des méthodes formelles dans un cadre pratique et/ou industriel, trop peu d'entre eux cherchent à en évaluer les bénéfices.

Quelques pistes pour réduire cette divergence

De la discussion précédente, il ressort que, pour tenter de réduire cette divergence, trois exigences apparaissent comme fondamentales pour que ces modèles puissent être largement adoptés par les fournisseurs de technologie.

Notation: dans toute la suite, les exigences d'un modèle de programmation ou les observables d'un réseau à large échelle seront respectivement notées `REQ.*` et `OBS.*`.

---

a) Exigence `REQ.EFFECTIVENESS` (Effectivité).

⇒ *Un modèle de programmation répartie doit être implémentable.*

Justification Pour être véritablement utile, un tel modèle doit pouvoir être vu comme le cœur d'un langage de programmation donnant lieu à une implémentation répartie.

---

b) Exigence `REQ.NO_HIDDEN_COSTS` (Transparence des coûts d'implantation).

⇒ *Un modèle de programmation répartie doit fidèlement refléter les primitives offertes par les infrastructures réparties existantes. Elle doivent se placer au bon niveau d'abstraction sans masquer leur coût réel d'implantation.*

Justification La programmation fine de systèmes répartis requiert un accès aux ressources de bas niveau. Aussi, un programmeur ne doit pas disposer d'une vue masquée des coûts d'implantation des primitives du langage de programmation qu'il utilise, en particulier la présence possible de défaillances. Le modèle doit ainsi refléter fidèlement les primitives de base de communication et de mobilité dont sont pourvues les infrastructures réparties courantes, tout en se plaçant au bon niveau d'abstraction. Par exemple, une étape de calcul atomique du modèle de programmation ne devrait pas masquer un algorithme complexe de synchronisation répartie se traduisant par l'envoi de multiples messages sur le réseau.

---

c) Exigence `REQ.DESIGN_TO_IMPLEMENTATION` (Raffinement).

⇒ *Une continuité formelle de bout en bout est nécessaire depuis le modèle jusqu'à son implantation.*

Justification Pour tenter de rapprocher théoriciens et programmeurs, une étude de bout en bout apparaît nécessaire pour examiner et évaluer les différentes étapes de raffinement dans la formalisation depuis la conception d'un modèle formel de programmation répartie clairement défini, jusqu'à son implémentation sur une infrastructure d'exécution répartie représentative de l'état de l'art en matière de construction de plates-formes distribuées, en passant par une phase de spécification rigoureuse. Il s'agit là du premier axe majeur de cette thèse.

2.1.2 Abstractions pour la programmation répartie mobile à large échelle

La divergence identifiée précédemment entre théorie et pratique de la programmation répartie n'est pas le seul obstacle à surmonter pour construire des infrastructures réparties fiables sur des réseaux à large échelle. Une seconde difficulté est constituée par le problème du passage à l'échelle auquel modèles et infrastructures actuels répondent de manière très imparfaite. Il se manifeste par un écart entre la programmation sur un réseau local, étudiée de manière extensive, et celle sur un réseau à large échelle où les points de repère connus ne sont plus valables, ce qui est renforcé en présence de mobilité dans le système. Dans ce qui suit, nous montrons que cette transition vers les systèmes répartis mobiles à large échelle peut être décrite par trois images fortes identifiées par Luca Cardelli [37, 38, 39] et détaillées dans les trois paragraphes suivants. Ces images mettent en évidence que l'univers du « global computing » se manifeste par un certain nombre d'observables que les modèles de programmation connus ne prennent pas en compte. Il devient alors nécessaire d'élaborer de nouveaux modèles de programmation répartie mobile pour décrire ces comportements. Une abstraction fondamentale, celle de *domaine*, est alors identifiée comme rendant compte d'une grande partie de ces observables. Après l'avoir définie de manière précise, nous illustrons son pouvoir expressif sur quelques exemples.

Cette notion de domaine constitue le second axe fort de la thèse, permettant de modéliser de manière fine et extensive la programmation répartie à large échelle.

Image 1: programmer sur un réseau local

La première image concerne la programmation sur un réseau local: ces réseaux répondent à un besoin de partage de ressources en connectant plusieurs machines autonomes. Un réseau local se présente sous la forme d'une collection de machines de puissances comparables connectées par des liens réseau de bande passante et de temps de réponse sensiblement équivalents (cf. figure 2.1a). Sa caractéristique essentielle est son aspect uniforme en terme de puissance de calcul, de bande passante et de latence réseau. De cette uniformité découle un caractère prévisible : les délais de propagation sont bornés, et le temps d'un calcul donné peut être estimé.

Un tel réseau est facile à administrer de manière centralisée. Il est de plus bien protégé car entouré de pare-feux repoussant les attaques venant de l'extérieur. Il constitue un *domaine d'administration*, brique de base de l'architecture des réseaux à large échelle.

La topologie d'un réseau local est statique: les machines physiques sont immobiles et résident à une adresse IP fixe. En terme de programmation, la complexité de cette topologie est cachée explicitement à l'aide de techniques classiques comme l'appel à distance de procédure ou de méthode (Remote Procedure

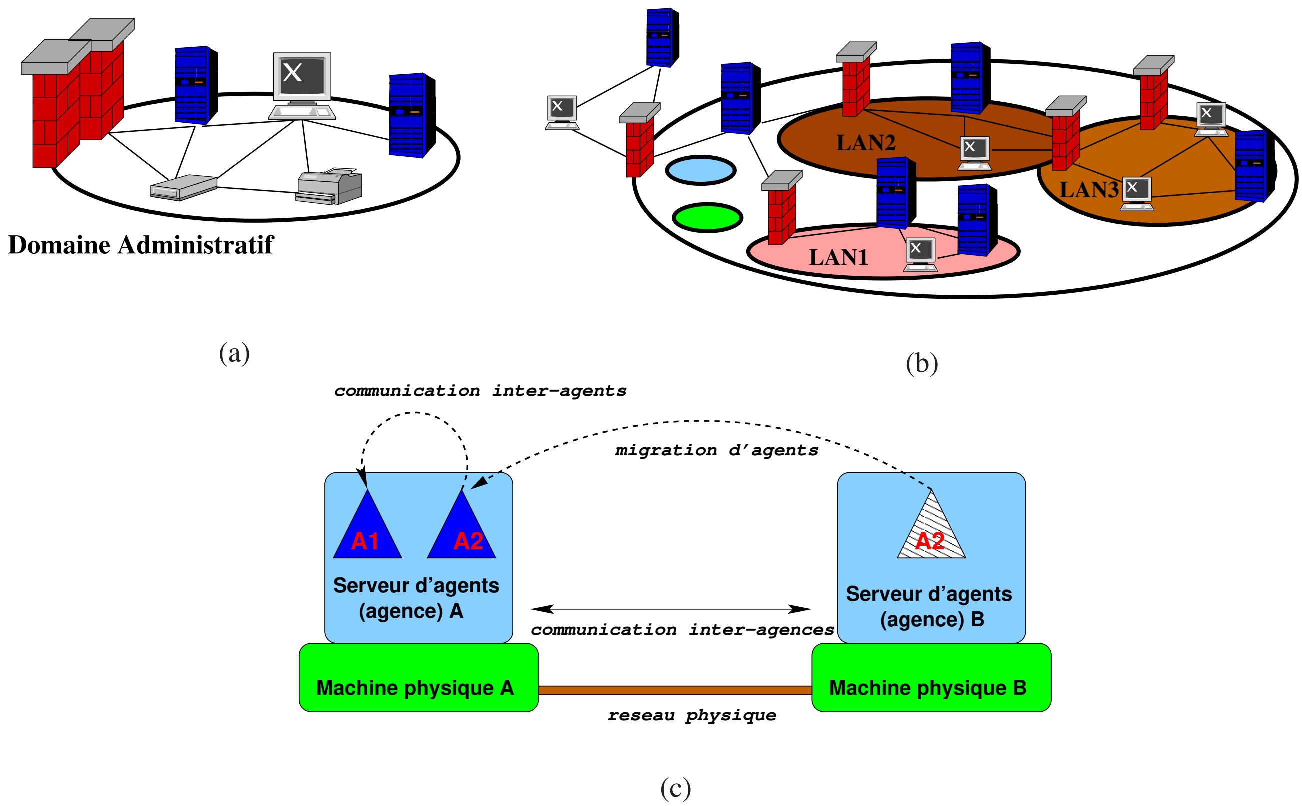


FIG. *2.1* – **Trois images de la programmation répartie mobile:** *(a) un réseau local; (b) un réseau à large échelle; (c) exécution d'agents mobiles*

Call, Remote Method Invocation), ou les courtiers d'objets (Object Request Brokers) qui font abstraction de la structure du réseau pour faciliter la tâche du programmeur. Ce dernier voit alors toutes les ressources distantes comme étant situées à seulement une étape logique de communication.

Même dans ce cadre simple, diverses notions de mobilité logicielle ont fait leur apparition. On peut citer la mobilité pure de données (sérialisation) entre deux sites comme le transfert de connexions réseau [20], l'exécution de code à distance, puis la migration de code sous différentes formes [46, 64]: migration de fermetures [35], objets ou agents mobiles dont le comportement est résumé par le manifeste TELESCRIPT [174], etc.

Image 2: les réseaux à large échelle

Les caractéristiques principales d'un réseau local en font un environnement de programmation bien identifiable: caractère uniforme rendant possible une gestion transparente des ressources réseau, administration centralisée, environnement protégé, et comportement prévisible donc offrant certaines garanties de fiabilité.

Ces propriétés ne sont plus vérifiées sur un réseau à large échelle comme le Web : on entre alors dans l'univers du « global computing ». Ainsi, si une gestion transparente des ressources réseau est encore possible sur un réseau local, une telle approximation n'est plus valable dans un système réparti à large échelle où la notion de localité se manifeste de manière aiguë. Quelles sont les causes de ce changement?

La raison principale ne réside pas dans la différence d'échelle entre les deux types de système. Elle tient au fait qu'un réseau à large échelle ne peut pas être sujet à un contrôle centralisé. Comme le montre la figure 2.1b, un tel réseau prend la forme d'une superposition complètement décentralisée de nombreux domaines administratifs indépendants, séparés par de fortes barrières de confiance pour des questions de sécurité, chaque domaine étant géré de manière autonome.

Le réseau perd l'aspect uniforme d'un réseau local : les machines ont des puissances et des résistances aux défaillances qui varient fortement, et les liens réseau sont de capacités et de fiabilité très diverses. La transparence réseau devient alors impossible à garantir, car les temps de propagation sur de longues distances physiques deviennent observables, et la topologie réseau devient trop complexe pour être masquée au programmeur d'applications.

De même, le réseau devient un environnement extrêmement dynamique dont la topologie n'est plus figée, certains sous-réseaux devenant temporairement inaccessibles à cause de défaillances de machines ou de liens de communication, et accessibles de nouveau lors de l'éventuelle reprise de la machine ou du lien défaillant. Ces pannes fréquentes contribuent à la perte de l'aspect prévisible du comportement du réseau: l'environnement est complètement asynchrone et les temps de communication deviennent non bornés. Dans cet environnement instable et peu fiable, il devient difficile de garantir des propriétés de qualité de service. Cette dynamicité est accentuée en présence de mobilité, certaines machines ou sous-réseaux pouvant franchir les frontières de domaines administratifs, ce qui se traduit en terme de topologie par des liens réseau de durée de vie limitée et par des noeuds de communication qui disparaissent puis réapparaissent à un autre endroit du réseau.

Image 3: la programmation répartie mobile

On aboutit alors à un environnement de programmation où deux types de mobilité se manifestent:

- La *mobilité logicielle*, également appelée *mobilité virtuelle* et qualifiée dans [39] de « mobile computation » : du code est déplacé (mobilité dite *objective*) ou se déplace de lui-même (mobilité dite *subjective*) d'un endroit à l'autre du réseau sous forme d'unités d'exécution. Le paradigme de programmation par *agents mobiles* [163, 173] (cf. figure 2.1c) constitue la meilleure illustration de ce type de mobilité.
- La *mobilité matérielle*, également appelée *mobilité physique* et qualifiée dans [39] de « mobile computing »: les composants matériels même du réseau deviennent mobiles, les machines physiques pouvant se déplacer dynamiquement d'un réseau local à l'autre. On peut citer ici des sous-réseaux eux-mêmes mobiles comme les Personal Area Networks d'IBM, et le monde des objets communiquants, capables de se constituer spontanément en réseaux locaux sans-fil (technologie Bluetooth), et d'accepter dynamiquement de nouveaux membres du réseau, tout en étant déplaçables physiquement (vêtements communicants).

Ces deux types de mobilité peuvent interagir et se manifestent par une *connectivité intermittente*, provoquée soit par un mouvement physique (téléphone portable temporairement déconnecté du réseau car situé dans un endroit non couvert) soit par un mouvement logique (agent logiciel mobile migrant dans un domaine administratif n'autorisant pas les communications vers l'extérieur pour des raisons de sécurité).

Dans toute sa généralité, la programmation répartie mobile se manifeste donc par de multiples frontières, physiques ou virtuelles, elles-mêmes éventuellement mobiles, généralement organisées en une hiérarchie mais capables d'interagir de manière complexe à cause de l'existence de multiples chemins d'accès aux ressources. Chacune de ces frontières requiert des autorisations pour son franchissement. Une fois la frontière franchie, les entités mobiles sont relativement libres de se déplacer à l'intérieur de ses limites. Ces frontières ne peuvent pas être masquées et constituent l'un des traits distinctifs fondamentaux du « global computing ».

Les observables du « global computing »

Les trois images précédentes illustrent que dans une infrastructure répartie à large échelle comme le Web, de nombreuses hypothèses familières de programmation ne sont plus valides. Un certain nombre de phénomènes observables ne peuvent plus être ignorés dans un environnement de type « global computing », alors qu'ils restaient cachés dans un contexte de réseau local. Pour simplifier, on peut regrouper ces phénomènes en quatre « observables » fondamentales [37, 39] que les modèles de programmation connus ne prennent pas en compte, auxquelles il faut ajouter une observable transverse, la mobilité (physique et logique), qui interagit avec les observables précédentes:

---

a) Observable `OBS.LOCALITY.V` (Localités virtuelles).

⇒ *Un environnement de type « global computing » manifeste diverses notions de localités virtuelles.*

Justification On peut voir un réseau à large échelle comme un univers structuré par des barrières qui délimitent un certain nombre de domaines administratifs ou *localités virtuelles* autonomes et souveraines dans leurs actions. Cependant, ces localités doivent pourvoir interagir au moyen de primitives de communication et de mobilité, ce qui rend nécessaire le franchissement de ces barrières. Malheureusement, la transparence réseau au niveau de la communication et de la migration est perdue. Elle est remplacée par un certain nombre d'étapes atomiques de franchissement de barrières. Le problème clé lors du choix des primitives de programmation sur un réseau à large échelle est alors de savoir sous quelles conditions autoriser ces franchissements sans mettre en danger la sécurité. Ces localités virtuelles induisent une notion de *distance virtuelle* entre localités, et une nécessité de prise de conscience pour un programme de la localité dans laquelle il s'exécute (« network-awareness »). Par rapport à cette observable, la mobilité consiste à franchir les frontières des localités virtuelles.

---

b) Observable `OBS.LOCALITY.P` (Localités physiques).

⇒ *Un environnement de type « global computing » manifeste une notion de localité physique.*

Justification La latence réseau est bornée inférieurement par le temps de propagation nécessaire à la lumière pour relier les deux extrémités de la communication. Sur un réseau à large échelle, cette borne inférieure devient observable et manifeste une notion de proximité et donc de localité physique dans l'exécution d'un programme. Par exemple, l'accès à une base de données répliquée se fera en fonction de la localisation physique de l'utilisateur qui choisira le serveur-miroir le plus proche pour bénéficier du meilleur temps d'accès. Comme pour les localités virtuelles, le code mobile peut également se déplacer d'une localité physique à une autre.

---

c) Observable `OBS.BANDWIDTH` (Fluctuations de bande passante).

⇒ Un environnement de type « global computing » manifeste des fluctuations de bande passante au niveau du réseau.

Justification Ces fluctuations, dues par exemple à la congestion du réseau, deviennent observables et peuvent se manifester par une connectivité intermittente, déjà évoquée lors la présentation de la mobilité. Ces fluctuations peuvent aller jusqu'à créer une partition du réseau en régions dans lesquelles la bande passante et le temps de réponse sont comparables, créant ainsi de nouvelles localités virtuelles. Les programmes mobiles peuvent être conçus pour réagir à ces fluctuations en se déplaçant vers une localité plus adaptée.

---

d) Observable `OBS.FAILURES` (Défaillances).

⇒ Un environnement de type « global computing » manifeste la présence de défaillances multiples.

Justification Dans un contexte réparti à large échelle, les pannes sont provoquées par la défaillance de certaines machines ou liens réseau et ne deviennent pas distinguables de latences réseau importantes : un réseau à large échelle est avant tout un système asynchrone où les temps d'exécution et de communication ne sont pas bornés supérieurement. Ces pannes sont fréquentes et imprévisibles. En provoquant des partitions du réseau – nouvelles formes de localités virtuelles – elles peuvent aussi se manifester par une connectivité intermittente. Ces pannes ont des effets observables et devraient pouvoir être détectées [42] pour mettre en oeuvre les parades appropriées. Par exemple, un agent mobile devrait pouvoir calculer son itinéraire pour contourner en migrant les diverses zones de défaillances.

Ces quatre observables, en présence de mobilité, mesurent les différences essentielles qui séparent un réseau à large échelle d'un réseau local et ne sont pas décrites par les modèles de programmation actuels (cf. section 2.3). La modélisation formelle d'un tel système réparti devrait donc prendre en compte ce nouvel ensemble d'observables pour déterminer les abstractions en capturant la sémantique d'exécution, nécessaires pour construire un cœur de langage de programmation adapté au contexte du « global computing ».

Caractéristiques fondamentales d'un modèle du « global computing »

Les paradigmes de programmation actuels ne rendent pas compte de manière satisfaisante des observables précédentes. Il est donc nécessaire de définir de nouveaux modèles de programmation pour le « global computing ». Sans prétendre à l'exhaustivité, nous tentons ici d'identifier un certain nombre de caractéristiques fondamentales que doit posséder un tel modèle. Pour une analyse plus approfondie, on pourra par exemple se reporter à [158].

Le modèle doit tout d'abord satisfaire à un certain nombre d'exigences générales:

- *Simplicité* (Exigence `REQ.SIMPLICITY`): le modèle doit rester simple pour que les techniques de preuves formelles associées restent tractables et puissent être utilisées pour raisonner sur les programmes.
- *Effectivité* (Exigence `REQ.EFFECTIVENESS`): déjà identifiée dans la section 2.1.1, cette exigence signifie que le modèle doit pouvoir donner lieu à une implémentation répartie efficace pour être utilisable comme cœur de langage de programmation.
- *Complétude* (Exigence `REQ.COMPLETENESS`): décrite dans [38] comme « WAN-completeness », cette exigence signifie que le modèle doit être suffisamment expressif pour que ses primitives permettent de décrire de manière complète les caractéristiques essentielles des systèmes répartis à large échelle, par exemple pour rendre compte du comportement d'applications exhibant répartition et mobilité sur le Web.
- *Ouverture* (Exigence `REQ.OPENNESS`): on observe une telle diversité sur des réseaux à large échelle – par exemple de primitives de communication (envoi de messages, espaces de tuples partagés...) ou de modes de défaillances (fail-stop, comportements byzantins...) – que pour la capter, le modèle doit être susceptible d'inclure de nouvelles sémantiques de communication, ou de nouveaux modes de défaillances.

Le modèle doit en outre satisfaire un certain nombre d'exigences techniques, séparables en deux catégories: quatre exigences fondamentales (asynchronisme, activité, composition et localité) décrivent l'exécution concurrente et répartie de processus; les autres exigences traduisent différents aspects de cette exécution dans un contexte hautement dynamique de répartition à large échelle. Ces exigences sont résumées dans la figure 2.2.

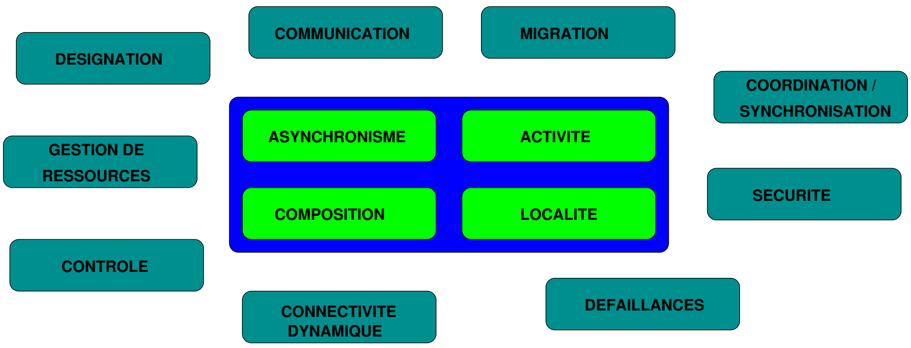


FIG. *2.2* – **Exigences d'un modèle de programmation pour le « global computing ».**

---

a1) Exigence `REQ.ASYNCHRONY` (Asynchronisme).

$\Rightarrow$ *Un modèle de programmation pour le « global computing » doit comporter un modèle d'exécution et de communication asynchrone.*

Justification Un environnement réparti à large échelle se caractérise par une large diversité en terme de fréquences de calcul des processeurs et de latences des liens réseau. Un modèle de programmation ne peut donc que s'appuyer sur un modèle d'exécution et de communication asynchrone au sens où il n'est pas possible d'observer les vitesses relatives des unités d'exécution ou de les synchroniser sans se référer à une horloge globale. Il devient ainsi impossible de se fier à un temps de réponse global du système. Cette exigence rend compte en partie des observables `OBS.BANDWIDTH` et `OBS.FAILURES`: les temps de réponse observés étant très fluctuants, une latence réseau élevée n'est pas dans ce contexte distinguable de la défaillance d'une machine.

---

a2) Exigence `REQ.PROCESS` (Activité).

⇒ *Un modèle de programmation pour le « global computing » doit fournir une notion d'activité permettant de représenter et de manipuler les unités d'exécution autonomes du système.*

Justification Un système à large échelle peut être vu comme un ensemble de constituants autonomes, c'est-à-dire disposant chacun d'une activité propre, ces composants étant manifestes à l'exécution. Au niveau de la modélisation, il convient donc de représenter ces unités d'exécution sous forme d'abstractions traduisant la notion d'activité, comme les processus ou les threads, et de primitives permettant de les manipuler. De plus, ces éléments ne devraient pouvoir interagir avec leur environnement qu'au moyen d'interfaces bien définies (*encapsulation*), ne pas révéler plus de détails que nécessaire sur leur structure interne (*abstraction*) et enfin disposer d'une identité les désignant sans ambiguïté et en permettant l'accès et la manipulation [33].

---

a3) Exigence `REQ.COMPOSITION` (Composition).

⇒ *Un modèle de programmation pour le « global computing » doit fournir une notion de composition de sous-systèmes.*

Justification Un système réparti à large échelle est un assemblage structuré de différents sous-systèmes ou *composants*: il est donc indispensable de disposer dans le modèle d'une primitive rendant cette superposition explicite. Généralement, dans les modèles examinés dans la section 2.3, il s'agira d'un opérateur de composition parallèle, commutatif, associatif, ayant pour élément neutre le processus inactif. La sémantique de cet opérateur est une exécution concurrente et asynchrone (cf. exigence `REQ.ASYNCHRONY`) d'unités de calcul (cf. exigence `REQ.PROCESS`) pouvant interagir à l'aide de primitives de communication définies par chaque modèle. Cet opérateur peut par exemple représenter les frontières entre machines connectées par le réseau. La structuration en sous-systèmes étant en général récursive, l'opérateur de composition doit pouvoir intervenir à différents niveaux de granularité pour traduire l'assemblage de composants en un composant de niveau supérieur. De plus, le caractère hautement dynamique du « global computing » implique la définition dans le modèle d'une relation d'imbrication pouvant être définie et modifiée à l'exécution.

Les réseaux à large échelle manifestent différentes notions de composition: par exemple l'assemblage d'un système à partir de composants existants, ou la possibilité de reconfigurer certains composants, ou d'en introduire de nouveaux en cours d'exécution. Le modèle de programmation ne doit donc pas imposer une forme particulière de composition mais être suffisamment expressif pour supporter des sémantiques de composition variées (cf. exigence `REQ.CONTROL`).

Les exigences a1, a2, et a3 correspondent au cadre de la programmation concurrente formalisée par de nombreuses algèbres de processus (cf. section 2.3.2): plusieurs unités d'exécution appelées *processus* sont placées en parallèle et peuvent interagir à l'aide de primitives de communication variées. Ces modèles constituent un bon point de départ pour capturer les fondements théoriques du « global computing », mais il est nécessaire d'aller au delà, en particulier pour refléter les notions fondamentales de *barrière* et de *localité*.

Les barrières, omniprésentes et multiformes constituent un élément structurant primordial d'un réseau à large échelle en le partitionnant en sous-systèmes. Les difficultés associées à leur franchissement doivent se retrouver dans un modèle formel du « global computing ». Une vision duale consiste à partir des localités qui sont délimitées par ces barrières.

---

a4) Exigence `REQ.LOCALITY` (Localité).

⇒ *Un modèle de programmation pour le « global computing » doit être construit autour d'une abstraction de localité, fournie de manière primitive et capturant le partitionnement spatial et logique d'un système réparti à large échelle.*

Justification La notion de localité capture le partitionnement d'un système à large échelle en différentes régions administratives, pouvant éventuellement se recouvrir. Chaque localité possède des propriétés distinctives et traite d'un aspect particulier de l'exécution répartie comme la mobilité, la tolérance aux défaillances, la sécurité ou la gestion de ressources. Ces localités sont délimitées par des barrières qui en déterminent l'étendue et permettent de préserver ces propriétés. Le terme de localité doit être également pris au sens de *propriété de localité* traduisant le fait que l'on se trouve ou non dans la même localité, selon qu'une barrière aura été franchie. Ces localités sont physiques (cf. observable `OBS.LOCALITY.P`) ou logiques (cf. observable `OBS.LOCALITY.V`). En particulier, une définition possible de la répartition consiste à la faire coïncider avec la coexistence de différentes localités physiques dans un système. L'existence de localités a un impact direct sur les primitives de communication et de migration: une transparence complète pour l'action à distance est impossible à garantir car le déplacement d'une localité à l'autre nécessite le franchissement de barrières, ce qui requiert un modèle de mobilité. La notion de *composant*, déjà utilisée pour désigner des sous-systèmes, apparaît dans le contexte du génie logiciel comme un cas particulier de localité, et peut se décliner par exemple comme unités de configuration, de distribution, de mobilité, etc.

En particulier, un besoin important pour la modélisation d'un environnement de type « global computing » consiste à disposer de différentes formes de localités que l'on puisse programmer explicitement pour rendre compte des sémantiques variées de composition (cf. exigence `REQ.COMPOSITION`), des formes d'interaction et de synchronisation multiples observées dans ce contexte. Au niveau des localités, des primitives de contrôle plus riches qu'une simple expression d'un partitionnement logique sont donc nécessaires (cf. exigence `REQ.CONTROL`).

b1) Exigence `REQ.SECURITY` (Sécurité).

⇒ *Un modèle de programmation pour le « global computing » doit fournir des primitives pour assurer la sécurité des localités.*

Justification Dans un réseau à large échelle, les différentes localités virtuelles ou physiques ne se font pas confiance. Un modèle de sécurité est nécessaire pour protéger ces localités des attaques. Les abstractions correspondantes doivent être reflétées au niveau d'un langage de programmation pour construire des applications dignes de confiance. Outre la notion de barrière qui constitue une abstraction forte de sécurité, duale de celle de localité, des opérations cryptographiques peuvent être introduites dans le modèle. Mais surtout, des primitives de contrôle du franchissement des barrières comme des capacités sont nécessaires. Ces primitives peuvent être utilisées pour restreindre – par exemple à l'aide de systèmes de types – la migration des localités, et limiter ainsi les possibilités de reconfiguration de la topologie du système ou pour sécuriser l'accès aux ressources (cf. exigence `REQ.RESOURCE`).

b2) Exigence `REQ.COMMUNICATION` (Communication).

⇒ *Un modèle de programmation pour le « global computing » doit fournir diverses abstractions de communication permettant la diffusion d'information entre localités. Cependant, il ne doit pas offrir de manière primitive de mode de communication nécessitant une connaissance globale du système, comme l'interaction à distance.*

Justification La communication est partitionnée par la présence de barrières et implique leur franchissement. On peut ainsi envisager les localités comme des unités de communication: si la communication purement locale, i.e., se produisant au sein d'une même localité, reste souhaitable, en revanche, des modes d'interaction distants ne doivent pas faire partie de manière primitive d'un modèle de programmation pour le « global computing »: cela nécessiterait une connaissance globale du système pour connaître à l'avance le récepteur distant ou pour déterminer parmi plusieurs récepteurs éligibles celui auquel revient la communication. Cette situation impliquerait la mise en oeuvre d'algorithmes de synchronisation répartis coûteux [118] (cf. exigence `REQ.COORDINATION`), voire impossible à implémenter en environnement asynchrone et en présence de défaillances [57]. La communication distante doit plutôt prendre la forme d'une série d'étapes atomiques non masquables, impliquant chacune le franchissement d'une seule barrière. Elle est alors codable de manière programmatique comme combinaison de primitives de migration et de communication locale. On perd ainsi la transparence globale des communications qui simplifiait la programmation de ressources sur un réseau local. En terme de modes de communication, le modèle devrait au moins inclure des primitives par envoi de messages qui semblent être le moyen le plus courant de communication dans les réseaux à large échelle. Cependant, la diversité des primitives manifestées dans ces environnements, implique la nécessité pour le modèle de pouvoir programmer différentes sémantiques d'interaction, par exemple en utilisant de la mémoire partagée [50] (cf. exigence `REQ.CONTROL`).

b3) Exigence `REQ.MIGRATION` (Migration).

⇒ *Un modèle de programmation pour le « global computing » doit fournir des primitives permettant d'exprimer la migration de composants logiciels et matériels.*

Justification Comme la communication, la mobilité d'entités logicielles et matérielles entre différentes localités implique le franchissement de barrières et constitue un besoin fondamental pour un modèle de programmation adapté au « global computing ». De même, une transparence à la mobilité ne peut plus être garantie, et la migration s'effectue explicitement étape par étape. A priori, une grande diversité de primitives d'expression de la migration est envisageable, les entités migrantes pouvant être des unités d'exécution, ou un ensemble de localités. Cette migration peut être déclenchée spontanément par l'entité migrante elle-même ou émaner de son environnement extérieur, par exemple en réaction à une interaction. Le modèle doit décrire la dynamicité d'un réseau à large échelle en reflétant les modifications de sa topologie dues à la migration de ses composants. D'autres aspects de cette dynamicité sont pris en compte au niveau du contrôle des localités, en particulier leur reconfiguration (cf. exigence `REQ.CONTROL`).

b4) Exigence `REQ.COORDINATION` (Coordination, synchronisation).

⇒ *Un modèle de programmation pour le « global computing » doit fournir des primitives exprimant la synchronisation et la coordination répartie de composants d'un réseau à large échelle.*

Justification Les mécanismes de synchronisation et plus généralement de coordination assurent l'orchestration de services dans des environnements répartis à large échelle. Cependant, pour être implantables dans un contexte réparti (cf. exigence `REQ.EFFECTIVENESS`), ces mécanismes doivent obéir à une contrainte de localité (cf. exigence `REQ.LOCALITY`): il s'agit avant tout d'éviter de masquer toute forme de synchronisation répartie entre composants situés sur des machines différentes, comme le consensus, coûteux à implémenter [31, 118], ou impossible à réaliser en présence de défaillances de manière non-probabiliste [57]. En général, les mécanismes de synchronisation sont intégrés aux primitives de communication. L'asynchronisme (cf. exigence `REQ.ASYNCHRONY`) a également un impact direct sur ces primitives: si la communication locale peut être synchrone, la communication distante doit être asynchrone pour éviter toute forme de rendez-vous atomique entre deux composants, non implémentable de manière efficace lorsque les participants sont séparés par une distance physique importante. La migration peut également nécessiter une synchronisation entre l'entité migrante, l'environnement de départ et celui d'arrivée [40, 61]. Comme pour la communication, la mise en place de mécanismes de coordination de plus haut niveau que de simples envois de messages, nécessite d'avoir à disposition dans le modèle de programmation des primitives permettant un contrôle fin du comportement des localités (cf. exigence `REQ.CONTROL`).

b5) Exigence `REQ.FAILURES` (Défaillances).

⇒ *Un modèle de programmation pour le « global computing » doit contenir des primitives pour gérer les défaillances et ainsi permettre la construction de systèmes robustes, répartis à large échelle.*

Justification Dans un environnement de type « global computing », les pannes surviennent suivant divers modes de défaillance : fail-stop, défaillances byzantines, etc. Un modèle de programmation doit fournir des abstractions pour gérer les pannes, afin de pouvoir construire des systèmes résistants aux défaillances. Leurs effets comme l'occurrence de partitions du réseau ne peuvent être masqués (cf. observables `OBS.FAILURES` et `OBS.BANDWIDTH`). Les zones où surviennent ces défaillances se comportent de manière indépendante et fournissent une autre manifestation de la notion de localité. L'environnement asynchrone (cf. exigence `REQ.ASYNCHRONY`) rend ces défaillances non distinguables de latences réseau élevées, et empêche l'implémentation d'algorithmes de synchronisation répartis comme le consensus [57] (cf. exigence `REQ.COORDINATION`). Une première possibilité consiste à inclure dans le modèle de programmation des primitives pour permettre leur détection [42]. D'autres pistes [8, 60] conduisent à introduire des notions de localités spécifiquement orientées défaillances pour modéliser des zones de confinement de fautes.

---

b6) Exigence `REQ.NAMING` (Désignation).

⇒ *Un modèle de programmation pour le « global computing » doit fournir des primitives pour la désignation des ressources d'exécution et la manipulation des noms associés.*

Justification Dans un environnement réparti à large échelle, la gestion des noms revêt une importance particulière: les noms permettent de désigner les ressources, en particulier lorsqu'elles sont situées de l'autre côté d'une barrière. Pour obtenir des stratégies de communication et de mobilité plus flexibles, il paraît fortement souhaitable de découpler la désignation de l'accès dans le choix des primitives du modèle de programmation: un nom doit être manipulable seul sans nécessairement mobiliser l'ensemble des ressources d'exécution pour l'établissement d'une communication ou d'une migration. Certains noms désignent les abstractions du médium de communication contenues dans le modèle de programmation, comme des canaux. Ils représentent des « capacités à communiquer » dont la possession permet l'établissement d'une communication avec une entité distante pour échanger divers types de valeurs. Des primitives de création de nouveaux noms sont également nécessaires. La portée des noms peut être statiquement déterminée par une primitive liante. Elle peut aussi être plus flexible: certains noms ont une portée potentiellement globale mais leur véritable signification dépend de leur site d'utilisation: ainsi, un agent mobile se lie dynamiquement à l'aide de noms bien connus aux ressources localement disponibles, par exemple, des bibliothèques.

---

b7) Exigence `REQ.DYNAMIC_BINDING` (Liaison dynamique).

⇒ *Un modèle de programmation pour le « global computing » doit fournir des primitives pour rendre compte de la dynamicité des liaisons aux ressources.*

Justification Un réseau à large échelle manifeste des variations importantes de bande passante (cf. observable `OBS.BANDWIDTH`) pouvant se traduire par une connectivité intermittente aux ressources. Un modèle de programmation pour le « global computing » doit donc refléter la non-accessibilité permanente de ses ressources. Les déconnexions sont fréquentes et observables, ce qui entraîne également la perte de la transparence réseau. La dynamicité de la liaison aux ressources se traduit aussi par la nécessité de pouvoir ajouter ou retirer dynamiquement des composants (cf. également l'exigence REQ.COMPOSITION), l'approche consistant à arrêter, mettre à jour, puis redémarrer l'ensemble du système n'étant pas réaliste à cause de sa taille. Le modèle doit donc fournir des primitives pour lier dynamiquement de nouvelles ressources aux composants en cours d'exécution. Ces primitives doivent se manifester au niveau même de la gestion des noms (cf. exigence REQ.NAMING), contrairement à l'approche traditionnelle de liaison statique dans les langages de programmation. On aboutit ainsi à un besoin général pour le modèle de programmation de « network awareness », un programme devant être attentif à l'environnement changeant dans lequel il s'exécute et aux ressources qu'il contient.

---

b8) Exigence `REQ.CONTROL` (Contrôle).

⇒ *Un modèle de programmation pour le « global computing » doit fournir des primitives pour programmer et contrôler le comportement des localités d'un réseau à large échelle.*

Justification Un environnement de type « global computing » manifeste entre ses différentes localités, diverses formes d'interaction et de synchronisation, de migration, de protection ou de composition. Face à la diversité de ces comportements, un modèle unique n'ayant qu'un nombre figé de primitives ne capturerait pas correctement la réalité. Le modèle doit être de plus capable d'accueillir de nouvelles sémantiques pour chacune de ces facettes de la programmation répartie mobile (cf. exigence REQ.OPENNESS), comme les exigences précédentes l'ont souligné. Il est donc impératif de rendre le modèle de programmation paramétrable selon ces différents comportements au niveau même où ils interviennent, celui de la localité, donc de définir des localités programmables et contrôlables dans le modèle. Une possibilité consiste à définir des localités « contrôlées » dont la frontière est réifiée explicitement sous forme de membrane programmable pour contrôler les ressources et unités d'exécution présentes dans la localité, et celles voulant y entrer ou en sortir. Un tel modèle de localités « augmentées » est résumé dans la figure 2.3, dans le cas où les localités sont organisées en une structure hiérarchique:

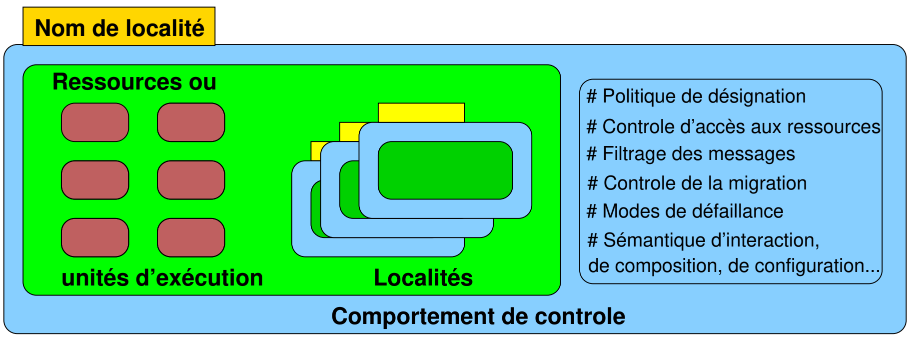


FIG. *2.3* – **Modèle hiérarchique de localités contrôlées.**

La supervision des activités contenues dans une localité peut nécessiter différentes formes de contrôle, depuis l'observabilité, pour examiner en utilisant des primitives d'introspection l'état d'une assemblée de composants en cours d'exécution, jusqu'à différents degrés de superposition [63] pouvant altérer l'exécution des composants d'une localité de manière plus ou moins intrusive à l'aide d'un comportement de contrôle spécifique, en passant par de l'interception pour filtrer les messages ou invocations entrant ou sortant de la localité. Ces formes de contrôle peuvent couvrir les aspects importants suivants:

- *Observabilité:* le comportement en cours d'exécution d'un composant d'un système à large échelle doit pouvoir être rendu manifeste à un observateur externe par la pose dynamique de points d'observation, contrairement à ce qui est réalisé actuellement dans les réseaux d'information [159] d'opérateurs de télécommunications, où l'observabilité d'un composant est définie de manière statique dès la phase de conception.
- *Sémantiques de liaison:* pour capturer les différents modes de composition et d'interaction entre les composants d'un système réparti, le modèle de programmation doit supporter des sémantiques arbitraires de *liaison* au sens du modèle de référence ODP [156] (cf. section 2.5), parmi les nombreuses déjà disponibles dans les infrastructures réparties courantes, comme la communication asynchrone, le multicast, ou la synchronisation entre plusieurs participants à un calcul réparti (cf. exigence `REQ.COORDINATION`).
- *Contrôle de la migration:* si l'on considère les localités comme des unités d'administration, on peut vouloir contrôler la migration d'unités d'exécution entre différentes localités, par exemple à l'aide de capacités autorisant ou non la migration et donc le franchissement de la frontière de la localité contrôlée. Il peut alors être nécessaire d'inclure dans le modèle de programmation des primitives de mobilité objective, permettant de déclencher la mobilité depuis l'extérieur de l'entité migrante. Ces primitives contrôlent aussi indirectement les communications de l'entité migrante: selon le choix de la localité de destination et sa politique de sécurité autorisant ou interdisant les communications vers l'extérieur, l'entité migrante pourra ou non communiquer, par exemple avec la localité de départ. Ce contrôle peut prendre la forme d'une négociation entre l'entité migrante et la localité cible.
- *Reconfiguration dynamique:* le contrôle de la migration apparaît comme un cas particulier de reconfiguration dynamique de la structure d'un système réparti dont les composants doivent pouvoir s'adapter à de nouveaux environnements. Le besoin de contrôle est principalement motivé par des exigences de flexibilité et d'adaptabilité du système. Un réseau à large échelle comporte différents niveaux de sous-systèmes manifestant entre eux des relations de dépendance, d'appartenance, de composition [159]. Le modèle de programmation doit fournir des primitives permettant d'accéder et de manipuler ces relations dans un but de reconfiguration, par exemple pour effectuer une opération de maintenance sur un composant défaillant, pour configurer dynamiquement un composant à cause d'un changement dans son environnement, pour ajouter un sous-système ou le mettre à jour, ou pour déplacer un composant vers un autre sous-système en vue de réaliser de l'équilibrage de charge. En général, ces fonctions de configuration sont présentes dans des systèmes à large échelle comme les réseaux de télécommunications, mais de manière statique, la reconfiguration nécessitant parfois l'arrêt total puis la reprise d'une partie du système. Ce besoin manifeste la nécessité d'une notion « riche » de localité comme primitive d'un modèle de programmation pour le « global computing », permettant de capturer avec un degré fin de granularité, la reconfiguration dynamique des composants d'un système réparti en terme d'agencement et de comportement.
- *Cycle de vie:* un aspect particulier de la reconfiguration des composants d'un système réparti est celui de leur cycle de vie, qui peut être complexe et se trouve au coeur des recherches sur les systèmes configurables [167] adaptables [103] et le génie logiciel [7]. Il doit pouvoir être reflété par les primitives du modèle de programmation, pour capturer des opérations de maintenance (installation, mise à jour, ou suppression de composants, activation, arrêt,...) ou de changements dans la structure

des composants (migration) ou dans l'environnement. En particulier, la liaison dynamique (cf. exigence `REQ.DYNAMIC_BINDING`) doit être reflétée au niveau du modèle afin de faire le lien avec les nombreux outils disponibles (cf. exigence `REQ.DESIGN_TO_IMPLEMENTATION`) pour la reconfiguration. La liaison tardive y joue un rôle important, pour ajouter, configurer, ou retirer des composants en cours d'exécution.

b9) Exigence `REQ.RESOURCES` (Ressources).

⇒ *Un modèle de programmation pour le « global computing » doit fournir des primitives pour contrôler et gérer de manière fine les ressources du système.*

Justification Un environnement de type « global computing » comprend de nombreux types de ressources, soit proche du matériel (e.g., mémoire, temps CPU, bande passante), soit de niveau d'abstraction plus élevé (e.g., connections virtuelles, sessions, canaux de communication). Ces ressources sont en général partagées (e.g., multiplexage des connexions dans les piles de protocoles). Le modèle de programmation doit disposer d'abstractions pour les représenter et de primitives pour les manipuler à différents niveaux de granularité. Ceci semble nécessaire si les coûts réel d'implantation doivent être reflétés dans le modèle (cf. exigence `REQ.NO_HIDDEN_COSTS`). La gestion comprend le contrôle d'accès à ces ressources ainsi qu'une mesure possible de leur utilisation (comptabilité de ressources).

Traits fondamentaux d'un modèle de programmation pour le « global computing » (synthèse)

On peut retenir des diverses exigences précédentes qu'un modèle de programmation pour le « global computing » doit posséder de manière fondamentale une notion de localité contrôlée permettant de représenter les activités concurrentes et réparties dans un système asynchrone et de les structurer de manière spatiale et logique. Elles servent de cadre de référence à l'exécution dont les étapes doivent s'y dérouler localement pour que le modèle de programmation soit implantable. Ce principe de localité signifie également que l'on ne peut pas accéder de manière transparente à des ressources distantes (« network awareness »). Les localités constituent aussi des unités de protection, de gestion des ressources ou de tolérance aux défaillances. Ces localités de natures différentes doivent pouvoir être superposées pour obtenir de nouvelles localités et refléter le passage à l'échelle. Le modèle doit fournir également des primitives de communication, mobilité, et coordination pour permettre à l'information de franchir les frontières de localités. Le caractère programmable des localités rend le modèle paramétrable selon ces sémantiques d'interaction, et permet de disposer de différents types de localités, ainsi que de rendre compte de la dynamicité du système en utilisant le comportement de contrôle de la localité à des fins d'introspection, d'interposition, de composition, ou de reconfiguration en cours d'exécution. Enfin, le modèle doit fournir une discipline de nommage permettant à la fois la liaison statique et dynamique de ressources.

La notion de domaine

Ayant identifié dans les paragraphes précédents les exigences que doit satisfaire un modèle de programmation pour le « global computing », il s'agit à présent d'identifier les abstractions rendant compte de ces exigences. Le but de ce paragraphe est de montrer que la plupart de ces exigences peuvent être capturées à l'aide d'une abstraction unique, celle de *domaine*. Pour plus de détails, on pourra se reporter au modèle de traitement [159] qui analyse formellement les exigences fondant le concept de domaine, et en tire les conséquences en terme de modélisation et de programmation. Nous tenons à présent le second axe majeur de la thèse, qui est l'identification des domaines comme abstraction fondamentale pour la programmation répartie sur des systèmes de grande taille.

Une première définition générale de la notion de domaine, extraite de [68], est la suivante:

> *A distributed system should primarily be understood as a partitioned system, with subsystems typically behaving asynchronously and in a largely autonomous fashion from one another. Components of a distributed system can be grouped in different, possibly overlapping sets, generally under the control of a single object or entity. [...] We shall call* domains *such sets of components or subsystems.*

Cette vision des réseaux à large échelle est en accord avec celle décrite par Luca Cardelli [37, 39] comme comportant de multiples frontières administratives et logiques, ou, de manière duale, des localités multiples (cf. observables `OBS.LOCALITY.P` et `OBS.LOCALITY.V`), par exemple servant de référentiel pour la mobilité [40]. Il s'agit ici d'aller plus loin, en décrivant un réseau à large échelle comme un système de systèmes, où coexistent un certain nombre de sous-systèmes largement autonomes les uns par rapport aux autres, et séparés par des barrières de différents types: chacune de ces régions délimitée par ces barrières dispose d'un comportement spécifique relatif à un aspect particulier de l'exécution répartie comme l'administration, les défaillances, la sécurité, ou la communication et se trouve sous le contrôle d'une autorité bien identifiée.

Ces régions ou *domaines* sont principalement des emplacements où peuvent se produire une exécution. Les domaines constituent un moyen uniforme de capturer les partitions que l'on trouve couramment dans la programmation de systèmes répartis de grande taille, comme des zones de confinement des défaillances, des unités de mobilité, ou des sous-systèmes sécurisés. Les domaines apparaissent liés de manière intrinsèque au caractère réparti du système, la répartition pouvant être identifiée à l'existence même de différents domaines d'exécution [29].

Une caractéristique importante d'un domaine est d'être un groupe autonome, organisé autour d'un comportement partagé par les entités membres du domaine (e.g., politique de contrôle d'accès, mécanisme de gestion des noms, mode de défaillance), selon un critère d'intérêt de l'exécution répartie (e.g., sécurité, nommage, défaillances).

La présence simultanée de plusieurs types de partitions du système global en sous-systèmes selon l'aspect d'exécution que l'on considère (sécurité, administration, désignation, défaillances, technologie, confiance...) aboutit à la coexistence dans le système global de différentes formes de domaines, représentant chacune une manière de partitionner le système selon un aspect d'exécution. Par exemple, un routeur entre deux sous-réseaux peut être vu comme une superposition d'un domaine de défaillance élémentaire, et d'un recouvrement de deux domaines de désignation et/ou de sécurité se partageant le routeur qui les relie.

La superposition de nombreux domaines de types différents rend nécessaire de disposer d'une primitive unique pour capturer ces partitions et manipuler ces comportements de manière uniforme au niveau d'un langage de programmation. Le caractère ouvert du système (cf. exigence `REQ.OPENNESS`) implique de pouvoir étendre le modèle de programmation en incluant de nouveaux types de domaines et leurs comportements associés. En introduisant une forme explicitement programmable de contrôle d'une localité, les domaines permettent d'ajouter simplement de nouvelles sémantiques de localités. Ainsi, il semble naturel pour un modèle de programmation pour le « global computing » d'être organisé autour d'une abstraction de domaine, considéré comme primitive de base permettant de programmer de manière explicite les comportements de contrôle des localités d'un système réparti à large échelle.

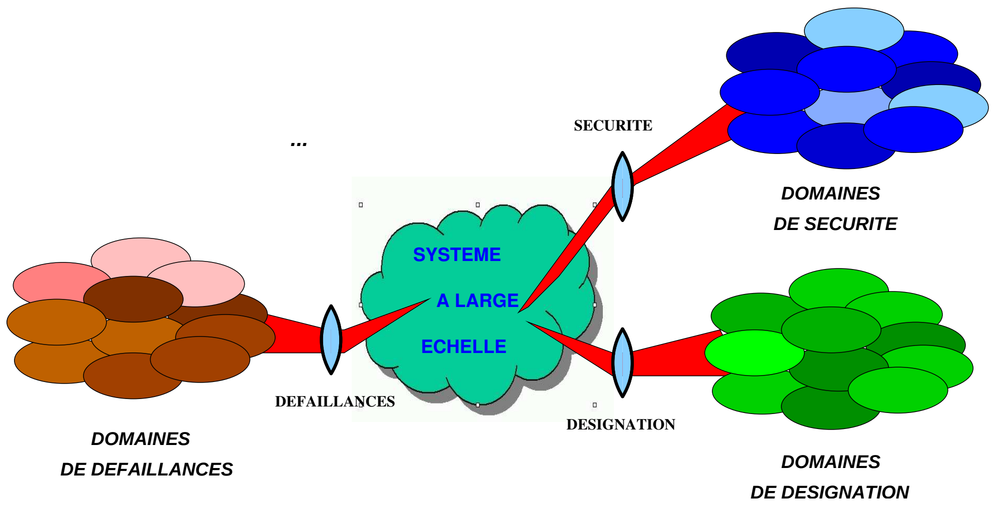


FIG. *2.4* – **Partitionnement multiple d'un système à large échelle.**

Un domaine doit ici être compris comme étant une *localité contrôlée*, c'est à dire la superposition:

- D'une *unité de composition:* c'est une abstraction permettant de partitionner spatialement ou logiquement une exécution répartie. Il s'agit donc d'une localité au sens traditionnel du terme: un endroit où peut se produire un calcul (cf. exigences `REQ.PROCESS`, `REQ.COMPOSITION` et `REQ.LOCALITY`).
- Et d'une *unité de contrôle:* c'est une primitive permettant de programmer un comportement de contrôle arbitraire (cf. exigence `REQ.CONTROL`) sur les activités d'exécution qui y sont hébergées. Par exemple, un comportement vis à vis des défaillances (cf. exigence `REQ.FAILURES`), une politique de sécurité (cf. exigence `REQ.SECURITY`), ou une manière de gérer la mobilité (cf. exigence `REQ.MOBILITY`).

Plus formellement, un domaine peut-être vu comme la composition $\omega[C]$ de deux éléments [33, 68, 159]:

- Une *portée* C ou *contenu du domaine* qui en délimite la frontière et décrit l'ensemble des entités qui font partie du domaine et qui se trouvent sous son contrôle.
- Une *membrane programmable* ou *contrôleur du domaine* ω qui décrit le comportement de contrôle du domaine sur les entités situées dans sa portée. Ce comportement définit la sémantique particulière du domaine. De nouvelles sémantiques de domaines peuvent être introduites en programmant explicitement un nouveau contrôleur de domaine.

Le contrôleur de domaine doit permettre une forme d'interception et de traitement des communications entrant et sortant du contenu d'un domaine, jouant ainsi le rôle d'un filtre abstrait de messages. Un exemple typique d'application d'un tel filtre est constitué d'un pare-feux qui réalise un domaine de sécurité autour d'un groupe d'entités, et veille à l'application d'une politique de sécurité des communications entre ces entités et l'environnement externe. Le contrôleur doit pouvoir également inclure un comportement de supervision plus riche se superposant aux activités d'exécution hébergées dans le domaine, par exemple permettant la passivation d'entités du domaine destinées à être communiquées à l'extérieur pour réaliser de la migration, ou la suspension puis la reprise de l'exécution d'un composant du domaine après une défaillance.

De manière générale, un domaine doit pouvoir évoluer, tout d'abord de manière passive en laissant évoluer les activités qu'il héberge, ce qui pourraît être modélisé dans un modèle de programmation pour le « global computing » par une règle du type [28, 159]

$$\frac{C \ \rightarrow \ C'}{\omega[C] \ \rightarrow \ \omega[C']}$$

sauf pour des domaines de défaillances ou des comportements de contrôle actifs. Le contrôleur de domaine peut également être actif, en modifiant l'état de la portée du domaine, par exemple en permettant dynamiquement la création de nouveaux domaines, en émettant ou recevant de nouveaux messages, ou en offrant des primitives pour la migration du contenu d'un domaine d'un domaine à l'autre, en sérialisant son état complet dans une structure transmise par message sur le réseau.

L'organisation des domaines peut être *plate* – tous les domaines étant situés au même niveau – ou *hiérarchique* – la portée d'un domaine pouvant contenir d'autres domaines ou *sous-domaines*. Un modèle hiérarchique, comme celui décrit dans la figure 2.3, a l'avantage de traiter de manière uniforme les domaines comme des unités atomiques de défaillances ou de migration au sein d'une arborescence de domaines [58], ainsi que de simplifier la gestion de la sécurité, en considérant en première approximation un système à large échelle comme une hiérarchie de sous-réseaux et de machines (domaines administratifs).

Les domaines peuvent se recouvrir partiellement, comme dans l'exemple du routeur cité plus haut: les entités appartenant à plusieurs domaines sont alors sous le contrôle conjoint de leurs contrôleurs respectifs. La conséquence immédiate sur l'organisation des domaines est que ceux-ci ne forment pas véritablement des partitions ou des structures arborescentes, mais dans le cas général constituent des structures de graphes. Ainsi, un système comme Internet peut-il être vu comme une fédération de hiérarchies de sous-réseaux, interconnectée à différents niveaux par des routeurs ou des passerelles. Ces structures complexes sont encore peu présentes au niveau des travaux actuels de modélisation, qui la plupart du temps se limitent à des structures arborescentes.

Outre le caractère distinctif des domaines de constituer des unités de composition et de contrôle – un domaine abritant un certain nombre de processus et de ressources, avec un processus particulier en charge du comportement de contrôle – on peut citer comme caractéristiques communes aux domaines [29]:

- *Les domaines sont nommés:* ce trait leur permet d'être identifiés de manière non-ambiguë, par exemple comme cible d'une communication ou d'une migration.
- *Les domaines sont observables:* un domaine n'interagit avec son environnement qu'à travers des points d'accès bien identifiés (interfaces) [33]. Le domaine peut exercer un contrôle sur la visibilité de ces interfaces vis-à-vis d'autres domaines ou de l'environnement externe. Des techniques comme les bisimulations ou les équivalences permettent de comparer et de raisonner sur le comportement observables des domaines, avec des degrés de finesse variables.

Quelques exemples

Dans un environnement de type « global computing », on trouve de nombreux types de domaines différents:

- Les *domaines d'exécution* constituent l'exemple le plus évident de domaines dans un système réparti. Il s'agit de localités dont les frontières capturent les barrières entre entités du système physiquement distantes. Les délais de communication ne peuvent pas être abstraits car bornés inférieurement par un temps observable de propagation de l'information. En général, ces domaines contiennent des ressources (processeurs, mémoire, bande passante) et gèrent leur allocation. Les unités d'exécution sont typiquement des threads, des processus système, des machines ou des sous-réseaux. Elles ont la caractéristique commune d'être situées à une adresse réseau donnée.
- Les *domaines administratifs* ou *d'administration* sont un autre type important de domaine, car formant l'ossature des réseaux à large échelle. Entourés de pare-feux, l'accès aux ressources y est contrôlé selon une politique de sécurité commune. Ils sont souvent de véritables *domaines de sécurité*, qui protègent une région du monde extérieur, où n'ont accès aux ressources que les entités dignes de confiance. Comme exemples, on peut citer la technique de « sandboxing » mise en oeuvre dans la machine virtuelle Java ou la protection d'un réseau local par des pare-feux. Le comportement de supervision du domaine comprend en particulier le contrôle des autorisations de franchissement de la frontière du domaine. Ce type de domaine fut l'un des premiers à être introduit en informatique, avec, dans les années 70, de nombreux travaux sur le support au niveau système d'exploitation et même matériel de domaines de sécurité. Par exemple, le système MULTICS [154] était composé d'anneaux concentriques de niveaux croissants de sécurité. D'autres incarnations de domaines de sécurité furent réalisées à base de systèmes à capacités [94, 107], au niveau logiciel [179], ou à l'aide de machines dédiées comme le Plessey 250 [88]. La fuite possible d'information hors d'un domaine de sécurité pose le problème dit du *confinement* [93] qui se révèle être indécidable.
- Les *domaines de communication et de coordination* sont des régions au sein desquelles les unités d'exécution peuvent échanger de l'information et synchroniser leurs activités [28]. Par exemple, il peut s'agir d'un espace d'adressage, d'une machine ou d'un réseau local, ou encore d'une cellule radio dans un réseau mobile.
- Les *domaines de mobilité* regroupent des activités d'exécution qui peuvent être déplacées ensemble dans un réseau. Il peut aussi s'agir d'emplacements capables d'accueillir des unités mobiles et constituant des référentiels pour la mobilité. Les domaines de mobilité peuvent être des objets, composants ou agents mobiles ou bien des serveurs d'agents, agences ou places d'exécution. Les entités membres d'un domaine de mobilité suivent une politique commune de migration. La sémantique de ces domaines est liée à celle des domaines de défaillances, la migration pouvant être utilisée pour contourner des zones de pannes.
- Les *domaines de défaillances* regroupent des processus ou localités qui tombent en panne simultanément en partageant un mode de défaillance commun, par exemple *fail-stop*.
- Les *domaines de désignation* regroupent des entités qui sont désignées suivant une certaine politique de nommage commune, par exemple, en utilisant un même schéma de formation d'identificateurs, ou un même protocole de gestion des références.

- Enfin les *domaines de* technologie regroupent des technologies matérielles ou logicielles communes utilisées à l'intérieur d'un sous-système comme un sous-réseau supportant un même ensemble de protocoles de communication.

On pourrait également citer les *domaines de performances* traduisant les fluctuations imprévisibles de bande passante et la congestion réseau qui partitionnent le réseau en régions où les liens de communication ont des temps de réponse similaires. De même, les *domaines de confiance* où les participants à un calcul réparti ne font confiance qu'aux membres du domaine et se méfient des éléments extérieurs.

La présence de ces nombreux types de domaines dans un système réparti à large échelle, souligne l'intérêt de disposer d'une primitive uniforme pour modéliser et programmer de tels systèmes, considérés dans toute la suite comme superposition d'un ensemble de domaines.

2.2 Problématique de la thèse

Une double difficulté

Dans la section précédente, nous avons identifié deux difficultés principales à la programmation d'applications réparties fiables sur un réseau à large échelle:

- *Une divergence profonde entre théorie et pratique de la programmation répartie:* cette difficulté souligne la nécessité d'une approche de bout en bout (cf. exigence `REQ.DESIGN_TO_IMPLEMENTATION`), à travers la définition d'un modèle de programmation implémentable dans un contexte réparti (cf. exigence `REQ.EFFECTIVENESS`) et ne masquant pas les coûts réels d'implémentation (cf. exigence `REQ.NO_HIDDEN_COSTS`).
- *Une insuffisance des techniques de modélisation et de programmation pour un environnement de type « global computing »:* cet environnement, bien différent d'un réseau local, rend nécessaire la définition de nouveaux modèles de programmation satisfaisant un certain nombre d'exigences fondamentales pour rendre compte d'un petit nombre d'observables (cf. section 2.1). Ceci peut être réalisé à l'aide d'une abstraction primordiale, la notion de *domaine*, qui apparaît comme fondement naturel de tout modèle de programmation de systèmes répartis de grande taille.

La problématique

La problématique de départ qui motive notre étude est la suivante: est-il possible de construire des infrastructures réparties fiables dans un environnement de type « global computing »? Cette question recouvre en fait deux problèmes bien distincts:

- *Le problème de la sûreté.* Traiter cet aspect est nécessaire pour que l'infrastructure soit digne de confiance. Une solution consiste à utiliser des méthodes formelles dans le processus de développement car elles apportent des garanties de correction et de sûreté. Malheureusement, ces méthodes sont souvent coupées de la pratique de la programmation. S'attaquer à ce problème signifie donc adopter une approche qui réconcilierait théorie et pratique de la programmation répartie. Cela semble possible par une formalisation de bout en bout d'un système.

- *Le problème du « global computing »*. Ce domaine de recherche reste largement inexploré. Pouvoir le modéliser signifie disposer d'abstractions suffisamment génériques pour refléter au niveau théorique la diversité des situations recouvertes par ce type d'environnement, et au niveau pratique les infrastructures existantes. Cela semble possible à l'aide de la notion très générale de domaine.

La question de départ peut alors être reformulée ainsi: est-il possible de réduire l'écart entre théorie et pratique de la programmation répartie en disposant d'abstractions suffisamment puissantes pour capturer la large variété de situations du « global computing »?

L'objet de cette thèse est précisément d'apporter des éléments de réponse à cette double question. La solution adoptée repose sur les deux pistes suggérées précédemment: la formalisation de bout en bout et la mise en oeuvre du concept de domaine.

Dans les travaux précédents, une approche de formalisation de bout en bout reste rare, et n'a été utilisée que dans quelques cas pour des systèmes centralisés, ou exhibant un faible degré de répartition. Elle semble avoir été peu appliquée à des systèmes du monde du « global computing », domaine de recherche très récent.

De même, la notion de domaine, manifeste dans de nombreux aspects de la programmation répartie, a été mise en évidence séparément dans chacune de ses déclinaisons depuis assez longtemps – par exemple, l'isolation de la notion de domaine de protection dans les systèmes d'exploitation durant les années 70. Par contre, les domaines n'ont été reliés que depuis peu au thème du « global computing ». Son identification comme abstraction fondamentale du « global computing » continue de faire l'objet de recherches actives, une théorie des domaines restant à construire – par exemple, dans le projet MIKADO [123] et l'ensemble de l'initiative IST Global Computing [70].

La combinaison de ces deux éléments, soit une formalisation de bout en bout d'un système construit à base de domaines, semble donc être une approche qui pourrait ouvrir de nouvelles perspectives pour la programmation d'application réparties fiables évoluant dans un environnement de type « global computing ».

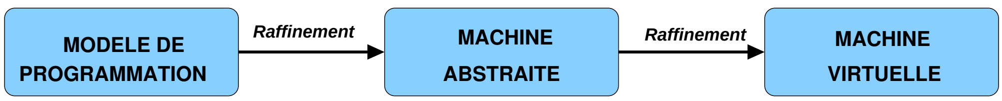


FIG. 2.5 – **Démarche de formalisation de bout en bout.**

Approche proposée: modèle $\rightarrow$ machine abstraite $\rightarrow$ machine virtuelle

Dans cette thèse, nous tentons de surmonter les deux difficultés identifiées précédemment en utilisant une méthodologie simple pour la construction d'infrastructures réparties de grande taille dignes de confiance. Cette démarche peut se décomposer en trois étapes (cf. figure 2.5):

- *Définition d'un modèle formel de programmation répartie fondé sur la notion de domaine:* le modèle doit être soigneusement élaboré, minimal, et disposer d'une sémantique opérationnelle formellement définie. Ses primitives reflètent les exigences déjà identifiées pour un modèle de programmation répartie à large échelle comme le contrôle, la composition, la reconfiguration dynamique, ou le fait de pouvoir être implanté efficacement. Le modèle est en particulier le fondement théorique et le cœur d'un langage. Il comprend les primitives minimales d'un langage de programmation de systèmes de grande taille comprenant répartition et mobilité. Il rend possible l'utilisation de techniques de preuves formelles et d'outils de vérification pour disposer de garanties rigoureuses sur l'exécution de programmes décrits par ce modèle.
- *Spécification formelle d'une implémentation du modèle par une machine abstraite:* elle est en général formellement décrite par un état qui offre une vue de haut niveau des structures d'exécution de l'implémentation du modèle, et par des règles de réductions, qui décrivent un pas de calcul élémentaire menant d'un état au suivant. La machine abstraite est un raffinement du modèle, au sens où un pas de calcul au niveau du modèle se traduit en général par plusieurs pas d'exécution au niveau de la machine abstraite. L'intérêt d'une machine abstraite est aussi d'être indépendant d'un langage d'implantation. Elle peut donc donner naissance à diverses implémentations du modèle réalisées dans différents langages de programmation, argument important compte tenu de l'hétérogénéité d'un réseau à large échelle.
- *Implémentation de la spécification formelle par une machine virtuelle répartie utilisant les technologies middleware courantes:* l'implémentation doit se conformer au modèle et être la plus efficace possible.

Nous développons ainsi l'idée que le développement d'infrastructures réparties fiables pour la programmation sur des réseaux de grande taille est rendu possible par l'utilisation d'une démarche de formalisation par raffinements successifs où la notion de domaine joue un rôle prépondérant, depuis le modèle de programmation jusqu'à son implantation par une machine virtuelle répartie.

Le but de la thèse est de rendre explicite ce cheminement, en montrant qu'à chaque étape les domaines sont au cœur de la démarche, et en illustrant les raffinements qui font passer d'une étape à l'autre. En particulier, nous montrons comment les domaines vus comme primitives d'un calcul de processus, c'est-à-dire comme localités programmables, sont raffinés dans des domaines d'exécution couramment rencontrés dans les plates-formes à objets répartis, c'est-à-dire comme composants éventuellement mobiles ou comme conteneurs de composants.

Cette approche présente un double avantage. D'une part, le modèle fournit un cadre théorique fort pour réaliser des preuves formelles de propriétés sur le système. D'autre part, la machine virtuelle valide le modèle tout en reflétant les techniques d'implémentation middleware, aujourd'hui largement répandues.

Dans la suite du chapitre, nous passons en revue les diverses manifestations de domaines dans l'état de l'art correspondant à chacune des étapes de ce cheminement, soit: les modèles formels et langages pour la programmation concurrente et répartie (cf. section 2.3), les machines abstraites et virtuelles pour calculs de processus (cf. section 2.4) et les plates-formes middleware et d'exécution répartie (cf. section 2.5). L'organisation de la suite du mémoire est alors présentée (cf. section 2.6).

2.3 Modèles formels et langages de programmation concurrente et répartie

Le but de cette section est de donner un bref survol de quelques modèles et langages importants pour la programmation concurrente et répartie, en mettant l'accent sur les différentes formes de domaines qui s'y manifestent. Pour une analyse comparative plus détaillée de ces modèles, on pourra par exemple se reporter à [29], [30], ou à [41].

2.3.1 La machine chimique

GAMMA

Dans [12], Banâtre et Le Métayer introduisent GAMMA, nouveau modèle d'exécution pour la programmation concurrente, basé sur la transformation de multi-ensembles. L'idée est de cacher la distribution physique des ressources qui sont rassemblées au sein d'un seul multi-ensemble S, sur lequel opèrent des transformations successives ne modifiant à chaque fois qu'une partie de S. Plus précisément, S subit des réécritures locales, spécifiées par des *fonctions d'action* A et par des *conditions de réaction* R. Les conditions de réaction sont des prédicats sur ces multi-ensembles qui autorisent la poursuite de l'exécution. L'avantage du modèle GAMMA est d'offrir une vue globale et symétrique de l'exécution, de nombreuses réécritures pouvant se produire simultanément.

Ce modèle peut être interprété à l'aide d'une métaphore chimique: le multi-ensemble de ressources S devient une *solution chimique* dans laquelle se déplacent au hasard des *molécules*, et les couples fonction d'action A-condition de réaction R deviennent des *règles de réaction chimiques* qui consomment des molécules existantes et en produisent de nouvelles. Le mouvement d'agitation des molécules est ignoré, les réactions se produisant au hasard de leurs collisions.

Le modèle vérifie une propriété importante, dite *de localité*: toute réécriture effectuée sur un multi-ensemble donné peut aussi être effectuée sur un multi-ensemble plus grand. Aussi, lors du déclenchement d'une réaction, devient-il possible de s'abstraire de l'environnement des réactifs et des autres réactions se produisant dans la solution. Ce modèle se prête aisément à une implémentation massivement parallèle ou répartie, une machine pouvant être considérée comme un *site de réaction* où sont co-localisées un certain nombre de règles de réaction. Une machine constitue ainsi une première forme de domaine d'exécution.

La CHAM

Dans [14], Berry et Boudol réutilisent la métaphore chimique à base de multi-ensembles de molécules introduite dans GAMMA en fournissant un cadre de référence théorique pour décrire la sémantique opérationnelle d'une classe importante de modèles de programmation concurrente et répartie, les calculs de processus (cf. section 2.3.2). Le modèle prend la forme d'une machine abstraite d'exécution, la machine chimique ou CHAM (Chemical Abstract Machine).

Dans les systèmes de réécriture, une étape de calcul est capturée par une *relation de réduction* $\rightarrow$ qui en contient la sémantique et permet de transformer un terme en un autre terme. Dans un système de réécriture concurrent, un *redex* ou expression simplifiable est constitué de plusieurs sous-termes dont les positions ne sont pas nécessairement contiguës au niveau de la syntaxe. La machine chimique simplifie la définition de la relation de réduction en éliminant les règles de commutativité et d'associativité de l'opérateur de composition parallèle permettant de rapprocher syntaxiquement les sous-termes d'un redex. Dans la métaphore chimique, ces règles de réarrangement syntaxique prennent la forme de règles structurelles réversibles $\rightleftharpoons$ de recombinaison entre molécules, soit en cassant une molécule en composants individuels (étape de réchauffement $\rightharpoonup$) ou en constituant un agrégat à partir de plusieurs molécules élémentaires (étape de refroidissement $\leftharpoondown$). La métaphore inclut également des règles de réaction irréversibles $\rightarrow$, qui consomment certaines molécules de la solution chimique et en produisent de nouvelles. Ces dernières règles traduisent les pas de calcul élémentaires du programme en cours d'exécution. Pour résumer, la machine chimique simplifie la présentation de la sémantique d'exécution d'un calcul de processus en l'écrivant comme constituée d'une relation de réduction $\rightarrow$, considérée à une équivalence structurelle $\rightleftharpoons$ près.

Par rapport à GAMMA, la machine chimique permet aussi de contrôler les portées syntaxiques, en délimitant et en isolant une sous-solution active du reste de la solution chimique grâce à une primitive de *membrane*. La solution chimique globale apparaît alors comme une hiérarchie de solutions actives. Une primitive d'*airlock* permet également de rendre disponibles au travers d'une membrane certaines capacités de communication, et ce de manière réversible.

La machine chimique est un modèle abstrait pour la programmation concurrente, qui capture les exigences de composition, d'activité, et d'asynchronisme. Elle dispose d'une forme simple de domaine, car permettant de délimiter grâce aux membranes des endroits ou des portées où peuvent se dérouler un calcul. En terme d'implémentation, certaines opérations du modèle peuvent apparaître « magiques », comme le transport des molécules d'un site de réaction à l'autre. Comme le remarque [58], certains réarrangements réversibles de la CHAM peuvent être irréversibles en pratique, par exemple si une machine tombe en panne, devenant ainsi observables alors qu'ils ne traduisent pas dans le modèle une progression de l'exécution. Cependant, la machine chimique reflète certains coûts d'implémentation: comme GAMMA, en définissant une notion de site de réaction, il devient possible de distinguer les règles activables localement, et celles nécessitant une synchronisation entre plusieurs sites. Une extension du modèle comprenant localité et réflexion a d'ailleurs été proposée dans [59] et constitue le modèle computationnel du join–calcul (cf. section 2.3.4): une localité regroupe un nombre fixé de règles de réaction; on obtient du parallélisme à grain fin en limitant à la portée d'une localité les synchronisations entre messages en attente de communication. En outre, de nouvelles règles peuvent être introduites dynamiquement.

Cependant, la machine chimique n'est pas un véritable modèle de programmation. Il s'agit plutôt d'un modèle computationnel abstrait – ou d'un *modèle de traitement* pris au sens de [159] – pour les calculs de processus. Leur description fait l'objet de la section suivante.

2.3.2 Premières algèbres de processus

Les *calculs de processus* constituent une classe importante de modèles permettant la description et la spécification de systèmes concurrents. Ils fournissent un cadre formel pour exprimer puis prouver des propriétés du système étudié, par exemple à l'aide de techniques comme les bisimulations. Considérés également comme cœurs de langages de programmation, ils offrent des primitives permettant la programmation explicite de systèmes manifestant concurrence et répartition.

Ces modèles sont nés des travaux de Milner sur les systèmes concurrents [125, 126, 128]. Ils prennent toute leur importance dans la modélisation de systèmes pour le « global computing », déjà identifiés comme formés par un ensemble de composants s'exécutant en parallèle, pouvant éventuellement coordonner leurs activités en vue d'un but commun. Un calcul de processus décrit un système comme constitué d'un certain nombre d'unités d'exécution appelées *processus*, assemblées à l'aide d'un opérateur de composition parallèle noté $|$. Le comportement d'un processus P est décrit par un système de transitions $P \rightarrow P'$. L'opérateur de composition parallèle est asynchrone et se comporte comme un produit: chaque branche peut évoluer indépendamment de l'autre,

$$\begin{cases} P \rightarrow P' \Rightarrow P \mid Q \rightarrow P' \mid Q \\ Q \rightarrow Q' \Rightarrow P \mid Q \rightarrow P \mid Q' \end{cases}$$

car il n'existe pas de synchronisation a priori entre les processus (cf. exigence `REQ.ASYCHRONY`).

CCS (Calculus of Communicating Systems) [124, 125, 126] constitue l'un des premiers modèles de programmation concurrente avec CSP (Communicating Sequential Processes) [78] et les réseaux de Pétri [32]. Chaque processus possède un nombre fixe de canaux de communication sur lequel il peut échanger des données avec les autres processus. La communication consiste en un rendez-vous sur un canal entre un processus émetteur et un processus récepteur. La topologie de communication, statique, est uniquement structurée par les noms de canaux, compris comme des capacités à communiquer déclenchant les différentes primitives du calcul (émission, réception). Ces opérations confèrent au modèle une structure d'algèbre, faisant ainsi de CCS la première *algèbre de processus*.

2.3.3 Le π–calcul

Extension de CCS, le π–calcul de Milner, Parrow et Walker [127, 128, 129] constitue un formalisme largement plus expressif qui s'est imposé comme modèle de référence de la programmation concurrente [150]. Contrairement à CCS, la topologie de communication du π–calcul évolue au cours du temps: les noms de canaux sont transmis comme valeurs sur des canaux existants, ce qui permet à un processus de recevoir un canal nouvellement créé et d'acquérir ainsi de nouvelles possibilités de communication. Le π–calcul capture une certaine forme de mobilité, la mobilité de noms de canaux.

De nombreux travaux ont été réalisés pour appliquer le π–calcul à des domaines comme la sécurité ou la mobilité. Par exemple, le spi–calcul [4] étudie l'extension du π–calcul avec des primitives cryptographiques. Il contient une forme limitée de domaine de sécurité, dans la mesure où le chiffrement d'une donnée introduit une frontière autour de cette donnée. Une autorisation (capacité) est nécessaire pour franchir cette frontière et prend la forme d'une clé de chiffrement [37]. De même, le pi-calcul d'ordre supérieur [146, 147] (HOπ)

explore une première forme de domaine de migration, puisque, grâce à l'ordre supérieur, il permet de transmettre un processus en cours d'exécution sur un canal de communication.

Le π–calcul a également été utilisé pour étudier les liens avec d'autres paradigmes de programmation. Par exemple, le calcul bleu [27, 48] étudie les rapports entre π–calcul et programmation fonctionnelle en réalisant une synthèse entre λ–calcul et π–calcul, combinant deux types d'unités élémentaires de calcul et de mécanismes de composition, les fonctions et l'application fonctionnelle d'une part, les processus et la composition parallèle d'autre part. De même, Sangiorgi [148] et Dal-Zillio [48] étudient une interprétation d'un calcul à objets, le ς–calcul [1] dans des variations du π–calcul typé.

Le π–calcul n'est pas intrinsèquement réparti et ne dispose pas de primitive de localité. La portée des noms est globale, à l'intérieur d'une localité unique et implicite. Une implémentation répartie du π–calcul est difficile à obtenir car nécessitant une forme de consensus réparti, pour choisir parmi n possibles le processus récepteur qui recevra effectivement un message envoyé sur un canal de communication donné. Une version simplifiée du π–calcul, le π–calcul asynchrone, dont les bases ont été établies simultanément par Boudol [26] et Honda et Tokoro [79], abandonne la communication synchrone du π–calcul comme primitive de base en découplant émission et réception des messages qui ne doivent plus être nécessairement simultanés. Cependant, les décisions de communication ne sont toujours pas localisées. Une étape décisive de l'émergence de la notion de domaine dans les calculs actuels a consisté à introduire diverses formes de *localités* dans un π–calcul asynchrone pour capturer la distribution [41]. Ces primitives sont aussi appelées selon les cas *sites*, *systèmes*, ou *réseaux*.

Actuellement, on peut classer les calculs de processus répartis en trois grandes catégories:

- *Les déclinaisons du π–calcul asynchrone avec localités :* dans ces calculs, la primitive de base est la communication par envoi de messages entre différents types de localités capturant la communication, la mobilité, la sécurité, la tolérance aux défaillances ou les transactions. Différentes stratégies de routage des messages et de migration y sont explorées. On peut par exemple citer Dπ [145] et sa version réceptive [10], le $\pi_{1\ell}$–calcul [8], le join–calcul [59], NOMADIC–π [155] (cf. section 2.4.1), le seal–calcul [171] et l'ATF–calcul [53]. La figure 2.6 résume les principales caractéristiques de cette famille de calculs.

- *Les variantes du calcul des ambients :* dans cette famille de calculs, les primitives de base sont des instructions de migration pour entrer ou sortir de localités appelées *ambients*, différents modèles de communication au sein d'une hiérarchie d'ambients étant étudiés. On peut citer ici les MOBILE AMBIENTS [40], les SAFE AMBIENTS [106] et les BOXED AMBIENTS [34] (cf. section 2.3.5).

- *La combinaison des algèbres de processus avec d'autres paradigmes de programmation :* il s'agit d'études d'application du π–calcul avec localités au cas de la programmation-orientée objets ou de schémas de coordination à base d'espace de tuples partagés à la LINDA [65]. On peut citer ici DITYCO [169] et KLAIM [50].

Dans ces calculs, les domaines regroupent un certain nombre de ressources qui partagent un comportement suivant un critère d'intérêt comme la mobilité, la tolérance aux défaillances, etc. Les domaines apparaissent souvent de manière semi-explicite, avec pour chaque calcul une sémantique unique, cette sémantique dépendant fortement de l'orientation du calcul.

2.3.4 Variantes du π–calcul à localités

Dπ

Dπ [77, 145] est une extension répartie du π–calcul, qui étudie le contrôle de l'accès aux ressources dans les systèmes d'agents mobiles. Les agents sont représentés par des threads localisés pouvant migrer d'une localité à l'autre. Les principales abstractions du calcul sont les *threads* qui s'exécutent dans des *localités* nommées. Dans Dπ, les domaines s'identifient avec les localités du calcul, et sont organisés de manière plate. Ils peuvent être créés dynamiquement, mais ne disparaissent jamais au cours de l'exécution. Outre l'aspect domaine d'exécution, une localité est utilisée pour contrôler l'accès aux canaux de communication à l'aide d'un système de types. Dπ impose à la communication d'être purement locale et synchrone, deux threads ne pouvant communiquer que par rendez-vous à l'intérieur d'une même localité. La communication distante n'est pas primitive et nécessite une migration préalable de l'émetteur du message vers la localité du récepteur. La migration est bloquante et objective, les unités de mobilité étant les threads se déplaçant de localité en localité.

Le π–calcul réceptif Dπ_1^r [10] est une version asynchrone de Dπ qui cherche à garantir une propriété dite de *réceptivité*: un message ne peut être envoyé vers une destination où il ne trouverait pas de récepteur. Comme dans Dπ, la communication est purement locale, le routage des messages étant explicite. Par rapport à Dπ, le π–calcul réceptif ne fait pas de différence entre processus localisé et processus en cours de migration. La réception est statique : il est impossible de créer un récepteur pour un nom ayant déjà été reçu. Une propriété *d'unicité des récepteurs* est également vérifiée: un seul récepteur est admis pour un nom de canal donné afin d'éviter le coût d'implémentation d'un algorithme d'élection répartie pour déterminer le récepteur devant traiter les messages sur ce canal.

Le $\pi_{1\ell}$–calcul

Cette propriété d'unicité des récepteurs est également au cœur du $\pi_{1\ell}$–calcul [8, 9], extension du π–calcul asynchrone avec une primitive de localité et un modèle de défaillances. Dans $\pi_{1\ell}$, les domaines sont des *localités* abritant des *processus*. L'organisation des localités est plate. Chaque localité dispose d'un unique processus de contrôle contenant l'état actif ou en panne de la localité et constitue un domaine de défaillances: une localité en panne ne peut plus émettre de messages ou accepter de processus en cours de migration. Le comportement de contrôle du domaine se limite à observer son état (`ping`), ou le faire tomber en panne de manière irréversible (`stop`). La portée des récepteurs est répartie, ce qui les rend accessibles depuis n'importe quelle localité. La communication distante est primitive et le routage des messages implicite. Outre la migration des messages, les primitives de mobilité se limitent à déclencher l'exécution d'un processus dans une localité distante active (`spawn`).

Le join–calcul

Né de la difficulté d'implémenter le π–calcul dans un contexte asynchrone et réparti, le join–calcul [58, 59, 60, 109] constitue une autre tentative pour exprimer les fondements théoriques de la programmation répartie à large échelle. Les canaux du π–calcul n'étant pas localisés, on peut aboutir rapidement à un consensus réparti dès que l'on dispose de plus de deux récepteurs sur un même canal, afin de déterminer le récepteur auquel revient la communication. Le join–calcul réparti résoud le problème au niveau de la syntaxe, en imposant à chaque récepteur d'être statiquement défini dans une localité unique, et en co-localisant les processus capables de recevoir sur un même nom de canal. La décision de synchronisation peut alors se prendre localement. Les abstractions principales introduites sont les *processus* qui communiquent suivant des règles spécifiées dans des *définitions*, et les *localités* qui sont des domaines d'exécution, de migration et de défaillances, organisés en une structure arborescente. Une primitive spéciale, le *join-pattern* (filtre de messages) permet d'exprimer la synchronisation des messages. La contrainte d'unicité des récepteurs rend possible un routage implicite et déterministe, puisque le récepteur cible d'un message est connu sans ambiguité. En terme de communication distante, le join–calcul obéit à la sémantique « move-and-communicate » : migration du message vers la localité distante et consommation par une règle de communication locale. Contrairement aux autres calculs, il ne s'agit pas de migration de processus, mais de migration de messages, le join–calcul disposant d'une règle de communication distante distincte de celle de migration. En join–calcul, les domaines se manifestent comme unités « topologiques » de migration et de défaillances: une localité migre ou tombe en panne avec l'ensemble des processus qu'elle héberge et avec toutes ses sous-localités. Au niveau contrôle, l'état d'une localité (active/en panne) peut être observé depuis l'extérieur. Cependant, les localités ne constituent pas des domaines de communication puisque l'interaction distante entre localités reste permise. Récemment, des extensions du join–calcul ont été proposées avec des primitives pour la liaison dynamique [151, 152], la cryptographie [2], et l'authentification [3].

Le seal–calcul

Le seal–calcul [171, 172] étudie les propriétés de sécurité de processus mobiles évoluant dans des systèmes répartis de grande taille: la communication entre localités est limitée et l'accès aux ressources est contrôlé. Les principales abstractions du seal–calcul sont des domaines nommés appelés *seals* hébergeant des *processus* à la manière du π–calcul et des *ressources* sous forme de canaux explicitement localisés pour la communication entre processus. Un seal est un domaine de sécurité et de mobilité: il contrôle à la fois l'accès aux canaux qu'il abrite et se comporte comme une unité atomique de migration. Les processus consistent en séquences de capacités de communication et de franchissement de barrière de domaine, la répartition de ces capacités entre les différents domaines assurant la sécurité du système. L'organisation des domaines est arborescente. La communication locale est synchrone. La communication distante n'est possible qu'entre processus hébergés par des seals ayant une relation hiérarchique directe dans l'arborescence des domaines. Pour cela, le seal s_1 qui possède le canal de communication doit en autoriser l'utilisation à l'aide de capacités appelées *portails* au seal s_2 (parent ou enfant de s_1) abritant le processus qui engage la communication. L'envoi du nom d'un seal s sur un canal de communication se traduit par la migration de s vers le seal qui possède le canal. Le contrôle de la migration est donc effectué par le même mécanisme que pour la communication. Un seal est donc un véritable domaine de protection: il limite la diffusion des noms de ressources grâce au mécanisme de portée lexicale et contrôle finement leur utilisation à l'aide de capacités lors de chaque étape de migration ou de communication.

L'ATF–calcul

L'ATF–calcul (ATomic Failure) [53, 54] se démarque des modèles précédents en se focalisant sur le traitement atomique (transactionnel) des défaillances et de la reprise après panne dans une exécution répartie, le mode fail-stop généralement adopté par les autres calculs étant jugé trop limitatif dans un contexte asynchrone. Dans ce calcul, outre leur rôle traditionnel d'unités d'exécution et de communication, les domaines se manifestent sous forme d'unités atomiques de défaillances et de reprise. Le calcul introduit une hiérarchie de *places*, abritant des *processus* du type π–calcul et des canaux. Les processus s'exécutent en parallèle au sein de *transactions* contrôlant les dépendances causales entre exécutions. Chaque place contient un *journal* où sont archivées les actions effectuées localement: ces informations seront utilisées pour confirmer la validation d'une transaction ou défaire ses effets (messages envoyés et reçus) en cas d'annulation. L'échange de messages entre transactions au sein d'une place crée un lien causal entre la transaction qui émet le message et celle qui le reçoit. Une place joue alors le rôle de domaine de synchronisation. La navigation des messages à travers les frontières de domaines, s'effectue par routage explicite à l'aide de capacités pour entrer ou sortir d'un domaine, primitives similaires à celles du calcul des ambients.

Calcul	**Dπ**	$\pi_{1\ell}$	**join**	**seal**
Principales abstractions	(Système) Localité Thread	(Messages) Localité Processus	Localité Processus Définition Join-pattern	Seal Processus Portail
Primitive de domaine	Localité	Localité	Localité	Seal
Sémantique de domaine	Exécution Communication Sécurité Référentiel de migration	Exécution Défaillances Référentiel de migration	Exécution Défaillances Unité de migration	Exécution Sécurité Unité de migration
Organisation des domaines	Plate	Plate	Hiérarchique	Hiérarchique
Communication	Locale Synchrone	Distante Asynchrone	Locale et distante Asynchrone	Locale et distante Synchrone
Routage	Explicite	Implicite	Implicite	Explicite
Comm. distante primitive	Non	Oui	Oui	Oui
Unité de migration	Thread	Processus	Localité	Seal
Migration	Objective	Objective	Subjective	Objective

FIG. *2.6* – **Principales caractéristiques de quelques variantes du π–calcul réparti.**

2.3.5 Formes du calcul des ambients

Contrairement aux calculs précédents, variantes du π–calcul asynchrone avec localités qui ont étudié diverses formes de communication à base de canaux, le calcul des ambients [37, 39, 40] (MOBILE AMBIENTS) considère la mobilité comme concept primitif, toute exécution étant exprimable en termes de primitives de migration. Le but poursuivi est de fournir un cadre théorique formel pour l'étude de la programmation mobile dans les réseaux à large échelle décrits comme partitionnés en hiérarchies de domaines administratifs. De part sa simplicité, son élégance et son pouvoir expressif, le calcul des ambients a connu une diffusion rapide et a réussi à s'imposer comme modèle de référence pour le « global computing » [29].

La mobilité y est vue comme le franchissement de frontières de domaines, la sécurité étant assurée par des autorisations de se déplacer entre domaines. Les domaines sont représentés par la notion d'*ambient* ou unité nommée de migration contenant des *processus* et d'autres ambients. Les opérations possibles d'un ambient – pénétrer dans un ambient, en sortir, ou en dissoudre la frontière – sont contrôlées à l'aide de *capacités* (`in`,`out`,`open`), comme le montre la figure 2.7. Les communications sont limitées à des interactions asynchrones purement locales. L'interaction distante est obtenue par combinaison de mobilité et de communication locale, le routage des messages étant explicite. Un ambient, endroit mobile où se produit un calcul et délimité par une frontière qui l'isole de l'environnement externe, est donc à la fois un domaine d'exécution, de communication, de migration et de sécurité. Le contrôle sur le domaine se manifeste par des instructions subjectives de migration (`in`,`out`) ou de destruction de la frontière du domaine (`open`). Ces opérations nécessitent la connaissance du nom de l'ambient, véritable mot de passe qui donne accès à un contrôle sans limites sur l'ambient et dont la diffusion est protégée par la portée lexicale.

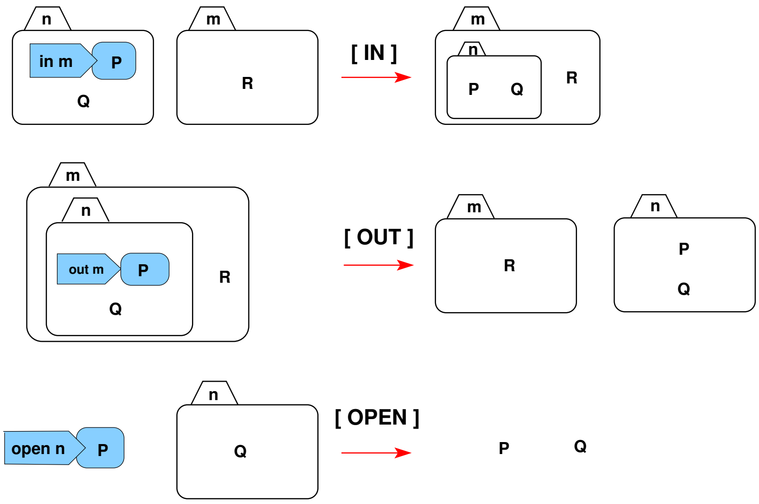


FIG. 2.7 – **Primitives du calcul des ambients (figure adaptée de [37]).**

Les SAFE AMBIENTS [106] étendent le calcul en introduisant des coactions ($\overline{\texttt{in}}$,$\overline{\texttt{out}}$,$\overline{\texttt{open}}$) pour éviter des formes graves d'interférence considérées comme des erreurs de programmation: chaque opération de contrôle sur un ambient (`in`, `out`, `open`) devient à présent une poignée de main entre ambient contrôleur et contrôlé, car ce dernier doit signifier son accord en émettant la coaction correspondante. La théorie algébrique du calcul s'en trouve simplifiée.

Les BOXED AMBIENTS [34] abandonnent la capacité `open` des ambients, jugée trop dangereuse, et réintroduisent une forme limitée de communication distante entre ambients, comme dans le seal–calcul: pour le franchissement d'une barrière, seules sont admises les interactions entre ambients ayant un lien direct de parenté. L'interprétation d'un ambient comme domaine de sécurité s'en trouve renforcée, car une communication distante devient une requête d'accès à un canal de communication situé de l'autre côté de la frontière de domaine.

2.3.6 Combinaison des algèbres de processus avec d'autres paradigmes de programmation

DITYCO

TYCO (TYped Concurrent Objects) [168, 170] est un calcul cherchant à formaliser les interactions asynchrones entre objets concurrents. Les principales abstractions introduites sont les *messages* et les *objets*, l'invocation de méthode étant modélisée par application d'un message sur un objet. Les classes sont représentées à l'aide de *déclarations de « templates »* ou spécifications paramétrées de processus dont l'instanciation fournit des objets. Une forme de domaine est présente dans DITYCO (DIstributed TyCO) [169] qui introduit répartition et mobilité dans TYCO à l'aide de la notion de *site* qui héberge un certain nombre de *processus*. Un *réseau* y représente une organisation plate de sites. Les canaux de communication sont localisés dans les sites qui jouent le rôle de domaines de désignation: la syntaxe du calcul distingue explicitement noms locaux et noms distants, et des primitives spécifiques (`export`, `import`) sont destinées à rendre visible un nom local au niveau d'un réseau ou inversement à résoudre un nom réseau en un nom local. Une discipline de portée lexicale permet à tout moment de conserver une trace du site de création du nom. L'interaction distante est codée à l'aide de primitives de migration de messages et de communication locale, seule véritable forme de communication (sémantique « move-and-communicate »). Un site est donc également un domaine de communication. Un site sert aussi de référentiel de migration pour les processus: grâce à la portée lexicale, un processus voulant communiquer sur un canal distant migrera implicitement vers le site auquel appartient ce canal. Les fondements théoriques de DITYCO sont étudiés plus en détail dans [144].

KLAIM

KLAIM (Kernel Language for Agents Interaction and Mobility) [25, 49, 50, 51] est un cœur de langage visant à coordonner un ensemble de processus partageant des ressources dans un contexte réparti. KLAIM est une fusion entre un calcul de processus d'ordre supérieur et un langage de coordination à la LINDA [65], avec des primitives pour la distribution, la mobilité et la sécurité. LINDA modélise la communication par mémoire partagée à l'aide d'un *espace de tuples*. Les champs des tuples contiennent des variables ou des valeurs. La communication repose sur un schéma producteur-consommateur synchronisant les opérations de dépôt (`out`) et de lecture-retrait (`read`, `in`) de tuples de l'espace de tuples. La sélection d'un tuple s'effectue par filtrage de motifs (*pattern-matching*) et attribue des valeurs aux paramètres formels contenus dans une requête. KLAIM étend LINDA en distribuant les espaces de tuples. Un système KLAIM (*réseau*) fédère un certain nombre de *nœuds d'exécution*, contenant chacun un espace de tuples et des *processus* en cours d'exécution. Les opérations de base sont asynchrones et permettent de lire, retirer ou déposer un tuple dans un espace de tuples distant ($\mathtt{read}(t)@\,\ell$, $\mathtt{in}(t)@\,\ell$, $\mathtt{out}(t)@\,\ell$), de déclencher l'exécution à distance d'un processus ($\mathtt{eval}(P)@\,\ell$) ou de créer un nouveau noeud d'exécution ($\mathtt{newloc}(u)$). Les domaines de KLAIM sont les nœuds d'exécution: ils permettent de synchroniser les processus et de contrôler l'accès aux ressources contenues dans un espace de tuples, en limitant communication et migration selon une politique de sécurité propre au domaine. L'organisation des domaines est plate, mais des travaux récents [19] ont étudié une forme hiérarchique de structuration des domaines où la connectivité réseau devient dynamique. La communication locale se fait comme dans LINDA, de manière anonyme et asynchrone. Lors d'une interaction distante, le transfert des tuples entre espaces de tuples est implicite. L'ordre supérieur permet la migration de processus (`eval`,`out`), le contexte d'exécution du processus migrant étant soit conservé statiquement par transmission d'une fermeture (`out`), soit lié dynamiquement dans le domaine d'arrivée (`eval`). La protection est réalisée à l'aide d'un système de types utilisant des capacités pour chacune des 5 opérations de base du calcul: cela limite la communication et la migration aux seuls processus et domaines qui détiennent ces capacités.

Calcul	**ambients**	**DiTyCO**	**Klaim**
Principales abstractions	Ambient Processus Capacité	(Réseau) Site Processus Objet Décl. de template	Réseau Nœud d'exécution Espace de tuples Processus Tuple
Primitive de domaine	Ambient	Site	Nœud d'exécution
Sémantique de domaine	Exécution Communication Sécurité Unité de migration	Exécution Communication Désignation Référentiel de migration	Exécution Synchronisation Sécurité Référentiel de migration
Organisation des domaines	Hiérarchique	Plate	Plate [50] Hiérarchique [19]
Communication	Locale Asynchrone	Locale Asynchrone	Locale et distante Asynchrone
Routage	Explicite	Implicite	Implicite [50] Explicite [19]
Comm. distante primitive	Non	Non	Oui
Unité de migration	Ambient	Processus	Processus
Migration	Subjective	Subjective	Objective

FIG. *2.8* – **Principales caractéristiques de quelques calculs pour la programmation répartie mobile.**

2.3.7 Autres modèles et langages

Parmi les autres modèles ou langages incorporant une notion de domaine, il est important de mentionner ODP et OBLIQ. Il ne s'agit pas de calculs de processus mais de modèles dont le niveau d'abstraction est différent, soit au niveau d'un « modèle de traitement » (ODP), soit comme véritable langage de programmation (OBLIQ).

ODP

Face aux besoins croissants de flexibilité et d'adaptabilité des applications réparties, le modèle de référence ODP (Open Distributed Processing) [23, 81, 82, 83, 156] constitue une réponse en terme de normalisation pour maîtriser l'hétérogénéité et la complexité des systèmes répartis ouverts. ODP constitue un cadre architectural générique et basé-objets pour la programmation répartie visant à s'abstraire d'un langage de programmation ou d'un système particulier. L'accent est mis sur la portabilité et l'interopérabilité de systèmes, en distinguant différents degrés de transparence dans la gestion de la répartition. La spécification de systèmes est séparée en points de vue complémentaires disposant chacun d'un langage de description spécifique. Parmi ceux-ci, le *modèle de traitement* ODP qui décompose un système réparti en objets interagissant via des interfaces fortement typées et le *modèle d'ingéniérie* ODP qui décrit les mécanismes nécessaires aux interactions réparties entre objets. Ces modèles ont exercé une influence déterminante sur l'émergence de la notion de domaine comme abstraction de programmation répartie.

Un système ODP est une composition d'*objets*, unités d'abstraction et d'encapsulation disposant d'une identité unique, d'un état, d'un comportement (ensemble accessible d'états) et fournissant des services, au moyen des multiples *interfaces* de l'objet, points d'accès uniques de l'objet avec son environnement. Les interfaces permettent de séparer clairement l'utilisation d'un service de sa réalisation. Si ODP décrit peu l'évolution interne d'un objet, les interactions entre objets, asynchrones, sont clairement exprimées par le concept important de *liaison* ou chemin de communication connectant différentes interfaces (cf. section 2.5.2) et permettant la définition de structures et de sémantiques de communication flexibles.

L'apport principal d'ODP consiste à décrire un système comme un assemblage de *composants* (cf. section 2.5.3) exportant des services via leurs interfaces et interconnectés par des liaisons. Le lien avec les domaines est ici direct car la structure de ces assemblages n'est pas plate mais va subir différents types de partitions à cause de l'hétérogénéité et de la distribution. L'originalité d'ODP est de capturer par le concept de liaison les différentes formes possibles de communication entre domaines. Par exemple, comme le montre la figure 2.9, le modèle d'ingéniérie d'ODP introduit partitionnement et contrôle à différents niveaux de granularité pouvant être vus comme des domaines d'exécution: une infrastructure répartie est organisée en *noeuds d'exécution* (e.g., machine physique) sous le contrôle d'un *noyau* (e.g., système d'exploitation), abritant des *capsules* (e.g., processus système avec leur propre espace d'adressage) elles-mêmes contenant des *grappes* (clusters) d'objets (e.g., machine virtuelle langage). D'un point de vue implémentation, ODP est également important car il constitue le modèle d'exécution implicite de plates-formes à objets répartis comme CORBA (cf. section 2.5.2).

La sémantique de programmes concurrents conformes au modèle de traitement ODP a fait l'objet d'une caractérisation formelle [131] sous forme d'un système de réécriture typé et d'un algorithme de vérification de types. Ces travaux ont servi de point de départ pour l'élaboration d'un « modèle de traitement » [68, 159] ou formalisme basé-objets comprenant une forme générique de domaine capable de simuler plusieurs calculs de processus répartis ($\pi_{1\ell}$–calcul, calcul des ambients, join–calcul). Ces travaux pourraient conduire à une théorie formelle des domaines.

OBLIQ

OBLIQ [35] est un langage de programmation répartie basé objets construit autour de l'idée de transparence réseau: les ressources sont accessibles de la même manière localement ou à distance. Une ressource est toujours atteignable, même en cas de migration grâce à une portée lexicale répartie, une référence locale pouvant être transformée en référence réseau. La notion de domaine y apparaît de manière implicite à travers des *sites* qui abritent des *objets* immobiles, des *ressources* (emplacements mémoire) et des *threads*. Diverses formes de mobilité de code sont rendues possibles par la transmission de fermetures entre sites. Les sites d'OBLIQ sont donc des domaines d'exécution et des référentiels pour la migration.

2.3.8 Conclusion

On observe dans les modèles décrits précédemment une grande diversité de primitives pour représenter la communication ou la migration dans les systèmes répartis de grande taille. Les figures 2.6 et 2.8 récapitulent les principaux choix de conception de quelques uns de ces modèles.

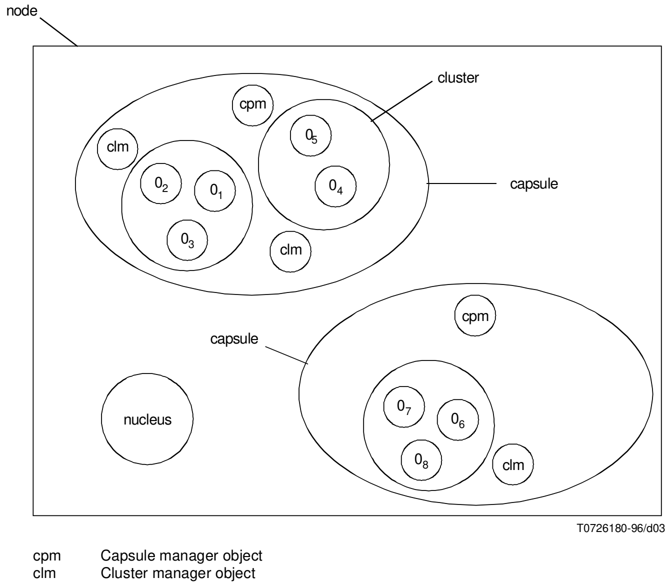


FIG. *2.9* – **Formes de domaines dans ODP: nœuds, noyaux, grappes, capsules et objets (tiré de [83])**.

Les notions de localité, d'ambient, de site, de noeud d'exécution ou de seal sont des illustrations diverses de la notion générique de domaine. Un point d'accord entre ces modèles est que ces formes de domaines sont avant tout des *domaines d'exécution*, permettant de rendre explicite le partitionnement spatial d'une exécution et par là même de modéliser la répartition: un domaine y est vu comme un site abritant un certain nombre d'entités et de ressources sur lesquelles il exerce un contrôle. Il s'agit de la forme la plus élémentaire de domaine. Dans la section 2.1.2, un domaine a été défini comme une localité contrôlée ou unité simultanée de composition et de contrôle. Malheureusement, le point commun entre ces modèles se limite à la composition spatiale: les calculs diffèrent largement sur la forme précise que prend le contrôle. Ainsi, on observe une gamme variée de sémantiques de domaines selon le modèle de programmation considéré (cf. figure 2.10):

Calcul	**Dπ**	$\pi_{1\ell}$	**join**	**seal**	**ambients**	**DITYCO**	**KLAIM**
Exécution	$\star\star$	$\star\star$	$\star\star$	$\star\star$	$\star\star$	$\star\star$	$\star\star$
Communication	$\star\star\star$				$\star\star\star$	$\star\star\star$	$\star\star$ (Synchronisation)
Migration R: Référentiel U: Unité	$\star$ (R)	$\star$ (R)	$\star\star\star$ (U)	$\star\star\star$ (U)	$\star\star\star$ (U)	$\star$ (R)	$\star$ (R)
Sécurité	$\star\star\star$			$\star\star\star$	$\star\star\star$		$\star\star\star$
Défaillances		$\star\star\star$	$\star\star\star$				

Les étoiles mesurent le degré de couverture de chaque sémantique de domaine pour les différents modèles de programmation.

FIG. *2.10* – **Sémantiques de domaines.**

- *Désignation:* le contrôle sur les ressources commence par une politique d'attribution et d'utilisation des noms, par exemple pour désigner un domaine comme cible d'une communication ou d'une migration. Ainsi, leur utilisation peut s'étendre à l'ensemble du système ($\pi_{1\ell}$, join) ou être restreinte à des parents proches dans une arborescence de domaines (seal). La liaison entre un nom et une ressource peut être statiquement définie (π–calcul) ou dépendre dynamiquement du domaine d'utilisation (KLAIM). Enfin, un même nom peut désigner une ressource unique ($\pi_{1\ell}$, join) ou un ensemble de domaines (ambients).

- *Communication:* les domaines sont aussi des unités de communication, l'interaction étant avant tout locale entre entités membres du domaine. La sémantique la plus stricte exlut toute autre forme de communication directe avec l'extérieur du domaine : la communication distante (inter-domaines) est alors codée à l'aide de primitives de migration (ambients, Dπ), les messages franchissant les frontières de domaine grâce à des chemins spécifiés explicitement. D'autres sémantiques plus souples consistent à restreindre l'interaction distante à des domaines voisins (seal, BOXED AMBIENTS), ou à autoriser un mode de communication complètement transparent, le routage des message entre domaines étant alors implicite (join, $\pi_{1\ell}$). Conformément à l'exigence `REQ.ASYNCHRONY`, la communication est asynchrone, sauf cas particulier (seal) où des questions de sécurité demandent une synchronisation plus forte entre partenaires de communication. La communication est en général à base d'envoi de messages sur des canaux, une synchronisation sur des domaines de ressources partagés étant également possible (KLAIM).
- *Migration:* les domaines apparaissent également comme cadre de référence pour la mobilité: les entités migrantes sont soit de simples processus (Dπ, KLAIM), soit de véritables domaines (join, ambients, seal). Cette distinction se reflète dans l'organisation des domaines, qui est en général plate dans le premier cas, et hiérarchique dans le second où un domaine se déplace avec l'ensemble de ses sous-domaines, ce qui conduit à la reconfiguration de l'arborescence de domaines. L'ordre de migration provient de l'entité migrante (ambients) ou de son environnement (KLAIM) [28].
- *Sécurité:* les membres d'un domaine de sécurité partagent les mêmes mécanismes et politiques d'accès aux ressources (KLAIM, Dπ). Dans la plupart des calculs, la portée lexicale offre une protection minimale en limitant la diffusion des noms de domaines. D'autres modèles vont plus loin en introduisant de véritables domaines de sécurité qui contrôlent la communication et la migration, où l'accès à un domaine (ambients) ou à ses ressources (seal, KLAIM) est réglementé par des capacités.
- *Défaillances:* un domaine est enfin une unité atomique de défaillance qui regroupe des entités ayant le même comportement en cas de panne: absence d'émission ou de réception de messages, et d'accueil de processus ou de domaines en cours de migration. Le mode de défaillance peut être irréversible ($\pi_{1\ell}$, join) ou présenter une possibilité de reprise grâce à un mécanisme transactionnel (ATF–calcul). Une défaillance d'un domaine est également détectable depuis d'autres domaines ($\pi_{1\ell}$, join).

Ces multiples comportements de contrôle d'une localité sont explorés à travers diverses formes de domaine, notamment pour la tolérance aux défaillance et la sécurité afin de mesurer leur impact sur la communication et la migration. La plupart des modèles de programmation répartie comportent donc une forme de domaine qu'ils ont identifié comme concept important de programmation, en particulier à travers la notion de localité. Cependant, les domaines y restent généralement implicites ou semi-explicites. De plus, ces modèles adoptent une perspective homogène des domaines, un domaine n'étant capable que d'un seul type de comportement selon l'orientation particulière du modèle choisi, par exemple la mobilité pour le calcul des ambients, ou les défaillances pour le join–calcul. Or, un système réparti à grande échelle se caractérise par la coexistence de nombreux types de domaines de sémantiques variées. Ces modèles sont donc insuffisants pour capturer pleinement le « global computing ».

Un tel modèle devrait permettre la prise en compte de comportements différents de domaines au sein du même modèle, par exemple pour spécifier une politique de contrôle d'accès et un comportement de défaillances pour une même localité. Capturer ainsi différentes facettes d'une exécution répartie requiert donc un modèle de programmation avec une forme programmable de domaine.

L'état de l'art précédent suggère que ce modèle devrait également:

- Permettre l'exécution en parallèle de code et l'échange de données et de code sur le réseau.
- Obéir à un principe de localité pour qu'un programme ait une notion de l'endroit où il s'exécute.
- Réaliser la communication à l'aide de canaux abstrayant le médium de communication, sur lesquels s'échangent des capacités cap à communiquer et à migrer, par exemple des noms, pour capturer la dynamicité d'une topologie de communication.
- Disposer d'une sémantique de communication synchrone ou asynchrone selon les besoins, si possible paramétrique par rapport au modèle, et sans interférence avec la sémantique de domaine.
- Exprimer la mobilité comme franchissement de frontière de domaine.
- Disposer d'une topologie hiérarchique de domaines $a[P]$, plus naturelle pour exprimer la mobilité et l'imbrication de domaines de sécurité.
- Contrôler finement la migration et la communication en s'inspirant du calcul des ambients, qui constitue une première tentative de théorie formelle de la sécurité des domaines.

Un cœur de syntaxe répondant à certaines des exigences précédentes pourrait être [28]:

$$P, Q \ldots ::= (P \mid Q) \mid a[P] \mid \mathsf{cap}.P \ldots$$

L'isolation du concept de domaine comme dénominateur commun entre ces modèles constitue une avancée importante par rapport à l'état de l'art car elle permet de rendre explicite les nombreuses formes de partionnement existant dans un système à large échelle. Elle ne demande qu'à être étudiée plus en détail au niveau théorique, et à être explicitée comme abstraction fondamentale au niveau d'un modèle de programmation (cf. chapitre 4) disposant d'une forme programmable de domaine.

Pour soutenir le cheminement de bout en bout exposé dans la problématique depuis un modèle de programmation jusqu'à son implémentation, la notion de domaine doit également se manifester au niveau d'une plate-forme d'exécution, pour réaliser les primitives du modèle et structurer leur implémentation. C'est ce que nous allons examiner dans les deux sections suivantes qui proposent un rapide tour d'horizon des manifestations de domaines dans les machines virtuelles qui implémentent les modèles décrits dans cette section (cf. section 2.4) et plus généralement au niveau des middleware et plates-formes d'exécution réparties (cf. section 2.5).

2.4 Machines virtuelles concurrentes et réparties

Parmi les multiples calculs de processus existants (cf. section 2.3), ceux ayant fait l'objet d'une implémentation sont peu nombreux. Nous explorons ici les manifestations de domaines dans les machines virtuelles qui leur servent de support à l'exécution, en adoptant la même classification que pour les calculs. Un état de l'art plus détaillé est disponible dans [24].

2.4.1 Machines virtuelles pour π-calculs asynchrones

PICT et NOMADIC PICT

PICT [140, 141, 142, 164] est un langage de programmation concurrente, de style fonctionnel, implémenté uniquement en terme de primitives du π–calcul: le cœur du langage nommé CORE PICT est un π–calcul asynchrone statiquement typé. Les domaines sont absents du langage ou des structures d'exécution du run-time PICT car répartition et mobilité ne sont ne pas prises en compte.

Cependant, PICT démontre que le π–calcul est utilisable comme cœur d'un langage de programmation implémentable efficacement: la structure compacte du calcul permet une représentation optimisée des canaux de communication et un schéma direct de compilation des primitives CORE PICT vers le langage C pour ensuite être compilé en code natif et se lier au run-time qui gère l'ordonnancement et la création de nouveaux processus. De plus, PICT fournit un premier exemple de formalisation de bout en bout d'un système puisque la structure du run-time est spécifiée formellement par une machine abstraite dont une forme de correction est prouvée par rapport au calcul sous-jacent.

NOMADIC PICT [155, 175, 177, 178] étend PICT pour la programmation et l'exécution d'agents mobiles. Son modèle de programmation, un calcul de processus appelé NOMADIC–π, distingue deux types de primitives de communication entre agents: celles de bas niveau nécessitent de connaître explicitement la localisation d'un agent pour communiquer avec lui; celles de haut niveau sont indépendantes de la localisation de l'agent ou de ses migrations éventuelles. Une infrastructure répartie est exprimée comme une traduction formelle des primitives de haut-niveau vers celles de bas niveau ce qui permet la vérification rigoureuse de telles infrastructures: une preuve de correction d'un exemple d'infrastructure codée en NOMADIC PICT est décrite dans [165, 166].

Les domaines se manifestent dans NOMADIC PICT sous forme d'unités de communication, les *sites*, organisés de manière plate et hébergeant des *agents* ou unités de migration qui se déplacent de site en site. Un agent contient des *processus* en cours d'exécution qui communiquent entre eux par primitives de type π–calcul. L'interaction est uniquement possible localement entre processus situés à l'intérieur d'un même agent, ou entre agents situés sur le même site. Les domaines n'apparaissent pas au niveau langage et peuvent seulement être désignés comme cible d'une migration, par exemple pour implémenter de la communication transparente à la localisation du destinataire, qui est une primitive de haut-niveau de NOMADIC PICT.

L'implémentation de NOMADIC PICT [176], écrite en OCAML prend la forme d'un compilateur et d'un run-time. Le compilateur, extension de celui de PICT, prend en entrée un programme de haut-niveau et un algorithme de raffinement et produit la traduction du programme en primitives de bas niveau avant transformation en code intermédiaire qui correspond sensiblement aux primitives du calcul de bas-niveau (émission, réception, composition parallèle, création et migration d'agents). Indépendant d'une architecture, ce code sera exécuté par le run-time. Ce dernier comprend une machine virtuelle et des serveurs d'entrées-sorties. Un domaine de migration NOMADIC PICT est implémenté dans la machine virtuelle par une *fermeture d'agent* comprenant une file d'exécution de processus prêts à être exécutés, des files de messages pour les communications internes, ou inter-agents. La machine virtuelle comprend également une file globale d'exécution d'agents qui assure un ordonnancement équitable des agents et une zone tampon contenant les fermetures d'agents dont l'exécution est bloquée. Les serveurs d'entrées-sorties gèrent la migration des fermetures d'agents entre machines virtuelles par transmission de leur code sur des liaisons TCP.

Par rapport à notre problématique, l'intérêt de NOMADIC PICT est de souligner le fossé entre les modèles d'exécution répartie où les primitives de communication et de migration sont généralement indépendantes de la localisation des ressources et les infrastructures réparties actuellement disponibles qui offrent des interfaces de bas-niveau où la localisation de la cible – de communication, de migration – doit être explicitement fournie. L'idée d'introduire un raffinement des primitives de haut-niveau vers celles de bas niveau est similaire à notre approche. Cependant, la solution consistant à introduire un encodage explicite pour spécifier l'infrastructure, même si elle est assez simple de mise en oeuvre et permet d'expérimenter différents schémas de communication et de migration [175] reste complexe à étudier formellement, comme en témoigne la preuve de correction présentée dans [165].

JCL et JOCAML

Le join–calcul a fait l'objet de deux implémentations réparties de référence aux structures d'exécution similaires: JCL [62, 99] est un run-time permettant l'exécution de programmes écrits en join–calcul pur. JOCAML [44, 96, 97, 98] étend le langage de programmation OBJECTIVE CAML [105] avec les primitives du join–calcul réparti: le résultat est un langage complet, statiquement typé, permettant de programmer en style fonctionnel, impératif, ou orienté-objets, et de réaliser des plates-formes d'agents mobiles. Le cœur du langage JOCAML est le λjoin–calcul [98], fusion du join–calcul et du λ–calcul avec appel par valeur. Les abstractions principales sont les *expressions* héritées d'OCAML et les *processus* du join–calcul. Les expressions sont évaluées par envoi de message sur des canaux synchrones qui retournent des valeurs (constantes, noms de canaux et de localités, définitions, threads et fermetures) tandis que les processus communiquent et se synchronisent sur des canaux asynchrones. Les processus du join–calcul sont compilés en expressions qui constituent le byte-code exécutable par la machine virtuelle JOCAML. Le byte-code est d'assez bas niveau avec des opérations comme la création ou la terminaison de threads, des constructions fonctionnelles (abstraction, application, `let`), des primitives du join–calcul (nouvelle définition ou localité, envoi de message asynchrone, migration, join-pattern, réponse synchrone): schématiquement, la machine virtuelle peut-être vue comme le couplage étroit entre un run-time pour les primitives du join–calcul et un évaluateur fonctionnel d'expressions OCAML. Elle prend la forme d'une machine à pile multithreadée, constituée du byte-code du programme à exécuter (segment de code), de la pile d'exécution du thread courant (segment de pile) et de diverses autres structures de données comme les définitions ou les localités (segment de tas). La machine virtuelle JOCAML a été spécifiée formellement sous forme d'une machine abstraite appelée JAM (JoCaml Abstract Machine) [138].

Les domaines se manifestent dans chaque machine virtuelle par des unités logiques de migration et de défaillances, les *localités*, organisées en arborescence et contenant chacune des pointeurs vers les threads et les définitions qu'elles abritent. Les threads contiennent le byte-code à exécuter et sont manipulables à l'aide de registres dédiés. Leur ordonnancement est géré par une file d'exécution unique. A l'aide des join-patterns, les définitions permettent la création simultanée de plusieurs canaux et commandent la synchronisation des messages stockés dans des files d'attentes.

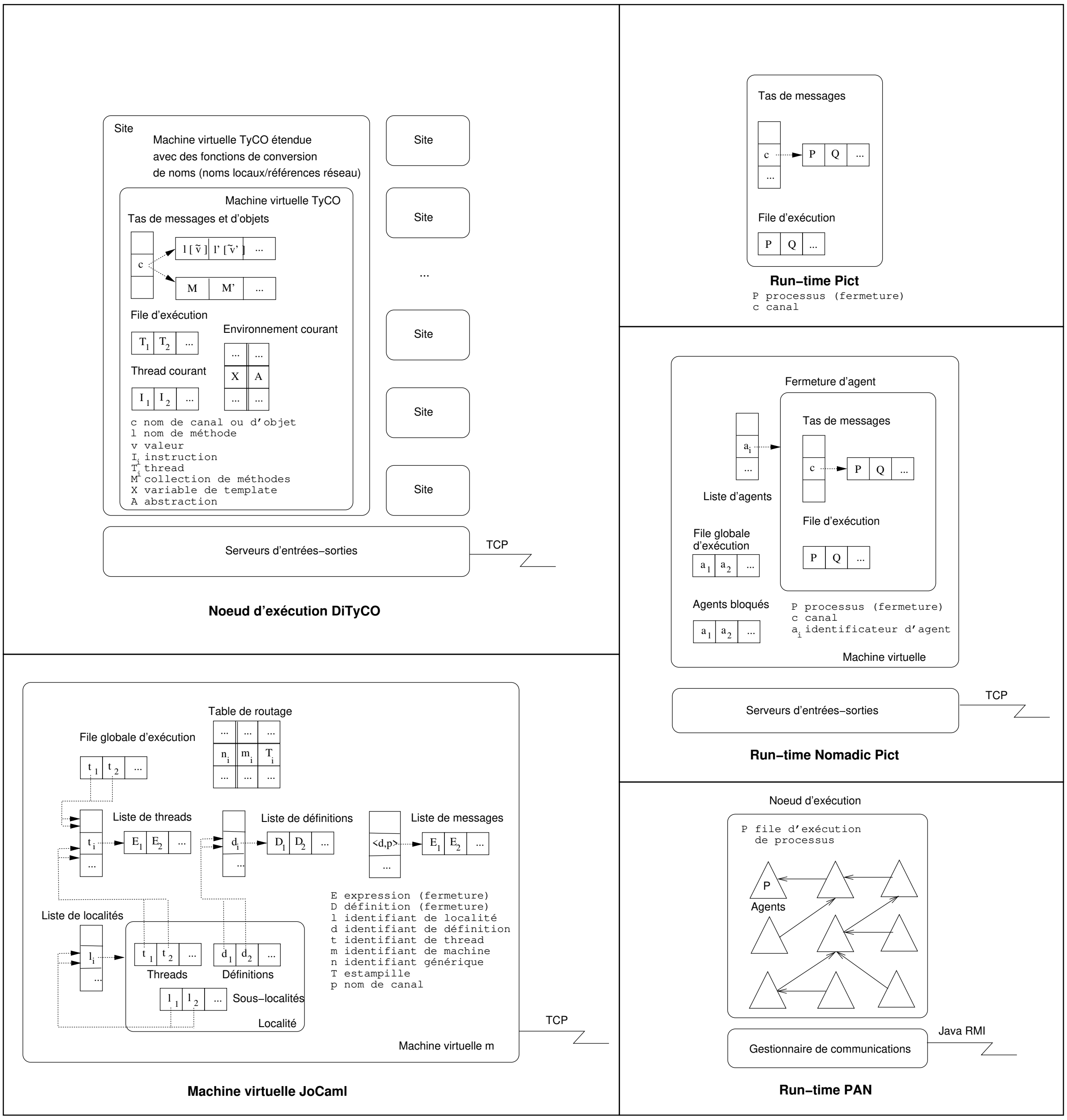


FIG. *2.11* – **Vue schématique des structures d'exécution de quelques machines virtuelles.**

Une machine virtuelle est un domaine de désignation. Chaque machine dispose d'un nom unique et conserve la trace des identificateurs créés localement désignant les structures d'exécution locales de la machine: définitions, localités et, threads. Un service de nommage permet le partage de noms entre machines distantes. L'unicité des noms de machine assure une portée répartie des identificateurs et facilite la migration des localités. Une machine est aussi un référentiel pour la migration. Elle dispose localement d'une table de routage, mise à jour lors de chaque migration de localité, qui indique le nom de la machine, éventuellement distante, contenant une structure d'exécution donnée. Des liens de poursuite laissés par la localité migrante sur la machine de départ permettent d'accéder aux ressources de la localité indépendamment de sa migration. La transmission de définitions et de threads autorisent l'implémentation de différents schémas de migration comme l'évaluation à distance, le téléchargement de code, ou les agents mobiles. L'accès à la pile d'exécution d'un thread permet la mobilité forte de domaines, l'exécution reprenant sur la machine d'arivée exactement au point où elle s'était interrompue sur la machine de départ.

Dans JOCAML, les domaines communiquent par envoi de message asynchrone provoquant la mise à jour des files d'attente à l'intérieur d'une même machine, ou l'utilisation de liens TCP entre différentes machines. Grâce à la co-localisation des récepteurs, la synchronisation entre messages est implémentée localement sans recours à un consensus réparti coûteux.

Dans l'optique de la programmation répartie fiable, le join–calcul est un modèle marquant par son souci d'être implantable: en mettant l'accent sur la contrainte de localité, le join–calcul dépasse une des faiblesses du π–calcul asynchrone, trop coûteux à implémenter dans un cadre réparti. La formalisation de JOCAML sous-forme de machine abstraite constitue un exemple intéressant de formalisation de bout en bout qu'il sera fructueux de comparer avec l'approche adoptée ici (cf. chapitre 7). D'autre part, la fusion du join–calcul avec un langage complet souligne l'intérêt de l'utilisation des méthodes formelles en pratique pour la programmation répartie. En particulier, l'intégration d'un noyau fonctionnel dans un calcul de processus au niveau de la machine abstraite JAM et de son implémentation fournissent un guide précieux pour implémenter un modèle de programmation orienté domaines incluant le λ–calcul (cf. chapitre 4). Cependant, la sémantique de domaine du join–calcul manque de généricité (cf. section 2.3.8), le protocole d'interaction avec une localité étant peu adaptable. Pour disposer d'une implémentation centrée-domaines adaptée à une programmation de type « global computing », il semble nécessaire d'aller au delà de JOCAML afin d'implémenter des formes plus flexibles de domaines.

2.4.2 Implémentations du calcul des ambients

Trois implémentations principales du calcul des ambients ont été réalisées, deux seulement étant réparties.

AMBIT

Cardelli a d'abord produit une première implémentation centralisée appelée AMBIT [36], constituée d'un interprète d'expressions du calcul écrit en JAVA. La synchronisation entre ambients affectés par un mouvement était gérée à l'aide de verrous tirant partie de la structure hiérarchique des ambients. Cette solution n'était pas généralisable au cas réparti, l'attente de libération d'un verrou distant pouvant bloquer la poursuite d'une exécution locale.

Ambients et join–calcul

Fournet, Lévy et Schmitt ont ensuite réalisé une implémentation répartie en JoCaml [61] en traduisant formellement le calcul des ambients en join–calcul, en prouvant la correction de cette traduction, puis en la raffinant par des interprètes d'expressions communiquant par envoi asynchrone de messages. L'obstacle recontré dans l'implémentation des primitives de mobilité du calcul était la gestion nécessaire de la synchronisation répartie entre les ambients participant à la migration, situés à des niveaux hiérarchiques différents. La solution proposée décomposait la migration en étapes atomiques ne nécessitant que des synchronisations locales. Cette implémentation met en évidence une forme de domaine de synchronisation: un *site* S contient un gestionnaire dédié qui implante le protocole de synchronisation et contrôle donc la migration de tous les ambients hébergés sur S.

Pan

Sangiorgi et Valente ont enfin formalisé une implémentation répartie en Java RMI du calcul des Safe Ambients sous forme d'une machine abstraite appelée Pan [149], prouvée correcte par rapport au calcul. L'utilisation des Safe Ambients au lieu du calcul des ambients d'origine, en éliminant certaines formes graves d'interférence entre opérations de migration, simplifie l'implémentation en réduisant la complexité du protocole de synchronisation par l'utilisation des coactions. La preuve de correction s'en trouve allégée. Pan sépare clairement la distribution physique de la structure logique d'un système d'ambients. Ainsi, les primitives `in` et `out` modifient seulement les relations hiérarchiques entre ambients sans changer leur localisation physique; au contraire, la capacité `open` s'accompagne d'une relocalisation physique de certains ambients. Cette séparation est source de simplifications supplémentaires lors de la simulation des primitives de migration du calcul. Ainsi, le contrôle de la migration d'un ambient est concentré au niveau de l'ambient englobant qui accepte ou rejette une requête de migration d'un de ses enfants et, en cas d'accord, lui retransmet un ordre de relocalisation. Cependant, la sémantique d'origine des ambients, qui représentait initialement le partitionnement spatial des ressources, est perdue.

L'implémentation manifeste deux types distincts de domaines: les *nœuds d'exécution* sont des domaines administratifs qui capturent la distribution physique. Il s'agit d'interprètes d'expressions du calcul, uniquement identifiés par leur adresse IP. Les nœuds hébergent des domaines d'exécution ou *agents*, threads Java qui représentent des ambients localisés. Les domaines de migration se manifestent à la fois en terme d'entités migrantes (les agents), et de reférentiels pour la migration (les nœuds d'exécution). Si l'organisation des noeuds est plate, celle des agents est arborescente et peut recouvrir plusieurs noeuds différents: chaque agent maintient un lien vers son parent qui réside éventuellement sur un noeud distant. Les agents communiquent entre nœuds d'exécution par envoi de message asynchrone en utilisant Java RMI. La migration d'un agent s'effectue par sérialisation de son contenu vers un nœud distant.

Implémentabilité du calcul des ambients

Les différentes déclinaisons du calcul des ambients constituent un modèle de programmation séduisant car décrivant un modèle minimal de localités hiérarchiques réifiant la notion de barrière et permettant de contrôler finement la migration et la communication. L'avantage des implémentations décrites ci-dessus est de présenter par rapport à leur calcul d'origine des preuves de correction rigoureuses. Leur principal défaut est de nécessiter des schémas sophistiqués de synchronisation répartie entre ambients, coûteux à mettre en oeuvre pour implémenter les primitives de mobilité. Cette difficulté importante d'implémentation des variantes du calcul des ambients se double d'un manque au niveau langage: jusqu'ici, aucun langage de haut niveau n'a été développé pour programmer selon le paradigme des ambients.

2.4.3 Machines virtuelles pour calculs à objets et à espaces de tuples

DiTyCO

TyCO et sa version répartie DiTyCO, langages de programmation concurrente basés-objets et fortement typés, ont été implémentés sous forme de machines virtuelles [111, 112, 113, 115], celle de DiTyCO étendant le run-time TyCO avec de la communication asynchrone. Les abstractions principales sont les *processus*, *les canaux*, les *définitions* et les *objets* provenant du calcul TyCO, auxquels il faut ajouter les *sites* et les *réseaux* de DiTyCO (cf. section 2.3.6). Au niveau de l'implémentation, DiTyCO ajoute un niveau intermédiaire, le *nœud d'exécution* qui abrite plusieurs sites s'exécutant de manière concurrente.

Un programme source DiTyCO est compilé [114] dans un byte-code intermédiaire qui reflète la structure imbriquée des processus. Ce code est ensuite exécuté par une machine virtuelle TyCO étendue qui implémente les sites du calcul et constitue un domaine d'exécution élémentaire. Les sites sont également des domaines de mobilité, au sens d'un référentiel de migration pour les définitions, les objets et les messages. Plusieurs sites sont regroupés au sein d'un noeud d'exécution qui assure deux fonctions. Il s'agit d'abord d'un domaine de désignation qui effectue la résolution des noms et réalise des opérations de transformation entre identifiants locaux et réseau. Il gère ensuite la connectivité physique des sites en fournissant plusieurs services de communication. Il est uniquement identifié sur le réseau par son adresse IP. La machine virtuelle TyCO a été formalisée par une machine abstraite [116, 117] et prouvée correcte par rapport au calcul. Il n'y a pas de résultat comparable pour la version répartie.

La machine virtuelle est la principale manifestation de domaine dans DiTyCO. A un niveau d'abstraction intermédiaire, elle assure l'exécution de *threads* ou séquence ininterruptible d'instructions constituées des processus du calcul. Ses structures d'exécution principales sont un tas, contenant des files d'objets et de messages en attente de réduction, et une file d'exécution qui sérialise l'exécution des différents threads. Un environnement global conserve les déclarations de « templates » (définitions) couramment actives.

Cette vision mono-threadée de la machine, peut être raffinée au niveau système par un pool de threads qui contiennent chacun du byte-code à exécuter. A ce niveau de description, la machine est composée d'une zone de programmes où est conservé le byte-code, d'un tas constitué de blocs contigus (« frames ») alloués dynamiquement et contenant les objets, messages, et définitions sous forme de fermetures. Enfin, une file d'exécution assure l'ordonnancement des différents threads. L'ensemble de ces structures est manipulable à l'aide de registres dédiés.

Les sites sont non seulement des domaines d'exécution mais aussi de communication, au sens où les réductions sur les objets et les messages sont purement locales à un site. La communication entre sites distants est effectuée par les nœuds d'exécution qui assurent la migration des messages, définitions et objets en émettant et recevant sur le réseau les fermetures correspondantes, i.e., le byte-code, sur des liaisons TCP/IP.

X–KLAIM et KLAVA

L'implémentation de KLAIM se compose de deux éléments: un langage de haut-niveau appelé X–KLAIM (eXtended Klaim) [15] étend le noyau KLAIM avec des types primitifs, des opérations temporisées, et des constructions impératives; une bibliothèque de classes JAVA appelée KLAVA (KLaim in Java) [17, 18] est à la fois un middleware qui supporte les opérations X–KLAIM et un canevas logiciel pour programmer en JAVA dans le style de KLAIM. Les programmes KLAIM sont compilés en JAVA par le compilateur X–KLAIM, leur exécution par la machine virtuelle JAVA reposant sur les classes KLAVA. Le byte-code JAVA sert à la fois de format d'exécution et de transmission pour le code mobile. Pour le moment, cette implémentation ne semble pas avoir fait l'objet d'une preuve de correction par rapport à KLAIM.

Comme dans KLAIM, l'implémentation se compose de deux niveaux: les *processus* qui constituent des unités d'exécution et de migration élémentaires et les *nœuds d'exécution* qui assurent la coordination des processus. Ces derniers manifestent différentes sémantiques de domaines.

La fonction la plus évidente d'un nœud KLAVA est celle de domaine d'exécution et de synchronisation: il contient un espace de tuples grâce auquel il va coordonner l'exécution des processus qu'il abrite; il sert aussi d'environnement d'exécution élargi aux processus, par exemple en fournissant des services de gestion de l'espace de tuples: ajout, retrait, etc. On peut aussi le considérer comme un entrepôt de données (les tuples de l'espace de tuples) dont il assure l'accès et la protection selon une politique de sécurité. Un nœud KLAVA est donc aussi un domaine de gestion de ressources et de sécurité. Jusqu'ici, les aspects sécurité n'ont fait l'objet que d'implémentations partielles sous forme de systèmes de types ou de primitives cryptographiques pour contrôler l'accès aux tuples.

Chaque nœud est identifié de manière absolue par une adresse physique symbolique qui n'est pas nécessairement une adresse IP. Ainsi, plusieurs nœuds d'exécution peuvent être présents sur une même machine si leurs adresses physiques sont distinctes. Les ressources distantes peuvent être aussi désignées par des adresses logiques ou alias dont la portée est relative au nœud courant: un nœud maintient un environnement d'allocation reliant adresses logiques et physiques, ce qui permet la résolution des noms de ressources lors de l'évaluation d'un processus. Un nœud KLAVA est donc également un domaine de désignation.

Un réseau KLAIM est implémenté par une collection de nœuds d'exécution KLAVA, coordonnés par un *serveur de réseau*: c'est un serveur de noms qui conserve la trace de l'ensemble des adresses physiques déjà utilisées par les nœuds du réseau, et en attribue de nouvelles; c'est aussi un gestionnaire de communications qui assure la délivrance des messages entre nœuds du réseau. Un réseau KLAIM a donc une sémantique de domaine de désignation et de communication.

La communication directe entre nœuds d'exécution est aussi possible et s'effectue par la sérialisation de processus KLAVA sur des liaisons TCP. La migration d'un processus KLAIM est réalisée en déterminant l'ensemble des classes JAVA nécessaires pour son exécution, et en les transférant avec le processus. Un nœud KLAVA est également un domaine de migration: il contient un chargeur de classes spécialisé qui récupère toutes les classes transportées par un processus migrant et s'assure de leur chargement à la demande lors l'exécution ultérieure du processus. KLAVA supporte uniquement de la mobilité faible, puisque la pile d'exécution d'un thread n'est pas accessible en JAVA. Une forme de migration forte est fournie par X–KLAIM: grâce à une précompilation du code devant migrer, l'exécution d'un thread peut reprendre au point où elle s'était interrompue avant la migration.

2.4.4 Conclusion

Machine virtuelle	JoCaml	Pan	DiTyCO	X–Klaim / Klava
Notion de domaine	Localité	Agent	Site	Nœud d'exécution
Niveaux supérieurs	Machine	Nœud d'exécution	Nœud d'exécution	Réseau
Structures d'exécution internes	Définitions Expressions Threads	Processus Thread	Définitions Objets Processus Threads	Espace de tuples Processus Threads Environnement
Organisation	Hiérarchique	Hiérarchique	Plate	Plate
Sémantique de domaine*	E Mi F	E Mi S	E Mo C	A D E Mo Co S
Sémantique des niveaux supérieurs*	A D Mo	A D Mo	A D	A D C
Désignation	Réference	Identificateur d'agent	Référence locale / réseau	Adresses logiques / physiques
Portée	Globale	Globale	Site / globale	Nœud / globale
Protocoles de communication	TCP	Java RMI Sérialisation	TCP	Sérialisation
Code intermédiaire	Byte-code dédié	Byte-code Java	Byte-code dédié	Byte-code Java
Type de mobilité **Evaluation à distance** **Téléchargement de code** **Agents mobiles**	Forte Oui Oui Oui	Faible Oui Non Oui	Faible Oui Oui Oui	Faible et forte Oui Oui Oui
Formalisation par machine abstraite	Oui	Oui	Oui	Non
Preuve de correction	Non	Oui	Oui	Non

* E: domaine d'exécution; D: domaine de désignation ; F: domaine de défaillance;
A: domaine d'administration; Mi: domaine de migration (entité migrante); Mo: domaine de migration (référentiel);
C: domaine de communication; Co: domaine de coordination/ synchronisation; S: domaine de sécurité.

FIG. *2.12* – **Comparaison de quelques machines virtuelles.**

En dépit de la diversité des modèles implémentés, on constate que toutes ces plates-formes manifestent à l'exécution une notion de domaine, qui la plupart du temps sert à capturer la structure physique ou logique du réseau (nœud d'exécution). L'effet d'une exécution répartie sera de modifier l'organisation topologique de ces domaines.

Comme pour les modèles, on observe également une grande variété de sémantiques de domaine (cf. figure 2.12). Malheureusement, ces domaines ont souvent une sémantique d'exécution figée dont ils héritent du modèle sous-jacent, et sont implémentés comme tels. Dans l'optique d'une infrastructure programmable à domaines, l'approche de NOMADIC PICT est intéressante car elle permet de concentrer la politique de migration dans la traduction formelle, rendant ainsi l'infrastructure paramétrable. Elle semble cependant prohibitive par la complexité de sa formalisation. Toutefois, l'approche générale, adoptée par l'ensemble des plates-formes, d'organiser l'infrastructure d'exécution en *agents* ou en *composants* indépendants nous paraît à retenir: elle manifeste à l'exécution les deux dimensions d'un domaine: le partitionnement, par une décomposition en unités élémentaires d'exécution, de défaillances, etc.; le contrôle, par l'encapsulation dans un agent d'une stratégie de mobilité, de sécurité, etc. Par son comportement, un agent peut ainsi superviser une partie du système et agir sur son organisation. L'aspect modulaire d'une telle infrastructure est significatif pour incarner à l'exécution le concept de « localité programmable », identifiée dans la section 2.1.2 comme prérequis important pour une infrastructure pour le « global computing ».

Dans les plates-formes examinées, on peut identifier quatre critères principaux d'analyse des manifestations de domaines :

- La topologie des domaines
- Les primitives de nommage et de liaison
- Les protocoles de communication
- La mobilité

Deux grandes alternatives se dessinent pour l'organisation des domaines: plate (DITYCO, KLAVA) ou hiérarchique (PAN, JOCAML), en utilisant généralement une structure arborescente. Si la notion de nœud d'exécution est bien présente dans toutes les implémentations, sa structure interne dépend fortement de l'implémentation. Ainsi, un nœud d'exécution peut inclure des structures de données comme des définitions (DITYCO, JOCAML), des processus (KLAVA, DITYCO, PAN, JOCAML), des canaux (DITYCO, PAN, JOCAML), des objets (DITYCO) ou des espaces de tuples (KLAVA). La structure précise de quelques nœuds d'exécution est détaillée dans la figure 2.11.

En terme de nommage, la plupart des modèles examinés appartiennent à la famille des calculs qui communiquent en échangeant des noms sur des canaux (sauf KLAIM qui utilise des espaces de tuples). Au niveau de l'implantation, ces noms sont représentés par des identificateurs dont la structure est, elle aussi, spécifique à une plate-forme donnée. La portée des références est en général globale, l'ensemble du système constituant un domaine de désignation implicite. D'autres plates-formes comme DITYCO, ou KLAIM manifestent des domaines de désignation explicites, qui distinguent références locales et réseau, ou adresses physiques et logiques. Ces domaines, fournissent de plus des services de transformation de noms, utilisés dans l'établissement d'une liaison distante.

Les protocoles de communication utilisés manifestent, eux aussi, une grande diversité. En première approximation, on peut les séparer en deux familles: des protocoles de haut niveau comme Java RMI sont bien intégrés avec la machine virtuelle JAVA et tirent parti de l'indépendance architecturale qu'elle fournit (PAN). D'autres protocoles sont plus proches des ressources matérielles, comme des implémentations de TCP/IP à base de sockets, accessibles directement en JAVA (KLAVA) ou d'autres langages de programmation permettant une manipulation plus simple des ressources systèmes comme OCAML (JCL) ou C (DITYCO, NOMADIC PICT). Les stratégies d'emballage et de déballage des paramètres utilisent des structures adaptées à un byte-code dédié (JOCAML, DITYCO) ou de la sérialisation JAVA (PAN, KLAVA).

Enfin, on peut distinguer les plates-formes basées sur JAVA qui se limitent à de la mobilité faible, et les autres systèmes qui offrent de la mobilité forte (JOCAML), car utilisant une machine virtuelle spécialisée. Les stratégies de migration sont variées, un processus pouvant migrer avec toutes les classes nécessaires pour son exécution (KLAVA) – approche utile en mode déconnecté – ou pouvant les télécharger en cours d'exécution (PAN), ce qui diminue la taille des agents migrants.

Du point de vue de la formalisation, on observe une dichotomie entre machines virtuelles spécifiées sous forme de machines abstraites et d'autres approches comme des traductions explicites de niveau langage (NOMADIC PICT, ambients en JOCAML). La plupart de ces machines sont construites « à la main », soit sans obtenir une correction vis-à-vis du modèle implémenté et dans un contexte réparti – très peu de preuves de correction sont disponibles –, soit lorsque c'est le cas, en prouvant une propriété de correction ayant un rapport ténu avec la sémantique initiale du modèle (PICT, PAN). Dans tous les cas, la correction s'avère très difficile à prouver.

On observe enfin un décalage entre les primitives de bas-niveau fournies par les plate-formes d'exécution, par exemple au niveau des protocoles de communication, et les primitives de haut-niveau des modèles que l'on cherche à implémenter. L'exigence de localité (`REQ.LOCALITY`) apparaît comme primordiale pour obtenir un modèle implémentable, comme l'illustrent l'exemple du π–calcul asynchrone réparti où un consensus distribué est nécessaire pour choisir le récepteur auquel revient la communication, et celui du calcul des ambients où des protocoles de synchronisation complexes sont un obstacle à l'implémentation des primitives de mobilité. Dans l'implémentation d'une machine virtuelle répartie à base de domaines, les primitives du modèle doivent donc être facilement traduisibles en structures d'exécution. Pour contribuer à réduire ce fossé, l'approche par raffinement sous forme d'une machine abstraite confirme ici tout son intérêt.

2.5 Plates-formes à objets répartis et intergiciels adaptables

Un *intergiciel* (« middleware ») est une couche logicielle située entre le système d'exploitation et les applications qui fournit un ensemble de services (communication, désignation, transactions, persistance, ...) pour la mise en œuvre d'applications réparties. Un middleware facilite la communication et les interactions entre processus distincts: ces derniers peuvent s'exécuter sur la même machine ou sur des machines physiquement distantes communiquant à travers un ou plusieurs réseaux. Outre la transparence à la répartition, un middleware apporte aux processus une transparence à l'hétérogénéité des systèmes d'exploitation, des protocoles de communication et du matériel sous-jacent.

Dans cette section, nous passons en revue les différentes manifestations de domaines dans plusieurs grandes familles de middleware: les plates-formes à objets répartis (cf. sections 2.5.1 et 2.5.2), à composants (cf. section 2.5.3) et d'agents mobiles (cf. section 2.5.4).

2.5.1 JAVA

JAVA [72] est un langage de programmation orienté-objets sur lequel s'appuient de nombreux middleware et plates-formes à objets répartis. Son principal avantage est la portabilité: le langage, d'abord compilé dans un code intermédiaire (le byte-code), est ensuite interprété par une machine virtuelle [110] qui assure l'indépendance de JAVA vis-à-vis d'une architecture donnée. JAVA comporte des mécanismes pour la mobilité de code (chargement dynamique de classes depuis un site distant), la sûreté (système de types et vérificateur de byte-code) et la sécurité. Son intérêt réside aussi dans les nombreuses bibliothèques de classes fournies pour la programmation concurrente et répartie (multi-threading, sockets,...) [24]. Ainsi, la sérialisation fournit un moyen simple de consigner l'état d'un objet sur un canal de communication arbitraire. De même, JAVA RMI (Remote Method Invocation) [160] permet aux objets situés dans des espaces d'adressage distincts de communiquer en intégrant un mécanisme de RPC avec l'environnement objet de JAVA. Les objets n'interagissent que par appel de méthode sur leurs interfaces. Les abstractions principales de JAVA RMI sont celles d'*objet servant* dont les méthodes peuvent être invoquées à partir d'une autre machine virtuelle, et de *serveur* ou machine virtuelle JAVA regroupant plusieurs objets servants.

JAVA manifeste plusieurs types de domaines: de manière évidente, un serveur JAVA RMI est un domaine d'exécution et de ressources. On peut distinguer ensuite plusieurs formes de domaines de désignation: au niveau langage, un paquetage JAVA constitue une unité de désignation pour les différentes classes qui le composent; au niveau système, un chargeur de classes (*classloader*) regroupe les différentes classes qu'il vient de charger: dans la machine virtuelle, ces classes sont référencées à la fois par leur nom et par celui de leur chargeur de classes. En contexte réparti, JAVA utilise comme notion de référence un *talon*, transmis entre espaces d'adressage à la place de l'objet référencé qui, lui, reste immobile. Un serveur de noms (le « registry ») permet la diffusion des références en leur associant des noms symboliques. Contrairement aux serveurs qui ont une structure naturellement plate, les paquetages et les chargeurs de classes sont organisés de manière hiérarchique.

Un chargeur de classes est également un domaine de protection. L'architecture de sécurité de JAVA a fortement évolué: dans les premières versions, le modèle de sécurité est celui du bac-à-sable (*sandbox*), forme rudimentaire de domaine de sécurité: on fait confiance au code local qui dispose d'un accès complet aux ressources système, contrairement au code téléchargé qui n'a accès qu'aux ressources contenues dans la sandbox. Une extension concerne le cas des applets signées qui peuvent uniquement accéder à l'ensemble des ressources si leur signature est reconnue. Avec la version 1.2 de JAVA [71], l'ensemble du code peut être soumis à une politique de sécurité définie par un ensemble de droits d'accès. Le code est organisé en domaines de sécurité contenant des droits d'accès identiques. Le modèle d'accès de JAVA repose sur des capacités: le code est identifié par son origine (URL) et les clés publiques associées aux clés privées utilisées pour le signer. Il détient un ensemble de permissions d'accès à des classes de ressources. Cependant, la gestion d'un domaine de sécurité reste implicite. Seule la politique de sécurité est facilement configurable à l'aide d'un outil d'administration (`PolicyTool`). Divers outils complémentaires et bibliothèques cryptographiques assurent la sécurité des communications entre domaines.

JAVA fournit deux exemples de domaines de sécurité: au niveau applicatif, chaque classe appartient à un *domaine de protection* et un seul. Les permissions sont accordées au niveau du domaine: aussi, pour contrôler l'accès à un objet, on détermine son domaine d'appartenance et on en déduit les permissions appropriées. Au niveau système, le *chargeur de classes* constitue une unité de protection supplémentaire en constituant son propre espace de nommage. Notons que le domaine système associé au chargeur de classes vide dispose de privilèges spéciaux pour accéder aux ressources physiques.

Les domaines communiquent par invocation de méthode distante: pour assurer la transparence à la répartition, le modèle de liaison de Java s'appuie sur un mécanisme à base de talons et squelettes. Un identificateur d'objet peut être exporté hors du domaine d'exécution courant à l'aide de la méthode `UnicastRemoteObject.exportObject` qui renvoie une référence réseau sous forme d'un talon: dès lors, l'objet va écouter sur un port paramétrable les requêtes venant de clients éventuels. Pour effectuer l'invocation, JAVA RMI s'appuie sur le mécanisme de sérialisation: les arguments locaux sont passés par copie; les arguments distants sont passés par référence, par transmission du talon correspondant.

Une machine virtuelle constitue enfin un domaine de référence pour la migration. JAVA RMI autorise la mobilité des noms, les références d'objet pouvant être transmises librement entre espaces d'adressage. Il supporte aussi une forme faible de mobilité de code exécutable, par sérialisation d'objets ou chargement dynamique de byte-code: les classes sérialisées sont annotées avec des éléments de localisation pour pouvoir être chargées dynamiquement par le récepteur. Les objets sont par défaut non-mobiles. Seuls des talons pour un objet distant seront transmis sur le réseau, et éventuellement téléchargés dynamiquement.

2.5.2 Courtiers d'objets et intergiciels flexibles

CORBA

Les *courtiers d'objets* ou *ORB* (Object Request Broker) constituent une catégorie importante de middleware: ils assurent la transparence des communications entre objets s'exécutant dans des processus séparés. La communication, synchrone, prend la forme d'une invocation de méthode distante. Un système de talons et de squelettes est utilisé pour implanter la communication, mais ceux-ci restent en grande partie cachés au programmeur, pour qui l'appel de méthode distante est syntaxiquement identique à un appel local.

CORBA (Common Object Request Broker Architecture) [133] normalise l'architecture d'un ORB en s'affranchissant d'un langage de programmation: des objets écrits dans des langages variés peuvent communiquer grâce à l'utilisation d'un *langage de définition d'interfaces* (IDL) qui spécifie clairement les interactions possibles entre objets et permet de générer talons et squelettes. L'IDL garantit aussi l'indépendance d'une plate-forme matérielle en normalisant l'emballage et le déballage des paramètres d'une invocation.

En plus d'un simple mécanisme de communication, CORBA offre de nombreux services (cf. figure 2.13) comme le nommage, la localisation et l'activation d'objets, la sécurité des communications, la persistance ou des transactions. Chacun de ces services présente des sémantiques variées de domaine: domaines de désignation hiérarchiques, contextes de courtage, domaines d'authentification, d'autorisation, de transactions, ou de persistance. Ces domaines se manifestent par des *contextes* qui sont propagés avec l'invocation et servent aux objets pour s'interfacer avec les services CORBA. Ils permettent, par exemple, de contrôler l'accès à un domaine de sécurité, ou d'entrer dans une transaction.

La sémantique générale d'interaction entre objets CORBA est décrite par le modèle de référence ODP. Cependant, la plate-forme a fait l'objet de peu de tentatives de formalisation poussées. Les domaines sont souvent présents de manière implicite, et leur sémantique, figée.

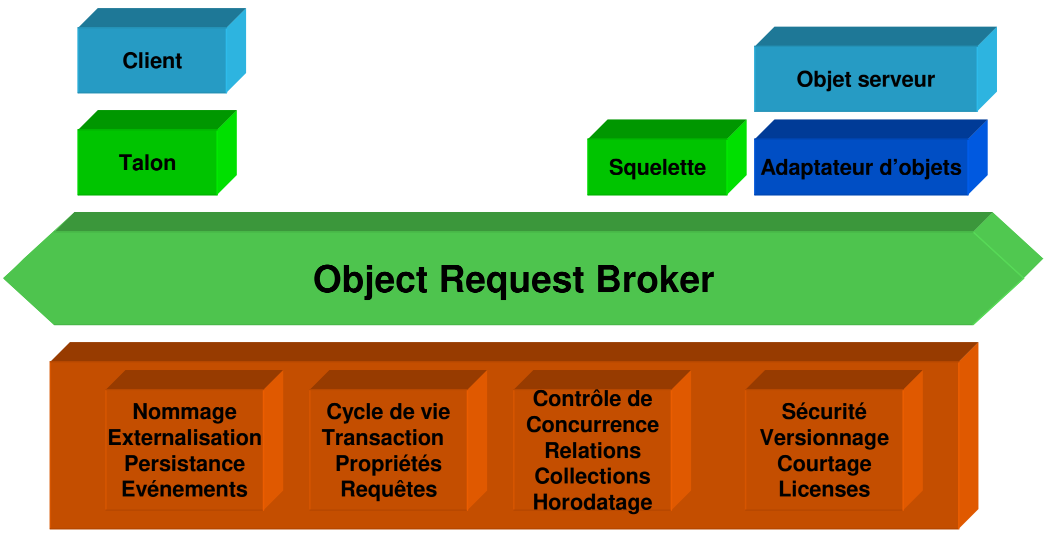


FIG. 2.13 – **Achitecture CORBA.**

Liaison flexible et intergiciels adaptables

Pour tenter de dépasser la rigidité de CORBA, un certain nombre de travaux ont été menés pour concevoir et développer des middleware adaptables. Parmi ceux-ci, on peut citer JONATHAN [55, 89, 135], OPEN ORB [21, 22, 45], OPEN CORBA [100, 101, 102] et FLEXINET [75]. Nous renvoyons le lecteur à [103] pour un état de l'art plus complet de ces plates-formes. Leur point commun est de fournir des cadres architecturaux minimaux pour la construction de middleware basés sur la notion de *liaison*, ce qui permet une flexibilité des communications. Par exemple, à partir des composants JONATHAN, il est possible de construire différents types de middleware appelés *personnalités*: la distribution JONATHAN fournit deux personnalités de type ORB, CORBA et JAVA RMI, respectivement appelées DAVID et JÉRÉMIE. D'autres personnalités de type MOM (Message-Oriented Middleware) sont aussi envisageables.

Créer une liaison signifie établir un chemin d'accès et de communication entre différentes entités d'un système réparti. Le lien ainsi créé entre les objets qu'il interconnecte peut aussi être qualifié de liaison [89]. Il est parfois réifié sous forme d'*objet de liaison*, comme dans ODP. Typiquement, il s'agit de trouver les entités à lier, vérifier les droits d'accès, initialiser des structures pour la communication, réserver des ressources, etc. Une fois la liaison établie, elle sert de contexte pour le déroulement des échanges entre participants à la liaison. On peut l'assimiler à un domaine de communications.

L'établissement d'une liaison repose sur un mécanisme de désignation. Les noms permettent à la fois d'identifier les ressources et d'être le point de départ pour la création de la liaison, en remplissant le rôle d'une capacité à communiquer. Les noms n'ont pas de signification absolue, mais existent au sein de *domaines de désignation* qui associent des noms bien formés avec des ressources et servent de référentiel à la résolution des noms. Ces domaines peuvent eux-mêmes être désignés par des noms appartenant à d'autres domaines. On a alors formation de chaînes de nommage, la résolution d'un identificateur nécessitant plusieurs étapes de navigation dans une structure hiérarchique de domaines de désignation [89]. La notion de domaine, sous forme de domaines de désignation, est donc au cœur de ces plates-formes middleware flexibles.

Différentes formes de liaison sont envisageables: une liaison langage s'effectue dans un même espace d'adressage, la désignation permettant une communication directe. Une liaison système raccorde des espaces d'adressage différents. Une liaison peut aussi être composite, selon le niveau d'abstraction où on se place. Par exemple, un appel de méthode distante dans un ORB établit une liaison entre objets applicatifs mais aussi entre talons côté client et serveur et entre points de communication (« endpoints ») des protocoles utilisés. Une liaison globale comprend donc un ensemble de sous-liaisons capables elles-mêmes de contribuer à plusieurs autres liaisons. Dans un contexte réparti, une liaison combine typiquement liaisons langage, système et ressources de communication.

Les sémantiques de liaison sont variées: interaction par invocation d'opération synchrone ou asynchrone ou par échange de flux; entités persistantes, mobiles, ou répliquées reliées par des protocoles de communication de groupe; réservation de ressources avec garantie d'une certaine qualité de service.

Un middleware flexible vise la possibilité d'offrir aux applications de contrôler dynamiquement l'établissement de liaisons de différents types et de les combiner. Cette flexibilité permet la mise en oeuvre de différentes sémantiques de domaines et de communication entre domaines.

Elle autorise une utilisation variée de protocoles et de stratégies de gestion des ressources de communication: introduction de nouveaux protocoles à la demande, contrôle fin du niveau de multiplexage des ressources. Pour atteindre ces objectifs, ces middleware mettent en oeuvre des principes architecturaux simples comme la séparation entre présentation des données et aspects protocolaires ou entre multiplexage des ressources et gestion de la concurrence. Plus généralement, ils distinguent clairement mécanismes et politiques afin d'offrir une adaptabilité maximale.

Ces différents middleware devraient pouvoir servir de base pour réaliser une infrastructure à base de domaines. Ces derniers n'y sont présents que de manière implicite au niveau des principes de conception. La sémantique des liaisons existantes est souvent peu explicitée et semble très liée à leur implantation. Leur formalisation est encore plus rare. Cependant, la compositionnalité des liaisons devrait permettre le rapprochement entre les primitives d'assez bas niveau offertes par de telles plates-formes, et celles de haut niveau spécifiées de manière abstraite par un modèle de programmation.

2.5.3 Plates-formes à composants

La section précédente a mis en évidence l'importance de la notion de liaison dans les plates-formes middleware actuelles. Les courtiers d'objets implantent en priorité l'aspect communication de la liaison: l'objectif principal est de connecter différents composants préexistants s'exécutant sur des sites distincts. Dans les plates-formes à composants, l'accent est au contraire mis sur l'aspect compositionnel de la liaison: il s'agit d'assembler des unités exécutables indépendantes de traitement et de déploiement pour obtenir un système de plus grande taille. Nous détaillons maintenant les manifestations de domaines dans les plates-formes à composants à travers quelques exemples.

Domaines et composition contextuelle

Dans [162], Szyperski donne une définition générale d'un composant logiciel:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Cette définition souligne qu'un composant interagit avec son environnement à travers des points d'accès bien identifiés, ses *interfaces*. Il doit spécifier clairement les interfaces qu'il fournit à son environnement, et celles dont il a besoin pour s'exécuter. Ces dernières manifestent des dépendances du composant vis à vis de son contexte de composition et de déploiement. Ce type de composition, commun aux principaux modèles de composants industriels actuels, est qualifié dans [162] de *composition contextuelle*.

Dans cette approche, pour qu'un système composite vérifie certaines propriétés non-fonctionnelles (sécurité, fiabilité, etc.), les composants doivent spécifier leurs dépendances vis-à-vis d'un *contexte de composition* (contexte de sécurité, transactionnel, etc.); ces contextes doivent être bien formés dans un sens qui dépend de la propriété recherchée; enfin, la composition n'est possible qu'à l'aide de règles de composition spécifiques dont l'application dans des contextes bien formés garantit la propriété voulue pour le composite [162]. Les contextes regroupent donc des instances de composants dont les propriétés sont comparables. Ils peuvent donc être vus comme une déclinaison particulière de la notion de domaine. Ainsi la composition de composants dans un contexte transactionnel représente l'exécution de ces composants au sein d'une même transaction, dont la portée s'étend jusqu'à la frontière du contexte. La composition contextuelle peut nécessiter le filtrage des messages qui traversent la frontière du contexte, comportement déjà mis en évidence pour les domaines.

Dans la composition contextuelle, un domaine se manifeste par un *conteneur de composants*, structure d'accueil qui héberge un ensemble de composants et garantit certaines propriétés non-fonctionnelles de leur exécution comme la sécurité, la persistance ou les transactions: le conteneur incarne le contexte de composition correspondant. Il intercepte les interactions des composants avec l'extérieur du contexte et offre un certain nombre de *services techniques* associés à chaque type de propriété visées. Selon les services fournis, un conteneur présente donc simultanément une sémantique de domaine de sécurité, de transactions, d'équilibrage de charge, etc. Nous donnons ci-dessous quelques exemples de conteneurs de composants (cf. figure 2.16). Pour plus de détails, le lecteur se réfèrera utilement à [162].

COM+

Des premières formes de contexte apparaissent dans COM+ [122] qui sépare les objets selon leur modèle d'exécution (multi-threadé ou non) et leur caractère transactionnel. Un conteneur COM+ est le moniteur transactionnel MTS (Microsoft Transaction Server) [121]. Il utilise des *contextes transactionnels* regroupant les ressources affectées par une transaction et intercepte les invocations qui leur sont destinées en rajoutant des pré- et post-traitements, par exemple pour initier ou valider une transaction. COM+ introduit également des *domaines* qui regroupent des objets partageant les mêmes propriétés de contexte. La portée d'un contexte se limite à un processus système tandis que celle d'un domaine peut être répartie. Les domaines peuvent se recouvrir. La figure 2.14 illustre deux domaines typiques COM+ et le partitionnement en contextes correspondant.

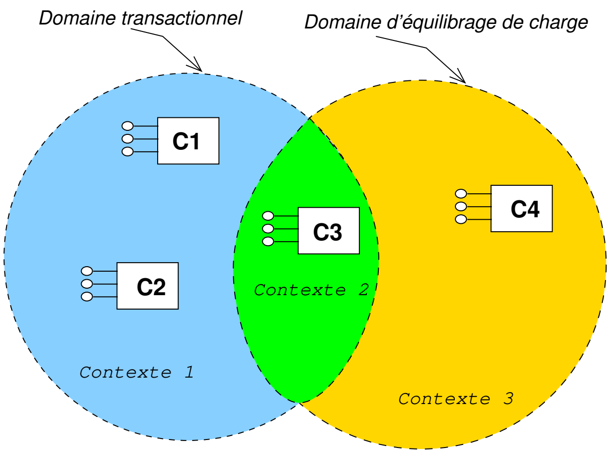


FIG. *2.14* – **Contextes et domaines COM+ (tiré de [162]).**

EJB

Les EJB (Enterprise Java Beans) [161] constituent l'exemple le plus intégré de composition contextuelle. Cœur de la plate-forme J2EE (Java 2 Enterprise Edition), les composants EJB sont destinés à la programmation de serveurs d'applications d'entreprise flexibles. L'architecture des EJB (cf. figure 2.15) comporte trois niveaux: un *serveur* EJB héberge des composants appelés *beans* qui constituent la logique métier du serveur d'applications. Cependant, il n'est pas possible pour un client de communiquer directement avec les beans. Un niveau intermédiaire, le *conteneur* EJB isole les beans du serveur en interceptant toutes leurs communications. Les services techniques fournis par le conteneur couvrent la persistance, la sécurité, et les transactions. Le conteneur est un contexte d'exécution des composants pour chacune de ces trois propriétés non-fonctionnelles et capture donc simultanément trois sémantiques de domaine différentes.

Tout accès au contenu d'un bean transite par deux interfaces du conteneur EJB. L'interface `Home` gère le cycle de vie des beans: création, destruction ou localisation de composants. L'interface `Remote` sert d'intercepteur pour toutes les invocations de méthodes métier d'un bean, le conteneur en contrôlant l'exécution à l'aide de pré- et de post-traitements qui implémentent le comportement transactionnel, de persistance ou de sécurité du bean. Ce code d'interposition s'appuie sur des services système situés dans le serveur EJB comme un moniteur transactionnel, une base de données, ou un service d'authentification et d'autorisation. Selon les implémentations des EJB, le conteneur peut fournir d'autres services, ce qui introduit de nouvelles sémantiques de domaines: par exemple, de désignation, en publiant le nom d'un bean sur le réseau, ou d'équilibrage de charge en fédérant l'exécution de plusieurs serveurs EJB.

Le conteneur est aussi un outil de déploiement qui va générer automatiquement les talons et squelettes pour établir une liaison distante entre un bean et l'extérieur. Le code d'interposition est généré à partir d'un *descripteur de déploiement*, sorte de contrat entre le bean et son conteneur décrivant le type d'exécution souhaité et utilisé pour configurer les différents services du conteneur. Par exemple, un bean peut décider d'implémenter lui-même son comportement transactionnel ou de gérer sa persistance sans faire appel aux services du conteneur.

Cependant, un conteneur EJB présente un degré d'adaptabilité limité: le processus de génération du code d'interposition reste figé et la prise en compte d'une nouvelle propriété non-fonctionnelle (et donc d'une nouvelle sémantique de domaine) implique la réécriture du générateur de code.

CCM

CCM (CORBA Component Model) [134] étend les conteneurs EJB pour les intégrer au sein de l'architecture CORBA. L'interface d'un composant CCM spécifie non-seulement les opérations fournies (*facettes*) et requises (*réceptacles*) qui servent aux composants pour communiquer par invocation synchrone. Elle comporte aussi des puits et sources d'événements, reliés à des canaux d'événements permettant l'interaction entre composants sur un mode publish/subscribe. Un composant se subdivise en *segments*, que l'on peut identifier à des domaines de persistance gérés au niveau du composant. Chaque composant est placé dans un domaine d'exécution séparé (*exécuteur*) qui interagit avec le conteneur CCM offrant un contexte d'accès à différents services. Par rapport aux EJB, un conteneur CCM fournit en plus un service de notification d'événements. CCM dispose donc des mêmes sémantiques de domaines que dans les EJB, avec introduction de domaines, à l'intérieur même d'un composant! L'architecture obtenue reste complexe, et peu extensible, dans la mesure où CCM ne permet pas le support de propriétés non-fonctionnelles supplémentaires.

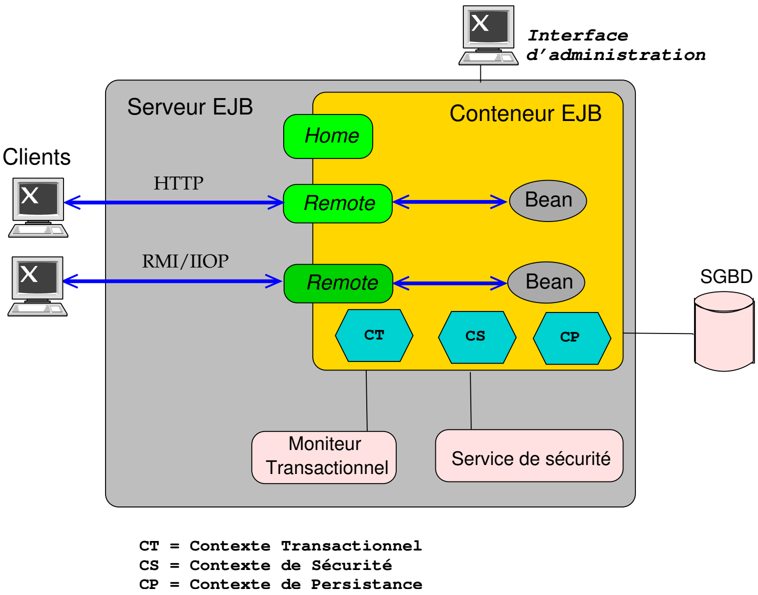


FIG. *2.15* – **Architecture des EJB.**

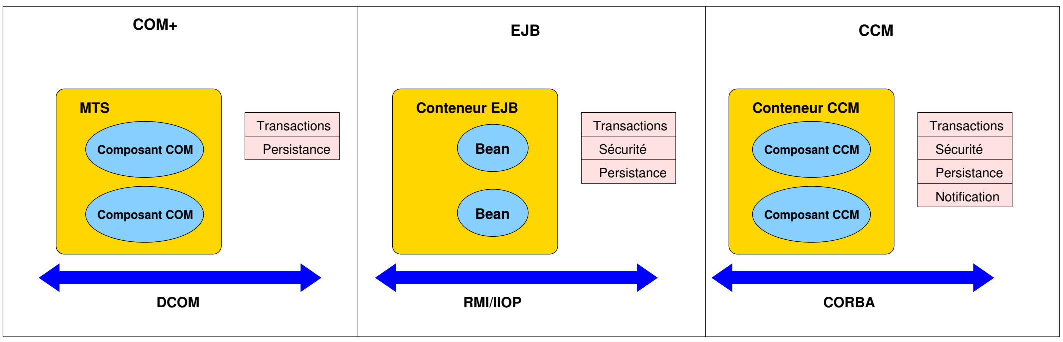


FIG. *2.16* – **Notions de conteneur dans les plates-formes à composants.**

FRACTAL

En introduisant des domaines à l'intérieur même des composants, CCM a contribué à rapprocher les deux notions, qui ont en commun d'exhiber une structure hiérarchique. Or, les modèles de composition présentés jusqu'ici ont une structure plate, et n'autorisent pas une composition arborescente de composants de profondeur arbitraire. D'autre part, ces modèles ne sont pas extensibles car ne capturant qu'un nombre restreint de propriétés non-fonctionnelles dont le nombre ne peut pas être augmenté. Ces plates-formes ne semblent donc pas adaptées pour la programmation en environnement de type « global computing » qui requiert des infrastructures hautement adaptables et la possibilité d'introduire des formes arbitraires de composition (cf. exigence REQ.CONTROL). FRACTAL [33, 136] tente de pallier ces insuffisances en rendant la notion de domaine manifeste à l'exécution sous forme de composant. Un composant FRACTAL comprend un *contrôleur* qui supervise l'exécution d'un *contenu* pouvant inclure d'autres composants. Le contrôleur implémente une sémantique arbitraire de domaine. Le modèle autorise le recouvrement de composants qui sont alors sous le contrôle de différents contrôleurs.

La liaison entre composants s'effectue par un système de connecteurs étendus qui relient les interfaces fournies et requises des composants. Le fait que ces connecteurs soient eux-mêmes des composants FRACTAL quelconques permet l'implémentation de sémantiques de liaison arbitraires entre les composants, ainsi que leur composition puisque le connecteur peut aussi être un composant composite.

Un contrôleur FRACTAL est un conteneur étendu qui autorise un contrôle fin du comportement des composants qu'il supervise et filtre tous les messages reçus et émis par ces composants. FRACTAL permet ainsi de contrôler les relations de contenance et les liaisons entre composants, deux éléments clés pour capturer la dynamicité d'un système à large échelle.

FRACTAL semble ainsi être très adapté prolonger la notion de domaine jusqu'au niveau d'une machine virtuelle. Cependant, les premières sémantiques de liaison FRACTAL en contexte réparti ne viennent d'être implantées que très récemment, ce qui ne rend pas le modèle directement utilisable pour valider notre approche.

Autres approches

La composition contextuelle n'est pas la seule approche de composition possible. Ainsi, les langages de description d'architecture (ADL) [7, 119, 120] séparent un système en *composants* ou domaines d'exécution élémentaires, et *connecteurs* qui encapsulent les interactions entre composants au sens large: communication, mais aussi services d'infrastructure comme la sécurité, les transactions ou la synchronisation. Les connecteurs s'apparentent donc à un contrôleur de domaine, mais l'étendue spatiale du contrôle n'est pas toujours rendue explicite. Enfin, les ADL considèrent principalement la configuration d'un système d'un point de vue statique. Dans un réseau à large échelle, cette approche semble peu compatible avec les exigences `REQ.DYNAMIC_BINDING` capurant le caractère dynamique des liaisons aux ressources et `REQ.CONTROL` incluant la reconfiguration dynamique de la structure d'un système réparti.

La méta-programmation [84] illustre une autre stratégie d'implémentation des domaines. La primitive de domaine est éclatée en général en deux niveaux: le *niveau de base* et le *niveau méta* capturent respectivement les comportement des entités contrôlées et de contrôle d'un domaine. Par exemple, à chaque objet du niveau de base est associé un ou plusieurs *méta-objets* qui interceptent les invocations sur cet objet et peuvent effectuer des pré- et post-traitements. La relation entre le nombre d'objets et de méta-objets est variable selon les implantations. Un domaine s'identifie alors avec un couple $(m, \tilde{o})$ où m est un méta-objet et $\tilde{o}$ l'ensemble des objets qu'il contrôle. La sémantique de domaine dépend du type du méta-objet et de la propriété non-fonctionnelle qu'il implémente. Diverses sémantiques de domaines peuvent être combinées en associant plusieurs méta-objets à un même groupe d'objets. Un cas particulier important de méta-programmation est celui de la programmation par aspects [85] qui sépare le comportement de contrôle en différentes préoccupations non-fonctionnelles (« concerns »), capturées au niveau langage par différents *aspects*. Ces derniers sont ensuite « tissés » avec le niveau de base qui contient la logique métier de l'application. L'entrelacement résultant d'aspects forme alors le programme exécutable. L'application de la programmation par aspects à l'univers du « global computing » reste encore inexplorée.

Domaines, liaison et composants

Cette section a montré que des liens étroits unissent domaines, composants et liaisons (cf. figure 2.17). Un domaine se manifeste sous forme de conteneur de composants, structure d'accueil pour leur exécution, contrôlant leur organisation, filtrant leurs interactions avec l'extérieur et leur donnant accès un certain nombre de services utilisés pour implanter la sémantique de domaine. Ces services sont souvent eux-mêmes implantés à base de composants.

Les liaisons sont utilisées pour interconnecter les composants. Comme ceux-ci constituent en général l'unité d'implémentation d'une sémantique de domaine, on peut considérer aussi les liaisons comme une forme flexible de connexion entre domaines. Comme les domaines, les liaisons peuvent aussi être réalisées par composition de composants. Les domaines contrôlent en général leur établissement.

Ces trois notions ont en commun de posséder une frontière, qui délimite l'étendue sur laquelle s'applique leur comportement ou leur contrôle. En première approche, on peut considérer un composant, comme une unité d'implémentation, une liaison comme une unité de communication, et un domaine comme une unité de contrôle. Cependant, les deux premières notions semblent en fait être des instances particulières de l'abstraction de domaine, le comportement de contrôle étant spécialisé soit dans la composition, soit dans la communication. Disposer d'une flexibilité au niveau des sémantiques de domaines, permet donc, par contre-coup d'obtenir à la fois des modèles flexibles de composition, et une adaptabilité des sémantiques de liaison. Elle devrait donc en principe rendre possible l'implémentation des différentes plates-formes à composants et middleware adaptables examinés ici.

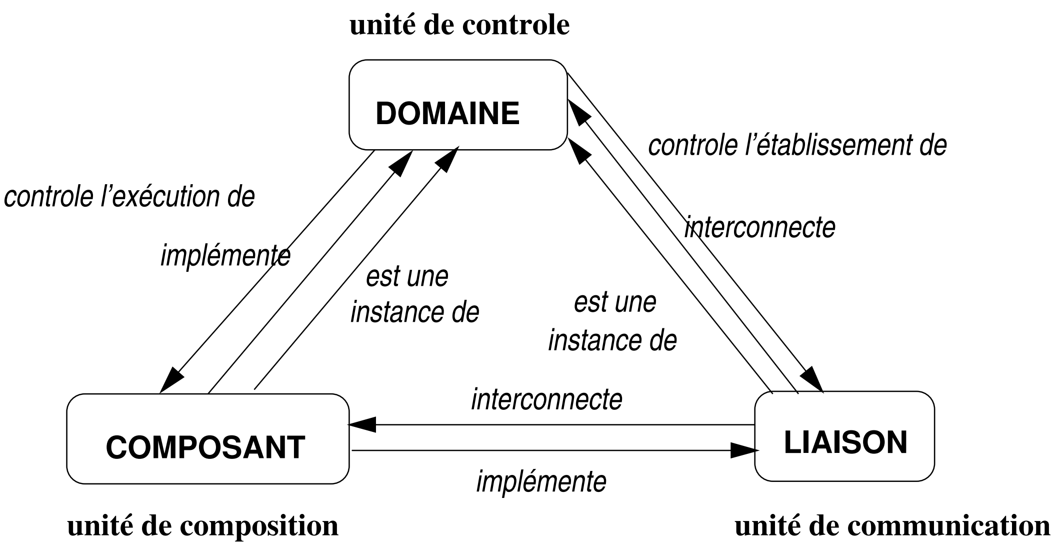


FIG. *2.17* – **Domaines, liaisons et composants.**

2.5.4 Plates-formes d'agents mobiles

La notion de domaine est présente aussi dans les plates-formes d'agents mobiles. Ces systèmes manifestent une diversité extrême et dans l'ensemble leur sémantique est assez peu formalisée. On peut citer par exemple AGENT TCL [47, 73] ou AGLETS [95]. Nous renvoyons le lecteur à de nombreux états de l'art existant sur le code mobile [46, 64, 163]. Un domaine de migration prend deux formes principales: une unité de migration, qui selon les plates-formes abrite plus ou moins de ressources et une structure d'accueil des entités mobiles. Un domaine s'identifie alors à la notion de *place*, mise en évidence dans le manifeste TELESCRIPT [174]. Il sert de référentiel pour l'implémentation d'une politique de mobilité, qui dépend fortement de la plate-forme considérée. Divers types de domaines de sécurité [173] permettent également d'assurer la confidentialité d'un groupe d'agents ou de protéger les ressources des places contre des agents malveillants.

2.5.5 Conclusion

Les plates-formes à objets répartis, middleware flexibles, systèmes à composants ou d'agents mobiles manifestent une diversité impressionnante dans leurs primitives de base (objets, agents, composants, ...), leurs modèles de communication et de migration. On observe quasiment autant de modèles de programmation différents que de plates-formes.

Dans la perspective du « global computing », ces systèmes se caractérisent par une faible adaptabilité: les modèles sous-jacents relativement simples montrent leurs limites face à des configurations de ressources hautement dynamiques s'exécutant sur des plates-formes hétérogènes, et nécessitant des schémas de communication et de migration très variables.

Chaque système tente une approche intégrée du problème en proposant de multiples extensions a posteriori. Le résultat est une divergence forte de spécifications, sans véritable modèle de programmation réparti uniforme pour les environnements de réseaux à large échelle. La sémantique de chaque plate-forme reste largement informelle, implicite et liée à son implantation, ce qui rend sa formalisation ardue. Cette lacune, rend quasiment impossible de garantir la fiabilité d'exécution de programmes s'exécutant sur ces plate-formes réparties dans ce type d'environnement. Ce fait vient s'ajouter au fossé qui sépare les primitives d'assez bas niveau offertes par ces systèmes des abstractions dégagées par les modèles classiques de programmation répartie.

La notion de domaine montre ici tout son intérêt pour capturer la diversité de ces situations dans un modèle de programmation uniforme. Elle permet de leur donner une dimension formelle, tout en étant reflétée implicitement dans l'ensemble des plates-formes middleware existantes.

2.6 Plan de la thèse

La suite de ce mémoire se compose de trois parties qui développent les idées exposées dans ce chapitre. Nous y défendons la thèse suivante: le développement d'infrastructures réparties fiables pour la programmation sur des réseaux à large échelle est rendu possible par l'utilisation d'une démarche de formalisation par raffinements successifs où la notion de domaine joue un rôle prépondérant, depuis le modèle de programmation jusqu'à son implantation par une machine virtuelle répartie.

La première partie, composée du chapitre 3, explore la faisabilité et le bien-fondé de la démarche de formalisation proposée en l'illustrant à l'aide d'un modèle simple de programmation répartie appelé DCP: dans cette expérimentation préliminaire, nous décrivons les différentes étapes de raffinement depuis la conception de ce modèle de programmation répartie qui manifeste une vue minimale des domaines jusqu'à son implémentation middleware. L'implémentation de DCP est spécifiée formellement par une machine abstraite. Sa réalisation prend la forme d'un run-time réparti écrit en JAVA, intégré avec une plate-forme middleware typique – l'ORB JONATHAN – qui fournit un support à la transparence des communications réseau.

La seconde partie, composée des chapitres 4, 5 et 6 applique les enseignements tirés de cette expérimentation au cas de la programmation répartie à large échelle, en détaillant les étapes de construction d'une infrastructure répartie à base de domaines.

Le chapitre 4 présente un nouveau modèle de programmation orientée domaines, le M–calcul. Ce calcul de processus réparti permet la manipulation de domaines sous forme de localités dont le comportement est explicitement programmable. Ses autres caractéristiques principales sont la communication d'ordre supérieur permettant la mobilité de processus, une forme de liaison dynamique, et un contrôle explicite du routage dans une arborescence de localités: les messages peuvent être filtrés lors de chaque franchissement de frontière de localité. L'ensemble de ces caractéristiques font du M–calcul un candidat sérieux comme modèle formel de programmation pour le « global computing ».

Le chapitre 5 décrit la deuxième étape de la démarche proposée, appliquée au M–calcul: une spécification formelle d'une implémentation du M–calcul est détaillée sous forme d'une machine abstraite répartie appelée CLAM (Cellular Abstract Machine). La machine présente une structure modulaire et distingue clairement l'évaluation d'expressions fonctionnelles du cœur du moteur d'exécution, lui-même séparé des aspects marshalling, localisation, et routage des messages.

Le chapitre 6 est consacrée à l'implémentation de la machine abstraite précédente sous forme de machine virtuelle centralisée, puis répartie. Cette plate-forme appelée OYSTERS offre un support pour une programmation effective à l'aide de domaines. La structure sous forme de composants de la machine virtuelle est clairement identifiée. Ce chapitre montre également comment les interfaces qui la composent pourraient être étendues à un framework générique pour la construction de machines virtuelles implantant des calculs de processus répartis.

Enfin, la dernière partie, composée du chapitre 7 évalue qualitativement la machine virtuelle OYSTERS par rapport à d'autres machines abstraites. Ce chapitre tire les enseignements et les limites de la démarche proposée, ainsi que les perspectives de recherche qu'elle dégage.

Chapitre 3

DCP : une expérience préliminaire de programmation répartie

Le chapitre précédent a mis en évidence l'intérêt de la primitive de domaine pour la programmation répartie à large échelle ainsi qu'une méthodologie de construction d'applications réparties fiables par raffinements successifs. Dans ce chapitre, nous analysons cette démarche à l'aide d'un modèle élémentaire de programmation répartie appelé DCP. Nous détaillons les étapes de formalisation depuis la conception du modèle jusqu'à son implémentation par une machine virtuelle, en passant par une phase de spécification formelle sous forme de machine abstraite. L'ensemble se fonde sur une forme de domaine qui se limite ici à rendre la répartition physique explicite : un domaine représente donc un sous-réseau, une machine, un processus UNIX, etc. D'autres formes de contrôle plus sophistiquées seront développées dans les chapitres suivants.

Après avoir précisé l'approche technique, nous décrivons dans la section 3.2 le modèle DCP en terme de choix de conception, de syntaxe et de sémantique opérationnelle. La section suivante est consacrée à la machine abstraite DCP et à ses propriétés. Enfin, nous présentons dans la section 3.4 une implémentation répartie du modèle DCP sur une plate-forme middleware typique. Le chapitre se termine par un bilan de l'expérimentation et les perspectives qu'elle ouvre pour la suite. Pour une description plus complète de cette expérimentation, le lecteur pourra par exemple se reporter à [91] ou à [92].

3.1 Approche technique

Pour obtenir un modèle de programmation implantable, nous avons choisi PICT (cf. section 2.4.1), langage dont les fondements théoriques sont particulièrement bien définis. Ce langage fonctionnel de haut niveau pour la programmation concurrente est implantable uniquement en terme de primitives du π–calcul: PICT est construit au-dessus d'un calcul minimal appelé CORE PICT qui est un π–calcul polyadique asynchrone statiquement typé. Turner [164] formalise une implémentation de PICT sous forme de machine abstraite et montre une forme de correction de la machine par rapport au π–calcul. Le run-time de PICT utilise le π–calcul comme code machine intermédiaire car il est compact et se prête à une implémentation efficace. Cependant, la répartition n'est pas prise en compte au niveau du modèle ou de son implantation.

Les raisons ci-dessus font de PICT un bon candidat comme point de départ pour notre expérimentation: nous construisons une version répartie de PICT pour valider l'approche en montrant les raffinements utilisés lors de chaque étape de formalisation, depuis un modèle de programmation répartie ayant une vue minimale des domaines jusqu'à son implantation middleware.

Plus précisément, nous introduisons un modèle simple de programmation appelé DCP (Distributed Core Pict) qui étend CORE PICT en introduisant une forme simple de domaine et des primitives de communication distante. Le modèle rend la répartition des ressources explicite tout en conservant une transparence des communication réseau.

Le but de ce chapitre est de montrer la validité de l'approche ci-dessus en se plaçant dans l'optique d'une étude de faisabilité: une implémentation de ce modèle est spécifiée formellement sous forme d'une machine abstraite répartie qui étend celle de Turner en introduisant des règles supplémentaires pour gérer la répartition. Cette spécification formelle est ensuite implantée de manière répartie en JAVA, en s'intégrant avec un middleware représentatif des technologies à objets répartis (le canevas d'ORB JONATHAN) pour assurer la transparence des communications réseau.

3.2 Le modèle de programmation DCP

3.2.1 Choix de conception

Démontrer la faisabilité d'une approche de formalisation de bout en bout signifie formaliser une implémentation d'un modèle simple de programmation répartie, et s'assurer que son implantation middleware suit effectivement la spécification formelle, elle-même obtenue par raffinement du modèle. Pour les besoins de cette expérimentation, les exigences du modèle étaient donc la *simplicité* (exigence `REQ.SIMPLICITY`) tout en reflétant fidèlement les infrastructures existantes d'exécution répartie (exigence `REQ.NO_HIDDEN_COSTS`). Dans cette perspective, le modèle DCP est basé sur les choix suivants:

- *Distribution explicite des ressources:* comme l'a montré le chapitre 2, la notion de domaine capture les partitions d'un système réparti. En plaçant cette notion au cœur d'un modèle de programmation, on peut modéliser de nombreuses situations de la programmation répartie mobile avec une primitive unique. Cependant, par souci de simplicité, les domaines DCP se limitent dans cette expérimentation à des localités passives capturant la distribution physique sans aucun autre comportement additionnel. Les domaines sont organisés en une structure plate.
- *Communication distante primitive:* les calculs existants recherchent un compromis souvent difficile entre primitives pour le contrôle de la communication et de la migration. Dans DCP, nous privilégions une approche résolument orientée communication [1]. DCP est une extension directe du π–calcul asynchrone. L'inclusion de primitives de communication distante directement dans le modèle, plutôt que par encodage à l'aide des primitives de mobilité – comme la sémantique « move and communicate » rencontrée dans NOMADIC PICT, DITYCO ou les calcul des ambients – nous semble plus réaliste en terme de coûts d'implantation. Ainsi, DCP se rapproche du join–calcul où la communication distante est une primitive distincte de celle de migration de processus.

[1] Cette approche permet aussi de modéliser la migration en incluant dans le modèle des primitives d'ordre supérieur: les processus sont alors transmis comme valeurs sur les canaux de communication [146].

- *Transparence des communications:* DCP utilise des noms dont la portée lexicale est potentiellement répartie, ce qui rend possible un routage implicite en cachant au programmeur la complexité du réseau, comme dans le join–calcul.

3.2.2 Syntaxe et sémantique opérationnelle

Dans DCP, un système réparti est représenté à l'aide d'un nombre limité d'abstractions: les *configurations*, les *domaines*, les *processus* et les *canaux*. La syntaxe est donnée dans la figure 3.1.

Une *configuration* $C, D \ldots$ est une composition parallèle de domaines $a[P]$ et de messages M [2] en transit entre domaines. Un système inerte est représenté par la configuration vide nil. Un *domaine* regroupe à l'intérieur de sa frontière des unités d'exécution élémentaires, les *processus*. Un domaine a contenant les processus P,Q,... qui s'exécutent en parallèle sera noté $a[P \mid Q \mid \ldots]$. Les domaines sont organisés de manière plate. Deux opérateurs distincts de composition parallèle sont utilisés, $\|$ pour les configurations, et $\mid$ pour les processus abrités par un domaine.

Les processus communiquent par envoi asynchrone de messages sur des *canaux*, à l'intérieur d'un même domaine ou entre domaines distincts. Les canaux ont des noms uniques $m, n, x, y \ldots$. Ces valeurs primitives peuvent être transmises sur d'autres canaux, ce qui provoque une reconfiguration de la topologie de communication des processus qui acquièrent de nouvelles capacités d'interaction. Le calcul dispose d'une seule notion de nom qui représente aussi bien un nom local, valide uniquement à l'intérieur d'un domaine donné qu'un nom réseau, valide en tout point du système réparti.

Les primitives de DCP sont directement inspirées de celles de CORE PICT et comprennent les constructions standard du π–calcul comme le processus inactif $\mathsf{0}$, le branchement conditionnel, la composition parallèle, la création $(\nu\ n)\ P$ d'un nouveau nom de canal dont la portée lexicale est restreinte à P ou la réplication $*P$ qui déclenche l'exécution en parallèle d'un nombre non borné de copies de P. Deux processus interagissent sur un canal x par l'envoi $x!\langle\tilde{y}\rangle$ et la réception $x?(\tilde{y}).P$ de tuples de valeurs $\tilde{y}$, avec exécution de P après la réception.

La sémantique opérationnelle de DCP est définie par une équivalence structurelle $\equiv$ et une relation de réduction $\longrightarrow$, décrites dans la figure 3.1. On a $P \longrightarrow Q \implies a[P] \longrightarrow a[Q]$: la couche de localités introduite est passive, un domaine ne pouvant évoluer que si son contenu évolue.

Les *communications intra-domaine* entre processus s'effectuent comme en π–calcul asynchrone (règle COM): l'interaction entre deux processus émettant et recevant sur un même canal m résulte en l'exécution de la continuation du récepteur dont les paramètres formels sont remplacés par les valeurs transmises.

Les *communications inter-domaines* sont décrites par les règles SEND et RECEIVE: si le récepteur m, cible de la communication, est situé dans un domaine différent – m n'appartient pas aux récepteurs libres $\mathsf{dn}(P)$ du domaine courant $a[P \mid \ldots]$ – le processus émetteur disparaît du domaine courant et un message est émis sur le réseau vers le domaine distant contenant les valeurs à transmettre. Ce message sera consommé en parvenant sur le domaine contenant le récepteur cible: un processus émetteur est alors créé localement, l'interaction se terminant par une communication locale.

[2] On notera x le récepteur cible d'un message $\langle x, \tilde{y}\rangle$ et $\tilde{y}$ ses arguments.

Syntaxe:

C	$::=$	$\mathsf{nil} \mid (\nu\, n)\; C \mid M \mid a[P] \mid C \parallel C$	Configuration
P	$::=$	$P \mid P \mid (\nu\, n)\; P \mid \mathsf{0} \mid x?(\tilde{y}).P \mid x!\langle\tilde{y}\rangle \mid *\,P \mid [u\;=\;v]\;P\;,\;Q$	Processus
M	$::=$	$\langle x, \tilde{y}\rangle$	Message

Noms libres et noms définis:

$$\begin{array}{rcl}
\mathsf{fn}(P \mid Q) & = & \mathsf{fn}(P) \cup \mathsf{fn}(Q)\\
\mathsf{fn}((\nu\, n)\; P) & = & \mathsf{fn}(P) \setminus \{n\}\\
\mathsf{fn}(\mathsf{0}) & = & \varnothing\\
\mathsf{fn}(x?(\tilde{y}).P) & = & \{x\} \cup (\mathsf{fn}(P) \setminus \{\tilde{y}\})\\
\mathsf{fn}(x!\langle\tilde{y}\rangle) & = & \{x\} \cup \{\tilde{y}\}\\
\mathsf{fn}(*P) & = & \mathsf{fn}(P)\\
\mathsf{fn}([u\;=\;v]\;P\;,\;Q) & = & \{u, v\} \cup \mathsf{fn}(P) \cup \mathsf{fn}(Q)\\
\mathsf{fn}(\mathsf{nil}) & = & \varnothing\\
\mathsf{fn}((\nu\, n)\; C) & = & \mathsf{fn}(C) \setminus \{n\}\\
\mathsf{fn}(\langle x, \tilde{y}\rangle) & = & \{x\} \cup \{\tilde{y}\}\\
\mathsf{fn}(a[P]) & = & \mathsf{fn}P\\
\mathsf{fn}(C \parallel D) & = & \mathsf{fn}(C) \cup \mathsf{fn}(D)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{dn}(P \mid Q) & = & \mathsf{dn}(P) \cup \mathsf{dn}(Q)\\
\mathsf{dn}((\nu\, n)\; P) & = & \mathsf{dn}(P) \setminus \{n\}\\
\mathsf{dn}(\mathsf{0}) & = & \varnothing\\
\mathsf{dn}(x?(\tilde{y}).P) & = & \{x\} \cup \mathsf{dn}(P)\\
\mathsf{dn}(x!\langle\tilde{y}\rangle) & = & \varnothing\\
\mathsf{dn}(*P) & = & \mathsf{dn}(P)\\
\mathsf{dn}([u\;=\;v]\;P\;,\;Q) & = & \mathsf{dn}(P) \cup \mathsf{dn}(Q)\\
\mathsf{dn}(\mathsf{nil}) & = & \varnothing\\
\mathsf{dn}((\nu\, n)\; C) & = & \mathsf{dn}(C) \setminus \{n\}\\
\mathsf{dn}(\langle x, \tilde{y}\rangle) & = & \varnothing\\
\mathsf{dn}(a[P]) & = & \mathsf{dn}P\\
\mathsf{dn}(C \parallel D) & = & \mathsf{dn}(C) \cup \mathsf{dn}(D)
\end{array}$$

Contextes d'évaluation:
$$\begin{array}{lcl}
\mathbf{E}(\cdot) & ::= & \cdot \mid (\nu\, n)\; \mathbf{E}(\cdot) \mid \mathbf{E}(\cdot) \mid P\\
\mathbf{F}(\cdot) & ::= & \cdot \mid (\nu\, n)\; \mathbf{F}(\cdot) \mid \mathbf{F}(\cdot) \parallel C
\end{array}$$

Equivalence structurelle $\equiv$: plus petite relation d'équivalence, fermée par α-conversion et par les contextes $\mathbf{E}(\cdot)$, $*(\cdot)$, $a[\cdot]$, et $x?(\tilde{y}).\cdot$ pour les processus, et $\mathbf{F}(\cdot)$ pour les configurations et vérifiant:

$$\begin{array}{llll}
(P1) & P \mid \mathsf{0} \equiv P & (C1) & C \parallel \mathsf{nil} \equiv C\\
(P2) & P \mid Q \equiv Q \mid\; P & (C2) & C \parallel D \equiv D \parallel C\\
(P3) & P \mid (Q \mid R) \equiv (P \mid Q) \mid R & (C3) & C \parallel (D \parallel E) \equiv (C \parallel D) \parallel E\\
\\
(R1) & *P \equiv P \mid *P & (Px) & (\nu\, n)\; P \mid Q \equiv (\nu\, n)\; (P \mid Q) \quad n \notin \mathsf{fn}(Q)\\
(R2) & a[(\nu\, n)\; P] \equiv (\nu\, n)\; a[P] & (Cx) & (\nu\, n)\; C \parallel D \equiv (\nu\, n)\; (C \parallel D) \quad n \notin \mathsf{fn}(D)
\end{array}$$

Relation de réduction $\longrightarrow$: plus petite relation compatible avec $\equiv$, fermée par les contextes $\mathbf{E}(\cdot)$, et $a[\cdot]$ pour les processus, et $\mathbf{F}(\cdot)$ pour les configurations et vérifiant:

$$\begin{array}{lll}
\text{IF-TRUE} & [u\;=\;u]\;P\;,\;Q \longrightarrow P & \\
\text{IF-FALSE} & [u\;=\;v]\;P\;,\;Q \longrightarrow Q & u \neq v\\
\text{COM} & m?(\tilde{u}).P \mid m!\langle\tilde{v}\rangle \longrightarrow \{\tilde{v}/\tilde{u}\}P & \\
\text{SEND} & a[P \mid m!\langle\tilde{v}\rangle] \longrightarrow a[P] \parallel \langle m, \tilde{v}\rangle & m \notin \mathsf{dn}(P)\\
\text{RECEIVE} & \langle m, \tilde{v}\rangle \parallel a[P] \longrightarrow a[P \mid m!\langle\tilde{v}\rangle] & m \in \mathsf{dn}(P)
\end{array}$$

FIG. *3.1* – **DCP : syntaxe et sémantique opérationnelle.**

3.2.3 Termes bien formés

Le caractère bien-formé des processus DCP est garanti à l'aide d'un système d'inférence adapté du π–calcul réceptif Dπ_1^r et décrit dans la figure 3.2. Un *environnement* est un ensemble fini de noms de canaux qui représente les récepteurs libres contenus dans un processus ou une configuration. Il capture ainsi l'interface de ce processus ou de cette configuration, au sens de l'ensemble des ports de communication qu'ils peuvent utiliser pour dialoguer avec le monde extérieur. Les jugements du système d'inférence sont de la forme $\Gamma \vdash P$ ou $\Gamma \vdash\ C$ où Γ est un environnement. Ils peuvent se lire : le processus P ou la configuration C sont bien formés dans l'environnement Γ. Un processus P bien formé satisfait les deux propriétés suivantes:

(LOC) P ne peut pas recevoir des messages sur un nom de canal qu'il vient de recevoir sur un autre canal.

(UR) P peut recevoir des messages sur le même nom de canal m qu'un autre processus Q s'ils appartiennent au même domaine, mais pas s'ils sont situés dans des domaines différents.

(LOC) signifie qu'un nom reçu ne peut pas être utilisé pour créer des récepteurs dans la continuation de P. Par exemple, le processus $x?(y).y?(z).0$ n'est pas bien-formé. Cette propriété reflète une certaine hypothèse de localité : un processus devant recevoir des messages ne devrait pouvoir écouter que sur des récepteurs locaux. (UR) est une variante de la contrainte d'unicité des récepteurs, mise en avant par plusieurs modèles de programmation répartis [8, 59, 82] comme étant nécessaire pour qu'un modèle d'exécution distribué soit implémentable. Ainsi, seul un consensus local est nécessaire au lieu d'un accord réparti pour déterminer le récepteur auquel revient la communication. Dans DCP, cette contrainte est imposée pour des récepteurs appartenant à des domaines différents (condition $\Gamma \cap \Delta = \varnothing$ dans la règle T-PAR-C), mais pas à l'intérieur d'un même domaine par compatibilité avec la sémantique de PICT qui autorise des récepteurs multiples sur un même canal de communication ($\Gamma \cap \Delta \neq \varnothing$ reste possible dans la règle T-PAR-P).

La propriété suivante dite de « subject reduction » garantit la préservation du caractère bien formé des processus ou des configurations au cours des réductions du calcul:

Théorème 3.1 (« Subject reduction »). *Soit Γ un environnement bien formé:*

1. *Si $\Gamma \vdash P$ et si $P \longrightarrow Q$, alors il existe un environnement Δ tel que $\Delta \vdash Q$.*
2. *Si $\Gamma \vdash C$ et si $C \longrightarrow D$, alors il existe un environnement Δ tel que $\Delta \vdash D$.*

3.2.4 Un exemple

Parmi les primitives utiles, les définitions récursives sont codables à l'aide de la réplication:

$$\texttt{def}\ X[\tilde{x}] = P\ \texttt{in}\ Q\ =_{def}\ (\nu\ X)(*X?(\tilde{x}).P \mid Q)$$

Considérons un serveur d'écho $ES\ =_{def}\ echo\_d[*echo?(p,r).r!\langle p\rangle]$ qui attend des messages p sur le port public $echo$ et les renvoie sur un canal de retour r passé en paramètre. Un client C envoyant en continu des messages m au serveur d'écho et attendant des réponses sur le canal de retour ack peut se représenter par la configuration:

$$C\ =_{def}\ client[(\nu\ ack)\ \texttt{def}\ P = (\nu\ m)\ echo!\langle m, ack\rangle \mid ack?(y).P!\langle\rangle\ \texttt{in}\ P!\langle\rangle]$$

Environnements: $\Gamma, \Delta, \ldots \quad ::= \quad \varnothing \mid \Gamma, x$ Environnement

Règles d'inférence:

$$\varnothing \vdash 0 \;\; \text{T-NIL-P} \qquad \varnothing \vdash x!\langle\tilde{y}\rangle \;\; \text{T-SND} \qquad \varnothing \vdash \mathsf{nil} \;\; \text{T-NIL-C} \qquad \varnothing \vdash M \;\; \text{T-MSG}$$

$$\frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma \cup \Delta \vdash P \mid Q} \;\text{T-PAR-P} \qquad \frac{\Gamma, n \vdash P \quad n \notin \Gamma}{\Gamma \vdash (\nu\ n)\ P} \;\text{T-RES-P}$$

$$\frac{\Gamma \vdash P \quad y_i \notin \Gamma \quad (1 \leq i \leq k)}{\Gamma, x \vdash x?(y_1, \ldots, y_k).P} \;\text{T-RCV} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash a[P]} \;\text{T-DOM}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash *P} \;\text{T-REPL} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash [u\ =\ v]\ P\ ,\ Q} \;\text{T-IF}$$

$$\frac{\Gamma \vdash C \quad \Delta \vdash D \quad \Gamma \cap \Delta = \varnothing}{\Gamma \cup \Delta \vdash C \parallel D} \;\text{T-PAR-C} \qquad \frac{\Gamma, n \vdash C \quad n \notin \Gamma}{\Gamma \vdash (\nu\ n)\ C} \;\text{T-RES-C}$$

FIG. *3.2* – **Termes DCP bien formés.**

On a alors les réductions successives suivantes qui décrivent l'envoi du message m_0 du client au serveur d'écho et la réponse du serveur sous forme d'acquittement ack:

$$\begin{array}{rll}
C \parallel ES & \longrightarrow & client[(\nu\ ack)\ (\nu\ m_0)\ \ \mathtt{def}\ P = (\nu\ m)\ echo!\langle m, ack\rangle \mid ack?(y).P!\langle\rangle \\
& & \qquad\qquad\qquad\qquad \mathtt{in}\ echo!\langle m_0, ack\rangle \mid ack?(y).P!\langle\rangle] \parallel ES \\
& \longrightarrow & (\nu\ ack)\ (\nu\ m_0)\ \ client[\mathtt{def}\ P = (\nu\ m)\ echo!\langle m, ack\rangle \mid ack?(y).P!\langle\rangle\ \mathtt{in}\ ack?(y).P!\langle\rangle] \\
& & \qquad\qquad\qquad \parallel \langle echo, m_0, ack\rangle \parallel echo_d[*echo?(p,r).r!\langle p\rangle] \\
& \longrightarrow & (\nu\ ack)\ (\nu\ m_0)\ \ client[\mathtt{def}\ P = (\nu\ m)\ echo!\langle m, ack\rangle \mid ack?(y).P!\langle\rangle\ \mathtt{in}\ ack?(y).P!\langle\rangle] \\
& & \qquad\qquad\qquad \parallel echo_d[echo!\langle m_0, ack\rangle \mid *echo?(p,r).r!\langle p\rangle] \\
& \longrightarrow & (\nu\ ack)\ (\nu\ m_0)\ \ client[\mathtt{def}\ P = (\nu\ m)\ echo!\langle m, ack\rangle \mid ack?(y).P!\langle\rangle\ \mathtt{in}\ ack?(y).P!\langle\rangle] \\
& & \qquad\qquad\qquad \parallel echo_d[ack!\langle m_0\rangle \mid *echo?(p,r).r!\langle p\rangle] \\
& \longrightarrow & (\nu\ ack)\ (\nu\ m_0)\ \ client[\mathtt{def}\ P = (\nu\ m)\ echo!\langle m, ack\rangle \mid ack?(y).P!\langle\rangle\ \mathtt{in}\ ack?(y).P!\langle\rangle] \\
& & \qquad\qquad\qquad \parallel \langle ack, m_0\rangle \parallel echo_d[*echo?(p,r).r!\langle p\rangle] \\
& \longrightarrow & client[(\nu\ ack)\ (\nu\ m_0)\ \ \mathtt{def}\ P = (\nu\ m)\ echo!\langle m, ack\rangle \mid ack?(y).P!\langle\rangle \\
& & \qquad\qquad\qquad\qquad \mathtt{in}\ ack!\langle m_0\rangle \mid ack?(y).P!\langle\rangle] \parallel ES \\
& \longrightarrow & C \parallel ES \ldots
\end{array}$$

3.3 La machine abstraite DCP

Cette section formalise l'implémentation du calcul DCP sous forme d'une machine abstraite répartie. La machine est présentée de manière informelle, les définitions principales et les règles de réduction étant rassemblées dans la figure 3.3. Comme la machine de Turner pour le π-calcul asynchrone [164], cette machine est déterministe et vérifie une propriété d'équité: tous les processus susceptibles d'être exécutés le seront éventuellement. Nous conjecturons également une correspondance correcte entre les réductions de la machine et des réductions valides du calcul sous-jacent.

Dans la suite, étant donnés des ensembles A et B, on notera $A \mapsto B$ l'ensemble des associations finies de A vers B, A^* l'ensemble des collections finies d'éléments de A, $\bullet$ la file vide, $a_1 :: \ldots :: a_n$ un élément Q de l'ensemble $Queue(A)$ des files d'éléments de A, et $|Q|$ le nombre d'éléments de Q.

3.3.1 Etat de la machine

Une configuration DCP est implémentée par un *réseau* ou composition parallèle de *machines* et de messages. Chaque domaine DCP est implémenté par une seule machine dont l'état est un quintuplet $S = (a, \mathcal{U}, \mathcal{V}, \mathcal{H}, \mathcal{R})$. Le *nom a de la machine* la désigne de manière unique. $\mathcal{U}$ et $\mathcal{V}$ sont deux ensembles de noms de *canaux locaux et distants*, connus de la machine au cours de son exécution. Le *tas* $\mathcal{H}$ rassemble les processus en attente de communication locale ou distante. Il associe des noms de canaux x à des files FIFO de processus en attente de communication sur x : émetteurs wr ou récepteurs rd (répliqués ou non). On a $x \in \mathcal{U}$ pour une communication locale, et $x \in \mathcal{V}$ pour une communication distante. La *file d'exécution* $\mathcal{R}$ contient les processus prêts à être exécutés, le processus en tête de file étant exécuté en premier. En général, pour assurer l'équité de l'exécution, la plupart des processus nouvellement créés seront placés à la fin de $\mathcal{R}$.

3.3.2 Règles de réduction

Pour gérer la distribution, la notion unique de nom présente dans le calcul doit être précisée par raffinement dans la machine abstraite: on introduit la notion de *référence réseau* pour représenter un nom utilisable en tout point du réseau. Une référence abstrait à la fois la représentation transmise physiquement sur le réseau d'un nom vu comme capacité de communication et les ressources de communication sous-jacentes nécessaires pour qu'une interaction distante puisse se produire. On se donne donc deux fonctions injectives de transformation de noms pickle et $\mathsf{unpickle}$. $\mathsf{pickle}(a, x)$ retourne une référence réseau correspondant à un nom local x créé au sein du domaine a. Réciproquement, $\mathsf{unpickle}(x)$ retourne un couple (a, y), ce qui reflète la résolution de la référence réseau x en un nom local y dans le contexte de nommage du domaine a. Dans DCP, les récepteurs libres d'un domaine sont automatiquement exportés pour être utilisables par les autres domaines.

Pour des raisons d'équité, une composition parallèle de processus est traitée en exécutant un des processus immédiatement, l'autre étant placé en fin de file, pour une exécution différée (règle PRL). Lorsqu'un nom de canal n est créé localement (règle RES), toutes les occurrences de n sont substituées par un nouveau nom u. La machine alloue une nouvelle entrée pour u dans le tas, et met à jour la liste des noms locaux connus.

La communication locale est décrite par les règles REDIN, REDOUT, REDREPL et REDOUT*. Par exemple, quand le redex est constitué d'un processus récepteur dans la file d'exécution et d'un processus émetteur dans le tas, c'est la continuation du récepteur qui est exécutée (règle REDIN). Les autres règles sont similaires, et traitent principalement du cas des récepteurs répliqués. Un processus dont la réduction immédiate n'est pas possible est transféré vers les files d'attente du tas (règles QUEUEOUT, QUEUEIN et QUEUEREPL). Il quittera le tas lorsqu'une interaction locale devient possible – un émetteur ou un récepteur apparaît dans la file d'exécution – ou lorsque la règle d'émission distante REMXMIT est déclenchée.

La communication distante est décrite par les règles REMXMIT et REMRCV qui implémentent l'extension de portée entre domaines des noms de canaux de niveau calcul en utilisant les opérations pickle et unpickle. Si le récepteur x est distant, l'application répétée de la règle REMXMIT vide le tampon de communication $\mathcal{H}(x)$ par émission d'un ou plusieurs messages sur le réseau. Si les arguments du message sont des noms créés localement, il doivent d'abord être transformés en références réseau pour être utilisables par le domaine cible du message. Sinon, ces noms créés sur un site distant sont déjà sous forme de références réseau et peuvent être transmis tels quels.

Lorsqu'un message $\langle x, \tilde{y} \rangle$ est reçu par une machine abstraite (règle REMRCV), celle-ci vérifie que le récepteur cible x appartient bien aux noms locaux connus. C'est le sens de la condition $\mathsf{unpickle}(x) = (a, x')\ \ x' \in \mathcal{U}$. Les arguments du message sont alors résolus localement dans le contexte de nommage de la machine: si le nom ainsi obtenu n'est pas un nom local, l'ensemble $\mathcal{V}$ des noms distants connus de la machine est mis à jour. La règle se termine en introduisant, en fin de file d'exécution locale, un processus émetteur sur le canal x cible du message, désigné ici par son nom local x'. Ce processus sera réduit ultérieurement par déclenchement d'une règle de communication locale.

Contrairement à d'autres machines abstraites comme PICT ou TYCO, on remarque que DCP n'utilise pas d'environnements pour modéliser les associations entre variables et valeurs pour optimiser l'implémentation de la substitution. De même la machine abstraite ne dispose pas de véritable règle d'ordonnancement, ou reste monothreadée avec un seul fil d'exécution. Ces différences sont justifiées par le but poursuivi dans cette expérience: comprendre et maîtriser d'un bout à l'autre les étapes de formalisation lors de la conception d'un modèle de programmation réparti et de son implémentation. La machine présentée, nécessairement plus simple, n'est donc qu'une première tentative pour construire une machine virtuelle calculant avec des domaines, pour l'instant passifs. Ces fonctionnalités seront intégrées à la machine abstraite CLAM décrite au chapitre 5 où les domaines ont des comportements propres.

3.3.3 Propriétés de la machine

Nous postulons que la machine abstraite précédente implémente de manière correcte le calcul DCP, de sorte que toute suite de réductions effectuées par une composition parallèle de machines abstraites correspondent à des réductions valides dans le calcul de base.

Conjecture 3.1 (Correction). *Il existe une fonction d'implémentation $[\![\cdot]\!]$ traduisant les termes de l'ensemble* Network *(réseaux) en termes du calcul* DCP *(configurations), et une notion d'équivalence $\sim$ sur les configurations telle que:* $\forall N \in \mathsf{Network},\ N \longrightarrow N' \Longrightarrow [\![N]\!] \xrightarrow{*} [\![N']\!]$*, à l'équivalence $\sim$ près.*

La machine abstraite DCP vérifie une certaine propriété d'équité d'exécution. Cependant, la propriété « un processus P placé dans la file d'exécution d'une machine sera exécuté après un nombre fini de réductions » n'est pas valide : une machine a peut déclencher autant de fois que nécessaire la règle de réception distante REMRCV, pourvu qu'il y ait un nombre suffisant de messages en route vers a, par exemple si une machine distante b envoie en continu des messages vers des récepteurs de a. Cette situation empêche P de progresser dans la file d'exécution. Il faut donc distinguer deux types d'étapes de réduction. Pour deux états d'une machine abstraite S et S', nous appelons $S \longrightarrow S'$ une *étape de calcul* si S' est obtenu à partir de S en appliquant n'importe quelle règle de réduction sauf REMRCV. Dans le cas contraire, $S \longrightarrow S'$ est une *étape d'écoute*. Pour $n \geq 1, 0 < k \leq n$, une dérivation est qualifiée de *(n,k)-équitable* si et seulement si parmi n étapes de réduction, au moins k sont des étapes de calcul. Une dérivation est dite *équitable* si et seulement s'il existe $n \geq 1$ et $0 < k \leq n$ telle que la dérivation soit (n,k)-équitable.

Définition 3.1 (Etat de machines bien formés). *L'état d'une machine $S = (a, \mathcal{U}, \mathcal{V}, \mathcal{H}, \mathcal{R})$ est* bien formé *si les files d'attente des canaux de $\mathcal{H}$ ne contiennent pas de mélange de processus émetteurs et récepteurs, et si $\mathcal{R}$ contient uniquement des processus bien formés, la réplication étant limitée à des récepteurs répliqués, l'émission (resp. la réception) ne se produisant que sur des canaux appartenant à $\mathcal{U} \cup \mathcal{V}$ (resp. $\mathcal{U}$).*

On obtient alors la propriété suivante, dont la preuve est détaillée dans [91]:

Théorème 3.2 (Equité d'exécution). *Pour une machine abstraite n'effectuant que des dérivations équitables à partir d'un état bien formé, un processus placé dans la file d'exécution sera exécuté après un nombre fini de réductions.*

3.3.4 Un exemple de réduction

Nous illustrons à présent le fonctionnement des règles de communication distante entre deux machines abstraites DCP sur l'exemple du serveur d'écho. Pour simplifier, nous supposons que le serveur connaît déjà le canal de retour ack pour l'acquittement. Considérons la configuration $(\nu\ echo)(\nu\ ack)\ C\ \parallel\ ES$ avec

$$\begin{array}{lll} C & =_{def} & client[(\nu\ m)\ echo!\langle m\rangle \mid ack?(y).0] \\ ES & =_{def} & echo\_d[*echo?(p).ack!\langle p\rangle] \end{array}$$

configurations bien formées dans les environnements respectifs $\{ack\} \vdash C$ et $\{echo\} \vdash ES$, qui reflètent les récepteurs locaux des machines clientes et serveur. On peut donc initialiser les noms connus par ces machines avec :

$$\mathcal{U}_0^{client} = ack \quad \mathcal{V}_0^{client} = echo \quad \mathcal{U}_0^{serveur} = echo \quad \mathcal{V}_0^{serveur} = ack$$

Nous obtenons donc les deux machines suivantes en parallèle:

$$\begin{array}{ll} & client, \{ack\}, \{echo\}, \{ack \mapsto \bullet, echo \mapsto \bullet\}, (\nu\ m)\ echo!\langle m\rangle \mid ack?(y).0 \\ \parallel & echo\_d, \{echo\}, \{ack\}, \{echo \mapsto \bullet, ack \mapsto \bullet\}, *echo?(p).ack!\langle p\rangle \end{array}$$

Côté client, on a les réductions suivantes pour traiter la composition parallèle, la création d'un nouveau nom n et la mise en attente du processus émetteur:

$$\begin{array}{lll} \ldots & \xrightarrow{\text{PRL}} & client, \{ack\}, \{echo\}, \{ack \mapsto \bullet, echo \mapsto \bullet\}, (\nu\ m)\ echo!\langle m\rangle :: ack?(y).0 \\ & \xrightarrow{\text{RES}} & client, \{ack, n\}, \{echo\}, \{ack \mapsto \bullet, n \mapsto \bullet, echo \mapsto \bullet\}, echo!\langle n\rangle :: ack?(y).0 \\ & \xrightarrow{\text{QUEUEOUT}} & client, \{ack, n\}, \{echo\}, \{ack \mapsto \bullet, n \mapsto \bullet, echo \mapsto echo!\langle n\rangle\}, ack?(y).0 \end{array}$$

Côté serveur, on traite le récepteur répliqué:

$$\ldots \xrightarrow{\text{QUEUEREPL}} \mathit{echo\_d}, \{\mathit{echo}\}, \{\mathit{ack}\}, \{\mathit{echo} \mapsto *\mathit{echo}?(p).\mathit{ack}!\langle p\rangle, \mathit{ack} \mapsto \bullet\}, \varnothing$$

La communication distante donne lieu aux réductions suivantes:

$$\begin{aligned}
\xrightarrow{\text{REMXMIT}}\ & \mathit{client}, \{\mathit{ack}, n\}, \{\mathit{echo}\}, \{\mathit{ack} \mapsto \bullet, n \mapsto \bullet, \mathit{echo} \mapsto \bullet\}, \mathit{ack}?(y).0 \parallel \langle \mathit{echo}, n@\mathit{client}\rangle \parallel \\
& \mathit{echo_d}, \{\mathit{echo}\}, \{\mathit{ack}\}, \{\mathit{echo} \mapsto *\mathit{echo}?(p).\mathit{ack}!\langle p\rangle, \mathit{ack} \mapsto \bullet\}, \varnothing \\
\xrightarrow{\text{REMRCV}}\ & \mathit{client}, \{\mathit{ack}, n\}, \{\mathit{echo}\}, \{\mathit{ack} \mapsto \bullet, n \mapsto \bullet, \mathit{echo} \mapsto \bullet\}, \mathit{ack}?(y).0 \parallel \\
& \mathit{echo_d}, \{\mathit{echo}\}, \{\mathit{ack}, n@\mathit{client}\}, \{\mathit{echo} \mapsto *\mathit{echo}?(p).\mathit{ack}!\langle p\rangle, \mathit{ack} \mapsto \bullet, \\
& \quad n@\mathit{client} \mapsto \bullet\}, \mathit{echo}!\langle n@\mathit{client}\rangle
\end{aligned}$$

Le serveur évolue de la manière suivante:

$$\begin{aligned}
\ldots \xrightarrow{\text{REDOUT}^*}\ & \mathit{echo_d}, \{\mathit{echo}\}, \{\mathit{ack}, n@\mathit{client}\}, \{\mathit{echo} \mapsto *\mathit{echo}?(p).\mathit{ack}!\langle p\rangle, \mathit{ack} \mapsto \bullet, \\
& \quad n@\mathit{client} \mapsto \bullet\}, \mathit{ack}!\langle n@\mathit{client}\rangle \\
\xrightarrow{\text{QUEUEOUT}}\ & \mathit{echo_d}, \{\mathit{echo}\}, \{\mathit{ack}, n@\mathit{client}\}, \{\mathit{echo} \mapsto *\mathit{echo}?(p).\mathit{ack}!\langle p\rangle, \\
& \quad \mathit{ack} \mapsto \mathit{ack}!\langle n@\mathit{client}\rangle, n@\mathit{client} \mapsto \bullet\}, \varnothing
\end{aligned}$$

La communication distante dans l'autre sens donne:

$$\begin{aligned}
\xrightarrow{\text{REMXMIT}}\ & \mathit{client}, \{\mathit{ack}, n\}, \{\mathit{echo}\}, \{\mathit{ack} \mapsto \bullet, n \mapsto \bullet, \mathit{echo} \mapsto \bullet\}, \mathit{ack}?(y).0 \parallel \langle \mathit{ack}, n@\mathit{client}\rangle \parallel \\
& \mathit{echo_d}, \{\mathit{echo}\}, \{\mathit{ack}, n@\mathit{client}\}, \{\mathit{echo} \mapsto *\mathit{echo}?(p).\mathit{ack}!\langle p\rangle, \\
& \quad \mathit{ack} \mapsto \bullet, n@\mathit{client} \mapsto \bullet\}, \varnothing \\
\xrightarrow{\text{REMRCV}}\ & \mathit{client}, \{\mathit{ack}, n\}, \{\mathit{echo}\}, \{\mathit{ack} \mapsto \bullet, n \mapsto \bullet, \mathit{echo} \mapsto \bullet\}, \mathit{ack}?(y).0 :: \mathit{ack}!\langle n\rangle \parallel \\
& \mathit{echo_d}, \{\mathit{echo}\}, \{\mathit{ack}, n@\mathit{client}\}, \{\mathit{echo} \mapsto *\mathit{echo}?(p).\mathit{ack}!\langle p\rangle, \\
& \quad \mathit{ack} \mapsto \bullet, n@\mathit{client} \mapsto \bullet\}, \varnothing
\end{aligned}$$

Le client finit alors son évolution en consommant l'acquittement:

$$\begin{aligned}
\ldots \xrightarrow{\text{QUEUEIN}}\ & \mathit{client}, \{\mathit{ack}, n\}, \{\mathit{echo}\}, \{\mathit{ack} \mapsto \mathit{ack}?(y).0, n \mapsto \bullet, \mathit{echo} \mapsto \bullet\}, \mathit{ack}!\langle n\rangle \\
\xrightarrow{\text{REDOUT}}\ & \mathit{client}, \{\mathit{ack}, n\}, \{\mathit{echo}\}, \{\mathit{ack} \mapsto \bullet, n \mapsto \bullet, \mathit{echo} \mapsto \bullet\}, 0
\end{aligned}$$

3.4 Une implémentation répartie

Nous décrivons à présent une implémentation de la machine abstraite répartie DCP sur une plate-forme middleware typique. Il est nécessaire d'effectuer un raffinement supplémentaire pour faire correspondre les abstractions fournies par la théorie de la programmation concurrente et répartie comme les noms de canaux, processus et domaines aux structures de données appropriées au contexte des plates-formes middleware comme les identificateurs, références, contextes de nommage ou objets implantant des protocoles ou des sessions. Le raffinement est présenté en utilisant l'exemple de l'ORB JONATHAN comme support pour la transparence des communications réseau.

Le langage d'entrée permet au programmeur de spécifier des noms biens connus pour que les différents interprètes puissent se découvrir lors de l'initialisation du système.

$m, n, u, v \ldots$	$\in$	Name	Nom de canal	$a, b, \ldots$	$\in$	DName	Nom de machine
$P, Q, \ldots$	$\in$	Pr	Processus	$\aleph$	$\in$	$\mathsf{Msg} = \mathsf{Name} \times \mathsf{Name}^*$	Message

$$\begin{array}{lcll} \mathsf{pickle} & : & \mathsf{DName} \times \mathsf{Name} \longrightarrow \mathsf{Name} & \mathsf{unpickle}(\mathsf{pickle}(a, x)) = (a, x) \\ \mathsf{unpickle} & : & \mathsf{Name} \longrightarrow \mathsf{DName} \times \mathsf{Name} & \mathsf{pickle}(\mathsf{unpickle}(x)) = x \end{array}$$

$\mathcal{U}$	$\in$	$\mathsf{LNames} = \mathsf{Name}^*$	Noms locaux
$\mathcal{V}$	$\in$	$\mathsf{RNames} = \mathsf{Name}^*$	Noms distants
$\mathcal{H}$	$\in$	$\mathsf{Heap} = \mathsf{Name} \mapsto \mathsf{Queue}(\mathsf{Pr})$	Tas de messages
$\mathcal{R}$	$\in$	$\mathsf{RQueue} = \mathsf{Queue}(\mathsf{Pr})$	File d'exécution
S	$\in$	$\mathsf{AM} = \mathsf{DName} \times \mathsf{LNames} \times \mathsf{RNames} \times \mathsf{Heap} \times \mathsf{RQueue}$	Etat de la machine abstraite
N	$\in$	$\mathsf{Network} = \mathsf{AM}^* \times \mathsf{Msg}^*$	Réseau

PRL
$$\frac{}{a, \mathcal{U}, \mathcal{V}, \mathcal{H}, P \mid Q :: \mathcal{R} \longrightarrow a, \mathcal{U}, \mathcal{V}, \mathcal{H}, P :: \mathcal{R} :: Q}$$

RES
$$\frac{u \notin \mathcal{U} \cup \mathcal{V} \quad \mathcal{U}' = \mathcal{U} \cup \{u\}}{a, \mathcal{U}, \mathcal{V}, \mathcal{H}, (\nu\ n)\ P :: \mathcal{R} \longrightarrow a, \mathcal{U}', \mathcal{V}, \mathcal{H}\ \{u \mapsto \bullet\}, \{u/n\}P :: \mathcal{R}}$$

REDIN
$$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = x!\langle \tilde{y} \rangle :: wr \quad \tilde{y} \in \mathcal{U} \cup \mathcal{V}}{a, \mathcal{U}, \mathcal{V}, \mathcal{H}, x?(\tilde{z}).P :: \mathcal{R} \longrightarrow a, \mathcal{U}, \mathcal{V}, \mathcal{H}\ \{x \mapsto wr\}, \{\tilde{y}/\tilde{z}\}P :: \mathcal{R}}$$

REDOUT
$$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = x?(\tilde{z}).Q :: rd \quad \tilde{y} \in \mathcal{U} \cup \mathcal{V}}{a, \mathcal{U}, \mathcal{V}, \mathcal{H}, x!\langle \tilde{y} \rangle :: \mathcal{R} \longrightarrow a, \mathcal{U}, \mathcal{V}, \mathcal{H}\ \{x \mapsto rd\}, \mathcal{R} :: \{\tilde{y}/\tilde{z}\}\ Q}$$

REDREPL
$$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = x!\langle \tilde{y} \rangle :: wr \quad \tilde{y} \in \mathcal{U} \cup \mathcal{V}}{a, \mathcal{U}, \mathcal{V}, \mathcal{H}, *x?(\tilde{z}).P :: \mathcal{R} \longrightarrow a, \mathcal{U}, \mathcal{V}, \mathcal{H}\ \{x \mapsto wr\}, *x?(\tilde{z}).P :: \mathcal{R} :: \{\tilde{y}/\tilde{z}\}\ P}$$

REDOUT*
$$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = *x?(\tilde{z}).Q :: rd \quad \tilde{y} \in \mathcal{U} \cup \mathcal{V}}{a, \mathcal{U}, \mathcal{V}, \mathcal{H}, x!\langle \tilde{y} \rangle :: \mathcal{R} \longrightarrow a, \mathcal{U}, \mathcal{V}, \mathcal{H}\ \{x \mapsto rd :: *x?(\tilde{z}).Q\}, \mathcal{R} :: \{\tilde{y}/\tilde{z}\}\ Q}$$

QUEUEIN
$$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = rd}{a, \mathcal{U}, \mathcal{V}, \mathcal{H}, x?(\tilde{z}).P :: \mathcal{R} \longrightarrow a, \mathcal{U}, \mathcal{V}, \mathcal{H}\ \{x \mapsto rd :: x?(\tilde{z}).P\}, \mathcal{R}}$$

QUEUEOUT
$$\frac{x \in \mathcal{U} \cup \mathcal{V} \quad \mathcal{H}(x) = wr \quad \tilde{y} \in \mathcal{U} \cup \mathcal{V}}{a, \mathcal{U}, \mathcal{V}, \mathcal{H}, x!\langle \tilde{y} \rangle :: \mathcal{R} \longrightarrow a, \mathcal{U}, \mathcal{V}, \mathcal{H}\ \{x \mapsto wr :: x!\langle \tilde{y} \rangle\}, \mathcal{R}}$$

QUEUEREPL
$$\frac{x \in \mathcal{U} \quad \mathcal{H}(x) = rd}{a, \mathcal{U}, \mathcal{V}, \mathcal{H}, *x?(\tilde{z}).P :: \mathcal{R} \longrightarrow a, \mathcal{U}, \mathcal{V}, \mathcal{H}\ \{x \mapsto rd :: *x?(\tilde{z}).P\}, \mathcal{R}}$$

REMXMIT
$$\frac{\begin{array}{c} x \in \mathcal{V} \quad \mathcal{H}(x) = x!\langle y_1, \ldots, y_s \rangle :: wr \\ (z_i)_{1 \le i \le s} = (\mathsf{pickle}(a, y_i) \text{ si } y_i \in \mathcal{U}\ ;\ y_i \text{ si } y_i \in \mathcal{V})_{1 \le i \le s} \end{array}}{a, \mathcal{U}, \mathcal{V}, \mathcal{H}, \mathcal{R} \longrightarrow a, \mathcal{U}, \mathcal{V}, \mathcal{H}\ \{x \mapsto wr\}, \mathcal{R}\ \parallel\ \langle x, \tilde{z} \rangle}$$

REMRCV
$$\frac{\begin{array}{c} \mathsf{unpickle}(x) = (a, x') \quad x' \in \mathcal{U} \quad \mathcal{V}' = \mathcal{V} \cup (T_i)_{1 \le i \le s} \\ (z_i, T_i)_{1 \le i \le s} = \begin{pmatrix} (y_i', \emptyset) & \text{si} & \mathsf{unpickle}(y_i) = (a, y_i'), y_i' \in \mathcal{U} \\ (y_i, \{y_i\}) & \text{si} & \mathsf{unpickle}(y_i) = (b, \_), b \neq a \end{pmatrix}_{1 \le i \le s} \end{array}}{\langle x, y_1, \ldots, y_s \rangle\ \parallel\ a, \mathcal{U}, \mathcal{V}, \mathcal{H}, \mathcal{R} \longrightarrow a, \mathcal{U}, \mathcal{V}', \mathcal{H}\ \{v \mapsto \bullet\}_{v \in \mathcal{V}' \setminus \mathcal{V}}, \mathcal{R} :: x'!\langle \tilde{z} \rangle}$$

FIG. *3.3* – **La machine abstraite DCP : définitions et règles de réduction.**

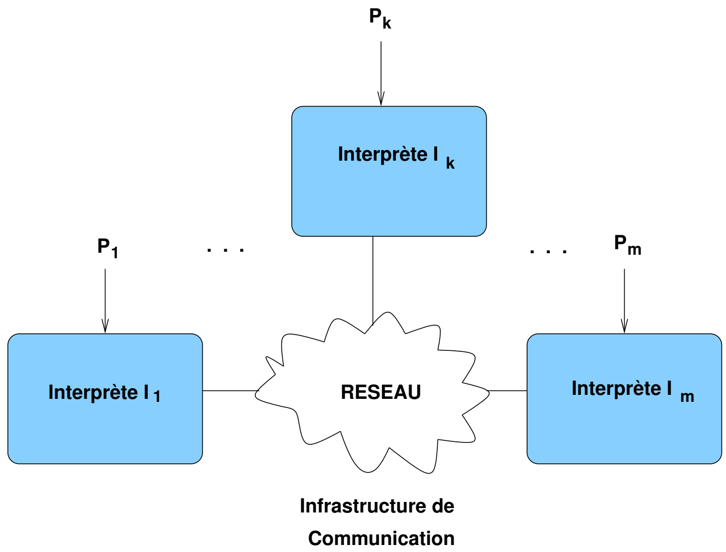


FIG. 3.4 – **Architecture d'ensemble du run-time DCP : le programme $a_1[P_1] \parallel \ldots \parallel a_m[P_m]$ est exécuté par m interprètes $I_1, \ldots, I_m$, l'interprète I_k exécutant le programme P_k avec $1 \leq k \leq m$.**

Du point de vue de l'implantation, l'opérateur $\parallel$ du modèle peut s'interpréter comme traduisant de façon naturelle la distribution physique des domaines. Aussi, le run-time réparti qui implémente le modèle DCP prend la forme d'un certain nombre d'interprètes s'exécutant en parallèle sur des sites identiques ou différents reliés par une infrastructure de communication. Chaque interprète, écrit en JAVA, exécute les processus DCP contenus dans le domaine qu'il implémente (cf. figure 3.4). Un interprète est composé d'un *moteur d'exécution* chargé de l'exécution des règles de réduction locales de la spécification formelle et d'un *gestionnaire de communications* qui implémente les primitives de communication distante.

3.4.1 Le moteur d'exécution

Une classe principale `DCPMachine` encapsule l'état de l'interprète (cf. figure 3.6), soit les structures $\mathcal{U}$, $\mathcal{V}$, $\mathcal{H}$, et $\mathcal{R}$ décrites dans la spécification formelle, et effectue la liaison avec le gestionnaire de communications. Ces structures sont initialisées à partir de l'arbre de syntaxique abstrait produit lors de la phase d'analyse syntaxique d'un programme DCP. Pour faciliter la mise au point, un front-end graphique permet au programmeur de visualiser directement l'état complet de chaque interprète, de naviguer à travers l'historique des réductions déjà effectuées ou d'exécuter un programme DCP en mode pas-à-pas. La figure 3.5 représente quelques réductions obtenues à partir de la configuration $a[((\nu s)\ s!\langle y\rangle \mid s?(u).u!\langle v\rangle \mid x!\langle z\rangle]$.

3.4.2 Le gestionnaire de communications

Pour assurer la transparence des communications réseau du calcul DCP, nous utilisons un framework minimal appelé JONATHAN [55, 89, 135] permettant la conception d'ORB hautement flexibles où les objets interagissent de manière transparente par invocation de méthode à distance sur des interfaces bien définies. Nous rappelons succinctement les principes d'architecture de JONATHAN, déjà brievement évoqués au chapitre 2, avant de décrire l'organisation du gestionnaire de communications.

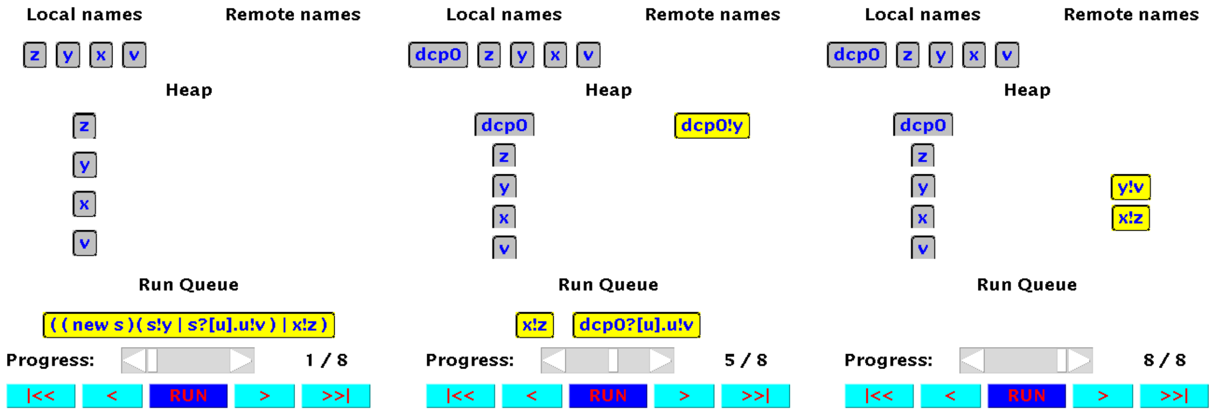


FIG. *3.5* – **Le front-end graphique.**

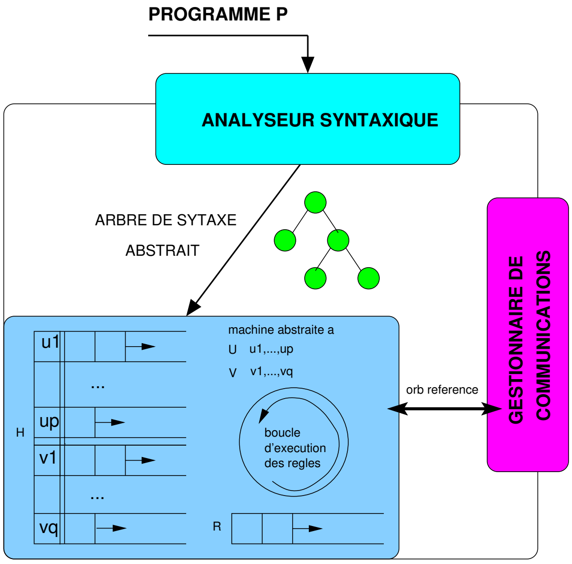


FIG. 3.6 – **Structure de l'interprète.**

Chaque règle de réduction de la machine abstraite est implémentée par le corps d'une méthode sans argument dont l'implémentation suit rigoureusement la spécification et qui retourne `true` si la règle est applicable compte tenu de l'état courant de la machine. La stratégie d'exécution choisie est gloutonne, la première règle applicable étant immédiatement déclenchée. D'autres politiques d'ordonnancement sont facilement implémentées en surchargeant la méthode `DCPMachine.run()`.

JONATHAN: un middleware adaptable

L'originalité de JONATHAN par rapport à des plates-formes à objets répartis classiques comme CORBA ou JAVA RMI est de fournir un cadre minimal pour la construction de différents types de middleware appelés *personnalités*. Les concepts clés sont ceux de *liaison* [3] et de *communications flexibles*. [4]. Cette

[3]Créer de nouvelles liaisons signifie instaurer des chemins d'accès et de communication entre différents composants d'un système réparti. De nombreuses sémantiques de liaison sont disponibles: liaisons mobiles, persistantes, avec une garantie de QoS, etc. Un framework d'ORB adaptable doit disposer de primitives pour définir des liaisons de sémantiques variées et les combiner de manière flexible.

[4]En utilisant certains principes architecturaux comme la séparation du « marshalling » de l'implémentation des protocoles ou la gestion des fils d'exécution de celle des ressources, JONATHAN permet par exemple d'introduire dynamiquement de nouveaux protocoles ou de contrôler le niveau de multiplexage des ressources.

flexibilité nous a permis d'implémenter le gestionnaire de communications DCP directement au-dessus des abstractions TCP/IP fournies par la distribution de JONATHAN afin de disposer d'une pile de protocoles réseau minimale allégeant ainsi le traitement des messages.

JONATHAN est construit autour d'un nombre minimal d'abstractions (cf. figure 3.7):

- Un *identificateur* est une notion générique de nom qui désigne un objet de façon unique dans un contexte de nommage. La sémantique de l'identificateur dépend du contexte de nommage et peut être répartie, mobile, persistante, etc. Plus généralement, une *référence* encapsule un ensemble d'identificateurs.
- Un *contexte de nommage* fournit des services de création et de gestion de noms. Il assure notamment la propriété de désignation unique: chacun des noms qu'il contrôle désigne un objet de manière non-ambiguë.
- Une *usine à liaison (« binder »)* est un contexte de nommage qui assure également la fonction d'établissement de liaisons. Étant donné un nom géré par un binder, ce dernier est capable d'établir un chemin d'accès vers l'objet ainsi désigné.

Ces définitions offrent une vision uniforme et générique de la notion de liaison et séparent clairement l'identification d'un objet (chaîne de nommage) de son accès (chaîne de liaison):

- Dans un contexte de nommage `nc`, la méthode `nc.export(o)` permet de créer un nouveau nom pour un objet `o` géré par `nc`. Son application répétée pour exporter ce nom vers d'autres contextes de nommage conduit à la création de chaînes de références.
- La création d'un chemin d'accès vers un objet `o` désigné par l'identificateur `id` est réalisée en invoquant la méthode `id.bind()` qui retourne un objet (proxy ou surrogate) pouvant être directement utilisé pour communiquer avec `o`.

JONATHAN dispose également d'un framework inspiré du x-kernel [80, 137] pour définir de manière flexible des piles protocolaires comme composition de micro-protocoles:

- Un objet *protocole* est un binder qui représente un protocole réseau. Il peut donner accès à des objets particuliers appelés *sessions*, plus connus dans la littérature sous le nom de « *endpoint* », et contrôle leur désignation en gérant des *identificateurs de session*. Un objet protocole gère des *connexions*, représentation de bas niveau d'un canal de communication spécifique à chaque protocole.
- Une *session* est un canal de communication logique dynamiquement créé par un objet protocole et contenant typiquement des connexions. Il dispose d'interfaces hautes (`Session_High`) et basses (`Session_Low`) pour faire descendre ou remonter les messages dans une pile protocolaire qui est représentée par une pile d'objets session [5].
- Un *identificateur de session* désigne une session qui a été exportée. Sa structure interne est spécifique au protocole associé. Cet identificateur est typiquement créé lorsqu'un objet côté serveur est exporté. Après transmission sur le réseau, l'identificateur peut être utilisé pour établir un canal de communication côté client.

[5]De manière générale, les objets de type `Session_Low` ont été exportés auprès d'objets protocole. Les objets de type `Session_High` sont plutôt des proxy pour les objets `Session_Low`. Ils sont retournés par une méthode `bind()`. Dans toute la suite, ces représentants d'un canal de communication distant seront désignés sous le nom de *surrogate*.

Une autre abstraction utile est le *graphe de protocoles*, graphe orienté sans cycle dont les nœuds sont des objets protocole. Il décrit le chemin suivi par les messages dans la pile protocolaire lorsqu'ils sont envoyés ou reçus [6]. On peut aussi citer les « *marshallers* » (resp. « *unmarshallers* ») ou représentation de haut niveau et indépendante d'un codage particulier des messages envoyés (resp. reçus) dans une pile de protocoles.

Abstraction	**Interfaces JONATHAN**
Identificateur	`Identifier`
Référence	`Reference`
Contexte de nommage	`NamingContext`
Protocole	`Protocol` (`TCPIPProtocol`, `GIOPProtocol`...); `ProtocolGraph`
Session	`Session_High` ; `Session_Low`
Identificateur de sessions	`SessionIdentifier`
Messages	`Marshaller` ; `Unmarshaller`

FIG. *3.7* – **Principales abstractions et interfaces de JONATHAN.**

Architecture du gestionnaire de communications

Le gestionnaire de communications est structuré en trois couches de gestion de noms (cf. figure 3.8):

- La *couche « applicative »* manipule les noms du calcul DCP. Ces noms sont représentés dans le run-time par des objets de type `Name`. Ils sont gérés par un binder `DCPORB`, interface du gestionnaire de communications vis à vis des autres composants d'un interprète DCP. Les sous-classes de `Name` capturent les deux types de noms manipulés par une machine abstraite: les noms locaux et les références réseau.
  1. Les nom locaux sont raffinés dans des objets de type `LocalName`. Ils implémentent également l'interface `Session_Low`, car formant le dernier élément d'une chaîne de liaison. Ces noms seront exportés auprès du contexte de nommage qui les gère (`DCPORB`).
  2. Les références réseau sont raffinées dans des objets de type `RemoteName`. Ces objets contiennent le nom de machine DCP ayant créé la référence, une version locale du nom valide seulement à l'intérieur de cette machine et une référence `DCPReference` contenant des informations pour l'établisssement de liaisons DCP/TCP/IP. Le binder `DCPORB` associe dans une table T_{orb} chaque nom local (objet `LocalName`) avec sa référence réseau (surrogate `RemoteName`). Il maintient également une référence vers le service de nommage.
- La *couche de gestion des références* sert d'interface entre les noms du calcul et les sessions TCP/IP. Les données de liaison sont regroupées dans un certain nombre d'identificateurs `DCPIdentifier` gérés par le binder `DCPBinder`. Ces identificateurs contiennent un tableau d'octets constituant la représentation « sur-le-fil » d'une référence réseau DCP. Un `DCPIdentifier` désigne un objet « applicatif » `LocalName` qui vient d'être exporté. Il contient un nom de machine DCP, la représentation locale du nom, et un identificateur de session TCP/IP.

[6] Un objet session peut être exporté en appliquant la méthode `export()` à la racine d'un graphe de protocoles: les méthodes `export()` de chaque noeud du graphe sont appelées récursivement, et un identificateur de session est retourné pour désigner la session exportée.

- La *couche de transport* permet l'envoi et la réception de messages sur des connections TCP/IP. Les sessions TCP sont désignées par des identificateurs `TcpIpSessionIdentifier` contenant un nom de machine physique et un numéro de port.

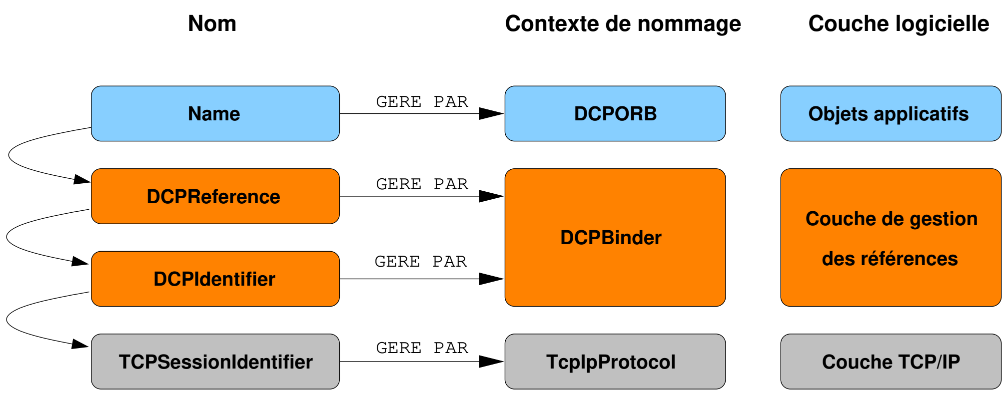


FIG. *3.8* – **Structure de la couche de communication.**

La couche de communication est initialisée récursivement par instanciation des différents binders lors de la création d'une nouvelle `DCPMachine`.

Création de nouveaux noms

La création d'un nouveau nom de canal DCP se traduit simplement par la création d'une nouvelle instance d'un objet `LocalName`.

Implémentation de l'extension de portée

Les opérations pickle et unpickle de la spécification formelle peuvent en partie être implémentées à l'aide de la méthode `export` du framework JONATHAN. La publication sur le réseau d'un nom `n` créé localement dans le domaine d s'effectue par l'appel `orb.export(n)` qui exporte `n` auprès de l'instance courante `orb` du binder `DCPORB`. Un surrogate de type `RemoteName` est retourné et représente pickle(d, n). Le nom n est ensuite exporté auprès des couches inférieures (cf. figure 3.9):

- La méthode `DCPORB.export : LocalName` $\rightarrow$ `RemoteName` renvoie un surrogate utilisable en tout point du réseau à partir d'un nom local `ln`. Selon l'état de la table T_{orb}, deux cas sont possibles: si `ln` possède déjà un surrogate, ce dernier est retourné; sinon, une nouvelle instance `rn` d'un `RemoteName` est créée à partir de `ln` et de la machine DCP courante, les informations de liaison DCP/TCP/IP étant initialisées à vide. Le nom local `ln` est ensuite exporté auprès du binder de niveau inférieur (`DCPBinder`):

```
DCPBinder binder;
...
Identifier id = binder.export(ln);
```

On récupère alors un `DCPIdentifier` `id`, lui-même encapsulé dans une `DCPReference` `ref`. La partie liaison de `rn` est mise à jour avec `ref` et l'entrée `ln` $\mapsto$ `rn` est consignée dans la table T_{orb}. Le surrogate `rn` est enfin retourné par la méthode `export`.

- La méthode `DCPBinder.export : LocalName` $\rightarrow$ `Identifier` renseigne les informations de liaison DCP/TCP/IP nécessaires à la couche applicative. Elle commence par créer une nouvelle instance d'un graphe de protocoles puis exporte le nom local `ln` auprès de la couche de transport:

```
TcpIpProtocol tcp_protocol;
...
ProtocolGraph p_graph = tcp_protocol.newProtocolGraph();
TcpIPSessionIdentifier tcp_session_id =
  (TCPSessionIdentifier) p_graph.export(ln);
```

  L'effet obtenu est la création d'une session serveur associée à `ln` pour écouter les communications clientes sur ce canal. La méthode `DCPBinder.export` renvoie alors un `DCPIdentifier` contenant le nom local, le nom de la machine DCP courante et les informations de liaison obtenues de la couche TCP/IP.

- La méthode `TcpIpProtocol.export : LocalName` $\rightarrow$ `Identifier` est appelée par le graphe de protocoles et informe la couche TCP/IP que le nom local `ln` passé en paramètre est prêt à recevoir des messages du réseau: on a création d'une nouvelle session serveur TCP/IP s à l'écoute de connections clientes éventuelles. s admet `ln` comme session de niveau supérieur: tout message reçu du réseau par s sera donc directement retransmis vers `ln` pour y être traité. La session s est identifiée par un `TcpIPSessionIdentifier` contenant un nom de machine physique et un numéro de port.

Lors de l'initialisation du système, un service de nommage permet la découverte respective des différents interprètes DCP. Tous les noms locaux contenus dans $\mathcal{U}$ (récepteurs libres) sont enregistrés auprès du service de nommage pour les rendre utilisables depuis des machines DCP distantes. Ces noms sont d'abord exportés auprès du gestionnaire de communications. Le surrogate retourné est alors enregistré dans le service de nommage.

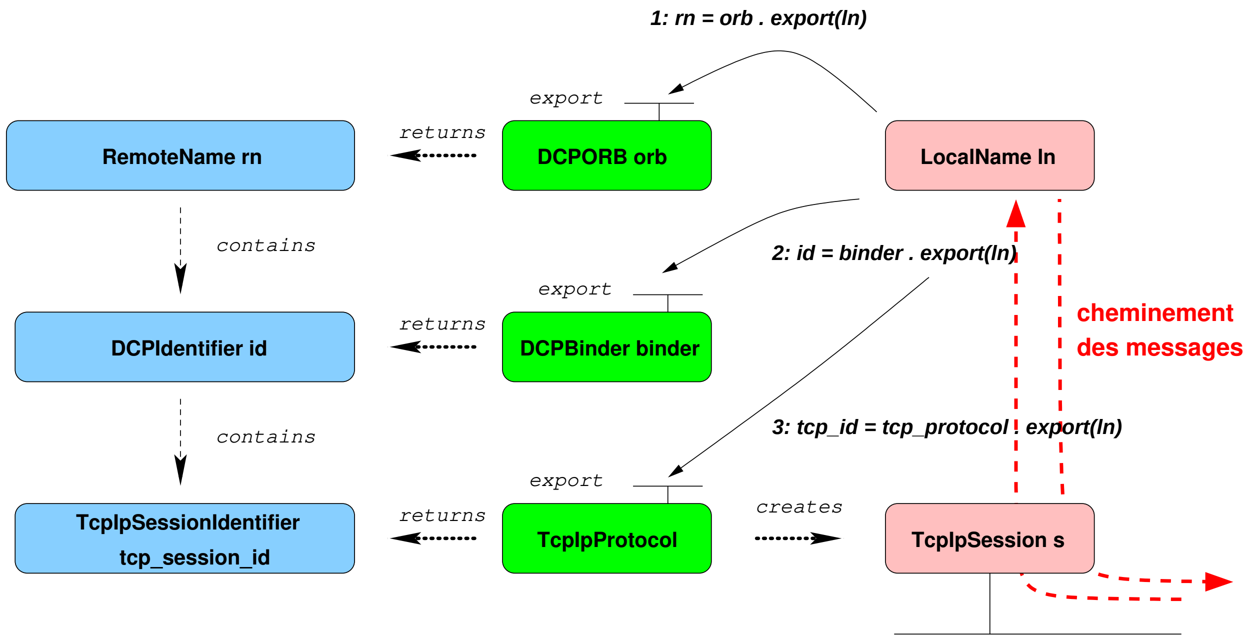


FIG. 3.9 – **Publication d'un nom local sur le réseau.**

```
public class DCPMachine extends Thread {
  ...
  Context naming_service;
  DCPORB orb;

  public DCPMachine(String machineName, Hashtable U, Hashtable V, Proc p) {
     ...
     naming_service = getNamingService();
     orb = (DCPORB) getDCPORB();
     orb.setParameters(machineName, naming_service);
     Enumeration e = U.elements();
     while (e.hasMoreElements()) {
        LocalName ln = (LocalName) e.nextElement();
        RemoteName ln_surrogate = orb.export(ln, null);
        naming_service.rebind(ln_surrogate.getName(), ln_surrogate);
      }
      ...
  }
  ...
}
```

Implémentation de la communication distante

Jusqu'ici, nous avons considéré et implémenté une référence réseau comme une capacité de communication sur un canal distant. Pour effectuer une véritable communication, il faut aller plus loin en créant une nouvelle liaison établissant un chemin d'accès vers l'objet qui représente le canal distant. Cette opération est réalisée à l'aide de la méthode `bind` du framework JONATHAN.

Lors d'une émission distante, tous les canaux de $\mathcal{V}$ sont examinés jusqu'à sélectionner le premier qui contienne dans le tas $\mathcal{H}$ un ou plusieurs processus en attente d'émission – ce qu'exprime la condition $x \in \mathcal{V} \quad \mathcal{H}(x) = x!\langle y \rangle :: wr$ dans la règle REMXMIT de la machine abstraite. Deux cas peuvent se présenter:

- Si $y \in \mathcal{U}$, y est un nom local et doit donc être exporté auprès de l'ORB avant d'être envoyé sur le réseau:

  ```
  RemoteName y_surrogate = orb.export((LocalName) y);
  ```

  Comme nous venons de le voir, cet appel crée un surrogate `y_surrogate` représentant y et une session TCP serveur écoutant sur un port TCP déterminé la connexion de clients souhaitant émettre sur y. Ce numéro de port et le nom de la machine physique associée sont conservés dans la partie de `y_surrogate` consacrée à la liaison.
- Si $y \in \mathcal{V}$, y est une référence réseau ayant déjà été exportée. Elle est implémentée par un objet de type `RemoteName` contenant des informations de liaison TCP/IP non vides:

  ```
  RemoteName y_surrogate = (RemoteName) y;
  ```

L'envoi de message proprement dit est réalisé par les deux appels suivants:

```
RemoteName x_chan = (RemoteName) orb.bind((RemoteName) x);
x_chan.output(y_surrogate);
```

La méthode `bind` établit la chaîne de liaison et retourne un objet `x_chan` directement utilisable pour communiquer sur le canal distant x: elle réalise récursivement des appels `bind` sur les binders de plus bas-niveau, ce qui instancie une pile d'objets session côté client. Une fois le chemin d'accès établi entre les deux interprètes DCP, la méthode `output` transmet y sur le réseau en faisant descendre un message contenant une version de y dans la pile d'objets session. Plus précisément:

1. La méthode `DCPORB.bind : RemoteName` $\rightarrow$ `Object` renvoie un objet `x_chan` sur lequel on peut effectuer des invocations de méthode pour envoyer des messages sur un canal distant `x` passé en paramètre.

   (a) Si `x` ne contient pas de données de liaison, celles-ci sont récupérées à partir du service de nommage: si `x` est un surrogate pour un canal $\texttt{x}_{\texttt{loc}}$ créé dans une machine `m`, les informations de liaison sont contenues dans le surrogate $\texttt{x}_{\texttt{surrogate}}$ de $\texttt{x}_{\texttt{loc}}$ enregistré dans le service de nommage lors de l'initialisation de m. Les identificateurs de liaison contenus dans $\mathtt{x}_{\mathtt{surrogate}}$ sont sérialisés puis reconstruits coté client. Pour chaque identificateur, sa méthode `encode` est invoquée pour construire une représentation « sur le fil » de l'identificateur, ensuite transmise sur le réseau. La reconstruction de l'identificateur utilise la méthode `decode` du contexte de nommage gérant l'identificateur (`DCPBinder`). On a en particulier création d'un nouvel identificateur de session TCP à partir d'un nom de machine physique et d'un numéro de port lus dans le flot d'octets:

```
tcp_session_id = tcp_protocol.newSessionIdentifier(host,port);
```

On passe alors à l'étape (b).

(b) On récupère dans `x` les identificateurs contenant les données de liaison, et on invoque leur méthode `bind` pour construire la liaison:

```
Identifier[] ids = x.getIdentifiers();
for (int i = 0; i < ids.length; i++) {
  Name result = (Name) ids[i].bind(ids);
  if (result != null)
    return result;
}
```

Le contrôle est alors transféré au binder `DCPBinder` qui effectue les actions suivantes:

```
Object bind(Identifier id) throws JonathanException {
   try {
     DCPClientIdentifier remote_id = (DCPClientIdentifier) id;
     TcpIpSessionIdentifier tcp_session_id =
       remote_id.tcp_session_id;
     String m = remote_id.machine_name;
     String x_loc = remote_id.channel_name;
     if (tcp_protocol.isLocal(tcp_session_id))
       return orb.getLocalName(m, x_loc);
     else
       return orb.newRemoteName(m, x_loc,
                                remote_id, tcp_session_id);
   } catch (Exception e) { ... } ;
}
```

A partir d'un identificateur de liaison `id`, le client peut récupérer une référence sur la session TCP/IP gérant le transport (`tcp_session_id`). Si cette session est gérée localement, le nom de canal référencé est un nom « applicatif » créé dans la machine courante, donc géré par l'instance locale du binder `DCPORB`. Ce nom est alors renvoyé, car constituant l'extrémité de la chaîne de liaison. Si la session TCP est gérée dans une machine distante m, on fait appel à l'ORB pour qu'il renvoie un objet `RemoteName` contenant les informations de liaison complètes: nom de machine m, version locale $\mathtt{x}_{\mathtt{loc}}$ du nom de canal et paramètres de la session distante.

2. Côté client, on dispose à présent d'un objet `x_chan` de type `RemoteName` représentant le récepteur distant $\mathtt{x}_{\mathtt{loc}}$ et directement utilisable pour lui transmettre des messages. C'est le rôle de la méthode `output : RemoteName` $\rightarrow$ `void` dont l'argument `arg` est la référence réseau `y_surrogate` transmise vers $\mathtt{x}_{\mathtt{loc}}$. L'envoi de message est délégué à la couche de gestion des références, puis à la couche de transport: `x_chan` contient un identificateur `tcp_session_id` d'une session TCP, utilisé pour créer une session TCP cliente `tcp_session` admettant comme session de niveau supérieur le surrogate `x_chan`. Cette opération traduit l'ouverture d'une connection TCP/IP vers la session TCP côté serveur identifiée par `tcp_session_id`:

```
tcp_session = tcp_session_id.bind();
```

   Ainsi, tous les messages reçus du réseau par l'objet session `tcp_session` seront remontés vers le surrogate `x_chan`. La référence réseau `arg` est alors encapsulée dans un message, puis transmise à la session `tcp_session` qui se chargera de l'envoyer sur le réseau via la connection TCP/IP ouverte précédemment:

```
Marshaller marshaller = marshaller_factory.newMarshaller();
marshaller.writeReference(arg.getRef());
tcp_session.send(marshaller);
tcp_session.close();
```

3. Côté serveur, le message va remonter la pile d'objets session. Le message reçu est transmis de la session TCP/IP vers sa session de niveau supérieur. Il s'agit d'un objet « applicatif » de type `LocalName` représentant le récepteur local, cible de la communication. On déballe alors du message un identificateur `id`, puis le nom « applicatif » transmis:

```
n = (Name) id.bind();
```

   Cette opération renvoie un surrogate pour l'argument du message, ou l'argument lui-même s'il s'agit d'un nom local. L'état de l'interprète est alors modifié, comme indiqué dans la règle REMRCV: un processus émetteur est inséré en fin de file d'exécution. Si `n` est une référence réseau, elle est ajoutée à celles déjà connues de la machine contenues dans $\mathcal{V}$. L'exécution de la machine DCP courante se poursuit alors par l'application de règles locales.

La figure 3.10 résume les différentes entités impliquées lors d'une communication entre deux machines DCP et indique le cheminement des messages dans les piles protocolaires.

3.5 Conclusions

3.5.1 Bilan

Dans ce chapitre, nous avons analysé dans le cadre d'une expérimentation préliminaire différentes étapes de formalisation, menant de la conception d'un modèle de programmation réparti construit autour

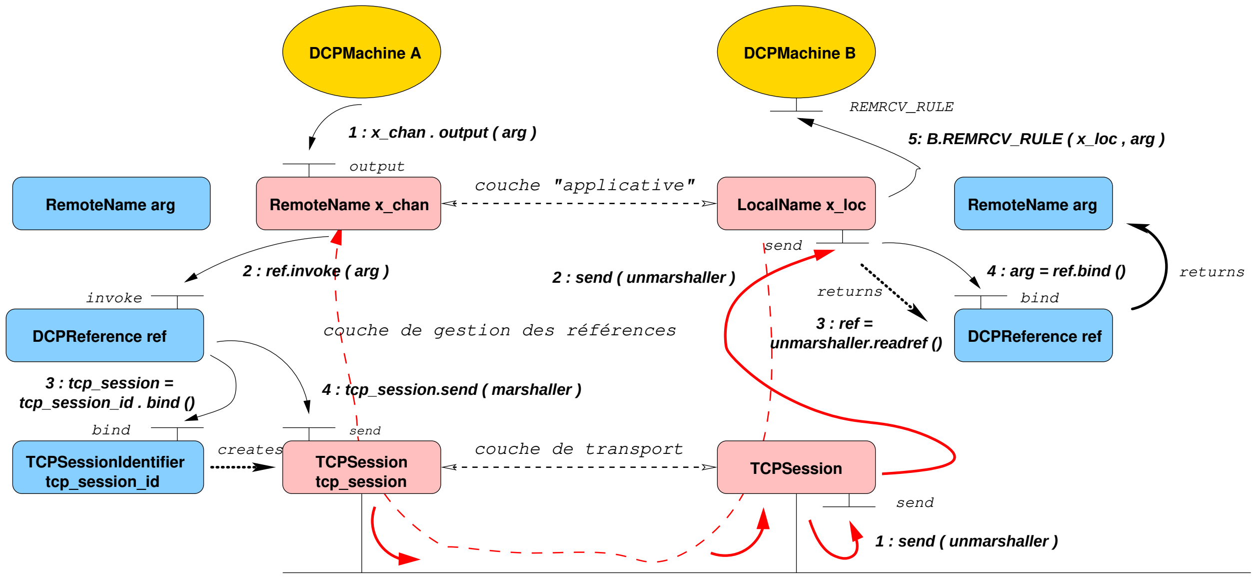


FIG. 3.10 – **Communication entre deux machines DCP : cheminement des messages.**

Etape de formalisation	Calcul	Machine abstraite	Run-time réparti
Modèle de nommage	Noms de canaux	Noms locaux	Noms locaux (objets `LocalName`)
		Références réseau	Noms exportés (surrogates) (objets `RemoteName`)
Création de noms	Opérateur ν	Création d'un nom local : règle RES	Instanciation d'un objet `LocalName`. Les nouveaux noms sont automatiquement exportés
Extension de portée	Equivalence structurelle $a[(\nu n)\ P] \equiv (\nu\ n)\ a[P]$	Opération pickle	`export()` sur les noms locaux
			Emballage / déballage des noms exportés
Communication inter-domaines	Règles SEND et RECEIVE	Opération pickle : règles REMXMIT et REMRCV	Création de nouvelles liaisons : `bind()` sur les noms exportés
			Envoi de message asynchrone sur des liaisons TCP

FIG. 3.11 – **Raffinements pour le nommage et la communication distante dans DCP.**

d'une forme très simple de domaine jusqu'à son implémentation répartie sur une plate-forme middleware courante. L'expérimentation confirme le bien fondé et la faisabilité de cette démarche pour construire des infrastructures réparties fiables qui soient correctement conçues. L'approche est centrée sur l'utilisation d'un modèle de programmation à la sémantique opérationnelle rigoureusement définie, et d'une implémentation, obtenue par raffinement du modèle et s'y conformant strictement.

Une extension simple, appelée DCP, du langage de programmation concurrente PICT avec des primitives de domaine et de communication distante a été introduite. Une implémentation de DCP a été formalisée sous forme de machine abstraite. Enfin, DCP a été implémenté par un run-time réparti, écrit en JAVA et intégré avec une plate-forme middleware typique, l'ORB JONATHAN, pour assurer la transparence des communication réseau. Pour chaque étape de formalisation de l'expérimentation, les formes de raffinement concernant le nommage et la communication distante sont résumées dans la figure 3.11.

3.5.2 Intérêt, limites, et prolongements

Valeur de l'expérimentation

L'expérimentation DCP a permis de tester et de valider la mise en oeuvre de cette démarche dans le cadre d'un modèle de programmation simple, en explorant la communication distante entre formes élémentaires de domaines, et son implantation dans un univers orienté-objets.

D'autre part, DCP est un calcul de processus formalisant à la fois son implantation sous forme de machine abstraite dans un contexte réparti, et étant intégré à une plate-forme à objets répartis (ORB). L'intérêt principal de cette expérimentation est donc de contribuer à réduire la divergence exposée au chapitre 2 entre théorie et pratique de la programmation répartie: cette expérience démontre qu'il est possible d'intégrer un cœur de machine virtuelle formelle pour calcul de processus – traditionnellement implémenté à l'aide de techniques de programmation fonctionnelle sophistiquées – avec une plate-forme à objets répartis. Cette combinaison au sein d'un même prototype rend possible l'interopérabilité avec des standards de facto du middleware comme CORBA ou JAVA RMI tout en conservant un cœur de machine très efficace. De plus, la compatibilité de la machine virtuelle avec le modèle formel d'exécution répartie constitue une garantie de fiabilité de la plate-forme. Cette fusion de deux approches de la répartition souvent opposées nous semble particulièrement importante pour la programmation d'applications réparties fiables dans des environnements hétérogènes.

Comparaison avec l'état de l'art

La machine abstraite DCP étend celle proposée par Turner pour PICT en incluant des primitives de communication distante. Elle semble être l'une des premières tentatives de formalisation d'une implémentation répartie par une machine abstraite distribuée. Les travaux les plus proches sont DITYCO, où la machine supporte l'exécution de threads multiples, mais est présentée uniquement dans un contexte non réparti. Dans NOMADIC PICT, l'objectif global est similaire, mais l'approche de formalisation est différente: un système réparti n'est pas formalisé par une machine abstraite réparti, mais par une traduction d'un langage de haut-niveau vers des primitives de bas niveau. Un travail similaire est PAN, implémentation en JAVA RMI du calcul des SAFE AMBIENTS, formalisé également à l'aide d'une machine abstraite répartie. Ce travail présente deux différences importantes avec DCP: (i) le modèle sous-jacent utilise des primitives de migration, plutôt que de communication; (ii) l'utilisation dans DCP de JONATHAN permet l'interopérabilité potentielle avec des applications ne venant pas uniquement du monde JAVA, par exemple en utilisant la personnalité CORBA de l'ORB.

Toutefois, de nombreux aspects restent ouverts à de futurs développements:

Limites de l'implémentation

On peut séparer la machine virtuelle DCP en trois composants principaux: un *analyseur* transformant le langage source en un code intermédiaire, un *moteur d'exécution* évoluant selon le contenu de cette représentation et un *gestionnaire de communications* reliant plusieurs machines virtuelles dans un contexte réparti. L'expérimentation soulève quelques points importants d'architecture de machine virtuelle:

1. Analyseur:
   - Quelle forme de représentation intermédiaire adopter? Cette représentation doit-elle être de haut niveau (JAVA), de bas niveau (JOCAML), ou coïncider avec l'arbre de syntaxe abstrait produit à partir du langage source (DCP)?
   - Quelle forme d'analyseur utiliser? Doit-il s'agir d'un interprète (DCP, JOCAML)? Faut-il inclure un compilateur qui génère la représentation intermédiaire ou qui la compile à la volée pour en optimiser l'exécution?
2. Moteur d'exécution:
   - Peut-on se contenter d'un seul thread langage (DCP)? La machine virtuelle doit-elle être multi-threadée (JOCAML, TYCO)? Quelle forme doit prendre la règle d'ordonnancement permettant de choisir le prochain thread à exécuter? Comment ces threads langage doivent-ils être mis en correspondance avec les threads système sous-jacents?
   - Quelle doit être la stratégie d'application des règles de réduction? Doit-on privilégier l'efficacité, viser certaines propriétés d'équité pour la machine, ou conserver une certaine flexibilité permettant de changer cette stratégie?
3. Gestionnaire de communications:
   - Que doit être une représentation efficace des canaux? Comment implémenter le routage des messages dans le cas où celui-ci est explicite/implicite? En particulier, comment faut-il implémenter les fonctions de transformation de noms pickle/ et unpickle? Quelle politique de gestion des références distantes est véritablement efficace?
   - Comment les primitives de mobilité du calcul doivent-elles être implémentées? Comment maintenir à jour les références lorsqu'elles désignent des entités susceptibles de migrer?
   - Peut-on construire de manière modulaire les piles protocolaires pour la communication distante?

L'expérimentation DCP fournit certaines réponses à ces questions. Ainsi, le run-time DCP diffère d'autres machines virtuelles plus élaborées (TYCO, JOCAML) composés d'un compilateur qui génère du byte-code et d'une machine virtuelle qui l'exécute. Dans DCP, on ne dispose pas de byte-code, l'interprète calculant directement sur la représentation abstraite produite par l'analyseur syntaxique. L'utilisation de JONATHAN permet une construction modulaire de piles de protocoles de communications.

De nombreuses optimisations restent cependant possibles: conception et implémentation d'un interpréteur de byte-code, implantation efficace des files de processus, conception d'un vérificateur de types plus élaboré, enrichissement de l'analyseur syntaxique pour prendre en compte un véritable langage de programmation au dessus du calcul DCP, programmation d'un ramasse-miettes réparti. . .Ces optimisations sont rendues nécessaires par le but même de l'expérimentation qui n'était pas de développer une machine virtuelle complète mais d'intégrer une machine abstraite formelle à la PICT avec une infrastructure gérant la répartition (ORB JONATHAN). DCP permet cependant de donner une idée de ce que serait une machine virtuelle élaborée, ayant une base formelle et intégrée avec les technologies middleware courantes.

L'ensemble de ces interrogations ne semble pas posséder pas de réponses générales, celles-ci dépendant largement du langage à implémenter. L'intérêt de l'expérimentation est également de soulever ces choix de conception, cruciaux pour le développement d'une machine virtuelle plus avancée. Ainsi, il semblerait utile de définir une forme de byte-code en raison des avantages suivants qui pourraient compenser la perte de performance due à l'utilisation d'un interprète : compacité du code, portabilité, conception claire et flexible, support pour la mobilité de code. . .Une telle machine pourrait également bénéficier de registres, l'interprétation des instructions du byte-code correspondant à des opérations simples sur ces registres.

Limites du modèle

Les domaines DCP sont purement passifs: pour disposer de domaines véritablement programmables comme ceux présentés dans le chapitre 2, des formes plus riches de contrôle sont nécessaires au niveau du modèle de programmation, par exemple pour créer de nouveaux domaines ou pour filtrer les messages entrant ou sortant d'un domaine. De même, le modèle DCP se limite à une topologie plate de domaines, et ne prend pas en compte la mobilité de processus ou la sécurité des communications. D'autre part, son modèle d'interaction repose sur la transparence des communications distantes: ceci est peu compatible avec un environnement de type « global computing » où la notion de barrière est omniprésente et devient observable au niveau des communications. Il semble également important d'étudier le passage à l'échelle dans des structures hiérarchiques de domaines.

Pour traiter l'ensemble de ces situations, un nouveau calcul plus expressif, fondé explicitement sur la notion de domaine est indispensable. L'objectif du chapitre suivant est précisément de présenter en détail un tel calcul appelé le M–calcul.

Prolongements

Le bénéfice principal de l'expérimentation DCP est de fournir un guide pour l'application de la démarche de formalisation par raffinement au cas d'un modèle de programmation adapté à la répartition à large échelle comme le M–calcul. Dans la suite de la thèse, l'expérimentation servira de fil directeur pour appliquer la démarche au cas du M–calcul, comme nous le verrons dans les trois chapitres suivants consacrés respectivement au modèle, à sa formalisation par une machine abstraite, et à son implémentation. En particulier, dans l'implémentation d'une machine virtuelle pour ce modèle, nous serons à nouveau confrontés aux choix de conception identifiés précédemment pour définir l'architecture de la machine. Les solutions découvertes lors de l'expérimentation DCP faciliteront le choix de techniques d'implantation adaptées aux spécificités de ce modèle de programmation, afin d'en obtenir une implémentation la plus efficace possible.

Chapitre 4

Le M–calcul : un modèle de programmation répartie orienté-domaines

Ce chapitre a pour but de présenter la première étape de la démarche de formalisation d'un système à base de domaines, suivant l'approche exposée au chapitre 2. Nous y décrivons brièvement un modèle de programmation répartie appelé M–calcul où les domaines sont manifestes au niveau de la syntaxe. Ce calcul de processus réparti servira de base aux implémentations décrites dans les chapitres suivants.

Après avoir rapidement rappelé l'insuffisance des modèles actuels pour les besoins fondamentaux du « global computing », nous décrivons les principales caractéristiques du modèle dans la section 4.1, puis en donnons la syntaxe et la sémantique opérationnelle dans les sections 4.2 et 4.3. Le pouvoir expressif du modèle est illustré à l'aide de quelques exemples dans la section 4.4, des résultats théoriques étant mentionnés dans la section 4.5. Enfin, certaines variations du calcul sont explorées dans la section 4.6.

Note : ce modèle de programmation, créé par Stefani et Schmitt, est décrit ici car il représente la base à partir de laquelle cette thèse a été réalisée. Il a fait l'objet de quelques aménagements dans le cadre de cette étude. Pour une description complète du M–calcul, le lecteur est invité à se référer à [153] ou [151].

4.1 Un modèle de programmation répartie à base de domaines

Parmi les exigences d'un modèle de programmation adapté au « global computing », on peut retenir son caractère implantable dans un contexte réparti, et un choix de primitives reflétant de près les caractéristiques d'un environnement distribué: asynchronisme de l'exécution et des communications; défaillances partielles; partitions du réseau en sous-systèmes multiples dont les éléments ont des comportements communs, par exemple d'administration et de gestion de ressources, de défaillances, ou de sécurité. Cette dernière exigence a mené à l'identification de la notion de domaine comme abstraction fondamentale possible et comme primitive d'un modèle de programmation pour le « global computing » (cf. chapitre 2).

Souvent appelée localité, cette notion se retrouve dans la plupart des calculs de processus répartis. Le join–calcul constitue une avancée importante par son souci d'implantabilité, son organisation hiérarchique des localités – unités de défaillance et de migration – ses primitives de synchronisation, et son routage transparent des messages. Pour disposer d'un modèle adapté de programmation répartie mobile, il paraît nécessaire d'aller au-delà du join–calcul dans trois directions:

- *Sémantique de localité programmable:* dans les calculs existants, chaque notion de localité est indissociable d'un protocole qui lui permet d'interagir avec son environnement selon une sémantique prédéfinie et non-modifiable: par exemple, pour la mobilité, les primitives `go` (join) ou `in` et `out` (ambients); pour les défaillances, la primitive `fail` (join). Dans un système réparti à large échelle, les formes de localités sont variées, une localité ayant souvent simultanément des sémantiques multiples. Il est donc souhaitable de pouvoir combiner à l'intérieur d'un même calcul des formes de domaines ayant des comportements différents, comme un modèle de défaillances et une politique de contrôle d'accès.
- *Contrôle de la communication et de la migration:* un calcul pour le « global computing » doit permettre d'exercer un contrôle fin sur la politique de communication et de migration d'un domaine, par exemple en filtrant les messages entrant et sortant, ou ayant la possibilité de définir une stratégie de routage particulière, et pas nécessairement implicite comme en join–calcul.
- *Liaison dynamique:* la ressource à laquelle un nom particulier sera lié n'est pas en général connue avant l'exécution, et dépend de la localité dans laquelle le programme s'exécute. Le programmeur doit pouvoir accéder à l'aide d'un même nom à un service dont l'implémentation dépend du site d'exécution. La liaison statique fournie par le join–calcul entre ressources et localités via les définitions est donc insuffisante. Une forme de liaison dynamique paraît alors nécessaire.

Le M–calcul conserve les principaux acquis du join–calcul comme une organisation arborescente des domaines ou la synchronisation des messages par des filtres dans des définitions. Il en diffère sur les points suivants:

- *L'introduction d'une notion de domaine programmable appelée cellule*. Chaque cellule est composée de deux parties: une *membrane* implante le protocole d'interaction du domaine avec son environnement. La membrane contrôle un *contenu* qui s'exécute à l'intérieur de la cellule. Membrane et contenu étant des processus du M–calcul, il devient possible de définir dans le calcul une sémantique arbitraire de domaine.
- *L'unification de la communication et de la migration*. Le M–calcul est un calcul à passage par valeur d'ordre supérieur: la migration de processus s'effectue par communication de formes gelées ou « *thunks* ». Ces processus dont l'exécution a été suspendue à l'aide d'un *opérateur de passivation* réifient dans un message le contrôleur et le contenu d'un domaine. L'acheminement des messages est un moyen-terme entre communication distante complètement implicite (join–calcul) et routage explicitement spécifié à l'aide de primitives de migration (calcul des ambients). Le routage du M–calcul est encore transparent, mais les messages sont interceptés à chaque fois qu'ils franchissent une frontière de cellule: une cellule est donc capable de superviser simultanément communication et migration.
- *Le support d'une forme de liaison dynamique*. En M–calcul, un même nom peut donner accès à différentes ressources situées dans des cellules distinctes: la liaison s'effectue seulement lorsque le message envoyé par le programmeur atteint sa destination finale.

Le M–calcul comprend également des réminiscences du calcul bleu [27, 48]: encapsulation directe du λ–calcul, identification entre messages et application fonctionnelle, et séparation entre définitions et λ-expressions.

Le M–calcul est donc une extension du join–calcul réparti qui combine les fonctionnalités suivantes: localités programmables, fonctions et processus d'ordre supérieur, communication (et mobilité) asynchrone transparente mais contrôlable, et liaison dynamique. La possibilité de définir dans le même calcul des localités avec des sémantiques différentes font du M–calcul un candidat intéressant pour un modèle de programmation adapté à l'univers du « global computing ».

4.2 Syntaxe

Les primitives du M–calcul, dont la syntaxe est résumée dans la figure 4.1, sont inspirées à la fois du λ–calcul avec appel par valeur et du join–calcul. Du premier, on récupère la λ-abstraction et l'application fonctionnelle. Du second, on retient la composition parallèle et les définitions de ressource, qui permettent la synchronisation des communications à l'aide de filtres de messages (*join-patterns*): le processus gardé P d'une définition $\langle r_1\widetilde{x_1} \mid \ldots \mid r_n\widetilde{x_n} = P\rangle$ ne sera activé que lorsque tous les récepteurs $r_1, \ldots, r_n$ contiendront des messages en attente. Le calcul ajoute en propre une primitive de *cellule* $a(P)[Q]$, qui étend les localités du join–calcul en les rendant programmables, et un opérateur de passivation `pass` qui autorise la migration de cellules par communication d'ordre supérieur. L'opérateur de restriction du π–calcul $\nu n.P$, utile pour créer de nouveaux noms n de ressource ou de cellule de portée P, et une primitive $([\mu = V]P, Q)$ de test conditionnel sur les noms manipulés, viennent compléter les opérateurs du calcul.

Les deux primitives $a(P)[Q]$ et $\mathtt{pass}_a$ sont responsables du pouvoir expressif du calcul. La principale innovation du M–calcul est d'introduire au niveau de la syntaxe une manifestation de domaine sous forme de localité programmable ou *cellule* dont le comportement de contrôle est clairement séparé du programme contrôlé. Une cellule est notée $a(P)[Q]$ où a est le nom de la cellule, P le processus de contrôle de la localité (également appelé *contrôleur* ou *membrane*) et Q le *contenu* de la cellule. Pour modéliser de façon réaliste les partitions d'un système réparti, l'organisation des cellules est hiérarchique.

La primitive $\mathtt{pass}_a$ permet à un processus situé dans la membrane d'une cellule a d'en bloquer complètement l'exécution. Cette instruction s'applique à une fonction de passivation à deux arguments qui attend des formes gelées (*thunk* ou processus de la forme $\lambda.P$) : un thunk représente la membrane de la cellule passivée dont le contenu est réifié par l'autre thunk. Cette primitive, conjuguée à la communication d'ordre supérieur, rend possible la mobilité de processus entre cellules.

Dans le M–calcul, la communication prend la forme d'échanges point-à-point et asynchrones de messages entre *ressources* (canaux de communication), ce qui reflète le mode principal d'interaction dans les réseaux à large échelle. Un message est représenté par une application fonctionnelle d'un des types suivants:

- Les messages de la forme $r\widetilde{V}$ où r est un nom de ressource (la cible du message) et $\widetilde{V}$ un tuple de valeurs (les arguments du message) sont des *messages locaux* qui ne franchissent pas la frontière d'une cellule. Ils sont utilisés pour la communication entre processus contenus dans une même cellule.
- Les messages de la forme $d.r\widetilde{V}$ où d désigne une cellule distante et r une ressource, sont utilisés pour la communication entre cellules. Si d est de la forme $\overline{a}$ (resp. $\underline{a}$), la ressource cible doit être située dans la membrane (resp. dans le contenu) de la cellule a. Ces messages explicitement adressés sont routés par le calcul, de cellule en cellule.

Les valeurs transmises peuvent être soit la valeur vide $()$, un nom u (de cellule, de ressource, ou de ressource avec adressage explicite), un tuple de valeurs ou une abstraction. On remarquera qu'il n'est pas possible d'effectuer une abstraction sur un nom de ressource apparaissant dans une définition. Par exemple, $\lambda x.\langle x \triangleright P\rangle$ n'est pas un processus bien formé.

Le M–calcul utilise différents types de noms qui sont regroupés dans la figure 4.1. Les principaux sont les *noms de ressource* définis par une restriction $\nu r.P$, les *variables* définies par une λ–abstraction, et les *noms de cellule*. On admet l'existence de trois ensembles infinis dénombrables de noms correspondants: Ref, Var et Cell. Nous emploierons les lettres $r, s, \dots, x, y, \dots$ et $a, b, \dots$ pour désigner leurs éléments respectifs. On suppose également donnés deux éléments $\mathbf{i}$ et $\mathbf{o}$ de Ref utilisés pour le routage des messages, ainsi qu'un élément ϵ de Cell pour désigner la cellule de plus haut niveau d'un système.

Nous utilisons les conventions suivantes: nous écrivons $\mathsf{fn}(P)$ pour l'ensemble des noms libres dans P. $\mathsf{dn}(J)$ désigne l'ensemble des noms définis par un filtre de messages J: $\mathsf{dn}(r_1\widetilde{x_1} \mid \dots \mid r_q\widetilde{x_q}) = \{r_1, \dots, r_q\}$. On note $\mathsf{dln}(P)$ l'ensemble des noms de ressource définis localement dans le processus P, sans se préoccuper des cellules qu'il peut contenir. On note enfin $\mathsf{cells}(P)$ l'ensemble des différentes cellules actives contenues dans P. La définition de ces différents ensembles est résumée dans la figure 4.2.

Les seuls véritables lieurs du modèle sont l'abstraction $\lambda x.P$ et la restriction $\nu n.P$. Une règle de réaction n'est pas un lieur pour les noms de ressource définis, contrairement au join–calcul: dans la définition $\langle r_1\widetilde{x_1} \mid \dots \mid r_q\widetilde{x_q} \triangleright P\rangle$, les r_i sont libres dans P. Bien entendu, les x_{ij} sont liés dans P.

Les autres notations utilisées sont habituelles: nous notons $\theta = \{t_1/u_1, \dots, t_q/u_q\}$ une substitution où le terme t_i remplace le nom u_i. On note $P\theta$ ou $P\{t_1/u_1, \dots, t_q/u_q\}$ l'image du terme P sous une telle substitution. Les tuples finis de termes $(t_1, \dots, t_q)$ sont notés $\widetilde{t}$. Lorsque les tuples $\widetilde{t}$ et $\widetilde{u}$ ont le même nombre d'éléments, on note alors $\{\widetilde{t}/\widetilde{u}\}$ pour la substitution $\{t_1/u_1, \dots, t_p/u_p\}$. On notera par abus $(PQ_1 \dots Q_n)$ pour $(\dots(PQ_1)\dots Q_n)$ et $\nu\tilde{x}.P$ et $\lambda\tilde{x}.P$ pour $\nu x_1.\dots\nu x_n.P$ et $\lambda x_1.\dots\lambda x_n.P$. On utilise aussi la notation $\lambda.P$ pour la forme gelée $\lambda x.P$, avec $x \notin \mathsf{fn}(P)$. Enfin, l'équivalence de deux processus P et Q à α-conversion près est notée $P =_\alpha Q$.

Un M-contexte, noté $P\{\cdot\}$, est un terme écrit en utilisant la même syntaxe que pour les termes du M–calcul, mais en y ajoutant une constante notée $\cdot$, le trou. Le fait de remplir le trou dans $P\{\cdot\}$ avec un terme Q du M–calcul résulte en un autre terme du calcul noté $P\{Q\}$. Les contextes d'évaluation du M–calcul sont des M-contextes qui obéissent à la grammaire suivante:

$$\mathbf{E} ::= \cdot \mid \mathbf{E}V \mid P\mathbf{E} \mid \nu n.\mathbf{E} \mid (\mathbf{E} \mid P) \mid a(P)[\mathbf{E}] \mid a(\mathbf{E})[P] \mid \epsilon[\mathbf{E}]$$

On notera que le trou n'apparaît pas sous une λ-abstraction.

4.3 Sémantique opérationnelle

La sémantique opérationnelle du M–calcul est définie dans un style chimique [14] en utilisant une équivalence structurelle $\equiv$ et une relation de réduction $\rightarrow$.

Syntaxe:

C	$::=$		Configuration
		$\epsilon[P]$	Cellule racine
	$\mid$	$\nu n.C$	Restriction
P	$::=$		Processus
		$\mathbf{0}$	Processus inerte
	$\mid$	V	Valeur
	$\mid$	$\mathbf{a}(P)[P]$	Cellule
	$\mid$	$(P \mid P)$	Composition parallèle
	$\mid$	(PP)	Application
	$\mid$	$\nu n.P$	Restriction
	$\mid$	$([\mu = V]P, P)$	Test conditionnel
	$\mid$	$\langle D \rangle$	Définition
	$\mid$	$\texttt{pass}_{\mathbf{a}}\ V$	Passivation
V	$::=$		Valeur
		$()$	Valeur vide
	$\mid$	u	Nom
	$\mid$	d	Cible de message
	$\mid$	$(V, \ldots, V)$	Tuple de valeurs
	$\mid$	$\lambda x.P$	Abstraction
D	$::=$		Définition
		$\top$	Définition vide
	$\mid$	$J \triangleright P$	Règle de réaction
	$\mid$	$D; D$	Composition
J	$::=$		Filtre de messages
		$r\widetilde{x}$	Message
	$\mid$	$J \mid J$	Synchronisation

Noms:

n	$::=$		Nom déjà résolu
		r	Nom de ressource
	$\mid$	a	Nom de cellule
$\mathbf{r}$	$::=$		Nom variable de ressource
		r	Nom de ressource
	$\mid$	x	Variable
$\mathbf{a}$	$::=$		Nom variable de cellule
		a	Nom de cellule
	$\mid$	x	Variable
d	$::=$		Cible d'un message
		$\overline{\mathbf{a}}$	Membrane cible
	$\mid$	$\underline{\mathbf{a}}$	Contenu cible
u	$::=$		Nom
		a	Nom de cellule
	$\mid$	$\mathbf{r}$	Nom variable de ressource
	$\mid$	$d.\mathbf{r}$	Ressource localisée
μ	$::=$		Filtre de noms
		u	Nom
	$\mid$	$\_$	Nom quelconque
	$\mid$	$d.\_$	Ressource située dans d
	$\mid$	$\_.\mathbf{r}$	Ressource r localisée

FIG. *4.1* – **Syntaxe et noms du M–calcul.**

Noms libres:

$\mathsf{fn}(\mathbf{a}) = \mathsf{fn}(\overline{\mathbf{a}}) = \mathsf{fn}(\underline{\mathbf{a}}) = \{\mathbf{a}\}$
$\mathsf{fn}(d.\mathbf{r}) = \mathsf{fn}(d) \cup \{\mathbf{r}\}$
$\mathsf{fn}(d.\_) = \mathsf{fn}(d)$
$\mathsf{fn}(()) = \mathsf{fn}(\mathbf{0}) = \varnothing$
$\mathsf{fn}(\nu n.P) = \mathsf{fn}(P) \setminus \{n\}$
$\mathsf{fn}(\langle D \rangle) = \mathsf{fn}(D)$
$\mathsf{fn}(\top) = \varnothing$
$\mathsf{fn}(r_1\widetilde{x_1} \mid \ldots \mid r_q\widetilde{x_q}) = \mathsf{fn}(r_1\widetilde{x_1}) \cup \ldots \cup \mathsf{fn}(r_q\widetilde{x_q})$
$\mathsf{fn}(a(P)[Q]) = \{a\} \cup \mathsf{fn}(P) \cup \mathsf{fn}(Q)$
$\mathsf{fn}([\mu = V]P, Q) = \mathsf{fn}(\mu) \cup \mathsf{fn}(P) \cup \mathsf{fn}(Q) \cup \mathsf{fn}(V)$
$\mathsf{fn}(\nu n.C) = \mathsf{fn}(C) \setminus \{n\}$
$\mathsf{fn}((V_1, \ldots, V_p)) = \mathsf{fn}(V_1) \cup \ldots \mathsf{fn}(V_p)$
$\mathsf{fn}(\mathbf{r}) = \{\mathbf{r}\}$
$\mathsf{fn}(\_) = \varnothing$
$\mathsf{fn}(\_.\{\mathbf{r}\}) = \{\mathbf{r}\}$
$\mathsf{fn}(\lambda x.P) = \mathsf{fn}(P) \setminus \{x\}$
$\mathsf{fn}(PQ) = \mathsf{fn}(P) \cup \mathsf{fn}(Q)$
$\mathsf{fn}(D; D') = \mathsf{fn}(D) \cup \mathsf{fn}(D')$
$\mathsf{fn}(J \triangleright P) = (\mathsf{fn}(P) \setminus \mathsf{fn}(J)) \cup \mathsf{dn}(J)$
$\mathsf{fn}(r(x_1, \ldots, x_p)) = \{r, x_1, \ldots, x_p\}$
$\mathsf{fn}(P \mid Q) = \mathsf{fn}(P) \cup \mathsf{fn}(Q)$
$\mathsf{fn}(\mathtt{pass}_{\mathbf{a}}\ V) = \{\mathbf{a}\} \cup \mathsf{fn}(V)$
$\mathsf{fn}(\epsilon[P]) = \mathsf{fn}(P)$

Noms définis localement:

$\mathsf{dln}(\nu n.P) = \mathsf{dln}(P) \setminus \{n\}$
$\mathsf{dln}(D; D') = \mathsf{dln}(D) \cup \mathsf{dln}(D')$
$\mathsf{dln}(\mathbf{a}(P)[Q]) = \varnothing$
$\mathsf{dln}(\mathtt{pass}_{\mathbf{a}}\ V) = \varnothing$
$\mathsf{dln}(\mathbf{0}) = \varnothing$
$\mathsf{dln}(PQ) = \varnothing$
$\mathsf{dln}(\top) = \varnothing$
$\mathsf{dln}(P \mid Q) = \mathsf{dln}(P) \cup \mathsf{dln}(Q)$
$\mathsf{dln}(C) = \varnothing$
$\mathsf{dln}(\langle D \rangle) = \mathsf{dln}(D)$
$\mathsf{dln}(J \triangleright P) = \mathsf{dn}(J)$
$\mathsf{dln}([\mu = V]P, Q) = \varnothing$
$\mathsf{dln}(V) = \varnothing$

Cellules actives:

$\mathsf{cells}(\nu n.P) = \mathsf{cells}(P) \setminus \{n\}$
$\mathsf{cells}(\langle D \rangle) = \varnothing$
$\mathsf{cells}(P \mid Q) = \mathsf{cells}(P) \cup \mathsf{cells}(Q)$
$\mathsf{cells}(\mathtt{pass}_{\mathbf{a}}\ V) = \varnothing$
$\mathsf{cells}(\epsilon[P]) = \mathsf{cells}(P)$
$\mathsf{cells}(\mathbf{0}) = \varnothing$
$\mathsf{cells}(PQ) = \varnothing$
$\mathsf{cells}(a(P)[Q]) = \{a\} \cup \mathsf{cells}(P) \cup \mathsf{cells}(Q)$
$\mathsf{cells}([\mu = V]P, Q) = \varnothing$
$\mathsf{cells}(\nu n.C) = \mathsf{cells}(C) \setminus \{n\}$
$\mathsf{cells}(V) = \varnothing$
$\mathsf{cells}(x(P)[Q]) = \varnothing$

FIG. *4.2* – **Noms libres, noms définis localement et cellules actives.**

4.3.1 Equivalence structurelle

L'équivalence structurelle est la plus petite relation d'équivalence satisfaisant les règles rassemblées dans la figure 4.3, où l'opérateur de composition parallèle « | » et l'opérateur de composition « ; » pour les définitions sont commutatifs, associatifs, avec comme éléments neutres respectifs **0** et $\top$. La signification intuitive des règles concerne l'extrusion de portée des noms d'un contrôleur ou du contenu d'un domaine, l'équivalence de termes α-convertibles, et des propriétés de congruence pour les contextes d'évaluation $\mathbf{E}$.

$(\nu n.P) \mid Q$	$\equiv$	$\nu n.P \mid Q$	$n \notin \mathsf{fn}(Q)$	S.NU.PAR
$\epsilon[\nu n.P]$	$\equiv$	$\nu n.\epsilon[P]$		S.NU.TOP
$a(\nu n.P)[Q]$	$\equiv$	$\nu n.a(P)[Q]$	$n \notin \mathsf{fn}(Q) \ \wedge \ n \neq a$	S.NU.MEMB
$a(P)[\nu n.Q]$	$\equiv$	$\nu n.a(P)[Q]$	$n \notin \mathsf{fn}(P) \ \wedge \ n \neq a$	S.NU.CONT
$P =_\alpha Q$	$\Rightarrow$	$P \equiv Q$		S.ALPHA
$P \equiv Q$	$\Rightarrow$	$\mathbf{E}\{P\} \equiv \mathbf{E}\{Q\}$		S.CONTEXT

FIG. *4.3* – **Equivalence structurelle.**

4.3.2 Réduction

La relation de réduction $\rightarrow$ est la plus petite relation sur l'ensemble des processus du M–calcul satisfaisant les règles données dans les figures 4.4, 4.5 et 4.6. Les règles de la figure 4.4 décrivent l'exécution locale dans une cellule. On a la β–réduction standard du λ–calcul (règle R.BETA) et des règles de branchement conditionnel (règles R.IF.THEN et R.IF.ELSE) qui utilisent le prédicat *match*() défini par:

$$\mathit{match}(\_, V) \qquad \mathit{match}(u, u) \qquad \mathit{match}(\mathbf{a}.\_, \mathbf{a}.\mathbf{r}) \qquad \mathit{match}(\_.\mathbf{r}, \mathbf{a}.\mathbf{r}) \qquad \mathit{match}(\_.\_, \mathbf{a}.\mathbf{r})$$

La règle R.COMM décrit l'interaction entre une définition et un ensemble de messages satisfaisant au filtre contenu dans la définition: après synchronisation, le corps de la définition est appliqué aux arguments des messages, la définition d'origine restant inchangée. Cette règle est similaire à la règle JOIN du join–calcul. La règle R.PASSIV est nouvelle et définit la sémantique de l'opérateur `pass`, utilisé pour modifier la structure des localités. Migration et communication distante sont unifiées, la migration étant vue comme la communication d'un processus gelé. Ce dernier peut être dégelé à tout instant par simple application de la valeur vide $()$. La primitive $\mathtt{pass}_a\ V$ ne peut être activée qu'à l'intérieur de la cellule a. Son effet est de passiver la cellule courante en gelant l'exécution des processus de membrane et de contenu: deux « thunks » sont créés et sont alors passés comme arguments à la fonction V spécifiée par le programmeur, par exemple pour être transmis vers d'autres cellules. Les autres règles traduisent les possibilités de réduction sous contexte. On notera en particulier que la réduction est impossible sous une λ-abstraction, qui peut donc être utilisée pour définir des formes gelées $\lambda.P$.

Les règles suivantes (cf. figures 4.5 et 4.6) modélisent la communication distante. Elles montrent comment les messages peuvent voyager dans l'arborescence des domaines. Une communication distante comporte deux étapes: une résolution de l'endroit où envoyer le message et l'envoi proprement dit [153]. L'étape de résolution fait partie intégrante du calcul et s'appuie sur la forme des messages explicitement adressés $a.rV$, où a représente le nom de la cellule cible et r est un nom de ressource. On dispose ainsi de différentes règles selon la localisation de la ressource par rapport à la cellule courante. L'acheminement du message vers la cellule cible s'effectue pas à pas, en franchissant au plus une frontière de cellule à chaque

R.BETA	$(\lambda x.P)V \to P\{^V/_x\}$	R.COMM	$\dfrac{\langle D\rangle = \langle D_0 \,;\, r_1\widetilde{x_1} \mid \ldots \mid r_n\widetilde{x_n} \triangleright P\rangle}{\langle D\rangle \mid r_1\widetilde{V_1} \mid \ldots \mid r_n\widetilde{V_n} \to \langle D\rangle \mid P\{^{\widetilde{V_i}}/_{\widetilde{x_i}}\}}$
R.IF.THEN	$\dfrac{match(\mu, V)}{([\mu = V]P, Q) \to P}$	R.PASSIV	$\dfrac{}{a(\mathtt{pass}_a\ V \mid P)[Q] \to V(\lambda.P)(\lambda.Q)}$
R.IF.ELSE	$\dfrac{\neg match(\mu, V)}{([\mu = V]P, Q) \to Q}$	R.P.EQUIV	$\dfrac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$
R.CONTEXT	$\dfrac{P \to Q}{\mathbf{E}\{P\} \to \mathbf{E}\{Q\}}$	R.C.EQUIV	$\dfrac{C_1 \equiv C_1' \qquad C_1' \to C_2' \qquad C_2' \equiv C_2}{C_1 \to C_2}$

FIG. *4.4* – **Réduction: pas d'exécution élémentaires.**

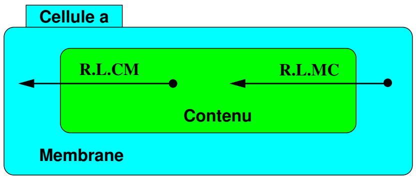
fois. Une étape de routage peut comporter des opérations de filtrage ou de reroutage, selon le comportement de contrôle de la localité courante. Cette possibilité permet au M–calcul de mettre en oeuvre différentes stratégies de routage des messages, depuis un acheminement complètement implicite (join–calcul) jusqu'à des routes explicitement spécifiées par le programmeur (calcul des ambients). L'isolation accompagnant la notion de domaine/cellule est rigoureusement appliquée: aucun message ne peut accéder ou sortir du contenu d'une cellule, sans avoir au préalable été traité par la membrane qui la contrôle.

Par définition, les messages locaux sont routés uniquement entre membrane et contenu d'une même cellule (cf. figure 4.5). Dans le routage des messages adressés, on note $\overline{\underline{b}}$ pour désigner indifféremment $\overline{b}$ ou $\underline{b}$. Les messages qui ont atteint leur destination (membrane ou contenu) sont simplement transformés en messages locaux (règles R.A$_1$.MM,EM,CM,CC,MC)[1]. Les messages entrants – messages situés à l'extérieur d'une cellule a et dont la ressource cible est située dans le contenu de a ou dont la cellule cible est une fille de a – sont filtrés par la membrane de la cellule sur le canal **i** (règles R.A$_1$.EC,R.A$_2$.IN). Les messages situés dans la membrane peuvent alors soit ressortir de la cellule ou se déplacer librement à l'intérieur de son contenu (règles R.A$_2$.MC,ME). Enfin, les messages sortants – messages situés dans le contenu d'une cellule a et destinés à des cellules situées dans la membrane de a ou extérieures à a – sont filtrés sur le canal **o** (règle R.A$_2$.OUT).

Intuitivement, c'est le contenu des définitions sur les canaux **i** et **o** qui gouverne la stratégie d'une membrane de cellule pour contrôler (filtrer) les informations entrant ou sortant de la cellule. Ces définitions peuvent être spécifiées explicitement par le programmeur, ce qui autorise la définition de différentes sémantiques de domaine.

[1] Les noms des différentes règles de routage doivent être lus comme suit: à chaque fois, le point de référence est la cellule courante. Les règles où le routage se produit selon la position de la ressource sont du type R.A$_1$.*; celles où le routage se produit selon la position de la cellule cible sont du type R.A$_2$.*. La fin du nom de la règle indique le trajet suivi par le message soit:

MC	membrane → contenu	MM	membrane → membrane	EM	extérieur → membrane	EC	extérieur → contenu
CM	contenu → membrane	CC	contenu → contenu	ME	membrane → extérieur		



$$\text{R.L.MC} \quad \frac{r \notin \mathsf{dIn}(P) \qquad r \in \mathsf{dIn}(Q)}{a(P \mid r\widetilde{V})[Q] \to a(P)[Q \mid r\widetilde{V}]} \qquad \text{R.L.CM} \quad \frac{r \in \mathsf{dIn}(P) \qquad r \notin \mathsf{dIn}(Q)}{a(P)[Q \mid r\widetilde{V}] \to a(P \mid r\widetilde{V})[Q]}$$

FIG. *4.5* – **Réduction: routage des messages locaux.**

4.4 Exemples

Nous illustrons le pouvoir expressif des différentes primitives du calcul à travers quelques exemples simples. D'autres exemples plus élaborés sont décrits dans [151]. Ils concernent la liaison ou la reconfiguration dynamique: mise à jour de composants, création de nouveaux domaines, ajout et/ou suppression d'un processus du contenu d'un domaine, migration d'un processus entre domaines, etc.

4.4.1 Une communication distante

Pour démontrer le fonctionnement des différentes règles de routage lors d'une communication distante, nous reproduisons un exemple tiré de [153] en l'illustrant graphiquement dans la figure 4.7. Les domaines sont représentés en séparant clairement le contrôleur (membrane) de son contenu par une barre horizontale. A partir de la configuration initiale:

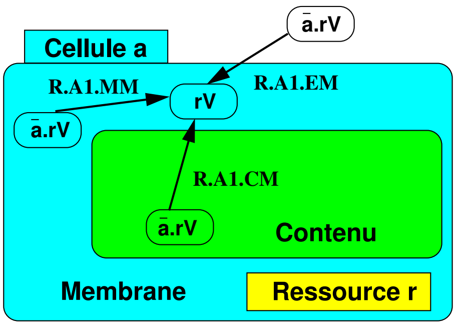
$$P \;=_{def}\; a.r\widetilde{V} \mid b(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[a(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[\langle r = \ldots \rangle]]$$

on obtient les réductions suivantes:

$$\begin{array}{lll}
P & \xrightarrow{\text{R.A}_2\text{.IN}} & b(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle \mid \mathbf{i}(a,r,\widetilde{V}))[a(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[\langle r = \ldots \rangle]] \\
 & \xrightarrow{\text{R.COMM}} & b(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle \mid a.r\widetilde{V})[a(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[\langle r = \ldots \rangle]] \\
 & \xrightarrow{\text{R.A}_2\text{.MC}} & b(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[a.r\widetilde{V} \mid a(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[\langle r = \ldots \rangle]] \\
 & \xrightarrow{\text{R.A}_1\text{.EC}} & b(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[a(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle \mid \mathbf{i}(a,r,\widetilde{V}))[\langle r = \ldots \rangle]] \\
 & \xrightarrow{\text{R.COMM}} & b(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[a(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle \mid a.r\widetilde{V})[\langle r = \ldots \rangle]] \\
 & \xrightarrow{\text{R.A}_1\text{.MM}} & b(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[a(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle \mid r\widetilde{V})[\langle r = \ldots \rangle]] \\
 & \xrightarrow{\text{R.L.MC}} & b(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[a(\langle \mathbf{i}(d,r,v) \triangleright d.rv \rangle)[\langle r = \ldots \rangle \mid r\widetilde{V}]] \\
 & \xrightarrow{\text{R.COMM}} & \ldots
\end{array}$$

Ces réductions illustrent le routage implicite pas à pas du calcul, avec possibilité de programmation explicite des fonctions de filtrage des messages franchissant les frontières de cellules.

Conversion en un message local:

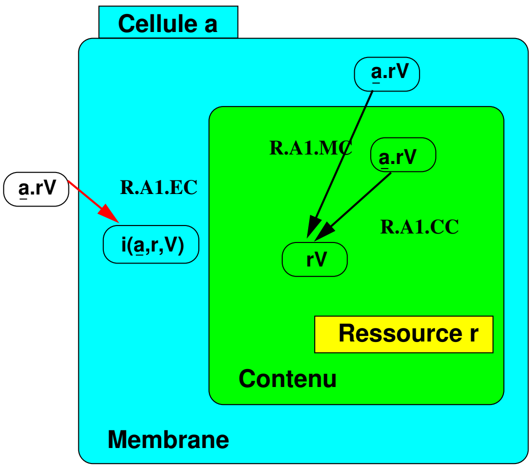


R.A$_1$.MM $$\frac{r \in \mathsf{dln}(P)}{a(P \mid \overline{a}.r\widetilde{V})[Q] \to a(P \mid r\widetilde{V})[Q]}$$

R.A$_1$.EM $$\frac{r \in \mathsf{dln}(P)}{\overline{a}.r\widetilde{V} \mid a(P)[Q] \to a(P \mid r\widetilde{V})[Q]}$$

R.A$_1$.CM $$\frac{r \in \mathsf{dln}(P)}{a(P)[Q \mid \overline{a}.r\widetilde{V}] \to a(P \mid r\widetilde{V})[Q]}$$

Conversion en un message local ou filtrage:

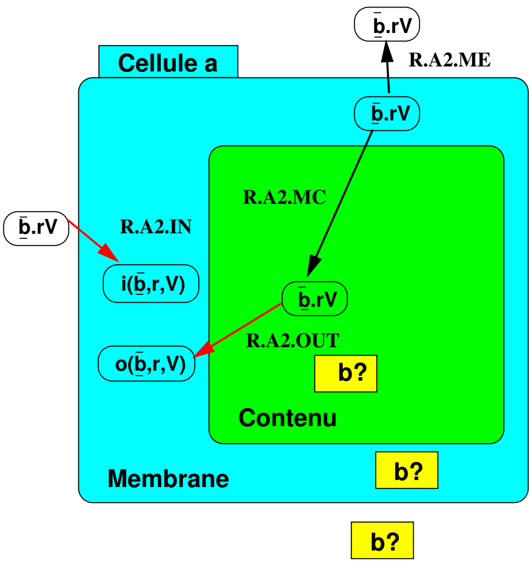


R.A$_1$.CC $$\frac{r \in \mathsf{dln}(Q)}{a(P)[Q \mid \underline{a}.r\widetilde{V}] \to a(P)[Q \mid r\widetilde{V}]}$$

R.A$_1$.MC $$\frac{r \in \mathsf{dln}(Q)}{a(P \mid \underline{a}.r\widetilde{V})[Q] \to a(P)[Q \mid r\widetilde{V}]}$$

R.A$_1$.EC $$\frac{r \in \mathsf{dln}(Q)}{\underline{a}.r\widetilde{V} \mid a(P)[Q] \to a(P \mid \mathbf{i}(\underline{a}, r, \widetilde{V}))[Q]}$$

Routage vers d'autres cellules:



R.A$_2$.IN $$\frac{b \in \mathsf{cells}(P) \cup \mathsf{cells}(Q) \quad b \neq a}{\overline{b}.r\widetilde{V} \mid a(P)[Q] \to a(P \mid \mathbf{i}(\overline{b}, r, \widetilde{V}))[Q]}$$

R.A$_2$.MC $$\frac{b \in \mathsf{cells}(Q) \setminus \mathsf{cells}(P) \quad b \neq a}{a(P \mid \overline{b}.r\widetilde{V})[Q] \to a(P)[Q \mid \overline{b}.r\widetilde{V}]}$$

R.A$_2$.ME $$\frac{b \notin \mathsf{cells}(P) \cup \mathsf{cells}(Q) \quad b \neq a}{a(P \mid \overline{b}.r\widetilde{V})[Q] \to a(P)[Q] \mid \overline{b}.r\widetilde{V}}$$

R.A$_2$.OUT $$\frac{b \notin \mathsf{cells}(Q) \quad b \neq a}{a(P)[Q \mid \overline{b}.r\widetilde{V}] \to a(P \mid \mathbf{o}(\overline{b}, r, \widetilde{V}))[Q]}$$

FIG. 4.6 – **Réduction: routage des messages adressés.**

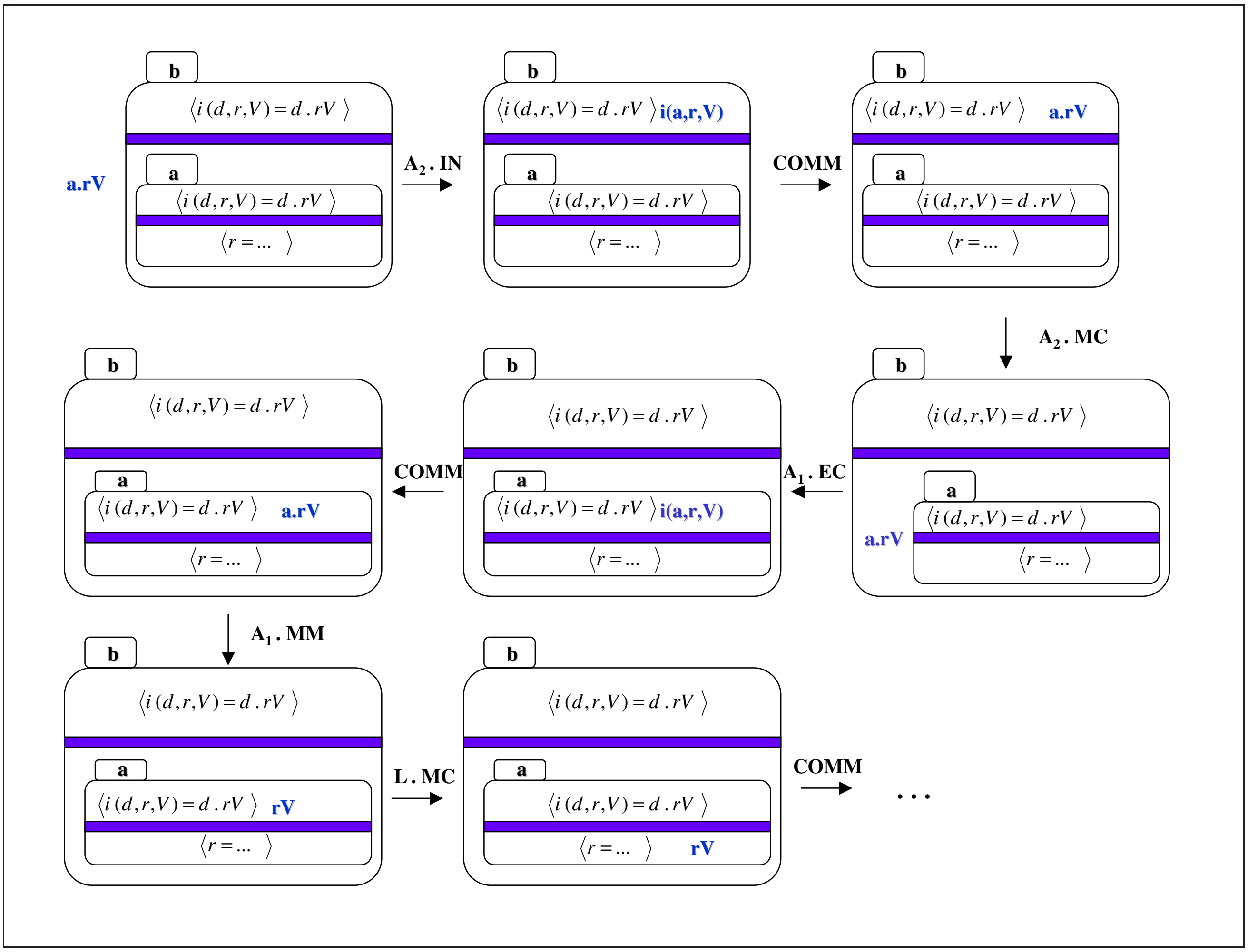


FIG. *4.7* – **Un exemple de communication distante en M–calcul.**

4.4.2 Un relais de messages

Des communications transparentes à la manière du join–calcul peuvent être réalisées en M–calcul en utilisant des relais de messages dans les membranes de cellules. Considérons le relais de messages suivant:

$$Fwd \; =_{def} \; \langle \mathbf{i}(d, r, v) \triangleright d.r \; v \; ; \;\; \mathbf{o}(d, r, v) \triangleright d.r \; v \rangle$$

Placé dans la membrane d'une cellule, Fwd relâche simplement dans la membrane les messages entrant et sortant de la cellule. Les règles de routage du calcul assurent que les messages seront correctement acheminés vers leur cellule cible. Ce relais de messages est sans état et fonctionne même si la cellule cible effectue une migration pendant le routage des messages.

4.4.3 Migration de domaines

En M–calcul, la mobilité de domaines peut être implantée en utilisant conjointement l'opérateur de passivation qui gèle l'exécution d'un domaine, et la communication d'ordre supérieur qui permet de transmettre un processus gelé sur le réseau. La cellule $Q_m(a)$ ci-dessous est capable de migrer vers une autre cellule:

$$\begin{array}{rcl} Q_m(a) & =_{def} & a(Fwd \mid \langle \mathsf{go}\, u \triangleright Go(a,u) \rangle)[Q] \\ Go(a,u) & =_{def} & \mathtt{pass}_a\ \lambda p\, q.(\overline{u}.\mathsf{enter}\ \lambda.a(p())[q()]) \end{array}$$

Une requête $(\mathsf{go}\, u)$ résulte dans la passivation de la cellule, et son envoi sous forme de « thunk » vers la cellule u. Si la requête provient de l'extérieur de la cellule, on réalise une forme objective de mobilité. Si elle provient du contenu de la cellule, la migration est subjective. La membrane de cellule u doit contenir le processus $Enter(u)$ pour permettre l'insertion de la cellule migrante dans le contenu de la cellule hôte:

$$Enter(u) \quad =_{def} \quad \langle \mathsf{enter}\, f \triangleright \mathtt{pass}_u\ \lambda p\, q.u(p())[q() \mid f()] \rangle$$

4.4.4 Contrôle des composants d'un domaine

Le contrôle mis en oeuvre par une membrane peut aller au-delà du filtrage de messages. Dans cet exemple, la cellule $Q_0(a)$ peut à volonté suspendre l'exécution de son contenu, la reprendre, dissoudre le contenu dans son environnement – comme la primitive `out` du calcul des ambients – ou effectuer une mise à jour arbitraire du processus de contrôle.

$$\begin{array}{rcl} Q_o(a) & = & \nu s\, on.a\ (Fwd \mid \langle\ \ \mathsf{suspend} \mid on \triangleright S(a,s);\ \mathsf{resume} \mid s\, f \triangleright R(a,f,on); \\ & & \qquad\qquad\qquad \mathsf{open} \triangleright O(a);\ \mathsf{update}\, f \triangleright U(a,f)\ \ \rangle\ )[Q] \\ S(a,s) & = & \mathtt{pass}_a\ \lambda p\, q.a(p() \mid (s\, q))[\mathbf{0}] \\ R(a,f,on) & = & \mathtt{pass}_a\ \lambda p\, q.a(p() \mid on)[f()] \\ O(a) & = & \mathtt{pass}_a\ \lambda p\, q.q() \\ U(a,f) & = & \mathtt{pass}_a\ \lambda p\, q.a(f())[q()] \end{array}$$

4.5 Expressivité

Un système de types pour le M–calcul est présenté dans [153]. Il garantit l'unicité des noms de localités, propriété fondamentale pour permettre le routage déterministe des messages dans une arborescence de localités. Le M–calcul permet de simuler le join–calcul réparti et donc de récupérer de nombreux résultats théoriques valides en join–calcul: une traduction du join–calcul (y compris avec défaillances) en M–calcul est présentée dans [151]. Des traductions en M–calcul du join–calcul avec liaison dynamique et du $\pi_{1\ell}$–calcul y sont également décrites.

4.6 Variantes du M–calcul

La version du M–calcul présentée ici correspond à celle publiée dans [67]. Cette version coïncide largement avec celle présentée par Schmitt et Stefani dans [153] et par Schmitt dans [151] que l'on peut qualifier de M–calcul « classique ».

4.6.1 Le M–calcul « classique »

Parmi les différences qui contribuent à réduire la complexité du calcul, on note l'abandon de la notation $\overline{a}.r$ ou $\underline{a}.r$ pour désigner un message destiné à une ressource située spécifiquement dans la membrane ou dans le contenu d'une cellule. Ces notations sont unifiées par un mode unique d'adressage $a.r$.

Le M–calcul classique simplifie également les règles de routage pour les messages adressés en n'en conservant qu'un nombre réduit. Ces règles sont regroupées dans la figure 4.8, en adoptant les notations de [153].

R.A.CTRL.FINAL

$$a(P \mid a.r\widetilde{V})[Q] \rightarrow a(P \mid r\widetilde{V})[Q]$$

R.A.CONT.FINAL

$$a(P)[Q \mid a.r\widetilde{V}] \rightarrow a(P)[Q \mid r\widetilde{V}]$$

R.A.IN

$$\frac{b \in \mathsf{cells}(P) \cup \mathsf{cells}(Q) \cup \{a\}}{b.r\widetilde{V} \mid a(P)[Q] \rightarrow a(P \mid \mathbf{i}(b, r, \widetilde{V}))[Q]}$$

R.A.OUT

$$\frac{b \notin \mathsf{cells}(Q) \quad b \neq a}{a(P)[Q \mid b.r\widetilde{V}] \rightarrow a(P \mid \mathbf{o}(b, r, \widetilde{V}))[Q]}$$

R.A.CTRL.TO.CONT

$$\frac{b \in \mathsf{cells}(Q) \setminus \mathsf{cells}(P) \quad b \neq a}{a(P \mid b.r\widetilde{V})[Q] \rightarrow a(P)[Q \mid b.r\widetilde{V}]}$$

R.A.CTRL.TO.ENV

$$\frac{b \notin \mathsf{cells}(P) \cup \mathsf{cells}(Q) \quad b \neq a}{a(P \mid b.r\widetilde{V})[Q] \rightarrow a(P)[Q] \mid b.r\widetilde{V}}$$

FIG. *4.8* – **M–calcul « classique » : routage des messages adressés.**

4.6.2 Le M–calcul « local »

Le M–calcul est un modèle de programmation très expressif qui présente différentes caractéristiques dont l'interaction est relativement complexe: domaines programmables, communication d'ordre supérieur, passivation, routage semi-transparent, noyau fonctionnel. Afin de faciliter son implantation dans les chapitres suivants, nous définissons dans cette section un sous-ensemble du M–calcul légèrement moins expressif. Le but est de comprendre comment un nombre restreint de ces caractéristiques pourront être implantées, si possible séparément, et d'étudier leurs interactions.

Dans un premier temps, l'accent est mis sur les domaines et leur organisation, ainsi que sur l'intégration au sein du modèle d'un noyau fonctionnel. L'aspect résolution des noms et routage des messages dans un système à large échelle est laissé de côté pour le moment: le modèle se limite à une forme simple de routage entre domaines, sans véritable distinction entre adressage local et distant. Ce modèle, appelé le M–calcul « local » et noté M_{loc} est adapté à une implantation centralisée du M–calcul. Les aspects concernant la répartition, comme le routage des messages et la mobilité des domaines seront examinés plus en détail dans le chapitre 5.

La principale modification apportée au calcul d'origine concerne les contrôleurs de domaine. En M–calcul, un contrôleur peut contenir d'autres domaines, par exemple dans un terme comme $a(S \mid b(P)[Q])[R]$. En interdisant les constructions de ce type dans M_{loc}, on peut associer à tout terme du calcul un arbre où la structure hiérarchique et celle de la relation de contrôle du calcul coïncident.

Cette restriction simplifie également la composition du contrôle. Dans une composition parallèle de processus de contrôle $a(P_1 \mid \ldots \mid P_n)[R]$, le contrôle global auquel est soumis R résulte d'une superposition entre les comportements individuels de contrôle des P_i. Il peut par exemple s'agir d'un consensus réparti entre les P_i. En se limitant au cas où un contrôleur de domaine ne comporte pas de terme définissant d'autres domaines, on restreint la composition du contrôle à une configuration « plate »: cette situation est moins coûteuse à implanter en terme de synchronisation qu'une composition de comportements de contrôle de niveaux de profondeur différents – problème illustré par la difficulté d'implantation des primitives de mobilité du calcul des ambients.

Enfin, afin de faciliter l'intégration du λ–calcul dans M_{loc}, on limite l'utilisation de l'opérateur d'application au seul cas où opérateur et opérande sont des processus capables de se réduire en une valeur. La figure 4.10 résume la syntaxe et la sémantique du M_{loc}–calcul. La signification des règles de routage est analogue à celles du M–calcul: les messages voyagent dans l'arborescence des domaines, en entrant et sortant de leurs contrôleurs et contenus.

On utilise un système de types pour garantir les propriétés informelles suivantes:

- Un contrôleur de domaine ne peut pas contenir de domaines.
- L'opérateur (resp. l'opérande) d'une application ne peut pas être un processus composé d'un domaine ou d'une définition.

Définition 4.1 (Ensemble de types). *On considère l'ensemble de types $\mathcal{T}$ suivants:*

$$\tau, \sigma \quad ::= \quad \top \mid \mathsf{fun}\,(\tau) \mid \mathsf{val} \mid \mathsf{flat} \mid \mathsf{dom}$$

On utilise $\tau, \sigma, \ldots$ et leurs variantes décorées pour désigner un type quelconque. Le système de types se présente sous la forme de séquents $\Gamma \vdash P : \tau$, où l'environnement Γ représente une association finie entre des noms du calcul et des types, soit formellement $\Gamma = \{u_1 : \tau_1, \ldots, u_n : \tau_n\}$. On note $u : \tau, \Gamma$ pour représenter l'environnement $\Gamma' = \Gamma \cup \{u : \tau\}$, avec $u \notin \mathsf{dom}\,\Gamma$. Intuitivement, le type fun caractérise les fonctions. Le type val caractérise les valeurs et les processus qui se réduisent en valeurs. Le type dom caractérise les domaines et les processus qui se réduisent en domaines. Enfin, le type flat caractérise les processus qui ne contiennent pas de domaines et n'en contiendront jamais en cours de réduction. Les règles du système de types sont données dans la figure 4.9.

$$\frac{}{\Gamma \vdash \mathbf{0} : \mathsf{flat}}\ \textsc{Nil} \qquad \frac{\Gamma(u) = \tau}{\Gamma \vdash u : \tau}\ \textsc{Env} \qquad \frac{x : \mathsf{val} \quad \Gamma \vdash P : \tau}{\Gamma \vdash \lambda\, x.P : \mathsf{fun}\,(\tau)}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash P : \mathsf{fun}\,(\tau) \quad \Gamma \vdash Q : \mathsf{val}}{\Gamma \vdash (P\ Q) : \tau}\ \textsc{App} \qquad \frac{r : \tau \quad \Gamma \vdash P : \sigma}{\Gamma \vdash (\nu\, r)\ P : \sigma}\ \textsc{New}$$

$$\frac{\Gamma \vdash P : \tau \quad \Gamma \vdash Q : \sigma}{\Gamma \vdash (P \mid Q) : \mathsf{dom}}\ \ \tau = \mathsf{dom}\ \text{ou}\ \sigma = \mathsf{dom} \quad \textsc{Par1}$$

$$\frac{\Gamma \vdash P : \tau \quad \Gamma \vdash Q : \sigma}{\Gamma \vdash (P \mid Q) : \mathsf{flat}}\ \ \tau \neq \mathsf{dom}\ \text{et}\ \sigma \neq \mathsf{dom} \quad \textsc{Par2}$$

$$\frac{\Gamma \vdash P : \mathsf{flat} \quad \Gamma \vdash Q : \tau}{\Gamma \vdash P[Q] : \mathsf{dom}}\ \textsc{Dom} \qquad \frac{\Gamma \vdash P : \tau \quad \Gamma \vdash Q : \tau}{\Gamma \vdash ([u\ =\ v]\ P\ ,\ Q) : \tau}\ \textsc{If}$$

$$\frac{\Gamma \vdash r : \mathsf{fun}\,(\mathsf{flat}) \quad \Gamma \vdash P : \mathsf{fun}\,(\mathsf{flat})}{\Gamma \vdash \langle r \triangleright P \rangle : \mathsf{flat}}\ \textsc{Decl1}$$

$$\frac{\Gamma \vdash r : \mathsf{fun}\,(\mathsf{flat}) \quad \Gamma \vdash P : \mathsf{fun}\,(\mathsf{dom})}{\Gamma \vdash \langle r \triangleright P \rangle : \mathsf{dom}}\ \textsc{Decl2}$$

$$\frac{\Gamma \vdash P : \mathsf{fun}\,(\tau)}{\Gamma \vdash P : \mathsf{val}}\ \textsc{Subtype1} \qquad \frac{\Gamma \vdash P : \mathsf{val}}{\Gamma \vdash P : \mathsf{flat}}\ \textsc{Subtype2}$$

FIG. *4.9* – **Systèmes de types pour le M_{loc}–calcul.**

Soit M^{-}_{loc} le sous-ensemble des processus du M_{loc}–calcul à l'exception de l'opérateur `pass` . On vérifie alors que le système de types précédent vérifie la propriété suivante de conservation du caractère bien formé des processus:

Théorème 4.1 (Conservation du caractère bien-formé des processus). *Soient P et Q des processus du M^{-}_{loc}–calcul, $\sigma \in \mathcal{T}$, et Γ un environnement:*

1. *Si $\Gamma \vdash P : \sigma$ et $P \equiv Q$, alors $\Gamma \vdash Q : \sigma$.*
2. *Si $\Gamma \vdash P : \sigma$ et $P \longrightarrow Q$, alors $\exists \tau \in \mathcal{T}$ tel que $\Gamma \vdash Q : \tau$.*

De plus, on a la proposition suivante:

Théorème 4.2. *Soient P et Q des processus du M^{-}_{loc}–calcul, et Γ un environnement:*

1. *Si $\Gamma \vdash P : \mathsf{flat}$ et $P \longrightarrow Q$, alors $\Gamma \vdash Q : \mathsf{flat}$.*
2. *Si $\Gamma \vdash P : \mathsf{val}$ et $P \longrightarrow Q$, alors $\Gamma \vdash Q : \mathsf{val}$.*

Syntaxe:

$$
\begin{array}{llll}
P & ::= & \mathbf{0} \mid V \mid P[P] \mid (P \mid P) \mid (P\ P) \mid (\nu\, r)\ P & \text{Processus} \\
 & \mid & ([u\ =\ u]\ P\ ,\ P) \mid \langle D \rangle \mid \mathtt{pass}\ V & \\
V & ::= & r \mid \lambda\, x.P & \text{Valeurs} \\
D & ::= & J \triangleright P \mid D; D & \text{Définitions} \\
J & ::= & r \mid J \mid J & \text{Filtres de messages} \\
u & ::= & x \mid r & \text{Noms}
\end{array}
$$

Contextes d'évaluation:

$$
\mathbf{E} \quad ::= \quad \cdot \mid (\mathbf{E}\ V) \mid (P\ \mathbf{E}) \mid (\mathbf{E} \mid P) \mid (\nu\, r)\ \mathbf{E} \mid \mathbf{E}[P] \mid P[\mathbf{E}]
$$

Equivalence structurelle:

$$
\begin{array}{rcll}
(\nu\, r)\ P \mid Q & \equiv & (\nu\, r)\ (P \mid Q) & r \notin \mathsf{fn}(Q) \\
((\nu\, r)\ P)[Q] & \equiv & (\nu\, r)\ P[Q] & r \notin \mathsf{fn}(Q) \\
P[(\nu\, r)\ Q] & \equiv & (\nu\, r)\ P[Q] & r \notin \mathsf{fn}(P) \\
P =_\alpha Q & \Rightarrow & P \equiv Q & \\
P \equiv Q & \Rightarrow & \mathbf{E}\{P\} \equiv \mathbf{E}\{Q\} &
\end{array}
$$

Réduction:

$$
\frac{}{([u\ =\ u]\ P\ ,\ Q) \longrightarrow P}\ \textsc{if.then} \qquad \frac{}{([u\ =\ v]\ P\ ,\ Q) \longrightarrow Q}\ u \neq v \quad \textsc{if.else}
$$

$$
\frac{}{(\lambda\, x.P\ V) \longrightarrow P\{V/x\}}\ \textsc{beta} \qquad \frac{\langle D \rangle = \langle D_0^*\ ;\ r_1 \mid \ldots \mid r_q \triangleright P \rangle}{\langle D \rangle \mid r_1\widetilde{V_1} \mid \ldots \mid r_q\widetilde{V_q} \longrightarrow \langle D \rangle \mid P\widetilde{V_1} \ldots \widetilde{V_q}}\ \textsc{com}
$$

$$
\frac{}{(r\widetilde{V} \mid P[Q]) \longrightarrow (r\widetilde{V} \mid P)[Q]}\ r \in \mathsf{dn}(P)\ \wedge\ r \notin \{\mathbf{i}, \mathbf{o}\} \quad \textsc{in}
$$

$$
\frac{}{(r\widetilde{V} \mid P)[Q] \longrightarrow P[(r\widetilde{V} \mid Q)]}\ r \in \mathsf{dn}(Q)\ \wedge\ r \notin \{\mathbf{i}, \mathbf{o}\} \quad \textsc{intrude}
$$

$$
\frac{}{(r\widetilde{V} \mid P[Q]) \longrightarrow ((\mathbf{i}\ \lambda.r\widetilde{V}) \mid P)[Q]}\ r \in \mathsf{dn}(Q)\ \wedge\ \mathbf{i} \in \mathsf{dn}(P)\ \wedge\ r \notin \{\mathbf{i}, \mathbf{o}\} \quad \textsc{filter.in}
$$

$$
\frac{}{(r\widetilde{V} \mid P)[Q] \longrightarrow (r\widetilde{V} \mid P[Q])}\ r \notin \mathsf{dn}(P[Q])\ \wedge\ r \notin \{\mathbf{i}, \mathbf{o}\} \quad \textsc{out}
$$

$$
\frac{}{P[(r\widetilde{V} \mid Q)] \longrightarrow (r\widetilde{V} \mid P)[Q]}\ r \in \mathsf{dn}(P)\ \wedge\ r \notin \{\mathbf{i}, \mathbf{o}\} \quad \textsc{extrude}
$$

$$
\frac{}{P[(r\widetilde{V} \mid Q)] \longrightarrow ((\mathbf{o}\ \lambda.r\widetilde{V}) \mid P)[Q]}\ r \notin \mathsf{dn}(P[Q])\ \wedge\ \mathbf{o} \in \mathsf{dn}(P)\ \wedge\ r \notin \{\mathbf{i}, \mathbf{o}\} \quad \textsc{filter.out}
$$

$$
\frac{}{(P \mid \mathtt{pass}\ V)[Q] \longrightarrow V(\lambda.P)(\lambda.Q)}\ \textsc{passiv}
$$

$$
\frac{P \longrightarrow Q}{\mathbf{E}\{P\} \longrightarrow \mathbf{E}\{Q\}}\ \textsc{context} \qquad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}\ \textsc{struct}
$$

FIG. *4.10* – **Syntaxe et sémantique du M$_{loc}$–calcul.**

4.7 Conclusion

Le M–calcul constitue une extension du join–calcul réparti avec des localités programmables, de la communication asynchrone paramétrable, de la liaison dynamique, et de la mobilité de code à l'aide de processus d'ordre supérieur. Ces caractéristiques font du M–calcul un candidat potentiel comme modèle de programmation adapté au contexte du « global computing ». Toutefois, un critère essentiel reste à vérifier: ce modèle doit être implantable en contexte réparti. Dans les deux chapitres suivants, nous examinons si cette exigence est satisfaite en décrivant une machine abstraite répartie, puis une machine virtuelle pour le M–calcul.

Chapitre 5

CLAM : une machine abstraite répartie pour le M–calcul

Ce chapitre a pour but de démontrer le caractère implantable en contexte réparti du modèle de programmation exposé au chapitre précédent. Il constitue la seconde étape de la démarche de formalisation d'un système à base de domaines. Nous y décrivons une spécification formelle du M–calcul sous forme de machine abstraite répartie appelée CLAM (CeLular Abstract Machine). Sa structure modulaire reflète la sémantique opérationnelle du calcul: ainsi, elle sépare la partie évaluation fonctionnelle standard du reste de l'exécution et identifie les mécanismes clés du routage réparti des messages en présence de mobilité.

Après avoir exposé dans la section 5.1 les principes généraux d'architecture ayant guidé la conception de la machine, nous présentons dans la section 5.2 une première version de la machine abstraite dans un contexte centralisé. Cette machine simplifiée est motivée par une approche graduelle de spécification des primitives du calcul. La machine implante un sous-ensemble « local » du M–calcul, le M_{loc}–calcul, déjà présenté au chapitre 4 . Nous passons ensuite dans la section 5.3 à la description formelle complète de la CLAM, machine abstraite répartie pour le M–calcul. Pour référence, des descriptions partielles de la machine figurent dans [66] et [67].

5.1 Principes généraux d'architecture

En vue de construire des infrastructures fiables pour la programmation répartie dans des environnements à large échelle, la validation d'une approche de formalisation de bout en bout fondée sur le M–calcul présuppose de démontrer l'effectivité du modèle, autrement dit son caractère implantable. Aussi, le M–calcul doit-il être raffiné en une spécification formelle qui servira de point de départ pour une implantation du modèle. Cette spécification prend ici la forme d'une machine abstraite.

Le niveau d'abstraction de la spécification doit à la fois refléter les infrastructures d'exécution répartie existantes, tout en étant proche du calcul pour faciliter l'expression et la preuve de propriétés relatives à son implantation. Dans cette optique, la machine abstraite répartie décrite dans ce chapitre repose sur les principes de conception suivants:

- *La machine abstraite doit être obtenue par raffinement immédiat du M–calcul.* Ses structures internes dérivent directement des primitives de base du calcul. De plus, la réduction sur les états de la machine abstraite est obtenue par une traduction quasiment un-pour-un de la relation de réduction du M–calcul.
- *La machine abstraite doit être effectivement implantable dans un contexte réparti asynchrone.* La machine abstraite du M–calcul fournit une séparation claire entre distribution logique et physique à la manière du join–calcul: la localité de plus haut niveau d'une arborescence de localités est identifiée avec le site qui héberge cette arborescence. Les interactions véritablement réparties se produisent par envois asynchrones de messages entre sites physiques, c'est-à-dire entre localités de plus haut niveau sur chaque site. Afin d'éviter toute synchronisation répartie inutile, les règles de réduction n'impliquent qu'une seule localité – avec si nécessaire un éther-global pour l'envoi et la réception de messages entre sites. En particulier, toutes les décisions relatives au routage réparti des messages sont prises localement.
- *L'organisation de la machine doit être modulaire et ouverte.*
  - La machine abstraite sépare clairement l'évaluation de termes fonctionnels, pris en charge par un évaluateur bien identifié, du cœur de la machine qui gère concurrence et distribution. Cette séparation se justifie par l'existence d'un sous-ensemble nettement délimité du M–calcul – le λ–calcul – et par un souci d'ouverture afin de permettre la spécification de plusieurs stratégies d'évaluation.
  - La gestion des noms et le routage des messages sont séparés de la partie de la machine abstraite en charge de l'exécution des programmes. Dès leur création, les noms de cellules sont garantis comme étant uniques dans l'ensemble du système. Au niveau du calcul, cette propriété repose sur l'utilisation de l'opérateur de restriction ν et de règles structurelles d'extension de portée des noms. Elle est raffinée au niveau de la machine abstraite en utilisant des noms contextuels qui incluent une référence au site qui les a créés..

    Pour obtenir un algorithme de routage déterministe des messages, il est primordial de garantir la préservation de l'unicité des noms de cellules au cours de l'exécution. Le calcul le réalise en utilisant un système de types décrit dans [153] et en prouvant une propriété de « subject reduction ». Au niveau de l'implantation, cette unicité est obtenue grâce à un service de nommage réparti qui maintient la localisation à la fois des définitions (dans quelle localité?) et des localités (dans quelle unité physique de distribution?). L'unicité des noms de cellules sert de base pour raffiner les mécanismes de routage du M–calcul dans la machine abstraite.

Dans la suite, nous décrivons une première version centralisée de la machine abstraite du M–calcul, avant de passer à la présentation de la machine abstraite répartie complète.

5.2 Une première machine abstraite centralisée

5.2.1 Langage d'entrée

Cette première machine abstraite vise à obtenir une spécification relativement simple. Aussi, se base-t-elle sur la syntaxe du M_{loc}–calcul décrit dans le chapitre précédent plutôt que sur le M–calcul complet. En M_{loc}, contrairement au M, la membrane d'une cellule reste « plate » au cours de l'exécution, c'est-à-dire qu'elle ne contient pas d'autres cellules. Cette propriété peut être garantie à l'aide d'un système de types (cf. section 4.6.2) et simplifie notablement la définition de la machine abstraite. Par ailleurs, pour alléger la présentation, nous ne prenons pas en compte les définitions multiples.

Nous introduisons la catégorie syntaxique des *expressions* afin de pouvoir isoler dans la machine un composant dont le seul rôle est d'évaluer des expressions fonctionnelles et de retourner des valeurs. Cette catégorie correspond à un sous-ensemble des processus et des valeurs du M–calcul.

Nous ajoutons deux nouvelles valeurs, $\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ ,\mathcal{R})$ et $\mathsf{reify}_{\mathsf{content}}(\mathcal{E}\ ,\ \mathcal{L})$, pour réifier respectivement la membrane et le contenu d'une cellule.

Définition 5.1 (Langage d'entrée). *La machine abstraite exécute les processus suivants du M_{loc}–calcul:*

$$
\begin{array}{lcll}
P & ::= & \mathbf{0} \mid P[P] \mid (P \mid P) \mid E \mid (\nu\, r)\ P & \textit{Processus} \\
 & \mid & ([u\ =\ u]\ P\ ,\ P) \mid \langle D \rangle \mid \mathtt{pass}\ V & \\
V & ::= & r \mid \lambda\, x.P & \textit{Valeurs} \\
 & \mid & \mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ ,\mathcal{R}) & \\
 & \mid & \mathsf{reify}_{\mathsf{content}}(\mathcal{E}\ ,\ \mathcal{L}) & \\
D & ::= & J \triangleright P & \textit{Définitions} \\
J & ::= & r \mid J \mid J & \textit{Filtres de messages} \\
u & ::= & x \mid r & \textit{Noms} \\
E & ::= & V \mid x \mid (E\ E) & \textit{Expressions}
\end{array}
$$

5.2.2 Architecture générale

Cette première machine abstraite est constituée d'une collection d'*agents* ou *localités* s'exécutant à l'intérieur d'un site physique unique, et d'un *service de localisation* des définitions pour que le routage des messages vers leurs définitions cibles puisse s'effectuer correctement.

Les agents sont organisés en une structure hiérarchique, qui est un arbre dans le cas le plus simple, ou une forêt si l'on dispose de multiples localités de plus haut niveau. Intuitivement, une localité regroupe un ensemble de processus cohabitant au même niveau dans la hiérarchie des domaines langage. Une localité conserve des liens vers un certain nombre de localités filles ou *sous-localités*.

Une membrane de cellule est représentée dans la machine abstraite par un type particulier de localité, appelé *localité-membrane* qui regroupe l'ensemble des processus *plats* constituant la membrane de cellule. Une localité-membrane dispose de deux types de sous-localités: une *localité-éther* (ou plus simplement *éther*) regroupe les processus plats situés au plus haut niveau du contenu d'une cellule; les autres sous-localités sont de type membrane et implantent les cellules contenues dans la cellule courante. La localité-éther joue le rôle d'un médium de communication partagé entre ces sous-localités pour permettre leur interaction. Chaque nœud non terminal de l'arbre des localités possède donc exactement un unique descendant terminal – une localité-éther – et N descendants non terminaux – les N localités-membranes représentant les membranes des cellules contenues dans la cellule courante. Par exemple, le processus du calcul $P[Q_1 \mid \ldots \mid Q_n \mid R_1[S_1] \mid \ldots \mid R_m[S_m]]$ sera représenté par l'arborescence suivante de localités:

$$\begin{tikzcd}
 & \boxed{P} \arrow[dl] \arrow[d] \arrow[drr] & & \\
\boxed{Q_1 \mid \ldots \mid Q_n} & \boxed{R_1} \arrow[d] & \cdots & \boxed{R_m} \arrow[d] \\
 & \boxed{S_1} & \cdots & \boxed{S_m}
\end{tikzcd}$$

La racine d'un arbre de localités sera appelée *localité-racine*, et sera généralement notée $l_\top$.

Un service de localisation est essentiel afin de pouvoir assurer correctement l'acheminement d'un message jusqu'à sa définition cible, quelle que soit sa position dans l'arborescence des localités, y compris en cas de migration. Ce service répond à la question: quelle est la localisation d'une ressource donnée à l'instant courant? Formellement, il est constitué d'une association $O = \{r_1 \mapsto l_1, \ldots, r_n \mapsto l_n\}$, où l_i est le nom de la localité abritant une définition en attente de messages sur la ressource r_i.

Chaque agent possède deux composants, un *moteur d'exécution* et un *évaluateur*, avec une interface bien définie entre les deux. Le rôle du moteur d'exécution est de maintenir l'état d'un agent et de le faire évoluer en fonction de son état propre et de celui des agents voisins dans l'arborescence. La seule fonction de l'évaluateur est d'évaluer des expressions fonctionnelles passées en paramètre et de retourner une valeur au moteur d'exécution. Plus précisément, l'évaluateur opère sur des applications et des abstractions. Les valeurs retournées sont soit des valeurs du M–calcul, soit des valeurs étendues correspondant aux primitives non-fonctionnelles du calcul: composition parallèle, restriction, définitions, branchement conditionnel, etc.

Ce type d'architecture présente l'avantage de rendre la machine abstraite adaptable: en isolant un composant ayant une interface clairement définie dont le seul rôle est d'évaluer des expressions fonctionnelles, il devient possible de changer de stratégie d'évaluation des expressions sans modification dans le reste de la spécification formelle. Le nouvel évaluateur doit simplement respecter l'interface avec le moteur d'exécution. A titre d'illustration d'intégration de l'évaluateur dans le reste du système, nous décrivons dans la suite du chapitre un évaluateur paresseux. Cependant, d'autres stratégies d'évaluation sont parfaitement envisageables.

L'interface entre ces deux composants est constituée par deux fonctions de traduction $[\![\,\cdot\,]\!]_{load}$, et $[\![\,\cdot\,]\!]_{unload}$, qui initialisent respectivement l'évaluateur avec l'expression à évaluer, et de récupèrent la valeur obtenue après l'évaluation pour la réinjecter dans l'état du moteur d'exécution. La structure d'ensemble d'un agent est représentée sur la figure 5.1a.

Le moteur d'exécution

L'architecture du moteur d'exécution est structurée en couches (cf. figure 5.1b), chaque couche étant composée d'un ensemble de règles traitant d'une fonction commune: le calcul séquentiel, les définitions et la communication locale, les cellules et la communication distante, et la mobilité des définitions assurée par la passivation des cellules. Ces règles de réduction sont décrites plus en détail dans la section 5.2.5.

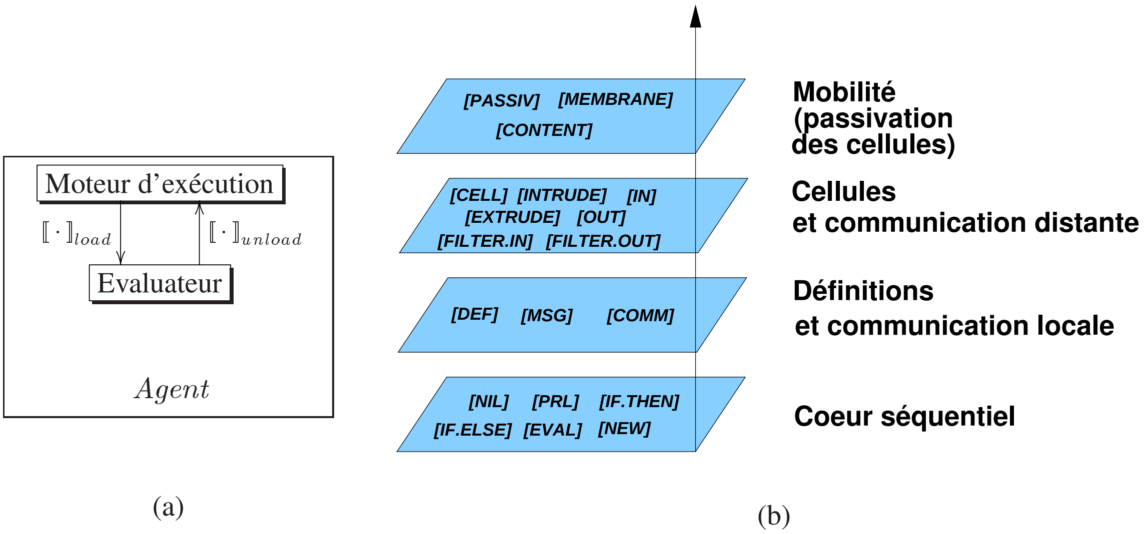


FIG. *5.1* – **Structure d'un agent:** *(a) composants d'un agent; (b) architecture du moteur d'exécution.*

L'évaluateur

L'évaluation se base sur des techniques de *réduction de graphes*, décrites dans [139]: l'expression à évaluer est représentée par un arbre dont les nœuds sont rassemblés à l'intérieur d'un tas $\mathcal{T}$. La structure de cet arbre va évoluer au fur et à mesure des réductions, éventuellement en se transformant en graphe si certaines sous-expressions sont partagées. L'évaluation se termine lorsque ce graphe ne contient plus d'expressions réductibles (*redex*).

L'exemple d'évaluateur présenté ici est celui d'un *évaluateur paresseux* – où les arguments d'une fonction sont évalués au plus une fois. Cette stratégie a deux conséquences:

1. Les arguments d'une fonction ne sont évalués qu'au moment où leur valeur est nécessaire à l'évaluation d'une expression et non avant.
2. L'utilisation ultérieure d'un argument de fonction ne nécessite pas de réévaluation, mais récupèrera la valeur calculée lors de la première évaluation.

L'évaluation est réalisée en suivant un *ordre normal de réduction*: la réduction se fait sur le redex le plus externe et le plus à gauche, i.e., l'application d'une lambda-abstraction à un argument sera réduite avant l'argument lui-même. L'évaluation peut ne pas se terminer. Dans ce cas, la localité devient bloquée.

Tant que l'expression est réductible, le processus d'évaluation consiste en trois étapes:

1. Trouver le redex le plus externe.
2. Effectuer la réduction.
3. Remplacer la racine du redex avec le résultat de la réduction.

Par exemple, une expression de la forme $(f\ E_1\ E_2\ E_3\ E_4)$ sera représentée dans le tas $\mathcal{T}$ par l'arbre décrit dans la figure 5.2. Pour trouver le prochain redex, l'évaluateur va partir de la racine de l'expression à

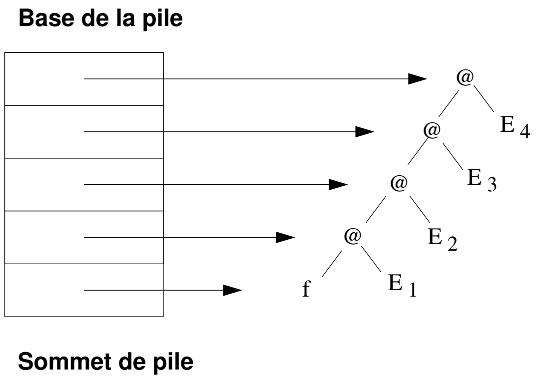


FIG. *5.2* – **Exemple d'évaluation.**

évaluer, et descendre en profondeur en suivant la branche gauche de chaque noeud d'application. L'ensemble des nœuds d'application ainsi rencontrés lors de la descente constitue l'« *épine dorsale* » de l'expression: ses « *vertèbres* » sont constituées des nœuds d'application, et ses « *côtes* » sont les fils droits de ces mêmes nœuds. Les côtes représentent les arguments de la fonction située à l'extrémité de l'épine dorsale de l'expression à évaluer. L'algorithme utilisé pour trouver le prochain redex consiste à dérouler cette épine dorsale jusqu'à rencontrer un nœud qui ne soit pas un nœud d'application. Le redex est alors formé par le nœud d'application immédiatement supérieur. Ainsi, dans l'exemple ci-dessus, si f est une fonction à un seul argument (une lambda-abstraction), le redex est $(f\ E_1)$. La descente de l'épine dorsale est effectuée à l'aide d'une pile $\mathcal{P}$, dont les éléments sont des pointeurs vers l'épine dorsale. Cette pile permet de conserver une trace des nœuds rencontrés lors de la descente.

5.2.3 Notations et catégories syntaxiques

La machine abstraite utilise des files pour ordonner les processus en attente de communication ou d'exécution dans une localité. On utilise les notations suivantes: étant donnée une catégorie syntaxique A, on note A^* (resp. $Queue(A)$) l'ensemble de toutes les séquences finies (resp. files) d'éléments de A. Un élément typique d'une telle file sera noté $a_1 :: \ldots :: a_n$. On note $\bullet$ la file vide. $Set(A)$ désigne l'ensemble des ensembles non ordonnés d'éléments de A. Si A et B sont des ensembles, on note $A \rightarrow B$ (resp. $A \mapsto B$) l'ensemble des applications (resp. des associations finies) de A vers B.

Etant donnés deux noms de localités l et l', les prédicats $desc(l, l')$ (resp. $ether\_child(l, l')$, $sibling(l, l')$) sont vérifiés si et seulement si la localité l est un descendant strict (resp. une sous-localité de type éther, une localité-soeur) de l'. La fermeture réflexive et transitive d'une relation de réduction $\longrightarrow$ est notée $\Longrightarrow$.

Définition 5.2 (Catégories syntaxiques de base). *On suppose également donnés un certain nombre d'ensembles correspondant aux catégories syntaxiques de base:*

$r, s, t \ldots$	$\in$	Ref	*Références*
$x, y, z \ldots$	$\in$	Var	*Variables*
$u, v, w \ldots$	$\in$	$\mathsf{Name} = \mathsf{Ref} \cup \mathsf{Var}$	*Noms*
$P, Q, R \ldots$	$\in$	Pr	*Processus*
$A, A' \ldots$	$\in$	$\mathsf{Abst} = \mathsf{Var} \times \mathsf{Pr}$	*Lambda-abstractions*
$V, W, \ldots$	$\in$	Val	*Valeurs*
$E, F, G \ldots$	$\in$	Expr	*Expressions*
$J, J' \ldots$	$\in$	$\mathsf{Join} = \mathsf{Ref}^*$	*Filtres de messages*
$D, D' \ldots$	$\in$	$\mathsf{Def} = \mathsf{Join} \times \mathsf{Pr}$	*Définitions*

et vérifiant les relations $\mathsf{Abst} \cup \mathsf{Ref} \subset \mathsf{Val}$, $\mathsf{Val} \subset \mathsf{Expr} \subset \mathsf{Pr}$, $\mathsf{Def} \subset \mathsf{Pr}$.

Les ensembles suivants sont également nécessaires pour décrire l'état de la machine abstraite :

- Les *noms internes* de la machine sont au cœur de la gestion des noms dans la machine abstraite. Typiquement, de tels noms – éléments de IntNm et notés i, j, k – sont des valeurs « riches » contenant des informations sur le domaine de désignation qui les gère, élément essentiel dans le cas de ressources réparties: le modèle de nommage réparti de la section 5.3 introduira des domaines de désignation multiples et explicitera les fonctions de transformation des noms internes en passant d'un domaine de désignation à l'autre. Dans cette section, les ressources sont centralisées au sein d'un domaine de désignation unique. Les noms internes sont donc assimilés à des valeurs « plates » comme les références. Aussi, dans la suite, on supposera que $\mathsf{IntNm} = \mathsf{Ref}$.
- Un *environnement* correspond à une association entre noms et valeurs, soit formellement $\mathsf{Env} = \mathsf{Name} \mapsto \mathsf{Val}$ dont les éléments sont notés ρ, ρ'. Etant donné un environnement ρ, on note $\rho \restriction \mathcal{E}$ la restriction de ρ à l'ensemble $\mathcal{E}$, et $closure(\rho, P)$ la restriction de ρ aux noms libres du processus P.
- Un *message* est constitué d'une référence (le récepteur cible) et d'une séquence de valeurs (les arguments). Il sera noté $(r\ \tilde{V})$. C'est un élément de $\mathsf{Msg} = \mathsf{Ref} \times \mathsf{Val}^*$.
- Les *noms de localités* sont des éléments de LocNm et sont notés l, l'. On note $loc\_id(\widetilde{V})$ l'ensemble des noms de localités qui apparaissent dans $\widetilde{V}$. On note aussi $rsc\_id(\widetilde{V})$ l'ensemble des noms de ressources qui sont définis dans $\widetilde{V}$. Ces notations sont utiles pour manipuler les valeurs $\mathsf{reify}_{\mathsf{membrane}}$ and $\mathsf{reify}_{\mathsf{content}}$.
- Les *adresses* utilisées dans l'évaluateur sont des éléments de Adr et sont notées α, β, γ.
- Les *nœuds* utilisés dans l'évaluateur pour représenter des sous-expressions à évaluer sont des éléments de l'ensemble Nodes. Des nœuds d'adresse α seront notés $e^\alpha, f^\alpha, g^\alpha$. Lorsque son adresse n'est pas importante, un nœud sera désigné par e, f, g par abus de notation. L'adresse α sera alors omise dans l'écriture du terme correspondant.

5.2.4 Etat de la machine

Moteur d'exécution

Chaque localité s'identifie à un quintuplet $l : \langle \mathcal{D}, \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{L} \rangle$. Le *nom de la localité* l est unique dans l'ensemble du système. $\mathcal{D}$ désigne l'ensemble des *définitions actives* dans la localité l, sous forme de couples (environnement, définition). $\mathcal{H}$ représente le *tas des messages* en attente de communication sur ces définitions. Il est constitué d'un ensemble de files de messages indexé par les noms des canaux définis dans l. $\mathcal{R}$ est la *file d'exécution* de la localité l et contient l'ensemble des processus prêts à être exécutés. Finalement, $\mathcal{E}$ et $\mathcal{L}$ sont utilisés pour se référer aux sous-localités de l de type éther et membrane respectivement. On notera que $\mathcal{E}$ est toujours un singleton pour des localités-membrane ou vide pour des localités-éther – auquel cas, on note $\mathcal{E} = \sharp$. Dans toute la suite, on notera $l.\mathcal{D}$ (resp. $l.\mathcal{H}, \ldots, l.\mathcal{L}$) le champ $\mathcal{D}$ (resp. $\mathcal{H}, \ldots, \mathcal{L}$) de la localité l. On note enfin $locrec(\mathcal{L})$ l'ensemble des noms de localités compris dans $\mathcal{L}$ en incluant tous leurs descendants.

Définition 5.3 (Etat du moteur d'exécution). *Formellement, la description de l'état du moteur d'exécution nécessite les catégories syntaxiques suivantes:*

$\mathcal{D}, \mathcal{D}'$	$\in$	$\mathsf{DefSpace} = Set(\mathsf{Env} \times \mathsf{Def})$	*Définitions locales actives*
$\mathcal{H}, \mathcal{H}'$	$\in$	$\mathsf{Heap} = \mathsf{IntNm} \mapsto \mathsf{VQueue}$	*Tas de messages en attente*
$\mathcal{V}, \mathcal{V}'$	$\in$	$\mathsf{VQueue} = Queue(\mathsf{Env} \times \mathsf{Val}^*)$	*Files de messages locaux*
$\mathcal{R}, \mathcal{R}'$	$\in$	$\mathsf{RQueue} = Queue(\mathsf{Env} \times \mathsf{Pr})$	*File d'exécution*
$\mathcal{E}, \mathcal{E}'$	$\in$	$\mathsf{Ether} = \mathsf{LocNm}$	*Ether intérieur*
$\mathcal{L}, \mathcal{L}'$	$\in$	$\mathsf{SubLoc} = Set(\mathsf{LocNm})$	*Sous-localités (de type membrane)*
$\mathbb{L}$	$\in$	$\mathsf{LocState} = \mathsf{DefSpace} \times \mathsf{Heap} \times$ $\mathsf{RQueue} \times \mathsf{Ether} \times \mathsf{SubLoc}$	*Etat du moteur d'exécution*

Dans cette version centralisée de la machine, le service de nommage est simplement une association entre noms de récepteurs et noms de localités, soit formellement un élément O de l'ensemble $\mathsf{Oracle} = \mathsf{IntNm} \mapsto \mathsf{LocNm}$.

Evaluateur

L'état de l'évaluateur d'une localité l dépend de la stratégie d'évaluation adoptée. A titre d'exemple, l'évaluateur paresseux présenté ici est constitué d'une pile, d'un environnement et d'un tas, soit un triplet $\mathbb{S}_l = \langle \mathcal{P}, \zeta, \mathcal{T} \rangle$. Le *tas* $\mathcal{T}$ est un ensemble de nœuds représentant les expressions à évaluer. Ces nœuds sont étiquetés par des adresses $\alpha \in \mathsf{Adr}$. La *pile* $\mathcal{P}$ est constituée d'un ensemble de pointeurs vers les nœuds du tas. Enfin, l'*environnement* ζ associe des références r à des adresses du tas. Il sert à garder une trace des références libres de l'expression en cours d'évaluation. La portée de ces références est potentiellement globale.

Définition 5.4 (Etat de l'évaluateur). *Formellement, la description de l'état de l'évaluateur nécessite les catégories syntaxiques suivantes:*

$\mathcal{P}, \mathcal{P}'$	$\in$	$\mathsf{EvalStack} = Queue(\mathsf{Adr})$	*Pile*
ζ, ζ'	$\in$	$\mathsf{EvalEnv} = \mathsf{Name} \mapsto \mathsf{Adr}$	*Environnement*
$\mathcal{T}, \mathcal{T}'$	$\in$	$\mathsf{EvalHeap} = \mathsf{Adr} \mapsto \mathsf{Nodes}$	*Tas*
$\mathbb{S}_l$	$\in$	$\mathsf{EvalState} = \mathsf{EvalStack} \times \mathsf{EvalEnv} \times \mathsf{EvalHeap}$	*Etat de l'évaluateur*

Définition 5.5 (Nœuds du tas). *Les nœuds du tas de l'évaluateur obéissent à la grammaire suivante:*

$$
\begin{array}{lcll}
e^\alpha & ::= & e_p^\alpha \mid e_i^\alpha & \textit{Nœuds du tas} \\
e_p^\alpha & ::= & \langle x\rangle^\alpha \mid \langle r\rangle^\alpha \mid \lambda^\alpha\, e.e \mid e@^\alpha e \mid \nabla^\alpha e & \textit{Nœuds purs} \\
 & \mid & \mathsf{reify}_{\mathsf{membrane}}{}^\alpha(\mathcal{D}\ ,\ \mathcal{H}\ ,\mathcal{R}) \mid \mathsf{reify}_{\mathsf{content}}{}^\alpha(\mathcal{E}\ ,\ \mathcal{L}) & \\
e_i^\alpha & ::= & 0^\alpha \mid e[e]^\alpha \mid (e \mid^\alpha e) \mid (\nu^\alpha\ e)\ e & \textit{Nœuds impurs} \\
 & \mid & ([e\ =\ e]^\alpha\ e\ ,\ e) \mid \langle \underbrace{e \mid \ldots \mid e}_{n\ fois} \triangleright e\rangle^\alpha \mid \mathsf{pass}^\alpha e &
\end{array}
$$

Les *noeuds purs* e_p représentent les expressions, tandis que les *nœuds impurs* e_i représentent les processus qui ne sont pas des expressions. On retrouve les principales constructions du calcul. A noter le nœud d'application noté ici $e@f$. Le nœud $\nabla\ e$ est un *nœud d'indirection* vers le nœud e. Ces indirections permettent à l'évaluateur de traiter le cas d'arbres syntaxiques où certaines expressions sont partagées et sont mises à jour après l'évaluation: l'ensemble de l'expression peut ainsi être évalué à l'aide d'une seule passe d'évaluation pour le sous-arbre partagé, au lieu d'une passe d'évaluation par point d'entrée vers ce sous-arbre.

Interface évaluateur-moteur d'exécution

Pour expliciter le lien entre nœuds de l'évaluateur et processus, on définit une fonction de traduction $[\![\cdot]\!]_e^{\mathcal{T}} : \mathsf{Nodes} \to \mathsf{Pr}$, qui associe à un nœud du tas $\mathcal{T}$ le processus correspondant du calcul. La définition de cette fonction est donnée dans la figure 5.3.

Réciproquement, étant donné un tas initial $\mathcal{T}$, on définit une fonction de traduction $[\![\cdot]\!]_p^{\mathcal{T}} : \mathsf{Pr} \to \mathsf{Adr} \times \mathsf{EvalHeap}$. L'image d'un processus P est un couple $(\alpha, \mathcal{T}')$, où $\mathcal{T}'$ est le tas $\mathcal{T}$ enrichi avec les nœuds correspondant au processus P et α l'adresse du point d'entrée dans ce tas. La définition de $[\![\cdot]\!]_p^{\mathcal{T}}$, donnée dans la figure 5.4, repose sur une fonction auxiliaire $\mathcal{F}^{\mathcal{T}}$ qui permet d'ajouter de nouveaux nœuds à un tas existant $\mathcal{T}$.

$$
\begin{array}{llll}
\mathcal{F}^{\mathcal{T}}: & \mathsf{Nodes} & \to & \mathsf{Adr} \times \mathsf{EvalHeap} \\
 & e & \mapsto & (\beta, \mathcal{T}\{\beta \mapsto e^\beta\}) \quad \beta \notin \mathsf{dom}\,(\mathcal{T})
\end{array}
$$

On peut alors expliciter formellement l'interface entre le moteur d'exécution et l'évaluateur: cette interface s'exprime à l'aide de deux fonctions de traduction $[\![\, E \,]\!]_{load}$, et $[\![\, \mathbb{S} \,]\!]_{unload}$ visant respectivement à initialiser l'évaluateur à partir d'une expression et à récupérer le résultat de l'évaluation sous forme d'une valeur. Plus précisément, pour l'évaluateur présenté ici:

Définition 5.6 (Fonction de traduction $[\![\, E \,]\!]_{load}$). $[\![\cdot]\!]_{load}$ *est un élément de* $\mathsf{Expr} \to \mathsf{EvalState}$ *qui renvoie un état initial* $\mathbb{S}$ *de l'évaluateur à partir d'une expression* E *à évaluer.*

$$[\![\, E \,]\!]_{load} \quad =_{def} \quad (\alpha, \{\}, \mathcal{T}) \qquad (\alpha, \mathcal{T}) = [\![\, E \,]\!]_p^{\{\}}$$

Définition 5.7 (Fonction de traduction $[\![\, \mathbb{S} \,]\!]_{unload}$). $[\![\cdot]\!]_{unload}$ *est un élément de* $\mathsf{EvalState} \to \mathsf{Pr}$ *qui prend en paramètre un état final* $\mathbb{S}$ *de l'évaluateur et renvoie un processus pouvant être réinjecté dans la file d'exécution de la localité correspondante.*

$$[\![\, (\alpha_1 :: \ldots :: \alpha_n, \zeta, \mathcal{T}) \,]\!]_{unload} \quad =_{def} \quad [\![\, \mathcal{T}(\alpha_n) \,]\!]_e^{\mathcal{T}}$$

$$
\begin{array}{rclrcl}
[\![\langle x\rangle]\!]_e^{\mathcal{T}} & =_{def} & x & [\![\langle r\rangle]\!]_e^{\mathcal{T}} & =_{def} & r \\
[\![\lambda\, e.f]\!]_e^{\mathcal{T}} & =_{def} & \lambda\, [\![e]\!]_e^{\mathcal{T}} . [\![f]\!]_e^{\mathcal{T}} & [\![e@f]\!]_e^{\mathcal{T}} & =_{def} & ([\![e]\!]_e^{\mathcal{T}}\ [\![f]\!]_e^{\mathcal{T}}) \\
[\![\nabla e]\!]_e^{\mathcal{T}} & =_{def} & [\![e]\!]_e^{\mathcal{T}} & [\![\mathsf{pass}\ e]\!]_e^{\mathcal{T}} & =_{def} & \mathsf{pass}\ [\![e]\!]_e^{\mathcal{T}} \\
[\![0]\!]_e^{\mathcal{T}} & =_{def} & \mathbf{0} & [\![e[f]]\!]_e^{\mathcal{T}} & =_{def} & [\![e]\!]_e^{\mathcal{T}}[[\![f]\!]_e^{\mathcal{T}}] \\
[\![(e \mid f)]\!]_e^{\mathcal{T}} & =_{def} & ([\![e]\!]_e^{\mathcal{T}} \mid [\![f]\!]_e^{\mathcal{T}}) & [\![(\nu\ e)\ f]\!]_e^{\mathcal{T}} & =_{def} & (\nu\ [\![e]\!]_e^{\mathcal{T}})\ [\![f]\!]_e^{\mathcal{T}}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ ,\mathcal{R})]\!]_e^{\mathcal{T}} & =_{def} & \mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ ,\mathcal{R}) \\
[\![\mathsf{reify}_{\mathsf{content}}(\mathcal{E}\ ,\ \mathcal{L})]\!]_e^{\mathcal{T}} & =_{def} & \mathsf{reify}_{\mathsf{content}}(\mathcal{E}\ ,\ \mathcal{L}) \\
[\![([e_1\ =\ e_2]\ f_1\ ,\ f_2)]\!]_e^{\mathcal{T}} & =_{def} & ([[\![e_1]\!]_e^{\mathcal{T}}\ =\ [\![e_2]\!]_e^{\mathcal{T}}]\ [\![f_1]\!]_e^{\mathcal{T}}\ ,\ [\![f_2]\!]_e^{\mathcal{T}}) \\
[\![\langle e_1 \mid \ldots \mid e_n \triangleright f\rangle]\!]_e^{\mathcal{T}} & =_{def} & \langle [\![e_1]\!]_e^{\mathcal{T}} \mid \ldots \mid [\![e_n]\!]_e^{\mathcal{T}} \triangleright [\![f]\!]_e^{\mathcal{T}} \rangle
\end{array}
$$

FIG. *5.3* – **Fonction de traduction $[\![\cdot]\!]_e^{\mathcal{T}}$.**

$$
\begin{array}{rcll}
[\![\mathbf{0}]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^0}[0] & \\
[\![P_1[P_2]]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^2}[\mathcal{T}^2(\alpha_1)[\mathcal{T}^2(\alpha_2)]] & [\![P_i]\!]_p^{\mathcal{T}^{i-1}} = (\alpha_i, \mathcal{T}^i) \quad i = 1,2 \\
[\![(P_1 \mid P_2)]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^2}[(\mathcal{T}^2(\alpha_1) \mid \mathcal{T}^2(\alpha_2))] & [\![P_i]\!]_p^{\mathcal{T}^{i-1}} = (\alpha_i, \mathcal{T}^i) \quad i = 1,2 \\
[\![r]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^0}[\langle r\rangle] & \\
[\![\lambda\, x.P]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^2}[\lambda\ \mathcal{T}^2(\alpha_1).\mathcal{T}^2(\alpha_2)] & [\![x]\!]_p^{\mathcal{T}^0} = (\alpha_1, \mathcal{T}^1) \quad [\![P]\!]_p^{\mathcal{T}^1} = (\alpha_2, \mathcal{T}^2) \\
[\![x]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^0}[\langle x\rangle] & \\
[\![(P_1\ P_2)]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^2}[\mathcal{T}^2(\alpha_1)@\mathcal{T}^2(\alpha_2)] & [\![P_i]\!]_p^{\mathcal{T}^{i-1}} = (\alpha_i, \mathcal{T}^i) \quad i = 1,2 \\
[\![(\nu\ r)\ P]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^2}[(\nu\ \mathcal{T}^2(\alpha_1))\ \mathcal{T}^2(\alpha_2)] & [\![r]\!]_p^{\mathcal{T}^0} = (\alpha_1, \mathcal{T}^1) \quad [\![P]\!]_p^{\mathcal{T}^1} = (\alpha_2, \mathcal{T}^2) \\
[\![\mathsf{pass}\ V]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^1}[\mathsf{pass}\mathcal{T}^1(\alpha_v)] & [\![V]\!]_p^{\mathcal{T}^0} = (\alpha_v, \mathcal{T}^1) \\
[\![\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ ,\mathcal{R})]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^0}[\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ ,\mathcal{R})] & \\
[\![\mathsf{reify}_{\mathsf{content}}(\mathcal{E}\ ,\ \mathcal{L})]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^0}[\mathsf{reify}_{\mathsf{content}}(\mathcal{E}\ ,\ \mathcal{L})] & \\
[\![([u_1\ =\ u_2]\ P_1\ ,\ P_2)]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^4}[([\mathcal{T}^4(\alpha_1) = \mathcal{T}^4(\alpha_2)]\mathcal{T}^4(\beta_1), \mathcal{T}^4(\beta_2))] & \\
 & & [\![u_i]\!]_p^{\mathcal{T}^{i-1}} = (\alpha_i, \mathcal{T}^i) \quad [\![P_i]\!]_p^{\mathcal{T}^{i+1}} = (\beta_i, \mathcal{T}^{i+2}) \quad i = 1,2 & \\
[\![\langle r_1 \mid \ldots \mid r_n \triangleright P\rangle]\!]_p^{\mathcal{T}^0} & =_{def} & \mathcal{F}^{\mathcal{T}^{n+1}}[\langle \mathcal{T}^{n+1}(\alpha_1) \mid \ldots \mid\ \mathcal{T}^{n+1}(\alpha_n) \triangleright \mathcal{T}^{n+1}(\beta)\rangle] & \\
 & & [\![r_i]\!]_p^{\mathcal{T}^{i-1}} = (\alpha_i, \mathcal{T}^i) \quad i = 1, \ldots, n \quad [\![P]\!]_p^{\mathcal{T}^n} = (\beta, \mathcal{T}^{n+1}) &
\end{array}
$$

FIG. *5.4* – **Fonction de traduction $[\![\cdot]\!]_p^{\mathcal{T}}$.**

5.2.5 Règles de réduction pour le moteur d'exécution

L'évolution d'une localité est décrite par des règles de réduction qui font intervenir l'état de cette localité, mais aussi celui des localités voisines dans l'arborescence des localités, ainsi que celui du service de nommage. On utilisera dans la description des règles un ensemble N de noms qui représente un service de création de nouveaux noms. Les règles modifient l'ensemble L de toutes les localités du système: L associe des noms de localités à des tuples $\langle \mathcal{D}, \mathcal{H}, \mathcal{R}, l_e, \mathcal{L} \rangle$[1]. Par exemple, on écrira

$$N : O : L\{l \xmapsto[l_e:\mathcal{L}]{\mathcal{D}:\mathcal{H}} \mathcal{R}, \ldots\} \longrightarrow N' : O' : L'\{l \xmapsto[l_e:\mathcal{L}']{\mathcal{D}':\mathcal{H}'} \mathcal{R}', \ldots\}$$

pour désigner une situation où: (i) l'état $\langle \mathcal{D}, \mathcal{H}, \mathcal{R}, l_e, \mathcal{L} \rangle$ d'une localité l devient $\langle \mathcal{D}', \mathcal{H}', \mathcal{R}', l_e, \mathcal{L}' \rangle$; (ii) de nouveaux noms sont créés au cours de la réduction; (iii) l'état du service de nommage passe de O à O' au cours de la transition. Dans un contexte de règle, toute partie non significative de l'état d'une localité sera notée $-$, comme dans $l \xmapsto[-]{-} \mathcal{R}$.

Cœur séquentiel

Le cœur du moteur d'exécution concerne le calcul séquentiel. Lorsque le processus en tête de file d'exécution d'une localité est le processus inerte $\mathbf{0}$, ce dernier est simplement supprimé et l'exécution se poursuit:

$$\text{M.NIL} \qquad N : O : L\{l \xmapsto[-]{-} (\rho, \mathbf{0}) :: \mathcal{R}\} \longrightarrow N : O : L\{l \xmapsto[-]{-} \mathcal{R}\}$$

Afin de garantir une propriété d'équité d'exécution – pour s'assurer qu'un processus situé dans la file d'exécution sera exécuté au bout d'un nombre fini d'étapes – les processus nouvellement créés sont en général placés en fin de file d'exécution. Par exemple, une composition parallèle $P \mid Q$ est interprétée en exécutant P immédiatement et en plaçant Q en fin de file pour une exécution différée:

$$\text{M.PRL} \qquad N : O : L\{l \xmapsto[-]{-} (\rho, (P \mid Q))\} \longrightarrow N : O : L\{l \xmapsto[-]{-} (\rho, P) :: \mathcal{R} :: (\rho, Q)\}$$

Les règles suivantes traduisent les réductions habituelles en présence d'expressions conditionnelles:

$$\text{M.IF.THEN} \qquad \frac{\rho(u) = \rho(v)}{N : O : L\{l \xmapsto[-]{-} (\rho, ([u \ = \ v] \ P \ , \ Q)) :: \mathcal{R}\} \longrightarrow N : O : L\{l \xmapsto[-]{-} (\rho, P) :: \mathcal{R}\}}$$

$$\text{M.IF.ELSE} \qquad \frac{\rho(u) \neq \rho(v)}{N : O : L\{l \xmapsto[-]{-} (\rho, ([u \ = \ v] \ P \ , \ Q)) :: \mathcal{R}\} \longrightarrow N : O : L\{l \xmapsto[-]{-} (\rho, Q) :: \mathcal{R}\}}$$

Lorsque le prochain processus à exécuter est une expression, son traitement est délégué à l'évaluateur. Ce dernier poursuit les réductions jusqu'à ce que l'expression ne contienne plus de redex ou ne soit plus une expression fonctionnelle pure. Le résultat de l'évaluation est alors placé en fin de file d'exécution pour une exécution ultérieure:

$$\text{M.EVAL} \qquad \frac{\mathbb{S}_l = [\![E]\!]_{load} \qquad \mathbb{S}_l \Longrightarrow_\beta \mathbb{S}'_l \qquad \mathbb{S}'_l \not\Longrightarrow_\beta \qquad P = [\![\mathbb{S}'_l]\!]_{unload}}{N : O : L\{l \xmapsto[-]{-} (\rho, E) :: \mathcal{R}\} \longrightarrow N : O : L\{l \xmapsto[-]{-} \mathcal{R} :: (\rho, P)\}}$$

[1]Formellement, L est un élément de l'ensemble $\mathsf{Locs} = \mathsf{LocNm} \mapsto \mathsf{LocState}$.

Cette règle constitue véritablement l'interface entre les deux composants bien identifiés d'une localité – l'évaluateur et le moteur d'exécution – chaque composant disposant de sa propre relation de réduction (respectivement $\longrightarrow_\beta$ et $\longrightarrow$). C'est à ce niveau que pourrait être ajouté un évaluateur pour le λ–calcul avec appel par valeur pour l'implémentation du M–calcul. Cet évaluateur devrait alors prendre en compte l'ensemble des valeurs définies dans le calcul de base. Plusieurs situations conduisent à une terminaison de l'évaluation:

- La pile de l'évaluateur est réduite à un seul élément: s'il s'agit de l'adresse d'un nœud pur correspondant à une expression ne contenant plus de redex, cette expression doit alors être remontée en fin de file d'exécution de la localité. S'il s'agit de l'adresse d'un nœud impur, c'est le processus correspondant qui doit être remonté en fin de file d'exécution.
- La pile de l'évaluateur ne se réduit pas à un seul élément: si le sommet de pile contient l'adresse d'un nœud pur, l'expression correspondante n'est plus réductible, aucune règle de l'évaluateur n'étant applicable. Il faut alors remonter l'ensemble de l'expression en cours d'évaluation dont la racine est constituée par la base de la pile de l'évaluateur. Sinon, le sommet de pile est constitué par l'adresse d'un nœud impur ce qui rend la poursuite de l'évaluation impossible. Comme dans les autres cas, il faut alors remonter l'ensemble de l'expression en cours d'évaluation.

Ces différents cas sont capturés par la fonction de traduction $[\![\cdot]\!]_{unload}$. Lors de la remontée du résultat de l'évaluation dans le moteur d'exécution, tous les nœuds d'indirection ∇e sont éliminés, seul le nœud e ainsi référencé étant remonté.

La création d'un nouveau nom de ressource n de portée P se traduit par la génération d'un nouveau nom interne i, toutes les occurrences de n dans P étant substituées par i.

$$\text{M.NEW} \quad \frac{\rho' = \rho\{n \mapsto i\} \quad i \notin N}{N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} (\rho, (\nu\, n)\ P) :: \mathcal{R}\} \longrightarrow N \oplus \{i\} : O : L\{l \overset{-}{\underset{-}{\longmapsto}} (\rho', P) :: \mathcal{R}\}}$$

Définitions et communication locale

Les règles de communication locale sont similaires à celles d'une machine abstraite pour le join–calcul [138], car les primitives concernées sont des constructions du join.

Toute nouvelle définition D rencontrée en tête de file d'exécution est placée dans l'ensemble $\mathcal{D}$ des définitions locales actives. L'environnement d'exécution ρ' conservé dans $\mathcal{D}$ est déduit de celui de la définition D par un opérateur de fermeture ne conservant que les variables libres. Les noms internes associés aux noms définis dans D deviennent des nouvelles entrées dans le tas $\mathcal{H}$ de messages en attente. Pour chaque nom défini dans D, une entrée est également créée dans le service de nommage, associant ce nom avec la localité courante.

$$\text{M.DEF} \quad \frac{\begin{array}{c}\rho' = closure(\rho, \langle D \rangle) \quad \mathcal{D}' = \mathcal{D} \oplus (\rho', D) \\ \mathcal{H}' = \mathcal{H}\ \{\rho(r) \mapsto \bullet\}_{r \in dn(D)} \quad O' = O\ \{\rho(r) \mapsto l\}_{r \in dn(D)}\end{array}}{N : O : L\{l \overset{\mathcal{D}:\mathcal{H}}{\underset{-}{\longmapsto}} (\rho, \langle D \rangle) :: \mathcal{R}\} \longrightarrow N : O' : L\{l \overset{\mathcal{D}':\mathcal{H}'}{\underset{-}{\longmapsto}} \mathcal{R}\}}$$

Les nouveaux messages sont rangés dans le tas $\mathcal{H}$ de messages en attente, à l'entrée associée au nom du destinataire – si cette entrée existe déjà, i.e., si la définition correspondante a déjà été rencontrée par le moteur d'exécution. Dans ce cas, le message est retiré de la file d'exécution. Sinon, le message est replacé en fin de file, pour laisser une chance à la localité de rencontrer la définition. Un mécanisme dans la machine virtuelle permet de limiter le nombre de telles remises en fin de file, afin de ne pas boucler indéfiniment sur cette règle.

$$\text{M.MSG}\quad \begin{cases} \dfrac{\mathcal{H}(\rho(r)) = \mathcal{V} \quad \mathcal{H}' = \mathcal{H}\ \{\rho(r) \mapsto \mathcal{V} :: (\rho, \widetilde{V})\}}{N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}} (\rho, r\widetilde{V}) :: \mathcal{R}\} \longrightarrow N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}'} \mathcal{R}\}} \\[2em] \dfrac{\rho(r) \notin \mathsf{dom}\,\mathcal{H}}{N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}} (\rho, r\widetilde{V}) :: \mathcal{R}\} \longrightarrow N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}} \mathcal{R} :: (\rho, r\widetilde{V})\}} \end{cases}$$

L'activation effective d'une définition $\langle J \triangleright P\rangle$ se produit lorsque chacun des noms définis dans J possède au moins un message en attente dans le tas $\mathcal{H}$. Chaque message est alors retiré de la file d'attente où il est placé et le corps P de définition est inséré en fin de file d'exécution.

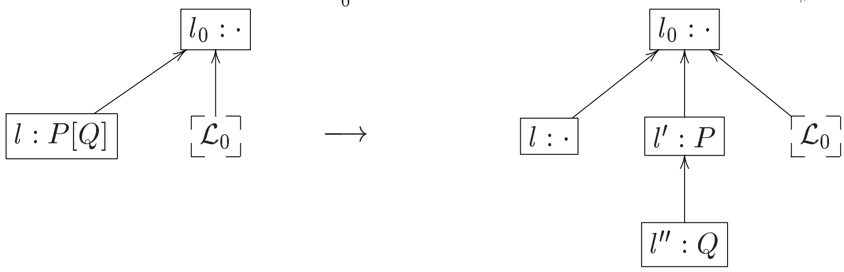
$$\text{M.COMM}\quad \frac{\begin{array}{c}(\rho, D) \in \mathcal{D} \quad D = J \triangleright P \quad J = r_1 \mid \ldots \mid r_q \\ \left[\mathcal{H}(\rho(r_i)) = (\rho_i, \widetilde{V_i}) :: \mathcal{V}_i\right]_{1 \le i \le q} \quad \rho' = \left[\rho \,\cup \bigcup_{1 \le i \le q} \rho_i\right] \quad \mathcal{H}' = \left[\mathcal{H}\ \{\rho(r_i) \mapsto \mathcal{V}_i\}\right]_{1 \le i \le q}\end{array}}{N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}} \mathcal{R}\} \longrightarrow N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}'} \mathcal{R} :: (\rho', P\widetilde{V_1} \cdots \widetilde{V_q})\}}$$

Cellules et communication distante

Les opérations qui changent la structure de la forêt de localités sont la création de nouvelles cellules et la passivation/activation de cellules, mécanisme dont la description sera effectuée dans le paragraphe sur la migration.

La création d'une cellule se produit lors de l'exécution d'un processus $P[Q]$. La localité courante l est alors nécessairement de type éther. Une localité l', sœur de l est créée pour accueillir la membrane de la nouvelle cellule. De même, une localité l'', fille de l' est introduite pour l'exécution du contenu de la cellule. Les files d'exécution des localités l' et l'' sont alors initialisées avec les processus P et Q respectivement.

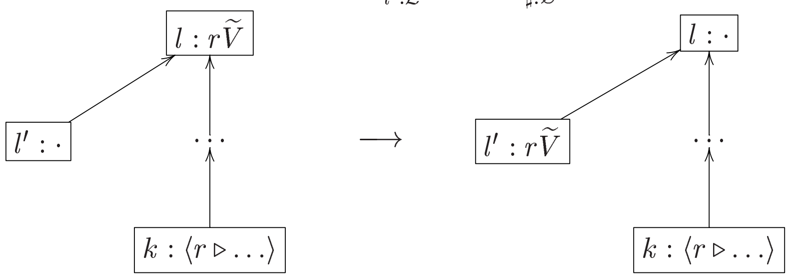
$$\text{M.CELL}\quad \frac{l' \notin N \quad l'' \notin N \quad \rho' = closure(\rho, P) \quad \rho'' = closure(\rho, Q) \quad \mathcal{L}'_0 = \mathcal{L}_0 \cup \{l'\}}{\begin{array}{c} N : O : L\{l_0 \xmapsto[l:\mathcal{L}_0]{-} \mathcal{R}_0, l \xmapsto[-]{-} (\rho, P[Q]) :: \mathcal{R}\} \longrightarrow \\ N \oplus \{l', l''\} : O : L\{l_0 \xmapsto[l:\mathcal{L}'_0]{-} \mathcal{R}_0, l \xmapsto[-]{-} \mathcal{R}, l' \xmapsto[l'':\varnothing]{\varnothing:\varnothing} (\rho', P), l'' \xmapsto[\sharp:\varnothing]{\varnothing:\varnothing} (\rho'', Q)\}\end{array}}$$



En terme de communication distante, cette première machine abstraite centralisée reste simple, en suivant de près la sémantique du calcul M_{loc}. On retrouve donc les six règles correspondantes de routage des messages dans l'arbre des localités.

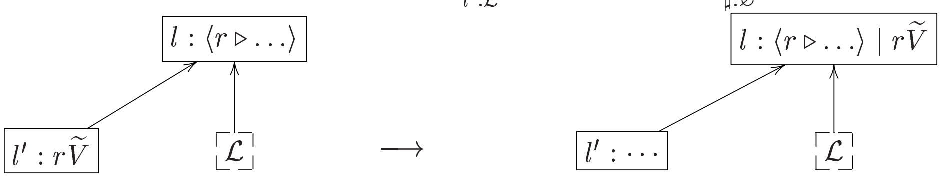
1. Lors de l'émission d'un message dans une membrane de cellule à destination de son contenu, la condition de déclenchement est que la destination du message soit un nom défini dans ce contenu, i.e., enregistré dans le service de nommage comme étant situé dans un descendant de la localité courante.

$$\text{M.INTRUDE}\quad \frac{O(\rho(r)) = k \quad desc(k, l) \quad r \notin \{\mathbf{i}, \mathbf{o}\}}{\begin{array}{c} N : O : L\{l \xmapsto[l':\mathcal{L}]{-} (\rho, r\widetilde{V}) :: \mathcal{R}, l' \xmapsto[\sharp:\varnothing]{-} \mathcal{R}'\} \longrightarrow \\ N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \mathcal{R}, l' \xmapsto[\sharp:\varnothing]{-} \mathcal{R}' :: (\rho, r\widetilde{V})\}\end{array}}$$



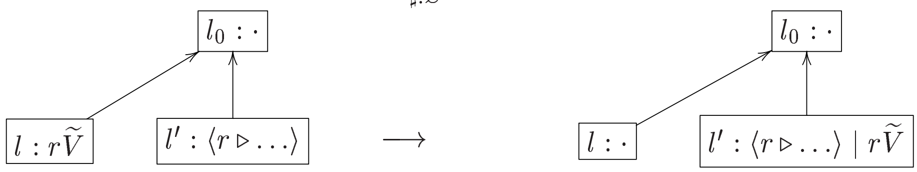
2. Lors de l'émission d'un message dans une localité-éther à destination de la membrane de cellule englobante, la condition de déclenchement est que la destination du message soit un nom défini dans cette membrane, i.e., enregistré dans le service de nommage comme étant situé dans la localité-mère de la localité-éther.

$$\text{M.EXTRUDE}\quad \frac{O(\rho(r)) = l \quad r \notin \{\mathbf{i}, \mathbf{o}\}}{\begin{array}{c} N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \mathcal{R}, l' \xmapsto[\sharp:\varnothing]{-} (\rho, r\widetilde{V}) :: \mathcal{R}'\} \longrightarrow \\ N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \mathcal{R} :: (\rho, r\widetilde{V}), l' \xmapsto[\sharp:\varnothing]{-} \mathcal{R}'\}\end{array}}$$



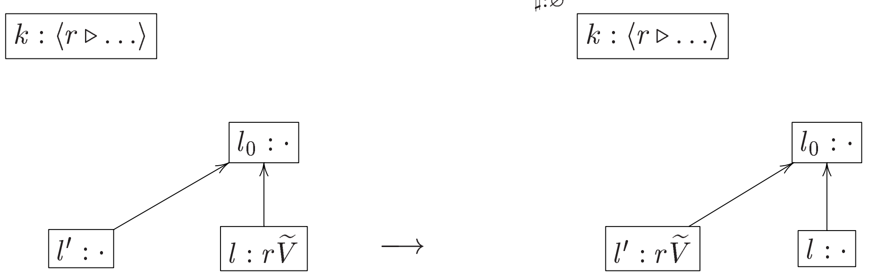
3. Lors de l'émission d'un message dans une localité-éther à destination d'une membrane de même niveau, la condition de déclenchement est que la destination du message soit un nom défini dans cette membrane, i.e., enregistré dans le service de nommage comme étant situé dans une localité sœur de la localité-éther.

$$\text{M.IN}\quad \frac{O(\rho(r)) = l' \quad siblings(l,l') \quad r \notin \{\mathbf{i},\mathbf{o}\}}{\begin{array}{c} N : O : L\{l \xmapsto[\sharp:\varnothing]{-} (\rho, r\widetilde{V}) :: \mathcal{R}, l' \xmapsto[-]{-} \mathcal{R}'\} \longrightarrow \\ N : O : L\{l \xmapsto[\sharp:\varnothing]{-} \mathcal{R}, l' \xmapsto[-]{-} \mathcal{R}' :: (\rho, r\widetilde{V})\} \end{array}}$$



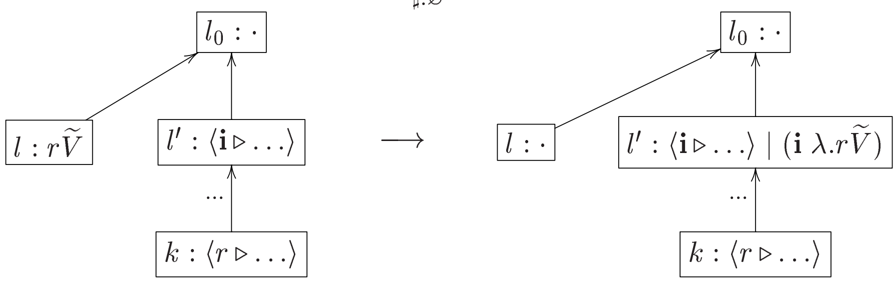
4. Lors de l'émission d'un message dans une membrane à destination de l'éther de même niveau, la condition de déclenchement est que la destination du message ne soit pas un nom défini dans la cellule courante – ni dans la membrane, ni dans son contenu –, i.e., ne soit pas enregistré dans le service de nommage comme étant situé dans la localité courante ou un de ses descendants.

$$\text{M.OUT}\quad \frac{O(\rho(r)) = k \quad k \neq l \quad \neg\, desc(k,l) \quad siblings(l,l') \quad r \notin \{\mathbf{i},\mathbf{o}\}}{\begin{array}{c} N : O : L\{l \xmapsto[-]{-} (\rho, r\widetilde{V}) :: \mathcal{R}, l' \xmapsto[\sharp:\varnothing]{-} \mathcal{R}'\} \longrightarrow \\ N : O : L\{l \xmapsto[-]{-} \mathcal{R}, l' \xmapsto[\sharp:\varnothing]{-} \mathcal{R}' :: (\rho, r\widetilde{V})\} \end{array}}$$



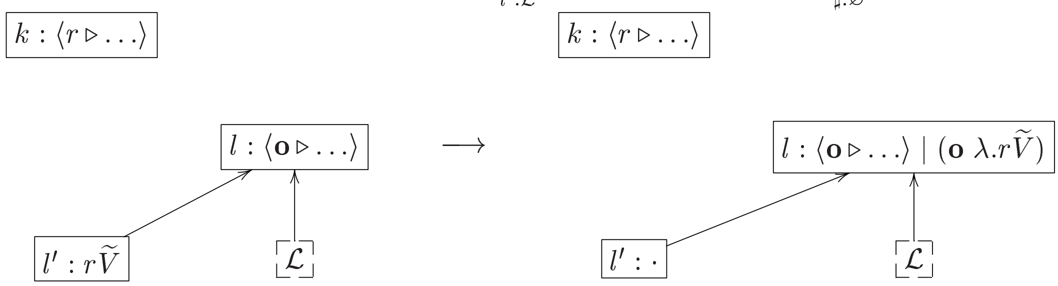
5. Lors de l'émission d'un message présent dans une localité-éther à destination du contenu d'une cellule située au même niveau, la condition de déclenchement est que la destination du message soit un nom défini dans le contenu de cette cellule, i.e. , enregistré dans le service de nommage comme étant situé dans une localité qui soit un descendant d'une soeur de la localité-éther.

$$\text{M.FILTER.IN}\quad \frac{O(\rho(r)) = k \quad desc(k, l') \quad siblings(l, l') \quad O(\rho(\mathbf{i})) = l' \quad r \notin \{\mathbf{i}, \mathbf{o}\}}{\begin{array}{c} N : O : L\{l \xmapsto[\sharp:\varnothing]{-} (\rho, r\widetilde{V}) :: \mathcal{R}, l' \xmapsto[-]{-} \mathcal{R}'\} \longrightarrow \\ N : O : L\{l \xmapsto[\sharp:\varnothing]{-} \mathcal{R}, l' \xmapsto[-]{-} \mathcal{R}' :: (\rho, (\mathbf{i}\ \lambda.r\widetilde{V}))\} \end{array}}$$



6. Lors de l'émission d'un message dans une localité-éther à destination de l'extérieur de la cellule courante, la condition de déclenchement est que la destination du message ne soit un nom défini ni dans la membrane ni dans le contenu de cette cellule, i.e. , ne soit pas enregistré dans le service de nommage comme étant situé dans une localité qui soit un descendant de la localité mère de la localité-éther.

$$\text{M.FILTER.OUT}\quad \frac{O(\rho(r)) = k \quad k \neq l \quad \neg\, desc(k, l) \quad O(\rho(\mathbf{o})) = l \quad r \notin \{\mathbf{i}, \mathbf{o}\}}{\begin{array}{c} N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \mathcal{R}, l' \xmapsto[\sharp:\varnothing]{-} (\rho, r\widetilde{V}) :: \mathcal{R}'\} \longrightarrow \\ N : O : L\{l \xmapsto[l':\mathcal{L}]{-} \mathcal{R} :: (\rho, (\mathbf{o}\ \lambda.r\widetilde{V})), l' \xmapsto[\sharp:\varnothing]{-} \mathcal{R}'\} \end{array}}$$



Mobilité

L'exécution de l'opérateur de passivation `pass` V conduit à la double réification de la localité courante et de ses sous-localités à l'aide des valeurs $\mathsf{reify}_{\mathsf{membrane}}$ et $\mathsf{reify}_{\mathsf{content}}$ passées ensuite en argument à la fonction V[2]. Le processus résultant est inséré dans l'éther de même niveau que la localité contenant le `pass`. Le sous-arbre passivé est alors supprimé de l'ensemble des localités. Enfin, récursivement, l'ensemble des noms définis dans des définitions situées dans des localités passivées sont retirés du service de nommage. La règle de passivation utilise les fonctions auxiliaires $extract$ et $unbind$ définies ci-dessous.

[2]Etant donné une localité $l : \langle \mathcal{D}, \mathcal{H}, \mathcal{R}, l_e, \mathcal{L} \rangle$, on utilise ici la valeur $\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ , \mathcal{R})$ pour réifier les définitions activables $\mathcal{D}$, le tas des messages $\mathcal{H}$, et la file d'exécution $\mathcal{R}$ de la localité l. La valeur $\mathsf{reify}_{\mathsf{content}}(l_e\ ,\ \mathcal{L})$ est utilisée pour réifier toutes les sous-localités de l, i.e. , la localité-éther l_e, et les autres sous-localités $\mathcal{L}$ de type membrane et ainsi que tous leurs descendants.

Définition 5.8 (Extraction d'un arbre de localités). *La fonction* $extract$ *est donnée par :*

$$\begin{array}{llll} extract: & \mathsf{LocNm} \times \mathsf{Locs} & \to & \mathsf{Val} \times \mathsf{Val} \times \mathsf{Locs} \\ & (l, L) & \mapsto & \langle \mathsf{reify}_{\mathsf{membrane}}(l.\mathcal{D}\ ,\ l.\mathcal{H}\ , l.\mathcal{R}), \mathsf{reify}_{\mathsf{content}}(l.\mathcal{E}\ ,\ l.\mathcal{L}), rm(l, L)\rangle \end{array}$$

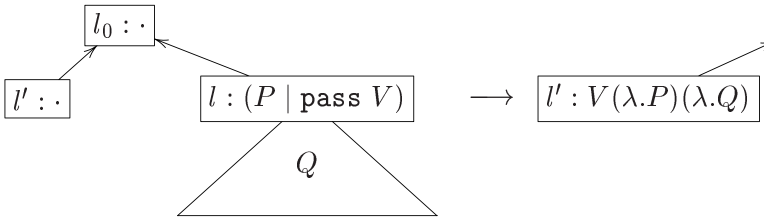
avec :

$$\begin{array}{llll} rm(l, L) & =_{def} & L\{l \mapsto \sharp\} & \text{si } l.\mathcal{L} = \varnothing \text{ et } l.\mathcal{E} = \sharp \\ rm(l, L) & =_{def} & rm(l.\mathcal{E}, L)\{l \mapsto \sharp\} & \text{si } l.\mathcal{L} = \varnothing \text{ et } l.\mathcal{E} \neq \sharp \\ rm(l, L) & =_{def} & rm(l_n, \ldots rm(l_1, rm(l.\mathcal{E}, L)) \ldots)\{l \mapsto \sharp\} & \text{sinon, avec } l.\mathcal{L} = \{l_1, \ldots, l_n\} \end{array}$$

où $\sharp$ *désigne une localité ayant été supprimée.*

La fonction $unbind(O, \rho, l)$ *est définie récursivement par:*

$$\begin{array}{llll} unbind(O, \rho, l) & =_{def} & O\{\rho(r) \mapsto \bullet\}_{r \in dn(D), D \in l.\mathcal{D}} & \text{si } l.\mathcal{L} = \varnothing \text{ et } l.\mathcal{E} = \sharp \\ unbind(O, \rho, l) & =_{def} & unbind(O, \rho, l.\mathcal{E})\{r \mapsto \bullet\}_{r \in dn(D), D \in l.\mathcal{D}} & \text{si } l.\mathcal{L} = \varnothing \text{ et } l.\mathcal{E} \neq \sharp \\ unbind(O, \rho, l) & =_{def} & unbind(\ldots unbind(unbind(O, \rho, l.\mathcal{E}), \rho, l_1), & \text{sinon, avec } l.\mathcal{L} = \{l_1, \ldots, l_n\} \\ & & \ldots, \rho, l_n)\{r \mapsto \bullet\}_{r \in dn(D), D \in l.\mathcal{D}} & \end{array}$$



M.PASSIV
$$\frac{\begin{array}{c} l \in \mathcal{L}_0 \quad \mathcal{L}_0' = \mathcal{L}_0 \backslash \{l\} \quad O' = unbind(O, \rho, l) \\ \langle \mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ , \mathcal{R}), \mathsf{reify}_{\mathsf{content}}(l_e\ ,\ \mathcal{L}), L'\rangle = extract(l, L) \end{array}}{\begin{array}{c} N : O : L\{l \xmapsto[l_e:\mathcal{L}]{\mathcal{D}:\mathcal{H}} (\rho, \mathsf{pass}\ V) :: \mathcal{R}, l_0 \xmapsto[l':\mathcal{L}_0]{-} \mathcal{R}_0, l' \xmapsto[-]{-} \mathcal{R}'\} \longrightarrow \\ N : O' : L'\{l_0 \xmapsto[l':\mathcal{L}_0']{-} \mathcal{R}_0, l' \xmapsto[-]{-} \mathcal{R}' :: (\rho, V\,\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ , \mathcal{R})\ \mathsf{reify}_{\mathsf{content}}(l_e\ ,\ \mathcal{L}))\} \end{array}}$$

Lorsque la localité passivée arrive à destination, les arguments du processus $\mathsf{reify}_{\mathsf{membrane}}$ sont déballés dans le contexte de la localité courante à l'aide d'une fonction $insert_{flat}$.[3]. Les définitions activables et les messages en attente réifiés rejoignent respectivement l'ensemble des définitions actives et le tas de messages locaux. De même, le contenu de la file d'exécution réifiée est inséré en fin de file d'exécution. Le service de nommage est alors informé de la nouvelle localisation des éléments qui viennent d'être déballés.

M.MEMBRANE
$$\frac{\begin{array}{c} \langle \mathcal{D}', \mathcal{H}', \mathcal{R}'\rangle = insert_{flat}(\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}_m, \mathcal{H}_m, \mathcal{R}_m), \mathcal{D}, \mathcal{H}, \mathcal{R}) \\ O' = O\{\forall D \in \mathcal{D}_m\ \ \forall r \in \mathsf{dn}(D)\ \ \rho(r) \mapsto l\} \end{array}}{N : O : L\{l \xmapsto[-]{\mathcal{D}:\mathcal{H}} (\rho, \mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}_m, \mathcal{H}_m, \mathcal{R}_m)\ ()) :: \mathcal{R}\} \longrightarrow N : O' : L\{l \xmapsto[-]{\mathcal{D}':\mathcal{H}'} \mathcal{R}'\}}$$

[3]Par exemple, cette fonction peut être définie par $insert_{flat}(\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}_m, \mathcal{H}_m, \mathcal{R}_m), \mathcal{D}, \mathcal{H}, \mathcal{R}) = \langle \mathcal{D} \cup \mathcal{D}_m, \mathcal{H} \cup \mathcal{H}_m, \mathcal{R} :: \mathcal{R}_m\rangle$.

Les arguments du processus $\mathsf{reify}_{\mathsf{content}}$ sont alors déballés à leur tour à l'aide d'une fonction $insert_{sublocs}$ dont la définition est similaire à $extract$. Les composants de la localité-éther réifiée sont directement insérés dans la localité courante. Le déballage des localités membrane entraine la création de soeurs de la localité courante. Le service de nommage est également mis à jour.

$$\text{M.CONTENT}\quad \frac{\begin{array}{c}\langle \mathcal{D}', \mathcal{H}', \mathcal{R}', \mathcal{L}_0', L' \rangle = insert_{sublocs}(\mathsf{reify}_{\mathsf{content}}(l_e, \mathcal{L}), \mathcal{D}, \mathcal{H}, \mathcal{R}, \mathcal{L}_0, L)\\ O' = O\{\forall D \in l_e.\mathcal{D}\ \ \forall r \in \mathsf{dn}(D)\ \ \rho(r) \mapsto l\}\\ O'' = O'\{\forall l_s \in locrec(\mathcal{L})\ \ \forall D \in l_s.\mathcal{D}\ \ \forall r \in \mathsf{dn}(D)\ \ \rho(r) \mapsto l_s\}\end{array}}{\begin{array}{c}N : O : L\{l_0 \xmapsto[l:\mathcal{L}_0]{-} \mathcal{R}_0, l \xmapsto[-]{\mathcal{D}:\mathcal{H}} (\rho, \mathsf{reify}_{\mathsf{content}}(l_e, \mathcal{L})\ ()) :: \mathcal{R}\} \longrightarrow\\ N : O'' : L'\{l_0 \xmapsto[l:\mathcal{L}_0']{-} \mathcal{R}_0, l \xmapsto[-]{\mathcal{D}':\mathcal{H}'} \mathcal{R}'\}\end{array}}$$

5.2.6 Règles de réduction pour l'évaluateur d'expressions

Règles d'évaluation

Les règles de l'évaluateur sont de la forme $\mathbb{S} \longrightarrow_\beta \mathbb{S}'$ où $\mathbb{S}$ et $\mathbb{S}'$ sont les états initiaux et finaux de l'évaluateur et $\longrightarrow_\beta$ une relation de réduction spécifique qui simule les réductions du λ–calcul. Nous décrivons ici un évaluateur paresseux avec appel par nom. Cependant, l'utilisation d'un évaluateur avec appel par valeur est parfaitement possible, à cause de la structure modulaire de la machine abstraite qui la rend adaptable.

Une première règle de réduction déroule un nœud d'application de l'épine dorsale de l'expression à évaluer. Son application répétée déroulera complètement l'épine dorsale de cette expression, jusqu'à trouver un nœud qui ne soit pas un nœud d'application. La condition $\mathcal{T}(\alpha_1) \neq \langle r \rangle$ permet d'éviter d'appliquer cette règle dans le cas d'un message $(r\ \tilde{V})$. La pile de l'évaluateur est mise à jour en empilant au dessus de l'adresse α d'un nœud d'application l'adresse α_1 de son fils gauche.

$$\text{M.UNWIND}\quad \frac{\mathcal{T}(\alpha) = e_1^{\alpha_1} @^{\alpha} e_2^{\alpha_2} \quad \mathcal{T}(\alpha_1) \neq \langle r \rangle}{\mathbb{S} = (\alpha :: \mathcal{P}, \zeta, \mathcal{T}) \longrightarrow_\beta \mathbb{S}' = (\alpha_1 :: \alpha :: \mathcal{P}, \zeta, \mathcal{T})}$$

Le cœur de l'évaluateur effectue les β-réductions du λ–calcul. Le redex est ici constitué de l'application d'une λ-abstraction (d'adresse α) à un argument (d'adresse γ). La règle consiste à remplacer le redex par une nouvelle instance du corps de la λ-abstraction, dans lequel le paramètre formel est remplacé par un pointeur vers l'argument.

Il est nécessaire de construire une nouvelle copie du corps de la λ-abstraction, pour que celle-ci puisse éventuellement être appliquée à d'autres arguments dans le cas où elle est partagée. Ce rôle est rempli par la fonction $\mathcal{I}(e_{body}, \epsilon, \zeta, \mathcal{T})$ qui enrichit le tas $\mathcal{T}$ passé en paramètre avec une nouvelle instance de e_{body} dans laquelle toutes les occurrences de $x \in \mathsf{dom}\,\epsilon$ sont remplacées par $\epsilon(x)$. L'adresse de la nouvelle instance est retournée. Le tas et l'environnement global sont alors mis à jour. L'environnement ϵ est utilisé de manière temporaire pour associer le paramètre formel avec l'adresse de l'argument.

Pour que les expressions partagées ne soient réduites qu'une seule fois (évaluation paresseuse), il est important de remplacer physiquement la racine du redex par le résultat. Un moyen simple de réaliser cette opération consiste à mettre à jour la racine du redex avec un nœud d'indirection pointant vers le résultat retourné par $\mathcal{I}$. C'est ce qu'exprime la condition $\mathcal{T}'(\beta) = \nabla^{\beta}\mathcal{T}(\alpha_r)$. La définition formelle de la fonction $\mathcal{I}(e, \epsilon, \zeta, \mathcal{T})$ est donnée dans la figure 5.5.

$$\text{M.BETA}\quad \frac{\mathcal{T}(\alpha) = e^{\alpha}_{lambda} \quad \mathcal{T}(\beta) = e^{\alpha}_{lambda} @^{\beta} e^{\gamma}_{arg} \quad e^{\alpha}_{lambda} = \lambda^{\alpha}\,\langle x\rangle.e_{body} \qquad \epsilon = \{x \mapsto \gamma\} \quad (\alpha_r, \mathcal{T}', \zeta') = \mathcal{I}(e_{body}, \epsilon, \zeta, \mathcal{T}) \quad \mathcal{T}'(\beta) = \nabla^{\beta}\mathcal{T}(\alpha_r)}{\mathbb{S} = (\alpha :: \beta :: \mathcal{P}, \zeta, \mathcal{T}) \longrightarrow_{\beta} \mathbb{S}' = (\beta :: \mathcal{P}, \zeta', \mathcal{T}')}$$

La dernière règle de l'évaluateur permet d'éliminer les nœuds d'indirection introduits lors de l'application de la règle BETA. En sommet de pile, l'adresse α du nœud d'indirection est remplacée par l'adresse β du nœud ainsi référencé.

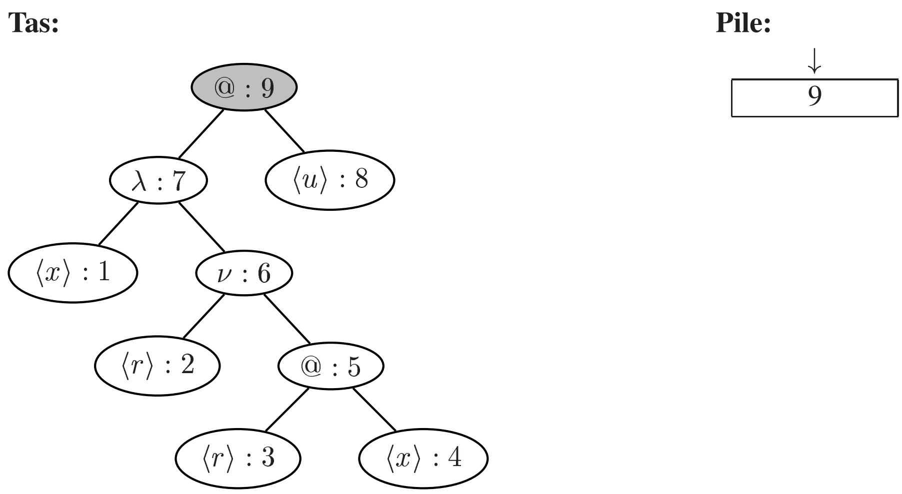
$$\text{M.INDIR}\quad \frac{\mathcal{T}(\alpha) = \nabla^{\alpha} e^{\beta}}{\mathbb{S} = (\alpha :: \mathcal{P}, \zeta, \mathcal{T}) \longrightarrow_{\beta} \mathbb{S}' = (\beta :: \mathcal{P}, \zeta, \mathcal{T})}$$

Un exemple d'évaluation

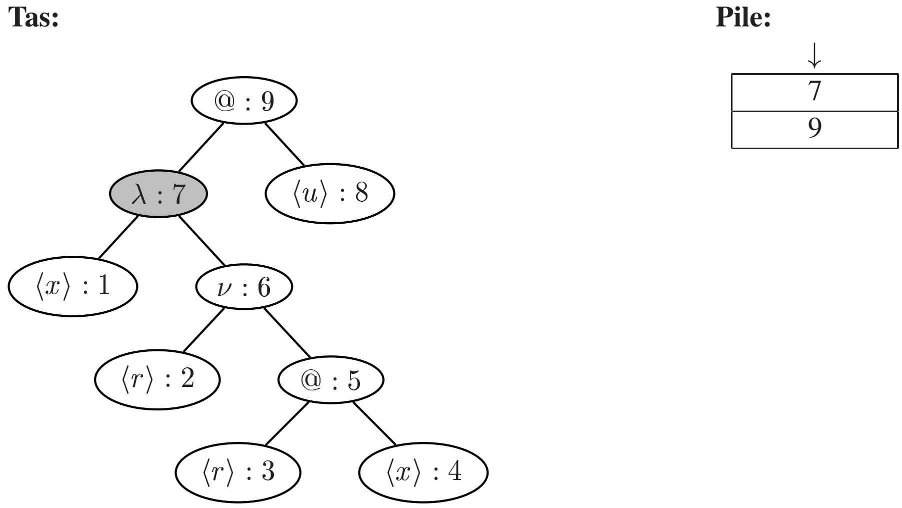
Nous illustrons le fonctionnement de l'évaluateur en détaillant les étapes d'évaluation de l'expression E suivante:

$$E \;=_{def}\; ((\lambda\, x.(\nu\; r)\; (r\; x))\; u)$$

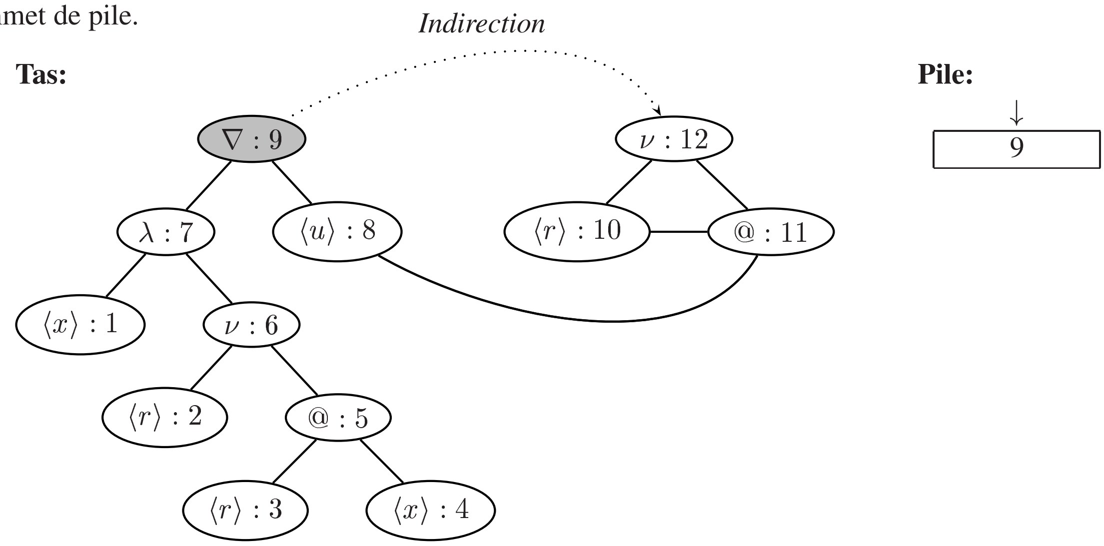
On suppose que la référence u a déjà été introduite par l'opérateur ν dans le courant de l'exécution principale. Le résultat de chaque étape d'évaluation est représentant en détaillant l'état courant du tas $\mathcal{T}$ et de la pile $\mathcal{P}$ de l'évaluateur. Chaque nœud du tas est présenté sous forme d'un couple $contenu$: $adresse$. Le sommet de pile est représenté en grisé.



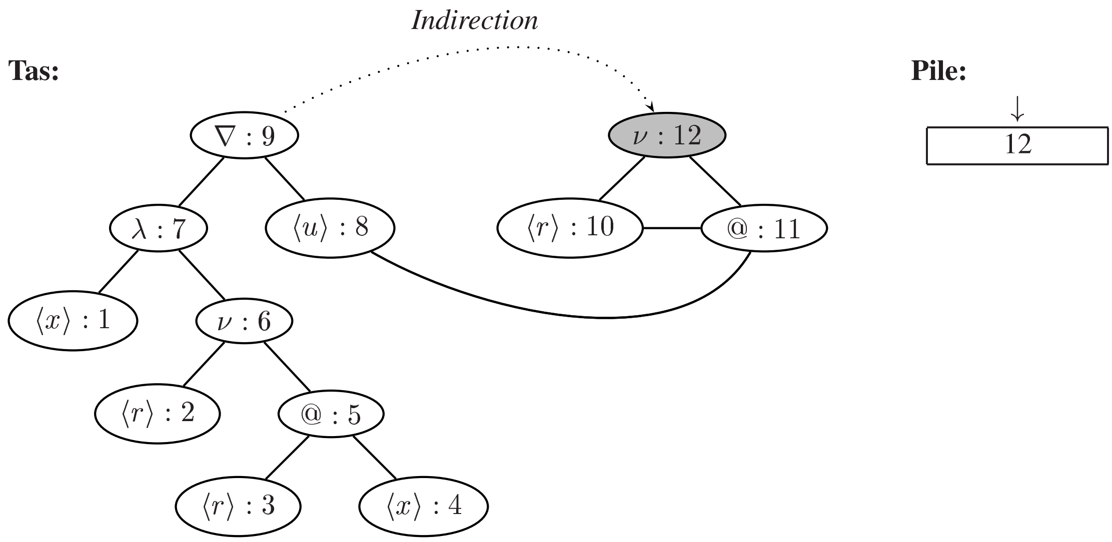
La racine de l'arbre courant est un nœud d'application, ce qui autorise le déclenchement de la règle UNWIND. Cette règle a pour effet de rajouter en sommet de pile l'adresse du nœud présent à l'adresse 7. L'état de l'évaluateur est alors:



Le nœud situé à l'adresse 7, référencé par le sommet de pile, est une λ-abstraction utilisée dans un contexte d'application (nœud d'adresse 9). Cette configuration autorise le déclenchement de la règle BETA, qui a pour effet d'instancier le corps de la λ-abstraction remplaçant son paramètre formel (nœud d'adresse 1) par la valeur de l'argument (nœud d'adresse 8). Cette opération conduit à l'introduction d'un nouveau nœud d'adresse 12 dans le tas, qui devient la racine d'un sous-graphe représentant le corps instancié de la λ-abstraction (nœuds d'adresses 8, 10, 11 et 12). Le nœud d'application situé l'adresse 9 est alors remplacé par une indirection vers le nœud d'adresse 12 nouvellement créé. Ce nœud d'indirection devient alors le sommet de pile.



Suite à l'application de la règle BETA, le sommet de pile pointe vers un nœud d'indirection vers un sous-graphe. La règle INDIR s'applique alors. Elle n'a pour effet que de résoudre l'indirection, sans modifier l'état du tas de l'évaluateur:



A l'issue de l'application de la règle INDIR, le sommet de pile pointe vers un nœud (d'adresse 12) qui n'est plus β-réductible. Il est alors retraduit récursivement en processus au moyen de la traduction $[\![\cdot]\!]_{unload}$. Le processus résultant – $(\nu\ r)\ (r\ u)$ – est finalement réinjecté dans l'état du moteur d'exécution.

Propriétés

Nous conjecturons la correction de l'évaluateur présenté par rapport à un λ–calcul avec appel par nom:

Conjecture 5.1 (Correction de l'évaluateur). *Soit* $E \in \mathsf{Expr}$. *Si* $[\![E]\!]_{load} \Longrightarrow_\beta \mathbb{S}$, *alors il existe* $P \in \mathsf{Pr}$ *tel que* $P = [\![\mathbb{S}]\!]_{unload}$ *et* $E \overset{*}{\longrightarrow} P$ *par des réductions* BETA *du* M_{loc}*-calcul.*

La preuve devrait être similaire à ce que prouve Plotkin dans [143] pour un évaluateur de λ–calcul avec appel par valeur (machine SECD) par rapport à ISWIM, et pour un évaluateur de λ–calcul avec appel par nom (machine SECD modifiée pour l'appel par nom) par rapport à ISWIM avec appel par nom. [11] prouve le même type de propriété pour un λ–calcul avec appel par nécessité. Les propriétés à prouver prennent alors la forme:

Théorème 5.1.

1. *Soit* $\longrightarrow_N$ *la relation de réduction du* λ*–calcul avec appel par nom, et* $Eval_N(M)$ *la fonction d'évaluation d'un terme de ce calcul. Alors:*

   $Eval_N(M) = N$ *ssi* $M \overset{*}{\longrightarrow}_N N$ *pour tout terme fermé* M *du calcul.*

2. *Soit* $\longrightarrow_V$ *la relation de réduction du* λ*–calcul avec appel par valeur, et* $Eval_V(M)$ *la fonction d'évaluation d'un terme de ce calcul. Alors:*

   $Eval_V(M) = N$ *ssi* $M \overset{*}{\longrightarrow}_V N$ *pour tout terme fermé* M *du calcul.*

NœUds purs:

$$
\begin{array}{rcl}
\mathcal{I}(\langle x\rangle,\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\epsilon(x),\mathcal{T}^0,\zeta^0) \quad x \in \mathsf{dom}\,(\epsilon) \\
\mathcal{I}(\langle r\rangle,\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\epsilon(r),\mathcal{T}^0,\zeta^0) \quad r \in \mathsf{dom}\,(\epsilon) \\
\mathcal{I}(\langle r\rangle,\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\zeta^0(r),\mathcal{T}^0,\zeta^0) \quad r \notin \mathsf{dom}\,(\epsilon) \quad r \in \mathsf{dom}\,\zeta^0 \\
\mathcal{I}(\langle r\rangle,\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\zeta^1(r),\mathcal{T}^1,\zeta^1) \quad r \notin \mathsf{dom}\,(\epsilon) \quad r \notin \mathsf{dom}\,\zeta^0 \\
 & & (\beta,\mathcal{T}^1) = \mathcal{F}^{\mathcal{T}^0}[\langle r\rangle] \quad \zeta^1 = \zeta^0\{r \mapsto \beta\} \\
\mathcal{I}(\lambda\ \langle x\rangle.e_{body},\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^2}[\lambda\ \mathcal{T}^2(\alpha).\mathcal{T}^2(\beta)],\zeta^1) \quad (\alpha,\mathcal{T}^1) = \mathcal{F}^{\mathcal{T}^0}[\langle x\rangle] \\
 & & (\beta,\mathcal{T}^2,\zeta^1) = \mathcal{I}(e_{body},\epsilon\{x \mapsto \alpha\},\zeta^0,\mathcal{T}^1) \\
\mathcal{I}(e_1@e_2,\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^2}[\mathcal{T}^2(\alpha_1)@\mathcal{T}^2(\alpha_2)],\zeta^2) \\
 & & (\alpha_i,\mathcal{T}^i,\zeta^i) = \mathcal{I}(e_i,\epsilon,\zeta^{i-1},\mathcal{T}^{i-1}) \quad i = 1,2 \\
\mathcal{I}(\nabla e,\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^1}[\nabla\mathcal{T}^1(\alpha)],\zeta^1) \quad (\alpha,\mathcal{T}^1,\zeta^1) = \mathcal{I}(e,\epsilon,\zeta^0,\mathcal{T}^0) \\
\mathcal{I}(\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ ,\mathcal{R}),\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^0}[\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}\ ,\ \mathcal{H}\ ,\mathcal{R})],\zeta^0) \\
\mathcal{I}(\mathsf{reify}_{\mathsf{content}}(\mathcal{E}\ ,\ \mathcal{L}),\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^0}[\mathsf{reify}_{\mathsf{content}}(\mathcal{E}\ ,\ \mathcal{L})],\zeta^0)
\end{array}
$$

NœUds impurs:

$$
\begin{array}{rcl}
\mathcal{I}(\mathsf{0},\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^0}[\mathsf{0}],\zeta^0) \\
\mathcal{I}(e_1[e_2],\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^2}[\mathcal{T}^2(\alpha_1)[\mathcal{T}^2(\alpha_2)]],\zeta^2) \\
 & & (\alpha_i,\mathcal{T}^i,\zeta^i) = \mathcal{I}(e_i,\epsilon,\zeta^{i-1},\mathcal{T}^{i-1}) \quad i = 1,2 \\
\mathcal{I}((e_1 \mid e_2),\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^2}[(\mathcal{T}^2(\alpha_1) \mid \mathcal{T}^2(\alpha_2))],\zeta^2) \\
 & & (\alpha_i,\mathcal{T}^i,\zeta^i) = \mathcal{I}(e_i,\epsilon,\zeta^{i-1},\mathcal{T}^{i-1}) \quad i = 1,2 \\
\mathcal{I}((\nu\ \langle r\rangle)\ e_{body},\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^2}[(\nu\ \mathcal{T}^2(\alpha))\ \mathcal{T}^2(\beta)],\zeta^1) \quad (\alpha,\mathcal{T}^1) = \mathcal{F}^{\mathcal{T}^0}[\langle r\rangle] \\
 & & (\beta,\mathcal{T}^2,\zeta^1) = \mathcal{I}(e_{body},\epsilon\{r \mapsto \alpha\},\zeta^0,\mathcal{T}^1) \\
\mathcal{I}(([e_1\ =\ e_2]\ e_3\ ,\ e_4),\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^4}[[\mathcal{T}^4(\alpha_1) = \mathcal{T}^4(\alpha_2)]\ \mathcal{T}^4(\alpha_3),\mathcal{T}^4(\alpha_4)],\zeta^4) \\
 & & (\alpha_i,\mathcal{T}^i,\zeta^i) = \mathcal{I}(e_i,\epsilon,\zeta^{i-1},\mathcal{T}^{i-1}) \quad i = 1,\ldots,4 \\
\mathcal{I}(\langle e_1 \mid \ldots \mid e_n = f\rangle,\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & \left(\mathcal{F}^{\mathcal{T}^{n+1}}\left[\left\langle \begin{array}{l} \mathcal{T}^{n+1}(\alpha_1) \mid \ldots \mid \\ \mathcal{T}^{n+1}(\alpha_n) = \mathcal{T}^{n+1}(\beta) \end{array} \right\rangle\right],\zeta^{n+1}\right) \\
 & & (\alpha_i,\mathcal{T}^i,\zeta^i) = \mathcal{I}(e_i,\epsilon,\zeta^{i-1},\mathcal{T}^{i-1}) \quad i = 1,\ldots,n \\
 & & (\beta,\mathcal{T}^{n+1},\zeta^{n+1}) = \mathcal{I}(f,\epsilon,\zeta^n,\mathcal{T}^n) \\
\mathcal{I}(\mathsf{pass}\ e,\epsilon,\zeta^0,\mathcal{T}^0) & =_{def} & (\mathcal{F}^{\mathcal{T}^1}[\mathsf{pass}\ \mathcal{T}^1(\alpha)],\zeta^1) \quad (\alpha,\mathcal{T}^1,\zeta^1) = \mathcal{I}(e,\epsilon,\zeta^0,\mathcal{T}^0)
\end{array}
$$

FIG. *5.5* – **Fonction d'instantiation $\mathcal{I}(e,\epsilon,\zeta,\mathcal{T})$ de nouveaux nœuds du tas.**

5.3 CLAM: une machine abstraite répartie pour le M–calcul

5.3.1 Architecture générale

La machine abstraite détaillée dans la section précédente est basée sur un double principe de centralisation des ressources d'exécution et de communication:

- L'ensemble des localités s'exécute sur une seule machine physique.
- Les ressources de communication sont centralisées. Il s'agit du service de création de nouveaux identificateurs – noms de localité et noms de canaux – et du service de nommage qui permet de localiser les récepteurs en maintenant un lien entre un nom de ressource et la localité qui le définit. Cette centralisation des ressources implique une synchronisation entre les différentes localités.

Ainsi, l'ensemble des identificateurs manipulés est géré au sein d'un seul contexte de nommage global. La machine maintient une table unique des localités du système qui est une ressource partagée entre toutes les localités: toute création d'un nouveau nom de localité se traduit par la création d'une nouvelle entrée dans la table.

Obtenir une machine véritablement répartie signifie simultanément:

- *Répartir les ressources d'exécution:* les manifestations de domaines comme les cellules doivent pouvoir s'exécuter sur des machines physiques distinctes et communiquer via les couches réseau.
- *Répartir les ressources de communication:* les services de communication nécessaires à l'établissement de liaisons réparties doivent être distribués sur les différentes machines physiques impliquées. Une telle architecture décentralisée permet d'éviter des synchronisations réparties inutiles entre localités. Elle signifie la mise en oeuvre de plusieurs contextes de nommage.

Ces deux exigences rendent nécessaire l'introduction d'un niveau supplémentaire de partitionnement des ressources par rapport à la machine abstraite décrite dans la section précédente. On obtient alors la machine abstraite répartie CLAM. Une *machine* m y constitue l'unité d'exécution de plus haut-niveau et abrite une collection de *localités*. Au niveau implémentation, une machine est l'abstraction d'un processus système, comme celui dans lequel s'exécutera la machine virtuelle implantant la spécification CLAM. Les localités peuvent alors être vues comme des groupes d'objets actifs de cette machine virtuelle. Les machines sont reliées par un *éther global* E qui regroupe l'ensemble des messages $M_1 \mid \cdots \mid M_p$ en transit entre machines.

Pour effectuer le routage des messages, chaque machine dispose d'un service de nommage qui permet de localiser les ressources de communication (dans quelle localité est définie une ressource donnée?) et d'exécution (dans quelle machine réside une localité donnée?). On parlera de *routage local* ou *distant*, selon que les messages sont acheminés au sein d'une même machine ou entre machines distinctes.

Chaque machine abrite une arborescence de localités. Leur structure est identique à celle décrite dans la section précédente. Pour conserver un modèle hiérarchique cohérent, il est utile d'introduire une localité virtuelle $\top$ dont les filles sont les différentes localités-racine (localités de plus haut niveau) de chaque machine. Le système global se présente alors comme une forêt de domaines/cellules/localités

$$\top\ (\mathbf{0})\ \big[\ E(\mathbf{0})\ [M_1 \mid \cdots \mid M_p\ ] \mid m_1\ [\ \mathbf{0}\ ](\ a_1(P_1[Q_1])\ ) \mid \cdots \mid m_n\ [\ \mathbf{0}\ ](\ a_n(P_n[Q_n])\ )\big]$$

dont la structure d'ensemble est celle de la figure 5.6.

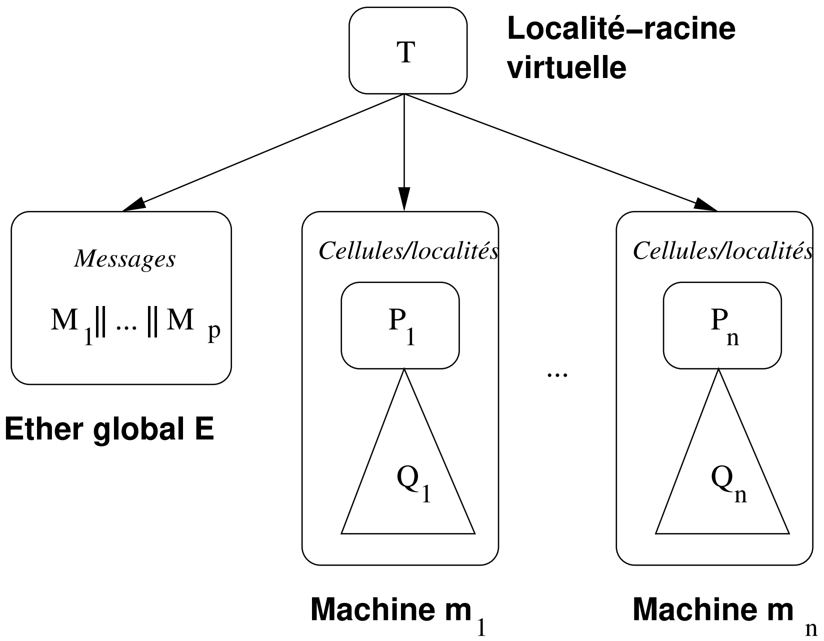


FIG. *5.6* – **Modèle de répartition de la CLAM.**

5.3.2 Etat de la machine

Contrairement à la machine de la section précédente qui s'était restreinte au M_{loc}–calcul, CLAM spécifie une implémentation du M–calcul complet. On se donne tout d'abord quelques catégories syntaxiques supplémentaires permettant de prendre en compte les différences entre le M_{loc}–calcul et le M–calcul, comme les cellules explicitement nommées $a(P)[Q]$, les messages explicitement adressés $a.r\tilde{V}$, les filtres de messages du join–calcul, ou les définitions multiples. Certains ensembles, déjà définis dans le cas centralisé, sont ici redéfinis.

Définition 5.9 (Catégories syntaxiques de base). *On suppose donnés les ensembles suivants :*

$a, b, \ldots$	$\in$	Cell	*Nom de cellule*
$n, \ldots$	$\in$	$\mathsf{Ref} \cup \mathsf{Cell}$	*Nom simple*
$\mathsf{a} \ldots$	$\in$	$\mathsf{VarCell} = \mathsf{Cell} \cup \mathsf{Var}$	*Nom variable de cellule*
$\mathsf{r} \ldots$	$\in$	$\mathsf{VarRef} = \mathsf{Ref} \cup \mathsf{Var}$	*Nom variable de ressource*
$d, \bar{\mathsf{a}}, \underline{\mathsf{a}} \ldots$	$\in$	$\mathsf{Dest} = \mathsf{VarCell}$	*Cible d'un message*
$d.\mathsf{r}, \ldots$	$\in$	$\mathsf{AdrRef} = \mathsf{Dest} \times \mathsf{VarRef}$	*Nom de ressource adressé*
$u, \ldots$	$\in$	$\mathsf{Name} = \mathsf{VarCell} \cup \mathsf{VarRef} \cup \mathsf{AdrRef}$	*Nom*
$d.r\tilde{V} \ldots$	$\in$	$\mathsf{Msg} = \mathsf{AdrRef} \times \mathsf{Val}^*$	*Message*
$J, J' \ldots$	$\in$	$\mathsf{Join} = Set(\mathsf{Ref} \times \mathsf{Var}^*)$	*Filtre de messages*
$D, D' \ldots$	$\in$	$\mathsf{Def} = Set(\mathsf{Join} \times \mathsf{Pr})$	*Définition*

Dans la machine abstraite répartie :

- Chaque machine est dotée d'un nom m, unique à l'échelle du système réparti. Ce nom est un élément de l'ensemble MName.

- Les noms internes jouent toujours un rôle important pour désigner dans la machine les entités venant du calcul comme les ressources ou les cellules. Ils sont typiquement utilisés au sein d'environnements où les noms du calcul sont associés à des noms internes. Cependant, en contexte réparti, on utilise pleinement le caractère « riche » de ces noms qui contiennent désormais des informations de localisation: chaque nom interne est lié à un contexte de nommage correspondant à la machine qui l'a créé. Les noms internes utilisent les catégories syntaxiques suivantes:

Définition 5.10 (Noms internes). *On suppose donnés les ensembles suivants :*

$l_{loc}, \ldots$	$\in$	LocLocNm	*Nom de localité « local »*
$l = \langle m, l_{loc} \rangle, l', \daleth \ldots$	$\in$	LocNm = MName × LocLocNm	*Nom de localité*
$i_{loc}, \ldots$	$\in$	LocIntNm	*Nom interne « local »*
$i = \langle m, i_{loc} \rangle \ldots$	$\in$	IntNm = MName × LocIntNm	*Nom interne*

Par rapport au cas centralisé:

- L'état d'une localité est quasiment inchangé: une localité l est décrite par un tuple $\langle \alpha, \mathcal{D}, \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{L} \rangle$ où α est le nom correspondant de la cellule au niveau calcul, et n'est défini que pour des localités de type membrane. Les autres paramètres du tuple ont la même signification que dans le cas centralisé.
- Une machine intègre localement un certain nombre de services qui faisaient précédemment partie de la description globale du système. Une machine m est représentée par un tuple $\langle m : N : O : L \rangle$. N est un service de création de nouveaux noms internes. O est le service de nommage local de la machine. Enfin, L contient la liste de toutes les localités hébergées par la machine courante et associe des noms de localité à des tuples $\langle \alpha, \mathcal{D}, \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{L} \rangle$.

Dans le service de nommage, chaque information de localisation est associée à une estampille pour caractériser la « fraîcheur » de l'information. Pour localiser les ressources et les localités dans le système, le service de nommage associe des noms internes de ressource à un ensemble de noms de localité, et des noms internes de localité à des noms de machine. On notera qu'un même nom de ressource peut comporter plusieurs entrées dans le service de nommage, puisqu'il est possible de définir en M–calcul des ressources portant le même nom mais situées dans des localités différentes.

Définition 5.11 (Etat d'une machine). *Formellement, la description de l'état d'une machine nécessite les catégories syntaxiques suivantes:*

$t, t' \ldots$	$\in$	TimeStamp	*Estampille*
α, α'	$\in$	CellName	*Nom interne de cellule*
$\mathbb{L}$	$\in$	LocState = CellName × DefSpace × Heap × RQueue × Ether × SubLoc	*Etat d'une localité*
N	$\in$	NameFactory = Set(LocLocNm ∪ LocIntNm)	*Création de nouveaux noms*
O	$\in$	LookupSrv = (IntNm ↦ Set(LocNm × TimeStamp)) ∪ (LocNm ↦ (MName × TimeStamp))	*Service de nommage*
L	$\in$	Locs = LocNm ↦ LocState	*Arborescence de localités*
$\mathbb{M}$	$\in$	MState = MName × NameFactory × LookupSrv × Locs	*Etat d'une machine*
$msg(d.\mathsf{r}, \widetilde{V}, \rho, O)$	$\in$	CMsg = AdrRef × Val* × Env × LookupSrv	*Message inter-machines*
$\mathcal{M}, \mathcal{M}'$	$\in$	MsgQueue = $Queue$(CMsg*)	*File globale de messages*

Dans la manipulation du service de nommage, on utilise la notation $l \Subset O(\rho(r))$ pour :

$$\exists t \in \mathsf{TimeStamp} \text{ tel que } \langle l, t \rangle \in O(\rho(r))$$

La négation est notée $l \not\Subset O(\rho(r))$. On note aussi $O\ \{\rho(r) \mapsto^{+} \langle l, 0 \rangle\}$ la formule $O\ \{\rho(r) \mapsto O(\rho(r)) \cup \{\langle l, 0 \rangle\}\}$.

5.3.3 Règles de réduction de base

L'évolution d'une localité est décrite par des règles de réduction qui concernent soit l'activité à l'intérieur d'une machine donnée, soit l'échange de messages (envoi ou réception) d'une machine avec l'éther global. Par exemple, on écrira

$$\begin{aligned} & m : N : O : L\{l \xmapsto[l_e:\mathcal{L}]{\mathcal{D}:\mathcal{H}} \langle \alpha, \mathcal{R} \rangle, \ldots\} \parallel E\{m \to m' \mapsto \mathcal{M}\} \longrightarrow \\ & m : N' : O' : L'\{l \xmapsto[l_e:\mathcal{L}']{\mathcal{D}':\mathcal{H}'} \langle \alpha, \mathcal{R}' \rangle, \ldots\} \parallel E\{m \to m' \mapsto \mathcal{M} :: msg\} \end{aligned}$$

pour une machine m où : (i) l'état $\langle \alpha, \mathcal{D}, \mathcal{H}, \mathcal{R}, l_e, \mathcal{L} \rangle$ d'une localité l devient $\langle \alpha, \mathcal{D}', \mathcal{H}', \mathcal{R}', l_e, \mathcal{L}' \rangle$; (ii) un message msg est envoyé sur le réseau (éther global E) de la machine m vers la machine m'.

M.NIL
$$\frac{}{m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, (\rho, \mathbf{0}) :: \mathcal{R} \rangle\} \longrightarrow m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, \mathcal{R} \rangle\}}$$

M.PRL
$$\frac{}{m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, (\rho, (P \mid Q)) \rangle\} \longrightarrow m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, (\rho, P) :: \mathcal{R} :: (\rho, Q) \rangle\}}$$

M.IF.THEN
$$\frac{\mathit{match}(\rho(\mu), \rho(V))}{m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, (\rho, ([\mu = V]\, P\ ,\ Q)) :: \mathcal{R} \rangle\} \longrightarrow m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, (\rho, P) :: \mathcal{R} \rangle\}}$$

M.IF.ELSE
$$\frac{\neg \mathit{match}(\rho(\mu), \rho(V))}{m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, (\rho, ([\mu = V]\, P\ ,\ Q)) :: \mathcal{R} \rangle\} \longrightarrow m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, (\rho, Q) :: \mathcal{R} \rangle\}}$$

M.EVAL
$$\frac{\mathbb{S}_l = [\![E]\!]_{load} \qquad \mathbb{S}_l \Longrightarrow_\beta \mathbb{S}'_l \qquad P = [\![\mathbb{S}'_l]\!]_{unload}}{m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, (\rho, E) :: \mathcal{R} \rangle\} \longrightarrow m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, \mathcal{R} :: (\rho, P) \rangle\}}$$

M.NEW
$$\frac{\rho' = \rho\{n \mapsto i\} \quad i = \langle m, i_{loc} \rangle \quad i_{loc} \notin N}{m : N : O : L\{l \xmapsto[-]{-} \langle \alpha, (\rho, (\nu\, n)\, P) :: \mathcal{R} \rangle\} \longrightarrow m : N \oplus \{i_{loc}\} : O : L\{l \xmapsto[-]{-} \langle \alpha, (\rho', P) :: \mathcal{R} \rangle\}}$$

FIG. *5.7* – **CLAM : Cœur séquentiel.**

Les règles du cœur séquentiel représentées dans la figure 5.7 ne subissent pratiquement pas de changement par rapport à la version centralisée. A noter dans les règles IF.THEN et IF.ELSE l'utilisation du prédicat $match$ défini par le M–calcul. Dans la règle NEW, la création de nouveaux noms internes devient un service local à la machine courante, plutôt que d'être unique au niveau du système.

Les seuls changements en terme de définitions et de communication locale, représentés dans la figure 5.8, concernent la prise en compte des définitions multiples et des filtres de messages où les noms définis possèdent des arguments, ce qui modifie légèrement les règles DEF et COMM.

Pour la création de cellules (cf. figure 5.9), il faut à présent associer dans le service de nommage les localités nouvellement créées avec la machine courante comme le montre la règle CELL. Toutefois, cette règle n'est pas applicable au cas où le processus de cellule $b(P)[Q]$ se trouve dans la file d'exécution d'une localité racine: lors de son initialisation, une machine n'héberge qu'une localité-racine $l_\top$ ne contenant dans

$$\text{M.DEF} \quad \frac{\begin{array}{c}\rho' = \rho\restriction \bigcup_{J_i \triangleright P_i \in D} fn(P_i) \setminus rn(J_i) \quad \mathcal{D}' = \mathcal{D} \oplus (\rho', D) \\ \mathcal{H}' = \mathcal{H}\ \{\rho(r) \mapsto \bullet\}_{r \in dn(D)} \quad O' = O\ \{\rho(r) \mapsto^{+} \langle l, 0\rangle\}_{r \in dn(D)}\end{array}}{m : N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}} \langle \alpha, (\rho, \langle D\rangle) :: \mathcal{R}\rangle\} \longrightarrow m : N : O' : L\{l \xmapsto[\_]{\mathcal{D}':\mathcal{H}'} \langle \alpha, \mathcal{R}\rangle\}}$$

$$\text{M.MSG} \quad \left\{\begin{array}{l} \dfrac{\mathcal{H}(\rho(r)) = \mathcal{V} \quad \mathcal{H}' = \mathcal{H}\ \{\rho(r) \mapsto \mathcal{V} :: (\rho, \widetilde{V})\}}{m : N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}} \left\langle \alpha, (\rho, r\widetilde{V}) :: \mathcal{R}\right\rangle\} \longrightarrow m : N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}'} \langle \alpha, \mathcal{R}\rangle\}} \\ \\ \dfrac{\rho(r) \notin \mathsf{dom}\,\mathcal{H}}{m : N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}} \left\langle \alpha, (\rho, r\widetilde{V}) :: \mathcal{R}\right\rangle\} \longrightarrow m : N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V})\right\rangle\}} \end{array}\right.$$

$$\text{M.COMM} \quad \frac{\begin{array}{c}(\rho, D) \in \mathcal{D} \quad D = \ldots; J \triangleright P; \ldots \quad J = r_1\widetilde{x_1} \mid \ldots \mid r_q\widetilde{x_q} \\ \left[\mathcal{H}(\rho(r_i)) = (\rho_i, \widetilde{V_i}) :: \mathcal{V}_i\right]_{1 \leq i \leq q} \quad \rho' = \left[\rho\ \cup \bigcup_{1 \leq i \leq q} \rho_i\right]\{\widetilde{x_i} \mapsto \widetilde{V_i}\}_{1 \leq i \leq q} \quad \mathcal{H}' = \left[\mathcal{H}\ \{\rho(r_i) \mapsto \mathcal{V}_i\}\right]_{1 \leq i \leq q}\end{array}}{m : N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}} \langle \alpha, \mathcal{R}\rangle\} \longrightarrow m : N : O : L\{l \xmapsto[\_]{\mathcal{D}:\mathcal{H}'} \langle \alpha, \mathcal{R} :: (\rho', P)\rangle\}}$$

FIG. *5.8* – **CLAM : Définitions et communication locale.**

$$\text{M.CELL} \quad \frac{\begin{array}{c} l' = \langle m, l'_{loc}\rangle \quad l'' = \langle m, l''_{loc}\rangle \quad l'_{loc} \notin N \quad l''_{loc} \notin N \\ \rho' = closure(\rho, P) \quad \rho'' = closure(\rho, Q) \quad \mathcal{L}'_0 = \mathcal{L}_0 \cup \{l'\}\end{array}}{\begin{array}{c} m : N : O : L\{l_0 \xmapsto[l:\mathcal{L}_0]{\_} \langle \alpha_0, \mathcal{R}_0\rangle, l \xmapsto[\_]{\_} \langle \sharp, (\rho, b(P)[Q]) :: \mathcal{R}\rangle\} \longrightarrow \\ m : N \oplus \{l'_{loc}, l''_{loc}\} : O \oplus \{l' \mapsto \langle m, 0\rangle, l'' \mapsto \langle m, 0\rangle\} : \\ L\{l_0 \xmapsto[l:\mathcal{L}'_0]{\_} \langle \alpha_0, \mathcal{R}_0\rangle, l \xmapsto[\_]{\_} \langle \sharp, \mathcal{R}\rangle, l' \xmapsto[l'':\varnothing]{\varnothing:\varnothing} \langle \rho(b), (\rho', P)\rangle, l'' \xmapsto[\sharp:\varnothing]{\varnothing:\varnothing} \langle \sharp, (\rho'', Q)\rangle\}\end{array}}$$

$$\text{M.CELL}_\top \quad \frac{l = \langle m, l_{loc}\rangle \quad l_{loc} \notin N \quad \rho' = closure(\rho, P) \quad \rho'' = closure(\rho, Q)}{\begin{array}{c} m : N : O : L\{l_\top \xmapsto[\sharp:\varnothing]{\_} \langle \sharp, (\rho, b(P)[Q])\rangle\} \longrightarrow \\ m : N \oplus \{l_{loc}\} : O \oplus \{l \mapsto \langle m, 0\rangle\} : L\{l_\top \xmapsto[l:\varnothing]{\_} \langle \rho(b), (\rho', P)\rangle, l \xmapsto[\sharp:\varnothing]{\varnothing:\varnothing} \langle \sharp, (\rho'', Q)\rangle\}\end{array}}$$

FIG. *5.9* – **CLAM : Création de nouvelles cellules.**

sa file d'exécution qu'un processus de cellule. Ce dernier est destiné à déclencher la création progressive d'un arbre de localités dans la machine courante au cours des réductions suivantes. Cette opération est amorcée en créant une nouvelle localité pour accueillir le contenu de la cellule de plus haut-niveau dont la membrane est représentée par la localité-racine (règle CELL$_\top$).

Nous décrivons à présent les autres règles de la machine abstraite qui concernent la communication distante et la migration.

5.3.4 Communication distante et mobilité

Dans la CLAM, le routage des messages s'appuie sur un service de nommage réparti. Ce service est considéré ici comme une base de données répartie. Les différentes instances du service de nommage, locales à une machine, constituent des vues partielles de cette base. Différentes politiques de mise à jour sont envisageables: par exemple, en maintenant des vues cohérentes de la base de données par diffusion atomique, ou en autorisant certaines vues à être temporairement incohérentes – mais en utilisant des liens de poursuite pour permettre aux messages d'atteindre correctement leur destination finale.

La base de données doit satisfaire la propriété suivante:

Définition 5.12 (Cohérence du service de nommage). *Un service de nommage réparti est dit* cohérent *si et seulement si, en cas de vues temporairement incohérentes de la base de données correspondante, la seule conséquence sur le routage, est une émission ou une réception retardée des messages, en supposant que les communications entre machines soient fiables, i.e., l'éther global ne détruit pas de messages en transit.*

La machine abstraite CLAM utilise une formalisation particulière pour représenter la base de données répartie: chaque fragment de la base de données est constitué du service de nommage O local à une machine qui permet de localiser définitions et localités. Un algorithme de routage particulier, noté R*, est mis en oeuvre dans la machine abstraite à titre illustratif. Cependant, d'autres algorithmes de routage peuvent être utilisés en modifiant de manière minimale la spécification formelle.

R* utilise des liens de poursuite et du « piggybacking » sur les messages qui transitent entre machines pour réaliser la mise à jour des informations de routage. Des estampilles représentent les différents instants de migration des cellules ou des définitions, afin d'éviter des incohérences de communication à cause d'informations de routage contenues dans le service de nommage qui ne soient pas à jour.

R* est capturé par deux types de règles de réduction, respectivement pour les messages locaux et les messages explicitement adressés. Ces règles sont rassemblées dans les figures 5.11, 5.12 et 5.13. La figure 5.10 résume le mouvement des messages entre les localités – représentées par des losanges – à la fois pour les messages locaux et explicitement adressés. La figure montre également les frontières de cellules en terme de membrane et de contenu. Ce sont ces frontières qui donnent leur nom aux règles de réduction, selon qu'un message va/provient de l'extérieur d'une cellule (E), de sa membrane (M) ou de son contenu (C), comme dans la règle M.L.CM qui décrit la migration d'un message local depuis le contenu d'une cellule vers sa membrane.

Les règles de routage des messages locaux implémentent fidèlement les règles associées du modèle de programmation. Par exemple, la règle M.L.MC de la machine abstraite implémente la règle R.L.MC du calcul. Elle permet aux messages de descendre dans l'arborescence des localités: par rapport à sa position courante, un message $r\widetilde{V}$ peut directement migrer vers une sous-localité de type éther si et seulement si cette sous-localité définit le nom de ressource r.

Le routage des messages explicitement adressés comprend:

- Des règles où la communication est (physiquement) locale comme M.A_1.EM.L, M.A_1.EC.L, ou M.A_2.ME.L: les messages changent de localité à l'intérieur d'une même machine.

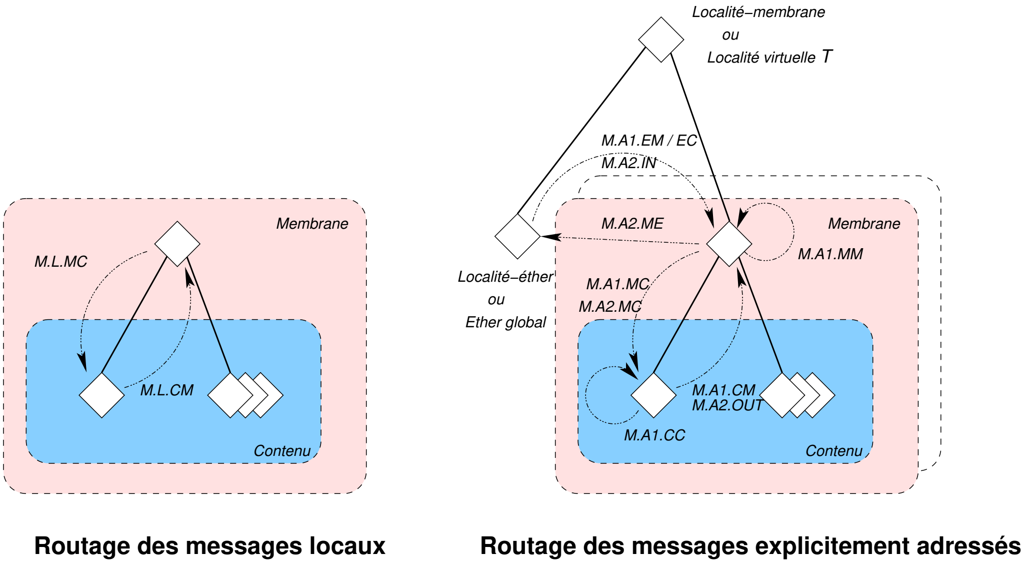


FIG. *5.10* – **Routage des messages dans la CLAM.**

- D'autres règles où l'interaction est (physiquement) distante comme M.A$_1$.EC.R et M.A$_2$.ME.R: les messages changent de localité et de machine. On a alors une communication distante réelle entre machines.

Dans les deux cas, les messages sont routés « tels quels » jusqu'à atteindre leur cellule de destination. A titre d'exemple:

- *Règles (physiquement) locales:* un message dont la cible est une ressource définie dans une localité l et situé dans l'éther intérieur englobant peut directement entrer dans sa localité de destination (règle M.A$_1$.EM.L). Le routage d'un message entre une localité-membrane et l'éther englobant est aussi spécifié par les règles M.A$_2$.IN.L et M.A$_2$.ME.L. La cellule de destination doit simplement être connue du service de nommage.
- *Règles (physiquement) distantes:* considérons un message ayant pour destination une localité physiquement distante. Lorsqu'il atteint la localité-racine de la machine courante, il est emballé avec des informations de routage (environnement courant et état du service de nommage local) qui lui permettront d'atteindre sa destination. Il est alors envoyé sur le réseau vers la machine distante. De plus, des liens de poursuite sont laissés sur la machine de départ pour tenir compte des identificateurs – par exemple, des localités en cours de migration sous forme passivée – présents dans les arguments du message (règle M.A$_2$.ME.R). Lorsque le message atteint la localité-racine $l_\top$ de la machine d'arrivée, le message est déballé et ajouté à la fin de la file d'exécution de $l_\top$ (règle M.A$_1$.EC.R). Le service de nommage est alors actualisé en fusionnant l'information de routage dont il dispose déjà avec celle reçue dans le message. Ces deux règles sont similaires aux règles de communication distante de la JAM [138], machine abstraite pour le join–calcul réparti. Enfin, la règle M.FORWARD traite le cas de messages qui parviennent sur une machine n'hébergeant plus la localité de destination du message à cause d'une migration antérieure. Le message est simplement rerouté vers sa nouvelle destination, compte tenu des informations de routage disponibles.

Définition 5.13 (Mise à jour du service de nommage). *Les fonctions suivantes contribuent à la mise à jour du service de nommage pour obtenir l'information de routage la plus récente:*

$$best(\langle i_0, t_0\rangle, \langle i_1, t_1\rangle) \quad =_{def} \quad \langle i_0, t_0\rangle \textit{ si } t_0 \geq t_1 \ ; \ \langle i_1, t_1\rangle \textit{ si } t_0 \leq t_1$$

$$merge(O_0, O_1)(i), i \in \mathsf{LocNm} \quad =_{def} \quad best(O_0(i), O_1(i)) \textit{ si } \ i \in \mathsf{dom}\, O_0 \cap \mathsf{dom}\, O_1; \quad O_0(i) \textit{ si } i \in \mathsf{dom}\, O_0 \setminus \mathsf{dom}\, O_1 \ ; \ O_1(i) \textit{ si } i \in \mathsf{dom}\, O_1 \setminus \mathsf{dom}\, O_0$$

$$merge(O_0, O_1)(i), i \in \mathsf{IntNm} \quad =_{def} \quad \{merge_0(\alpha_0, \alpha_1), \forall (\alpha_0, \alpha_1) \in \Theta^2\} \quad \textit{avec } \Theta = O_0(i) \cup O_1(i) \textit{ si } i \in \mathsf{dom}\, O_0 \cap \mathsf{dom}\, O_1; \quad O_0(i) \textit{ si } i \in \mathsf{dom}\, O_0 \setminus \mathsf{dom}\, O_1 \ ; \ O_1(i) \textit{ si } i \in \mathsf{dom}\, O_1 \setminus \mathsf{dom}\, O_0$$

$$merge_0(\alpha_0, \alpha_1) \quad =_{def} \quad best(\alpha_0, \alpha_1) \textit{ si } \ \textit{fst}\, \alpha_0 = \ \textit{fst}\, \alpha_1 \ ; \ \alpha_0 \textit{ sinon}$$

$$\textsc{m.l.mc} \quad \frac{l \not\Subset O(\rho(r)) \quad l' \Subset O(\rho(r))}{\begin{array}{c} m : N : O : L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \left\langle \alpha, (\rho, r\widetilde{V}) :: \mathcal{R}\right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\} \longrightarrow \\ m : N : O : L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, \mathcal{R}' :: (\rho, r\widetilde{V})\right\rangle\} \end{array}}$$

$$\textsc{m.m.cm} \quad \frac{l \Subset O(\rho(r)) \quad l' \not\Subset O(\rho(r))}{\begin{array}{c} m : N : O : L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, (\rho, r\widetilde{V}) :: \mathcal{R}'\right\rangle\} \longrightarrow \\ m : N : O : L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V})\right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\} \end{array}}$$

FIG. *5.11* – **Clam : Règles de routage pour les messages locaux.**

Les règles qui décrivent la mobilité des localités (passivation et activation) sont rassemblées dans la figure 5.15. Dans leur principe, elles sont similaires au cas centralisé: la passivation est réalisée à l'aide de la fonction $extract$ qui construit les valeurs $\mathsf{reify}_{\mathsf{membrane}}$ et $\mathsf{reify}_{\mathsf{content}}$ et met à jour la liste L des localités hébergées par la machine courante en faisant disparaître les descendants stricts de la localité migrante. La reconstruction de l'arbre de localités est effectuée après migration par les règles M.MEMBRANE et M.CONTENT. Pour conserver une arborescence de localités sur une machine, la passivation est restreinte au cas d'une localité différente d'une localité-racine.

Lors de la passivation, la mise à jour des informations de routage est retardée jusqu'à l'envoi d'un message sur le réseau depuis la localité-racine de la machine de départ (cf. figure 5.14). Un lien de poursuite est alors laissé sur la machine d'origine. Contrairement à la JAM, le service de nommage ne peut pas être mis à jour avec la nouvelle localisation des ressources migrantes, i.e., celles contenues dans le sous-arbre passivé: au moment de la passivation, la localité dans laquelle sera activée une ressource passivée n'est pas connue (seule la machine cible est connue). Cette localité cible sera déterminée dynamiquement (par les règles de routage physiquement localisées) sur la machine d'arrivée après réception du message.

On modifie alors le service de nommage de départ, pour chaque localité et chaque ressource sous forme passivée contenue dans le message:

$$\text{M.A}_1\text{.MM}\quad \frac{\rho(a)=\alpha \quad l \Subset O(\rho(r))}{\begin{array}{c} m:N:O:L\{l \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, (\rho, \overline{a}.r\widetilde{V}) :: \mathcal{R} \right\rangle\} \longrightarrow \\ m:N:O:L\{l \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V}) \right\rangle\} \end{array}}$$

$$\text{M.A}_1\text{.EM.L}\quad \frac{\rho(a)=\alpha \quad l \Subset O(\rho(r)) \quad siblings(l,l')}{\begin{array}{c} m:N:O:L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R} \rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, (\rho, \overline{a}.r\widetilde{V}) :: \mathcal{R}' \right\rangle\} \longrightarrow \\ m:N:O:L\{l \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V}) \right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}' \rangle\} \end{array}}$$

$$\text{M.A}_1\text{.EM.R}\quad \frac{O_0(\rho(a)) = \langle l_\top, t\rangle \quad l_\top \Subset O(\rho(r)) \quad O' = merge(O, O_0)}{\begin{array}{c} m:N:O:L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R} \rangle\} \parallel E\{m' \to m \mapsto msg(\overline{a}.r, \widetilde{V}, \rho, O_0) :: \mathcal{M}\} \longrightarrow \\ m:N:O':L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V}) \right\rangle\} \parallel E\{m' \to m \mapsto \mathcal{M}\} \end{array}}$$

$$\text{M.A}_1\text{.CM}\quad \frac{\rho(a)=\alpha \quad l \Subset O(\rho(r))}{\begin{array}{c} m:N:O:L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \langle \alpha, \mathcal{R} \rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, (\rho, \overline{a}.r\widetilde{V}) :: \mathcal{R}' \right\rangle\} \longrightarrow \\ m:N:O:L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, r\widetilde{V}) \right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}' \rangle\} \end{array}}$$

$$\text{M.A}_1\text{.CC}\quad \frac{\rho(a)=\alpha \quad l' \Subset O(\rho(r))}{\begin{array}{c} m:N:O:L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \langle \alpha, \mathcal{R} \rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, (\rho, \underline{a}.r\widetilde{V}) :: \mathcal{R}' \right\rangle\} \longrightarrow \\ m:N:O:L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \langle \alpha, \mathcal{R} \rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, \mathcal{R}' :: (\rho, r\widetilde{V}) \right\rangle\} \end{array}}$$

$$\text{M.A}_1\text{.MC}\quad \frac{\rho(a)=\alpha \quad l' \Subset O(\rho(r))}{\begin{array}{c} m:N:O:L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \left\langle \alpha, (\rho, \underline{a}.r\widetilde{V}) :: \mathcal{R} \right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}' \rangle\} \longrightarrow \\ m:N:O:L\{l \overset{-}{\underset{l':\mathcal{L}}{\longmapsto}} \langle \alpha, \mathcal{R} \rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, \mathcal{R}' :: (\rho, r\widetilde{V}) \right\rangle\} \end{array}}$$

$$\text{M.A}_1\text{.EC.L}\quad \frac{\rho(a)=\alpha \quad k \Subset O(\rho(r)) \quad ether\_child(k,l) \quad siblings(l,l')}{\begin{array}{c} m:N:O:L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R} \rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, (\rho, \underline{a}.r\widetilde{V}) :: \mathcal{R}' \right\rangle\} \longrightarrow \\ m:N:O:L\{l \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, \mathbf{i}\,(\underline{a}, r, \widetilde{V})) \right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}' \rangle\} \end{array}}$$

$$\text{M.A}_1\text{.EC.R}\quad \frac{O_0(\rho(a)) = \langle l_\top, t\rangle \quad k \Subset O(\rho(r)) \quad ether\_child(k, l_\top) \quad O' = merge(O, O_0)}{\begin{array}{c} m:N:O:L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R} \rangle\} \parallel E\{m' \to m \mapsto msg(\underline{a}.r, \widetilde{V}, \rho, O_0) :: \mathcal{M}\} \longrightarrow \\ m:N:O':L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, \mathbf{i}\,(\underline{a}, r, \widetilde{V})) \right\rangle\} \parallel E\{m' \to m \mapsto \mathcal{M}\} \end{array}}$$

FIG. *5.12* – **CLAM : Règles de routage pour les messages explicitement adressés (1er groupe).**

- Chaque localité passivée acquiert une nouvelle localisation: la machine d'arrivée.
- Chaque ressource passivée est associée à une localité spéciale notée $\daleth$ qui représente une localité en cours de migration. Cette localité est associée avec la machine d'arrivée.

Une optimisation directe est possible, puisque toutes les ressources provenant d'une sous-localité stricte de la localité passivée conservent leur localisation après activation du sous-arbre migrant.

Un message à destination d'une définition comprise dans une localité passivée qui parviendrait sur la machine de départ après l'opération de passivation doit être informé que cette définition existe toujours.

M.A2.IN.L
$$\frac{O(\rho(b)) = \langle k,t\rangle \quad desc(k,l) \quad siblings(l,l')}{\begin{array}{c} m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, (\rho, \overline{\underline{b}}.r\widetilde{V}) :: \mathcal{R}'\right\rangle\} \longrightarrow \\ m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, \mathbf{i}\ (\overline{\underline{b}}, r, \widetilde{V}))\right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\}\end{array}}$$

M.A2.IN.R
$$\frac{O_0(\rho(b)) = \langle k,t\rangle \quad desc(k,l_\top) \quad O' = merge(O,O_0)}{\begin{array}{c} m : N : O : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\} \parallel E\{m' \to m \mapsto msg(\overline{\underline{b}}.r, \widetilde{V}, \rho, O_0) :: \mathcal{M}\} \longrightarrow \\ m : N : O' : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, \mathbf{i}\ (\overline{\underline{b}}, r, \widetilde{V}))\right\rangle\} \parallel E\{m' \to m \mapsto \mathcal{M}\}\end{array}}$$

M.A2.MC
$$\frac{O(\rho(b)) = \langle k,t\rangle \quad desc(k,l)}{\begin{array}{c} m : N : O : L\{l \underset{l':\mathcal{L}}{\overset{-}{\longmapsto}} \left\langle \alpha, (\rho, \overline{\underline{b}}.r\widetilde{V}) :: \mathcal{R}\right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\} \longrightarrow \\ m : N : O : L\{l \underset{l':\mathcal{L}}{\overset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, \mathcal{R}' :: (\rho, \overline{\underline{b}}.r\widetilde{V})\right\rangle\}\end{array}}$$

M.A2.ME.L
$$\frac{O(\rho(b)) = \langle k,t\rangle \quad k \neq l \quad \neg\, desc(k,l) \quad siblings(l,l')}{\begin{array}{c} m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, (\rho, \overline{\underline{b}}.r\widetilde{V}) :: \mathcal{R}\right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\} \longrightarrow \\ m : N : O : L\{l \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, \mathcal{R}' :: (\rho, \overline{\underline{b}}.r\widetilde{V})\right\rangle\}\end{array}}$$

M.A2.ME.R
$$\frac{\begin{array}{c} O(\rho(b)) = \langle l'',t''\rangle \quad O(l'') = \langle m',t'\rangle \quad m' \neq m \\ O' = O\{\daleth \mapsto \langle m',0\rangle\} \quad \daleth = \langle m, \daleth_{loc}\rangle \quad \daleth_{loc} \notin N \\ O'' = O'\{\forall l \in loc\_id(\widetilde{V}) \text{ où } O'(l) = \langle m, t_l\rangle\ , \\ (\forall r \in rsc\_id(\widetilde{V}) \text{ où } O'(\rho(r)) = \{\langle l, t_{lr}\rangle\} \uplus \Theta_{lr}\ ,\ \rho(r) \mapsto \{\langle \daleth, t_{lr}+1\rangle\} \uplus \Theta_{lr}), \\ l \mapsto \langle m', t_l + 1\rangle\}\end{array}}{\begin{array}{c} m : N : O : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \left\langle \alpha, (\rho, \overline{\underline{b}}.r\widetilde{V}) :: \mathcal{R}\right\rangle\} \parallel E\{m \to m' \mapsto \mathcal{M}\} \longrightarrow \\ m : N \oplus \{\daleth_{loc}\} : O'' : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\} \parallel E\{m \to m' \mapsto \mathcal{M} :: msg(\overline{\underline{b}}.r, \widetilde{V}, \rho, O'')\}\end{array}}$$

M.A2.OUT
$$\frac{O(\rho(b)) = \langle k,t\rangle \quad k \neq l \quad \neg\, desc(k,l)}{\begin{array}{c} m : N : O : L\{l \underset{l':\mathcal{L}}{\overset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \left\langle \sharp, (\rho, \overline{\underline{b}}.r\widetilde{V}) :: \mathcal{R}'\right\rangle\} \longrightarrow \\ m : N : O : L\{l \underset{l':\mathcal{L}}{\overset{-}{\longmapsto}} \left\langle \alpha, \mathcal{R} :: (\rho, \mathbf{o}\ (\overline{\underline{b}}, r, \widetilde{V}))\right\rangle, l' \overset{-}{\underset{-}{\longmapsto}} \langle \sharp, \mathcal{R}'\rangle\}\end{array}}$$

M.FORWARD
$$\frac{O_0(\rho(b)) = \langle k,t\rangle \quad O(k) = \langle m',t'\rangle \quad m \neq m' \quad O' = merge(O,O_0)}{\begin{array}{c} m : N : O : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\} \parallel E\{m_0 \to m \mapsto msg(\overline{\underline{b}}.r, \widetilde{V}, \rho, O_0) :: \mathcal{M}_0,\ m \to m' \mapsto \mathcal{M}\} \longrightarrow \\ m : N : O' : L\{l_\top \overset{-}{\underset{-}{\longmapsto}} \langle \alpha, \mathcal{R}\rangle\} \parallel E\{m_0 \to m \mapsto \mathcal{M}_0,\ m \to m' \mapsto \mathcal{M} :: msg(\overline{\underline{b}}.r, \widetilde{V}, \rho, O')\}\end{array}}$$

FIG. *5.13* – **CLAM : Règles de routage pour les messages explicitement adressés (2ème groupe).**

Les informations de routage relatives à cette définition, périmées, mais toujours existantes, remplissent cette fonction. Le message ne pourra repartir vers sa définition cible qu'une fois la localité passivée la contenant est bien arrivée à destination (au moins dans la machine cible de la migration), et que la machine de départ dispose de nouvelles informations de routage concernant cette définition.

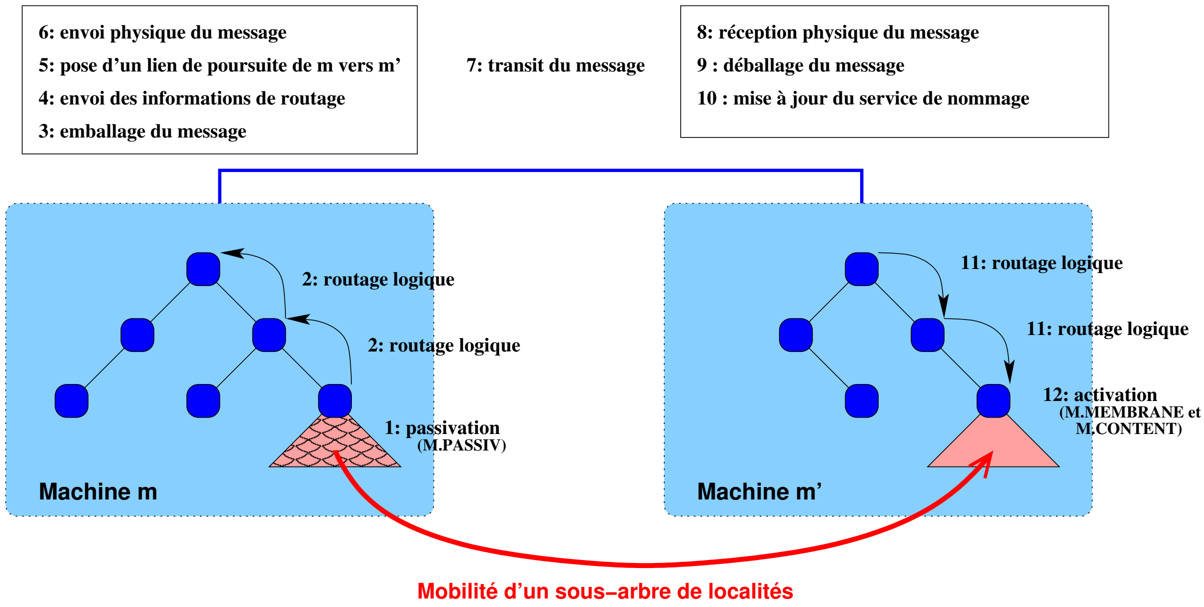


FIG. *5.14* – **Etapes d'une migration d'un sous-arbre de localités dans la CLAM.**

M.PASSIV
$$\frac{\begin{array}{c}\rho(a)=\alpha \quad l\in\mathcal{L}_0 \quad \mathcal{L}'_0=\mathcal{L}_0\backslash\{l\}\\ \langle \mathsf{reify}_{\mathsf{membrane}}(\mathcal{D},\mathcal{H},\mathcal{R}),\mathsf{reify}_{\mathsf{content}}(l_e,\mathcal{L}),L'\rangle = extract(l,L)\end{array}}{\begin{array}{c}m:N:O:L\{l \xmapsto[l_e:\mathcal{L}]{\mathcal{D}:\mathcal{H}} \langle\alpha,(\rho,\mathsf{pass}_a\ V)::\mathcal{R}\rangle, l_0 \xmapsto[l':\mathcal{L}_0]{-} \langle\alpha_0,\mathcal{R}_0\rangle, l' \xmapsto[-]{-} \langle\sharp,\mathcal{R}'\rangle\} \longrightarrow\\ m:N:O:L'\{l_0 \xmapsto[l':\mathcal{L}'_0]{-} \langle\alpha_0,\mathcal{R}_0\rangle, l' \xmapsto[-]{-} \langle\sharp,\mathcal{R}'::(\rho,V\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D},\mathcal{H},\mathcal{R})\ \mathsf{reify}_{\mathsf{content}}(l_e,\mathcal{L}))\rangle\}\end{array}}$$

M.MEMBRANE
$$\frac{\begin{array}{c}\langle\mathcal{D}',\mathcal{H}',\mathcal{R}'\rangle = insert_{flat}(\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}_m,\mathcal{H}_m,\mathcal{R}_m),\mathcal{D},\mathcal{H},\mathcal{R})\\ O' = O\{\forall D\in\mathcal{D}_m \ \ \forall r\in dn(D) \text{ tel que } O(\rho(r))=\{\langle\daleth,t_r\rangle\}\uplus\Theta, \rho(r)\mapsto\{\langle l,t_r\rangle\}\uplus\Theta\}\end{array}}{m:N:O:L\{l \xmapsto[-]{\mathcal{D}:\mathcal{H}} \langle\alpha,(\rho,\mathsf{reify}_{\mathsf{membrane}}(\mathcal{D}_m,\mathcal{H}_m,\mathcal{R}_m)\ ())::\mathcal{R}\rangle\} \longrightarrow m:N:O':L\{l \xmapsto[-]{\mathcal{D}':\mathcal{H}'} \langle\alpha,\mathcal{R}'\rangle\}}$$

M.CONTENT
$$\frac{\begin{array}{c}\langle\mathcal{D}',\mathcal{H}',\mathcal{R}',\mathcal{L}'_0,L'\rangle = insert_{sublocs}(\mathsf{reify}_{\mathsf{content}}(l_e,\mathcal{L}),\mathcal{D},\mathcal{H},\mathcal{R},\mathcal{L}_0,L)\\ O' = O\{\forall D\in l_e.\mathcal{D} \ \ \forall r\in dn(D) \text{ tel que } O(\rho(r))=\{\langle\daleth,t_r\rangle\}\uplus\Theta, \rho(r)\mapsto\{\langle l,t_r\rangle\}\uplus\Theta\}\\ O'' = O'\{\forall l_s\in locrec(\mathcal{L}) \ \ \forall D\in l_s.\mathcal{D} \ \ \forall r\in dn(D) \text{ tel que } O(\rho(r))=\{\langle\daleth,t_r\rangle\}\uplus\Theta, \rho(r)\mapsto\{\langle l_s,t_r\rangle\}\uplus\Theta\}\end{array}}{\begin{array}{c}m:N:O:L\{l_0 \xmapsto[l:\mathcal{L}_0]{-} \langle\alpha_0,\mathcal{R}_0\rangle, l \xmapsto[-]{\mathcal{D}:\mathcal{H}} \langle\sharp,(\rho,\mathsf{reify}_{\mathsf{content}}(l_e,\mathcal{L})\ ())::\mathcal{R}\rangle\} \longrightarrow\\ m:N:O'':L'\{l_0 \xmapsto[l:\mathcal{L}'_0]{-} \langle\alpha_0,\mathcal{R}_0\rangle, l \xmapsto[-]{\mathcal{D}':\mathcal{H}'} \langle\sharp,\mathcal{R}'\rangle\}\end{array}}$$

FIG. *5.15* – **CLAM : Mobilité des localités.**

5.3.5 Propriétés

La machine abstraite satisfait quelques propriétés qui sont conservées au cours de l'exécution, étant préservées par réduction.

Invariant 5.1. *Sur chaque machine, les localités forment une arborescence.*

Invariant 5.2. *Si une machine m héberge une localité l, alors le nom de cette localité est enregistré dans le service de nommage de m:*

$$\forall (m, l) \in (\mathsf{MName} \times \mathsf{LocNm}) \;\; l \in \mathsf{dom}\, m.L \Longrightarrow \exists\, t \in \mathsf{TimeStamp} \;\; \{l \mapsto \langle m, t\rangle\} \subset m.O$$

Invariant 5.3.
1. *Lorsqu'un message dont la destination est une ressource explicitement adressée $a.r$ est reçu sur une machine m: ou bien, il existe sur m une localité appelée a; ou bien, il existe un lien de poursuite vers une autre machine, pour que le message puisse être acheminé correctement.*
2. *Lorsqu'une cellule est envoyée sous forme passivée dans un message d'une machine m vers une machine m', un lien de poursuite de m vers m' est créé sur m.*

Le troisième invariant garantit le déterminisme du routage entre les machines (un message peut seulement être routé vers une machine unique à un instant donné) et entre les localités d'une même machine (sur une machine donnée, un message peut seulement être routé vers une localité unique à un instant donné).

Conjecture 5.2 (Cohérence). *Le service de nommage réparti $(O_m)_{m \in \mathsf{MName}}$ en utilisant R* comme protocole de mise à jour est cohérent.*

Enfin, nous estimons que la propriété suivante est préservée par réduction, ce qui fournit un premier point de départ pour prouver la correction de la CLAM par rapport au M–calcul:

Conjecture 5.3 (Correction). $\forall m \in \mathsf{MName}\; \forall (l, r) \in (\mathsf{LocNm} \times \mathsf{Ref})$ *tel que* $l \in \mathsf{dom}\, m.L \;\wedge\; l.\mathcal{R} = \ldots :: (\rho, P) :: \ldots \;\wedge\; r \in \mathsf{dom}\, \rho$:

$$l \Subset m.O(\rho(r)) \Longrightarrow r \in \mathsf{dln}(P)$$

La propriété de correction proprement dite semble trop difficile à prouver dans le cadre de cette thèse. Elle est laissée ouverte pour des travaux futurs.

5.4 Conclusion

Dans ce chapitre, nous avons présenté une spécification formelle d'une implantation du M–calcul sous forme de machine abstraite répartie. Cependant, la validation de la méthodologie présentée dans le chapitre 2, ne peut être complète sans la réalisation d'un prototype qui permette de conclure à l'effectivité d'un modèle de programmation à base de domaines. C'est ce que nous allons découvrir dans le chapitre suivant, en détaillant la mise en oeuvre de l'implantation de cette spécification dans un contexte centralisé et réparti.

Chapitre 6

OYSTERS : une machine virtuelle répartie pour le M–calcul

Ce chapitre présente une implantation du M–calcul sous forme d'une machine virtuelle, dont le modèle d'exécution a été spécifié dans le chapitre précédent par une machine abstraite. Cette plate-forme, appelée OYSTERS, est constituée d'un *compilateur* et d'une *machine virtuelle*. Le compilateur, écrit en OCAML, prend en entrée un programme à la syntaxe dérivée du M–calcul et produit un code intermédiaire, ensuite exécuté par la machine virtuelle, écrite en JAVA.

Après avoir présenté dans la section 6.1 les motivations pour la réalisation de cette plate-forme, la section 6.2 justifie l'approche adoptée mi-compilée, mi-interprétée et présente le compilateur et les composants de la machine virtuelle dans un cas centralisé, en insistant sur les stratégies possibles d'exécution du modèle. La section 6.3 étend la plate-forme au cas réparti en mettant l'accent sur la liaison et les composants. Enfin, nous généralisons dans la section 6.4 les différentes interfaces mises en évidence pour esquisser un canevas logiciel d'implantation de machines virtuelles pour calculs de processus répartis.

6.1 Buts poursuivis

6.1.1 Objectifs de l'implantation

Le principal objectif justifiant l'implantation de la plate-forme OYSTERS est de réaliser la troisième étape de la démarche de construction de systèmes fiables, formalisés par raffinements successifs à l'aide de l'abstraction de domaine. Les résultats décrits dans les deux chapitres précédents sont dans une large mesure d'ordre théorique. Il reste maintenant à disposer d'une implantation concrète du modèle de programmation à base de domaines. L'accent est mis sur la faisabilité de mise en oeuvre d'une telle implantation du calcul, en visant une complexité raisonnable, mais sans rechercher l'optimisation à tout prix. Cela signifie:

- *Démontrer le caractère effectivement implémentable du M–calcul:* des programmes écrits dans ce cœur de langage doivent pouvoir s'exécuter en consommant des ressources modérées (empreinte mémoire, temps d'exécution) sur des machines usuelles.

- *Démontrer l'intérêt et la faisabilité de la démarche de formalisation par raffinements:* cette démarche a déjà été illustrée dans le développement du run-time réparti DCP détaillé au chapitre 3. La plate-forme OYSTERS constitue la dernière étape de ce cheminement : une machine virtuelle qui implémente la spécification formelle (machine abstraite) d'un modèle de programmation (calcul de processus). Un point important dans cette dernière étape est de montrer que la spécification formelle sous forme de machine abstraite peut effectivement donner lieu à une implantation utilisant les technologies à objets répartis et plates-formes middleware usuelles.

6.1.2 Propriétés attendues

La machine virtuelle obtenue doit être:

- *Portable:* il s'agit de pouvoir exécuter le calcul sur différentes architectures, sans dépendre d'un système d'exploitation ou d'une infrastructure de communication.
- *Extensible:* il s'agit de pouvoir tester différentes stratégies d'exécution dans une implantation du M–calcul, afin de déterminer les compromis à réaliser. Un prolongement naturel de ces travaux serait d'élaborer un langage de programmation de haut-niveau, orienté domaines et fondé sur le M–calcul – par exemple par hybridation du calcul avec un langage de programmation plus traditionnel comme ML ou JAVA. Disposer d'une implantation extensible permettra aussi:
  1. d'obtenir facilement une plate-forme d'exécution pour ce langage par enrichissement du compilateur: ajout de code pour des constructions langage dérivées par « sucre syntaxique » ou pour des systèmes de types.
  2. de disposer d'une plate-forme d'expérimentation, pour tester à faible coût différentes extensions du modèle de programmation : primitives cryptographiques, caractéristiques non-fonctionnelles (nommage, mobilité, sécurité, transactions), etc.

6.2 Implantation centralisée

Dans cette section, nous décrivons une implantation du M_{loc}–calcul spécifiée par la machine abstraite centralisée définie dans la section 5.2.

6.2.1 Choix d'architecture

Approches existantes et limites

Dans [104], Xavier Leroy passe en revue les principales techniques d'implantation d'un langage $\mathcal{L}$ donné, en soulignant leurs avantages et inconvénients. On peut distinguer trois grandes approches :

1. Compilation de $\mathcal{L}$ par génération de code natif avec comme cible une architecture matérielle précise.
2. Traduction de $\mathcal{L}$ vers un autre langage $\mathcal{L}'$ et utilisation d'un compilateur efficace pour $\mathcal{L}'$.
3. Interprétation de code pour une machine abstraite.

L'approche (1) est ardue à cause du choix des instructions machine pour exécuter les primitives du langage et de celui d'une stratégie efficace d'allocation des registres du processeur. Ce travail difficile est à refaire lorsque l'on change d'architecture cible. De plus, la correction du code produit est souvent compromise par des optimisations peu formalisables. Cette approche semble donc peu compatible avec les exigences de simplicité et de portabilité de la plate-forme OYSTERS.

L'approche (2) est la méthode retenue par PICT qui utilise C comme langage cible. L'ensemble des optimisations « machine » sont réalisées par le compilateur C. Même si ce choix de langage reste relativement portable, l'implémentation d'un gestionnaire mémoire efficace peut constituer un obstacle en terme de simplicité de la plate-forme, surtout en vue d'obtenir rapidement un premier prototype. Cette technique semble peu compatible avec une approche interactive pour l'évaluation d'expressions du M–calcul, afin de faciliter la mise au point de la plate-forme en inspectant directement ses structures d'exécution.

Approche retenue

La dernière approche a été retenue pour l'implémentation d'OYSTERS. En définissant un format d'exécution commun à plusieurs architectures, elle offre l'avantage de la portabilité. Cette exigence est importante pour un calcul de processus réparti mobile qui doit exécuter et échanger du code sur des réseaux hétérogènes. L'inconvénient principal de cette approche – dans le cas d'un modèle d'exécution purement interprété – est celui d'une dégradation des performances. Ce handicap peut être compensé à l'aide d'une solution hybride: compilation du langage $\mathcal{L}$ vers du code intermédiaire indépendant d'une architecture, et émulation d'une machine pour ce code; son avantage est d'autant plus évident que le code intermédiaire est de bas-niveau. L'utilisation d'un compilateur permet de soulager les surcoûts à l'exécution en faisant disparaître de nombreuses opérations par des techniques d'analyse statique ou de typage. Cette solution hybride semble la plus adaptée en terme de simplicité, sans exclure pour autant les performances: une version efficace de la plate-forme d'exécution pourra être développée par la suite en rapprochant du système d'exploitation le niveau d'abstraction du code intermédiaire.

Une fois l'architecture globale établie (compilateur vers du code intermédiaire et interprète) se pose le problème du choix du langage d'implantation de ces deux composants du système. Bien que plusieurs langages soient adaptés (C/C++, JAVA, OCAML), le choix s'est porté sur OCAML pour l'écriture du compilateur, et sur JAVA pour l'écriture de la machine virtuelle :

- Pour le compilateur, le choix d'OCAML [105] permet la réutilisation de techniques d'implantation présentes dans PICT, JCL, ou JOCAML, tous écrits dans des langages dérivés de ML. Ce point est important pour permettre un prototypage rapide de la plate-forme. D'autre part, ce langage est performant et simple d'emploi pour le traitement de structures arborescentes typiquement manipulées par un compilateur.

- Pour la machine virtuelle, JAVA a semblé plus adapté, de nombreux outils JAVA étant disponibles pour gérer la distribution (JAVA RMI, ORB comme JONATHAN, etc.) ou la mobilité de code. D'autre part, ce choix a permis d'adapter du code provenant de l'expérimentation DCP (cf. chapitre 3) réalisée lors de l'étude d'une première version de machine virtuelle à base de domaines.

6.2.2 Le compilateur

Structure d'ensemble

Le compilateur OYSTERS comprend trois étages principaux (cf. figure 6.1), un par phase de compilation: un *analyseur syntaxique*, un *module d'analyse statique* et un *générateur de code*. D'autres étages pourront être ajoutés par la suite, comme un optimiseur, un vérificateur de types ou un système d'inférence de types.

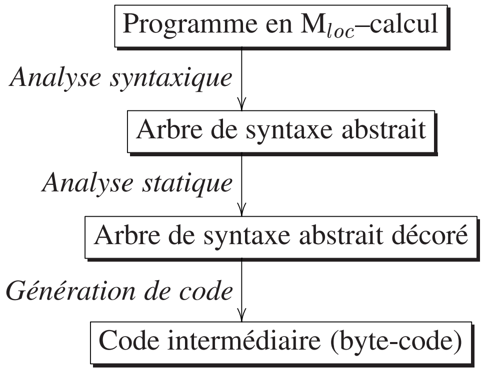


FIG. *6.1* – **Structure du compilateur OYSTERS.**

L'analyseur syntaxique

La syntaxe concrète du langage d'entrée, dérivée du M_{loc}–calcul, est donnée dans la figure 6.2, où Id représente un identificateur quelconque. L'analyseur syntaxique transforme ce langage d'entrée en un arbre de syntaxe abstrait en utilisant les outils `ocamllex` et `ocamlyacc` fournis avec la distribution d'OBJECTIVE CAML. L'arbre de syntaxe abstrait généré a la structure donnée dans la figure 6.3.

Le module d'analyse statique

Une fois cet arbre construit, la phase suivante de compilation le décore avec des informations concernant l'ensemble des noms libres et liés dans un processus ou une expression.

Le langage d'entrée contient deux lieurs `lambda` et `new`. Afin d'éviter des conflits de noms, et afin de pouvoir gérer correctement des portées syntaxiques imbriquées et les associations entre lieurs et identificateurs liés, nous utilisons des indices de De Bruijn – voir, par exemple [13]. Chaque identificateur possède un indice de De Bruijn généralisé, positif (resp. négatif) s'il est lié par une λ-abstraction (resp. par une restriction). La valeur absolue de l'indice donne le nombre total de lieurs traversés entre la racine de l'expression et l'occurrence de l'identificateur. Son signe permet de retrouver le type de lieur associé à cet identificateur. Si l'indice est positif, il s'agit d'une abstraction. Sinon, le lieur est une restriction.

$Proc$	::=	`nil`	Processus inerte
	|	Val	Valeur
	|	Var	Variable
	|	`dom` Id `do` $Proc$ `controls` $Proc$ `end`	Domaine/cellule
	|	`(` $Proc \mid \ldots \mid Proc$ `)`	Composition parallèle
	|	`(` $Proc \ldots Proc$ `)`	Application
	|	`(` `new` $Ref\ Proc$ `)`	Déclaration
	|	`if` $VarOrRef$ `=` $VarOrRef$ `then` $Proc$ `else` $Proc$	Branchement conditionnel
	|	Def	Définition
	|	`pass` Val	Passivation
	|	`(` $Proc$ `)`	Groupement de processus
Val	::=	Ref	Référence
	|	`lambda` Var `.` $Proc$	Abstraction
	|	`()`	Valeur nulle
Def	::=	`def` $DefBinding$ `and ... and` $DefBinding$	Définitions multiples
$DefBinding$	::=	$JoinPattern$ `=` $Proc$	Définition simple
$JoinPattern$	::=	$Ref \mid \ldots \mid Ref$	Filtre de messages
$VarOrRef$	::=	$Ref \mid Var$	Noms
Ref	::=	Id	Référence
Var	::=	Id	Variable

FIG. *6.2* – **Langage d'entrée du compilateur OYSTERS.**

Par exemple, le programme suivant

```
( new x ( lambda x . ( lambda x . ( x  x )  x )  x ) )
```

aura comme indices

```
( new x:-1 ( lambda x:2 . ( lambda x:3 . ( x:3  x:3 )  x:2 )  x:-1 ) )
```

Structure du code intermédiaire

Le code intermédiaire généré par le compilateur est une représentation indépendante d'une architecture donnée. Ce code est structuré en *blocs*, unité de base qui correspond à un entier, soit 4 octets sur des plates-formes 32 bits. Chaque primitive du langage est représentée par une séquence d'instructions, une instruction étant formée d'une séquence de blocs. Dans ces séquences, le premier bloc indique le type d'instruction (opcode). Les blocs suivants contiennent les arguments de cette instruction (opérandes). En général, les opérandes sont des pointeurs de code : leur valeur indique le décalage entre la position courante dans le flot d'instructions et celle du code obtenu par compilation des arguments de la primitive du calcul associée à l'opcode en cours de traitement. Les instructions sont de deux types:

```
type proc = {                              (* processus *)
  pproc_desc          : proc_desc ;   (* descripteur de processus *)
  pproc_info          : info ;        (* information de compilation *)
  mutable pproc_free  : IdSet.t ;     (* noms libres *)
  mutable pproc_bound : IdSet.t       (* noms liés *)
}

and proc_desc =                       (* descripteur de processus *)
  SKIP                                (* processus inerte *)
| VALUE of value                      (* valeur *)
| VAR of Id.id                        (* variable *)
| DOM of Id.id * proc * proc          (* domaine/cellule *)
| PRL of proc * proc                  (* composition parallèle*)
| APP of proc * proc                  (* application *)
| LET of decl * proc                  (* declaration *)
| IF of Id.id * Id.id * proc * proc   (* branchement conditionnel *)
| DEF of def                          (* définition *)
| PASS of value                       (* passivation *)
...
and value_desc =                      (* descripteur de valeur *)
  REFv of Id.id                       (* référence *)
| ABSv of pat * proc                  (* lambda-abstraction *)
| NILv                                (* value nulle*)
and pat_desc =                        (* descripteur de variable *)
  VARp of Id.id                       (* variable *)
...
and decl_desc =                       (* descripteur de déclaration *)
  NEW of Id.id                        (* création d'un nouveau nom *)
...
and def_desc =
  DEFS of (jp * proc) list            (* descripteur de définition *)
...
and jp_desc =
  JP of Id.id list                    (* descripteur de join-pattern *)

type toplevel = proc                  (* programme *)
```

FIG. *6.3* – **Arbre de syntaxe abstrait.**

- Les instructions à nombre fixe d'arguments. Il s'agit de la majorité des instructions comme `prl` ou `app`. Certaines instructions comme `domid`, `var` ou `ref` comportent un nombre variable de blocs car un de leurs arguments est une chaîne de caractères. Le premier argument indique alors la longueur de la chaîne.
- Les instructions à nombre variable d'arguments. Dans des instructions comme `def` (resp. `join`), le nombre de définitions (resp. de références dans un filtre de messages) peuvent être variables. Le premier argument indique alors le nombre d'arguments de l'instruction.

Dans chaque bloc, deux bits sont réservés pour distinguer une valeur pure d'une adresse, ou une instruction d'une valeur (cf. figure 6.4).

Contrairement à des implémentations comme ZINC [104], TYCO ou JOCAML qui utilisent des registres ou des fermetures de code, le code intermédiaire d'OYSTERS reste plus proche du niveau langage que du système – caractéristique également partagée par le byte-code de la machine virtuelle JAVA. Cela est

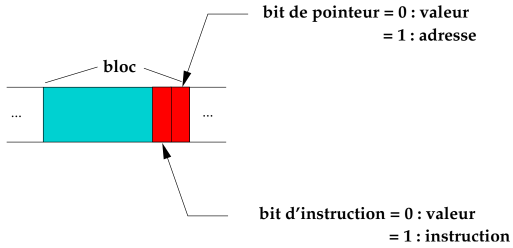


FIG. *6.4* – **Structure d'un bloc de code intermédiaire.**

cohérent avec la spécification formelle de la machine abstraite CLAM qui reste d'un niveau d'abstraction relativement élevé par rapport au système.

La décision d'implanter pratiquement « telle quelle » la machine abstraite CLAM est motivée par une exigence de simplicité. Une plate-forme complètement optimisée est spécifiable par raffinement de la machine abstraite présentée dans le chapitre 5 en faisant apparaître fermetures de code, registres, et autres structures mémoire. Son code intermédiaire serait de beaucoup plus bas niveau (proche de l'assembleur), permettant l'exécution d'une instruction par un petit nombre d'opérations dans la machine virtuelle, comme dans un processeur réel. Dans une telle implantation, l'essentiel du travail est effectué par le compilateur, la machine virtuelle étant relativement simple.

Dans la mise en oeuvre présentée ici, le compilateur n'a qu'un rôle relativement restreint: il sert simplement à fournir des crochets pour des modules d'analyse statique ou d'implémentation d'un langage complet fondé sur le M–calcul. L'essentiel des traitements est effectué par la machine virtuelle.

Le développement d'une infrastructure optimisée nécessitant largement plus de temps et de ressources que disponibles, il nous a semblé préférable de nous restreindre à une plate-forme plus simple qui permette de montrer l'implémentabilité du calcul et le caractère applicable de la démarche d'ensemble de formalisation par raffinement.

Cependant, le choix de manifester dans la structure du byte-code l'organisation imbriquée des termes du calcul – en particulier la structure hiérarchique des domaines – ouvre aussi la voie à des optimisations au niveau de la machine virtuelle. En particulier, cela permet à la machine virtuelle de manipuler directement des séquences d'instructions de byte-code obtenues par compilation d'un processus du calcul plutôt que les structures d'exécution internes correspondantes dans la machine, opération bien plus légère en terme de coûts d'exécution.

Jeu d'instructions

Le code intermédiaire est formé des instructions suivantes:

Instruction `app`
Utilisation $\texttt{app addr}_1\texttt{ , addr}_2$
Fonction Construit l'application $(P\ Q)$ du processus P au processus Q. Par rapport au bloc courant, le code du processus P est situé à l'adresse $\texttt{addr}_1$, et celui du processus Q est situé à l'adresse $\texttt{addr}_2$.
Format

`app`	@P	@Q

Instruction `def`
Utilisation $\texttt{def n, addr}_1\texttt{ , ... , addr}_n$
Fonction Construit une définition mutuelle $\langle D_1; \ldots ; D_n\rangle$ à partir des définitions individuelles $D_1, \ldots, D_n$. La première opérande correspond au nombre `n` de telles définitions individuelles. Par rapport au bloc courant, les codes des définitions individuelles sont situés aux adresses $\texttt{addr}_1, \ldots, \texttt{addr}_n$.
Format

`def`	size	@D_1	...	@$\mathrm{D}_{\mathrm{size}}$

Instruction `dom`
Utilisation $\texttt{dom addr}_1\texttt{, addr}_2\texttt{ , addr}_3$
Fonction Construit un domaine $a(P)[Q]$ nommé a, de contrôleur P, et de contenu Q. Par rapport au bloc courant, le nom du domaine est situé à l'adresse $\texttt{addr}_1$, le code du processus P est situé à l'adresse $\texttt{addr}_2$, et celui du processus Q est situé à l'adresse $\texttt{addr}_3$.
Format

`dom`	@a	@P	@Q

Instruction `domid`
Utilisation `domid sz , str`
Fonction Alloue de l'espace mémoire pour un nom de domaine a. Le nom interne du nom de domaine est stocké dans la chaîne de caractères `str` qui est de taille `sz` blocs.
Format

`domid`	size	string (size blocs)

Instruction `join`
Utilisation $\texttt{join n, addr}_1\texttt{ , ... , addr}_n\texttt{, addr}$
Fonction Construit une définition individuelle $\langle r_1 \mid \ldots \mid r_n \triangleright P\rangle$ à partir des références $r_1, \ldots, r_n$, et du corps P de la définition. La première opérande correspond au nombre `n` de références contenues dans le filtre de messages. Par rapport au bloc courant, les noms des références sont situés aux adresses $\texttt{addr}_1, \ldots, \texttt{addr}_n$, et le code du corps de la définition est situé à l'adresse `addr`.
Format

`join`	size	@r_1	...	@$\mathrm{r}_{\mathrm{size}}$	@P

Instruction `lambda`
Utilisation $\texttt{lambda addr}_1\texttt{ , addr}_2$
Fonction Construit une lambda-abstraction $\lambda\ x.P$, à partir d'une variable x et d'un processus P. Par rapport au bloc courant, le nom de la variable x est situé à l'adresse $\texttt{addr}_1$, et le code du processus P est situé à l'adresse $\texttt{addr}_2$.
Format

`lambda`	@x	@P

Instruction `new`
Utilisation $\texttt{new addr}_1\texttt{ , addr}_2$
Fonction Construit un processus de restriction $(\nu\ r)\ P$, à partir d'une référence r et d'un processus P. Par rapport au bloc courant, le nom de la référence r est situé à l'adresse $\texttt{addr}_1$, et le code du processus P est situé à l'adresse $\texttt{addr}_2$.
Format

`new`	@r	@P

Instruction `pass`
Utilisation `pass addr`
Fonction Construit le processus déclenchant une passivation **pass** V à partir de la valeur V. Par rapport au bloc courant, le code de la valeur V est situé à l'adresse `addr`.
Format

`pass`	@V

Instruction `prl`
Utilisation `prl` $\texttt{addr}_1$ `,` $\texttt{addr}_2$
Fonction Construit la composition parallèle $P \mid Q$ des processus P et Q. Par rapport au bloc courant, le code du processus P est situé à l'adresse $\texttt{addr}_1$, et celui du processus Q est situé à l'adresse $\texttt{addr}_2$.
Format

`prl`	@P	@Q

Instruction `ref`
Utilisation `ref sz , str`
Fonction Alloue de l'espace mémoire pour une référence r. Le nom interne de la référence est stocké dans la chaîne de caractères `str` qui est de taille `sz` blocs.
Format

`ref`	size	string (size blocs)

Instruction `skip`
Utilisation `skip`
Fonction Construit le processus inactif $\mathbf{0}$.
Format

`skip`

Instruction `test`
Utilisation `test` $\texttt{addr}_1$`,` $\texttt{addr}_2$ `,` $\texttt{addr}_3$ `,` $\texttt{addr}_4$
Fonction Construit un processus de test $([u \;=\; v]\; P \;,\; Q)$ à partir de deux noms u et v et de deux processus P et Q. Par rapport au bloc courant, les noms de u et v sont situés aux adresses $\texttt{addr}_1$ et $\texttt{addr}_2$, et le code des processus P et Q est situé aux adresses $\texttt{addr}_3$ et $\texttt{addr}_4$.
Format

`test`	@u	@v	@P	@Q

Instruction `var`
Utilisation `var sz , str`
Fonction Alloue de l'espace mémoire pour une variable x. Le nom interne de la variable est stocké dans la chaîne de caractères `str` qui est de taille `sz` blocs.
Format

`var`	size	string (size blocs)

Le générateur de code

Le code intermédiaire est représenté par la structure décrite dans la figure 6.5. Le back-end du compilateur est formé d'un générateur de code qui va prendre l'arbre de syntaxe abstrait décoré et le transformer en code intermédiaire. Cette phase de compilation calcule les différentes adresses comprises dans les blocs du byte-code.

```
type instr =
  ISKIP                                        (* processus inerte *)
| IREF of int * string                         (* référence *)
| ILAMBDA of int * int                         (* abstraction *)
| IDOM of int * int * int                      (* domaine *)
| IPRL of int * int                            (* composition parallèle *)
| IAPP of int * int                            (* application *)
| INEW of int * int                            (* restriction *)
| ITEST of int * int * int * int               (* branchement conditionnel *)
| IDEF of int * int list                       (* définition *)
| IPASS of int                                 (* passivation *)
| IJOIN of int * int list * int                (* filtre de messages *)
| IVAR of int * string                         (* variable *)
| IDOMID of int * string                       (* nom de domaine *)

type opcode =
  SKIP | REF | LAMBDA | DOM | PRL | APP | NEW
| TEST | DEF | PASS | JOIN | VAR | DOMID
```

FIG. *6.5* – **Code intermédiaire (byte-code).**

Un exemple de compilation

Ainsi, le programme suivant

```
(
  new r
    dom d1 do
      (r x)
    controls
      def r = lambda x . x
    end
)
```

sera compilé en :

```
Dumping byte-codes...
0000    new      [+2] , [+4]
0003    ref      1 , r
0006    dom      [+3] , [+6] , [+14]
0010    domid    2 , d1
0014    app      [+2] , [+4]
0017    ref      1 , r
0020    ref      1 , x
0023    def      1 , [+1]
0026    join     1 , [+2] , [+4]
0030    ref      1 , r
0033    lambda   [+2] , [+4]
0036    var      1 , x
0039    ref      1 , x
```

6.2.3 La machine virtuelle

Structure générale

L'implémentation de la machine virtuelle suit de près la spécification formelle CLAM. Sa structure, résumée dans la figure 6.6, comprend les composants suivants :

- Un *chargeur de code intermédiaire*, équivalent du chargeur de classes de la machine virtuelle JAVA, convertit le byte-code en structures de données utilisées lors de l'exécution.
- Des unités d'activité bien délimitées, appelées « *oysters* », s'exécutent de manière concurrente. Ils implémentent les localités/cellules du M–calcul et forment une arborescence à l'exécution. Chaque oyster comprend son propre évaluateur d'expressions et moteur d'exécution. Comme spécifié par les règles de la machine abstraite, un oyster évolue en fonction de son état propre et de celui de ses voisins dans la hiérarchie d'oysters.
- Un *ordonnanceur global* identifie le prochain oyster dont l'état doit évoluer.
- Un certain nombre de *services techniques* sont utilisés par l'ensemble des oysters : un générateur permet la création de nouveaux noms, uniques à l'échelle de la machine virtuelle. Un service de nommage localise les définitions dans l'arborescence des oysters. Enfin, un gestionnaire de topologie maintient dans une table unique l'ensemble des références d'oysters et permet la création de nouveaux oysters ou la suppression d'oysters existants.

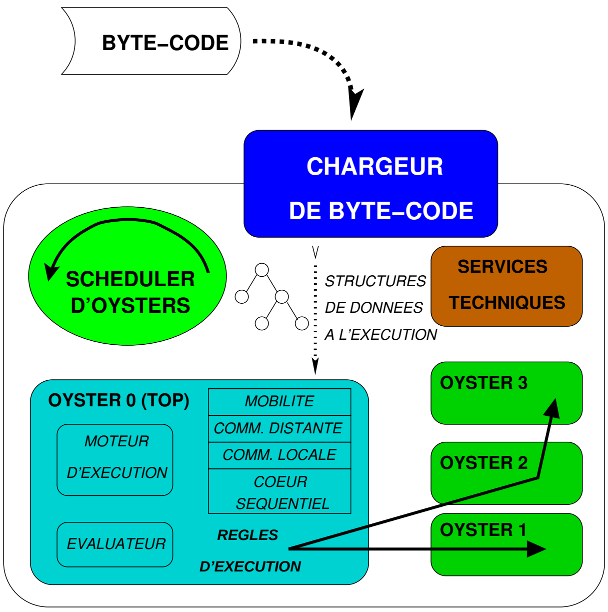


FIG. 6.6 – **Structure de la machine virtuelle OYSTERS.**

Le chargeur de code

Le rôle de ce composant est de convertir le code intermédiaire en une structure de données (« run-time data structure »), modifiée lors de l'exécution, dont la structure est proche de l'arbre de syntaxe abstrait d'un programme écrit en M–calcul. L'opération est similaire au chargement d'une classe à l'intérieur de la JVM (Java Virtual Machine): le byte-code est récupéré depuis le réseau, et traité par un analyseur syntaxique qui produit une arborescence d'objets constituant la représentation run-time du byte-code. Ces objets sont alors utilisés par le fil d'exécution principal, ou dans notre cas, par l'oyster de plus haut niveau. A l'exécution, un élément générique de code intermédiaire est représenté par l'interface `Klass`, qui contient uniquement des méthodes de conversion de types. L'interface `Klass` est implémentée par la classe abstraite `Process` dont dérivent toutes les classes représentant les termes du calcul. L'interface `Expression`, dérivée de `Klass` sert à représenter les λ–termes. Le point d'entrée du chargeur de code est la méthode `loadCode` qui renvoie un objet `Klass`, racine de la nouvelle arborescence d'objets, à partir d'un flot d'instructions de byte-code. La définition du chargeur de code est la suivante :

```
public class ByteCodeLoader {
  public static final int SKIP   = 0, ...  , DOMID  = 12;
  int [] bytecodes;

  public ByteCodeLoader() { ... }

  int [] loadCodeDataFromFile(String name) throws Exception { ... }

  public Klass defineKlass(int [] bc,int pc) {
    int instr = bc [pc];
    switch(instrValue(instr)) {
      case SKIP:  return new NILProc();
      ...
    }
    return null;
  }

  public Klass loadCode(String name) throws Exception {
    bytecodes = loadCodeDataFromFile(name);
    Klass k = defineKlass(bytecodes,0);
    return k;
  }
}
```

Implémentation d'un oyster

Le cœur de la machine virtuelle est formé par les oysters qui implantent fidèlement les localités de la machine abstraite CLAM. Conformément à la spécification formelle, chaque oyster – représenté par une classe `Oyster` – comprend deux sous-composants : un évaluateur et un moteur d'exécution. L'évaluateur obéit à l'interface suivante :

```
public interface Evaluator {
   public void load(Expression e);
   public void eval() throws EvaluationException;
   public Process unload();
}
```

Les méthodes `load` et `unload` implémentent respectivement les fonctions de traduction $[\![\cdot]\!]_{load}$, et $[\![\cdot]\!]_{unload}$ constituant l'interface décrite formellement au chapitre 5 entre moteur d'exécution et évaluateur d'expressions fonctionnelles. Cette interface permet la mise en oeuvre de diverses stratégies d'évaluation comme l'appel par valeur. A titre d'exemple, l'évaluateur avec appel par nom décrit dans le chapitre précédent implémente cette interface par :

```
public class CallByNameEvaluator implements Evaluator {
    ...
    EvalStack stack;   // Tas d'adresses
    EvalEnv zeta;      // Environnement global

    public Evaluator() { .. }

    public void load(Expression e) {
        stack.clear(); stack.push(e);
        zeta = new EvalEnv();
    }

    public void eval() throws EvaluationException {
        boolean active = true;
        while (active) {
            try {
                checkType(stack.peek());
                if (UNWINDRULE()) { active = true ; continue; }
                if (BETARULE())) { active = true; continue ; }
                if (INDIRRULE()) { active = true; continue ; }
                active = false;
            }
            catch(InvalidExpressionException e) { ... }
        }
    }

    public Process unload() {
       return stack.bottom().trimIndir().duplicate();
    }

    boolean UNWINDRULE() { ... }
    boolean BETARULE() { ... }
    boolean INDIRRULE() { ... }
    Klass instantiate(Klass expr, EvalEnv epsilon, EvalEnv zeta)  { ... }
}
```

La méthode `eval` est constituée d'une boucle qui va tenter successivement d'appliquer les règles de réduction UNWIND, BETA, et INDIR décrivant l'évolution de l'état de l'évaluateur. En cas d'échec, une exception est levée.

Le moteur d'exécution d'un oyster est chargé du déclenchement des règles qui gouvernent son évolution locale. Chaque oyster est identifié par un nom, de type `OysterName`, unique au niveau de la machine virtuelle. On utilise des noms internes « riches » pour représenter les références, ou des files pour les processus en attente de communication ou d'exécution. La structure du moteur d'exécution est la suivante:

- Les définitions potentiellement activables sont stockées dans un tas spécial, le *Defspace*, qui associe un couple (environnement, corps de définition) à un descripteur de définition. Ce descripteur comprend un indice de position dans le Defspace, un champ contenant le caractère activable de la définition, et un tableau de booléens. Ce dernier indique pour chaque référence du filtre de messages de la définition, si au moins un message est en attente de communication sur cette référence.

Pour implanter la synchronisation des messages, au moins deux solutions ont été envisagées [99]: (i) l'approche statique utilisée dans JCL où les définitions sont compilées en automates déterministes dont le code est activé en cas de déclenchement d'une communication locale; ou (ii) la solution dynamique mise en oeuvre dans JOCAML, où le caractère activable des définitions est testé et mis à jour à l'exécution lors de l'arrivée de chaque nouveau message. La solution retenue ici est celle de JOCAML qui paraît plus simple: l'ensemble des traitements sont réalisables à l'exécution sans nécessiter de phase de pré-compilation.

- Le *tas de messages* associe à chaque référence r une file de processus en attente de communication sur r. La référence r est implémentée par un nom interne qui contient, outre le nom r, l'indice de position dans le Defspace de la définition définissant r dans un de ses filtres de messages.
- La *file d'exécution* regroupe l'ensemble des processus qu'un oyster doit exécuter. L'évolution de l'état de l'oyster est principalement gouvernée par la forme du processus en tête de file.
- Un oyster O maintient également des références vers d'autres oysters qui représentent les sous-localités de la localité implémentée par O. Comme dans la spécification formelle, ces oysters sont de type membrane ou éther: un oyster de type éther est destiné à stocker les messages en attente de communication entre deux oysters.

Stratégies d'exécution

Chaque oyster est implémenté par un thread JAVA. L'évolution de son état est gouvernée par une stratégie d'exécution à deux niveaux qui détermine :

- *Au niveau global, le choix du prochain oyster dont l'état va évoluer.* L'ordonnanceur implanté est de type « round-robin »: une boucle d'exécution globale visite successivement chacun des oysters de la machine virtuelle en autorisant leur thread à poursuivre son exécution et en bloquant l'exécution des autres oysters. L'exécution se termine lorsqu'aucun oyster n'a pu être activé lors de la dernière itération de la boucle globale. L'interface de l'ordonnanceur permet de changer simplement cette stratégie d'exécution.

```
public interface Scheduler {
  public void schedulingLoop();
}
```

- *Au niveau d'un oyster, le choix de la prochaine règle de réduction à déclencher.* Chaque règle d'exécution peut être décomposée en pré- et post-traitements, selon l'interface suivante:

```
public interface RuleMethod {
  public String getRuleName();
  public boolean beforeTransition(Oyster oy, Vector params) throws Exception;
  public boolean afterTransition(Oyster oy, Vector params) throws Exception;
}
```

La première méthode renvoie le nom de la règle de réduction. Les méthodes suivantes testent l'applicabilité de la règle (pré-conditions) et modifient l'état de l'oyster (post-traitement). Par exemple, une implantation possible de la règle PRL est donnée par:

```
public class PRLRuleMethod extends BasicRuleMethod {

    public PRLRuleMethod() { super(); }

    public String getRuleName() { return "PRL"; }


    public boolean beforeTransition(Oyster oy, Vector params) throws Exception {
        RunQueue R = oy.getRunQueue();
        Environment rho = R.peek().getEnv();
        Process p = R.peek().getProc();
        if (!(p instanceof PRLProc)) return false;
        params.addElement(R);
        params.addElement(rho);
        params.addElement((PRLProc) p);
        return true;
    }

    public boolean afterTransition(Oyster oy, Vector params) throws Exception {
        RunQueue R = (RunQueue) params.elementAt(0);
        Environment rho = (Environment) params.elementAt(1);
        PRLProc p0 = (PRLProc) params.elementAt(2);
        R.dequeue();
        R.insertinFront(new EnvProc(rho, p0.getP1()));
        R.enqueue(new EnvProc(rho.duplicate(), p0.getP2()));
        return true;
    }
```

L'implantation des autres règles de la machine abstraite est similaire et ne pose pas de problème particulier en environnement centralisé: il s'agit d'une simple transposition en JAVA de la spécification formelle.

On enregistre chaque règle auprès de la machine virtuelle grâce à l'interface suivante:

```
public interface RuleManager {
  public RuleMethod getReductionRule(int index);
  public int getTotalNumberOfRules();
  public void addNewReductionRule(RuleMethod rule);
}
```

Le cœur du moteur d'exécution d'un oyster est implémenté par la méthode `run()` du thread associé. Cette méthode prend la forme d'une séquence de tests relatifs au caractère activable de chaque règle de réduction. Toute règle activable est immédiatement déclenchée, et l'exécution est à nouveau suspendue en rendant la main à l'ordonnanceur, comme le montre le code de la figure 6.7.

Dans l'implémentation actuelle, on commence par tester la possibilité d'effectuer une communication locale (règle COMM), règle indépendante de l'état de la file d'exécution. En cas d'échec, l'état de cette file est examiné. L'exécution de l'oyster est suspendue si cette file est vide. D'autres stratégies locales d'exécution peuvent être simplement implémentées en redéfinissant la méthode `run()`.

Les interfaces `RuleMethod` et `RuleManager` contribuent à rendre la machine virtuelle extensible. Le code d'un oyster reste générique et indépendant de l'implémentation des règles de réduction de la machine abstraite ou de leur nombre. On peut donc introduire de nouvelles fonctionnalités dans la machine virtuelle: il suffit d'y enregistrer – grâce à un gestionnaire de règles implantant l'interface `RuleManager` – des classes décrivant de nouvelles règles de réduction, et respectant l'interface `RuleMethod`.

```
package vm.locations.engine;

class OystersRuleManager implements RuleManager {

   static final int rule_nb = ... ; /* Nombre de règles CLAM implémentées */


   static String [] rule_names =    /* Nom des règles de réduction CLAM  */
     { "NIL", "PRL" , ... };

   Vector rule_methods = new Vector();  /* Objets de type RuleMethod */

   public RuleMethod getReductionRule(int index) {
     return (RuleMethod) rule_methods.elementAt(index);
   }

   public int getTotalNumberOfRules() { return rule_methods.size(); }

   public void addNewReductionRule(RuleMethod rule) {
     rule_methods.addElement(rule);
   }

   OystersRuleManager() {
     for (int i = 0; i<rule_nb; i++) {
       try {
         Class c = Class.forName("vm.locations.engine." +
                                 rule_names[i] + "RuleMethod");
         addNewReductionRule((RuleMethod) c.newInstance());
       } catch(Exception ex) {...}
     }
   }
}

public class Oyster extends Thread {
    ...
    OystersRuleManager OyRMgr;
    ...
    /* Boucle d'exécution */
    public void run() {
        ...
    startloop:
        // Bloquage du thread courant
        ...
        while (true) {
            // Examen des règles de réduction
            for (int i = 0; i < OyRMgr.getTotalNumberOfRules() ; i++) {

                RuleMethod rule = OyRMgr.getReductionRule(i);
                if (rule.beforeTransition(this)) {

                    // Transition !
                    transisition(rule.getRuleName());

                    // Mise à jour de l'état de la localité
                    rule.afterTransition(this);

                    // Débloquage des autres threads d'oysters
                    ...
                    continue startloop;
                }
            }
            // Pas de transition applicable
            continue startloop;
        }
    }
}
```

FIG. *6.7* – **Enregistrement et ordonnancement des règles de réduction dans un oyster.**

Services techniques d'exécution

Au fur et à mesure de la création des oysters, leurs noms sont enregistrés auprès du gestionnaire de topologie, ressource centralisée accessible depuis n'importe quel oyster. Ce service fournit des informations sur l'organisation de la hiérarchie des oysters. Il renvoie le type (membrane/éther) d'un oyster donné, le nom de ses descendants immédiats, et répond à des requêtes concernant les liens de parenté entre oysters. Il est aussi capable de créer de nouveaux oysters à un point fixé de l'arborescence, ou de supprimer définitivement un oyster existant. Il maintient une table globale à la machine associant noms d'oyster et implémentations. Son interface est la suivante:

```
public interface TopologyManager {
    public Oyster getOysterImplementation(OysterName oy_name);

    public boolean isEtherOyster(OysterName oy_name);

    public OysterName getParent(OysterName oy_name);
    public OysterName getEtherSubOyster(OysterName oy_name);
    public OysterNames getMembraneSubOysters(OysterName oy_name);
    public boolean isSiblingOf(OysterName oy_name1, OysterName oy_name2);
    public boolean isParentOf(OysterName oy_name1, OysterName oy_name2);
    public boolean isChildOf(OysterName oy_name1, OysterName oy_name2);
    public boolean isAncestorOf(OysterName oy_name1, OysterName oy_name2);
    public boolean isDescendantOf(OysterName oy_name1, OysterName oy_name2);

    public Oyster newOyster(OysterName oy_name, boolean oy_type);
    public void addOyster(OysterName oy_name, Oyster oy, OysterName parent);
    public void removeOyster(OysterName oy_name);
}
```

Le service de création de noms, fournit de nouveaux identificateurs à la fois de ressources et d'oysters, uniques au niveau de la machine virtuelle. Son interface est la suivante:

```
public interface NameGenerator {
    public InternalName newInternalName();
    public OysterName newOysterName();
}
```

Enfin, le service de nommage permet de retrouver à tout moment de l'exécution, l'association entre un nom de ressource et le nom d'un oyster abritant la définition définissant cette ressource. Ce service obéit à l'interface suivante:

```
public interface LookupService {
    public void bind(String rsc_name, Oystername oy_name);
    public OysterName lookup(String rsc_name) throws RscNotFoundException;
}
```

L'initialisation de la machine virtuelle consiste à instancier l'ordonnanceur, à lancer les services techniques d'exécution (topologie, nommage, générateur de noms) et à créer un premier oyster – qui donnera naissance par la suite à une hiérarchie d'oysters. On transmet alors les structures de données instanciées lors du chargement du byte-code au moteur d'exécution de ce premier oyster. Le contrôle est ensuite transféré à l'ordonnanceur pour l'exécution du programme proprement dit.

Outils

Outre le compilateur et la machine virtuelle, la plate-forme OYSTERS comprend les outils suivants:

- Une *boucle interactive* (top-level à la Lisp) autorise l'évaluation en direct de termes du calcul. Le terme saisi par l'utilisateur est compilé à la volée et le code intermédiaire ainsi généré est directement interprété par la machine virtuelle. Sont affichés l'état interne final de la machine ou une trace d'exécution regroupant l'ensemble des règles de réduction déclenchées.
- Un *front-end graphique* permet de visualiser à tout moment l'état complet de chaque oyster contenu dans la machine virtuelle, de parcourir les différents états d'un oyster ayant déjà donné lieu à réduction, ou de guider l'exécution pas à pas: on indique explicitement à l'ordonnanceur de la machine virtuelle qu'un oyster donné doit être le prochain à changer d'état par déclenchement d'une règle de réduction. Cet outil permet ainsi d'explorer de manière interactive le non-déterminisme présent dans la machine abstraite CLAM.

6.3 Implantation répartie

6.3.1 Vers une machine virtuelle répartie

Quels sont les principaux points d'architecture à examiner pour obtenir à partir de la plate-forme OYSTERS précédente une infrastructure distribuée implantant complètement la machine abstraite répartie CLAM décrite dans la section 5.3? L'introduction dans la spécification formelle du concept de « machine » comme unité physique de répartition implique une description plus précise de l'environnement de la machine virtuelle OYSTERS.

Désormais, chaque machine virtuelle est identifiée par un nom symbolique, unique dans le système. Ces noms sont attribués par un *serveur de domaine* qui contient la liste de l'ensemble des machines s'étant enregistrées comme faisant partie du système. Il contient aussi l'adresse IP des machines physiques associées – une machine physique pouvant abriter plusieurs machines virtuelles. Comme dans KLAIM, on obtient alors un *réseau* OYSTERS. En première approximation, on peut considérer que le réseau n'est pas structuré: les noms de machines font partie d'un seul contexte de nommage[1].

Pour implanter les domaines du M–calcul en contexte réparti, l'architecture de la machine virtuelle doit évoluer dans deux directions principales : la liaison et les composants (cf. chapitre 2).

- *L'établissement de liaisons réparties entre machines virtuelles* est nécessaire pour implanter les règles de communication (physiquement) distante de la machine abstraite CLAM et permettre à des oysters situés sur des machines distinctes d'interagir.

[1]Cependant, pour que le système passe à l'échelle, une extension intéressante consiste à partitionner un réseau OYSTERS en sous-réseaux, chacun gérant un sous-ensemble de machines. Les sous-réseaux, de noms distincts, constituent chacun un contexte de nommage propre – on suppose toujours acquise l'unicité globale des noms de machine. Ils possèdent leur propre serveur de domaine, contenant la liste des machines virtuelles membres du sous-réseau. Chaque serveur de domaine est aussi une passerelle assurant le routage des messages d'un sous-réseau à l'autre (cf. figure 6.8). Ce partitionnement n'a pas véritablement d'impact sur la machine abstraite CLAM: un nom de machine est alors simplement formé par un couple (nom de sous-réseau, nom de machine dans le sous-réseau). En itérant le processus, il est possible de partitionner les machines physiques en une hiérarchie de domaines administratifs.

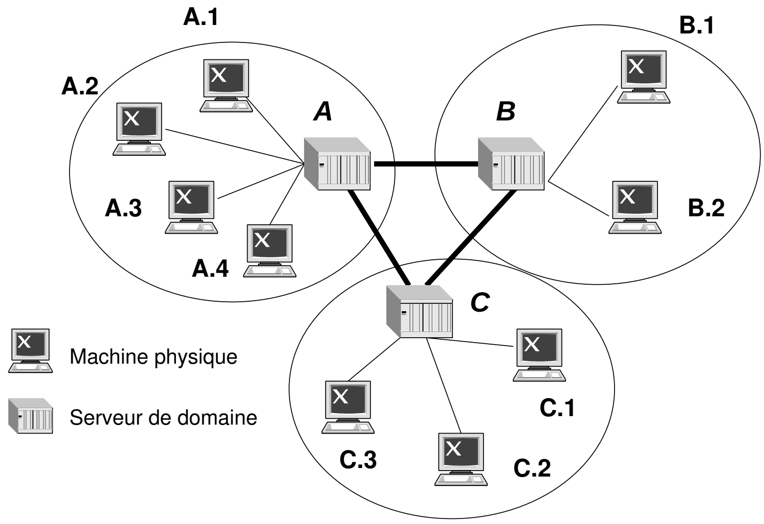


FIG. *6.8* – **Réseaux et sous-réseaux OYSTERS.**

- *L'identification claire des composants de la machine virtuelle* est indispensable pour disposer d'une infrastructure extensible et ouverte. L'utilisation d'un modèle de composants contribue à rendre l'architecture de la machine virtuelle plus uniforme, et facilite l'introduction de nouvelles fonctionnalités dans la plate-forme.

Ces points sont détaillés dans les deux sections suivantes. Nous ne donnons que les grandes lignes d'implémentation de la plate-forme répartie, certains des composants nécessaires à son fonctionnement n'ayant été que partiellement implantés à ce jour.

6.3.2 Mise en oeuvre de liaisons réparties

Dans cette section, nous décrivons l'impact de l'introduction de la répartition dans la plate-forme OYSTERS en terme de désignation et de communication.

Architecture générale

La machine virtuelle se compose désormais d'un noyau et d'une interface vers un middleware dont le rôle se limite à offrir des services de communication distante et de mobilité. Cette interface doit être suffisamment générique pour que la machine virtuelle puisse s'exécuter sur différentes infrastructures de communication. Nous esquissons l'établissement de liaisons réparties en utilisant comme exemple l'ORB JONATHAN, déjà présenté dans le chapitre 3. Cependant, le couplage avec d'autres infrastructures mériterait d'être étudié: ainsi, le bus logiciel DSS (Distribution Sub-System) [86] est un composant middleware qui permet à des machines virtuelles langage de communiquer de façon transparente, indépendamment d'un langage de programmation.

Dans la plate-forme répartie, le noyau contient toujours un certain nombre d'oysters, organisés en une hiérarchie, et constitués chacun d'un évaluateur et d'un moteur d'exécution.

- L'évaluateur doit être enrichi, l'ensemble des valeurs étant plus élaboré en M–calcul qu'en M_{loc}–calcul: noms de ressources, de localités, ou de ressources explicitement adressées. L'introduction de ces nouveaux types de noms nécessite l'ajout d'instructions dans le byte-code et la modification du compilateur et du chargeur de code de la machine virtuelle.
- Le moteur d'exécution implante un plus grand nombre de règles de réduction qu'en contexte centralisé. Les classes implantant les règles du cœur séquentiel ou concernant les définitions restent quasiment inchangées. De même, l'implémentation des règles de routage logique (intra-machine) des messages est obtenue par simple transposition de leur version centralisée. Sont plus complexes les règles relatives à la communication distante inter-machines (règles M.A$_1$.EM.R, M.A$_1$.EC.R, M.A$_2$.IN.R, M.A$_2$.ME.R, et M.FORWARD) et à la mobilité d'oysters (règles M.MEMBRANE et M.CONTENT).

Les services techniques sont désormais répartis sur chaque machine virtuelle:

- *Le générateur de noms:* dans la plate-forme répartie, chaque machine virtuelle constitue un domaine de désignation distinct. Les noms internes manipulés par les oysters contiennent désormais une information de localisation supplémentaire: le nom de la machine virtuelle qui les a créés.
- *Le gestionnaire de topologie:* les fonctions importantes restent la création et suppression d'oysters, et la fourniture d'informations sur la structure de la hiérarchie d'oysters hébergée par la machine virtuelle pour assurer le routage logique des messages.
- *Le service de nommage:* son interface est étendue, car ce service doit gérer la localisation à la fois des définitions (dans quel oyster?) et des oysters (dans quelle machine?). C'est le rôle des méthodes `lookup` dans l'interface suivante:

```
public interface LookupService {
    public OysterNameSet lookup (String rsc_name) throws RscNotFoundException;
    public void lookup(String oy_name,
                       Machinename m, TimeStamp t) throws OysterNotFoundException;
    public boolean isMember(String rsc_name, Oystername oy_name);
    public void bind(String rsc_name, Oystername oy_name, TimeStamp t);
    public void bind(String oy_name, Machinename m, TimeStamp t);
    public void add(String rsc_name, Oystername oy_name, TimeStamp t);
    public void rebind(String rsc_name, OysterNameSet oy_names);
    public void rebind(String oy_name, Machinename m, TimeStamp t);
}
```

Des estampilles qualifient la « fraîcheur » de l'information de routage. Le service de nommage doit en particulier implanter l'opérateur $\Subset$ défini dans la section 5.3.2 (méthode `isMember`), et permettre d'ajouter un nouveau couple $\langle l, t\rangle$ à l'ensemble $O(\rho(r))$ (méthode `add`), ou effectuer des mises à jour (méthodes `rebind`). Il doit aussi implanter la fonction $merge$ (voir section 5.3.4) pour la mise à jour des informations de routage lors de la réception d'un message par une machine virtuelle.

Liaison statique et dynamique

Contrairement au join–calcul où la liaison est purement statique, le M–calcul introduit une forme de liaison dynamique:

- En join–calcul, un nom de ressource est déclaré et défini à l'aide d'une même primitive `def` qui crée simultanément un nouvel identificateur de ressource (déclaration) et introduit l'implémentation de cette ressource (définition). La portée d'un identificateur est statiquement définie. Elle peut être étendue au delà de la localité (ou de la machine) courante pour devenir répartie. Cela se traduit dans l'implantation par transformation des références locales en références réseau.

  La liaison entre noms et ressources est statique. Un nom de ressource ne peut être défini qu'à un seul endroit du système (contrainte d'unicité des récepteurs) pour que le calcul reste implantable en contexte réparti. Autrement dit, le nom d'une ressource l'identifie sans ambiguité au niveau du système global. Aussi, le comportement exécuté à la réception d'un message ne dépend pas de l'endroit où se trouve ce dernier. Cette caractéristique est fondamentale pour assurer la transparence des communications à la répartition. Comme le montre la règle COMM du join-calcul réparti, si le nom de ressource désigne la cible d'un message situé dans une localité α, différente de celle définissant la ressource (β), on aura migration du message vers le lieu de définition de la ressource, quelque soit la localité de départ du message:

$$\frac{x \text{ est localement défini dans } D'}{\alpha[D : x\langle\tilde{v}\rangle \mid P] \parallel \beta[D' : P'] \parallel \mathcal{S} \longrightarrow \alpha[D : P] \parallel \beta[D' : x\langle\tilde{v}\rangle \mid P'] \parallel \mathcal{S}}$$

- Le M–calcul sépare déclaration et définition des ressources dans deux primitives distinctes. L'opérateur de restriction ν est utilisé pour créer de nouveaux identificateurs. La primitive de définition est conservée pour définir l'implantation d'une ressource, mais sans création d'un nouveau nom. La portée des identificateurs est toujours statique comme en join–calcul et peut devenir répartie à l'aide de deux règles structurelles d'extension de portée, les conditions auxiliaires visant à éviter la capture de noms libres:

$$\begin{array}{lcll} a(\nu n.P)[Q] & \equiv & \nu n.a(P)[Q] \quad n \notin \mathsf{fn}(Q) \ \wedge \ n \neq a & \text{S.NU.MEMB} \\ a(P)[\nu n.Q] & \equiv & \nu n.a(P)[Q] \quad n \notin \mathsf{fn}(P) \ \wedge \ n \neq a & \text{S.NU.CONT} \end{array}$$

  Cependant, le comportement exécuté à la réception du message dépend de l'endroit où se trouve ce dernier: le M–calcul autorise la définition de ressources portant le même nom mais situées dans des localités différentes. Le routage du message dépend alors fortement de la position par rapport à la localité courante d'une définition définissant le nom de ressource désignée comme cible du message.

Conséquences en terme d'implantation

- *Portée statique:* pour implémenter la portée potentiellement répartie des identificateurs et les règles d'extension de portée du calcul, les techniques mises en oeuvre dans l'expérimentation DCP peuvent être entièrement réutilisées. Elles doivent s'appliquer aux identificateurs du calcul ayant été créés explicitement à l'aide de l'opérateur de restriction ν.

  Aucune opération n'est nécessaire tant que la portée de ces noms n'est étendue que vers des localités situés sur une même machine virtuelle car cette dernière constitue un contexte de nommage pour ces identificateurs. Si l'extension dépasse la frontière d'une machine, l'opération `export` (cf. chapitre 3) rend les noms internes (de ressources ou de localités) visibles et potentiellement accessibles depuis l'ensemble du système réparti. Cette opération implémente les fonctions de transformation de noms entre références locales et références réparties. Comme dans DCP, on peut utiliser la politique d'exporter par défaut les noms créés dans une machine par l'intermédiaire du générateur de noms local.

L'implémentation de la portée statique répartie est réalisable à l'aide des interfaces suivantes:

```
public interface DistributedIdentifier {
    public NamingContext getContext();
    public Object bind();
    public Object resolve();
    public byte [] encode();
    public void encode(Marshaller m);
}

public interface NamingContext {
    public DistributedIdentifier export(Object obj);
    public DistributedIdentifier decode (byte [] data);
    public DistributedIdentifier decode(UnMarshaller u);
}

public class InternalName implements DistributedIdentifier {...}
public class OystersMachine implements NamingContext {...}
```

- *Liaison dynamique:* la stratégie de routage des messages présente dans la sémantique du calcul est complètement décrite par la machine abstraite. L'algorithme R*, présenté dans le chapitre 5, spécifie clairement le chemin que doit suivre un message: d'après la position de sa ressource cible (règles $\mathrm{M.A}_1$.*); ou d'après la position de sa localité cible (règles $\mathrm{M.A}_2$.*). Ces positions sont définies par rapport à la localité qui héberge actuellement le message. L'unicité des noms de localités dans le système assure un routage déterministe.

  Cet algorithme peut donc être implanté tel quel. Seules les règles impliquant plusieurs machines dans la spécification nécessitent l'établissement d'une liaison entre machines virtuelles. Là encore, les techniques mises en oeuvre dans l'expérimentation DCP sont réutilisables: l'oyster-racine de la machine de départ établit une liaison avec la machine d'arrivée par une invocation `oy_name.bind()`, où `oy_name` désigne le nom de l'oyster-racine sur la machine d'arrivée. Cette opération instancie une pile d'objets–session comme dans DCP, par exemple au-dessus de TCP/IP. Il suffit alors d'envoyer le message par invocation d'une méthode `send` (cf. chapitre 3).

  Reste à déterminer le choix du format de transmission des messages. Une première solution consiste à utiliser les emballeurs fournis par la plate-forme de communication sous-jacente pour envoyer sur le réseau une représentation plate des structures de données transmises (processus, état du service de localisation, ...). Cette approche a l'avantage d'être simplement implémentable, mais reste coûteuse car l'état de nombreux objets JAVA doit être transmis. On peut aussi enrichir le code intermédiaire OYSTERS pour que la représentation plate soit exprimée dans ce code. Plus satisfaisante, puisque le code intermédiaire est directement utilisé pour réaliser de la mobilité de code, cette solution nécessite toutefois plus de travail pour définir cette nouvelle représentation pivot et pour implémenter l'emballeur et le déballeur correspondant. Un compromis intéressant consiste à utiliser le byte-code OYSTERS pour transmettre le code des processus, et les emballeurs de la plate-forme de communication pour les autres structures de données. On peut ainsi conserver le byte-code existant, tout en allégeant sensiblement les communications.

6.3.3 Mise en oeuvre des composants

Faire évoluer l'architecture de la machine virtuelle OYSTERS en la structurant à l'aide de *composants* rend plus explicite son organisation modulaire. Le modèle de composants FRACTAL [33, 136] semble particulièrement indiqué pour réaliser cette évolution, car FRACTAL manifeste les domaines à l'exécution (cf. section 2.5.3). Un composant FRACTAL est formé de de deux parties, un *contrôleur* et un *contenu*, qui implantent respectivement les deux éléments d'un domaine, à savoir sa membrane programmable et son contenu. Le modèle est récursif et autorise l'assemblage de composants distincts pour obtenir des composants de niveau supérieur. Chaque composant est un composite comprenant différents sous-composants dans son contenu, dont l'exécution est supervisée par le contrôleur. Un oyster devient alors un composant FRACTAL manifeste à l'exécution, un envoi de message entre oysters étant une invocations de méthode sur une interface d'un composant FRACTAL.

Cette architecture rend plus explicite la relation de contrôle: on passe ainsi d'une hiérarchie logique d'oysters, où la relation de contrôle se manifeste par les références entre oysters, à une structure imbriquée où le contrôle s'exprime par la relation de contenance entre composants. Ces relations sont modifiables à l'exécution, pour traduire la dynamicité de la topologie des domaines.

Nous identifions ci-dessous certaines des principales interfaces d'un composant de la machine virtuelle OYSTERS (cf. figure 6.9).

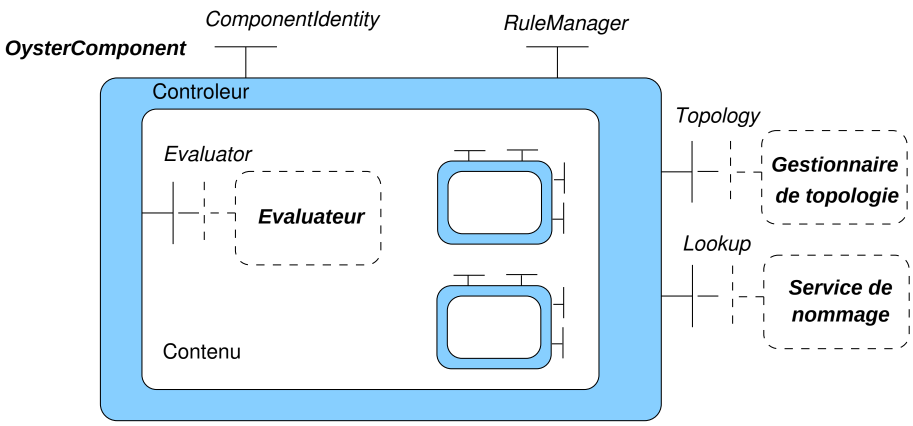


FIG. *6.9* – **Interfaces d'un `OysterComponent`.**

Un composant, de type `OysterComponent`, fournit une interface `ComponentIdentity` qui l'identifie de manière unique, tout comme l'interface `OysterName` définie dans la section 6.2.3. Chaque composant est en fait un composite comprenant deux sous-composants liés par l'interface `Evaluator` définie dans la section 6.2.3: le moteur d'exécution, représenté par le contrôleur du composant courant et un évaluateur fonctionnel. Un `OysterComponent`, dispose également d'une interface `RuleManager` grâce à laquelle on peut agir sur les différentes règles de réduction implantées par un oyster.

Les autres interfaces permettent de faire la liaison avec les services techniques de la machine virtuelle. Un composant dispose d'une interface `Topology` pour interagir avec le gestionnaire de topologie, afin de créer de nouveaux composants, de supprimer des composants existants, ou d'examiner la structure d'un composant particulier. Ces opérations sont implantables en réalisant des appels aux méthodes `addFcSubComponent`, `removeFcSubcomponent` et `getFcSubComponent` de l'interface FRACTAL `ContentController`. De même, des appels aux méthodes `bindFc`, et `unbindFc` de l'interface FRACTAL `BindingController` permettent de créer ou de supprimer des liaisons entre composants. Un `OysterComponent` possède aussi une interface `Lookup` pour interroger le service de nommage concernant la localisation des définitions ou d'autres composants.

Les autres éléments de la machine virtuelle, comme l'ordonnanceur ou le chargeur de byte-code, sont inchangés par rapport au cas centralisé.

6.4 IMC: un canevas de construction de machines virtuelles réparties

Les interfaces décrites précédemment répondent à des besoins d'établissement de liaisons réparties ou de structuration d'une machine virtuelle sous forme de composants. Ces besoins sont communs à la plupart des machines virtuelles réparties existantes qui implantent des calculs de processus manifestant une sémantique de domaine (cf. chapitre 2). Nous esquissons ici les bases d'un canevas logiciel appelé IMC (Implementing Mobile Calculi) pour la construction de nouvelles machines virtuelles de ce type. L'intérêt sera de fournir une boite à outils pour implémenter de nouveaux calculs à base de domaines. Ce travail, mené dans le cadre du projet MIKADO [123], est en cours. Il est décrit plus en détail dans [16].

L'objectif du canevas IMC est d'offrir un cadre générique pouvant être utilisé comme middleware pour l'implémentation de calculs de processus mobiles – en particulier les modèles de programmation déjà existants comme KLAIM, DITYCO, le join–calcul ou le M–calcul. Pour des raisons de portabilité, le canevas se présente sous forme d'un ensemble d'interfaces JAVA. Il est construit autour de la notion générale de *nœud d'exécution*, déjà identifiée dans la section 2.4.4 comme point commun à l'implantation de ces modèles. IMC comporte quatre paquetages principaux:

- *Topologie des nœuds d'exécution:* ce paquetage doit gérer l'organisation des différents nœuds d'exécution et permettre sa reconfiguration. Cette organisation reflète la topologie de communication du modèle de programmation.
- *Nommage et liaison:* ce paquetage doit fournir un moyen de désigner et d'interconnecter les objets nécessaires à l'établissement de liaisons entre nœuds d'exécution.
- *Protocoles de communication:* ce paquetage doit permettre l'utilisation de stratégies de communication flexibles entre nœuds d'exécution.
- *Mobilité:* ce paquetage doit fournir les outils pour réaliser la migration de nœuds d'exécution, d'une machine virtuelle à l'autre.

Nous présentons ci-dessous les interfaces des trois premiers paquetages, le quatrième étant en cours de définition.

6.4.1 Topologie des nœuds d'exécution

Les machines virtuelles existantes disposent toutes:

- *D'une notion de nœud d'exécution,* identifiable à un composant de machine virtuelle, manifeste à l'exécution, et implantant un domaine du modèle de programmation. Par contre, la structure interne du nœud d'exécution dépend du modèle de programmation considéré. IMC doit donc fournir une abstraction de nœud d'exécution, mais sans chercher à en décrire la structure interne.
- *D'une notion d'identificateur* pour désigner un nœud d'exécution. Les interfaces d'IMC doivent donc refléter cette abstraction, mais sans en spécifier la structure interne qui reste spécifique d'un modèle de programmation.
- *De services pour créer de nouveaux identificateurs:* IMC doit donc aussi spécifier de type de service.
- *D'une organisation des nœuds d'exécution,* soit plate, soit arborescente. IMC doit supporter des schémas hiérarchiques de composition, plus généraux, et être capable de naviguer dans l'arborescence correspondante.
- *De primitives d'ajout de nouveaux nœuds, ou de suppression de nœuds existants*, utilisées lors des réductions. Ces primitives doivent donc aussi figurer parmi les interfaces d'IMC.

Ce constat suggère les interfaces suivantes:

```
public interface NodeIdentifier {}

public interface NodeIdentifierFactory {
    public NodeIdentifier newNodeIdentifier();
}

public interface Node {
    public NodeIdentifier getNodeIdentifier();
    public Object getImplementation();
}

public interface TopologyManager {
    public NodeIdentifier [] getSubNodes(NodeIdentifier node_id);
    public Node getSubNode(NodeIdentifier node_id, NodeIdentifier subnode_id);
    public void addSubNode(NodeIdentifier node_id, NodeIdentifier subnode_id);
    public void removeSubNode(NodeIdentifier node_id, NodeIdentifier subnode_id);
}
```

La première interface identifie de manière générique un nœud d'exécution. Des structures spécifiques d'identificateurs peuvent être obtenues en implémentant des classes respectant cette interface. Ces identificateurs sont créés par des générateurs de noms qui implémentent l'interface `NodeIdentifierFactory`. L'interface `Node` permet d'accéder au nom et à la structure interne d'un nœud exécution. Cette dernière devra être décrite par une classe spécifique d'une implémentation. Enfin, l'interface `TopologyManager` gère une arborescence de nœuds d'exécution, en fournissant des informations sur sa structure, et en permettant d'ajouter ou de supprimer des nœuds en un point donné de la hiérarchie.

6.4.2 Nommage et liaison

De nombreux calculs de processus exhibant une sémantique de domaine décrivent la communication comme un échange de noms sur des canaux, eux-mêmes identifiés par des noms. Les noms constituent une abstraction fondamentale de ces modèles et doivent donc être reflétés dans IMC. Au niveau implémentation, un nom n'est valide que dans un *contexte de nommage* associant un nom à l'entité qu'il désigne. Enfin, un nom est aussi une capacité de communication dont la connaissance doit permettre la création d'un chemin d'accès vers l'entité qu'il représente.

IMC inclut donc une notion d'*identificateur*, nom générique qui désigne un objet de manière unique dans un *contexte de nommage*, qui fournit des services de gestion de noms. IMC inclut un type particulier de contexte de nommage appelé *binder*, capable de créer un chemin d'accès vers un objet désigné par un nom donné. On en déduit les interfaces suivantes exploitant le schéma `export/bind` déjà décrit dans JONATHAN (cf. chapitre 3) pour séparer désignation et communication: `export` crée un nouveau nom pour rendre un objet accessible dans d'autres contextes de nommage; l'appel `id.bind` établit un chemin d'accès vers l'objet désigné par l'identificateur `id`.

```
public interface Identifier {
    public NamingContext getContext();
    public Object bind();
    public void unexport();
    public Object resolve();
    public byte [] encode();
    public void encode(Marshaller m);
}

public interface NamingContext {
    public Identifier export(Object obj);
    public Identifier decode (byte [] data);
    public Identifier decode(UnMarshaller u);
}
```

L'interface `Identifier` maintient une référence vers le contexte de nommage d'un identificateur et porte l'opération `bind` pour établir une liaison entre deux objets potentiellement distants. D'autres méthodes permettent d'annuler une opération `export` en rendant un identificateur invalide dans un contexte de nommage (méthode `unexport`) ou retournent le prochain élément dans une chaîne d'identificateurs obtenus par applications successives d'`export` (méthode `resolve`). L'interface gère aussi l'emballage d'un identificateur, en retournant un tableau d'octets ou une représentation d'un message prêt à être envoyé sur le réseau (interface `Marshaller`). L'interface `NamingContext` représente un contexte de nommage contenant des noms de type `Identifier`. Elle porte l'opération `export` et gère le déballage des identificateurs (méthodes `decode`).

6.4.3 Protocoles de communication

La diversité des protocoles de communication utilisés par les machines virtuelles existantes (cf. section 2.4.4) est telle qu'un canevas générique doit viser la *flexibilité*: IMC doit supporter des stratégies multiples d'emballage des messages et autoriser l'utilisation de protocoles dédiés, sans se restreindre à un ensemble fixe de primitives de communication ou de protocoles réseau. Le canevas de communication doit aussi être *minimal*: de nouveaux protocoles réseau doivent pouvoir être facilement introduits en implémentant directement les interfaces IMC, ou en définissant des incréments du canevas qui viendront compléter les interfaces et bibliothèques de base.

Un certain nombre de middleware adaptables (cf. section 2.5.2) fournissent des canevas minimaux et flexibles de communication en se basant sur la notion de liaison, déjà étudiée au chapitre 2. Cette approche présente l'avantage d'autoriser la construction de différents types de middleware ou « personnalités » et semble adaptée pour un canevas comme IMC où il n'y a pas d'accord sur un seul protocole de communication. La partie communication d'IMC s'appuie sur le modèle de nommage et de liaison défini précédemment et spécifie les interfaces des composants impliqués dans les interactions entre objets. La transparence des communications nécessaire pour implanter un calcul de processus donné est obtenue en utilisant directement les bibliothèques fournies ou en définissant de nouvelles personnalités au-dessus des abstractions de base d'IMC: on peut ainsi obtenir des liaisons « poids-mouche » directement basées sur TCP/IP comme dans l'expérimentation DCP, ou de type JAVA RMI pour une meilleure intégration avec le monde JAVA.

Les interfaces proposées sont inspirées de JONATHAN (cf. chapitre 3). Elle s'appuient sur les notions déjà définies de *protocole* (binder représentant un protocole réseau et gérant des identificateurs de session) et de *session* (canal logique de communication créé dynamiquement par un objet–protocole et disposant d'interfaces hautes et basses pour faire descendre ou monter un message dans les couches protocolaires constituées d'un empilement d'objets–session):

```
public interface Protocol {}

public interface ProtocolGraph {
    public SessionIdentifier export(Session_Low higher_level_session);
}

public interface SessionIdentifier {
    public Protocol getProtocol();
    public Session_High bind(Session_Low higher_level_session);
}

public interface Session_High {
    public void send(Marshaller m);
}

public interface Session_Low {
    public void send(UnMarshaller message, Session_High session);
}
```

L'interface `Protocol` représente un protocole réseau comme TCP ou GIOP. L'interface `ProtocolGraph` décrit l'empilement de micro-protocoles dans une couche de communication. Sa structure reflète le cheminement des messages lorsqu'ils sont envoyés ou reçus. La méthode `export` informe récursivement les différentes couches que l'objet-session exporté est prêt à recevoir des messages. Un identificateur de session alors est retourné. Il représente l'objet–session côté serveur et peut être transmis au client. Pour communiquer avec la session exportée, le client doit simplement appeler la méthode `bind` de l'identificateur de session qui créera un « surrogate » (de type `Session_High`) directement utilisable pour interagir avec la session serveur exportée.

Un identificateur de session désigne une session exportée. Il contient des informations spécifiques à un protocole: nom de machine, numéro de port, etc. Côté serveur, ces identificateurs sont emballés après l'opération `export` pour transmission sur le réseau. Ils seront déballés côté client, et utilisés pour établir un canal de communication à l'aide de l'opération `bind`. Le paramètre additionnel de cette méthode permet de recevoir les messages envoyés par l'objet–session distant. Les deux interfaces de session font circuler les messages dans la pile de protocoles. L'interface `Session_High` envoie les messages vers le réseau. Il s'agit en général d'un surrogate pour un objet de type `Session_Low` situé côté serveur, qui a été exporté vers un protocole. L'interface `Session_Low` redirige les messages provenant du réseau vers leur destination. Le paramètre supplémentaire de la méthode `send` représente l'émetteur du message et peut servir pour renvoyer une réponse, si nécessaire.

Des emballeurs et des déballeurs sont utilisés pour obtenir une représentation de haut-niveau, indépendante d'une plate-forme, des messages qui vont être envoyés ou reçus. L'interface `Marshaller` d'un emballeur est du type:

```
public interface Marshaller {
    public public void writeByte(byte b);
    public void writeBoolean(boolean b);
    public void writeChar(char i);
    public void writeShort(short i);
    public void writeInt(int i);
    ...
    public void writeIdentifier(Object obj);
    public boolean isLittleEndian();
    public void close();

}
```

L'interface `UnMarshaller` d'un déballeur est similaire, des opérations de lecture remplaçant les opérations d'écriture.

6.5 Conclusion

Dans ce chapitre décrivant la dernière étape de la démarche de conception d'un système à base de domaines par raffinements successifs, nous avons montré que le M–calcul était effectivement implantable, y compris en contexte réparti. Les interfaces de la plate-forme OYSTERS permettent de faire le lien entre l'implantation des domaines, la liaison répartie et les composants. Nous avons aussi vu que ces interfaces peuvent être utilisées pour implanter d'autres calculs de processus répartis présentant des sémantiques de domaines. Il est à présent temps d'évaluer le travail réalisé en comparant les résultats d'ensemble obtenus par rapport aux objectifs fixés initialement.

Chapitre 7

Evaluation

L'objet de ce chapitre est de faire le point sur les résultats obtenus dans les chapitres précédents pour examiner dans quelle mesure les objectifs initiaux ont été atteints. Nous commençons par rappeler la problématique et le travail effectué (cf. section 7.1). Nous résumons ensuite l'intérêt des domaines comme abstraction et primitive pour le « global computing » (cf. section 7.2) avant d'évaluer la démarche proposée pour construire des applications fiables dans de tels environnements (cf. section 7.3). Enfin, nous tentons d'évaluer qualitativement les résultats obtenus, au niveau du modèle de programmation, de la machine abstraite CLAM et de la machine virtuelle OYSTERS (cf. section 7.4). Le chapitre se termine sur quelques perspectives concernant des travaux futurs.

7.1 Rappel de la problématique et du travail réalisé

7.1.1 Rappel de la problématique

Les besoins en fiabilité des applications réparties se sont largement accrus à cause de la croissance en taille et en complexité des infrastructures distribuées, ce qui implique le développement de méthodes et d'outils appropriés. Ce problème est longtemps resté non-résolu en dépit du développement des méthodes formelles dont l'application s'est jusqu'ici principalement restreinte à des systèmes centralisés ou répartis de taille réduite, sans véritablement s'attaquer à des environnements de type « global computing ». L'obstacle rencontré était double:

P1 *Un manque d'abstractions, modèles et primitives de programmation pour le « global computing »*, domaine de recherche nouveau. Cette lacune se traduit par un manque de plates-formes intégrées, et une diversité impressionnante d'infrastructures explorant tel ou tel aspect de la programmation répartie, mais difficilement comparables.

P2 *Une divergence entre théorie et pratique de la programmation répartie*. D'un côté, les méthodes formelles ne sont pas suffisamment intégrées au monde du génie logiciel car jugées trop coûteuses, difficiles d'accès et seulement manipulables par quelques spécialistes. De l'autre, la sémantique des infrastructures middleware actuelles déployées sur des environnements à large échelle reste malheureusement souvent empirique, et peu explicitée.

7.1.2 Rappel de la démarche utilisée

Dans ce travail de thèse, après avoir identifié certaines des exigences que doit satisfaire un modèle de programmation répartie à large échelle, nous avons souligné l'intérêt de la notion de domaine comme abstraction fondamentale possible pour le « global computing », en montrant notamment qu'il s'agit d'une abstraction présente – en général sous forme hautement implicite – dans la plupart des modèles existants de programmation répartie, dans les machines virtuelles qui les implémentent ainsi que dans les plates-formes middleware les plus répandues.

Nous avons ensuite présenté une méthodologie simple pour la construction d'applications réparties dignes de confiance où l'implantation s'effectue par raffinements successifs: définition d'un modèle formel de programmation répartie fondé sur la notion de domaine; spécification formelle d'une implémentation du modèle par une machine abstraite; implémentation du modèle par une machine virtuelle répartie utilisant les technologies middleware courantes (cf. chapitre 2).

Dans une expérimentation préliminaire nous avons cherché à déterminer le bien-fondé et la faisabilité de cette méthodologie dans le cadre d'un modèle de programmation appelé DCP manifestant une forme simple de domaine (cf. chapitre 3). Nous avons ensuite appliqué cette démarche à un véritable calcul de processus réparti, explicitement fondé sur la notion de domaine, le M–calcul (cf. chapitre 4): nous avons défini une machine abstraite centralisée puis répartie pour ce calcul appelée CLAM (cf. chapitre 5) puis implanté cette spécification formelle sous forme d'une machine virtuelle appelée OYSTERS, dans un cadre centralisé puis distribué, montrant ainsi l'effectivité du modèle de programmation de départ (cf. chapitre 6).

7.1.3 Contributions de la thèse

En résumé, les deux contributions principales de cette thèse sont:

C1 La démonstration de l'intérêt des domaines comme primitive de programmation pour le « global computing ».

C2 La mise en oeuvre d'une méthodologie de formalisation par raffinements successifs pour obtenir des infrastructures middleware réparties fiables.

Dans la suite du chapitre, nous allons tenter d'évaluer ces contributions par rapport à la double problématique initiale P_1 et P_2.

7.2 Intérêt du concept de domaine

La notion de domaine constitue une avancée en terme de théorie de la programmation répartie. Son intérêt est double en fournissant des éléments de réponse à la fois aux problèmes $P1$ et $P2$.

7.2.1 Une abstraction fondamentale pour le « global computing »

L'identification explicite de la notion de domaine répond en partie au problème $P1$ en définissant une abstraction unique pour le « global computing » et rendant compte des observables identifiées au chapitre 2:

localités virtuelles et physiques, fluctuations de bande passante et défaillances. L'adéquation de cette notion par rapport à ces observables peut être mesurée qualitativement en comparant le modèle de programmation fondé sur la notion de domaine étudié – le M–calcul – par rapport aux exigences pour un modèle du « global computing » définies au chapitre 2. Le résultat obtenu est représenté dans la figure 7.1.

Nom de l'exigence	**Caractéristique**	**Degré de satisfaction**
`REQ.ASYNCHRONY`	Asynchronisme	++
`REQ.PROCESS`	Activité	++
`REQ.COMPOSITION`	Composition	++
`REQ.LOCALITY`	Localité	+++
`REQ.SECURITY`	Sécurité	+ / ++
`REQ.FAILURES`	Défaillances	+ / ++
`REQ.COMMUNICATION`	Communication	+++ ; –
`REQ.MIGRATION`	Migration	+++ ; –
`REQ.NAMING`	Désignation	++
`REQ.DYNAMIC_BINDING`	Liaison dynamique	++
`REQ.CONTROL`	Contrôle	+++
`REQ.COORDINATION`	Coordination, synchronisation	+
`REQ.RESOURCES`	Ressources	+

FIG. *7.1* – **Satisfaction par le M–calcul des exigences pour un modèle du « global computing ».**

La satisfaction des trois premières exigences provient de la forme même du modèle de programmation – un calcul de processus réparti – qui représente et manipule les activités sous forme de processus pouvant être composés à l'aide d'un opérateur de composition parallèle et où les interactions sont asynchrones.

L'exigence de localité est pleinement satisfaite puisque cette notion constitue la primitive de base du calcul et la manifestation de l'abstraction de domaine en définissant un partitionnement spatial et logique de l'exécution. On notera que la réduction du calcul vérifie une propriété de localité: une localité n'évolue que selon son état et celui des localités directement voisines, sans impliquer de synchronisations cachées, comme le prouve l'implantation du calcul. L'organisation hiérarchique des localités traduit l'imbrication des différents domaines d'administration et d'exécution dans la réalité. Cependant, dans la modélisation, le calcul se limite à des localités qui ne se recouvrent pas, sans partage possible de sous-localités: la structure des localités reste donc une arborescence et non un graphe. Ce dernier point semble un sujet d'étude particulièrement difficile.

L'exigence de sécurité est en partie satisfaite dans la mesure où une localité du M–calcul peut filtrer les messages entrant et sortant selon un comportement arbitraire, équivalent d'un pare-feu ou d'une stratégie de contrôle d'accès. Toutefois, la politique de sécurité ne fait pas vraiment partie du modèle, cet aspect du calcul n'ayant été que relativement peu exploré. De même, en terme de défaillances, l'exigence est remplie puisque le calcul peut exprimer des modèles de pannes comme celui du $\pi_{1\ell}$–calcul ou du join–calcul. Cependant, ces modèles se limitent à des modes de défaillances de type fail-stop. Par la suite, il serait intéressant de pouvoir expérimenter d'autres types de défaillances, par exemple avec reprise.

Le M–calcul semble parfaitement adapté au « global computing » en terme de communication et de migration: ses primitives d'envoi de messages point-à-point reflètent les infrastructures réparties existantes. En M–calcul, l'interaction distante n'est pas primitive mais prend la forme d'une série d'étapes atomiques non masquables sans faire abstraction des barrières de localités. La transparence des communications réseau reste toutefois possible en utilisant des relais de messages. Le modèle est capable d'exprimer plusieurs sémantiques d'interaction en filtrant les messages à la frontière d'un domaine et en pouvant les rediriger vers d'autres domaines selon une stratégie programmable: si le calcul dispose d'un mécanisme de routage implicite de proche en proche, il est aussi possible de spécifier explicitement la destination d'un message. La migration de domaines est codée à l'aide de la communication grâce à une combinaison originale de passivation et d'ordre supérieur. La migration s'effectue donc aussi étape par étape et dispose des mêmes propriétés de contrôle que pour la communication, des formes objectives ou subjectives de mobilité étant programmables. Ces points forts du modèle doivent être toutefois nuancés par la complexité de la stratégie de routage, qui se manifeste par un nombre de règles de réduction important. Cette complexité s'explique par la combinatoire des situations pour faire migrer les messages entre l'extérieur d'un domaine, sa membrane et son contenu. Cette complexité est accentuée dans l'implémentation puisque l'on doit aussi tenir compte de la co-localisation physique possible des domaines de départ et d'arrivée du message.

Communication et migration s'appuient sur des mécanismes de désignation qui satisfont en grande partie les exigences `REQ.NAMING` et `REQ.DYNAMIC_BINDING`: le calcul manipule différents types de noms pour désigner ressources et localités et autorise la création de nouveaux noms. La résolution des noms fait partie du calcul en étant intégrée aux règles de routage. Toutefois, les contextes de nommage restent implicites et s'identifient aux domaines d'exécution. Le modèle dispose aussi d'une forme de liaison dynamique, le comportement déclenché lors de la réception d'un message par une ressource dépendant fortement du domaine où se trouve le message.

L'exigence de contrôle est la mieux satisfaite, dans la mesure où la notion de localité programmable qui constitue l'originalité du M–calcul permet de superposer un comportement de contrôle assez arbitraire sur les processus et localités qui sont contrôlés. On peut ainsi obtenir différentes formes de domaines à l'intérieur du même modèle de programmation. L'interception des messages entrant ou sortant d'un domaine exerce aussi un contrôle sur la communication et la migration, en permettant la programmation de stratégies explicites ou en assurant la sécurité d'une localité.

Enfin, d'autres aspects de l'exécution comme la gestion des ressources ou la coordination ont été moins pris en compte ou étudiés dans ce modèle. Toutefois, les localités du M–calcul disposent du mécanisme de synchronisation hérité du join–calcul à l'aide de filtres de messages. Elles permettent aussi de programmer des comportements de coordination des membres d'un domaine, comme dans KLAIM.

Ainsi, le M–calcul, en alliant localités programmables, ordre supérieur, passivation, et liaison dynamique paraît bien adapté comme modèle de programmation pour le « global computing ». Plus généralement, l'intérêt de la notion de domaine est de disposer d'une abstraction aussi bien reflétée dans les modèles existants que dans les nombreuses plates-formes réparties sous-jacentes. Cette double unité de composition et de contrôle a l'avantage principal de traiter de manière uniforme les différents types de partitions qui se présentent dans un système réparti de grande taille, identifiées comme autant de déclinaisons différentes de la sémantique de domaine. En ce sens, l'identification de la notion de domaine comme abstraction possible pour la programmation répartie à large échelle est susceptible de constituer une avancée importante pour les fondements et la programmation du « global computing ».

7.2.2 Une réduction de la divergence entre théorie et pratique de la programmation répartie

La notion de domaine fournit aussi une réponse au problème $P2$ car elle contribue à réduire la divergence entre théorie et pratique de la programmation répartie.

Depuis une vingtaine d'années, on assiste à une croissance sans précédent des besoins en informatique. La pratique de la programmation tente de suivre, mais en proposant souvent des solutions au cas par cas ou visant le court terme. Avec le développement des technologies de l'information, le champ à modéliser du point de vue théorique est devenu de plus en plus vaste. On constate cependant que, globalement, les modèles de programmation comme les calculs de processus représentent toujours plus finement la réalité en introduisant progressivement de nouvelles primitives. L'impact de ces nouveaux concepts est direct en terme de langages de programmation puisqu'un calcul de processus constitue un cœur de langage. Chaque étape majeure est l'occasion d'un rapprochement entre théorie et pratique de la programmation répartie: on assiste à chaque fois à la naissance de paradigmes de programmation fournissant aux développeurs de nouveaux outils pour programmer plus efficacement les applications répondant aux besoins du moment. On observe aussi la diffusion progressive du formalisme vers les techniques de programmation courante et donc vers la communauté des développeurs. La figure 7.2 résume quelques unes de ces étapes importantes.

Paradigme de programmation	**Concepts clés**	**Modèles principaux**
Programmation séquentielle/fonctionnelle	Fonction	λ–calcul
Programmation concurrente	Processus	π–calcul
Programmation répartie mobile	Localité	π–calcul à localités; join–calcul ; ambients; KLAIM, ...
Programmation répartie mobile à large échelle	Domaine,...	M–calcul ,...

FIG. *7.2* – **Paradigmes de programmation: concepts et modèles fondamentaux.**

Dans cette perspective, l'intérêt de la notion de domaine est d'appliquer à l'univers nouveau du « global computing » un concept déjà répandu dans la pratique des systèmes depuis longtemps – par exemple dans les techniques matérielles et logicielles de protection – pour répartir le contrôle en définissant des partitions logiques dont chaque élément est contrôlé par une autorité. Cette notion est également apparue plus récemment dans les calculs de processus sous forme de localité. On dispose donc d'une abstraction reflétée tant dans la théorie que dans la pratique de la programmation répartie.

Dans les chapitres précédents, nous avons montré que la notion de domaine est effective. Autrement dit, un modèle de programmation fondé sur cette notion comme le M–calcul est directement implémentable en pratique. Ce résultat pourrait être l'occasion d'un nouveau point de synchronisation entre théorie et pratique du « global computing », puisque les résultats théoriques sur les domaines sont directement transposables en pratique sous forme de liaisons et de composants. L'impact sur les techniques actuelles et futures de programmation répartie à large échelle est donc potentiellement très important.

7.3 Intérêt de la méthodologie de formalisation par raffinements successifs

Si, en démontrant l'intérêt de la notion de domaine comme nouvelle primitive effective de programmation répartie, la contribution $C1$ de la thèse a un impact d'ordre théorique et pratique, la contribution $C2$ se place, elle, résolument sur le plan pratique. Il s'agit d'appliquer une méthodologie de formalisation d'un système par raffinements successifs à la construction à base de domaines d'infrastructures réparties fiables.

En appliquant les avancées en génie logiciel à la nouvelle primitive de programmation que constituent les domaines, cette contribution fournit des éléments de réponse au problème $P2$: la méthodologie éprouvée de formalisation par raffinements successifs depuis la phase de spécification jusqu'à celle d'implantation permet ainsi de vérifier de manière expérimentale la capacité d'un calcul à base de domaines à être implanté en contexte réparti. L'intérêt de cette méthodologie est de relier théorie et pratique de la programmation répartie à large échelle, en faisant passer de l'une à l'autre par un continuum d'étapes de raffinement, depuis le modèle formel jusqu'aux infrastructures middleware existantes.

Dans cette thèse, la méthodologie se limite à deux étapes de raffinement R_1 et R_2: R_1 transforme le calcul en une spécification formelle prenant la forme d'une machine abstraite; R_2 transforme la spécification formelle en son implantation. La figure 7.3 résume les éléments de la démarche selon leur niveau d'abstraction par rapport au matériel, à la fois en terme de modélisation et d'exécution.

On désigne par $\mathcal{M}$ l'ensemble des éléments de la démarche, qui seront notés $\mathsf{MP}, \mathsf{L}, \mathsf{CN}, \ldots$. On suppose donnée une fonction $\mu : \mathcal{M} \rightarrow \mathbb{N}$ qui mesure le niveau d'abstraction ou de complexité de chaque élément de $\mathcal{M}$ par rapport au matériel. On définit une relation d'ordre $\prec$ (resp. $\preceq$) sur $\mathcal{M}$ par : $\mathsf{L} \prec \mathsf{MP}$ (resp. $\mathsf{L} \preceq \mathsf{MP}$) ssi $\mu(\mathsf{L}) < \mu(\mathsf{MP})$ (resp. $\mu(\mathsf{L}) \leq \mu(\mathsf{MP})$) .

Les éléments de la démarche sont de deux types:

- *Les éléments fixes*: il s'agit d'une part du modèle et du langage de programmation à implanter (notés respectivement MP et L) et d'autre part du jeu d'instructions–machine (noté CN) de l'architecture matérielle sur laquelle s'exécutera l'implantation du modèle. Ces différents éléments vérifient la relation $\mathsf{CN} \prec \mathsf{L} \preceq \mathsf{MP}$. On suppose que, comme dans la plate-forme OYSTERS, l'implantation prend la forme d'un couple (compilateur, machine virtuelle): le compilateur transforme le langage en code intermédiaire ou byte-code, ensuite exécuté par la machine virtuelle.
- *Les éléments variables*: il s'agit de degrés de liberté pour l'implantation du modèle, soit d'une part, le niveau d'abstraction la machine virtuelle, et en particulier le degré de complexité du byte-code (noté BC); d'autre part, celui de la machine abstraite (noté MA) qui reflète à un niveau de détail plus ou moins proche du système, les structures d'exécution de la machine virtuelle. Ces éléments vérifient les relations $\mathsf{CN} \preceq \mathsf{BC} \preceq \mathsf{L}$ et $\mathsf{BC} \preceq \mathsf{MA} \preceq \mathsf{MP}$, qui mesurent la latitude du programmeur pour la spécification et l'implantation du modèle.

Les degrés de liberté précédents jouent d'une part sur la complexité de réalisation des composants de la plate-forme ainsi que sur leur efficacité, et d'autre part sur la difficulté de preuve de la correction des raffinements entre étapes de la méthodologie:

- *Complexité de réalisation:* lorsque $\mu(\mathsf{BC})$ augmente, le byte-code s'éloigne du système et se rapproche du langage de programmation. Le compilateur est plus simple à réaliser, mais la machine virtuelle devient plus complexe. Elle est aussi moins efficace, de nombreuses instructions–machine

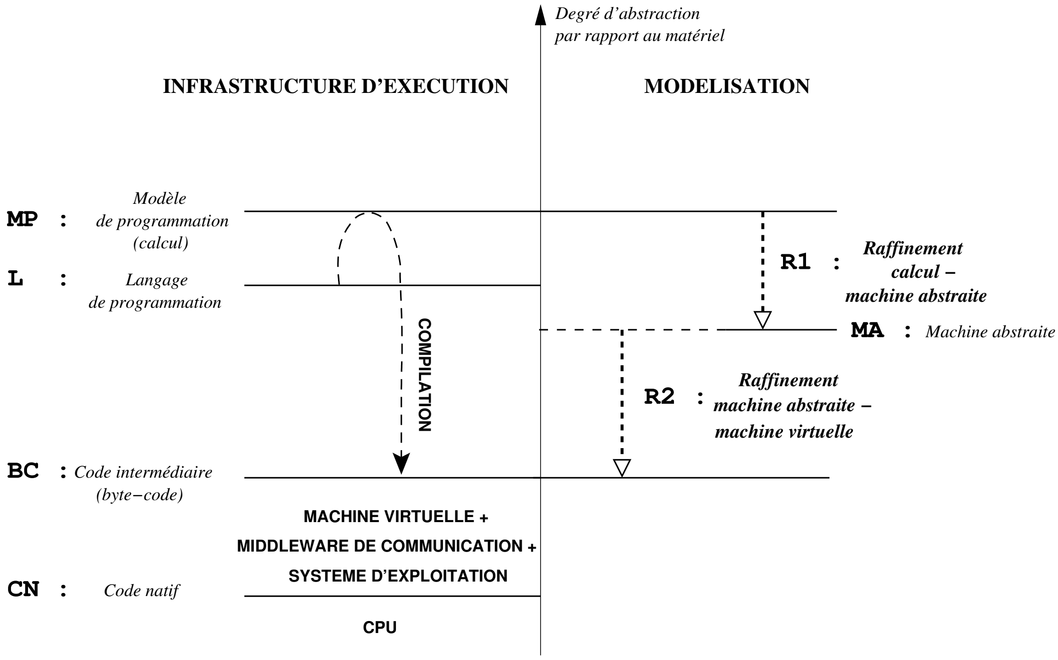


FIG. *7.3* – **Formalisation d'un système par raffinements: aspects modélisation et exécution.**

devant être exécutées pour une seule instruction du byte-code. Réciproquement, lorsque $\mu(\mathsf{BC})$ diminue, le byte-code s'éloigne du langage et se rapproche du code natif. La machine virtuelle est alors plus simple et plus efficace, le compilateur devenant plus difficile à réaliser.

Si $\mathsf{BC} = \mathsf{L}$, le compilateur est inexistant, le langage étant interprété. Si $\mathsf{BC} = \mathsf{CN}$, il n'y a pas de machine virtuelle, le langage étant compilé.

- *Difficulté de preuve de correction des raffinements:* lorsque $\mu(\mathsf{MA})$ se rapproche de $\mu(\mathsf{MP})$, la traduction R_1 du modèle de programmation vers la machine abstraite est simplifiée. De même, la preuve de correction de la machine abstraite par rapport au modèle est facilitée. En revanche, la traduction R_2 de la spécification vers l'implémentation devient plus compliquée, tout comme la preuve de correction correspondante. Inversement, lorsque $\mu(\mathsf{MA})$ se rapproche de $\mu(\mathsf{BC})$, le raffinement R_1 et sa correction deviennent plus complexes à formuler et à prouver. Le raffinement R_2 et sa correction s'en trouvent simplifiés.

  Lorsque $\mu(\mathsf{MA}) = \mu(\mathsf{BC})$, on a formalisation de bout en bout, et, si R_1 est correct, l'implantation du modèle peut être qualifiée de « prouvablement correcte ».

  Lorsque $\mu(\mathsf{MA}) \approx \mu(\mathsf{MP})$, la preuve de correction de R_1 est relativement aisée car il devient facile de faire correspondre les réductions de la machine et avec celles du calcul puisque ces deux relations coïncident pratiquement, la distance $\Delta_1 = \mu(\mathsf{MP}) - \mu(\mathsf{MA})$ étant faible. Par contre, R_2 devient un raffinement faisant passer directement du modèle de programmation à son implantation. L'importance de la distance $\Delta_2 = \mu(\mathsf{MA}) - \mu(\mathsf{BC})$ montre que quasiment la totalité du travail reste à faire pour développer la plate-forme et prouver sa correction par rapport au modèle.

En ce sens, l'intervalle $[\mu(\mathsf{BC});\mu(\mathsf{MA})]$ constitue une brèche plus ou moins importante dans la formalisation de bout en bout. La distance Δ_2 mesure l'écart qui sépare l'implémentation d'une véritable plate-forme prouvablement correcte. Cet écart peut être comblé si l'on dispose d'outils de développement (compilateur, optimiseur, etc.) préalablement certifiés qui prennent en entrée une spécification et génèrent le code intégral de la plate-forme. C'est rarement le cas dans la pratique, sauf pour le développement de systèmes hautement critiques. Pour les modèles de programmation existants disposant d'une machine abstraite, R_1 est parfois défini de manière formelle afin de prouver sa correction. Par contre, en général, R_2 est peu formalisé et reste implicite. La preuve de sa correction est encore plus rare.

Pour garantir la sûreté des infrastructures réparties à large échelle, l'intérêt de la méthodologie employée est de faire passer de manière quasi-continue d'un modèle de programmation à son implantation. Le bénéfice principal de la formalisation est d'expliciter de manière précise la sémantique de l'infrastructure, en la décrivant à l'aide d'un modèle mathématique.

Toutefois, par rapport aux pratiques de développement dans l'industrie du logiciel, la méthodologie employée ici présente certaines limites. Nous nous sommes limités dans ce travail à simplement deux étapes de raffinement afin de donner une première idée de la démarche de formalisation. Un cycle réel de développement mettrait en oeuvre de nombreuses étapes supplémentaires de raffinement, en incluant plusieurs degrés de spécification: spécification à gros-grain, spécifications détaillées de certains composants, etc.

D'autre part, nous avons vu que les raffinements utilisés ne sont pas toujours prouvés corrects en pratique. Les raisons peuvent être d'ordre théorique (e.g., difficulté de la preuve, insuffisance des techniques de preuve existantes) ou motivées par des coûts de mise en oeuvre prohibitifs (e.g., taille de la preuve dépassant les capacités des prouveurs de théorèmes). Une conséquence importante pour une infrastructure à large échelle est que l'on peut ainsi distinguer:

- Les composants du système ayant été spécifiés formellement avec preuve de correction.

- Les composants ayant été spécifiés formellement, mais où les raffinements n'ont pas été prouvés corrects ou sont présentés de manière informelle.

- Les composants n'ayant pas été formalisés et ayant été implantés « tels quels ».

Selon le niveau de formalisation atteint, on peut ainsi qualifier le degré de confiance à accorder aux différents composants d'une infrastructure. On retrouve cette classification en certification des systèmes, où l'on distingue plusieurs niveaux de certification, comme par exemple dans l'échelle des Critères Communs [43]. L'objectif initial de la méthodologie présentée était une formalisation complète d'un système à base de domaines depuis la spécification jusqu'à l'implémentation. En pratique, par exemple pour la certification de systèmes d'exploitation sécurisés, la formalisation d'un système complet paraît hors de portée en l'état actuel des techniques de preuve. Cette méthodologie ne semble donc pas applicable à la conception de la totalité d'un système. Elle se révèle par contre très utile au développement de composants critiques d'une infrastructure répartie qui méritent alors une formalisation poussée. Les composants non-critiques peuvent alors se contenter de degrés de formalisation moindres.

7.4 Evaluation des résultats obtenus

Mise en oeuvre de la méthodologie pour l'implantation du M–calcul

En terme de méthode de développement, nous avons appliqué la technique de génie logiciel consistant à réaliser un premier prototype pour évaluer la méthodologie avant de passer à une réalisation plus importante. En ce sens, l'expérimentation DCP a permis de tester la mise en oeuvre d'un modèle de programmation manifestant une sémantique de domaine simple au sein d'une plate-forme à objets répartis classique. DCP a servi de champ d'expérimentation pour relier, au cours des différentes étapes de formalisation, les abstractions du modèle de programmation (canaux, processus, domaines, ...) aux notions mises en oeuvre dans les plates-formes middleware réparties (protocoles, sessions, connexions, contextes de nommage, ...). L'expérience ainsi acquise a permis d'identifier les techniques d'implantation adaptées aux spécificités du modèle de programmation et les choix de conception importants dans l'architecture d'une machine virtuelle à base de domaines. Cette expérience a été directement applicable pour la réalisation de la machine virtuelle OYSTERS en contexte centralisé ainsi que pour le développement de liaisons réparties. Cette expérimentation préliminaire a réduit de manière importante le temps de conception et de développement de la machine virtuelle OYSTERS.

Nous avons alors été en mesure de définir assez rapidement une première version de la machine abstraite CLAM dans un contexte centralisé et de développer la machine virtuelle OYSTERS correspondante. L'implémentation représente environ 4600 lignes de code OCAML pour le compilateur et 10000 lignes de code JAVA pour la machine virtuelle. La phase suivante de développement a été plus longue et difficile. Elle a été consacrée à la vérification de la validité des règles de réduction de la machine abstraite et de leur implantation, en exécutant divers programmes de test pour étudier l'interaction des différentes primitives du M–calcul. Cette démarche a permis de relever certaines erreurs dans la conception initiale de la machine abstraite et de raffiner son architecture. L'interface graphique de la machine virtuelle OYSTERS a été développée spécifiquement pour exécuter l'interpréteur de byte-code en mode pas-à-pas, avec possibilité d'orienter le non-déterminisme des règles de la machine abstraite dans une direction définie par le programmeur. Cette interface s'est avérée très utile pour détecter les erreurs dans l'interprétation du byte-code, ou, lorsque le byte-code généré était incorrect, pour corriger le code du compilateur. Notons qu'une part non négligeable du temps de développement a été consacrée à l'implantation de l'évaluateur fonctionnel et à son intégration dans le moteur d'exécution de la machine virtuelle, de nombreux cas de terminaison de l'évaluation étant possibles.

Effectivité du modèle de programmation

Le premier résultat obtenu est la démonstration de l'effectivité d'un modèle de programmation fondé sur la notion de domaine par la réalisation d'une machine virtuelle répartie implantant le M–calcul. Une évaluation qualitative de ce résultat consiste à comparer l'adéquation du couple (modèle de programmation, implantation) en terme d'exigences de réduction de la divergence entre théorie et pratique de la programmation répartie à large échelle (cf. chapitre 2). Les résultats sont rassemblés dans la figure 7.4.

Nom de l'exigence	Caractéristique	Degré de satisfaction
REQ.EFFECTIVENESS	Effectivité	++
REQ.DESIGN_TO_IMPLEMENTATION	Raffinement	++
REQ.NO_HIDDEN_COSTS	Transparence des coûts d'implantation	++
REQ.SIMPLICITY	Simplicité	+
REQ.COMPLETENESS	Complétude	?
REQ.OPENNESS	Ouverture	++

FIG. *7.4* – **Couverture par le couple (M–calcul,OYSTERS) des exigences de réduction de la divergence entre théorie et pratique du « global computing ».**

L'objectif d'effectivité est en grande partie satisfait puisque l'on obtient une implantation répartie du M–calcul sous forme d'une machine virtuelle de complexité raisonnable. Il serait possible d'améliorer les performances de la plate-forme obtenue par de nombreuses optimisations. Mais, cette tâche dépasse le cadre que nous nous étions fixé pour cette étude, qui visait uniquement à déterminer la faisabilité d'implantation d'un modèle à base de domaines.

L'objectif de raffinement est lui-aussi satisfait à cause de la méthodologie employée qui relie les primitives du modèle de programmation et leur implantation à l'aide d'une spécification formelle. Toutefois, la correction de la machine abstraite CLAM par rapport au M–calcul reste encore une question ouverte, la preuve semblant très difficile à réaliser. Pour attaquer ce problème, il semble nécessaire de disposer d'outils théoriques plus puissants que ceux actuellement disponibles. A titre de comparaison, la preuve de correction de l'infrastructure NOMADIC PICT [165] par rapport au calcul sous-jacent a nécessité l'introduction de nouvelles techniques de preuve comme des notions avancées de bisimulations et d'équivalences (« translocating bisimulations ») pour, en présence de mobilité dans le système, pouvoir prouver des propriétés sur un processus indépendamment de la localité qui l'abrite. Ainsi, il serait intéressant de développer la théorie de l'équivalence pour le M–calcul, afin d'étudier précisément quels types de propriétés formelles sont vérifiées par des programmes s'exécutant sur la plate-forme OYSTERS. Toutefois, ces travaux dépassent largement le cadre de cette thèse.

L'objectif de transparence des coûts d'implantation paraît atteint, dans la mesure où le modèle de programmation, comme le join–calcul, vérifie une propriété de localité: l'évolution d'une localité ne dépend que de localités voisines, mais pas de localités distantes. Cette propriété est reflétée dans les règles de la machine abstraite répartie, qui ne spécifie le comportement que d'une machine virtuelle à la fois, indépendamment des autres machines virtuelles connectées sur un réseau OYSTERS.

La satisfaction des exigences précédentes doit être tempérée quant à la simplicité du résultat obtenu. D'une part, le modèle inclut un mécanisme relativement complexe de routage entre domaines, qui se retrouve dans la spécification et dans l'implantation. Cette complexité se justifie par la présence d'une double barrière dans les domaines du M–calcul – entre l'extérieur d'un domaine et le contrôleur de domaine, et entre le contrôleur de domaine et son contenu – contrairement aux localités d'autres calculs de processus qui ne possèdent qu'une simple barrière – entre l'extérieur et l'intérieur de la localité.

D'autre part, le modèle de programmation inclut le λ–calcul dans son ensemble, alors que les primitives d'application fonctionnelle et de λ–abstraction ne sont principalement utilisées que pour geler et dégeler des processus. Aussi, avec le recul, n'était-il peut-être pas indispensable d'aller aussi loin dans l'implantation d'une stratégie complète d'évaluation d'expressions fonctionnelles, par exemple en prenant en compte le partage possible de sous-expressions. Cela a toutefois permis d'explorer plus en détail l'interface entre les primitives fonctionnelles et de gestion de la répartition dans le M–calcul. Ces écueils sont largement compensés en pratique, par le concept nouveau de localité programmable.

La complétude du modèle de programmation reste plus difficile à évaluer. Elle signifie que le modèle est capable d'exprimer le comportement de programmes évoluant sur un réseau à large échelle. Elle pourrait être obtenue en définissant, comme le fait le calcul des ambients dans [40], un encodage des machines de Turing en M–calcul. Toutefois, la pertinence d'un tel encodage n'est pas évidente pour un réseau de grande taille.

Niveau de complexité de la machine abstraite

Pour tenter d'évaluer qualitativement la complexité de la machine abstraite par rapport au calcul de départ, nous comparons un certain nombre de calculs de processus avec leurs implantations, en utilisant comme critère de mesure le nombre de règles de réduction définissant la sémantique de ces modèles, et celui que comprend la définition de leurs machines abstraites. Bien que moyennement précise, cette mesure permet de donner une idée de la complexité de la CLAM par rapport au M–calcul. Les résultats sont résumés dans la figure 7.5. A titre de comparaison, est représentée la complexité du λ–calcul, du π–calcul, et de leurs implantations, bien qu'il ne s'agisse pas de modèles de programmation répartis.

On constate que l'enrichissement d'un modèle de programmation par introduction de la notion de domaine se traduit par une augmentation significative du nombre des règles de réduction du calcul et donc de celles de la machine abstraite, comme le montre le passage du couple (join–calcul, JAM) au couple (CLAM, M–calcul), qui pourrait mesurer le coût de l'introduction de la notion de domaine dans le join–calcul. Cependant, on remarque aussi que cet enrichissement s'effectue sans coefficient multiplicateur entre le modèle et son implantation, comme le montre le parallélisme des courbes de complexité du calcul et de la machine abstraite entre les deux couples précédents. L'impact du développement de la notion de domaine est donc comparable au niveau du calcul et de la machine abstraite, sans accroissement supplémentaire de complexité, lié à l'incarnation des domaines dans une plate-forme middleware.

Modularité et extensibilité de la machine virtuelle

Les propriétés attendues de l'implantation du M–calcul (cf. chapitre 6) sont pour une grande part vérifiées: simplicité relative, portabilité, et modularité/extensibilité.

La machine virtuelle réalisée démontre le caractère implantable du M–calcul. Elle confirme l'intérêt de la démarche d'ensemble de formalisation suivie dans cette étude, depuis la conception d'un modèle de programmation répartie sous forme de calcul de processus jusqu'à son implantation par une machine virtuelle, en passant par une spécification formelle sous forme de machine abstraite.

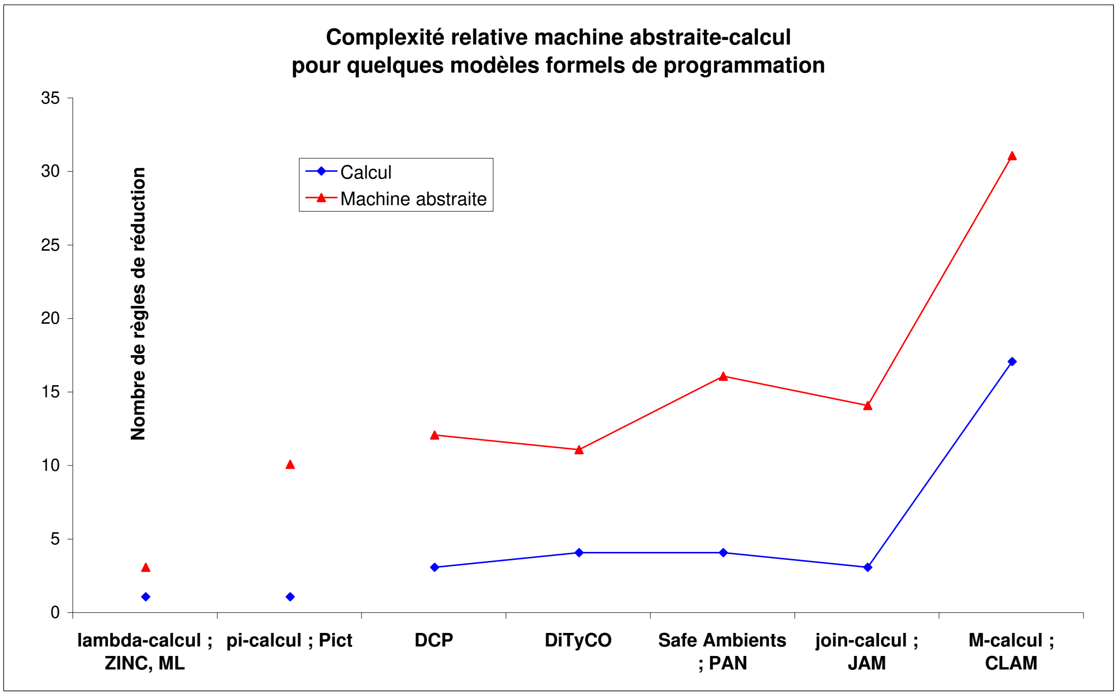


FIG. *7.5* – **Complexité de modèles de programmation et de leur machine abstraite.**

Le caractère portable de la plate-forme est garanti par les langages de programmation utilisés pour la développer (OCAML, JAVA). Les expérimentations en cours d'intégration de la machine virtuelle OYSTERS avec des plates-formes à objets répartis confirment qu'il est possible de faire communiquer des implémentations du M–calcul de manière indépendante d'une architecture matérielle donnée, l'interface entre la machine virtuelle et la plate-forme middleware étant relativement minimale.

Enfin, la conception modulaire de la machine virtuelle permet une flexibilité certaine en terme: de stratégie d'évaluation des expressions fonctionnelles, tant au niveau de la spécification que de l'implantation; de stratégie de déclenchement des règles de réduction, globalement ou au niveau d'un oyster; ou d'isolation de différents services de communication (nommage, localisation). Sa structure modulaire lui permet d'intégrer relativement facilement de nouvelles fonctionnalités, qui sont décrites comme autant de règles de réduction supplémentaires dans la machine abstraite. Enfin, les interfaces identifiées, en séparant clairement gestion du nommage, communication, mobilité et topologie des domaines, rend une conception de machine virtuelle à composants potentiellement applicable à l'implantation d'autres calculs de processus, comme le montre le canevas logiciel IMC en cours de réalisation.

Comparaison avec l'état de l'art

On assiste à un nombre croissant de tentatives de formalisation des infrastructures réparties d'exécution pour la mobilité. Les travaux les plus proches de la plate-forme CLAM–OYSTERS sont NOMADIC

PICT [175, 176], l'implantation répartie PAN [149] du calcul des SAFE AMBIENTS, DITYCO [111, 112], et l'implantation JOCAML [96, 98] du join–calcul, spécifiée par une machine abstraite appelée JAM [138].

On peut considérer que PICT [140, 164] a formé le point de départ de ces différents travaux en formalisant une implémentation concurrente du π–calcul. Cependant, PICT ne prenait pas en compte les problèmes de répartition et de mobilité.

NOMADIC PICT a étendu PICT pour pouvoir spécifier et construire des infrastructures d'agents mobiles sur lesquelles on puisse prouver des propriétés formelles. Plusieurs différences importantes séparent NOMADIC PICT du M–calcul : NOMADIC PICT n'est pas un modèle de programmation d'ordre supérieur; les formes de domaines de NOMADIC PICT ne sont pas véritablement programmables; enfin, NOMADIC PICT ne dispose pas de modèle hiérarchique de domaines et se contente d'une description plate. Contrairement à l'approche suivie dans cette étude, NOMADIC PICT ne formalise pas une infrastructure répartie par une machine abstraite. La plate-forme distribuée est modélisée en définissant une traduction de primitives de haut-niveau de communication transparente en des primitives de bas-niveau, dépendant de la localisation. Bien que la spécification formelle ait une forme différente, cette traduction est équivalente à la notion de raffinement développée dans CLAM et OYSTERS. Toutefois, NOMADIC PICT va plus loin en prouvant la correction de la traduction.

Le calcul des SAFE AMBIENTS, TYCO et le join–calcul disposent tous les trois de spécifications formelles de leurs implantations sous forme de machines abstraites. TYCO et le join–calcul suivent une approche orientée communication en utilisant les canaux du π–calcul asynchrone dans des objets concurrents ou dans des définitions. L'optique des SAFE AMBIENTS est axée vers la migration, en exprimant toute exécution en terme de primitives de mobilité d'ambients. Les travaux menés sur l'implantation du M–calcul sont plus proches de la première catégorie, la migration étant exprimée en terme de communication.

Le M–calcul constitue une extension directe du join–calcul en intégrant de nouvelles primitives comme les localités programmables, les processus d'ordre supérieur, et la liaison dynamique. La JAM a donc constitué une source d'inspiration importante lors du travail de conception de la machine abstraite CLAM. En comparant la description de la CLAM (cf. section 5.3.2) et celle de la JAM (cf. annexe A), on constate que l'état des deux machines abstraites est similaire. Chacune possède un nom, un générateur de nouveaux identifiants, des zones pour stocker les définitions et les messages en attente, et une file d'exécution. Les deux spécifications disposent également d'un moyen d'assurer l'acheminement des messages entre deux machines: table de routage pour la JAM, service de localisation pour la CLAM. Les deux machines disposent aussi d'évaluateurs fonctionnels séparés du reste de l'exécution. La CLAM semble aller plus loin en identifiant clairement l'interface de l'évaluateur et en autorisant de multiples stratégies d'évaluation des termes. Les différences essentielles reposent sur les primitives de migration utilisées: requêtes de migration (join–calcul), ou passivation et communication distante (M–calcul). Ces différences impliquent des stratégies distinctes de gestion des tables de routage: si les mises à jour sont immédiates dans la JAM, elles sont retardées dans la CLAM jusqu'au moment où le message parvient dans une localité-racine. La prise en compte de la liaison dynamique dans le M–calcul nécessite un service de localisation plus développé et une stratégie de routage plus complexe dans la CLAM, même si dans les deux algorithmes utilisés sont comparables dans leur principe. Par rapport à la JAM, la CLAM implémente des sémantiques programmables de domaines – en particulier celles du join–calcul – tout en conservant une complexité équivalente par rapport au calcul sous-jacent.

En conclusion, on voit que si les différents modèles de programmation diffèrent quant à la manière de partitionner une exécution répartie (modèle de domaine plat ou hiérarchique), dans l'implantation on retrouve toujours une séparation claire entre distribution physique (de niveau site/machine) et logique (de niveau localité/oyster). Une autre différence importante concerne le modèle d'exécution des localités qui peuvent être soit mono-threadées (PAN, CLAM) ou autoriser l'exécution de threads multiples (DITYCO, JAM).

7.5 Perspectives

La plate-forme OYSTERS ne permet d'exécuter que des programmes rudimentaires écrits en M–calcul. Pour qu'elle soit véritablement utilisable, il faudrait inclure dans le coeur de langage des types primitifs, quelques fonctions prédéfinies et implanter un vérificateur de types, par exemple pour le système défini dans [153]. L'ensemble ne pose pas de difficulté particulière et représente simplement du travail de développement en utilisant des techniques bien connues. Un étape supplémentaire mérite d'être franchie pour définir un véritable langage construit au-dessus du M–calcul, par exemple par hybridation avec un langage de haut niveau comme OCAML ou JAVA afin de déterminer les bons schémas de compilation pour le sucre syntaxique. Cela permettrait d'ouvrir la voie à une « programmation par domaines », au même titre que la programmation par aspects [85] ou par composants.

La conception modulaire de la machine virtuelle autorise une certaine flexibilité au niveau de la spécification et de l'implantation: stratégie d'évaluation des expressions fonctionnelles, politique de déclenchement des règles, ou isolation de services techniques nécessaires au fonctionnement de la plate-forme. Cette modularité est encore plus nécessaire en contexte réparti, afin d'interfacer la machine virtuelle avec différentes plates-formes de communication. Une première évolution de l'architecture de la machine virtuelle vers une structure générique à composants a été esquissée dans la section 6.3.3, un prolongement naturel consistant à appliquer les interfaces identifiées à la construction d'autres machines virtuelles pour calculs de processus manifestant une sémantique de domaine (canevas IMC). Toutefois, il ne s'agit que de travaux préliminaires qu'il serait intéressant d'étendre pour obtenir une machine virtuelle dont l'architecture est définie entièrement sous forme de composants, et implantée à l'aide d'un modèle de composants comme FRACTAL qui transpose la notion de domaine en terme de composants JAVA.

D'autre part, dans les services techniques d'OYSTERS, les modèles de désignation ou les algorithmes de localisation des ressources font partie du cœur de la machine virtuelle et n'apparaissent pas comme entités externes et indépendantes. Ainsi, dans le modèle de nommage utilisé, les contextes de désignation restent implicites, alors qu'ils pourraient être décrits par des domaines et se manifester à l'exécution par des oysters. Un prolongement prometteur consiste à étendre la machine virtuelle pour prendre en compte des dimensions non-fonctionnelles supplémentaires comme le nommage ou les transactions, possédant chacune leur propre sémantique de domaine: chaque dimension pourrait être spécifiée par un système de règles de réduction, dépendant du système de règles principal de la machine abstraite qui spécifie la dimension d'exécution ; chaque système de règles pourrait ensuite être implanté par un ou plusieurs composants, la machine virtuelle étant générée par « tissage » de ces différents composants à l'aide d'outils comme ASPECTJ [85]. Ce type d'expérimentation permettrait d'explorer les liens possibles entre formalisation d'un système par domaines et programmation par aspects ou par séparation des préoccupations (« separation of concerns »).

Enfin, on pourrait envisager de spécialiser la machine virtuelle OYSTERS pour étudier comment réaliser au niveau de l'infrastructure une ou plusieurs notions de domaine pour la sécurité, modélisées à l'aide d'extensions sécurisées du M–calcul. Une première possibilité consisterait à inclure des primitives cryptographiques comme en spi–calcul ou sjoin–calcul, en vue d'aider à la conception et à la mise en oeuvre de nouveaux protocoles d'authentification. Cette tâche relativement simple devrait être facilitée par l'extensibilité de la machine virtuelle qui autorise l'introduction de nouvelles règles de réduction décrivant les fonctionnalités ajoutées au modèle. Un travail plus ambitieux consisterait à contrôler les accès aux ressources en utilisant dans le calcul des domaines étendus faisant apparaître de manière explicite une politique de sécurité, et en spécifiant puis implantant dans la machine virtuelle la vérification de cohérence de cette politique et les mécanismes de contrôle des accès.

Conclusion

Dans cette thèse, nous avons montré qu'il était possible de surmonter le double obstacle lié à la construction d'infrastructures fiables pour la programmation répartie et mobile sur des réseaux à large échelle. Ce double écueil consiste en : d'une part, un manque de cadre de référence pour le nouveau domaine de recherche appelé « global computing », en matière d'abstractions, de modèles et de primitives de programmation ; et d'autre part, la divergence entre théorie et pratique de la programmation répartie. Les méthodes formelles sont encore insuffisamment intégrées au monde du génie logiciel, et de ce fait, la sémantique des infrastructures middleware actuellement déployées reste peu ou pas explicitée.

Nous avons exploré l'intérêt de la notion de domaine comme abstraction fondamentale possible pour le « global computing » et montré que ce concept est présent aussi bien dans la plupart des modèles de programmation répartie existants, que dans les machines virtuelles qui les implantent ou bien dans les plates-formes middleware les plus répandues. Cette notion constitue une avancée en terme de théorie de la programmation répartie car elle modélise au sein d'une même abstraction composition et contrôle. Elle permet de traiter de manière uniforme les différents types de partitions qui se présentent dans un système distribué de grande taille, identifiées comme autant de sémantiques différentes de domaine.

La notion de domaine est effective: un modèle de programmation fondé sur ce concept, comme le M–calcul, est directement implantable en pratique. Pour le vérifier, nous avons utilisé une méthodologie de génie logiciel pour la construction d'infrastructures dignes de confiance: la formalisation du système s'effectue par raffinements successifs, en définissant un modèle formel de programmation répartie à base de domaines, puis en spécifiant formellement une implantation du modèle à l'aide d'une machine abstraite. La dernière étape consiste à réaliser cette implantation sous forme de machine virtuelle répartie en utilisant les technologies middleware les plus répandues.

Après une expérimentation préliminaire visant à évaluer la faisabilité de cette approche pour implanter dans une plate-forme à objets répartis un modèle de programmation comprenant une forme simple de domaine, nous avons appliqué cette méthodologie à un véritable calcul de processus réparti, explicitement fondé sur la notion de domaine, le M–calcul : nous avons défini une machine abstraite répartie, pour ce calcul appelée CLAM. Nous avons implanté cette spécification formelle sous forme d'une machine virtuelle nommée OYSTERS, dans un cadre centralisé puis distribué, montrant ainsi l'effectivité du modèle de programmation initial. L'évaluation qualitative de la démarche montre la viabilité du concept de domaine, l'enrichissement du modèle de programmation choisi avec cette primitive ne se traduisant que par une augmentation de complexité raisonnable, comparable au niveau du modèle et de son implantation.

L'intérêt de cette démarche est de relier théorie et pratique de la programmation répartie à large échelle, par un continuum d'étapes de raffinement, depuis le modèle formel jusqu'aux infrastructures middleware existantes. Le point commun entre ces étapes est une nouvelle primitive de programmation pour le « global computing »: le domaine.

L'identification de cette abstraction est susceptible de constituer une avancée importante pour les fondements et la programmation du « global computing ». De même, la démarche de formalisation par raffinements successifs fournit un guide applicable pour la conception et la réalisation de systèmes critiques évoluant dans ce type d'environnement.

Toutefois, le fossé entre théorie et pratique de la programmation répartie reste encore largement ouvert. Ainsi, on constate que les avancées progressives en terme de primitives de programmation restent freinées par la difficulté des preuves correspondantes, les outils adaptés faisant encore défaut aujourd'hui. De même, le caractère systématique d'application de la démarche reste limité par la difficulté de prouver la correction des raffinements mis en jeu à chaque étape, ce qui ralentit les capacités à assurer la fiabilité des systèmes. L'importance et la difficulté de la question nécessitent une attention soutenue afin de développer la théorie et de construire les outils nécessaires pour effectuer les preuves de manière semi-automatisée ou automatisée, en vue d'une application de cette démarche dans un contexte industriel.

Cet enjeu important montre que beaucoup reste encore à faire pour rapprocher théoriciens et praticiens de la programmation répartie. Nous estimons que cette étude constitue un pas dans cette direction. De nombreux projets de recherche en cours vont également dans le sens de ce rapprochement. Ainsi, ce travail n'est pas isolé mais résulte d'une collaboration avec les participants des projets RNRT MARVEL [90] puis IST MIKADO [123]. Il s'inscrit dans le cadre de la Global Computing Initiative [70] dont le but est de poser les fondements théoriques du « global computing » et d'explorer les techniques facilitant le développement d'infrastructures de programmation dans ce contexte. Cette étude a d'ailleurs fait l'objet de délivrables dans le cadre de ces projets [16, 24, 29, 30, 66, 92] et de publications dans des colloques internationaux [67, 91].

Pour conclure, ce travail ouvre de nombreuses perspectives, comme l'étude des liens entre domaines et programmation par composants. Afin d'aller plus loin dans la fiabilité des infrastructures pour le « global computing », une piste prometteuse semble être d'approfondir la relation entre domaines et sécurité des systèmes. Par exemple, les domaines de ressources où les accès sont contrôlés en suivant une politique commune de sécurité se manifestent comme unités de confinement dans les systèmes d'exploitation. De même, les domaines de confiance où les membres se sont mutuellement authentifiés, et peuvent donc faire confiance aux autres membres du groupe, sont à la base de nombreuses infrastructures de gestion de la confiance. Les domaines semblent offrir un guide précieux pour aborder la sécurité des systèmes répartis sous l'angle architectural en fournissant à un architecte de sécurité: d'une part, une abstraction pour décomposer un système sécurisé en une superposition de différents types de domaines de sécurité lors de la phase de spécification; et d'autre part, des primitives de programmation pour composer de manière sûre les différents composants de sécurité correspondants lors de la réalisation de l'infrastructure répartie. A ce titre, les domaines devraient contribuer à la quête d'infrastructures réparties à large échelle toujours plus fiables.

Annexe A

La machine abstraite JAM

Cette annexe présente les principales définitions relatives à la machine abstraite JAM (JoCaml Abstract Machine). La JAM est une spécification formelle d'assez haut niveau des structures d'exécution de la machine virtuelle JoCaml, plate-forme permettant la programmation répartie à l'aide des primitives du join–calcul. Nous commençons par rappeler la syntaxe du langage d'entrée (cf. section A.1) avant de décrire l'état de la machine abstraite (cf. section A.2) et ses principales règles de réduction (cf. section A.3). Pour une présentation plus complète, le lecteur pourra utilement se référer à [138] dont sont extraites l'ensemble des figures de cette annexe.

A.1 Syntaxe du langage d'entrée

La figure A.1 résume les catégories syntaxiques nécessaires à la description de la JAM, ainsi que la syntaxe du langage exécuté par la machine abstraite. On y retrouve les principales abstractions du join–calcul comme les définitions, les filtres de messages ou les localités, et des constructions comme l'envoi asynchrone de message ou la demande de migration d'une localité. On notera que la machine permet l'exécution de multiples threads.

A.2 Etat de la machine abstraite

La figure A.2 donne l'état de la machine abstraite JAM. L'unité de répartition est la *machine* qui dispose d'un nom unique m, d'un ensemble de noms $\mathcal{N}$ nouvellement créés et de structures pour stocker les définitions actives ou les messages en attente de communication. La machine héberge un ensemble de localités dont chacune s'identifie avec un triplet $\langle \bar{d}, \bar{l}, \bar{\theta} \rangle$ désignant les définitions $\bar{d}$ qu'elle abrite, les noms $\bar{l}$ de ses sous-localités, et les threads $\bar{\theta}$ en cours d'exécution dans cette localité. Enfin, chaque thread de la machine s'identifie à une file contenant l'ensemble des opérations qu'il doit exécuter.

Les différentes machines communiquent par envoi de messages sur le réseau. Ce dernier est modélisé par un ensemble de files de communication contenant les différents messages en transit entre machines. Ainsi,

Catégories syntaxiques:

E	$\in \mathsf{Expr}$	Expression	d	$\in \mathsf{DefId} \subset \mathsf{Id}$	Nom de définition
v	$\in \mathsf{Value}$	Valeur	l	$\in \mathsf{LocId} \subset \mathsf{Id}$	Nom de localité
x	$\in \mathsf{Var}$	Variable	θ	$\in \mathsf{ThreadId} \subset \mathsf{Id}$	Nom de thread
$\mathcal{D}$	$\in \mathsf{Def}$	Définition	$p, q \ldots$	$\in \mathsf{PortId}$	Nom de port
$\langle \bar{d}, \bar{l}, \bar{\theta} \rangle$	$\in \mathsf{Loc}$	Localité	ρ	$\in \mathsf{Env} = \mathsf{Var} \mapsto \mathsf{Expr}$	Environnement
n	$\in \mathsf{Id}$	Nom	t	$\in \mathsf{Date}$	Estampille

Syntaxe du langage d'entrée:

E	$::=$		Expression
		v	Valeur
	$\mid$	x	Variable
	$\mid$	$\mathtt{def}\ \mathcal{D}\ \mathtt{in}\ E$	Nouvelle définition
	$\mid$	$\mathtt{loc}\ a\ \mathtt{def}\ \mathcal{D}\ \mathtt{do}\ E_{body}\ \mathtt{in}\ E_{cont}$	Nouvelle localité
	$\mid$	$\mathtt{exit}$	Fin de thread
	$\mid$	$\mathtt{spawn}\ E_{new};\ E_{cont}$	Composition parallèle
	$\mid$	$E_{chan}!(E_{arg});\ E_{cont}$	Envoi de message asynchrone
	$\mid$	$\mathtt{go}(E_{dest});\ E_{cont}$	Requête de migration
v	$::=$		Valeur
		$\langle d, p \rangle$	Canal (nom de définition et port)
	$\mid$	l	Nom de localité
$\mathcal{D}$	$::=$		Définition
		$\top$	Définition vide
	$\mid$	$J \triangleright E$	Join-pattern
	$\mid$	$\mathcal{D}_0 \wedge \mathcal{D}_1$	Définitions multiples
n	$::=$		Nom
		d	Nom de définition
	$\mid$	l	Nom de localité
	$\mid$	θ	Nom de thread
J	$::=$	$p(x) \mid q(y) \mid \ldots$	Join-pattern

FIG. *A.1* – **Catégories syntaxiques et langage d'entrée.**

Etat d'une machine JAM: $m : \mathcal{N} : U : D : H : L : R$

$m \in \mathsf{Machine}$	Nom de la machine
$\mathcal{N} \in \mathsf{Id}^*$	Noms utilisés par une machine
$U \in \mathsf{Id} \to \mathsf{Machine} \times \mathsf{Date}$	Table de routage
$D \in \mathsf{DefId} \to \mathsf{Env} \times \mathsf{Def}$	Table des définitions
$H \in \mathsf{DefId} \to \mathsf{PortId} \to \mathsf{Queue(Msg)}$	File de messages en attente
$L \in \mathsf{LocId} \to \mathsf{DefId}^* \times \mathsf{LocId}^* \times \mathsf{ThreadId}^*$	Table des localités
$R \in \mathsf{ThreadId} \to \mathsf{Env} \times \mathsf{Expr}$	Threads en cours d'exécution

FIG. *A.2* – **Etat de la machine abstraite JAM.**

une file de communication contenant les messages $\aleph_1, \aleph_2, \ldots$ envoyés de la machine m vers la machine m' sera notée $[m \to m' \vdash \aleph_1 :: \aleph_2 :: \ldots]$. Chaque machine contient aussi une table de routage U afin d'assurer l'acheminement des messages destinés à des machines distantes.

A.3 Règles de réduction

La figure A.3 résume les principales règles de réduction de la JAM concernant le calcul séquentiel et la communication locale. La règle VAR décrit l'évaluation d'une variable, en utilisant les contextes définis dans la figure A.4. La règle NIL termine le thread courant, tandis que la règle SPW exécute une composition parallèle en créant un nouveau thread. Les règles DEF, MSG_LSND, et TRG implantent le mécanisme de communication-synchronisation du join–calcul à l'intérieur d'une localité. Enfin, la règle LOC crée une nouvelle localité dans la machine courante.

VAR
$$\frac{v = \rho(x)}{\begin{array}{c} m : \mathcal{N} : U : D : H : L : R \oplus \{\theta \mapsto \langle \rho, \mathbf{C}[x] \rangle\} \longmapsto \\ m : \mathcal{N} : U : D : H : L : R \oplus \{\theta \mapsto \langle \rho, \mathbf{C}[v] \rangle\} \end{array}}$$

NIL
$$\frac{\theta \in \bar{\theta}}{\begin{array}{c} m : \mathcal{N} : U : D : H : L\left\{l \mapsto \left\langle \bar{d}, \bar{l}, \bar{\theta} \oplus \{\theta\} \right\rangle\right\} : R \oplus \{\theta \mapsto \langle \rho, \mathbf{C}[\texttt{exit}] \rangle\} \longmapsto \\ m : \mathcal{N} : U : D : H : L\left\{l \mapsto \left\langle \bar{d}, \bar{l}, \bar{\theta} \right\rangle\right\} : R \end{array}}$$

SPW
$$\frac{\theta' \notin \mathcal{N} \quad \theta \in \bar{\theta} \quad R(\theta) = \langle \rho, \mathbf{C}[\texttt{spawn}\ E';\ E] \rangle \quad \rho' = \rho \restriction fv(E')}{\begin{array}{c} m : \mathcal{N} : U : D : H : L\left\{l \mapsto \left\langle \bar{d}, \bar{l}, \bar{\theta} \right\rangle\right\} : R \longmapsto \\ m : \mathcal{N} \oplus \{\theta'\} : U\{\theta' \mapsto \langle m, 0 \rangle\} : D : H : L\left\{l \mapsto \left\langle \bar{d}, \bar{l}, \bar{\theta} \cup \{\theta'\} \right\rangle\right\} : \\ R\{\theta \mapsto \langle \rho, \mathbf{C}[E] \rangle, \theta' \mapsto \langle \rho', E' \rangle\} \end{array}}$$

DEF
$$\frac{\begin{array}{c} d \notin \mathcal{N} \quad \theta \in \bar{\theta} \quad \rho' = \rho\{\forall p \in dn(\mathcal{D}), p \mapsto \langle d, p \rangle\} \\ \rho'' = \rho' \restriction \bigcup_{J_i \triangleright E_i \in \mathcal{D}} (fv(E_i) \setminus rn(J_i)) \end{array}}{\begin{array}{c} m : \mathcal{N} : U : D : H : L\left\{l \mapsto \left\langle \bar{d}, \bar{l}, \bar{\theta} \right\rangle\right\} : R\{\theta \mapsto \langle \rho, \mathbf{C}[\texttt{def}\ \mathcal{D}\ \texttt{in}\ E] \rangle\} \longmapsto \\ m : \mathcal{N} \oplus \{d\} : U\{d \mapsto \langle m, 0 \rangle\} : D\{d \mapsto \langle \rho'', \mathcal{D} \rangle\} : \\ H\{d \mapsto \{\forall p \in dn(\mathcal{D}), p \mapsto []\}\} : L\left\{l \mapsto \left\langle \bar{d} \cup \{d\}, \bar{l}, \bar{\theta} \right\rangle\right\} : R\{\theta \mapsto \langle \rho', \mathbf{C}[E] \rangle\} \end{array}}$$

MSG_LSND
$$\frac{U(d) = \langle m, t \rangle}{\begin{array}{c} m : \mathcal{N} : U : D : H : L : R\{\theta \mapsto \langle \rho, \mathbf{C}[\langle d, p \rangle!(v);\ E] \rangle\} \longmapsto \\ m : \mathcal{N} : U : D : H\{d \mapsto p \mapsto H(d)(p) :: v\} : L : R\{\theta \mapsto \langle \rho, \mathbf{C}[E] \rangle\} \end{array}}$$

TRG
$$\frac{\begin{array}{c} \theta \notin \mathcal{N} \quad d \in \bar{d} \quad D(d) = \langle \rho, J \triangleright E \wedge \ldots \rangle \quad J = p(x_p) \mid \ldots \\ \forall p \in dn(J), H(d)(p) = v_p :: \bar{v}_p \quad \rho' = \rho\{\forall p \in dn(J), x_p \mapsto v_p\} \end{array}}{\begin{array}{c} m : \mathcal{N} : U : D : H : L\left\{l \mapsto \left\langle \bar{d}, \bar{l}, \bar{\theta} \right\rangle\right\} : R \longmapsto \\ m : \mathcal{N} \cup \{\theta\} : U\{\theta \mapsto \langle m, 0 \rangle\} : D : H\{d \mapsto H(d)\{\forall p \in dn(J), p \mapsto \bar{v}_p\}\} : \\ L\left\{l \mapsto \left\langle \bar{d}, \bar{l}, \bar{\theta} \cup \{\theta\} \right\rangle\right\} : R\{\theta \mapsto \langle \rho', E \rangle\} \end{array}}$$

LOC
$$\frac{\begin{array}{c} d' \notin \mathcal{N} \quad l' \notin \mathcal{N} \quad \theta' \notin \mathcal{N} \quad \theta \in \bar{\theta} \\ \rho' = \rho\{a \mapsto l'\}\{\forall p \in dn(\mathcal{D}), p \mapsto \langle d, p \rangle\} \quad \rho'' = \rho' \restriction \bigcup_{J_i \triangleright E_i \in \mathcal{D}} (fv(E_i) \setminus rn(J_i)) \end{array}}{\begin{array}{c} m : \mathcal{N} : U : D : H : L\left\{l \mapsto \left\langle \bar{d}, \bar{l}, \bar{\theta} \right\rangle\right\} : R\{\theta \mapsto \langle \rho, \mathbf{C}[\texttt{loc}\ a\ \texttt{def}\ \mathcal{D}\ \texttt{do}\ E'\ \texttt{in}\ E] \rangle\} \longmapsto \\ m : \mathcal{N} \oplus \{d', l', \theta'\} : U\{d' \mapsto \langle m, 0 \rangle, l' \mapsto \langle m, 0 \rangle, \theta' \mapsto \langle m, 0 \rangle\} : \\ D\{d' \mapsto \langle \rho'', \mathcal{D} \rangle\} : H\{d' \mapsto \{\forall p \in dn(\mathcal{D}), p \mapsto []\}\} : \\ L\left\{l \mapsto \left\langle \bar{d}, \bar{l} \cup \{l'\}, \bar{\theta} \right\rangle, l' \mapsto \langle \{d'\}, \varnothing, \{\theta'\} \rangle\right\} : R\{\theta \mapsto \langle \rho', \mathbf{C}[E] \rangle, \theta' \mapsto \langle \rho' \restriction fv(E'), E' \rangle\} \end{array}}$$

FIG. *A.3* – **Règles de réduction : cœur séquentiel, localités et primitives de communication locale.**

Les règles de la figure A.5 décrivent la communication entre deux machines distantes: envoi et réception de messages (règles MSG_RSND et MSG_RCV) ou relai de messages vers une machine tierce (règle MSG_FWD). Enfin, les règles de la figure A.6 décrivent la migration des localités, soit dans le contexte de la même machine (règle GO_LSND), soit entre deux machines distinctes (règles GO_RSND, GO_RCV et GO_FWD). L'ensemble de ces règles s'appuie fortement sur les tables de routage des différentes machines.

$$\mathbf{C} \quad ::= \quad \cdot \quad | \quad E_0!(\mathbf{C});\ E_2 \quad | \quad \mathbf{C}!(v);\ E_2 \quad | \quad \mathtt{go}(\mathbf{C});\ E_1$$

FIG. *A.4* – **Contextes d'évaluation JAM.**

MSG_RSND

$$\frac{U(d) = \langle m', t\rangle \quad m \neq m' \quad \langle \hat{v}, \bar{n}\rangle = pack(v) \quad \hat{U} = \{\forall n \in \bar{n}, n \mapsto U(n)\}}{\begin{array}{c} m : \mathcal{N} : U : D : H : L : R\{\theta \mapsto \langle \rho, \mathbf{C}[\langle d, p\rangle!(v);\ E]\rangle\} \ ; \ [m \to m' \vdash \bar{\aleph}] \longmapsto \\ m : \mathcal{N} : U : D : H : L : R\{\theta \mapsto \langle \rho, \mathbf{C}[E]\rangle\} \ ; \ \left[m \to m' \vdash \bar{\aleph} :: \langle d, p\rangle!(\hat{v}, \hat{U})\right] \end{array}}$$

MSG_RCV

$$\frac{U(d) = \langle m, t\rangle \quad v = unpack(\hat{v}) \quad U' = merge(U, \hat{U})}{\begin{array}{c} m : \mathcal{N} : U : D : H : L : R ; \left[{}^{\backprime}m \to m \vdash \langle d, p\rangle!(\hat{v}, \hat{U}) :: \bar{\aleph}\right] \longmapsto \\ m : \mathcal{N} : U' : D : H\{d \mapsto p \mapsto H(d)(p) :: v\} : L : R ; [{}^{\backprime}m \to m \vdash \bar{\aleph}] \end{array}}$$

MSG_FWD

$$\frac{U(d) = \langle m', t\rangle \quad m' \neq m \quad U' = merge(U, \hat{U}) \quad \hat{U}' = \{\forall n \in dom\left(\hat{U}\right), n \mapsto U'(n)\}}{\begin{array}{c} m : \mathcal{N} : U : D : H : L : R ; \left[{}^{\backprime}m \to m \vdash \langle d, p\rangle!(\hat{v}, \hat{U}) :: \bar{\aleph}\right] \ ; \ [m \to m' \vdash \bar{\aleph}'] \longmapsto \\ m : \mathcal{N} : U' : D : H : L : R ; [{}^{\backprime}m \to m \vdash \bar{\aleph}] \ ; \ \left[m \to m' \vdash \bar{\aleph}' :: \langle d, p\rangle!(\hat{v}, \hat{U}')\right] \end{array}}$$

FIG. *A.5* – **Règles de réduction : communication distante.**

GO_LSND

$$\frac{U(l_d) = \langle m, t\rangle \quad l_m \in \bar{l}_p \quad \theta \in \bar{\theta}_m \quad l_d \notin descendants(l_m)}{\begin{array}{c} m : \mathcal{N} : U : D : H : L\{l_p \mapsto \langle \bar{d}_p, \bar{l}_p, \bar{\theta}_p\rangle, l_d \mapsto \langle \bar{d}_d, \bar{l}_d, \bar{\theta}_d\rangle\} : R\{\theta \mapsto \langle \rho, \mathbf{C}[\mathtt{go}(l_d);\ E]\rangle\} \longmapsto \\ m : \mathcal{N} : U : D : H : L\{l_p \mapsto \langle \bar{d}_p, \bar{l}_p \setminus \{l_m\}, \bar{\theta}_p\rangle, l_d \mapsto \langle \bar{d}_d, \bar{l}_d \cup \{l_m\}, \bar{\theta}_d\rangle\} : R\{\theta \mapsto \langle \rho, \mathbf{C}[E]\rangle\} \end{array}}$$

GO_RSND

$$\frac{\begin{array}{c} U(l_d) = \langle m', t\rangle \quad l_m \in \bar{l}_p \quad \theta \in \bar{\theta}_m \quad m' \neq m \\ \langle w, D', H', L', R'\rangle = extract(l_m, D, H, L, R\{\theta \mapsto \langle \rho, \mathbf{C}[E]\rangle\}) \quad \langle \hat{w}, \hat{\bar{n}}\rangle = pack_{loc}(w) \\ \hat{U} = \{\forall n \in \hat{\bar{n}}, n \mapsto U(n)\} \quad U' = U\{\forall n \in \bar{n}, n \mapsto \langle m', t+1\rangle \text{ tel que } U(n) = \langle m, t\rangle\} \\ N = \{\forall n \in \bar{n}, n \mapsto t' \text{ tel que } U'(n) = \langle m', t'\rangle\} \end{array}}{\begin{array}{c} m : \mathcal{N} : U : D : H : L\{l_p \mapsto \langle \bar{d}_p, \bar{l}_p, \bar{\theta}_p\rangle\} : R\{\theta \mapsto \langle \rho, \mathbf{C}[\mathtt{go}(l_d);\ E]\rangle\} \ ; \ [m \to m' \vdash \bar{\aleph}] \longmapsto \\ m : \mathcal{N} : U' : D' : H' : L'\{l_p \mapsto \langle \bar{d}_p, \bar{l}_p \setminus \{l_m\}, \bar{\theta}_p\rangle\} : R' \ ; \ \left[m \to m' \vdash \bar{\aleph} :: go(l_m, l_d, \hat{w}, \hat{U}, N)\right] \end{array}}$$

GO_RCV

$$\frac{\begin{array}{c} U(l_d) = \langle m, t\rangle \quad w = unpack_{loc}(\hat{w}) \quad \langle D', H', L', R'\rangle = insert(w, D, H, L, R) \\ U' = (merge(U, \hat{U}))\{\forall n \in dom(N), n \mapsto \langle m, N(n)\rangle\} \end{array}}{\begin{array}{c} m : \mathcal{N} : U : D : H : L : R ; \left[{}^{\backprime}m \to m \vdash go(l_m, l_d, \hat{w}, \hat{U}, N) :: \bar{\aleph}\right] \longmapsto \\ m : \mathcal{N} : U' : D' : H' : L'\{l_d \mapsto \langle \bar{d}, \bar{l} \cup \{l_m\}, \bar{\theta}\rangle\} : R' ; [{}^{\backprime}m \to m \vdash \bar{\aleph}] \end{array}}$$

GO_FWD

$$\frac{\begin{array}{c} U(l_d) = \langle m', t\rangle \quad m' \neq m \\ U' = merge(U, \hat{U}) \quad \hat{U}' = \{\forall n \in dom\left(\hat{U}\right), n \mapsto U'(n)\} \quad U'' = U'\{\forall n \in dom(N), n \mapsto \langle m', N(n)+1\rangle\} \end{array}}{\begin{array}{c} m : \mathcal{N} : U : D : H : L : R ; \left[{}^{\backprime}m \to m \vdash go(l_m, l_d, \hat{w}, \hat{U}, N) :: \bar{\aleph}\right] \ ; \ [m \to m' \vdash \bar{\aleph}'] \longmapsto \\ m : \mathcal{N} : U'' : D : H : L : R ; [{}^{\backprime}m \to m \vdash \bar{\aleph}] \ ; \ \left[m \to m' \vdash \bar{\aleph}' :: go(l_m, l_d, \hat{w}', \hat{U}', N')\right] \end{array}}$$

FIG. *A.6* – **Règles de réduction : mobilité.**

Bibliographie

[1] MARTÍN ABADI et LUCA CARDELLI. « *A Theory of Objects* ». Springer-Verlag, 1996.

[2] MARTÍN ABADI, CÉDRIC FOURNET, et GEORGES GONTHIER. « Secure Implementation of Channel Abstractions ». Dans *Proc. 13th IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 105–116, 1998.

[3] MARTÍN ABADI, CÉDRIC FOURNET, et GEORGES GONTHIER. « Authentication Primitives and their Compilation ». Dans *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 302–315. ACM Press, 2000.

[4] MARTÍN ABADI et ANDREW GORDON. « A Calculus for Cryptographic Protocols : The Spi Calculus ». *Information and Computation*, 148(1):1–70, 1999. Paru également dans *Proc. CCS'97.*

[5] GHUL AGHA. « *ACTORS: A Model of Concurrent Computation in Distributed Systems* ». MIT Press, 1986.

[6] GHUL AGHA et CARL HEWITT. « Actors: a Conceptual Foundation for Concurrent Object-Oriented Programming ». Dans *Research Directions on Object-Oriented Programming*, pages 49–74. 1987.

[7] ROBERT ALLEN et DAVID GARLAN. « A Formal Basis for Architectural Connection ». *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[8] ROBERTO AMADIO. « An Asynchronous Model of Locality, Failure and Process Mobility ». Dans *Proc. Second International Conference on Coordination Models and Languages (COORDINATION'97)*, *Lecture Notes in Computer Science* 1282, pages 374–391. Springer Verlag, 1997. Une version étendue est parue dans le rapport de recherche INRIA RR-3109, INRIA Sophia-Antipolis, 1997.

[9] ROBERTO AMADIO. « On Modelling Mobility ». *Theoretical Computer Science*, 240:147–176, 2000.

[10] ROBERTO AMADIO, GÉRARD BOUDOL, et CÉDRIC LHOUSSAINE. « The Receptive Distributed Pi-Calculus ». *ACM Transactions on Programming Languages and Systems*, 25(5):1–29, 2003.

[11] ZENA ARIOLA, MATTHIAS FELLEISEN, JOHN MARAIST, MARTIN ODERSKY, et PHILIP WADLER. « A Call-By-Need Lambda Calculus ». Dans *Proc. 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 233–246, San Francisco, California (USA), 1995. ACM Press.

[12] JEAN-PIERRE BANÂTRE et DANIEL LE MÉTAYER. « The Gamma Model and its Discipline of Programming ». *Science of Computer Programming*, 15(1):55–77, 1990.

[13] HENDRIK BARENDREGT. « *The Lambda Calculus : Its Syntax and Semantics* ». North-Holland, 1984.

[14] GÉRARD BERRY et GÉRARD BOUDOL. « The Chemical Abstract Machine ». *Theoretical Computer Science*, 96(1):217–248, 1992. Paru également dans *Proc. POPL'90*.

[15] LORENZO BETTINI, ROCCO DE NICOLA, GIANLUIGI FERRARI, et ROSARIO PUGLIESE. « Interactive Mobile Agents in X-Klaim ». Dans *Proc. 7th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'98)*, pages 110–115. IEEE Computer Society Press, 1998.

[16] LORENZO BETTINI, ROCCO DE NICOLA, MARC LACOSTE, LUÍS LOPES, et VASCO VASCONCELOS. « Virtual Machine Technologies: Core Software Framework v0 – towards MIKADO Abstract Machines ». Rapport technique D3.1.0, IST MIKADO Project, 2003.

[17] LORENZO BETTINI, ROCCO DE NICOLA, et ROSARIO PUGLIESE. « X-Klaim and Klava: Programming Mobile Code ». Dans *Proc. Final Workshop of TOSCA Project*, *Electronic Notes in Theoretical Computer Science* 62. Elsevier Science Publishers, 2001.

[18] LORENZO BETTINI, ROCCO DE NICOLA, et ROSARIO PUGLIESE. « Klava: a Java Package for Distributed and Mobile Applications ». *Software - Practice and Experience*, 32(14):1365–1394, 2002.

[19] LORENZO BETTINI, MICHELE LORETI, et ROSARIO PUGLIESE. « An Infrastructure Language for Open Nets ». Dans *Proc. ACM Symposium on Applied Computing (SAC'02), Special Track on Coordination Models, Languages and Applications*, pages 373–377. ACM Press, 2002.

[20] ANDREW BIRELL, GREG NELSON, SUSAN OWICKI, et EWARD WOBBER. « Network Objects ». *Software-Practice and Experience*, 25(S4):87–130, 1995. Paru également dans *Proc. SOSP'93*.

[21] GORDON BLAIR, FABIO COSTA, GEOFF COULSON, FABIEN DELPIANO, HECTOR DURAN, BRUNO DUMANT, FRANÇOIS HORN, NIKOS PARLAVANTZAS, et JEAN-BERNARD STEFANI. « The Design of a Resource-Aware Reflective Middleware Architecture ». Dans *Proc. Second International Conference on Reflection and Meta-Level Architectures (REFLECTION'99)*, *Lecture Notes in Computer Science* 1616, pages 115–134, Saint-Malo, France, 1999. Springer-Verlag.

[22] GORDON BLAIR, GEOFF COULSON, ANDERS ANDERSEN, LYNNE BLAIR, MICHAEL CLARKE, FABIO COSTA, H. DURAN-LIMON, TOM FITZPATRICK, LEE JONHNSTON, RUI MOREIRA, NIKOS PARLAVANTZAS, et KATIA SAIKOSKI. « The Design and Implementation of Open ORB 2 ». *IEEE Distributed Systems Online Journal, Special Issue on Reflective Middleware*, 2(6), 2001.

[23] GORDON BLAIR et JEAN-BERNARD STEFANI. « *Open Distributed Processing and Multimedia* ». Addison-Wesley, 1997.

[24] MICHELE BOREALE, ROCCO DE NICOLA, MARC LACOSTE, et VASCO VASCONCELOS. « Analysis of Distribution Structures: State of the art ». Rapport technique D3.1.1, IST MIKADO Project, 2002.

[25] MICHELE BOREALE, ROCCO DE NICOLA, et ROSARIO PUGLIESE. « Programming Access Control: The Klaim Experience ». Dans *Proc. 11th International Conference on Concurrency Theory (CONCUR'00)*, *Lecture Notes in Computer Science* 1877, pages 48–65. Springer-Verlag, 2000.

[26] GÉRARD BOUDOL. « Asynchrony and the π-Calculus ». Rapport technique INRIA RR-1702, INRIA Sophia-Antipolis, 1992.

[27] GÉRARD BOUDOL. « The π-Calculus in Direct Style ». *Higher-Order and Symbolic Computation*, 11(2):177–208, 1998. Paru également dans *Proc. POPL'97*.

[28] GÉRARD BOUDOL. « Core Programming Model ». Rapport technique D1.2.0, IST MIKADO Project, 2002.

[29] GÉRARD BOUDOL, ILARIA CASTELLANI, FLORENCE GERMAIN, et MARC LACOSTE. « Models of Distribution and Mobility: State of the art ». Rapport technique D1.1.1, IST MIKADO Project, 2002.

[30] GÉRARD BOUDOL, FLORENCE GERMAIN, et MARC LACOSTE. « Analyse des langages et modèles de la mobilité ». Rapport technique INRIA RR-3930, INRIA Sophia-Antipolis, 2000.

[31] GILAD BRACHA et SAM TOUEG. « Asynchronous Consensus and Broadcast Protocols ». *Journal of the ACM*, 32(4):824–840, 1985.

[32] WILFRIED BRAUER, éditeur. « *Net Theory and Applications: Proc. Advanced Course on General Net Theory of Processes and Systems* », *Lecture Notes in Computer Science* 84, Hamburg, Germany, 1980. Springer-Verlag.

[33] ERIC BRUNETON, THIERRY COUPAYE, et JEAN-BERNARD STEFANI. « Recursive and Dynamic Software Composition with Sharing ». Dans *Proc. 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Málaga, Spain, 2002.

[34] MICHELE BUGLIESI, GIUSEPPE CASTAGNA, et SILVIA CRAFA. « Boxed Ambients ». Dans *Proc. Fourth International Conference on Theoretical Aspects of Computer Software (TACS'01)*, *Lecture Notes in Computer Science* 2215, pages 38–63. Springer-Verlag, 2001.

[35] LUCA CARDELLI. « Obliq: A Language with Distributed Scope ». *Computing Systems*, 8(1):27–59, 1995. Paru également dans *Proc. POPL'95*.

[36] LUCA CARDELLI. « Ambit », 1997. Software and documentation available from http://www.luca.demon.co.uk/Ambit/ambit.html.

[37] LUCA CARDELLI. « Abstractions for Mobile Computation ». Dans JAN VITEK et CHRISTIAN JENSEN, éditeurs, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, *Lecture Notes in Computer Science* 1603, pages 51–94. Springer-Verlag, 1999.

[38] LUCA CARDELLI. « Mobility and Security ». Lecture Notes for the Marktoberdorf Summer School, 1999.

[39] LUCA CARDELLI. « Wide Area Computation ». Dans *Proc. 26th International Colloquium on Automata, Languages and Programming (ICALP'99)*, *Lecture Notes in Computer Science* 1644, pages 10–24. Springer-Verlag, 1999.

[40] LUCA CARDELLI et ANDREW GORDON. « Mobile Ambients ». *Theoretical Computer Science*, 240(1):177–213, 2000. Paru également dans *Proc. FoSSaCS'98*.

[41] ILARIA CASTELLANI. « Process Algebras with Localities ». Dans JAN BERGSTRA, ALBAN PONSE, et SCOTT SMOLKA, éditeurs, *Handbook of Process Algebra*, pages 945–1045. Elsevier, 2001.

[42] TUSHAR CHANDRA et SAM TOUEG. « Unreliable Failure Detectors for Reliable Distributed Systems ». *Journal of the ACM*, 43(2):225–267, 1996. Paru également dans *Proc. PODC'91*.

[43] COMMON CRITERIA IMPLEMENTATION BOARD. « Common Criteria for Information Technology Security Evaluation (Parts 1-3) ». Rapport technique CCMIB-99-031/033, National Institute of Standards and Technology, 1999.

[44] SYLVAIN CONCHON et FABRICE LE FESSANT. « JoCaml: Mobile Agents for Objective-Caml ». Dans *Proc. Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA'99)*, pages 22–29, Palm Springs, California (USA), 1999. IEEE Computer Society.

[45] FABIO COSTA, HECTOR DURAN, NIKOS PARLAVANTZAS, KATIA SAIKOSKI, GORDON BLAIR, et GEOFF COULSON. « The Role of Reflective Middleware in Supporting the Engineering of Complex

Applications ». Dans *Reflection and Software Engineering*, *Lecture Notes in Computer Science* 1826, pages 79–98. Springer-Verlag, 2000.

[46] GIANPAOLO CUGOLA, CARLO GHEZZI, GIAN-PETRO PICCO, et GIOVANNI VIGNA. « Analyzing Mobile Code Languages ». Dans *Proc. Second International Workshop on Mobile Object Systems (MOS'96) : Towards the Programmable Internet*, *Lecture Notes in Computer Science* 1222, pages 93–110, Linz, Austria, 1997. Springer-Verlag.

[47] D'AGENTS PROJECT. Center for Mobile Computing, Dartmouth College. Voir http://agent.cs.dartmouth.edu/.

[48] SILVANO DAL-ZILLIO. « *Le calcul bleu : types et objets* ». PhD thesis, Université de Nice–Sophia Antipolis, 1999.

[49] ROCCO DE NICOLA, GIAN-LUIGI FERRARI, et ROSARIO PUGLIESE. « Locality-based Linda: Programming with Explicit Localities ». Dans *Proc. Seventh International Conference on the Theory and Practice of Software Development (TAPSOFT'97)*, *Lecture Notes in Computer Science* 1214, pages 712–726. Springer-Verlag, 1997.

[50] ROCCO DE NICOLA, GIAN-LUIGI FERRARI, et ROSARIO PUGLIESE. « Klaim: a Kernel Language for Agents Interaction and Mobility ». *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[51] ROCCO DE NICOLA, GIAN-LUIGI FERRARI, ROSARIO PUGLIESE, et B. VENNERI. « Types for Access Control ». *Theoretical Computer Science*, 240(1):215–254, 2000.

[52] RICHARD DEMILLO, RICHARD LIPTON, et ALAN PERLIS. « Social Processes and Proofs of Theorems and Programs ». Dans *Proc. 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 206–214, Los Angeles, California (USA), 1977. ACM Press.

[53] DOMINIC DUGGAN. « Atomic Failure in Wide-Area Computation ». Dans *Proc. Fourth IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'00)*, Stanford, California (USA), 2000. Kluwer.

[54] DOMINIC DUGGAN. « Abstractions for Fault-Tolerant Global Computing ». Dans *Proc. EATCS Workshop on Foundations of Wide Area Network Computing (F-WAN'02)*, *Electronic Notes in Theoretical Computer Science* 66(3), Málaga, Spain, 2002. Elsevier Science Publishers.

[55] BRUNO DUMANT, FRANÇOIS HORN, FRÉDÉRIC DANG TRAN, et JEAN-BERNARD STEFANI. « Jonathan: an Open Distributed Processing Environment in Java ». *Distributed Systems Engineering*, 6(1):3–12, 1999. Paru également dans *Proc. MIDDLEWARE'98*.

[56] JAMES FETZER. « Program Verification: the Very Idea ». *Communications of the ACM*, 31(9):1048–1063, 1988.

[57] MICHAEL FISCHER, NANCY LYNCH, et MIKE PATERSON. « Impossibility of Distributed Consensus with One Faulty Process ». *Journal of the ACM*, 32(2):374–382, 1985.

[58] CÉDRIC FOURNET. « *The Join-Calculus: a Calculus for Distributed Mobile Programming* ». PhD thesis, Ecole Polytechnique, 1998.

[59] CÉDRIC FOURNET et GEORGES GONTHIER. « The Reflexive Chemical Abstract Machine and the Join-Calculus ». Dans *Proc. 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385, St. Petersburg Beach, Florida (USA), 1996. ACM Press.

[60] CÉDRIC FOURNET, GEORGES GONTHIER, JEAN-JACQUES LÉVY, LUC MARANGET, et DIDIER RÉMY. « A Calculus of Mobile Agents ». Dans *Proc. 7th International Conference on Concurrency*

Theory (CONCUR'96), *Lecture Notes in Computer Science* 1119, pages 406–421, Pisa, Italy, 1996. Springer-Verlag.

[61] CÉDRIC FOURNET, JEAN-JACQUES LÉVY, et ALAN SCHMITT. « An Asynchronous Distributed Implementation of Mobile Ambients ». Dans *Proc. International IFIP Conference (TCS'00)*, *Lecture Notes in Computer Science* 1872, Sendai, Japan, 2000. Springer-Verlag.

[62] CÉDRIC FOURNET et LUC MARANGET. « The Join-Calculus Language », 1997. Software and documentation available from http://join.inria.fr/.

[63] NISSIM FRANCEZ et IRA FORMAN. « *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming* ». Addison-Wesley, 1996.

[64] ALFONSO FUGGETTA, GIAN-PIETRO PICCO, et GIOVANNI VIGNA. « Understanding Code Mobility ». *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[65] DAVID GELERNTER. « Generative Communication in Linda ». *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[66] FLORENCE GERMAIN et MARC LACOSTE. « Spécification d'une machine virtuelle répartie (version 2) – Volume 1: κ-VM: la machine virtuelle κ ». Rapport technique D3.3, Projet RNRT MARVEL, 2002.

[67] FLORENCE GERMAIN, MARC LACOSTE, et JEAN-BERNARD STEFANI. « An Abstract Machine for a Higher-Order Distributed Process Calculus ». Dans *Proc. EATCS Workshop on Foundations of Wide Area Network Computing (F-WAN'02)*, *Electronic Notes in Theoretical Computer Science* 66(3), Málaga, Spain, 2002. Elsevier Science Publishers.

[68] FLORENCE GERMAIN, ELIE NAJM, et JEAN-BERNARD STEFANI. « Elements of an Object-Based Model for Distributed and Mobile Computation ». Dans *Proc. Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'00)*, Palo Alto, California (USA), 2000. Kluwer.

[69] ROBERT GLASS. « The Proof of Correctness Wars ». *Communications of the ACM*, 45(8):19–21, 2002.

[70] GLOBAL COMPUTING INITIATIVE. Voir http://www.cordis.lu/ist/fetgc.htm.

[71] LI GONG. « *Inside Java 2 Platform Security: Architecture, API Design, and Implementation – 2nd Edition* ». Addison-Wesley, 2003.

[72] JAMES GOSLING, WILLIAM JOY, et GUY STEELE. « *The Java Language Specification* ». Addison-Wesley, 1996.

[73] ROBERT GRAY. « *Agent Tcl: A Flexible and Secure Mobile-Agent System* ». PhD thesis, Dartmouth College, 1997.

[74] PIETER HARTEL et LUC MOREAU. « Formalizing the Safety of Java, the Java Virtual Machine and Java Card ». *ACM Computing Surveys*, 33(4):517–558, 2001.

[75] RICHARD HAYTON et ANDREW HERBERT. « FlexiNet: A Flexible, Component-Oriented Middleware System ». Dans SACHA KRAKOWIAK et SANTOSH SHRIVASTAVA, éditeurs, *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, *Lecture Notes in Computer Science* 1752. Springer-Verlag, 1999.

[76] MATTHEW HENNESSY, JULIAN RATHKE, et ELIO GIOVANNETTI. « Type Systems and Static Analysis for Mobile and Distributed Computing: State of the Art ». Rapport technique D2.1.0, IST MIKADO Project, 2002.

[77] MATTHEW HENNESSY et JAMES RIELY. « Resource Access Control in Systems of Mobile Agents ». Rapport technique 2/98, School of Cognitive and Computer Sciences, University of Sussex, 1998.

[78] C.A.R. HOARE. « *Communicating Sequential Processes* ». Prentice-Hall International, 1985.

[79] KOHEI HONDA et MARIO TOKORO. « An Object Calculus for Asynchronous Communication ». Dans *Proc. 5th European Conference on Object-Oriented Programming (ECOOP'91)*, *Lecture Notes in Computer Science* 512, pages 133–147, Geneva, Switzerland, 1991. Springer Verlag.

[80] NORMAN HUTCHINSON et LARRY PETERSON. « The x-Kernel: An Architecture for Implementing Network Protocols ». *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[81] INTERNATIONAL STANDARDS ORGANIZATION. « ODP Reference Model: Architecture ». Rapport technique ISO/IEC JTC1/SC21/WG7 10746-3, 1995.

[82] INTERNATIONAL STANDARDS ORGANIZATION. « ODP Reference Model: Foundations ». Rapport technique ISO/IEC JTC1/SC21/WG7 10746-2, 1995.

[83] INTERNATIONAL STANDARDS ORGANIZATION. « ODP Reference Model: Overview ». Rapport technique ISO/IEC JTC1/SC21/WG7 10746-1, 1995.

[84] GREGOR KICZALES, JIM DES RIVIÈRES, et DANIEL BOBROW. « *The Art of the Metaobject Protocol* ». MIT Press, 1991.

[85] GREGOR KICZALES, JOHN LAMPING, ANURAG MENDHEKAR, CHRIS MAEDA, CRISTINA VIDEIRA-LOPES, JEAN-MARC LOINGTIER, et JOHN IRWIN. « Aspect-Oriented Programming ». Dans *Proc. 11th European Conference on Object-Oriented Programming (ECOOP'97)*, *Lecture Notes in Computer Science* 1241, pages 220–242, Jyväskylä, Finland, 1997. Springer Verlag.

[86] ERIK KLINTSKOG, ZACHARIAS EL BANNA, et PER BRAND. « A Generic Middleware for Intra-Language Transparent Distribution ». Rapport technique SICS-T-2003/01-SE, Swedish Institue of Computer Science, 2003.

[87] DAVID KOTZ, ROBERT GRAY, SAURAB NOG, DANIELA RUS, SUMIT CHAWLA, et GEORGE CYBENKO. « Agent Tcl: Targeting the Needs of Mobile Computers ». *IEEE Internet Computing*, 1(4):58–67, 1997.

[88] SACHA KRAKOWIAK. « *Principes des systèmes d'exploitation des ordinateurs* ». Dunod Informatique, 1985.

[89] SACHA KRAKOWIAK. « *The Jonathan Tutorial: Overview, Binding, Communication, Configuration and Resource Frameworks* ». ObjectWeb Consortium, 2002.

[90] MARC LACOSTE. « Introducing the French RNRT MARVEL Project ». Dans *Proc. Fifth International ECOOP Workshop on Mobile Object Systems : Programming Languages for Wide Area Networks (MOS'99)*, Lisbon, Portugal, 1999.

[91] MARC LACOSTE. « Building Reliable Distributed Infrastructures Revisited: a Case Study ». Dans *Proc. International DOA Workshop on Foundations of Middleware Technologies (WFoMT'02)*, Irvine, California (USA), 2002.

[92] MARC LACOSTE, FABRICE LE FESSANT, et KATHLEEN MILSTED. « Spécification d'une machine virtuelle répartie (version 1) ». Rapport technique D3.2, Projet RNRT MARVEL, 2000.

[93] BUTLER LAMPSON. « A Note on the Confinement Problem ». *Communications of the ACM*, 16(10):613–615, 1973.

[94] BUTLER LAMPSON. « Protection ». *Operating Systems Review*, 8(1):18–24, 1974.

[95] DANNY LANGE et MITSURU OSHIMA. « *Programming and Deploying Java Mobile Agents with Aglets* ». Addison-Wesley, 1998.

[96] FABRICE LE FESSANT. « The JoCaml System Prototype », 1998. Software and documentation available from http://join.inria.fr/jocaml.

[97] FABRICE LE FESSANT. « JoCaml: programmation distribuée et agents mobiles ». Dans *Actes de la Première Conférence Française sur les Systèmes d'Exploitation (CFSE'99)*, Rennes, France, 1999.

[98] FABRICE LE FESSANT. « *JoCaml : conception et implémentation d'un langage à agents mobiles* ». PhD thesis, Ecole Polytechnique, 2001.

[99] FABRICE LE FESSANT et LUC MARANGET. « Compiling Join-Patterns ». Dans *Proc. Third International CONCUR Workshop on High-Level Concurrent Languages (HLCL'98)*, *Electronic Notes in Theoretical Computer Science* 16(3), pages 35–54, Nice, France, 1998. Elsevier Science Publishers.

[100] THOMAS LEDOUX. « *Réflexion dans les systèmes répartis : application à CORBA et Smalltalk* ». PhD thesis, Ecole des Mines de Nantes, 1998.

[101] THOMAS LEDOUX. « OpenCorba : un bus logiciel réflexif adaptable ». Dans *Proc. 5e Conference sur les Langages et Modèles à Objets (LMO'99)*, Villefranche sur Mer, France, 1999. Hermès Science.

[102] THOMAS LEDOUX. « OpenCorba: a Reflective Open Broker ». Dans *Proc. Second International Conference on Reflection and Meta-Level Architecure (REFLECTION'99)*, *Lecture Notes in Computer Science* 1616, Saint-Malo, France, 1999. Springer-Verlag.

[103] THOMAS LEDOUX et al. « Etat de l'art sur l'adaptabilité ». Rapport technique D1.1, Projet RNTL ARCAD, 2001.

[104] XAVIER LEROY. « The ZINC Experiment: an Economical Implementation of the ML Language ». Rapport technique INRIA RR-117, INRIA Rocquencourt, 1990.

[105] XAVIER LEROY. « The Objective Caml System », 1996. Software and documentation available from http://caml.inria.fr/ocaml/.

[106] FRANCESCA LEVI et DAVIDE SANGIORGI. « Controlling Interference in Ambients ». Dans *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 352–364, Boston, Massachusetts (USA), 2000. ACM Press.

[107] HENRY LEVY. « *Capability-Based Computer Systems* ». Digital Press, 1984.

[108] JEAN-JACQUES LÉVY. « *Réductions correctes et optimales dans le lambda-calcul* ». Thèse de Doctorat es Sciences, Université Paris VII, 1978.

[109] JEAN-JACQUES LÉVY. « Some Results in the Join-Calculus ». Dans *Proc. Third International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, *Lecture Notes in Computer Science* 1281, pages 233–249. Springer-Verlag, 1997.

[110] TIM LINDHOLM et FRANK YELLIN. « *The Java Virtual Machine Specification* ». Addison-Wesley, 1997.

[111] LUÍS LOPES. « The TyCO Programming Language and System ». Software and documentation available from http://www.ncc.up.pt/~lblopes/tyco/.

[112] LUÍS LOPES. « *On the Design and Implementation of a Virtual Machine for Process Calculi* ». PhD thesis, University of Porto, 1999.

[113] LUÍS LOPES, FERNANDO SILVA, ALVARO FIGUEIRA, et VASCO VASCONCELOS. « DiTyCO: an Experiment in Code Mobility from the Realm of Process Calculi ». Dans *Proc. Fifth International ECOOP Workshop on Mobile Object Systems : Programming Languages for Wide Area Networks (MOS'99)*, Lisbon, Portugal, 1999.

[114] LUÍS LOPES, FERNANDO SILVA, et VASCO VASCONCELOS. « Compiling Object Calculi ». Rapport technique DCC-98-3, Departamento de Ciência de Computadores, Facultade de Ciências da Universidade do Porto, 1998.

[115] LUÍS LOPES, FERNANDO SILVA, et VASCO VASCONCELOS. « A Virtual Machine for a Process Calculus ». Dans *Proc. First ACM International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, *Lecture Notes in Computer Science* 1702, pages 244–260, Paris, France, 1999. Springer-Verlag.

[116] LUÍS LOPES, VASCO VASCONCELOS, et FERNANDO SILVA. « An Abstract Machine for an Object Calculus ». Rapport technique DCC-97-5, Departamento de Ciência de Computadores, Facultade de Ciências da Universidade do Porto, 1997.

[117] LUÍS LOPES, VASCO VASCONCELOS, et FERNANDO SILVA. « TyCO Abstract Machine - The Definition ». Rapport technique DCC-97-1, Departamento de Ciência de Computadores, Facultade de Ciências da Universidade do Porto, 1997.

[118] NANCY LYNCH. « *Distributed Algorithms* ». Morgan-Kaufmann, 1996.

[119] NENAD MEDVIDOVIC et RICHARD TAYLOR. « A Classification and Comparison Framework for Software Architecture Description Languages ». *IEEE Transactions of Software Engineering*, 26(1):70–93, 2000.

[120] NIKUNJ MEHTA, NENAD MEDVIDOVIC, et SANDEEP PHADKE. « Towards a Taxonomy of Software Connectors ». Dans *Proc. 22nd International Conference on Software Engineering (ICSE 2000)*, pages 178–187, 2000.

[121] MICROSOFT CORPORATION. « Microsoft Transaction Server ». Voir http://www.microsoft.com/com/tech/MTS.asp.

[122] MICROSOFT CORPORATION. « The Component Object Model Specification », 1995. Available from http://www.microsoft.com/com/resources/comdocs.asp.

[123] MIKADO PROJECT. Voir http://mikado.di.fc.ul.pt/.

[124] ROBIN MILNER. « A Calculus of Communicating Systems », *Lecture Notes in Computer Science* 92. Springer-Verlag, 1980.

[125] ROBIN MILNER. « *Communication and Concurrency* ». Prentice-Hall, 1989.

[126] ROBIN MILNER. « Operational and Algebraic Semantics of Concurrent Processes ». Dans JAN VAN LEEUWEN, éditeur, *Handbook of Theoretical Computer Science, Vol 2: Formal Models and Semantics*, pages 1201–1242. MIT Press, 1990.

[127] ROBIN MILNER. « The Polyadic π-Calculus: a Tutorial ». Rapport technique ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[128] ROBIN MILNER. « *Communicating and Mobile Systems: the π-Calculus* ». Cambridge University Press, 1999.

[129] ROBIN MILNER, JOACHIM PARROW, et DAVID WALKER. « A Calculus of Mobile Processes, Parts I and II ». *Journal of Information and Computation*, 100:1–77, 1992.

[130] JEAN-FRANÇOIS MONIN. « *Introduction aux Méthodes Formelles* ». Hermès, 2000.

[131] ELIE NAJM et JEAN-BERNARD STEFANI. « A Formal Semantics for the ODP Computational Model ». *Computer Networks and ISDN Systems*, 27(8):1305–1329, 1995.

[132] GEORGE NECULA. « Proof-Carrying Code ». Dans *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, 1997.

[133] OBJECT MANAGEMENT GROUP. « Common Object Request Broker Architecture (CORBA/IIOP), Version 3.0.2 ». Rapport technique formal/2002-12-06, décembre 2002. Available from http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[134] OBJECT MANAGEMENT GROUP. « The CORBA Component Model, Version 3.0 ». Rapport technique formal/2002-06-65, 2002. Available from http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[135] OBJECTWEB CONSORTIUM. « Jonathan: an Open Distributed Objects Platform ». Voir http://jonathan.objectweb.org/.

[136] OBJECTWEB CONSORTIUM. « The Fractal Software Composition Framework ». Voir http://fractal.objectweb.org/.

[137] SEAN O'MALLEY et LARRY PETERSON. « A Dynamic Network Architecture ». *ACM Transactions on Computer Systems*, 10(2):110–143, 1992.

[138] GILLES PESKINE. « Spécification d'une machine virtuelle répartie (version 2) – Volume 2: JAM: The JoCaml Abstract Machine ». Rapport technique D3.3, Projet RNRT MARVEL, 2002.

[139] SIMON PEYTON JONES. « *The Implementation of Functional Programming Languages* ». Prentice-Hall, 1987.

[140] BENJAMIN PIERCE et DAVID TURNER. « The Pict Programming Language ». Software, language definition, and tutorial available from http://www.cis.upenn.edu/~bcpierce/papers/pict/Html/Pict.html.

[141] BENJAMIN PIERCE et DAVID TURNER. « Concurrent Objects in a Process Calculus ». Dans *Theory and Practice of Parallel Programming*, *Lecture Notes in Computer Science* 907, pages 187–215, Sendai, Japan, 1995. Springer-Verlag.

[142] BENJAMIN PIERCE et DAVID TURNER. « Pict: A Programming Language Based on the π-Calculus ». Rapport technique CSCI 476, Indiana University, 1997.

[143] GORDON PLOTKIN. « Call-by-name, Call-by-value and the λ-Calculus ». *Theoretical Computer Science*, 1:125–159, 1976.

[144] ANTÓNIO RAVARA, ANA MATOS, VASCO VASCONCELOS, et LUÍS LOPES. « A Lexically Scoped Distributed π-Calculus ». Rapport technique DI/FCUL TR 02-4, Department of Computer Science, University of Lisbon, 2002.

[145] JAMES RIELY et MATTHEW HENNESSY. « A Typed Language for Distributed Mobile Processes ». Dans *Proc. of the 25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 378–390. ACM Press, 1998.

[146] DAVIDE SANGIORGI. « *Expressing Mobility in Process Algebra: First-Order and Higher-Order Paradigms* ». PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[147] DAVIDE SANGIORGI. « From π-Calculus to Higher-Order π-Calculus – and Back ». Dans *Proc. 4th International CAAP/FASE Joint Conference on Theory and Practice of Sofware Development (TAPSOFT'93)*, *Lecture Notes in Computer Science* 668, pages 151–166, Orsay, France, 1993. Springer Verlag.

[148] DAVIDE SANGIORGI. « An Interpretation of Typed Object into Typed π-calculus ». *Information and Computation*, 143(1):34–73, 1998.

[149] DAVIDE SANGIORGI et ANDREA VALENTE. « A Distributed Abstract Machine for Safe Ambients ». Dans *Proc. 28th International Colloquium on Automata, Languages and Programming (ICALP'01)*, *Lecture Notes in Computer Science* 2076, pages 408–420, Heraklion, Greece, 2001. Springer-Verlag.

[150] DAVIDE SANGIORGI et DAVID WALKER. « *The π-Calculus: A Theory of Mobile Processes* ». Cambridge University Press, 2001.

[151] ALAN SCHMITT. « *Conception et Implémentation de Calculs d'Agents Mobiles* ». PhD thesis, Ecole Polytechnique, 2002.

[152] ALAN SCHMITT. « Safe Dynamic Binding in the Join-Calculus ». Dans *Proc. Second IFIP International Conference on Theoretical Computer Science (TCS'02)*, pages 563–575, Montreal, Canada, 2002. Kluwer.

[153] ALAN SCHMITT et JEAN-BERNARD STEFANI. « The M-calculus: A Higher-Order Distributed Process Calculus ». Dans *Proc. 30th ACM Symposium on Principles of Programming Languages (POPL'03)*, New-Orleans, Louisiana (USA), 2003. ACM Press. Une version étendue est parue dans le rapport de recherche INRIA RR-4361, INRIA Rhône-Alpes, 2002. Paru également comme rapport technique du projet RNRT MARVEL, 2002.

[154] MICHAEL SCHROEDER et JEROME SALTZER. « A Hardware Architecture for Implementing Protection Rings ». *Communications of the ACM*, 15(3):157–170, 1972.

[155] PETER SEWELL, PAWEL WOJCIECHOWSKI, et BENJAMIN PIERCE. « Location-Independent Communication for Mobile Agents: a Two-Level Architecture ». Rapport technique 462, Computer Laboratory, University of Cambridge, 1999.

[156] JEAN-BERNARD STEFANI. « Open Distributed Processing : an Architectural Basis for Information Networks ». *Computer Communications*, 18(11):849–862, 1995.

[157] JEAN-BERNARD STEFANI. « A Calculus of Higher-Order Distributed Components ». Rapport technique INRIA RR-4692, INRIA Rhône-Alpes, 2003.

[158] JEAN-BERNARD STEFANI. « Requirements for a Global Computing Programming Model ». Rapport technique D1.1.2, IST MIKADO Project, 2003.

[159] JEAN-BERNARD STEFANI et FLORENCE GERMAIN. « Modèle de traitement MARVEL – Eléments d'un modèle de traitement pour MARVEL ». Rapport technique D2.0, Projet RNRT MARVEL, 2000.

[160] SUN MICROSYSTEMS. « Java Remote Method Invocation Specification ». Available from http://java.sun.com/products/jdk/rmi/.

[161] SUN MICROSYSTEMS. « Enterprise JavaBeans Specification, Version 2.1 », 2002. Available from http://java.sun.com/products/ejb/docs.html.

[162] CLEMENS SZYPERSKI. « *Component Software Systems* ». Addison-Wesley, 2002.

[163] TOMMY THORN. « Programming Languages for Mobile Code ». *ACM Computing Surveys*, 29(3):213–239, 1997.

[164] DAVID TURNER. « *The Polymorphic Pi-calculus: Theory and Implementation* ». PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1995.

[165] ASIS UNYAPOTH. « *Nomadic π Calculi: Expressing and Veriyfying Communication Infrastructure for Mobile Computation* ». PhD thesis, University of Cambridge, 2001.

[166] ASIS UNYAPOTH et PETER SEWELL. « Nomadic Pict: Correct Communication Infrastructure for Mobile Computation ». Dans *Proc. 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 116–127. ACM Press, 2001.

[167] ANDRÉ VAN DER HOEK, DENNIS HEIMBIGNER, et ALEXANDER WOLF. « Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage à Trois ». Rapport technique CU-CS-849-98, Department of Computer Science, University of Colorado, Boulder (USA), 1998.

[168] VASCO VASCONCELOS. « Typed Concurrent Objects ». Dans *Proc. 8th European Conference on Object-Oriented Programming (ECOOP'94)*, *Lecture Notes in Computer Science* 821, pages 100–117. Springer-Verlag, 1994.

[169] VASCO VASCONCELOS, LUÍS LOPES, et FERNANDO SILVA. « Distribution and Mobility with Lexical Scoping in Process Calculi ». Dans *Proc. Third International CONCUR Workshop on High-Level Concurrent Languages (HLCL'98)*, *Electronic Notes in Theoretical Computer Science* 16(3), pages 19–34, Nice, France, 1998. Elsevier Science Publishers.

[170] VASCO VASCONCELOS et MARIO TOKORO. « A Typing System for a Calculus of Objects ». Dans *Proc. First International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, *Lecture Notes in Computer Science* 742, pages 460–474. Springer-Verlag, 1993.

[171] JAN VITEK et GIUSEPPE CASTAGNA. « Seal: a Framework for Secure Mobile Computations ». Dans *Proc. Workshop on Internet Programming Languages*, *Lecture Notes in Computer Science* 1686, Chicago, Illinois (USA), 1998. Springer-Verlag.

[172] JAN VITEK et GIUSEPPE CASTAGNA. « Mobile Agents and Hostile Hosts ». Dans *Proc. 10e Journées Francophones des Langages Applicatifs (JFLA'99)*, Avoriaz, France, 1999.

[173] JAN VITEK, MANUEL SERRANO, et DIMITRI THANOS. « Security and Communications in Mobile Object Systems ». Dans *Proc. Second International Workshop on Mobile Object Systems (MOS'97) : Towards the Programmable Internet*, *Lecture Notes in Computer Science* 1222, Linz, Austria, 1996. Springer-Verlag.

[174] JIM WHITE. « Telescript Technology: the Foundation for the Electronic Marketplace ». Rapport technique, General Magic, 1994.

[175] PAWEL WOJCIECHOWSKI. « *Nomadic Pict: Language and Infrastructure Design for Mobile Computation* ». PhD thesis, University of Cambridge, 2000.

[176] PAWEL WOJCIECHOWSKI et PETER SEWELL. « Nomadic Pict: Programing Language and Infrastructure ». Software, language definition, and tutorial available from http://lsrwww.epfl.ch/~pawel/nomadicpict.html.

[177] PAWEL WOJCIECHOWSKI et PETER SEWELL. « Nomadic Pict: Language and Infrastructure Design for Mobile Agents ». Dans *Proc. First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents (ASA/MA'99)*, 1999.

[178] PAWEL WOJCIECHOWSKI et PETER SEWELL. « Nomadic Pict: Language and Infrastructure ». *IEEE Concurrency*, 8(2):42–52, 2000.

[179] WILLIAM WULF, ELLIS COHEN, WILLIAM CORWIN, ANITA JONES, ROY LEVIN, C. PIERSON, et FRED POLLAK. « HYDRA: the Kernel of a Multiprocessor Operating System ». *Communications of the ACM*, 17(6):337–345, 1974.