HAL open science

Répartition modulaire de programmes synchrones

Gwenaël Delaval

To cite this version:

Gwenaël Delaval. Répartition modulaire de programmes synchrones. Langage de programmation [cs.PL]. Institut National Polytechnique de Grenoble - INPG, 2008. Français. NNT : . tel-00750832

HAL Id: tel-00750832

https://theses.hal.science/tel-00750832v1

Submitted on 12 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

Thèse

pour obtenir le grade de

Docteur de l'Institut Polytechnique de Grenoble

Spécialité : Informatique

préparée à l'Institut National de Recherche en Informatique et en Automatique
centre Grenoble-Rhône-Alpes, projet Pop-Art

École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

présentée et soutenue publiquement par

Gwenaël Delaval

le 1er juillet 2008

Répartition modulaire de programmes synchrones

Co-directeurs de thèse : Marc Pouzet
Alain Girault

Jury

Denis Trystram	INP Grenoble	Président
François Pottier	Inria Rocquencourt	Rapporteur
Jean-Pierre Talpin	Inria Rennes	Rapporteur
Marc Pouzet	Université Paris-Sud	Co-directeur de thèse
Alain Girault	Inria Rhône-Alpes	Co-directeur de thèse
Mamoun Filali Amine	CNRS, IRIT, Toulouse	Examinateur
Luc Maranget	Inria Rocquencourt	Examinateur

Résumé

Nous nous intéressons à la conception sûre de systèmes répartis. Nous montrons qu'avec la complexité et l'intégration croissante des systèmes embarqués, la structure fonctionnelle du système peut entrer en conflit avec la structure de son architecture. L'approche traditionnelle de conception par raffinement de cette architecture compromet alors la modularité fonctionnelle du système.

Nous proposons donc une méthode permettant de concevoir un système réparti défini comme un programme unique, dont la structure fonctionnelle est indépendante de l'architecture du système. Cette méthode est basée sur l'ajout de primitives de répartition à un langage flots de données synchrone. Ces primitives permettent d'une part de déclarer l'architecture sous la forme d'un graphe définissant les ressources existantes et les liens de communication existant entre ces ressources, et d'autre part de spécifier par des annotations la localisation de certaines valeurs et calculs du programme.

Nous définissons ensuite la sémantique formelle de ce langage étendu. Cette sémantique a pour but de rendre compte de manière formelle l'effet des annotations ajoutées par le programmeur. Un système de types à effets permet ensuite de vérifier la cohérence de ces annotations. Ce système de types est muni d'un mécanisme d'inférence, qui permet d'inférer, à partir des annotations du programmeur, la localisation des calculs non annotés. Nous définissons ensuite, à partir de ce système de types, une méthode de répartition automatique permettant d'obtenir, à partir d'un programme annoté, un fragment de programme par ressource de l'architecture. La correction du système de types avec la sémantique du langage est prouvée, ainsi que l'équivalence sémantique de l'exécution des fragments obtenus par la méthode de répartition automatique avec le programme initial.

Cette méthode a été implémentée dans le compilateur du langage Lucid Synchrone, et testée sur un exemple de radio logicielle.

Abstract

We address the problem of safe design of distributed embedded systems. We show that with the increasing complexity and integration of embedded systems, the functional structure of the system can conflict with the structure of its architecture. The usual approach of design by refinement of this architecture can then jeopardize the functional modularity of the system.

We propose a method allowing the design of a distributed system as a unique program, which functional structure is independent of the system's architecture. This method is based on the extension of a synchronous dataflow programming language with primitives for program distribution. These primitives allow on one hand to describe the architecture as a graph defining existing computing resources and communication links between them, and on the other hand to specify by mean of annotations the localization of some values and computations of the program.

We define the formal semantics of this extended language. The purpose of this semantics is to provide to the programmer a formal meaning to the annotations added to his program. A type and effect system is provided to ensure, at compilation time, the consistency of these annotations. A type inference mechanism allow the inference, from the annotations of the programmer, of the non-annotated values and computations. We finally define, from this type system, an automatic distribution method, allowing the computation, from an annotated program, of one program fragment for each computing resource of the architecture. The soundness of the type system with respect to the semantics has been proven, as well as the semantical equivalence of the composition of the fragments obtained via the automatic distribution method with the initial program.

This method has been implemented in the compiler of the Lucid Synchrone language, and experimented with an example of software-defined radio.

Remerciements

Je tiens à remercier tout d'abord Alain Girault et Marc Pouzet de m'avoir permis de réaliser cette thèse sous leur co-direction. Leur apport, humain et scientifique, dans ce travail, est inestimable. Ils m'ont énormément appris, et je souhaite qu'ils sachent que malgré toutes les difficultés rencontrées, c'est grâce à eux, à leur patience, leurs encouragements, que cette thèse a pu être menée à bon terme.

Je remercie les membres de mon jury, et particulièrement mes deux rapporteurs, François Pottier et Jean-Pierre Talpin. J'ai eu la chance d'avoir eu des échanges très constructifs avec eux, je les remercient donc pour leur disponibilité. Ils m'ont permis d'améliorer significativement ce mémoire. Je remercie aussi Denis Trystram d'avoir accepté de présider ce jury, et Mamoun Filali-Amine et Luc Maranget d'y avoir été examinateurs.

J'ai poursuivi, pendant ces quatre années, ma collaboration avec Éric Rutten, commencée à l'occasion de mon stage de master. Je le remercie chaleureusement pour cette collaboration, et pour la distance et l'ouverture d'esprit qu'il sait apporter en toute circonstance.

Je remercie aussi Laurent Mounier, Philippe Bizard, Anne Rasse, Guillaume Huard, Jean-Marc Vincent et Cyril Labbé de m'avoir intégré dans leur équipe d'enseignement.

L'environnement de travail n'étant rien sans environnement humain, au sens large, je remercie vivement :
Hamoudi, Louis, Élodie, Florence, Alexandre, Mouaiad et Gérald, avec qui j'ai partagé... plus que mes directeurs de thèse ; leur soutien et leurs conseils amicaux ont été plus que décisifs...

... L'équipe Pop-Art : Pascal, Simplice, Gregor, Xavier, Daniel, Bertrand, Emil, Shara, ainsi que les membres et ex-membres de l'équipe Bipop, Roger et Soraya...

... L'équipe VerTeCs, pour son accueil et ses pauses-café-polémiques interminables : Hervé, Thierry, Jérémy, Camille, Tristan, Christophe, Nathalie, Hatem, Florimont, Patricia et Vlad...

... Laurent, Aurélie et Lalao, rencontrés à l'occasion du monitorat...

... Les potes de Villeneuve et de Mounier : Pauline, Florian, Pauline, Olivier, Véro...

... Toute ma famille pour son soutien sans faille...

... Ceux que j'aurais oublié...

... Anne, enfin, sans qui rien ne serait aujourd'hui possible.

Table des matières

1 Introduction **13**

2 Problématique **17**
2.1 Programmation de systèmes répartis 17
2.2 Compilation modulaire 18
2.3 Programmation synchrone 19
2.3.1 Systèmes réactifs synchrones 19
2.3.2 Langages synchrones 22
2.3.3 Intégration langage et répartition 24
2.3.4 Compilation, répartition et désynchronisation 25
2.3.5 Compilation modulaire de programmes synchrones 26
2.3.6 Ordre supérieur et reconfiguration dynamique 30
2.4 Lucid Synchrone 31
2.4.1 Présentation générale 31
2.4.2 Formalisation du langage 41
2.4.3 Sémantique comportementale 43
2.5 La radio logicielle 47
2.6 État de l'art 50
2.6.1 Conception de systèmes répartis 50
2.6.2 Répartition de programmes synchrones 53
2.6.3 Systèmes de types pour la répartition de programmes 55
2.7 Contribution 56

3 Extension du langage pour la répartition **59**
3.1 Rappel de la problématique 59
3.2 Architectures homogènes 60
3.3 Architectures hétérogènes 61
3.4 Communications 63
3.5 Conclusion 64

4 Sémantique synchrone et localisation 65
4.1 Une sémantique pour le programmeur . . . 65
4.2 Localisation des valeurs . . . 66
4.3 Exécution répartie . . . 67
4.4 Exemple . . . 69
4.5 Sémantique pour architectures hétérogènes . . . 73
4.6 Conclusion . . . 73

5 Système de types pour la répartition 75
5.1 Justification et description . . . 75
5.2 Exemples . . . 77
5.2.1 Typage d'équations . . . 77
5.2.2 Typage des nœuds . . . 78
5.3 Formalisation . . . 81
5.4 Illustration . . . 85
5.5 Adaptation pour architectures hétérogènes . . . 87
5.6 Correction du système de types . . . 88
5.7 Implémentation . . . 95
5.8 Discussion . . . 97

6 Répartition 99
6.1 Principe . . . 99
6.1.1 Projection d'ensembles d'équations . . . 99
6.1.2 Projection de nœuds . . . 101
6.2 Inférence des canaux de communication . . . 102
6.3 Opération de projection . . . 109
6.4 Correction de l'opération de projection . . . 114
6.4.1 Complétude . . . 114
6.4.2 Équivalence sémantique . . . 122
6.5 Application au canal de radio logicielle . . . 130
6.6 Répartition sur des systèmes GALS . . . 131
6.7 Conclusion . . . 135

7 Horloges et introduction du contrôle 137
7.1 Horloges en Lucid Synchrone . . . 137
7.2 Horloges globales . . . 139
7.3 Structures de contrôle . . . 142
7.3.1 Principe et exemples . . . 142
7.3.2 Application à la radio logicielle . . . 144
7.4 Extension du langage . . . 145
7.5 Sémantique synchrone . . . 145

7.6 Système de types 151
7.7 Projection 151
7.8 Discussion 154

8 Architectures locales 157
8.1 Motivation 157
8.2 Extension du langage 158
8.3 Système de types modifié 160
8.4 Discussion 161

9 Implémentation et discussion 165
9.1 Implémentation 165
9.2 Limites d'expressivité 167
9.3 Application à d'autres cadres de répartition 169
9.3.1 Asynchronie des rythmes 169
9.3.2 Tolérance aux fautes 170
9.3.3 Synthèse de contrôleurs 171
9.4 Ordre supérieur et reconfiguration dynamique 172
9.5 Conclusion 174

10 Conclusion 175
10.1 Résumé 175
10.2 Perspectives 176

Chapitre 1

Introduction

Les systèmes informatiques embarqués font maintenant partie de notre quotidien, depuis les téléphones portables et l'électronique grand public, jusqu'à l'informatique embarquée dans des systèmes critiques, systèmes dont la défaillance peut avoir de graves conséquences matérielles, humaines ou écologiques, tels que les systèmes automobiles, aéronautiques ou nucléaires. Ce sont les systèmes informatiques les plus présents autour de nous, et les plus invisibles : leur transparence est une condition essentielle de leur sécurité ainsi que de leur confort d'utilisation.

Cette transparence et cette criticité rendent nécessaire plus qu'ailleurs l'absence de défaillance du système développé. De plus, les concepteurs de tels systèmes sont souvent des spécialistes du domaine d'application, et non des spécialistes de méthodes informatiques. Il est donc important de leur fournir des méthodes formelles de conception sûres, qui soient transparentes et intelligibles aux non-spécialistes de ces méthodes formelles.

Nous nous plaçons donc dans le cadre de l'approche synchrone pour la conception de systèmes embarqués [10]. Cette approche a permis, dans les deux dernières décennies, de définir des langages de haut niveau de plus en plus expressifs, associés à des outils mathématiques de description (sémantiques formelles), d'analyse (model-checking, outils de test automatique) et de transformation de programmes (compilation [6, 46], synthèse de contrôleurs discrets [63]). Ces outils formels, en automatisant des parties entières du processus de conception, permettent d'améliorer à la fois la sécurité et l'intelligibilité du système conçu.

Une autre caractéristique importante des systèmes embarqués est le fait que leur architecture matérielle devient de plus en plus complexe, composée de ressources à la fois plus nombreuses et plus spécialisées, et destinées à intégrer simultanément plus de fonctionnalités du système. Ces préoccupations se rencontrent par exemple dans le domaine de la radio logicielle [65], et dans les architectures modulaires intégrées du domaine aéronautique (architectures « IMA » [12]) ou automobile (préconisations du consortium Autosar [1]). Dans ce contexte, la

méthode classique de répartition manuelle [22], ou en utilisant un langage de description d'architecture [36], affaiblit l'intérêt de l'utilisation des outils formels. Ces méthodes supposent en effet de programmer séparément, au moins à partir d'un certain stade de la conception, chaque ressource matérielle du système. Or, la spécialisation et l'intégration simultanée de plusieurs fonctionnalités impliquent que la structure fonctionnelle du système programmé n'est pas un raffinement de la structure matérielle de l'architecture : une ressource matérielle peut être impliquée simultanément dans plusieurs fonctionnalités, tandis qu'une fonctionnalité peut impliquer plusieurs ressources matérielles. La programmation séparée de ces ressources contrevient donc aux besoins d'analyses, de tests et de simulations à effectuer sur des modules fonctionnels cohérents.

Nous nous intéressons donc ici à la conception sûre de systèmes embarqués répartis, et plus particulièrement à l'intégration langage de la description de tels systèmes. Cette intégration doit permettre de décrire en un unique programme un système composé de plusieurs ressources matérielles, dans l'hypothèse où la structure fonctionnelle ne suit pas la structure de l'architecture.

D'autre part, nous nous situons dans le cadre de la compilation modulaire de programmes synchrones [24]. L'argument principal sur lequel repose ce travail tient à la volonté de garder, tout au long du cycle de compilation, la structure fonctionnelle du système telle que décrite par le programmeur. Or, les méthodes de répartition automatique existantes [16, 9, 55] opèrent sur des programmes entièrement mis à plat, dans lesquels cette structure n'est plus visible.

Nous proposons donc une méthode de répartition automatique intégrée à la compilation modulaire d'un langage synchrone, étendu à l'aide de primitives de répartition. Ces primitives prennent la forme d'annotations permettant au programmeur de spécifier, à n'importe quel niveau du programme, sur quelle ressource physique un calcul est exécuté. Le code ci-dessous est la composition de deux fonctions `f` et `g`, `f` étant exécutée sur le site `A` et `g` sur le site `B` :

```
    y = f(x) at A
and z = g(y) at B
```

La méthode de répartition automatique est ensuite basée sur la définition d'un système de types, appelé « système de types spatiaux », permettant l'inférence de la localisation sur l'architecture des valeurs du programme en suivant la structure fonctionnelle du programme. Ce système de types permet de vérifier la cohérence des annotations, et de les compléter par un mécanisme d'inférence calculant la localisation des calculs non annotés. La répartition elle-même est une opération dirigée par ces types spatiaux, suivant l'approche classique de compilation dirigée par les types (par exemple, pour l'analyse de la mémoire utilisée [76], ou l'optimisation de la représentation des données [57]). Nous montrons que cette approche intégrée permet de traiter l'ordre supérieur statique, c'est-à-dire l'expression de fonctions

paramétrées par d'autres fonctions, offrant ainsi un cadre expressif pour la conception de systèmes répartis. L'approche ainsi décrite est motivée par un exemple de conception d'un système de radio logicielle. Nous avons appliqué cette méthode au langage Lucid Synchrone [2], langage flots de données synchrone inspiré de Lustre [44] et intégrant des caractéristiques issues des langages fonctionnels (ordre supérieur, polymorphisme, inférence de types). Nous avons étendu ce langage avec des constructions de déclaration d'architecture, et d'annotation de programmes flots de données en fonction de cette architecture. Le système de types spatiaux et l'opération de répartition automatique ont été implémentés dans le compilateur du langage.

Cette thèse s'inscrit dans deux tendances actuelles concernant la recherche sur les méthodes de conception des systèmes embarqués. La première est une tendance d'intégration de méthodes existantes au sein de mêmes outils, ou par la définition de formats ou de langages communs. La deuxième tendance est la définition de langages, ou plus généralement d'interfaces permettant d'utiliser ces méthodes formelles de façon transparente par des utilisateurs non spécialistes du domaine de ces méthodes formelles.

Nous présenterons tout d'abord en détail dans le chapitre 2 les motivations et le contexte dans lequel s'inscrit ce travail. Le chapitre 3 discute de l'extension du langage considéré pour la répartition. Nous donnons une sémantique de ce langage étendu au chapitre 4. La définition du système de types spatiaux est donnée au chapitre 5, et la méthode de répartition automatique associée au chapitre 6. Nous discutons de l'ajout de structures de contrôle à cette méthode au chapitre 7, puis de l'extension du langage de description de l'architecture au chapitre 8. Enfin, au chapitre 9, une discussion pose les limites de cette approche, et ouvre des perspectives sous la forme de domaines d'application différents, ainsi qu'en considération de langages plus expressifs.

Chapitre 2

Problématique

Nous présentons ici plus en détail les motivations et le contexte de ce travail. Les motivations s'articulent autour de deux points principaux : la programmation de systèmes répartis et la compilation modulaire. L'utilisation de l'ordre supérieur pour l'expression de fonctionnalités de reconfiguration dynamique est une motivation supplémentaire, permise par les deux premières. La section 2.1 présente le type de systèmes et le type de répartition considérés, et motive le problème sous la forme d'un conflit entre la modularité fonctionnelle, décrite par le programmeur, et la description de l'architecture physique du système. Nous présentons ensuite en section 2.3 le problème sous l'angle de la programmation synchrone. Le langage Lucid Synchrone est présenté en section 2.4, et permet de motiver la définition du noyau d'un langage flot de données muni de l'ordre supérieur statique, sur lequel notre méthode de répartition va être définie. Enfin, un exemple de motivation est présenté en section 2.5, avant un rappel de l'état de l'art, et de la contribution présentée dans cette thèse.

2.1 Programmation de systèmes répartis

Les systèmes répartis considérés ici le sont pour des raisons de répartition fonctionnelle. Il s'agit donc, à partir d'un unique programme décrivant le système complet, d'obtenir pour chaque ressource du système, la partie du programme adéquat à cette ressource. Ces ressources, ou sites, peuvent être soit des ressources physiques (processeurs, circuits), soit des ressources logiques (processus légers, tâches temps-réel exécutées à différents rythmes). On parlera de répartition physique dans le premier cas, de répartition logique dans le second.

Dans le contexte des systèmes embarqués, la répartition physique est intrinsèque compte tenu de l'architecture de ces systèmes. Le caractère réparti de ces architectures traduit en premier lieu le fait que les ressources physiques en contact

avec l'environnement, tels que les capteurs et les actionneurs, sont elles-mêmes réparties dans le système. D'autre part, la criticité de ces systèmes impose l'application de stratégies de tolérance aux fautes, impliquant la redondance physique des ressources de calcul. Ces considérations concernent par exemple l'industrie automobile ou aéronautique.

Pour d'autres systèmes moins répartis géographiquement parlant, la répartition physique peut être nécessaire pour des raisons d'adéquation des différentes ressources physiques de l'architecture à certains calculs. C'est le cas par exemple des systèmes de radio logicielle [65], où les fonctions de filtrage et de démodulation sont chacune exécutées sur des ressources spécifiques.

La répartition logique a un intérêt pour exécuter différentes parties du programme de manière cyclique, avec des périodes différentes. C'est le cas par exemple de certains systèmes robotiques, où certaines tâches de calcul, plus coûteuses et plus longues (calculs de matrices d'inertie), sont exécutées par des tâches différentes que d'autres calculs moins coûteux et plus urgents (calcul de la trajectoire et de la commande à appliquer) [74].

Ce besoin de répartition va de pair, et entre en contradiction avec un besoin croissant d'intégration, que celle-ci soit physique (ressources de calcul partagées) ou fonctionnelle (interaction forte entre les fonctionnalités du système impliquant la conception globale de celui-ci). Ce besoin d'intégration se traduit par exemple, dans l'industrie aéronautique, par le développement du concept d'architectures « IMA »[1], permettant un plus grand partage des ressources, une plus grande maintenabilité du système, une réduction de la redondance matérielle [12]. Ces mêmes préoccupations sont présentes dans les préconisations du consortium Autosar [1] pour la conception des architectures matérielles et logicielles dans l'industrie automobile. Le développement du concept de radio logicielle provient lui-même de ce besoin d'intégration de diverses fonctionnalités des systèmes radio, par exemple dans la téléphonie mobile, que ce soit pour les terminaux ou les stations relais.

Les systèmes répartis traités ici sont donc des systèmes dont le nombre de ressources est de l'ordre de la dizaine. Ce travail ne concerne pas les systèmes utilisés pour les calculs massivement parallèles composés de centaines ou de milliers de ressources de calcul, telles que les grilles de calcul.

2.2 Compilation modulaire

La compilation modulaire de programmes consiste à compiler une fonction du langage source en une unique fonction du langage cible, indépendamment du contexte d'appel ultérieur de cette fonction. Cette approche permet de compiler le

[1] Integrated Modular Avionics

langage de manière compositionnelle, et permet au code produit d'avoir une plus grande traçabilité. Cette méthode de compilation passe aussi plus facilement à l'échelle. Elle est utilisée de manière classique dans les langages de programmation généralistes.

L'approche par programmation séparée d'éléments différents de l'architecture entre en contradiction avec l'approche d'ingénierie classique : conception d'éléments atomiques fonctionnels, tests et/ou simulations unitaires, intégration et simulation globale du système. Il y a donc un conflit entre l'objectif de modularité de l'architecture (impliquant la conception conjointe d'éléments exécutés sur le même composant de l'architecture), et celui de modularité fonctionnelle (impliquant la conception conjointe d'éléments participant à la même fonctionnalité). Nous souhaitons ici concilier ces deux approches, en proposant une méthode de répartition qui permette de conserver et d'utiliser la structure fonctionnelle lors de la compilation du programme, même en présence de ce conflit.

La méthode que nous proposons n'est pas modulaire au sens strict proposé ci-dessus, étant donné que le résultat de la répartition d'une fonction est un ensemble de fonctions spécialisées pour chacune des ressources de calcul de l'architecture. Cependant, nous qualifions cette méthode de répartition modulaire, au sens où la répartition d'une fonction sera effectuée indépendamment de son contexte d'appel, ainsi que du contenu des autres fonctions qui la compose.

2.3 Programmation synchrone

Nous nous plaçons dans le cadre de la vision réactive synchrone des systèmes embarqués. Cette section présente les systèmes réactifs synchrones, et les langages synchrones existants. Ces langages sont compilés de manière classique en un programme séquentiel, exprimé dans un langage cible général (C par exemple). Nous montrons de manière intuitive comment adapter cette méthode de compilation dans un cadre réparti.

2.3.1 Systèmes réactifs synchrones

Systèmes réactifs

Un système réactif [50] est un système qui réagit à son environnement à la vitesse imposée par celui-ci. Cette définition se pose en distinction avec d'une part les *systèmes transformationnels*, et d'autre part les *systèmes interactifs*.

- Les systèmes transformationnels, qui sont les systèmes qui prennent une entrée en début d'exécution, effectuent un calcul à partir de cette entrée, et produisent une sortie, le résultat de ce calcul, en terminant. La sortie est

```
state := initial;
foreach tick do
    read inputs;
    (outputs,state') := f(inputs,state);
    state := state';
    write outputs;
done
```

FIG. 2.1 – Modèle d'exécution d'un système réactif.

considérée comme la *transformation* de l'entrée. Les compilateurs sont des exemples de programmes transformationnels.

– Les systèmes interactifs sont des systèmes dont la production d'une sortie ne signifie pas la fin de l'exécution : ils *interagissent* avec l'environnement. Une entrée de celui-ci déclenche un traitement, puis une sortie, puis le système se remet en attente d'une nouvelle entrée. La différence avec les systèmes réactifs est que cette interaction se déroule à un rythme imposé par le système : la durée du traitement entre deux entrées successives, du fait du système, peut être arbitrairement longue. Les systèmes d'exploitation sont typiquement des systèmes interactifs.

À l'inverse, la réaction à son environnement d'un système réactif est supposée immédiate ou quasi-immédiate. En pratique, ce sont des systèmes interactifs dont on assure, si possible par construction, la faible durée du traitement entre l'entrée de l'environnement et la réaction du système à cette entrée. Quand l'environnement est continu, les entrées considérées sont échantillonnées à un rythme pertinent vis-à-vis du système et de sa criticité.

L'exécution d'un système réactif peut donc être modélisée sous la forme du programme de la figure 2.1. Une fonction transformationnelle f, prenant en entrée les entrées du système et l'état courant, et calculant les sorties à émettre et l'état suivant [8], est exécutée à l'intérieur d'une boucle sans fin. Cette boucle est activée par l'évènement noté `tick`, représentant soit une horloge permettant d'échantillonner les entrées, soit l'occurrence d'une entrée dans le cas d'un système répondant aux évènements de l'environnement.

Programmation de systèmes réactifs

De par leur nature critique, les méthodes de programmation des systèmes réactifs doivent obéir à deux caractéristiques majeures : le *déterminisme* des programmes décrits par ces méthodes, et le *parallélisme d'expression*.

Déterminisme : Cette caractéristique est particulièrement importante pour la conception et la mise au point de systèmes critiques : elle permet de simuler un système en conception en étant sûr que le fonctionnement simulé sera identique au fonctionnement réel de ce système.

De manière usuelle, les méthodes de programmation des systèmes réactifs utilisent le caractère atomique des réactions décrites pour offrir une sémantique déterministe au programmeur. L'atomicité de la réaction est assurée par le fait que la durée du traitement à tout instant est supposée plus courte que la réaction de l'environnement (ou de son échantillon). Cette atomicité permet d'assurer, à partir du déterminisme de la réaction elle-même, que deux séquences d'entrées identiques de la part de l'environnement aboutiront nécessairement à la même séquence de sorties.

Parallélisme d'expression : Un système réactif est lui-même composé de sous-systèmes eux-mêmes réactifs. Chacun de ces sous-systèmes réagit avec l'environnement, ainsi qu'aux autres sous-systèmes. Un système réactif sera donc usuellement décrit au moyen de la composition parallèle de programmes réactifs. Ce parallélisme est un parallélisme *d'expression*, au sens où si deux sous-systèmes mis en parallèle évoluent et réagissent simultanément, cette simultanéité peut ne pas avoir de traduction physique : la réaction des deux sous-systèmes à l'environnement peut être la séquence des deux réactions, pourvu que la durée de cette séquence elle-même soit faible en regard de la réaction de l'environnement.

Approche synchrone

La composition parallèle asynchrone de sous-systèmes réactifs rend la compréhension du système obtenu souvent difficile, en raison de la nécessité de définir un ordre de réaction entre les sous-systèmes composés.

Un système réactif synchrone [43] est un système sur lequel les abstractions suivantes sont faites :

- ce système est muni d'une échelle de temps discrète ;
- chaque réaction de l'environnement (ou son échantillon si cet environnement est continu) et du système est atomique et se déroule à un instant discret donné : les événements composant ces réactions sont indissociables et non ordonnés à l'intérieur de cet instant ;
- à un instant donné, toute réaction d'une partie du système ou de l'environnement est instantanément propagée *dans le même instant* à l'ensemble du système, qui peut y réagir dans ce même instant.

Cette dernière abstraction signifie qu'une analyse de causalité est effectuée statiquement, à la compilation, permettant de déterminer un ordre d'évaluation du programme synchrone, ordre qui sera suivi à chaque instant.

Ces trois abstractions permettent de concevoir des systèmes réactifs déterministes et concurrents munis d'une sémantique formelle simple et intelligible pour le programmeur. L'échelle de temps discrète est une modélisation du temps lors de l'exécution du programme ; ce modèle du temps est donc implicitement intégré à tout programme synchrone. Cela permet de tester un programme synchrone par simulation et vérification formelle, indépendemment de son contexte réel d'exécution (architecture matérielle et environnement). Pour cette raison, cette approche est particulièrement adaptée à la programmation de systèmes embarqués.

2.3.2 Langages synchrones

Les langages intégrant l'approche synchrone sont historiquement basés sur deux tendances complémentaires : les langages flots de données, à caractère fonctionnel, et les langages de style impératif.

Langages flots de données

Les valeurs portées par l'échelle de temps discret définissant le système sont considérées comme des *flots*, c'est-à-dire des séquences infinies de valeurs. La relation à chaque instant discret entre la valeur des entrées et celle des sorties est définie par un ensemble d'équations entre les flots. La notion d'état du système est exprimée par la possibilité de faire référence aux valeurs passées des flots. Le caractère synchrone se traduit par le fait que ces équations, sémantiquement parlant, ne sont pas évaluées en séquence, mais de manière concurrente à l'intérieur d'un instant discret. Un ordre d'évaluation est déterminé statiquement à la compilation à partir des dépendances entre les valeurs introduites par ces équations.

Un langage flots de données permet ainsi de décrire un système synchrone évoluant dans le temps, en écrivant des invariants entre les valeurs du programme : $x = e$ signifie qu'à chaque instant logique n, $x_n = e_n$. Par exemple, si x est un flot de données ($x = x_0, x_1, \ldots$), un langage flot de données permettra de définir le flot $y = y_0, y_1, \ldots$, tel que la valeur de y est la somme des valeurs passées de x, par les équations :

$$\begin{cases} y_0 = x_0 \\ y_n = y_{n-1} + x_n \quad \text{si } n \geq 1 \end{cases}$$

En Lustre [44], ce programme s'écrit :

```
y = x -> (pre y + x)
```

La syntaxe Signal [56] de ce programme est d'autre part :

```
Y := (Y $ 1 init 0) + X
```

La syntaxe et la sémantique des langages flots de données ont été conçues de manière à pouvoir implémenter de façon la plus directe possible des systèmes spécifiés sous forme d'équations mathématiques ou de diagrammes de blocs. Un programme flots de données peut être considéré comme un réseau de Kahn, c'est-à-dire un réseau de processus déterministes communiquant par files d'attentes [54]. Le caractère synchrone est de ce point de vue établi par le fait que les files d'attentes sont de taille bornée. De plus, lorsque le programme est synchrone et causal, il existe un ordonnancement pour lequel les files d'attente sont de taille nulle. La sémantique de flots de données synchrones correspond par ailleurs à la sémantique habituelle des circuits synchrones. La figure 2.2 montre la traduction du programme Lustre ci-dessus sous forme de réseau de Kahn.

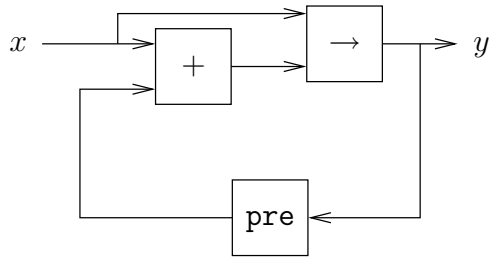


FIG. 2.2 – Traduction sous forme de réseau de Kahn du programme Lustre `y = x -> pre y + x`.

Les langages Lustre [44], Signal [56] et Lucid Synchrone [23, 2] sont des représentants de cette famille de langage. Lustre et Lucid Synchrone sont des langages fonctionnels, Lucid Synchrone étant une extension de Lustre à l'ordre supérieur. Signal est un langage relationnel : une équation est, de manière plus générale, une relation entre les flots, et non la définition d'un flot en fonction d'autres flots comme c'est le cas pour Lustre et Lucid Synchrone. L'outil industriel SCADE, basé sur le langage académique Lustre, est muni d'un langage graphique sous forme de diagrammes de blocs.

Langages impératifs

Les langages synchrones impératifs permettent de décrire explicitement un système réactif sous la forme d'un système de transitions, dans lequel l'état du système est explicite. Le langage Esterel [11] permet de décrire des processus de manière séquentielle. Ce langage est muni de constructions impératives (boucles, exceptions, structures de contrôle), et d'un opérateur de composition parallèle synchrone.

Les langages StateCharts [49], SyncCharts [7] et Argos [62] sont des représentants de cette famille de langages. Ils permettent de décrire des machines de

Mealy généralisées (aux types de données autres que booléens) et sont munis d'une opération de composition synchrone.

Langages synchrones actuels

Les langages décrits dans les sections précédentes ont été définis par rapport à des besoins et des contextes d'utilisation précis. Les langages flots de données sont ainsi plus adaptés à la programmation de systèmes dynamiques issus de l'automatique ou du traitement du signal ; les langages impératifs permettent d'exprimer de manière plus aisée des systèmes à évènements discrets, ou l'ordonnancement dans le temps de tâches ou d'évènements. Ces deux approches offrent donc des moyens, et des outils associés complémentaires pour la programmation de systèmes réactifs. La programmation de systèmes de plus en plus complexe nécessite cependant, en général, de combiner ces deux paradigmes. Les approches langages actuelles tendent donc à intégrer différents paradigmes, ou différents outils de programmation au sein d'une même méthode, d'un même outil (plateforme) ou d'un même langage. Ainsi, l'extension SignalGTi [71] du langage Signal permet d'exprimer les intervalles de temps à l'intérieur desquels certaines équations seront évaluées. Le langage des automates de modes [61], et sa généralisation dans le langage Lucid Synchrone [29], sont des exemples d'intégration de fonctionnalités impératives (description de systèmes sous la forme de machines à états) dans des langages flots de données.

Cette volonté d'intégration se retrouve dans l'outil Ptolemy [15]. Ptolemy est un outil de conception de systèmes dits « hétérogènes ». Plutôt que de proposer un langage unique, cet outil propose un ensemble de « domaines ». Ces domaines sont autant de langages spécialisés (chacun mettant en œuvre un modèle de calcul particulier), permettant la conception d'un système, de manière globale, comme la conception de sous-systèmes décrits dans un formalisme adapté à chacun de ces sous-systèmes. Ceux-ci peuvent être décrits par exemple sous forme de programmes flots de données synchrones ou de systèmes à évènements discrets. La difficulté de cet outil réside dans la multiplication des sémantiques offertes au programmeur : aux sémantiques associées à chaque domaine, s'ajoutent celles des interactions entre les domaines utilisés. Cet environnement a été utilisé pour développer le modèle *charts [40], formalisation de l'intégration des modèles de flots de données synchrones (SDF) et d'automates d'états finis (FSM).

2.3.3 Intégration langage et répartition

L'approche habituelle de répartition manuelle de programmes synchrones entre en conflit avec cette tendance d'intégration. En effet, cette approche, qui rejoint l'approche utilisant les langages de description d'architecture [36], suppose la pro-

grammation séparée de chaque ressource matérielle de l'architecture. Or, dans le contexte évoqué en section 2.1, dans lequel les ressources matérielles des systèmes embarqués sont à la fois plus nombreuses, plus spécialisés, et impliqués dans un nombre croissant de fonctionnalités orthogonales, la programmation séparée des ressources matérielles entre en contradiction avec les besoins d'analyses, de tests et de simulations de modules fonctionnels cohérents, ceux-ci pouvant impliquer l'utilisation de plusieurs ressources de manière simultanée.

Notre approche est donc basée sur l'hypothèse que la structure fonctionnelle du système n'est pas un raffinement de la structure matérielle. Le programmeur doit pouvoir décrire de manière globale un système, et sa répartition dans le même langage. Ce langage doit donc fournir des primitives de répartition, qui peuvent apparaître à n'importe quel niveau de description : au niveau des instructions de base, ou à un niveau d'intégration plus élevée (répartition de parties entières du programme). Une conséquence importante, est le fait que les directives de répartition peuvent, d'une part, ne pas être cohérentes entre elles, et d'autre part, peuvent ne pas couvrir l'ensemble du programme. Il faut donc fournir, avec ce langage muni de primitives de répartition, des outils permettant :

1. de vérifier la cohérence des annotations de répartition introduites par le programmeur,
2. de compléter ces directives en annotant les parties du programmes non annotées par le programmeur,
3. d'obtenir automatiquement le système réparti à partir du programme ainsi annoté.

2.3.4 Compilation, répartition et désynchronisation

La méthodologie de conception de systèmes répartis au moyen de langages synchrones est résumée dans la figure 2.3. Le programme synchrone considéré dans cette exemple est donné par les équations flots de données suivantes :

```
    x = f(w)
and y = g(x)
and z = h(x,y)
```

Les étapes de la conception d'un système réparti à partir de ce programme sont :

1. La sémantique d'exécution sur une échelle de temps discret du programme attribue une horloge unique cl aux trois opérateurs f, g et h composant ce programme (figure 2.3(a)). Cette horloge unique signifie que ces trois opérateurs seront effectués dans un même instant logique, et que les valeurs calculées seront propagées et disponibles dans ce même instant.

2. La vérification ou la simulation du programme de manière centralisée est possible en affectant une horloge concrète à l'horloge *cl*. Cela peut se traduire par exemple par l'encapsulation d'un code séquentiel produit par la compilation du programme dans une boucle infinie :

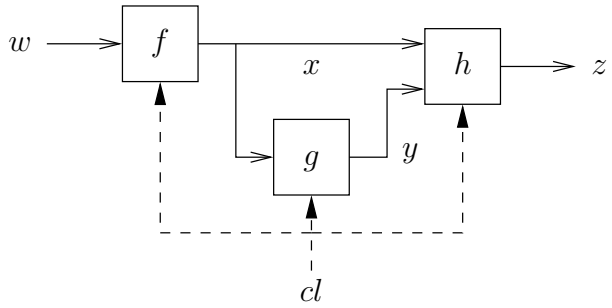
```
while true do
    read(w);
    x := f(w);
    y := g(x);
    z := h(x,y);
    write(z);
done
```

3. La répartition du programme synchrone consiste en affectant une ressource de calcul à chaque opérateur du programme (figure 2.3(b)). On obtient ainsi un fragment de programme par ressource de l'architecture considérée.
4. Les différents fragments peuvent ensuite être désynchronisés, en attribuant une horloge différente à chaque fragment, et en ajoutant des communications entre les opérateurs d'horloges différentes : par exemple, une file (figure 2.3(c)). Cette dernière opération n'est pas possible sans hypothèses sur les horloges du programme [68]. Notamment, cette opération n'est pas possible si une ressource n'est pas en mesure de récupérer l'horloge d'une autre ressource dont elle reçoit des valeurs. Le langage utilisé dans cette thèse garantit l'endochronie des programmes décrits, qui est une condition suffisante pour que la désynchronisation des fragments répartis soit possible.

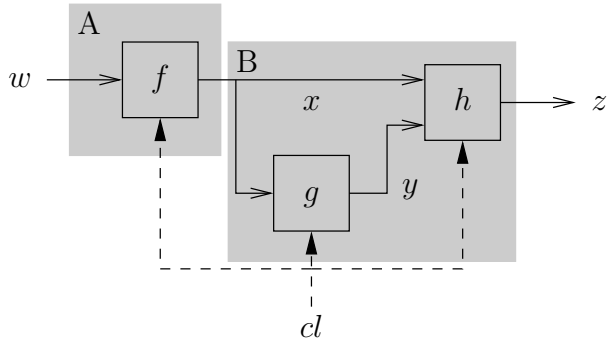
Nous nous intéressons dans cette thèse à la phase de répartition de programmes, dans un contexte où cette répartition n'est pas effectuée sur un réseau d'opérateurs mis à plat, mais sur un programme dont on souhaite conserver la structure fonctionnelle. Cette structure peut être indépendante de la structure de l'architecture (figure 2.4(a)). La répartition consiste alors à produire, pour chaque nœud du programme (ici f et g), un ensemble de nœuds spécialisés pour chacun des sites où le nœud comprend un calcul (figure 2.4(b)).

2.3.5 Compilation modulaire de programmes synchrones

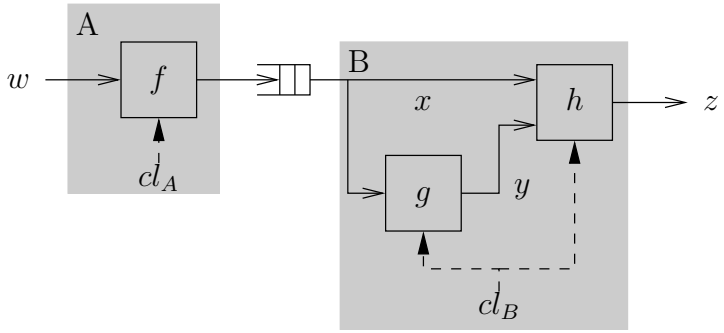
La compilation modulaire de programmes synchrones n'est pas aussi directe que celle des langages généralistes. Tout d'abord, l'instanciation d'un nœud, correspondant à un appel de fonction, ne correspond pas à un calcul unique, se terminant par le renvoi d'une valeur par la fonction, mais à l'instanciation d'un processus qui peut ne pas terminer, et qui sera exécuté en parallèle d'autres instanciations. Le code issu de la compilation d'un nœud devra donc n'exécuter qu'un pas de calcul, avant de rendre la main au reste du système, en gardant en mémoire l'état courant.



(a) Exécution centralisée par affectation d'une unique horloge aux nœuds

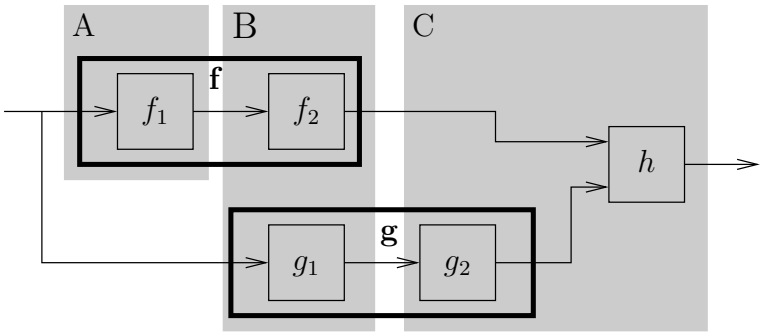


(b) Répartition du programme par choix de la localisation des opérateurs

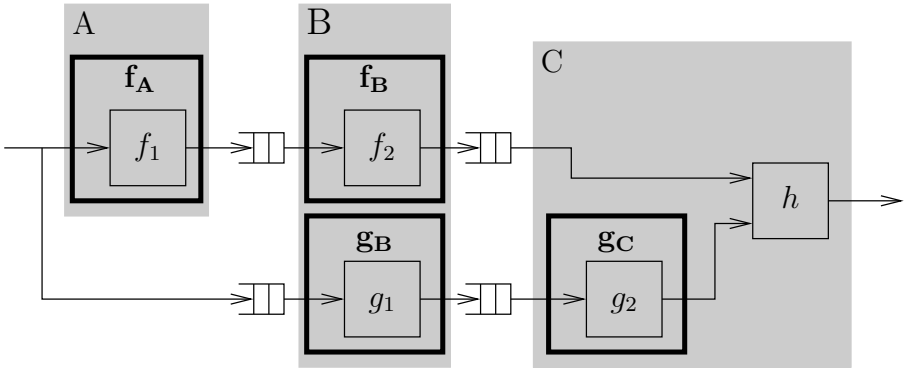


(c) Désynchronisation des fragments répartis

FIG. 2.3 – Conception de systèmes répartis au moyen de langages synchrones



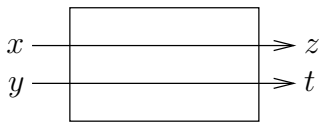
(a) Programme synchrone structuré



(b) Répartition et désynchronisation conservant la structure fonctionnelle

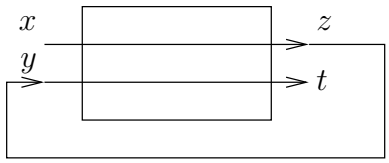
FIG. 2.4 – Répartition de programmes synchrones en conservant leur structure fonctionnelle

Ensuite, dans le cas général, la compilation modulaire n'est pas possible, pour des raisons de causalité. Dans sa thèse [42], Georges Gonthier a donné un exemple, repris en figure 2.5, qui illustre ce problème. Un nœud `f` est défini (figure 2.5(a)), comportant deux entrées `x` et `y`, et deux sorties `z` et `t`, définies respectivement par `x` et `y`. Le langage Lustre permet aux entrées d'un nœud de dépendre de ses sorties, ce qui autorise les instanciations décrites en figures 2.5(b) et 2.5(c) : ces deux programmes sont causaux, puisque ne comportant pas de cycle de dépendance. Dans le premier cas, l'entrée `y` dépend de la sortie `z`, ce qui nécessite que le calcul de cette sortie précède celui de `t`. Dans le deuxième cas, c'est l'entrée `x` qui dépend de `t` : le calcul de `t` doit précéder celui de `z`. Le nœud `f` ne peut donc être compilé séparément, puisque l'ordre des calculs ne peut être prédéterminé.




```
node f(x:int; y:int) returns (z:int; t:int);
let
  z = x;
  t = y;
tel
```

(a) Définition du nœud `f`.



`(z,t) = f(x,z)`

(b) Première instanciation.

`(z,t) = f(t,y)`

(c) Deuxième instanciation.

FIG. 2.5 – Problèmes de causalité pour la compilation modulaire de programmes synchrones.

Afin de traiter ce dernier problème, les langages synchrones Lustre [44] et Esterel [11] sont compilés par « inlining », c'est-à-dire par remplacement systématique des appels de nœuds ou de modules par leur code avant la compilation globale du programme « inliné ». L'approche modulaire est utilisée par la compilation du langage Lucid Synchrone, ou encore dans un cadre industriel par le compilateur SCADE. Dans ce cas, l'analyse de causalité est restreinte : les entrées d'un nœud ne peuvent pas dépendre de sa sortie, ce qui interdit les programmes 2.5(b) et 2.5(c).

Le langage Signal permet de traiter ce cas de manière modulaire sans restreindre l'analyse de causalité. L'approche polychrone de ce langage permet d'inférer des contraintes plus générales sur les horloges des calculs des nœuds, et ainsi

de compiler ceux-ci sous la forme de fonctions séparées pour les différentes horloges présentes. Ainsi, pour le nœud ci-dessus, les contraintes d'horloges inférées par Signal seraient :

- le flot de sortie `z` est de même horloge que le flot d'entrée `x` (contrainte notée `z ^= x` en Signal),
- le flot de sortie `t` est de même horloge que le flot d'entrée `y`.

Étant donné qu'il n'y a pas d'autres contraintes d'horloges, le compilateur peut produire deux fonctions différentes dans le langage cible (une par arbre de contraintes). La première fonction `f1` calculera l'équation `z = x`, la fonction `f2` calculera l'équation `t = y`. Enfin, le code séquentiel de l'instanciation de la figure 2.5(b) sera :

```
z := f1(x);
t := f2(z)
```

Le code séquentiel de l'instanciation de la figure 2.5(c) sera :

```
t := f2(y);
z := f1(t)
```

On peut remarquer que, malgré les éventuelles restrictions qu'elle apporte, l'approche modulaire permet de compiler, tester et simuler un programme synchrone en étant certain du fait que le code testé et simulé sera le même que le code réellement exécuté : il n'y a en effet aucune phase d'« inlining » intermédiaire, or cette phase peut s'avérer complexe et difficilement intelligible par le programmeur. Cette approche permet donc de répondre de manière plus stricte aux besoins de simulation et de vérification tels qu'énoncés à la section 2.3.1, tout en ajoutant un critère de traçabilité, essentiel dans le milieu industriel.

2.3.6 Ordre supérieur et reconfiguration dynamique

La compilation modulaire permet d'exprimer et de compiler des programmes d'ordre supérieur, c'est-à-dire au sein desquels les fonctions sont manipulées comme valeurs de bases : passées en arguments, ou encore communiquées d'une partie du programme à l'autre.

Ce caractère peut être utilisé pour l'expression de reconfiguration dynamique de programmes. La reconfiguration dynamique consiste à remplacer, au cours de l'exécution d'un système, une partie fonctionnelle de ce système par un nouveau programme, afin de changer sa fonctionnalité globale. La partie reconfigurable du système peut alors être vue comme l'exécution d'une fonction, elle-même considérée comme paramètre extérieur du système.

Cependant, pour utiliser ce caractère au sein d'un outil ou d'un langage intégré, il est nécessaire, dans un premier temps, de donner un cadre formel à la conception modulaire de systèmes répartis. Le travail présenté ici vise à la définition de ce

cadre formel, et d'une méthode de conception de systèmes répartis intégrée dans la compilation modulaire d'un langage de programmation.

2.4 Lucid Synchrone

Notre travail est basé sur le langage Lucid Synchrone. La méthode que nous décrivons a été implémentée dans le compilateur de ce langage. Nous donnons donc dans cette section une présentation générale de Lucid Synchrone, à travers des exemples. Ensuite, nous présenterons une formalisation d'un noyau du langage au moyen de sa sémantique opérationnelle. Ce noyau nous servira par la suite pour la formalisation de notre méthode de répartition. Les bases du langage ont été établies en [23] et un manuel complet est disponible en [2].

2.4.1 Présentation générale

Flots de données

Lucid Synchrone est un langage synchrone flot de données. À ce titre, les valeurs manipulées sont des flots, c'est-à-dire des séquences infinies de valeurs. Ainsi, la valeur `1` du langage n'est pas la valeur scalaire 1, mais le flot 1.1.1 . . . Le type `int` représente donc, de la même manière, le type des flots d'entiers.

Le caractère synchrone du langage est établi par le fait que les données que composent ces flots sont présentes à certains instants, et que ces instants définissent une échelle de temps discret. Les opérations binaires de base, arithmétiques et booléennes par exemple, sont appliquées point à point à deux flots dont les valeurs sont présentes aux mêmes instants. Par exemple, l'expression `x + y` définit un flot présent aux mêmes instants que `x` et `y`, et dont la valeur à chaque instant est la somme des valeurs portées par `x` et par `y` à cet instant.

`x`	x_0	x_1	x_2	...
`y`	y_0	y_1	y_2	...
`x + y`	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	...

Délais et mémoires

L'opérateur de délai `pre` est utilisé pour faire référence aux instants précédents des flots, et peut être vu comme un opérateur de mémorisation (ou une bascule dans un cadre circuits). Ainsi, le résultat de l'expression `pre` e est un flot ne portant aucune valeur au premier instant (symbolisé par la valeur spéciale *nil*), puis à chaque instant la valeur de e à l'instant précédent.

$$(\texttt{pre}\ e)_i = \begin{cases} nil & \text{si } i = 0 \\ e_{i-1} & \text{sinon} \end{cases}$$

L'opérateur d'initialisation `->` est utilisé pour initialiser un flot comportant une mémoire. Le résultat de e `->` e' est le flot dont la valeur au premier instant est la valeur au premier instant de e, puis aux instants suivants les valeurs de e'.

$$(e\ \texttt{->}\ e')_i = \begin{cases} e_0 & \text{si } i = 0 \\ e'_i & \text{sinon} \end{cases}$$

L'expression e `fby` e' est un raccourci pour e_1 `->` `pre` e_2.

`x`	x_0	x_1	x_2	...
`y`	y_0	y_1	y_2	...
`pre y`	nil	y_0	y_1	...
`x -> y`	x_0	y_1	y_2	...
`x fby y`	x_0	y_0	y_1	...

Ces opérateurs de délai peuvent être utilisés pour l'expression de flots récursifs, un flot récursif étant un flot dont l'expression fait référence à ses valeurs précédentes. Le flot `nat` ci-dessous est un exemple de flot récursif. Le mot-clé `rec` est utilisé pour introduire un ensemble d'équations mutuellement récursives.

$$\texttt{rec nat} = 0\ \texttt{->}\ \texttt{pre nat} + 1$$

Ce flot prend successivement pour valeurs les entiers naturels : il est initialisé à 0 au premier instant, puis sa valeur précédente est incrémentée de 1 à chaque instant.

`pre nat`	nil	0	1	2	...
`pre nat` $+ 1$	nil	1	2	3	...
0 `->` `pre nat` $+ 1$	0	1	2	3	...
`nat`	0	1	2	3	...

Fonctions combinatoires et séquentielles

Pour des raisons de compilation, le langage distingue les fonctions combinatoires, dont la sortie à un instant donné ne dépend que de la valeur des entrées à cet instant, et les fonctions séquentielles, ou *nœuds*, comportant de la mémoire.

La fonction `average` ci-dessous est une fonction combinatoire, dont le flot de sortie est la moyenne à chaque instant des valeurs des flots de ses entrées :

```
let average x y = z where
```

```
z = (x + y) / 2
```

Les opérations arithmétiques et booléennes de base (`+`, `*`, `or`,...) sont des fonctions combinatoires.

Les nœuds sont définis par le mot-clé `node`. Le nœud suivant `sum` prend comme entrée un flot d'entiers, et rend un flot dont la valeur à chaque instant est la somme des valeurs du flot d'entrée jusqu'à cet instant.

```
let node sum x = y where
  rec y = x -> pre y + x
```

Le compilateur, à l'aide d'un système d'inférence de types, renvoie au programmeur le type du nœud :

```
val sum : int -> int
```

En appliquant le nœud `sum` au flot `nat` défini dans la section précédente, on obtient le chronogramme :

`nat`	0	1	2	3	4	5	...
`sum nat`	0	1	3	6	10	15	...

Horloges

La notion d'horloge [19] permet d'exprimer le fait que les valeurs de deux flots de données peuvent ne pas être présentes aux mêmes instants. L'horloge d'un flot représente les instants pour lesquels une valeur est présente. Deux flots peuvent être combinés (par un opérateur arithmétique appliqué point-à-point par exemple) seulement s'ils ont la même horloge. Une horloge « lente » peut être définie à partir d'une horloge « plus rapide » cl et d'un flot booléen c dont l'horloge est cl (noté par la suite « $c :: cl$ »). Cette nouvelle horloge est appelée cl `on` c. Les valeurs d'un flot sur cette nouvelle horloge sont présentes aux instants où le flot c est présent et porte la valeur vraie. L'opérateur `when` permet d'échantillonner un flot d'horloge cl sur l'horloge c :

$x :: cl$	x_0	x_1	x_2	x_3	x_4	x_5
$c :: cl$	`true`	`true`	`false`	`true`	`false`	`false`
(x `when` c) $:: cl$ `on` c	x_0	x_1		x_3		

Un calcul d'horloge [30], basé sur un système d'inférence de types, permet d'inférer l'horloge d'un nœud, à partir des opérations d'échantillonnage utilisées dans ce nœud. Par exemple, pour le nœud `sum` ci-dessus, le compilateur renvoie l'horloge :

```
val sum :: 'a -> 'a
```

Le nœud `sum` prend en entrée un flot d'horloge `'a` (quel que soit `'a`), et renvoie en sortie un flot de même horloge `'a` : l'entrée et la sortie sont synchronisées, notamment par l'opérateur `(+)`. Cet opérateur prend en effet en entrée deux flots sur la même horloge, et renvoie un flot synchronisé sur cette même horloge :

```
val (+) :: 'a -> 'a -> 'a
```

Une horloge est définie, à partir d'un flot booléen, au moyen de la construction `let clock`. Par exemple, l'horloge `c` définie ci-dessous désigne les instants où le flot `x` est présent et porte une valeur positive :

```
let clock c = (x > 0)
```

L'horloge de `c` est la même que l'horloge de `x`.

L'opérateur `when` (et son complément `whenot`) permet d'échantillonner un flot sur une horloge. L'opérateur `merge` permet de combiner deux flots d'horloges complémentaires.

c	`true`	`true`	`false`	`false`	`true`	`false`	...
x	x_0	x_1	x_2	x_3	x_4	x_5	...
y	y_0	y_1	y_2	y_3	y_4	y_5	...
$x' = x$ `when` c	x_0	x_1			x_4		...
$y' = y$ `whenot` c			y_2	y_3		y_5	...
`merge` $c\ x'\ y'$	x_0	x_1	y_2	y_3	x_4	y_5	...

Nous pouvons donc définir le nœud suivant, qui fait la somme des carrés des valeurs positives de son entrée :

```
let sq x = x * x

let node positive_sqsum x = y where
  rec clock c = (x > 0)
  and y = (0 -> pre y) + merge c (sq (x when c)) 0

val sq : int -> int
val sq :: 'a -> 'a
val positive_sqsum : int -> int
val positive_sqsum :: 'a -> 'a
```

Quand l'horloge `c` est vraie, le carré de `x`, échantillonné sur `c`, est ajouté à la valeur précédente de `y`. L'opération `sq` n'est effectuée qu'aux instants où `c` est vraie : dans le cas contraire, c'est l'expression `0` qui est évaluée. À noter que cette dernière expression étant un flot constant, son échantillonnage sur `c` est implicite. L'expression `0` est équivalente à `0 whenot c`.

x	2	3	−1	0	1	2	...
c	true	true	false	false	true	true	...
x when c	2	3			1	2	...
sq(x when c)	4	9			1	4	...
0 -> pre y	0	4	13	13	13	14	...
positive_sqsum x	4	13	13	13	14	18	...

Le nœud suivant prend en paramètre une horloge `c`, un flot `x` échantillonné sur cette horloge, et une valeur par défaut. Ce nœud prolonge la valeur de `x` pendant les instants où `c` est fausse, c'est-à-dire les instants où `x` n'est pas présent :

```
let node hold c x xdef = y where
  rec y = merge c x (xdef -> pre y whenot c)

val hold : clock -> 'a -> 'a -> 'a
val hold :: (c:'a) -> 'a on c -> 'a on not c -> 'a
```

Dans l'horloge du nœud `hold` ci-dessus, l'horloge `(c:'a)` représente un flot booléen d'horloge `'a`, dont les instants vrais sont représentés par `c`.

c	false	true	true	false	true	false	true	...
x		x_0	x_1		x_2		x_3	...
d	d_0	...						
hold c x d	d_0	x_0	x_1	x_1	x_2	x_2	x_3	...

Un dernier exemple est l'opérateur d'activation `condact`. Cette opérateur, présent dans le langage SCADE, permet d'activer ou non, au moyen d'une horloge `c`, l'application d'une fonction `f` à son entrée. Quand l'horloge `c` est fausse, le résultat de l'application précédente est pris.

```
let node condact f c default x = y where
  rec y = merge c (f (x when c))
                  ((default -> pre y) whenot c)

val condact : ('a -> 'b) -> clock -> 'b -> 'a -> 'b
val condact :: ('a on _c0 -> 'a on _c0) -> (_c0:'a) -> 'a -> 'a -> 'a
```

x	x_0	x_1	x_2	x_3	x_4	x_5	...
c	false	true	true	false	true	false	...
d	d_0	...					
f(x)		y_1	y_2		y_4		...
condact f c d x	d_0	y_1	y_2	y_2	y_4	y_4	...

Horloges globales

Jusqu'ici, les horloges étaient définies et utilisées à l'intérieur des nœuds. Nous verrons au chapitre 7 que ces horloges locales sont difficiles à traiter du point de vue de la répartition. Nous introduisons donc ici un type particulier d'horloges, définies de manière globale. Seules ces horloges, ainsi que la structure `match/with` introduite dans la section suivante, seront traitées au cours de cette thèse. Ces horloges sont définies au niveau le plus haut du langage par la construction `let clock`, et ne dépendent d'aucun flot. Ces horloges peuvent donc être instanciées sur n'importe quelle autre horloge.

Par exemple, l'horloge `half` ci-dessous est définie à partir d'un flot `h` vrai un instant sur deux :

```
let clock half = h where
  rec h = true fby not h

val half : bool
val half :: 'a
```

L'horloge polymorphe `'a` signifie que l'horloge `half` peut être utilisée à n'importe quel rythme.

À l'aide de cette horloge globale `half`, et du nœud `hold` défini plus haut, on peut définir une opération de sur-échantillonnage `over` :

```
let node over x = hold half x 0

val over : int -> int
val over :: 'a on half -> 'a
```

Ce nœud prend en paramètre un flot `x` $= x_0, x_1, \ldots$ sur l'horloge α `on half` (α étant une horloge quelconque), et renvoie un flot `y` $= y_0, y_1, \ldots$ sur l'horloge α, tel que $y_{2n} = y_{2n+1} = x_n$.

half	true	false	true	false	true	false	...
x	x_0		x_1		x_2		...
over x	x_0	x_0	x_1	x_1	x_2	x_2	...

L'utilisation d'horloges globales permet d'itérer un calcul « long » sur plusieurs pas de temps. Ainsi, on peut calculer l'élévation des valeurs successives d'un flot à la puissance 5, en effectuant une unique multiplication à chaque pas de temps. On commence pour cela par définir l'horloge `four`, qui sera vraie tous les quatre instants :

```
let node sample n = ok where
  rec cpt = 1 -> if pre cpt = n then 1 else pre cpt + 1
```

```
  and ok = cpt = 1

let clock four = sample 4

val sample : int -> bool
val sample :: 'a -> 'a
val four : bool
val four :: 'a
```

On utilise ensuite cette horloge pour définir le nœud `power`. Ce nœud prend en entrée un flot `x` d'horloge `four`. Ce flot est ensuite sur-échantillonné (pour donner le flot `i`), et ce sur-échantillon est successivement accumulé par multiplication avec le flot `o`. Tous les quatre instants, le flot `o` contient donc la valeur précédente de `x`, élevée à la puissance 5 : ce flot est à nouveau échantillonné sur l'horloge `four`.

```
let node power x = y where
  rec i = merge four x ((1 -> pre i) whenot four)
  and o = 1 -> pre (i * merge four x (o whenot four))
  and y = o when four

val power : int -> int
val power :: 'a on four -> 'a on four
```

four	t	f	f	f	t	f	f	f	t	f	...
x	x_0				x_1				x_2		...
i	x_0	x_0	x_0	x_0	x_1	x_1	x_1	x_1	x_2	x_2	...
o	1	x_0^2	x_0^3	x_0^4	x_0^5	x_1^2	x_1^3	x_1^4	x_1^5	x_2^2	...
power x	1				x_0^5				x_1^5		...

Enfin, le calcul d'horloge permet de rejeter à la compilation les programmes ne pouvant pas être exécuté de manière synchrone : par exemple, l'expression `x` + (`x when half`). Cette dernière expression est en effet impossible à exécuter en mémoire bornée. Son réseau de Kahn nécessite en effet une file d'attente de longueur infinie pour accumuler les valeurs de `x`, qui arrivent deux fois plus vite que les valeurs de `x when half`.

Contrôle

L'expression conditionnelle `if/then/else` n'est pas une structure de contrôle, mais un opérateur conditionnel sur des flots présents aux mêmes instants. Dans l'expression `if` e_1 `then` e_2 `else` e_3, les trois expressions *réagissent aux mêmes*

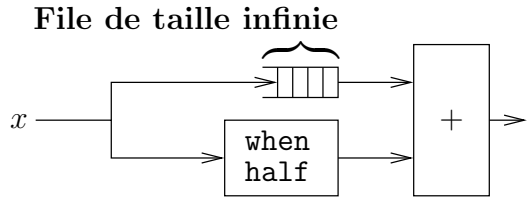


FIG. 2.6 – Réseau de Kahn représentant l'expression `x+(x when half)`.

instants. Le résultat de cette expression est alors, selon la réaction de e_1, le résultat de la réaction de e_2 ou de celle de e_3.

On peut par exemple donner une implémentation naïve, en utilisant la structure `if/then/else`, du nœud `positive_sqsum` de la section précédente :

```
let sq x = x * x

let node positive_sqsum x = y where
  rec y = (0 -> pre y) + (if (x > 0) then (sq x) else 0)

val sq : int -> int
val sq :: 'a -> 'a
val positive_sum : int -> int
val positive_sum :: 'a -> 'a
```

Le flot résultant est le même, mais dans ce cas la fonction combinatoire `sq` est évaluée à chaque instant, même quand sa valeur de retour n'est pas utilisée :

`x`	2	3	-1	0	1	2	...
`sq x`	4	9	1	0	1	4	...
`if (x > 0) then (sq x) else 0`	4	9	0	0	1	4	...
`0 -> pre y`	0	4	13	13	13	14	...
`positive_sum x`	4	13	13	13	14	18	...

En Lucid Synchrone (et les autres langages flots de données synchrones tels que Lustre et Signal), ce sont les horloges qui permettent d'exprimer un certain contrôle sur la réaction des expressions du programme. Une opération ne sera ainsi pas calculée à un instant où ses composantes ne sont pas présentes.

Grégoire Hamon a utilisé ce mécanisme d'horloges pour ajouter des structures de contrôle au langage [48, 47]. Ainsi, lors de l'exécution du nœud ci-dessous, les expressions `f(x)` et `g(x)` sont évaluées à des instants complémentaires, selon la valeur du flot booléen `c` à chacun de ces instants : de même, l'évolution de l'état de chacune de ces expressions n'est effectuée qu'à ses instants de réactions.

```
let node m c x = y where
```

```
  match c with
  | true -> do y = f(x) done
  | false -> do y = g(x) done
  end

val m : bool -> int -> int
val m :: 'a -> 'a -> 'a
```

x	x_0	x_1	x_2	x_3	x_4	x_5	...
c	true	true	false	true	false	true	...
f(x)	y_0	y_1		y_3		y_5	...
g(x)			y_2		y_4		...
m(c, x)	y_0	y_1	y_2	y_3	y_4	y_5	...

Ce nœud `m` est équivalent, sans la structure `match/with`, au nœud suivant :

```
let node m c x = y where
  y = merge c (f(x when c)) (g(x whenot c))
```

Cette structure de contrôle permet donc de rendre implicite l'utilisation d'une horloge définie à partir d'un flot booléen, ainsi que l'échantillonnage des entrées et la combinaison des variables définies dans chaque branche.

À partir de ce travail, une construction d'automates hiérarchiques a ensuite été ajoutée [29]. Par exemple, le nœud `incr_decr` comprend deux états, `Incr` et `Decr`. Dans l'état `Incr`, la variable `y` définissant la sortie du nœud est incrémentée ; elle est décrémentée dans l'état `Decr`. Le mot-clé `last` permet de récupérer la dernière valeur d'une variable partagée par plusieurs états, quel que soit l'état dans lequel l'automate était à l'instant précédent. L'entrée `switch` du nœud permet de passer d'un état à l'autre.

```
let node incr_decr switch = y where
  rec automaton
      | Incr -> do y = 0 -> last y + 1
      until switch then Decr
      | Decr -> do y = last y - 1
      until switch then Incr
      end

val incr_decr : bool -> int
val incr_decr :: 'a -> 'a
```

switch	f	f	f	f	f	t	f	f	t	f	f	...
incr_decr switch	0	1	2	3	4	3	2	1	2	3	4	...

De même que pour le `match/with`, cette construction rend implicite l'échantillonnage des entrées dans chacun des états, la combinaison des sorties, ainsi que la définition d'une variable d'état. Ce nœud peut être programmé en utilisant explicitement une horloge :

```
let node incr_decr switch = y where
  rec clock state = true -> pre next_state
  and next_state =
        merge state
              (if (switch when state) then false else true)
              (if (switch whenot state) then true else false)
  and last_y = 0 -> pre y
  and y = merge state
                ((last_y when state) + 1)
                ((last_y whenot state) - 1)
```

Ordre supérieur

L'ordre supérieur est, avec l'inférence de types et le polymorphisme, un caractère hérité des langages fonctionnels de type ML [64]. Lucid Synchrone est un langage d'ordre supérieur, ce qui signifie que les fonctions (séquentielles et combinatoires) sont des valeurs à part entières, et peuvent être manipulées comme telles, par exemple en entrées ou en sorties d'autres fonctions.

L'ajout de ce caractère dans un langage dans lequel les valeurs sont des flots impose de faire la différence entre ordre supérieur *statique* et ordre supérieur *dynamique*. L'ordre supérieur statique consiste à considérer les fonctions ou les nœuds comme des flots statiques, c'est-à-dire constants, qui n'évoluent pas dans le temps. On peut ainsi paramétrer un nœud par un autre nœud. L'exemple ci-dessous est un nœud prenant en paramètre une fonction binaire `f`, et itérant cette fonction successivement sur la valeur courante de son deuxième paramètre `x` et la valeur précédente de sa sortie.

```
let node iter f init x = y where
  rec y = f (x, init -> pre y)
```

Ce nœud générique, instancié par exemple avec la fonction `(+)`, permet de programmer le nœud `sum` de la section 2.4.1 :

```
let node sum x = iter (+) 0 x
```

L'ordre supérieur dynamique permet d'exprimer des *flots de nœuds* dont la valeur peut évoluer dans le temps. Contrairement aux flots de fonctions combinatoires, l'expression de flots de fonctions séquentielles pose deux problèmes majeurs

exposés en [27] : l'utilisation de variables libres dans les fonctions composant ces flots, et la préservation de la mémoire d'une instanciation à l'autre. L'ordre supérieur dynamique n'est pas traité du point de vue de la répartition. Nous reviendrons sur l'intérêt qu'il représente, et nous présenterons à cette occasion la manière dont ce problème est traité en Lucid Synchrone, au chapitre 9 (en section 9.4).

2.4.2 Formalisation du langage

Par la suite, pour la formalisation de la méthode de répartition, la syntaxe concrète complète est simplifiée en une syntaxe formelle plus réduite. On considère un sous-ensemble strict du langage Lucid Synchrone. Cette simplification permet d'exprimer de manière plus simple et plus directe des opérations définissant cette méthode. Notamment, la distinction entre les fonctions combinatoires et les nœuds (fonctions séquentielles), nécessaire pour la compilation efficace du langage, est ici ignorée. La curryfication et l'ordre supérieur dynamique ne sont pas traités. Les horloges, et les opérateurs d'échantillonnage et de combinaison ne sont pas traités : leur traitement dans le cadre de la répartition est discuté au chapitre 7, consacré à l'introduction du contrôle.

La syntaxe du noyau que nous utiliserons par la suite est la suivante :

$$
\begin{array}{lcl}
P & ::= & d_1\,;\ldots;d_n\,;D \\
d & ::= & \texttt{node}\ f(x) = e\ \texttt{where}\ D \\
D & ::= & \epsilon \mid p = e \mid x = x(e) \mid D\ \texttt{and}\ D \mid \texttt{let}\ D\ \texttt{in}\ D \\
e & ::= & i \mid x \mid (e,e) \mid \mathrm{op}(e,e) \mid e\ \texttt{fby}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\mathrm{op} & ::= & \mathrm{fby}_v \mid \mathrm{ifte} \mid (+) \mid (\&) \mid \ldots \\
p & ::= & (x,\ldots,x) \mid x \\
i & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid \ldots
\end{array}
$$

Un programme P est constitué d'une suite de définition de nœuds ($d_1\,;\ldots;d_n$), suivie d'un ensemble d'équations D. Cet ensemble d'équations constitue le programme principal : les variables libres de ces équations sont les entrées du programme, les variables qui y sont définies constituent ses sorties. Les nœuds $d_1\,;\ldots;d_n$ sont définis en séquence : la portée d'un nœud d_i s'étend à tous les nœuds définis après d_i, ainsi qu'au programme principal.

Un nœud f est défini par $\texttt{node}\ f(x) = e\ \texttt{where}\ D$, où x constitue l'entrée de f, e sa sortie (qui peut éventuellement être un tuple), et D un ensemble d'équations dans la portée desquelles la sortie e est évaluée. Les noms introduits par p sont dans la portée de D et de e, et sont donc distincts des noms introduits par D.

Un ensemble d'équations D peut être vide (ϵ), ou une équation simple ($p = e$), ou une application d'un nœud à une expression ($x = x(e)$: le résultat de l'application définit immédiatement une variable, afin de rendre la sémantique opérationnelle plus simple), ou deux ensembles d'équations en parallèle (D `and` D), ou

encore un ensemble d'équations définies dans la portée d'un ensemble d'équations locales (`let` D `in` D).

Une expression e peut être soit une valeur immédiate i (une constante booléenne ou scalaire), l'instanciation d'une variable x, la construction d'une paire $((e, e))$ ou l'application d'un opérateur $(\mathrm{op}(e))$.

Un opérateur op est soit l'opérateur de délai fby_v (noté v `fby` $\cdot$, v étant la valeur d'initialisation de cet opérateur), l'opérateur conditionnel ifte (noté `if` $\cdot$ `then` $\cdot$ `else` $\cdot$), ou un opérateur binaire arithmétique ou booléen (par exemple, l'opérateur $(+)$, le « et » booléen &, etc.).

La construction de paires est associative à gauche : on notera donc $e_1, \ldots, e_n$ l'expression $(\ldots(e_1, e_2), e_3), \ldots), e_n)$.

Remarque 1. *Nous utilisons dans les exemples une syntaxe plus expressive, permettant d'utiliser plus largement les motifs* $(x_1, \ldots, x_n)$ *pour lier des variables dans un programme :*

$$
\begin{array}{lcl}
P & ::= & d_1 \textbf{;} \ldots \textbf{;} d_n \textbf{;} D \\
d & ::= & \epsilon \mid \textbf{node}\ f(p) = e\ \textbf{where}\ D \\
D & ::= & \epsilon \mid p = e \mid p = x(e) \mid D\ \textbf{and}\ D \mid \textbf{let}\ D\ \textbf{in}\ D \\
e & ::= & i \mid x \mid (e, e) \mid \mathrm{op}(e) \\
\mathrm{op} & ::= & \mathrm{fby}_v \mid \mathrm{ifte} \mid (+) \mid (-) \mid \ldots \\
p & ::= & (x, \ldots, x) \mid x \\
i & ::= & \textbf{true} \mid \textbf{false} \mid 0 \mid 1 \mid \ldots
\end{array}
$$

Un programme P exprimé dans cette dernière syntaxe peut être réécrit, afin d'en obtenir un équivalent dans la syntaxe du noyau. Cette réécriture est définie par l'opérateur $P \Rightarrow_V P'$, où V est l'ensemble des variables liées dans P'. On note $V_1 \uplus V_2$ l'union disjointe d'ensemble de variables, définie ssi $V_1 \cap V_2 = \emptyset$.

On note **var** (p) *l'ensemble des variables liées par le motif p :*

$$
\begin{array}{lcl}
\textbf{var}\ (x) & = & \{x\} \\
\textbf{var}\ (x_1, x_2) & = & \{x_1, x_2\}
\end{array}
$$

Cet opérateur est étendu aux nœuds et aux équations, et défini ainsi :

$$\frac{d_i \Rightarrow_{V_i} d'_i \qquad D \Rightarrow_V D' \qquad V' = V_1 \uplus \ldots \uplus V_n \uplus V}{d_1 \texttt{;} \ldots \texttt{;} d_n \texttt{;} D \Rightarrow_{V'} d'_1 \texttt{;} \ldots \texttt{;} d'_n \texttt{;} D'}$$

$$\frac{D \Rightarrow_V D' \qquad V' = V \uplus \texttt{var}\ (p) \uplus \{x\}}{\texttt{node}\ f(p) = e\ \texttt{where}\ D \Rightarrow_{V'} \texttt{node}\ f(x) = e\ \texttt{where}\ D\ \texttt{and}\ p = x}$$

$$p = e \Rightarrow_\emptyset p = e \qquad \frac{x' \notin \texttt{var}\ (p)}{p = x(e) \Rightarrow_{\{x'\}} x' = x(e)\ \texttt{and}\ p = e}$$

$$\frac{D_1 \Rightarrow_{V_1} D'_1 \qquad D_2 \Rightarrow_{V_2} D'_2 \qquad V = V_1 \uplus V_2}{D_1\ \texttt{and}\ D_2 \Rightarrow_V D'_1\ \texttt{and}\ D'_2}$$

$$\frac{D_1 \Rightarrow_{V_1} D'_1 \qquad D_2 \Rightarrow_{V_2} D'_2 \qquad V = V_1 \uplus V_2}{\texttt{let}\ D_1\ \texttt{in}\ D_2 \Rightarrow_V \texttt{let}\ D'_1\ \texttt{in}\ D'_2}$$

2.4.3 Sémantique comportementale

Cette section décrit de manière formelle l'exécution d'un programme écrit dans le langage donné en section précédente. La sémantique présentée ici est basée sur celle, plus complète, présentée en [27]. La sémantique d'un programme est donnée par la séquence de valeurs émises par ce programme, en fonction de la séquence de valeurs données en entrée.

Une valeur v est soit une valeur immédiate i, soit un couple de valeurs (v, v). Les valeurs de fonctions sont entendues à $\alpha-$renommage près. Un environnement de réaction R associe une valeur, ou une fonction $\lambda x.e$ `where` D, à chaque variable de l'environnement.

$$\begin{array}{lcl} v & ::= & i \mid (v, v) \\ rv & ::= & v \mid \lambda x.e\ \texttt{where}\ D \\ R & ::= & [rv_1/x_1, \ldots, rv_n/x_n] \\ & & \text{si } \forall i \neq j, x_i \neq x_j \\ S & ::= & R_1 \ldots . R_n \ldots \end{array}$$

On note dans la suite R, R' la concaténation des environnements R et R'. Celle-ci est définie seulement si leurs domaines sont disjoints. S dénote une séquence d'environnements de réaction, et $R.S$ l'ajout de l'environnement R à gauche de la séquence S.

L'exécution d'un programme synchrone P, dans une séquence S d'environnement de réaction ($S = R_1.R_2 \ldots$) représentant les entrées du programme, est

définie par une séquence S' d'environnement de réaction ($S' = R'_1.R'_2.\ldots$) représentant ses sorties. Cette exécution est décrite par une relation notée $S \vdash P : S'$. Les définitions $d_i = \texttt{node}\ f_i(x_i) = e_i\ \texttt{where}\ D_i$ définissent des valeurs $\lambda x_i.e_i\ \texttt{where}\ D_i$. Ces définitions sont globales, on considère implicitement qu'elles définissent un environnement initial R_d tel que pour tout i, $R_d(f_i) = \lambda x_i.e_i\ \texttt{where}\ D_i$.

$$\frac{R_d, R_{in}, R_{out} \vdash D \xrightarrow{R_{out}} D' \qquad S_{in} \vdash (d_1\,;\ldots;d_n\,;D') : S_{out}}{R_{in}.S_{in} \vdash (d_1\,;\ldots;d_n\,;D) : R_{out}.S_{out}}$$

La sémantique d'un programme est définie par la relation entre les séquences d'entrées et de sorties : à chaque instant, la relation entre l'environnement d'entrée R_{in} et de sortie R_{out} est établie par le prédicat $R \vdash D \xrightarrow{R'} D'$, où R est la concaténation de R_d, R_{in} et R_{out} (les équations étant mutuellement récursives). R_d est un environnement de réaction initial, dans lequel toutes les réactions sont effectuées, et contenant les définitions des nœuds. L'ensemble d'équations D' est utilisé pour la suite de l'exécution du programme.

La réaction d'un ensemble d'équations D est définie par une émission de valeurs par ces équations, et par la réécriture de cet ensemble d'équations pour la réaction à l'instant suivant. De même, la réaction d'une expression e est définie par l'émission d'une valeur par cette expression, et par la réécriture de cette expression pour la réaction suivante. À chaque instant, une expression émet une valeur, et un ensemble d'équations émet un environnement de réaction associant une valeur à chaque variable définie par ces équations. Ces réactions sont définies par les trois prédicats de réaction :

- $\mathrm{op}(v) \xrightarrow{v'} \mathrm{op}'$: l'opérateur op, auquel est appliqué la valeur v, émet la valeur v', et se réécrit en l'opérateur op$'$ pour la réaction à l'instant suivant ;
- $R \vdash e \xrightarrow{v} e'$: dans l'environnement de réaction R, l'expression e émet la valeur v, et se réécrit en l'expression e' pour la réaction à l'instant suivant ;
- $R \vdash D \xrightarrow{R'} D'$: dans l'environnement de réaction R, l'ensemble d'équations D émet l'environnement de réaction R', et se réécrit en l'ensemble d'équations D' pour la réaction à l'instant suivant.

Cette définition sous forme d'émission/réécriture permet d'exprimer le caractère itératif de l'exécution d'un programme flot de données (une émission par instant), et les délais par la mémorisation dans la réécriture de valeurs pour l'instant suivant.

La figure 2.7 donne les règles définissant la sémantique des opérateurs de base :

- Règle Op-Fby : l'opérateur fby_{v_1} est l'opérateur de délai initialisé avec la valeur v_1. Appliqué à une valeur v_2, cet opérateur émet la valeur v_1, et se réécrit en l'opérateur fby_{v_2} pour la réaction suivante.
- Règle Op-Plus : les opérateurs binaires arithmétiques ou booléens sont appliqués point-à-point sur des valeurs immédiates (i_1, i_2).

– Règles Op-If-True et Op-If-False : si la valeur conditionnelle est vraie, la première valeur v_1 est émise, v_2 sinon. L'opérateur se réécrit en lui-même : cette configuration correspond à une application point-à-point de l'opérateur, sans mémorisation.

(Op-Fby)
$\text{fby}_{v_1}(v_2) \xrightarrow{v_1} \text{fby}_{v_2}$

(Op-Plus)
$(+)(i_1, i_2) \xrightarrow{i_1+i_2} (+)$

(Op-If-True)
$\text{ifte}(\texttt{true}, v_1, v_2) \xrightarrow{v_1} \text{ifte}$

(Op-If-False)
$\text{ifte}(\texttt{false}, v_1, v_2) \xrightarrow{v_2} \text{ifte}$

Fig. 2.7 – Sémantique des opérateurs de base

La figure 2.8 donne les règles définissant les prédicats de réaction :
– Règle Imm : une valeur immédiate, constante, émet sa valeur, et se réécrit en elle-même (créant ainsi un flot constant).
– Règle Inst : une variable émet sa valeur courante définie par l'environnement de réaction, et se réécrit en elle-même.
– Règle Op : un opérateur est appliqué sur la valeur émise dans l'instant par l'expression opérande. La réaction suivante est définie par la réécriture de l'opérateur et de son opérande.
– Règle Pair : la valeur émise est le couple formé des valeurs émises par les composantes de l'expression.
– Règle Def : une équation $p = e$ émet l'environnement de réaction définissant les variables du motif p avec la valeur émise par e.
– Règle App : au premier instant, l'application d'un nœud est remplacée par la définition locale de ses entrées. Le corps du nœud réagit dans l'environnement de l'application.
– Règle And : deux ensembles d'équations en parallèle sont mutuellement récursives. Les équations de D_2 réagissent dans l'environnement courant, concaténé à l'environnement de réaction émis par la réaction de D_1.
– Règle Let : les équations de D_2 réagissent dans l'environnement courant, concaténé à l'environnement de réaction émis par la réaction de D_1.

Remarque 2. *Tous les programmes ne réagissent pas avec cette sémantique : un système de types, un calcul d'horloge et une analyse de causalité permettent de spécifier quels sont les programmes acceptés par cette sémantique (programmes bien typés, synchrones, sans boucle de causalité). Ces analyses ne sont pas présentées ici : on ne considère pour la suite que les programmes synchrones réagissant avec cette sémantique.*

(IMM)
$$R \vdash i \xrightarrow{i} i$$

(INST)
$$R, [v/x] \vdash x \xrightarrow{v} x$$

(OP)
$$\frac{R \vdash e \xrightarrow{v} e' \qquad \mathrm{op}(v) \xrightarrow{v'} \mathrm{op}'}{R \vdash \mathrm{op}(e) \xrightarrow{v'} \mathrm{op}'(e')}$$

(PAIR)
$$\frac{R \vdash e_1 \xrightarrow{v_1} e_1' \qquad R \vdash e_2 \xrightarrow{v_2} e_2'}{R \vdash (e_1, e_2) \xrightarrow{(v_1,v_2)} (e_1', e_2')}$$

(DEF)
$$\frac{R \vdash e \xrightarrow{(v_1,\ldots,v_n)} e'}{R \vdash (x_1, \ldots, x_n) = e \xrightarrow{[v_1/x_1,\ldots,v_n/x_n]} (x_1, \ldots, x_n) = e'}$$

(APP)
$$\frac{R(f) = \lambda y.e \ \texttt{where}\ D \qquad R \vdash \texttt{let}\ y = e'\ \texttt{and}\ D\ \texttt{in}\ x = e \xrightarrow{R'} D'}{R \vdash x = f(e') \xrightarrow{R'} D'}$$

(AND)
$$\frac{R, R_2 \vdash D_1 \xrightarrow{R_1} D_1' \qquad R, R_1 \vdash D_2 \xrightarrow{R_2} D_2'}{R \vdash D_1\ \texttt{and}\ D_2 \xrightarrow{R_1,R_2} D_1'\ \texttt{and}\ D_2'}$$

(LET)
$$\frac{R \vdash D_1 \xrightarrow{R_1} D_1' \qquad R, R_1 \vdash D_2 \xrightarrow{R_2} D_2'}{R \vdash \texttt{let}\ D_1\ \texttt{in}\ D_2 \xrightarrow{R_2} \texttt{let}\ D_1'\ \texttt{in}\ D_2'}$$

FIG. 2.8 – Sémantique opérationnelle centralisée.

Remarque 3. *La forme de la règle d'application des nœuds, due à la nécessité de conserver l'état interne de l'instance d'un nœud appliqué d'un instant à l'autre, rend nécessaire le fait que les noms de variables soient différents deux à deux au sein d'un même environnement de réaction. Dans le cas contraire, des problèmes de capture de noms peuvent apparaître :*

```
node f(x) = y where y = x + 1;

node g(x) = y where y = f(x);

node h(x) = y where
      f = x - 1
  and y = g(f)
```

Le corps de **h** *serait évalué dans un environnement de réaction de la forme :*

$$\left[\begin{array}{l} \lambda x.y \ \textbf{where}\ y = x + 1/f, \\ \lambda x.y \ \textbf{where}\ y = f(x)/g, \ldots \end{array}\right]$$

L'application de la règle App *de la sémantique réécrit le corps de* **h** *(après $\alpha-$renommage de* **x** *en* **x'** *et* **y** *et* **y'***) en :*

```
      f = x - 1
  and let x' = f and y' = f(x') in y = y'
```

Cette réécriture modifie le sens du programme. La sémantique impose donc de renommer **f** *dans* **h***, afin d'éviter le masquage de* **f** *(cette contrainte étant assurée par typage) :*

```
node h(x) = y where
      f' = x - 1
  and y = g(f')
```

Le corps de **h** *est alors réécrit en :*

```
      f' = x - 1
  and let x' = f' and y' = f(x') in y = y'
```

Cette réécriture conserve le sens du programme.

2.5 La radio logicielle

L'exemple de motivation introduit ici servira de fil conducteur tout au long de l'exposé de ces travaux. Il s'agit d'un exemple de programmation d'un canal de réception multiple d'une radio logicielle. Une radio logicielle est une radio dont les composants (décodage du signal radio, conversion analogique/numérique,...), plutôt que d'être implantés au moyen de composants matériels spécifiques, sont

des logiciels exécutés sur des plateformes matérielles composées de composants matériels non spécifiques (processeurs, circuits reconfigurables) [65]. Cette conception logicielle sur une architecture non spécifique permet de réutiliser cette architecture pour la conception d'autres radios, et la reconfiguration, éventuellement dynamique, de la radio implantée. Cette technologie est donc adaptée et utilisée pour la conception de systèmes radio embarqués tels que les stations relais et les terminaux de téléphonie mobile [52].

Un canal de réception est usuellement composé de trois traitements : un filtre passe-bande permettant de sélectionner le signal porteur parmi un ensemble de fréquences, un composant de démodulation permettant d'extraire l'information reçue du signal porteur, et un traitement ultérieur de cette information (décodage, correction d'erreur, etc.). Pour simplifier, ne seront ici considérés que les deux premiers traitements.

Pour des raisons de performances, chacun de ces traitements est effectué sur un composant matériel spécialisé : le filtre passe-bande sur un FPGA[2], et le composant de démodulation sur un processeur spécialisé dans le traitement du signal[3].

La conception d'un tel système par des méthodes usuelles (langages de description d'architectures, et conception séparée des éléments d'architecture) aboutirait à la programmation séparée de ces deux traitements. Or, l'enjeu de la radio logicielle est de pouvoir, de manière plus simple que par conception matérielle, implanter sur un système plusieurs standards radio, ou de manière générale plusieurs fonctionnalités [53]. Ces fonctionnalités sont ensuite destinées à être utilisées en parallèle, ou activées ou chargées dynamiquement. En d'autres termes, la motivation principale du principe de radio logicielle est le *contrôle dynamique*.

L'exemple étudié ici sera un système logiciel de réception radio à canaux multiples supportant les standards de téléphonie mobile GSM et UMTS.

- Le standard GSM est implanté par un filtre passe-bande de fréquences de 1800 MHz et un démodulateur GMSK ;
- le standard UMTS par un filtre de fréquences de 2 GHz et un démodulateur QPSK.

La figure 2.9 montre l'implémentation de cette radio logicielle sur un système composé de deux composants matériels : un FPGA pour l'exécution des deux filtres passe-bandes, et un processeur de traitement du signal (DSP) pour les fonctions de démodulation. Ce système a une entrée `x`, le signal radio non filtré et modulé provenant de l'antenne. La sortie `y` est le signal filtré et démodulé. Les transitions d'un canal à l'autre expriment le contrôle déterminant à chaque instant lequel de ces deux canaux est utilisé. Ce contrôle prend ici la forme d'une fonction `g`, qui détermine le canal à utiliser en fonction de la valeur du signal démodulé. La

[2]Field Programmable Gate Array, circuit programmable.
[3]DSP pour Digital Signal Processor

localisation de cette dernière fonction n'est pas déterminée *a priori*.

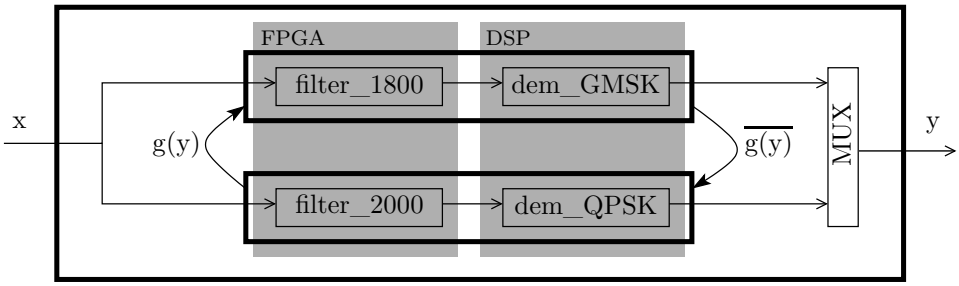


FIG. 2.9 – Modèle fonctionnel d'un système de réception radio logicielle à canaux multiples.

De manière usuelle, la conception de cette radio logicielle impliquerait la conception séparée de ces deux composants matériels. Cette conception séparée amènerait deux problèmes majeurs.

1. Le premier problème provient de la nécessité de contrôle dynamique. Du fait de la conception séparée, il n'y a aucune garantie que les composants interagissent comme spécifié : c'est-à-dire, que le filtre 1800 MHz soit exécuté en même temps que le démodulateur GMSK. Pour cela, il est nécessaire de dupliquer la fonction de contrôle (matérialisée par le multiplexeur MUX sur la figure) sur les deux composants matériels. Cette duplication compromet la modularité fonctionnelle du système.
2. Le deuxième problème relève de la cohérence imposée par la modularité fonctionnelle. Concevoir les deux composants matériels de manière indépendante mène à la programmation séparée de composants logiciels fonctionnellement proches (par exemple, les fonctions de filtre et de démodulation d'un même canal de réception).

La modularité fonctionnelle impose donc de concevoir (puis vérifier, tester et simuler) indépendamment chaque canal de réception, plutôt que de concevoir indépendamment chaque composant matériel. Cette situation montre clairement l'intérêt d'une approche orientée langage, permettant d'exprimer conjointement, au sein d'un même programme, des aspects fonctionnels et des aspects de répartition. Cette approche orientée langage implique d'ajouter dans un langage existant des primitives permettant, d'une part, de décrire l'architecture ou son modèle visé par le système, et d'autre part, la localisation de parties du programme séparément de la description de cette architecture.

Une telle approche intégrée permet ainsi d'effectuer les tests, simulations et vérifications sur le programme global. La répartition automatique permet de garantir la sûreté de fonctionnement du programme réparti, en garantissant l'équivalence

entre la sémantique centralisée du programme global, c'est-à-dire sans tenir compte des directives de répartition, et sa sémantique répartie. Par exemple, la cohérence des types de données communiquées entre les sites est assurée par le fait que la vérification de cohérence des données du programme par typage est faite de manière globale. Les incohérences dues, par exemple, à la sérialisation des données peuvent ainsi être détectées à la compilation.

La figure 2.10 donne l'implémentation centralisée de ce canal de réception multiple. L'extension du langage aura ici pour but de déclarer l'existence de deux ressources (le FPGA et le DSP), l'implémentation du canal de réception `channel` supposant l'exécution de la fonction `filter` sur le FPGA, et `demod` sur le DSP.

```
node filter_1800 = ...

node filter_2000 = ...

node demod_gmsk = ...

node demod_qpsk = ...

node channel(filter,demod,x) = y where
      f = filter(x)
  and y = demod(f)

node multichannel_sdr(x) = y where
      c = g(y)
  and match (true fby c) with
      | true -> do y = channel(filter_1800,demod_gmsk,x) done
      | false -> do y = channel(filter_2000,demod_qpsk,x) done
      end
```

FIG. 2.10 – Implémentation du système de canal de réception multiple d'une radio logicielle.

2.6 État de l'art

2.6.1 Conception de systèmes répartis

Nous présentons ici les diverses approches orientées langages pour la programmation de systèmes répartis.

Les langages de programmation généraux (C, Ada, Java) sont pourvus de librairies permettant la programmation répartie. La répartition n'est cependant pas automatique, dans le sens où la conception d'un système réparti au moyen de ces langages induit une première phase de répartition manuelle du système, suivie de la conception séparée de chaque ressource. Ces méthodes permettent une intégration aisée des fragments ainsi conçus, la communication entre ressources se faisant au moyen de librairies standardisées. Parmi les méthodes les plus récentes, on peut citer l'API Java RMI[4], qui permet l'invocation distante de méthodes, en faisant abstraction de la technique sous-jacente (connections TCP/IP). Le standard CORBA[5] [77] permet l'intégration de systèmes hétérogènes, conçus éventuellement au moyen de langages de programmation différents, en faisant abstraction des techniques de communication entre les différentes parties du système. Enfin, l'approche de l'annexe *Distributed Systems* du langage Ada [69] est celle qui se rapproche le plus de celle développée dans cette thèse. Cette annexe permet la conception d'un système réparti sous forme d'un programme global, comprenant des partitions exprimant le programme à exécuter sur chacune des ressources de calcul. Parmi les langages moins notoires, on peut citer le langage Oz [51], qui permet la conception de systèmes répartis au moyen de la notion de ports, sur lesquels une primitive `send` permet d'envoyer des valeurs. Là encore, les partitions du systèmes sont explicites, ainsi que les communications entre ces partitions. Le langage Sequoia [35] permet de décrire l'architecture de la machine, puis d'instancier par compilation de ce langage un programme décrit au moyen de primitives de parallélisation (localisation et synchronisation des calculs) sur l'architecture décrite. L'approche proposée dans [26] permet de partitionner un unique programme en un client et un serveur, en fonction des politiques de sécurité définies par le programmeur au moyen d'annotations sur le programme à répartir.

Parmi les approches les plus récentes concernant les langages de programmation généralistes de systèmes répartis, on peut citer le langage Acute [73]. Ce langage est une extension d'un noyau ML permettant le développement de systèmes répartis, par le biais de primitives de sérialisation et de désérialisation de données incluant la sérialisation du type des données communiquées d'un site à l'autre. Cette caractéristique permet de vérifier, à l'exécution, que le type de données d'une donnée reçue correspond au type attendu, étendant ainsi le concept de typage fort de ML d'un unique programme à une architecture répartie. Cependant, cette vérification de type est faite lors de la désérialisation des données, et est donc forcément dynamique : on perd ainsi l'avantage du typage statique de ML, permettant de vérifier le type des données à la compilation. L'approche que nous proposons permet de conserver le typage statique, effectué sur le programme

[4]Remote Method Invocation
[5]Common Object Request Broker Architecture

complet avant sa répartition.

Les langages de description d'architecture, et particulièrement le langage AADL [36, 3], permettent de concevoir un système composé de plusieurs ressources géographiquement réparties. Dans le cas d'AADL, la conception d'un tel système est effectuée de manière hiérarchique : l'architecture matérielle est décrite en terme de processeurs, mémoires, bus et périphériques. Des contraintes temps-réel de périodes d'exécution sont associées à ces éléments matériels. l'architecture logicielle est ensuite un raffinement de cette architecture matérielle, ce qui ne permet pas la conception du système indépendamment de l'architecture, lorsque la modularité fonctionnelle rentre en conflit avec la modularité de l'architecture.

Le principe de la conception orientée plateforme (platform-based design, [18]) est de concevoir le système en définissant des couches d'abstraction (les dénommées plateformes). Ces couches permettent d'abstraire l'implantation à un niveau donné du système, sous forme d'un ensemble de ressources offertes, associé à des contraintes entre ces ressources. On peut voir cette notion comme la généralisation de la notion de module à l'échelle d'un système entier, architecture matérielle comprise. Cette généralisation permet la conception conjointe architecture/logiciel, sans toutefois perdre en modularité : l'objectif affiché de la conception orientée plateforme, utilisée dans l'industrie électronique, est de réduire la durée de développement de systèmes induite par cette conception conjointe. Dans le cadre d'un système réparti, la notion de « plateforme réseau » (network platform) permet de décrire une architecture composée de plusieurs ressources de calculs, reliées par des liens de communication. La méthode exposée dans cette thèse peut ainsi être considérée comme une instance de cette notion de conception orientée plateforme, instance dans laquelle l'architecture matérielle est la plateforme considérée.

Enfin, plusieurs modèles de calculs formels de description de systèmes répartis ont été définis. La plupart de ces calculs sont basés sur le π-calcul, afin d'exprimer l'exécution concurrente de ces systèmes répartis. Ils consistent en la définition de sites (ou *locations* en anglais), permettant de délimiter la localisation de parties du calcul. La différence entre ces différents calculs tient à l'expression de l'architecture, et l'expression des communications entre sites. L'architecture peut ainsi soit être non hiérarchique : c'est le cas du $\pi_{1\ell}$-calcul [4] et du Dπ-calcul (*distributed π-calculus*, [5]) ; soit hiérarchique ou arborescente, par exemple pour les ambients [17] ou le join-calcul [37]. Le cas non hiérarchique correspond à la volonté de représenter une architecture physique (ressources de calcul tels que des processeurs) ou logique (processus légers sur un même processeur) composée de ressources s'exécutant de manière concurrente. Les architectures hiérarchiques permettent d'exprimer des propriétés de sécurité (par exemple, interdire l'accès à certaines informations depuis d'autres ressources), par encapsulation de ressources logiques à l'intérieur même de ressources parentes. L'expression des communications entre les ressources

diffère d'un calcul à l'autre : elles peuvent prendre la forme d'émission de valeurs sur un canal global ($\pi_{1\ell}$-calcul et join-calcul), ou faire l'objet d'une migration du processus ou de la ressource qui elle-même effectuera une communication locale une fois reçue (Dπ-calcul et ambients). Parmi les modèles synchrones, on peut enfin citer ULM (Un Langage pour la Mobilité, [14]), qui permet au sein d'un même programme de déclarer plusieurs processus légers, et offre la possibilité de migrer un calcul en cours d'exécution d'un processus à l'autre. Un programme ULM permet de décrire un *réseau asynchrone de machines synchrones*. Le point commun entre ces modèles formels est qu'ils décrivent des systèmes dont la répartition des calculs est entièrement déterminée. De plus, sauf pour le Dπ-calcul, la structure fonctionnelle du programme est un raffinement de la structure de l'architecture. Notre approche est donc différente en deux points : tout d'abord, nous proposons un langage permettant de décrire de manière partielle la répartition du programme, cette répartition étant complétée automatiquement à la compilation. D'autre part, ces calculs sont basés sur le π-calcul, ce qui leur donne un caractère impératif. Le formalisme que nous utilisons, les réseaux de Kahn, permet de rendre compte plus aisément de l'exécution de programmes flots de données répartis, avec une sémantique globale synchrone.

2.6.2 Répartition de programmes synchrones

Le modèle le plus couramment utilisé d'architecture pour la répartition de programmes synchrones sont les architectures *globalement asynchrones et localement synchrones* (GALS [25]). Ces architectures sont composées de composants synchrones, communiquant entre eux de manière asynchrone. Une telle architecture permet d'éviter la propagation d'une horloge globale, et est donc plus réaliste pour décrire un système réparti.

Le travail exposé dans cette thèse repose sur la méthode proposée par Alain Girault [39, 21, 16]. Cette méthode opère sur un programme synchrone compilé sous la forme d'un automate d'états finis, dont les états comportent des actions séquentielles (affectations, calculs) sur les variables d'une mémoire bornée associée à cet automate (format OC). À partir de la localisation des entrées et des sorties du programme, cette méthode permet d'inférer automatiquement la localisation de chaque opération séquentielle. L'automate est ensuite dupliqué sur chaque ressource de l'architecture, supposée complètement connectée, puis les opérations séquentielles sont effacées des sites sur lesquels elles ne sont pas localisées. Cette méthode permet de conserver la structure de contrôle du programme. Nous avons voulu étendre cette méthode, d'une part dans l'expression de la localisation de calculs du programme, et non plus seulement des entrées et des sorties, et par l'expression de contraintes de communications entre les ressources de l'architecture. La modularité de la méthode de répartition, et son application à des programmes

flots de données d'ordre supérieur, sont aussi des apports importants à la méthode initiale.

Une méthode de répartition de programmes Signal a été proposée en [9]. Cette méthode s'applique sur la description de programmes Signal dans le formalisme SDFG (Synchronous Data-Flow Graphs). Les nœuds de ces graphes flots de données sont localisés sur un ensemble de processeurs. Ce formalisme est conservé pour décrire l'exécution répartie du programme, transformé par l'introduction de nœuds de communications entre les nœuds exécutés sur deux processeurs différents. La répartition du contrôle est effectuée par une analyse d'horloges, permettant d'inférer les horloges nécessaires sur chaque nœud, et ainsi éviter la diffusion globale de toutes les horloges. Nous utilisons dans ce travail sensiblement la même méthode : les communications sont introduites dans un programme flot de données synchrone décrit sous la forme d'un réseau de Kahn. D'autre part, le caractère polychrone de Signal rend plus complexe la compilation modulaire et l'exécution répartie de programmes Signal. Les travaux de Julien Ouy [68, 67] définissent une méthode de compilation séparée de programmes polychrones, vérifiant la propriété d'endochronie faible [70]. Dans notre cas, le langage utilisé est endochrone, ce qui permet d'utiliser une sémantique de Kahn. De plus, les composants répartis sont compilés à partir d'une même source, et la sémantique des liens de communications (liens synchrones ou FIFOs) permet d'assurer la préservation de la sémantique du programme réparti.

La répartition automatique de programmes flots de données synchrones a été également étudiée dans le cadre de l'outil SynDEx [55]. Cet outil propose une méthode nommée AAA (Adéquation Algorithme Architecture), qui calcule la localisation de blocs logiciels d'un algorithme représenté par un graphe flot de données sur un graphe d'architecture multi-processeurs hétérogènes, en fonction de caractéristiques d'exécution des blocs logiciels sur les composants de l'architecture. Des contraintes supplémentaires de répartition, ou de durée maximale d'exécution d'un cycle, peuvent être spécifiées par l'utilisateur. La répartition calculée est un ordonnancement statique des blocs logiciels qui minimise un critère donné, la longueur du chemin critique. L'approche que nous proposons ici ne permet pas de spécifier de telles contraintes de temps d'exécution, et les contraintes de répartition ne concernent que la localisation de certains calculs, et non leurs temps d'exécution sur les ressources de l'architecture. La répartition calculée n'est donc pas un ordonnancement optimal, mais une répartition respectant les contraintes de localisation spécifiées par le programmeur. Cependant, la méthode que nous proposons pourrait être un support intéressant pour l'intégration d'une méthode telle qu'AAA pour la répartition modulaire de programmes synchrones. Le langage de contraintes que nous proposons devrait alors être étendu afin d'y inclure des contraintes de temps d'exécution.

L'approche quasi-synchrone [22] a été définie à l'occasion du projet Crisys, pour la conception, à l'aide des langages Lustre et Scade, de systèmes de contrôle/commande répartis. Un système réparti est défini par le biais de plusieurs domaines d'horloges. La répartition est effectuée manuellement en affectant aux nœuds l'horloge correspondant à la localisation de ce nœud. L'introduction de délais permet de communiquer des valeurs d'un domaine d'horloge à un autre. La robustesse du système réparti est assurée par la définition de propriétés sur les domaines d'horloges, ainsi que sur la variation dans le temps des valeurs des entrées échantillonnées.

Alan Curic a utilisé Scade/Lustre comme couche intermédiaire de programmation et de validation, pour la compilation de modèles Simulink vers une architecture TTA (Time-Triggered Architecture) [20]. L'architecture TTA comporte un ensemble de ressources de calculs, exécutées suivant une unique horloge globale, et reliées par un bus synchrone. En vue de leur exécution sur une telle architecture, certaines équations Lustre peuvent être localisées au moyen d'annotations par le programmeur. Un algorithme global permet d'inférer, à partir de ces annotations, la localisation de toutes les opérations. La question de la modularité de cette méthode n'est ici pas posée.

Enfin, l'approche présentée dans cette thèse peut être comparée aux méthodes de conception conjointe logiciel/matériel. Le logiciel et le matériel sont alors deux ressources de calculs nécessitant deux processus de compilation différents. Le langage SHIM [34] permet une approche synchrone de la conception conjointe logiciel et matériel. Ce langage intègre, dans une syntaxe unique, une sémantique impérative pour la partie logicielle, et une sémantique de transfert de registres (RTL, « register transfer level ») pour la partie matérielle. Le compilateur génère automatiquement les deux parties, dans deux langages cibles adaptés (VHDL et C), et ajoute les communications nécessaires. L'approche que nous avons choisie est au contraire de donner une unique sémantique au programme, sémantique utilisée pour décrire l'exécution du système réparti. L'hypothèse est qu'une sémantique unique permet une compréhension plus globale des programmes écrits, ainsi que l'abstraction de l'architecture visée par le programmeur.

2.6.3 Systèmes de types pour la répartition de programmes

Le système de types défini dans cette thèse s'inspire des systèmes de types à régions et effets [76]. Ces systèmes de types ne visent pas à proprement parler la répartition de programmes sur une architecture comportant plusieurs ressources physiques, mais permettent d'inférer automatiquement l'organisation en mémoire des données de programmes fonctionnels. Cette organisation est déterminée statiquement à la compilation, et permet d'éviter le recours aux systèmes dynamiques de récupération automatique de mémoire (*garbage collector*). Dans notre cas, le fait d'utiliser un tel système de types pour la répartition nécessite l'utilisation d'un

mécanisme de sous-typage, permettant l'expression dans le système de types des communications de valeurs d'une ressource à l'autre. D'autre part, alors que pour les régions, toute valeur est associée à une région déterminée (pointeurs vers des fermetures, ou vers un couple dont les composantes elles-mêmes ont des régions déterminées), nous souhaitons exprimer le fait que les valeurs réparties n'ont pas « une » localisation ; il en va de même pour les nœuds comportant des calculs sur plusieurs ressources.

Un système de types a été proposé pour le Dπ-calcul [59], permettant de vérifier que l'émission ou la réception sur un canal de communication est bien effectuée au sein du site sur lequel ce canal a été défini. Il s'agit ici non pas d'inférer, mais de vérifier la cohérence de la répartition décrite par le programmeur.

Le système de types proposé en [60] permet de déterminer automatiquement, par inférence de types, le type d'accès aux valeurs de programmes impératifs, selon que ces valeurs soient locales ou distantes. La notion de pointeur global est introduite, permettant l'accès aux valeurs distantes : la méthode d'accès est alors différente, et nécessite l'introduction d'une communication réseau.

Enfin, l'opération de projection que nous définissons se rapproche de celle définie en [66]. Cette opération se base sur un programme impératif séquentiel dont les valeurs sont annotées par les sites sur lesquels elles sont présentes. Cette approche permet la programmation d'application multi-tiers de manière intégrée : un processus par tiers est extrait du programme initial. Un système de types, avec un mécanisme de sous-typage, permet de vérifier la cohérence des annotations.

2.7 Contribution

L'objet de ce travail est donc de proposer l'extension d'un langage flots de données avec des primitives d'annotation permettant d'exprimer la localisation de certaines parties d'un programme, puis une méthode de répartition automatique par compilation modulaire de ce langage étendu.

L'extension du langage sera accompagnée d'une sémantique annotée, permettant de formaliser l'exécution répartie d'un programme écrit dans ce langage étendu. Ensuite, un *système de types « spatial »* permet :

- de vérifier la cohérence des annotations introduites par le programmeur ;
- de guider le programmeur en fournissant, pour chaque nœud correctement annoté, son « type spatial », c'est-à-dire une abstraction de la localisation de ce nœud suffisante pour son instanciation (localisation des entrées, des sorties, et des calculs internes) ;
- d'inférer de manière automatique et modulaire la localisation des parties du programme non annotées en fonction des directives du programmeur dans les parties annotées ;

– et enfin, d'instrumentaliser le code avec ces nouvelles annotations et les points de communication en vue de sa répartition automatique par compilation modulaire dirigée par ces types spatiaux.

Ce système de types permet ensuite de définir une *opération de projection* guidée par les types et permettant d'obtenir, pour chaque ressource déclarée de l'architecture, le programme ne contenant que le code réellement exécuté sur cette ressource.

Cette méthode permet donc de définir un langage pour programmer, de manière globale, un système composé de plusieurs ressources matérielles. Cela présente de nombreux avantages :

– cette approche permet d'appliquer sur le programme d'un système réparti les mêmes analyses que sur un programme centralisé : analyse de types, calcul d'horloge, analyse de causalité ;
– ce programme pourra être testé et simulé de manière centralisée, avant sa répartition effective ;
– enfin, la conservation de la modularité permet de passer à l'échelle et de gagner en traçabilité, préalable indispensable à l'application de notre méthode dans un contexte industriel.

Ce travail a fait l'objet d'une publication à LCTES 2008 [33]. Une autre publication, dans le workshop SLA++P 2008 [32], définit une application de cette méthode de répartition pour permettre l'application de la synthèse de contrôleurs discret à des systèmes comprenant des structures et des types de données non uniquement booléens. Cette méthode est résumée au chapitre 9, section 9.3.3.

Chapitre 3

Extension du langage pour la répartition

Ce chapitre présente deux extensions de Lucid Synchrone permettant au programmeur d'exprimer la répartition de son programme. Le choix de ces extensions s'appuie sur les caractéristiques des systèmes visés, qui sont rappelés dans une première section. La première extension concerne les architectures homogènes, au sens où les ressources sont symétriques et offrent le même ensemble d'opérations. La deuxième extension concerne les architectures hétérogènes.

3.1 Rappel de la problématique

Les systèmes visés par le langage étendu sont des systèmes temps-réel répartis. La criticité de ces systèmes impose certaines contraintes sur le langage. Notamment, pour assurer le caractère temps-réel, les analyses de correction du système sont faites à la compilation. La répartition de programmes écrits dans ce langage sera donc effectuée à la compilation, et non à l'exécution. Pour la même raison, l'architecture, c'est-à-dire les sites présents et leur connectivité, est fixée par le programmeur une fois pour toutes au début du programme. Cette architecture n'évolue pas dans le temps : les aspects de « plasticité » de l'architecture n'ont pas été abordés ici.

L'architecture déclarée consiste donc en un ensemble de sites symboliques, représentant chacun une ressource physique du système visé. Le programme décrit est ensuite déployé sur cet ensemble de sites, en respectant les contraintes de connectivité associées à ceux-ci. Nous avons choisi de décrire l'architecture visée sous forme de graphe orienté : la connectivité entre deux sites est abstraite par la *possibilité* de communication d'un site à l'autre. L'existence effective et le nombre de ces communications seront inférées par l'analyse du programme lui-même, et

non pas contraintes par la description de l'architecture.

Nous avons vu au chapitre 2 que le conflit existant entre la modularité fonctionnelle et la modularité architecturale justifie l'extension du langage, en vue de permettre la conception modulaire du système, abstraction faite de son architecture. Cette extension devra donc permettre au programmeur :

- de déclarer les ressources disponibles, sous forme d'un ensemble de « sites » symboliques (notés par la suite `A`, `B`, ...) ;
- de placer certains calculs sur ces ressources, indépendamment de la modularité fonctionnelle : le langage doit permettre de ne pas avoir à regrouper les calculs s'effectuant sur une même ressource ;
- de laisser certains placements de calculs non spécifiés ;
- de s'abstraire du placement et de l'expression des communications.

Le placement des calculs non spécifiés, ainsi que l'emplacement des communications, seront inférés par analyse statique à la compilation, grâce au système de types présenté au chapitre 5.

Deux approches sont présentées ici : une extension adaptée aux architectures homogènes, et une aux architectures hétérogènes.

3.2 Architectures homogènes

Les ressources sont dans ce cas un ensemble de sites homogènes : tout calcul peut être placé indifféremment sur n'importe quel site de l'architecture. C'est le cas par exemple de machines identiques en réseau (un site correspond à une machine), ou de machines multi-processeurs (les sites sont alors les processeurs de la machine).

L'extension consiste alors en l'ajout d'une part de constructions de déclarations de sites symboliques (auxquels aucune propriété particulière n'est associée) ainsi que les liens de communication possibles entre eux ; et d'autre part de constructions permettant au programmeur de placer le calcul d'une expression sur un des sites déclarés.

La déclaration suivante est celle d'une architecture composée de deux sites symboliques `FPGA` et `DSP` (introduits par le mot-clé `loc`), et d'un lien de communication de `FPGA` à `DSP` (non symétrique, introduit par le mot-clé `link`).

```
loc FPGA
loc DSP
link FPGA to DSP
```

L'expression `e at A` signifie alors que l'expression `e` est calculée sur le site `A`. On peut alors écrire, par exemple, le nœud suivant :

```
node channel(filter,demod,x) = y where
      m = filter(x) at FPGA
  and y = demod(m) at DSP
```

Ce nœud `channel` consiste en l'enchaînement d'une opération de filtrage et d'une opération de démodulation, la première étant effectuée sur le site `FPGA`, et la seconde sur le site `DSP`.

Pour décrire les architectures homogènes, la syntaxe formelle est étendue ainsi :

$$\begin{array}{rcl} P & ::= & \mathcal{A}; d_1; \ldots; d_n; D \\ \mathcal{A} & ::= & \mathcal{A}; \mathcal{A} \mid \texttt{loc}\ A \mid \texttt{link}\ A\ \texttt{to}\ A \\ & & \vdots \\ e & ::= & \ldots \mid e\ \texttt{at}\ A \end{array}$$

L'ajout des annotations ... `at` s ne modifie pas la sémantique centralisée. Le prédicat de réaction pour les expressions, défini figure 2.8, est modifié en ajoutant la règle At ci-dessous :

$$(\textsc{At})\quad \frac{R \vdash e \xrightarrow{v} e'}{R \vdash e\ \texttt{at}\ A \xrightarrow{v} e'\ \texttt{at}\ A}$$

3.3 Architectures hétérogènes

Les ressources de calcul détiennent cette fois des opérations spécialisées, disponibles uniquement sur une ressource (architecture hétérogène : FPGA/DSP,...). Une ressource de calcul est donc déclarée avec une liste d'opérations disponibles sur cette ressource. Celle-ci peut alors être considérée comme un module, offrant un ensemble de fonctionnalités. L'utilisation, dans le programme, d'une de ces fonctionnalités, est alors conditionnée par son exécution sur cette ressource particulière. Par exemple, le code ci-dessous déclare une architecture composée de deux sites `FPGA` et `DSP`, avec un lien de communication de `FPGA` vers `DSP`, et tels que les fonctions `(*^)` et `(+^)` seront exécutées sur le site `FPGA`, et `(*@)` et `(+@)` sur le site `DSP`. Chacune de ces opérations prend en paramètre deux entiers, et renvoie un entier.

```
loc FPGA = sig
  (+) : int -> int -> int
  (-) : int -> int -> int
end
```

```
loc DSP = sig
  (*) : int -> int -> int
  (/) : int -> int -> int
end

link FPGA to DSP
```

Cette approche permet de programmer notre exemple de radio logicielle, non pas en spécifiant la localisation des fonctions elles-mêmes, mais celle des opérateurs utilisés pour la conception de ces fonctions. Le code ci-dessous montre ainsi l'implémentation du canal de radio logicielle sur une architecture hétérogène. Cette architecture est composée de deux sites, l'un représentant un FPGA, et l'autre représentant un DSP. Chacun de ces sites propose deux fonctions spécialisées, un multiplicateur et un additionneur.

```
node filter_bp(x) = y where
      y  = (bp1 *^ x) +^ (0 -> pre u2)
  and u2 = (bp2 *^ x) +^ (0 -> pre u3)
  and u3 = (bp3 *^ x) +^ (0 -> pre u4)
  and u4 = (bp4 *^ x)

node filter_lp(x) = y where
      y  = (lp1 *@ x) +@ (0 -> pre u2)
  and u2 = (lp2 *@ x) +@ (0 -> pre u3)
  and u3 = (lp3 *@ x) +@ (0 -> pre u4)
  and u4 = (lp4 *@ x)

node demodulation(x) = y where
      c = carrier_recovery(x)
  and d = c *@ x
  and f = filter_lp(d)
  and y = str(f)

node channel(x) =  y where
      m = filter_bp(x)
  and y = demodulation(m)
```

Le filtre passe-bande, destiné à être exécuté sur le FPGA, est ainsi programmé avec le multiplicateur et l'additionneur de celui-ci (notés respectivement `(*^)` et `(+^)`). La démodulation est programmée avec les opérations présentes sur le DSP (notés respectivement `(*@)` et `(+@)`).

Cette approche permet, en utilisant l'ordre supérieur, de ne programmer qu'une

fois certains schémas redondants, en mettant en paramètres des nœuds les opérations de base que ces schémas utilisent. Ces schémas peuvent ensuite être instanciés avec les opérations déclarées sur les différents sites.

Par exemple, le schéma d'un filtre à réponse impulsionnelle finie (FIR), tel que programmé pour le filtre à bande passante présenté précédemment, peut être programmé de manière plus générale en mettant en paramètre les coefficients multiplicateurs, ainsi que les opérations de multiplication et d'addition :

```
node fir_filter (w1, w2, w3, w4, (*), (+), x) = y where
      y  = (w1 * x) + (0 -> pre u2)
  and u2 = (w2 * x) + (0 -> pre u3)
  and u3 = (w3 * x) + (0 -> pre u4)
  and u4 = (w4 * x)
```

Ce nœud peut ensuite être instancié avec les opérateurs spécialisés de chacun des sites, afin d'obtenir un nouveau nœud exécutable par chacun des sites :

```
node filter_bp x = y where
  y = fir_filter (bp1, bp2, bp3, bp4, (*^), (+^), x)

node filter_lp x = y where
  y = fir_filter (lp1, lp2, lp3, lp4, (*@), (+@), x)
```

La syntaxe formelle devient, pour les architectures hétérogènes :

$$
\begin{array}{lcl}
P & ::= & \mathcal{A}; d_1; \ldots; d_n; D \\
\mathcal{A} & ::= & \mathcal{A}; \mathcal{A} \mid \texttt{loc}\ A = \texttt{sig}\ m\ \texttt{end} \mid \texttt{link}\ A\ \texttt{to}\ A \\
m & ::= & m; m \mid \mathrm{op} : dt \mid \epsilon \\
dt & ::= & dt \rightarrow dt \mid dt \times dt \mid \alpha \mid b \\
b & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots \\
 & & \vdots \\
e & ::= & \ldots \mid e\ \texttt{at}\ A
\end{array}
$$

Chacune de ces extensions sera prise en compte dans la définition de la sémantique du langage au chapitre 4, puis dans la définition de la méthode de répartition dans les chapitres suivants.

3.4 Communications

Les communications sont introduites de manière transparente pour le programmeur, soit quant à leur position dans le programme, soit quant à leur expression technique (moyen technique utilisé). Le but est ainsi de permettre l'expression séparée de deux préoccupations orthogonales : la fonctionnalité et sa répartition.

L'emplacement des communications doit par ailleurs obéir à deux préoccupations opposées :

- permettre au programmeur de ne pas exprimer la position exacte de toutes les communications ;
- laisser au programmeur un certain contrôle sur leur emplacement possible : cette préoccupation implique notamment que ces communications ne puissent pas être effectuées à n'importe quel point du programme.

De plus, on peut remarquer que la façon dont seront traitées ces deux préoccupations a un impact sur la complexité de la répartition : plus les points du programme où une communication est possible sont nombreux, plus la combinatoire de l'algorithme de répartition est importante. Répondre à la deuxième préoccupation, en offrant au programmeur un moyen de contraindre ces points de communication, permet de traiter simultanément ce problème.

3.5 Conclusion

Nous avons présenté au cours de ce chapitre deux extensions possibles d'un langage flots de données synchrones, afin d'intégrer la répartition de programmes synchrones dans ce langage. Cette intégration dans le langage nous permettra, dans les chapitres suivants, de définir une sémantique pour ce langage étendu, puis une méthode de répartition de programmes écrit dans ce même langage. Nous avons proposé, pour cette extension, deux approches complémentaires : la première est plus adaptée aux architectures homogènes, dont les ressources offrent les même fonctionnalités. Dans ce cas, la localisation des calculs est spécifiée au sein du programme. La deuxième extension permet de décrire des architectures hétérogènes, dont les ressources offrent des fonctionnalités différentes : la localisation des calculs dépend alors des opérations utilisées. Ces deux approches peuvent tout à fait être utilisées simultanément.

Chapitre 4

Sémantique synchrone et localisation des valeurs

Ce chapitre a pour but de proposer un moyen de décrire l'exécution du programme synchrone en prenant en comte les annotations de répartition introduites par le programmeur. On s'intéresse donc ici à la manière dont un programme réparti peut être représenté formellement, afin d'indiquer au programmeur le sens des annotations qu'il introduit. La formalisation concerne tout d'abord la représentation des valeurs réparties, et donc celle de leur localisation ainsi que celle de la localisation de leur calcul, puis la représentation de l'exécution d'un programme en tenant compte des annotations de répartition.

4.1 Une sémantique pour le programmeur

Nous avons vu, au chapitre précédent, que les annotations introduites par le programmeur n'altéraient pas la sémantique du programme. Au chapitre 6, définissant l'opération de répartition, la sémantique du programme réparti sera définie par le produit synchrone des fragments issus de la répartition du programme initial. Le but de la sémantique que nous définissons ici est de tenir compte des annotations, en faisant abstraction de la méthode de répartition utilisée. Elle permet donc au programmeur de donner un sens aux annotations de répartition. En ce sens, cette nouvelle sémantique est liée au système de types défini au chapitre 5, et permet notamment de définir sa correction.

Cette sémantique consiste en l'ajout, aux valeurs, d'annotations permettant de définir leur localisation sur l'architecture. L'exécution du programme réparti est ensuite définie comme un calcul à partir de valeurs annotées. Notamment, une opération binaire primitive (addition, etc.) ne pourra être effectuée qu'entre deux valeurs présentes sur le même site, c'est-à-dire annotées par ce même site. La com-

munication d'une valeur d'un site à l'autre est représentée par la modification de son annotation, en cohérence avec les liens de communication déclarés dans l'architecture. Cette sémantique ne restreint pas la localisation des communications au sein du programme ; elle autorise d'autre part à effectuer plusieurs communications à la suite les unes des autres. L'intérêt et l'implantation de telles restrictions seront abordés au chapitre 5.

Cette sémantique reste une sémantique synchrone : les valeurs annotées ou valeurs réparties décrites ici sont des valeurs synchrones, c'est-à-dire des valeurs dont les composantes sont présentes au même instant logique. De même, l'exécution des programmes annotés est une exécution synchrone.

4.2 Localisation des valeurs

Une valeur synchrone répartie, notée $\hat{v}$, est soit une valeur locale vl associée à un site A (vl `at` A), soit un couple réparti $(\hat{v}, \hat{v})$, soit une fonction. Une fonction n'a pas de localisation : celle-ci est déterminée par la localisation des calculs définissant cette fonction. Une valeur locale est une valeur immédiate ou un couple local (c'est-à-dire, dont les deux composantes se trouvent sur le même site). Seules les valeurs locales à un site peuvent être communiquées d'un site à l'autre. On note $\hat{R}$ un environnement de réaction associant aux variables des valeurs réparties.

$$\begin{array}{rl} \hat{v} ::= & vl \texttt{ at } A \mid (\hat{v}, \hat{v}) \mid \lambda x.e \texttt{ where } D \\ vl ::= & i \mid (vl, vl) \\ \hat{R} ::= & [\hat{v}_1/x_1, \ldots, \hat{v}_n/x_n] \\ & \text{si } \forall i \neq j, x_i \neq x_j \\ \hat{S} ::= & \hat{R}_1 \ldots\ldots \hat{R}_n \ldots \end{array}$$

D'autre part, l'égalité entre les valeurs réparties est étendue comme suit :

$$(vl_1, vl_2) \texttt{ at } A = (vl_1 \texttt{ at } A, vl_2 \texttt{ at } A)$$

$$\frac{\hat{v}_1 = \hat{v}'_1 \quad \hat{v}_2 = \hat{v}'_2}{(\hat{v}_1, \hat{v}_2) = (\hat{v}'_1, \hat{v}'_2)}$$

Ainsi, la valeur répartie 42 `at` A représente la valeur 42 localisée sur le site A ; $(42 \texttt{ at } A, 71 \texttt{ at } A) = (42, 71) \texttt{ at } A$ représente le couple $(42, 71)$, dont chaque composante est localisée sur le même site A. Ces deux valeurs peuvent être communiquées vers n'importe quel site accessible depuis le site A. *A contrario*, la valeur $(42 \texttt{ at } A, 71 \texttt{ at } B)$ ne peut être communiquée. Elle représente un couple dont la première composante est sur le site A et la deuxième sur le site B.

L'opérateur $|\cdot|$ permet d'obtenir, à partir d'une valeur répartie, une valeur où toutes les informations de localisation ont été « oubliées » :

$$\begin{aligned} |vl \texttt{ at } s| &= vl \\ |(\hat{v}_1, \hat{v}_2)| &= (|\hat{v}_1|, |\hat{v}_2|) \\ |\lambda x.e \texttt{ where } D| &= \lambda x.e \texttt{ where } D \end{aligned}$$

Cet opérateur est étendu aux environnement de réaction, et aux séquence d'environnements :

$$|[\hat{v}_1/x_1, \ldots, \hat{v}_n/x_n]| = [|\hat{v}_1|/x_1, \ldots, |\hat{v}_n|/x_n] \qquad |\hat{R}.\hat{S}| = |\hat{R}|.|\hat{S}|$$

D'autre part, nous définissons la fonction $\mathrm{loc}(\cdot)$, qui permet d'obtenir l'ensemble des sites sur lesquelles une valeur est répartie :

$$\begin{aligned} \mathrm{loc}(i \texttt{ at } s) &= \{s\} \\ \mathrm{loc}((\hat{v}_1, \hat{v}_2) \texttt{ at } s) &= \mathrm{loc}(\hat{v}_1) \cup \mathrm{loc}(\hat{v}_2) \end{aligned}$$

4.3 Exécution répartie

L'exécution répartie d'un programme synchrone suppose l'existence d'un ensemble fini de sites $\mathcal{S}$, et d'une relation $\mathcal{L} \subseteq \mathcal{S} \times \mathcal{S}$ représentant les liens de communication existant entre ces sites. $(A_1, A_2) \in \mathcal{L}$ signifie qu'une valeur présente sur le site A_1 peut être communiquée vers le site A_2. Formellement, une description d'architecture $\mathcal{A}$ définit un graphe $G = \langle \mathcal{S}, \mathcal{L} \rangle$: le prédicat $G \vdash \mathcal{A} : G'$ signifie qu'à partir du graphe G, la description d'architecture $\mathcal{A}$ définit le nouveau graphe d'architecture G'. Les règles Arch, Def-Site et Def-Link ci-dessous définissent ce prédicat :

(Arch)
$$\frac{G \vdash \mathcal{A}_1 : G_1 \qquad G_1 \vdash \mathcal{A}_2 : G_2}{G \vdash \mathcal{A}_1 ; \mathcal{A}_2 : G_2}$$

(Def-Site)
$$\langle \mathcal{S}, \mathcal{L} \rangle \vdash \texttt{loc } A : \langle \mathcal{S} \cup \{A\}, \mathcal{L} \rangle$$

(Def-Link)
$$\frac{A_1, A_2 \in \mathcal{S}}{\langle \mathcal{S}, \mathcal{L} \rangle \vdash \texttt{link } A_1 \texttt{ to } A_2 : \langle \mathcal{S}, \mathcal{L} \cup \{A_1 \mapsto A_2\} \rangle}$$

Pour un programme $P = \mathcal{A}; d_1; \ldots; d_n; D$, le graphe G de son architecture est construit par le prédicat $\langle \emptyset, \emptyset \rangle \vdash \mathcal{A} : G$.

L'exécution répartie d'un programme synchrone P, dans une séquence $\hat{S}$ d'environnements de réaction de valeurs réparties ($\hat{S} = \hat{R}_1.\hat{R}_2.\ldots$) représentant les

entrées du programme, est définie par une séquence $\hat{S}'$ d'environnements de réaction ($\hat{S}' = \hat{R}'_1.\hat{R}'_2.\ldots$) représentant ses sorties.

Le prédicat $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$ signifie que le programme D, dans l'environnement de réaction de valeurs réparties $\hat{R}$, réagit en produisant un environnement de réaction $\hat{R}'$, et se réécrit en D'. ℓ est l'ensemble de sites impliqués dans la réaction. De même que pour la sémantique non annotée, l'exécution du programme $P = \mathcal{A}; d_1; \ldots; d_n; D$ avec la séquence d'entrées $\hat{S}_{in}$ a pour résultat la séquence de sorties $\hat{S}_{out}$ tel que $\hat{S}_{in} \Vdash P : \hat{S}_{out}$; les définitions $d_i = \texttt{node}\ f_i(p_i) = e_i\ \texttt{where}\ D_i$ définissent des valeurs $\lambda p_i.e_i\ \texttt{where}\ D_i$, et on considère que pour tout i, $\hat{R}_d(f_i) = \lambda p_i.e_i\ \texttt{where}\ D_i$. Les ensembles de sites $\mathcal{S}$ et de liens de communication $\mathcal{L}$ sont définis de manière globale à partir de la description d'architecture $\mathcal{A}$.

$$\frac{\hat{R}_d, \hat{R}_{in}, \hat{R}_{out} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}_{out}} D' \qquad \ell \subseteq \mathcal{S} \qquad \hat{S}_{in} \Vdash \mathcal{A}; d_1; \ldots; d_n; D' : \hat{S}_{out}}{\hat{R}_{in}.\hat{S}_{in} \Vdash \mathcal{A}; d_1; \ldots; d_n; D : \hat{R}_{out}.\hat{S}_{out}}$$

$$\text{où } \langle \emptyset, \emptyset \rangle \vdash \mathcal{A} : \langle \mathcal{S}, \mathcal{L} \rangle$$

Les règles définissant la relation $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$ sont données en figure 4.1.

- Règle Imm : une valeur immédiate peut être placée sur n'importe quel site. L'ensemble des sites impliqués dans la réaction est réduit à ce site.
- Règle Inst : l'ensemble des sites impliqués dans la réaction d'une instanciation est l'ensemble des sites présents dans la valeur émise.
- Règle At : une expression e peut être évaluée sur le site A seulement si l'ensemble des sites impliqués dans le calcul de cette expression est réduit à ce site A.
- Règle Comm : si une expression e réagit en émettant la valeur $\hat{v} = vl\ \texttt{at}\ A$ localisée sur un unique site A, alors cette valeur peut être communiquée vers un site A' accessible depuis A. Cette règle peut être appliquée un nombre arbitraire de fois entre deux autres règles ; cela revient à considérer comme ensemble de liens de communication la fermeture transitive de $\mathcal{L}$. Celle-ci n'est toutefois pas directement calculée, afin de conserver l'ensemble des sites par lesquels une valeur a transité lors d'une communication entre deux sites non directement connectés.
- Règle Op : un opérateur s'applique sur une valeur présente sur un unique site. Le résultat de l'opération se trouve sur ce même site. Par exemple, pour être évaluée, la condition d'une expression conditionnelle doit se trouver sur le même site que les deux valeurs alternatives entre lesquelles choisir.
- Règle Pair : l'ensemble des sites impliqués par la réaction est l'union des sites impliqués dans le calcul des deux composantes du couple.

– Règles DEF et APP : l'ensemble des sites impliqués dans la réaction de l'équation est l'ensemble des sites impliqués dans l'expression ou l'application calculée.
– Règle AND : l'ensemble des sites impliqués est l'union des sites impliqués dans la réaction des deux ensembles d'équations mis en parallèle.

Le lemme suivant exprime le fait que tout programme réagissant de manière répartie réagit de manière centralisée.

Lemme 1. *Pour toutes équations D et D' (respectivement, pour toutes expressions e et e'), pour tout environnement de réaction $\hat{R}$, et pour tout environnement de réaction $\hat{R}'$ (resp. toute valeur répartie $\hat{v}$), si $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$ (resp. $\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\hat{v}} e'$), alors il existe un environnement de réaction non réparti R tel que $|\hat{R}| = R$, et R' tel que $|\hat{R}'| = R'$ (resp., une valeur v telle que $|\hat{v}| = v$) tels que $R \vdash D \xrightarrow{R'} D'$ (resp. $R \vdash e \xrightarrow{v} e'$).*

La démonstration de ce lemme consiste en l'« effacement » systématique des informations de répartition attachées à l'exécution répartie, en vue d'obtenir une sémantique équivalente à la sémantique centralisée.

Démonstration. La démonstration repose sur l'extension de $|\cdot|$ aux prédicats $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$ et $\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\hat{v}} e'$. Pour chaque règle définissant ces prédicats, on peut remplacer $\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\hat{v}} e'$ par $|\hat{R}| \vdash e \xrightarrow{|\hat{v}|} e'$ et $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$ par $|\hat{R}| \vdash D \xrightarrow{|\hat{R}'|} D'$ pour obtenir une sémantique centralisée équivalente à celle définie en section 2.4.3. □

4.4 Exemple

Pour illustration, on suppose l'existence d'une architecture composée de deux sites A et B, avec un lien de communication de A vers B.

La sémantique est illustrée par l'exemple suivant :

```
    y = x + 1
and z = y + 2 at B
```

Ce programme est exécuté dans la séquence d'environnement de réaction $\hat{S}_{in} = \hat{R}_1.\hat{R}_2.\ldots$ où $\hat{R}_j = [i_j \texttt{ at } A/x]$. La première réaction répartie de ce programme est décrite par la réduction de la figure 4.2.

Il faut remarquer ici que deux réactions différentes sont possibles, selon que l'opération $x + 1$ est exécutée sur A (comme dans la figure 4.2) ou sur B. Dans le dernier cas, la communication est placée avant l'application du premier opérateur `(+)`. La réduction devient alors celle exposée en figure 4.3.

(Imm)

$$\hat{R} \overset{\{s\}}{\Vdash} i \xrightarrow{i \texttt{ at } s} i$$

(Inst)

$$\hat{R}, [\hat{v}/x] \overset{\mathrm{loc}(\hat{v})}{\Vdash} x \xrightarrow{\hat{v}} x$$

(At)

$$\frac{\hat{R} \overset{\{A\}}{\Vdash} e \xrightarrow{\hat{v}} e'}{\hat{R} \overset{\{A\}}{\Vdash} e \texttt{ at } A \xrightarrow{\hat{v}} e' \texttt{ at } A}$$

(Comm)

$$\frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{vl \texttt{ at } A} e' \qquad (A, A') \in \mathcal{L}}{\hat{R} \overset{\ell \cup \{A'\}}{\Vdash} e \xrightarrow{vl \texttt{ at } A'} e'}$$

(Op)

$$\frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{v \texttt{ at } A} e' \qquad \mathrm{op}(v) \xrightarrow{v'} \mathrm{op}'}{\hat{R} \overset{\ell}{\Vdash} \mathrm{op}(e_1, e_2) \xrightarrow{v' \texttt{ at } A} \mathrm{op}'(e')}$$

(Pair)

$$\frac{\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{\hat{v}_1} e_1' \qquad \hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{\hat{v}_2} e_2'}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} (e_1, e_2) \xrightarrow{(\hat{v}_1, \hat{v}_2)} (e_1', e_2')}$$

(Def)

$$\frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{(\hat{v}_1, \hat{v}_2)} e'}{\hat{R} \overset{\ell}{\Vdash} (x_1, \ldots, x_n) = e \xrightarrow{[\hat{v}_1/x_1, \ldots, \hat{v}_n/x_n]} (x_1, \ldots, x_n) = e'}$$

(App)

$$\frac{\hat{R}(f) = \lambda y.e \texttt{ where } D \qquad \hat{R} \overset{\ell}{\Vdash} \texttt{let } y = e' \texttt{ and } D \texttt{ in } x = e \xrightarrow{\hat{R}'} D'}{\hat{R} \overset{\ell}{\Vdash} x = f(e') \xrightarrow{\hat{R}'} D'}$$

(And)

$$\frac{\hat{R}, \hat{R}_2 \overset{\ell_1}{\Vdash} D_1 \xrightarrow{\hat{R}_1} D_1' \qquad \hat{R}, \hat{R}_1 \overset{\ell_2}{\Vdash} D_2 \xrightarrow{\hat{R}_2} D_2'}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} D_1 \texttt{ and } D_2 \xrightarrow{\hat{R}_1, \hat{R}_2} D_1' \texttt{ and } D_2'}$$

(Let)

$$\frac{\hat{R} \overset{\ell_1}{\Vdash} D_1 \xrightarrow{\hat{R}_1} D_1' \qquad \hat{R}, \hat{R}_1 \overset{\ell_2}{\Vdash} D_2 \xrightarrow{R_2} D_2'}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} \texttt{let } D_1 \texttt{ in } D_2 \xrightarrow{\hat{R}_2} \texttt{let } D_1' \texttt{ in } D_2'}$$

Fig. 4.1 – Sémantique annotée.

$$
\begin{array}{c}
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}[i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A\}}{\Vdash} x \xrightarrow{i_1 \texttt{ at } A} x \\ [i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A\}}{\Vdash} 1 \xrightarrow{1 \texttt{ at } A} 1 \\ (i_1 \texttt{ at } A, 1 \texttt{ at } A) = (i_1, 1) \texttt{ at } A \end{array}}{[i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A\}}{\Vdash} (x,1) \xrightarrow{(i_1,1) \texttt{ at } A} (x,1)}\,(\textsc{Pair})}{(+)(i_1,1) \xrightarrow{i_1+1} (+)}\,(\textsc{Op})}{[i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A\}}{\Vdash} x+1 \xrightarrow{i_1+1 \texttt{ at } A} x+1}\,(\textsc{Comm})}{[i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A,B\}}{\Vdash} x+1 \xrightarrow{i_1+1 \texttt{ at } B} x+1}\,(\textsc{Def})}{[i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A,B\}}{\Vdash} y = x+1 \xrightarrow{[i_1+1 \texttt{ at } B/y]} y = x+1} \\
\vdots \\
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}[i_1 \texttt{ at } A/x, i_1+1 \texttt{ at } B/y] \overset{\{B\}}{\Vdash} y \xrightarrow{i_1+1 \texttt{ at } B} y \\ [i_1 \texttt{ at } A/x, i_1+1 \texttt{ at } B/y] \overset{\{B\}}{\Vdash} 2 \xrightarrow{2 \texttt{ at } B} 2 \\ (i_1+1 \texttt{ at } B, 2 \texttt{ at } B) = (i_1+1, 2) \texttt{ at } B \end{array}}{[i_1 \texttt{ at } A/x, i_1+1 \texttt{ at } B/y] \overset{\{B\}}{\Vdash} (y,2) \xrightarrow{(i_1+1,2) \texttt{ at } B} (y,2)}\,(\textsc{Pair})}{(+)(i_1+1,2) \xrightarrow{i_1+1+2} (+)}\,(\textsc{Op})}{[i_1 \texttt{ at } A/x, i_1+1 \texttt{ at } B/y] \overset{\{B\}}{\Vdash} y+2 \xrightarrow{i_1+1+2 \texttt{ at } B} y+2}\,(\textsc{At})}{[i_1 \texttt{ at } A/x, i_1+1 \texttt{ at } B/y] \overset{\{B\}}{\Vdash} y+2 \texttt{ at } B \xrightarrow{i_1+1+2 \texttt{ at } B} y+2 \texttt{ at } B}\,(\textsc{Def})}{[i_1 \texttt{ at } A/x, i_1+1 \texttt{ at } B/y] \overset{\{B\}}{\Vdash} z = y+2 \texttt{ at } B \xrightarrow{[i_1+1+2 \texttt{ at } B/z]} z = y+2 \texttt{ at } B}\,(\textsc{And}) \\
[i_1 \texttt{ at } A/x] \overset{\{A,B\}}{\Vdash} \begin{array}{l} y = x+1 \\ \texttt{and} \quad z = y+2 \texttt{ at } B \end{array} \xrightarrow{\left[\begin{array}{c} i_1+1 \texttt{ at } B/y, \\ i_1+1+2 \texttt{ at } B/z \end{array}\right]} \begin{array}{l} y = x+1 \\ \texttt{and} \quad z = y+2 \texttt{ at } B \end{array}
\end{array}
$$

FIG. 4.2 – Exemple de réduction des règles de la sémantique annotée.

$$
\begin{array}{rl}
(\textsc{Def}) & \dfrac{(\textsc{Op})\ \dfrac{(\textsc{Pair})\ \dfrac{(\textsc{Comm})\ \dfrac{[i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A\}}{\Vdash} x \xrightarrow{i_1 \texttt{ at } A} x}{[i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A,B\}}{\Vdash} x \xrightarrow{i_1 \texttt{ at } B} x} \quad [i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{B\}}{\Vdash} 1 \xrightarrow{1 \texttt{ at } B} 1 \quad (i_1 \texttt{ at } B, 1 \texttt{ at } B) = (i_1, 1) \texttt{ at } B}{[i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A,B\}}{\Vdash} (x,1) \xrightarrow{(i_1,1) \texttt{ at } B} (x,1)} \quad (+)(i_1,1) \xrightarrow{i_1+1} (+)}{[i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A,B\}}{\Vdash} x+1 \xrightarrow{i_1+1 \texttt{ at } B} x+1}}{[i_1 \texttt{ at } A/x, i_1+1+2 \texttt{ at } B/z] \overset{\{A,B\}}{\Vdash} y = x+1 \xrightarrow{[i_1+1 \texttt{ at } B/y]} y = x+1}
\end{array}
$$

FIG. 4.3 – Réduction alternative pour l'exemple de la figure 4.2.

La répartition décrite par la sémantique n'est donc pas déterminée : pour un même programme, plusieurs exécutions réparties peuvent exister. Cet indéterminisme pour l'affectation des calculs aux sites reflète le fait que le langage ne contraint pas le programmeur à donner la localisation de toutes les parties du programme : plusieurs répartitions sont donc possibles.

De plus, certains programmes n'admettent pas d'exécution répartie. C'est le cas du programme ci-dessous, pour la même architecture (deux sites A et B, un lien de communication de A vers B) :

```
    x = 2 at B
and y = x + 1
and z = y + 2 at A
```

Ce programme n'a pas de sémantique répartie car aucune communication n'est possible du site B vers le site A. Le chapitre 5 propose une méthode, basée sur un système de types, pour vérifier de manière modulaire la cohérence du programme avec l'architecture, et pour placer automatiquement les communications. L'application de ce système de types permet aussi de compléter, par inférence, les informations de localisation partielles issues des annotations du programmeur. On obtient ainsi une répartition particulière parmi l'ensemble des répartitions possibles. Le type des valeurs du programme permet, comme retour vers le programmeur, de décrire cette répartition.

4.5 Sémantique pour architectures hétérogènes

Pour prendre en compte l'exécution de programmes répartis sur des architectures hétérogènes, telles que décrites en section 3.3, le langage d'architecture est modifié de la manière suivante :

$$\mathcal{A} ::= \mathcal{A};\mathcal{A} \mid \texttt{loc}\ A = \texttt{sig}\ \mathrm{op} : dt \ldots \mathrm{op} : dt\ \texttt{end} \mid \texttt{link}\ A\ \texttt{to}\ A$$

Une déclaration $\texttt{loc}\ A = \texttt{sig}\ \mathrm{op} : dt\ \texttt{end}$ signifie que l'architecture est composée d'un site A, sur lequel l'opération op, de type de données dt, est disponible.

À chaque opérateur est donc associé un site. On note op : A le fait que l'opérateur op ait été déclaré sur le site A.

La sémantique est alors donnée par l'ensemble des règles de réduction données en figure 4.1, avec pour règle Op la modification suivante, exprimant le fait qu'un opérateur doit être évalué sur le site où il a été déclaré présent :

$$(\textsc{Op})\quad \frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{v\ \texttt{at}\ A} e' \qquad \mathrm{op} : A \qquad \mathrm{op}(v) \xrightarrow{v'} \mathrm{op}'}{\hat{R} \overset{\ell}{\Vdash} \mathrm{op}(e) \xrightarrow{v'\ \texttt{at}\ A} \mathrm{op}'(e')}$$

Un opérateur déclaré sur le site A est appliqué à deux opérandes présentes sur le site A. Le résultat de cette application est présent sur ce même site A.

4.6 Conclusion

Ce chapitre a permis de donner une sémantique au langage flots de données défini au chapitre 3. Cette sémantique permet de rendre compte de la localisation des valeurs calculées par un programme annoté, ainsi que de leur communication d'un site à l'autre. Elle permettra, par la suite, de définir la correction du système de types présenté au chapitre suivant, et qui permet d'inférer, à partir des annotations de localisation de certaines valeurs, la localisation de l'ensemble des valeurs et calculs du programme. Cette sémantique fait abstraction de la méthode de répartition effectivement utilisée. La sémantique du programme réparti, en tenant compte de cette méthode, sera présentée lors de la définition de celle-ci, au chapitre 6.

Chapitre 5

Système de types pour la répartition

Ce chapitre présente un système de types permettant la répartition modulaire de programmes synchrones. Ce système permet, d'une part, de vérifier la cohérence des annotations de répartition données par le programmeur, et d'autre part, d'inférer, à partir de celles-ci et de manière modulaire, la localisation des flots et des calculs quand elle n'a pas été explicitée.

5.1 Justification et description

Le chapitre précédent donnait un cadre formel de description de l'exécution répartie d'un programme synchrone. Mais ce cadre formel n'est pas une méthode de répartition de ces programmes. Il permet de décrire l'ensemble des exécutions réparties possibles, mais ne permet ni de décider si un programme synchrone est exécutable de manière répartie, ni de décider de la forme de la répartition elle-même. L'approche proposée, un système de types avec inférence, est issue de ces deux exigences. Le parallèle peut être fait avec l'inférence de types de données : le système de types permet de vérifier la cohérence des annotations de répartition, entre elles et vis-à-vis de l'architecture visée, et l'inférence de types permet au programmeur de ne pas spécifier toutes les localisations, laissant ainsi le système de types décider de la localisation des calculs non annotés.

Cette approche par inférence correspond à l'approche par « coloration » proposée par [21, 16], où les « couleurs » sont les sites sur lesquels sont placés les entrées et les sorties, informations à partir desquelles l'ensemble du programme est « coloré ». L'approche par système de types permet d'étendre cette méthode en considérant ces couleurs comme des types, permettant ainsi la répartition modulaire de programmes.

Dans la suite, les types spatiaux sont notés t, les sites s, les constantes de sites (déclarés dans l'architecture) A. On appelle type centralisé un type représentant une valeur localisée sur un unique site ; les types centralisés sont notés tc.

Le type spatial d'une valeur est sa localisation. Une valeur du type tc `at` s est une valeur centralisée présente sur le site s. Une telle valeur a été soit calculée sur le site s, soit communiquée vers le site s depuis un site où s est accessible. Elle est à son tour communicable vers n'importe quel site s' accessible depuis s. Cela correspond à un mécanisme de sous-typage : si l'architecture permet de communiquer une valeur de s vers s', alors une valeur de type tc `at` s peut être utilisée comme étant de type tc `at` s'.

On associe de plus, à une expression, non seulement son type, mais aussi l'ensemble des sites dont cette expression a besoin pour être évaluée. Le type d'un nœud est donc de la forme $t_1 \prec\!\langle\ell\rangle\!\succ t_2$, où :

- t_1 est le type spatial des entrées du nœud ; c'est-à-dire, la localisation de ces entrées ;
- t_2 est le type spatial des sorties du nœud ; c'est-à-dire, la localisation des valeurs une fois le nœud évalué ;
- ℓ est l'*ensemble des sites* nécessaires au calcul du nœud.

Cet ensemble de sites sera utilisé lors de la répartition : en effet, pour déterminer si l'exécution d'un nœud doit être ou non placée sur un site donné, le type de ses entrées et de ses sorties ne suffisent pas.

L'exemple de la figure 5.1 illustre ce problème par un nœud `f`, composé de trois nœuds `f1`, `f2` et `f3`, le premier et le dernier étant exécutés sur le site `A`, et le second sur `B`. `f1(x)` étant spécifié sur `A`, l'entrée `x` du nœud `f` sera localisée sur `A` (c'est-à-dire, d'un type spatial de la forme t `at` A). De même, l'exécution de `f3` étant spécifiée sur `A`, la sortie de `f` sera localisée sur `A`. Le type spatial de `f` sera donc de la forme :

$$\texttt{f} : tc \ \texttt{at}\ A \prec\!\langle\ell\rangle\!\succ tc' \ \texttt{at}\ A$$

Pour pouvoir déterminer, à la répartition, sur quels sites placer l'instanciation de cette fonction `f`, son type spatial doit permettre de connaître, sans examen du contenu de `f`, l'ensemble des sites intermédiaires nécessaires pour son exécution. Ici, on aura donc $\ell = \{A, B\}$. Cet ensemble est appelé l'*effet* du nœud, et permet par ailleurs d'interdire les instanciations de `f` spécifiées sur `A`. Le code `y = f(x) at A` sera ainsi rejeté par le système de types, l'effet $\{A, B\}$ étant incompatible avec une exécution sur l'unique site A.

Ce système de types spatial servira, dans le chapitre 6, à définir une opération de projection effectuant la répartition d'un programme sur l'ensemble des sites symboliques déclarés dans la description de l'architecture jointe à ce programme. Les choix d'expressivité et les restrictions effectués sur ce système de types sont justifiés par cette opération de projection. Le chapitre 8 présente un système de

```
loc A;
loc B;
link A to B;
link B to A;

node f(x) = y3 where
      y1 = f1(x) at A
  and y2 = f2(y1) at B
  and y3 = f3(y2) at A
```

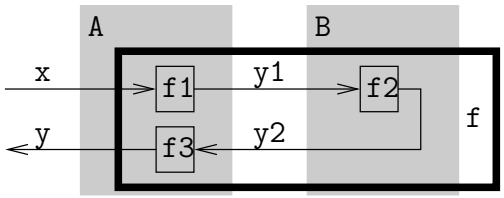


FIG. 5.1 – Nœud dont l'entrée et la sortie sont sur un site `A`, et comprenant un calcul intermédiaire sur un site `B`.

types où certaines de ces restrictions sont relâchées, notamment en ce qui concerne la possibilité de multiples variables de sites dans les schémas de type.

5.2 Exemples

5.2.1 Typage d'équations

Reprenons l'exemple proposé pour l'illustration de la sémantique annotée :

```
    y = x + 1
and z = y + 2 at B
```

Supposons maintenant que le typage de ce programme soit effectué dans l'environnement $H_0 = [x : b \texttt{ at } A]$, qui signifie que ce programme est exécuté dans un environnement dans lequel `x` est dans un premier temps disponible sur `A`.

L'expression `y + 2` étant localisée par le programmeur sur `B`, le type de `y` et de `z` est b `at` B. Il y a ensuite deux possibilités de typage, qui correspondent aux deux répartitions possibles de ce programme, telles que décrites en section 4.4.

– La première possibilité correspond à la communication de `x` de `A` vers `B`, suivie de l'exécution de la première addition sur `B`. Dans ce cas, par sous-typage, l'expression `x` est considérée comme étant de type b `at` B. Les deux instances de l'opération `(+)` reçoivent donc le type spatial $(b \times b)$ `at` $B \prec\!\{B\}\!\succ\!\rightarrow b$ `at` B.

– La deuxième possibilité correspond à l'exécution de la première addition sur `A`, suivie de la communication de son résultat de `A` vers `B`, permettant la définition de `y`. Dans ce cas, la première instance de l'opération `(+)` reçoit le type spatial b `at` $A \prec\!\{A\}\!\succ\!\rightarrow b$ `at` A. L'expression `x + 1` reçoit le type b `at` A, et est sous-typée en b `at` B pour la définition de `y`.

5.2.2 Typage des nœuds

Résolution des localisations non déterminées

Encapsulons maintenant les équations de la section 5.2.1 dans un nœud, avec la définition de `x` :

```
node f(w) = z where
      x = w at A
  and y = x + 1
  and z = y + 2 at B
```

Nous rappelons ici que le but de l'analyse de types est de permettre de répartir le programme de manière modulaire. Cela signifie que la répartition produira, pour ce nœud `f`, et pour chaque site s de l'architecture, un unique nœud $\mathtt{f}_s$ contenant seulement la partie du code de `f` à exécuter sur s. Par conséquent, il est nécessaire, au moment du typage de `f`, de lever l'indéterminisme exhibé dans la section précédente. Une des deux possibilités de typage de ces équations sera donc choisie statiquement, et ce pour toutes les instances futures de `f`.

Dans l'exemple ci-dessus, l'entrée `w` de `f` doit être disponible sur le site `A`, et sa sortie `z`, après évaluation de `f`, sera disponible sur `B`. Les deux sites `A` et `B` participent à l'évaluation de `f`. Le type de `f` est donc b `at` $A \prec\!\{A, B\}\!\succ\!\rightarrow b$ `at` B.

Supposons maintenant que, pour le même nœud, la localisation de `x` ne soit pas contrainte par le programmeur :

```
node f(x) = z where
      y = x + 1
  and z = y + 2 at B
```

Les flots `y` et `z` seront là encore localisés sur `B`. L'expression `x + 1` peut donc, en théorie, être évaluée sur n'importe quel site d'où le site `B` est accessible. Cependant, l'opération de répartition n'est définie que sur des nœuds dont toutes les valeurs, expressions et équations, sont annotées par un unique site de l'architecture. Le type spatial de ce nœud sera alors de la forme b `at` $s \prec\!\{s, B\}\!\succ\!\rightarrow b$ `at` B, où s peut être n'importe quel site de l'architecture depuis lequel le site B est accessible. Le système de types permet d'inférer un tel type ; cependant, pour des raisons pratiques d'intelligibilité et de prévisibilité par le programmeur des types inférés,

l'inférence implémentée tentera de minimiser le nombre de sites présents dans le type des nœuds. Ainsi, le type inféré pour ce nœud sera b `at` $B \prec\!\{B\}\!\succ\!\rightarrow b$ `at` B. De même, le système de types permet, dans les nœuds précédents, la communication de `z` entre sa définition et son utilisation en tant que sortie du nœud. En pratique, une sortie sera localisée sur le site même où elle est définie.

Polymorphisme et ordre supérieur

Le nœud `h` suivant est la composition de deux nœuds passés en paramètre, le premier étant évalué sur `A` et le deuxième sur `B` :

```
node h(f,g,x) = z where
      y = f(x) at A
  and z = g(y) at B
```

De même que pour les équations précédentes, `x` est inféré comme devant être disponible sur le site `A` pour l'application de `h`, et `y` après application de `h` sera disponible sur `B`. `f` est un nœud centralisé et disponible sur le site `A`, et `g` un nœud centralisé sur le site `B`. Les types des entrées et des sorties de ces deux nœuds sont n'importe quel type centralisé, et sont donc représentés par des variables de type α, β, γ. Le type spatial de `h` est donc :

$$\texttt{h} : \forall \alpha, \beta, \gamma.$$
$$\Big((\alpha \texttt{ at } A \prec\!\{A\}\!\succ\!\rightarrow \beta \texttt{ at } A) \times (\beta \texttt{ at } B \prec\!\{B\}\!\succ\!\rightarrow \gamma \texttt{ at } B) \times \alpha \texttt{ at } A\Big)$$
$$\prec\!\{A, B\}\!\succ\!\rightarrow \gamma \texttt{ at } B$$

Les variables de types correspondent, dans tous les cas, à des types représentant des valeurs entièrement centralisées : l'opération de projection, présentée au chapitre 6, est dépendante de la localisation des valeurs, et nécessite donc que toutes les valeurs soient annotées par le ou les sites où elles sont présentes. Par conséquent, un schéma de type de la forme $\forall \alpha.\alpha \prec\!\ell\!\succ\!\rightarrow \alpha$, où α peut être instancié par n'importe quel type comprenant n'importe quels sites, ne peut pas être traité à la projection.

La première conséquence importante est l'absence de principalité des types inférés : il n'est pas possible d'exprimer le type le plus général d'un nœud polymorphe. Cela est directement dû à la forme de l'opération de projection présentée au chapitre 6.

Polymorphisme de sites

Le polymorphisme de la section précédente concernait les types, au sens où une variable de type peut être instanciée par n'importe quel type centralisé. Le

polymorphisme de sites concerne le fait de pouvoir instancier un site apparaissant dans un type spatial par n'importe quel site de l'architecture.

L'utilisation du polymorphisme dans les systèmes de types a pour but, usuellement, d'inférer le type le plus général, afin de permettre l'instanciation de valeurs dans un contexte le moins contraint possible. Cependant, dans le contexte considéré ici, où le système de types est utilisé pour une opération particulière, cette préoccupation rentre en conflit avec la définition de l'opération de projection. Pour illustration, prenons l'exemple ci-dessous :

```
node f(g,x) = y where
  y = g(x)
```

Le type le plus général de ce nœud devrait être ici de la forme :

$$\texttt{f} : \forall \alpha, \beta. \forall \ell_1, \ell_2. \Big((\alpha \prec\!\langle \ell_1 \rangle\!\succ \beta) \times \alpha \Big) \prec\!\langle \ell_2 \rangle\!\succ \beta$$

Cependant, l'utilisation d'un tel type pose deux problèmes vis-à-vis de la répartition automatique guidée par les types :

- Le premier problème concerne les ensembles de sites ℓ_1 et ℓ_2. Compte tenu de la forme de l'opération de projection utilisée pour la répartition de `f`, ces ensembles doivent être fixés et connus dès sa définition, et non seulement à son instanciation. Il ne peut donc y avoir de quantification de variables d'ensembles de sites.
- Le deuxième problème concerne l'instanciation de `g` : la répartition ne peut être effectuée si les sites contenus dans les instances de α et β ne sont pas connus. Il est donc nécessaire que le type de `g` inféré soit de la forme α `at` $s \prec\!\langle \ell \rangle\!\succ \beta$ `at` s'. D'autre part, toujours pour des raisons de répartition, il est nécessaire de connaître le nombre et la direction des communications d'une fonction à son instanciation. Il est donc nécessaire ici de supposer que la fonction `g` ne comporte aucune communication : elle doit donc être entièrement localisée sur un unique site, et son type doit être de la forme α `at` $s \prec\!\langle \{s\} \rangle\!\succ \beta$ `at` s.

Par conséquent, le nœud `f` recevra le type :

$$\texttt{f} : \forall \alpha, \beta. \forall \delta. \Big((\alpha \ \texttt{at}\ \delta \prec\!\langle \{\delta\} \rangle\!\succ \beta \ \texttt{at}\ \delta) \times \alpha \ \texttt{at}\ \delta \Big) \prec\!\langle \{\delta\} \rangle\!\succ \beta \ \texttt{at}\ \delta$$

Lors de la projection, ce nœud sera simplement dupliqué sur tous les sites de l'architecture.

De même, reprenons le nœud de la section précédente en enlevant toutes les annotations de sites :

```
node h(f,g,x) = z where
      y = f(x)
  and z = g(y)
```

On a vu précédemment que le type spatial des valeurs d'un nœud partiellement annoté était contraint à l'analyse de ce nœud, et sans tenir compte de ses instanciations futures, et que ce type spatial ne contenait pas de variables de sites. De même, en absence d'annotations, le type spatial d'un nœud sera contraint de manière à ce que ce nœud soit centralisé, et dupliqué sur tous les sites de l'architecture. Le type du nœud `h` est donc :

$$\texttt{h} : \forall \alpha, \beta, \gamma . \forall \delta . \Big((\alpha \ \texttt{at} \ \delta \prec\!\{\delta\}\!\succ\!\rightarrow \beta \ \texttt{at} \ \delta) \times (\beta \ \texttt{at} \ \delta \prec\!\{\delta\}\!\succ\!\rightarrow \gamma \ \texttt{at} \ \delta) \times \alpha \ \texttt{at} \ \delta \Big) \prec\!\{\delta\}\!\succ\!\rightarrow \gamma \ \texttt{at} \ \delta$$

Suite à ces deux dernières sections, on peut remarquer que la méthode de répartition proposée impose donc une restriction majeure sur l'utilisation de l'ordre supérieur : les nœuds passés en paramètre d'autres nœuds seront forcément des nœuds centralisés.

5.3 Formalisation

La syntaxe des types spatiaux est la suivante :

$$\begin{array}{rcl} \sigma & ::= & \forall \alpha_1, \ldots, \alpha_n . \forall \delta . t \mid \forall \alpha_1, \ldots, \alpha_n . t \\ t & ::= & t \prec\!\langle \ell \rangle\!\succ\!\rightarrow t \mid t \times t \mid tc \ \texttt{at} \ s \\ tc & ::= & b \mid \alpha \mid tc \rightarrow tc \mid tc \times tc \\ \ell & ::= & \{s_1, \ldots, s_n\} \\ s & ::= & \delta \mid A \\ H & ::= & H \ \texttt{at} \ A \mid [x_1 : \sigma_1, \ldots, x_n : \sigma_n] \end{array}$$

On note H les environnements de typage, associant un type spatial à un nom. Pour localiser une expression e sur un site A, celle-ci sera typée dans un environnement modifié noté H `at` A, appelé *environnement localisé*. Toutes les instances des types issus d'un environnement localisé sur A représentent des valeurs entièrement localisées sur A.

Les schémas de types σ sont des types dont les variables de types (α) et les variables de sites (δ) sont quantifiées. Pour le moment, le nombre de variables de sites d'un schéma de types spatiaux est limité à un. Cette limitation est nécessaire pour la définition de l'opération de projection ; nous verrons au chapitre 8 comment elle peut être levée.

La distinction est faite entre les types spatiaux (t) et les types centralisés (tc). Les types spatiaux représentent des valeurs éventuellement réparties sur plusieurs sites, alors que les types centralisés représentent des valeurs entièrement localisées sur un unique site.

Un type spatial peut être un type de nœud ($t \prec\!\ell\!\succ\!\rightarrow t$), une paire répartie ($t \times t$), ou un type représentant une valeur localisée sur un site s (tc `at` s).

Un type centralisé peut être un type flot (noté b : par exemple, `int`, `bool`...), une variable de type (α), un type de nœud entièrement localisé ($tc \rightarrow tc$) ou un type de paire centralisée ($tc \times tc$).

On impose ainsi que tous les types d'un programme soient des arbres dont tous les chemins de la racine aux feuilles comportent un et un seul site. Cela correspond au fait que, pour répartir un nœud, il est nécessaire que toutes les valeurs de ce nœud soient annotées, et que leur localisation soit entièrement déterminée.

Afin de permettre l'instanciation du type spatial α `at` s par un type de la forme tc_1 `at` $s \times tc_2$ `at` s, ou encore tc_1 `at` $s \prec\!\{s\}\!\succ\!\rightarrow tc_2$ `at` s, l'égalité entre types spatiaux est étendue comme suit :

$$\begin{aligned}(tc_1 \times tc_2)\ \texttt{at}\ s &= (tc_1\ \texttt{at}\ s) \times (tc_2\ \texttt{at}\ s)\\ (tc_1 \rightarrow tc_2)\ \texttt{at}\ s &= (tc_1\ \texttt{at}\ s) \prec\!\{s\}\!\succ\!\rightarrow (tc_2\ \texttt{at}\ s)\end{aligned}$$

$$\frac{t_1 = t_1' \quad t_2 = t_2'}{(t_1 \times t_2) = (t_1' \times t_2')} \qquad \frac{t_1 = t_1' \quad t_2 = t_2'}{t_1 \prec\!\ell\!\succ\!\rightarrow t_2 = t_1' \prec\!\ell\!\succ\!\rightarrow t_2'}$$

Cette relation d'égalité permet de garder le polymorphisme de type, tout en permettant au système de types d'annoter avec un unique site toutes les valeurs du programme, permettant ainsi la répartition.

La relation $\leq$ permet d'instancier un type quantifié. Cette relation est définie, pour les environnements non localisés, comme suit :

$$t[tc_1/\alpha_1, \ldots, tc_n/\alpha_n] \leq \forall \alpha_1 \ldots \alpha_n.t$$

$$t[tc_1/\alpha_1, \ldots, tc_n/\alpha_n, s/\delta] \leq \forall \alpha_1 \ldots \alpha_n.\forall \delta.t$$

Les ensembles des variables de types et de variables de sites libres d'un type t sont notées respectivement $\mathrm{FTV}(t)$ et $\mathrm{FLV}(t)$. Ces deux fonctions sont étendues de manière directe aux environnements de types.

La généralisation d'un type t en schéma de type, dans un environnement de typage H, est notée $\mathrm{gen}_H(t)$. Cette fonction de généralisation est définie seulement si le nombre de variables de sites apparaissant dans t est au plus de un. De plus, dans ce dernier cas, le type t doit être de la forme t' `at` δ.

$$\begin{cases}\mathrm{gen}_H(t) = \forall \alpha_1, \ldots, \alpha_n.\forall \delta.tc\ \texttt{at}\ \delta & \text{ssi } t = tc\ \texttt{at}\ \delta \text{ et } \delta \notin \mathrm{FLV}(H)\\ \mathrm{gen}_H(t) = \forall \alpha_1, \ldots, \alpha_n.t & \text{ssi } \mathrm{FLV}(t) = \emptyset\end{cases}$$
$$\text{où } \{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(t) - \mathrm{FTV}(H)$$

La fonction locations($\cdot$) donne l'ensemble des sites apparaissant dans un type spatial. Elle est définie comme suit :

$$\begin{array}{ll} \text{locations}(t_1 \times t_2) & = \text{locations}(t_1) \cup \text{locations}(t_2) \\ \text{locations}(t_1 \prec\!\ell\!\succ\!\rightarrow t_2) & = \ell \\ \text{locations}(tc\ \texttt{at}\ s) & = \{s\} \end{array}$$

Par construction du système de types, on aura locations($t_1 \prec\!\ell\!\succ\!\rightarrow t_2$) $= \ell \supseteq$ locations(t_1) $\cup$ locations(t_2).

Cette fonction est utilisée à l'instanciation des variables. L'instanciation d'une variable `x`, de type spatial t, à pour effet initial, c'est-à-dire pour ensemble de sites impliqués dans l'instanciation, l'ensemble locations(t).

Le typage d'un programme est effectué dans un environnement initial H_0 :

$$H_0 = \left[\begin{array}{r} \cdot\ \texttt{fby}\ \cdot\ : \forall\alpha.\forall\delta.(\alpha \times \alpha \rightarrow \alpha)\ \texttt{at}\ \delta, \\ \texttt{if}\ \cdot\ \texttt{then}\ \cdot\ \texttt{else}\ \cdot\ : \forall\alpha.\forall\delta.((b \times \alpha \times \alpha) \rightarrow \alpha)\ \texttt{at}\ \delta, \\ (+)\ : \forall\delta.(b \times b \rightarrow b)\ \texttt{at}\ \delta, \\ \ldots \end{array} \right]$$

Un délai `fby` est évalué sur deux arguments présents sur le même site ; son résultat est sur ce même site.

Un `if/then/else` prend trois arguments sur le même site, et renvoie une valeur sur ce même site. Enfin, les opérations binaires comme l'addition prennent en argument deux flots disponibles sur un site, et ont pour résultat un flot disponible sur ce même site.

Le système de types spatiaux est défini par trois prédicats, définissant respectivement les types spatiaux des programmes, des nœuds, des ensembles d'équations et des expressions :

$$H \vdash P : H' \qquad H|G \vdash d : H'/\ell \qquad H|G \vdash D : H'/\ell \qquad H|G \vdash e : t/\ell$$

- $H \vdash P : H'$ signifie que dans l'environnement de typage H, le programme P définit un nouvel environnement de typage H'. H représente ici le type des entrées de P et H' le type de ses sorties.
- $H|G \vdash d : H'/\ell$ signifie que, dans l'environnement de typage H, et l'architecture G, la séquence de nœuds d définit un nouvel environnement de typage H', ces nœuds utilisant l'ensemble de sites ℓ.
- $H|G \vdash D : H'/\ell$ signifie que, dans l'environnement de typage H, et l'architecture G, l'ensemble d'équations D définit un nouvel environnement de typage H' et que l'évaluation de ces équations implique l'ensemble de sites ℓ.
- $H|G \vdash e : t/\ell$ signifie que, dans l'environnement de typage H, et l'architecture G, l'expression e est de type spatial t et que l'évaluation de cette expression implique l'ensemble de sites ℓ.

Ces prédicats sont définis par les règles de la figure 5.2. Par ailleurs, la définition de ces prédicats utilise le prédicat $G \vdash \mathcal{A} : G'$, défini à la section 4.3.

- Règle Prog : typer un programme $\mathcal{A}; d; D$ revient à construire le graphe d'architecture à partir de sa déclaration $\mathcal{A}$, puis à construire l'environnement de typage résultant de la séquence de nœuds d, puis à typer l'ensemble d'équations D. Ces dernières équations sont mutuellement récursives, ce qui explique le fait qu'elles soient typées dans l'environnement élargi à H'.
- Règle Imm : une valeur immédiate est typée de telle manière qu'elle soit disponible sur n'importe quel site s, si le typage intervient dans un environnement H non localisé.
- Règle Imm-Loc : une valeur immédiate typée dans un environnement localisé sur le site A est disponible sur le site A.
- Règle Inst : le type d'une variable est instancié depuis l'environnement de typage ; l'ensemble des sites nécessaires à cette instanciation est l'ensemble des sites compris dans le type instancié.
- Règle Pair : l'ensemble des sites nécessaires à la construction d'une paire est l'union des sites nécessaires à l'évaluation de ses composantes.
- Règle Op : un opérateur est une fonction centralisée sur un site s.
- Règle At : une expression annotée par le programmeur comme étant localisée sur le site A est typée dans un environnement localisé.
- Règle Comm : cette règle est la règle de sous-typage, permettant d'insérer une communication dans le programme. Une expression e dont la valeur est localisée sur un site s, peut être considérée comme une valeur localisée sur n'importe quel site s' accessible depuis s. Cette règle peut être appliquée à plusieurs reprises sur une même valeur, permettant ainsi d'accéder à un site par communication à travers un site intermédiaire.
- Règle Node : le typage d'un nœud consiste à typer le corps de ce nœud, et à généraliser par rapport à l'environnement H le type $t \prec\!\langle \ell \rangle\!\succ t_1$, où t est le type de l'entrée du nœud, t_1 le type de son résultat, et ℓ l'ensemble de sites nécessaires à l'évaluation du corps du nœud.
- Règle Def : le typage d'une équation $p = e$ vérifie que le type de e correspond au motif p, et construit un environnement de typage associant chaque variable de p au type correspondant.
- Règle App : l'ensemble des sites nécessaires à l'évaluation de l'application d'un nœud f de type $t \prec\!\langle \ell_1 \rangle\!\succ t'$ est l'union de ℓ_1 et de l'ensemble des sites nécessaires à l'évaluation de l'expression en argument de f.
- Règle And : deux ensembles d'équations en parallèle sont mutuellement récursifs. L'ensemble des sites nécessaires pour leur évaluation est l'union de l'ensemble des sites nécessaires à l'évaluation de chaque composante.
- Règle Let : l'environnement de typage construit pour typer les équations

D_1 est utilisé pour typer les équations D_2. L'ensemble des sites nécessaires est l'union des sites nécessaires à l'évaluation de D_1 et D_2.

5.4 Illustration

Pour illustration, nous reprenons l'exemple des équations vu en section 5.2.1. La figure 5.3 montre l'application du système de types spatiaux à ces équations.

$$
\begin{array}{c}
\dfrac{\begin{array}{c} b \texttt{ at } A \times b \texttt{ at } A \prec\!\{A\}\!\succ\!\rightarrow b \texttt{ at } A \\ \leq \forall\delta.b \texttt{ at } \delta \times b \texttt{ at } \delta \prec\!\{\delta\}\!\succ\!\rightarrow b \texttt{ at } \delta \end{array}}{H_1|G \vdash (+) : b \texttt{ at } A \times b \texttt{ at } A \prec\!\{A\}\!\succ\!\rightarrow b \texttt{ at } A/\{A\}}\ (\textsc{Inst}) \\
\vdots \qquad H_1|G \vdash x : b \texttt{ at } A/\{A\} \qquad H_1|G \vdash 1 : b \texttt{ at } A/\{A\} \\
\hline
H_1|G \vdash x+1 : b \texttt{ at } A/\{A\} \\
\hline
H_1|G \vdash x+1 : b \texttt{ at } B/\{A,B\} \\
\hline
H_1|G \vdash y = x+1 : [y : b \texttt{ at } B]/\{A,B\} \\
\vdots \\
\dfrac{\begin{array}{c} b \texttt{ at } B \times b \texttt{ at } B \prec\!\{B\}\!\succ\!\rightarrow b \texttt{ at } B \\ \leq \forall\delta.b \texttt{ at } \delta \times b \texttt{ at } \delta \prec\!\{\delta\}\!\succ\!\rightarrow b \texttt{ at } \delta \end{array}}{H_1|G \vdash (+) : b \texttt{ at } B \times b \texttt{ at } B \prec\!\{B\}\!\succ\!\rightarrow b \texttt{ at } B/\{B\}}\ (\textsc{Inst}) \\
\vdots \qquad H_1|G \vdash y : b \texttt{ at } B/\{B\} \qquad H_1|G \vdash 2 : b \texttt{ at } B/\{B\} \\
\hline
H_1|G \vdash y+2 : b \texttt{ at } B/\{B\} \\
\hline
H_1|G \vdash y+2 \texttt{ at } B : b \texttt{ at } B/\{B\} \\
\hline
H_1|G \vdash z = y+2 \texttt{ at } B : [z : b \texttt{ at } B]/\{B\} \\
\hline
H_1|G \vdash \begin{array}{c} y = x+1 \\ \texttt{and} \quad z = y+2 \texttt{ at } B \end{array} : \left[\begin{array}{c} y : b \texttt{ at } B, \\ z : b \texttt{ at } B \end{array}\right]/\{A,B\}
\end{array}
\begin{array}{l}
(\textsc{Op}) \\
(\textsc{Comm}) \\
(\textsc{Def}) \\
\\
(\textsc{Op}) \\
(\textsc{At}) \\
(\textsc{Def}) \\
(\textsc{And})
\end{array}
$$

Avec $H_1 = H_0, [x : b \texttt{ at } A, y : b \texttt{ at } B, z : b \texttt{ at } B]$ et $G = \langle\{A,B\},\{(A,B)\}\rangle$.

FIG. 5.3 – Exemple d'application du système de types spatiaux.

De même que ces équations acceptent plusieurs sémantiques annotées, il y a plusieurs dérivations possibles pour leur typage. Ainsi, pour l'évaluation de la première addition sur B au lieu de A, le début de l'arbre de dérivation placerait la communication de `x` avant cette addition, en appliquant la règle de sous-typage

(Prog)

$$\frac{\langle\emptyset,\emptyset\rangle \vdash \mathcal{A} : \langle\mathcal{S},\mathcal{L}\rangle \qquad H, H_0|\langle\mathcal{S},\mathcal{L}\rangle \vdash d : H_1/\ell \qquad H, H_1, H'\langle\mathcal{S},\mathcal{L}\rangle \vdash D : H'/\ell' \qquad \ell' \subseteq \mathcal{S}}{H \vdash \mathcal{A}; d; D : H'}$$

(Arch)

$$\frac{G \vdash \mathcal{A}_1 : G_1 \qquad G_1 \vdash \mathcal{A}_2 : G_2}{G \vdash \mathcal{A}_1; \mathcal{A}_2 : G_2}$$

(Def-Site)

$$\langle\mathcal{S},\mathcal{L}\rangle \vdash \texttt{loc}\ A : \langle\mathcal{S}\cup\{A\},\mathcal{L}\rangle$$

(Def-Link)

$$\frac{A_1, A_2 \in \mathcal{S}}{\langle\mathcal{S},\mathcal{L}\rangle \vdash \texttt{link}\ A_1\ \texttt{to}\ A_2 : \langle\mathcal{S},\mathcal{L}\cup\{A_1 \mapsto A_2\}\rangle}$$

(Imm)

$$H|\langle\mathcal{S},\mathcal{L}\rangle \vdash i : b\ \texttt{at}\ s/\{s\}$$

(Inst)

$$\frac{t \leq H(x)}{H|G \vdash x : t/\,\mathrm{locations}(t)}$$

(Pair)

$$\frac{H|G \vdash e_1 : t_1/\ell_1 \qquad H|G \vdash e_2 : t_2/\ell_2}{H|G \vdash (e_1, e_2) : t_1 \times t_2/\ell_1 \cup \ell_2}$$

(Op)

$$\frac{H|G \vdash \mathrm{op} : t_1 \prec\!\{s\}\!\succ\!\rightarrow t_2/\{s\} \qquad H|G \vdash e : t_1/\ell}{H|G \vdash \mathrm{op}(e) : t_2/\{s\} \cup \ell}$$

(At)

$$\frac{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : t/\{A\} \qquad A \in \mathcal{S}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e\ \texttt{at}\ A : t/\{A\}}$$

(Comm)

$$\frac{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : tc\ \texttt{at}\ s/\ell \qquad (s, s') \in \mathcal{L}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : tc\ \texttt{at}\ s'/\ell \cup \{s'\}}$$

(Node)

$$\frac{H, x : t_1, H_1|G \vdash D : H_1/\ell_1 \qquad H, x : t_1, H_1|G \vdash e : t_2/\ell_2}{H|G \vdash \texttt{node}\ f(x) = e\ \texttt{where}\ D : [\mathrm{gen}_H(t_1 \prec\!\ell_1 \cup \ell_2\!\succ\!\rightarrow t_2)/f]/\ell_1 \cup \ell_2}$$

(Def)

$$\frac{H|G \vdash e : t_1 \times \ldots \times t_n/\ell}{H|G \vdash (x_1, \ldots, x_n) = e : [t_1/x_1, \ldots, t_n/x_n]/\ell}$$

(App)

$$\frac{H|G \vdash f : t_1 \prec\!\ell_1\!\succ\!\rightarrow t_2/\ell_2 \qquad H|G \vdash e : t_1/\ell_3}{H|G \vdash x = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3}$$

(And)

$$\frac{H|G \vdash D_1 : H_1/\ell_1 \qquad H|G \vdash D_2 : H_2/\ell_2}{H|G \vdash D_1\ \texttt{and}\ D_2 : H_1, H_2/\ell_1 \cup \ell_2}$$

(Let)

$$\frac{H|G \vdash D_1 : H_1/\ell_1 \qquad H, H_1|G \vdash D_2 : H_2/\ell_2}{H|G \vdash \texttt{let}\ D_1\ \texttt{in}\ D_2 : H_2/\ell_1 \cup \ell_2}$$

Fig. 5.2 – Système de types spatial.

juste après l'instanciation de `x` :

$$
(\textsc{App})\ \dfrac{\dfrac{\dfrac{\begin{array}{c} b\ \texttt{at}\ B \times b\ \texttt{at}\ B \prec\!\{B\}\!\succ\!\rightarrow b\ \texttt{at}\ B \\ \leq \forall\delta.b\ \texttt{at}\ \delta \times b\ \texttt{at}\ \delta \prec\!\{\delta\}\!\succ\!\rightarrow b\ \texttt{at}\ \delta \end{array}}{H|G \vdash (+) : b\ \texttt{at}\ B \times b\ \texttt{at}\ B \prec\!\{B\}\!\succ\!\rightarrow b\ \texttt{at}\ B/\{B\}}\ (\textsc{Inst}) \quad \vdots \quad (\textsc{Comm})\ \dfrac{H|G \vdash x : b\ \texttt{at}\ A/\{A\}}{H|G \vdash x : b\ \texttt{at}\ B/\{A,B\}} \qquad H|G \vdash 1 : b\ \texttt{at}\ B/\{B\}}{H|G \vdash x+1 : b\ \texttt{at}\ B/\{A,B\}}}{(\textsc{Def})\ \dfrac{H|G \vdash x+1 : b\ \texttt{at}\ B/\{A,B\}}{H|G \vdash y = x+1 : [y : b\ \texttt{at}\ B]/\{A,B\}}}
$$

5.5 Adaptation pour architectures hétérogènes

L'adaptation de ce système de types pour les architectures hétérogènes consiste à considérer que l'architecture définit elle-même un environnement de typage. Chaque site définit ainsi un environnement de typage associant aux opérations déclarées sur ce site, leur type spatial donné par leur type centralisé localisé sur ce site.

On modifie donc le prédicat définissant le graphe d'architecture : on note $G/H \vdash \mathcal{A} : G'/H'$ le fait que l'architecture $\mathcal{A}$, à partir du graphe G et de l'environnement de typage H, définit le nouveau graphe d'architecture G' et l'environnement de typage H', composé de H étendu avec le type spatial des opérations définies dans les sites de l'architecture $\mathcal{A}$. Ce prédicat est défini en figure 5.4.

$$
(\textsc{Arch})\ \frac{G/H \vdash \mathcal{A}_1 : G_1/H_1 \qquad G_1/H_1 \vdash \mathcal{A}_2 : G_2/H_2}{G/H \vdash \mathcal{A}_1 ; \mathcal{A}_2 : G_2/H_2}
$$

$$
(\textsc{Def-Site})\ \langle \mathcal{S}, \mathcal{L} \rangle / H \vdash \boxed{\begin{array}{l} \texttt{loc}\ A = \texttt{sig} \\ \quad x_1 : tc_1 \\ \quad \vdots \\ \quad x_n : tc_n \\ \texttt{end} \end{array}} : \langle \mathcal{S} \cup \{A\}, \mathcal{L} \rangle / \left[\begin{array}{c} tc_1\ \texttt{at}\ A/x_1, \\ \vdots \\ tc_n\ \texttt{at}\ A/x_n \end{array} \right]
$$

$$
(\textsc{Def-Link})\ \frac{A_1, A_2 \in \mathcal{S}}{\langle \mathcal{S}, \mathcal{L} \rangle / H \vdash \texttt{link}\ A_1\ \texttt{to}\ A_2 : \langle \mathcal{S}, \mathcal{L} \cup \{A_1 \mapsto A_2\} \rangle / H}
$$

FIG. 5.4 – Système de types spatial modifié pour les architectures hétérogènes.

Le prédicat définissant le type d'un programme est modifié comme suit :

(Prog)

$$\frac{\begin{array}{c}\langle\emptyset,\emptyset\rangle/H_0 \vdash \mathcal{A} : \langle\mathcal{S},\mathcal{L}\rangle/H_1 \\ H,H_1|\langle\mathcal{S},\mathcal{L}\rangle \vdash d : H_2/\ell \qquad H,H_2,H'|\langle\mathcal{S},\mathcal{L}\rangle \vdash D : H'/\ell' \qquad \ell' \subseteq \mathcal{S}\end{array}}{H \vdash \mathcal{A};d;D : H'}$$

5.6 Correction du système de types

La correction du système de types spatial consiste à montrer qu'un programme accepté et annoté par le système de types de la figure 5.2 peut être exécuté de manière répartie dans l'architecture déclarée.

Définition 1 (Compatibilité entre valeurs et type spatiaux)**.** *La compatibilité d'une valeur répartie $\hat{v}$ et d'un type t, dans un environnement de typage H, est définie à l'aide d'une interprétation I_H, définie comme suit :*

- *vl `at` $s \in I_H(b$ `at` $s)$*
- *$(\hat{v}_1,\hat{v}_2) \in I_H(t_1 \times t_2)$ ssi $\hat{v}_1 \in I_H(t_1)$ et $\hat{v}_2 \in I_H(t_2)$*
- *$\lambda x.e$ `where` $D \in I_H(t_1 \prec\!\langle\ell\rangle\!\succ t_2)$ ssi il existe H_1,ℓ_1,ℓ_2, pour tous R, v, $\hat{v}$, v', $\hat{v}'$, R', $\hat{R}'$, D', e' tels que :*
  - *$\ell = \ell_1 \cup \ell_2$,*
  - *$H,y:t_1|G \vdash D : H_1/\ell_1$, $H,y:t_1,H_1|G \vdash e : t_2/\ell_2$,*
  - *$R,[v/y] \vdash D \xrightarrow{R'} D'$, $R,[v/y],R' \vdash e \xrightarrow{v'} e'$,*

  il existe $\hat{R},\hat{v},\hat{R}',\hat{v}'$ tels que :
  - *$|\hat{R}| = R$, $\hat{R} : H$, $|\hat{R}'| = R'$, $\hat{R}' : H_1$,*
  - *$|\hat{v}| = v$, $\hat{v} \in I_H(t_1)$, $|\hat{v}'| = v'$, $\hat{v}' \in I_H(t_2)$,*
  - *$\hat{R},[\hat{v}/y] \overset{\ell_1}{\Vdash} D \xrightarrow{\hat{R}'} D'$, $\hat{R},[\hat{v}/y],\hat{R}' \overset{\ell_2}{\Vdash} e \xrightarrow{\hat{v}'} e'$,*
  - *$H,y:t_1|G \vdash D' : H_1/\ell_1$ et $H,y:t_1,H_1|G \vdash e' : t_2/\ell_2$*
- *$\hat{v} \in I_H(\sigma)$ ssi pour tout $t \leq \sigma$, $\hat{v} \in I_H(t)$.*

Cette relation de typage des valeurs réparties est étendue aux environnements de réaction : on note $\hat{R} : H$ la compatibilité entre l'environnement de réaction $\hat{R}$ et l'environnement de typage H :

$$\hat{R} : H \Leftrightarrow \forall x \in \mathrm{dom}(\hat{R}), x \in \mathrm{dom}(H) \wedge \hat{R}(x) \in I_H(H(x))$$

Enfin, cette relation est étendue aux séquences d'environnements :

$$\hat{R}.\hat{S} : H \Leftrightarrow \hat{R} : H \wedge \hat{S} : H$$

Le lemme suivant permet d'établir l'unicité des valeurs réparties liées par la relation de compatibilité de types, et la relation $|\cdot|$.

Lemme 2. *Pour tous* $\hat{v}, \hat{v}', t, H$*, si* $|\hat{v}| = |\hat{v}'|$*,* $\hat{v} \in I_H(t)$ *et* $\hat{v}' \in I_H(t)$*, alors* $\hat{v} = \hat{v}'$*.*

Démonstration. Par induction sur la structure de $\hat{v}$:
- Si $\hat{v} = vl$ `at` s :
  Alors il existe tc tel que $t = tc$ `at` s.
  $\hat{v}' \in I_H(t)$ ssi il existe vl' tel que $\hat{v}' = vl'$ `at` s.
  $|\hat{v}| = vl$ et $|\hat{v}'| = vl'$. $|\hat{v}| = |\hat{v}'| \Rightarrow vl = vl'$.
- Si $\hat{v} = (\hat{v}_1, \hat{v}_2)$:
  Alors il existe t_1, t_2 tels que $t = t_1 \times t_2$, $\hat{v}_1 \in I_H(t_1)$ et $\hat{v}_2 \in I_H(t_2)$.
  $\hat{v}' \in I_H(t)$ ssi il existe $\hat{v}'_1, \hat{v}'_2$ tels que $\hat{v}' = (\hat{v}'_1, \hat{v}'_2)$, $\hat{v}'_1 \in I_H(t_1)$ et $\hat{v}'_2 \in I_H(t_2)$.
  $|\hat{v}| = |\hat{v}'|$ ssi $|\hat{v}_1| = |\hat{v}'_1|$ et $|\hat{v}_2| = |\hat{v}'_2|$.
  Par hypothèse d'induction, $\hat{v}_1 = \hat{v}'_1$ et $\hat{v}_2 = \hat{v}'_2$. Par conséquent $\hat{v} = \hat{v}'$.
- Si $\hat{v} = \lambda x.e$ `where` D alors trivialement, $\hat{v}' = \hat{v}$.

□

Le résultat de correction ne concerne que les programmes possédant une sémantique synchrone centralisée : les programmes ne pouvant être exécutés de manière synchrone sont rejetés par d'autres analyses, telles que l'analyse de types standard et l'analyse de causalité [31].

Le théorème de correction établit qu'un programme réagissant avec la sémantique centralisée et accepté par le système de types spatiaux admet une exécution avec la sémantique prenant en compte les annotations de répartition. La séquence de valeurs produites par cette exécution est compatible avec le type spatial du programme.

Théorème 1 (Correction (programmes))**.** *Pour tous* P, S, S', H, H'*, si* $S \vdash P : S'$ *et* $H \vdash P : H'$*, alors il existe* $\hat{S}$ *et* $\hat{S}'$ *tels que :*
- $\hat{S} \Vdash P : \hat{S}'$,
- $\hat{S} : H$,
- $\hat{S}' : H'$,
- $|\hat{S}| = S$,
- *et* $|\hat{S}'| = S'$.

La preuve de ce théorème est basée sur deux lemmes, établissant respectivement la correction des équations et des expressions.

Le lemme 3 établit que, si une expression réagit avec la sémantique centralisée et est acceptée par le système de types spatiaux, alors cette expression peut réagir avec la sémantique répartie, en émettant une valeur compatible avec le type spatial de l'expression et égale sans annotations à la valeur émise par la sémantique centralisée. De plus, le type spatial de cette expression est préservé par cette réaction.

Lemme 3 (Correction (expressions)). *Pour tous* $e, e', H, G, t, \ell, R, v$, *si* $H|G \vdash e : t/\ell$ *et* $R \vdash e \xrightarrow{v} e'$, *alors il existe* $\hat{R}$ *et* $\hat{v}$ *tels que*

- $\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\hat{v}} e'$,
- $\hat{R} : H$,
- $\hat{v} \in I_H(t)$,
- $|\hat{R}| = R$
- $|\hat{v}| = v$,
- *et* $H|G \vdash e' : t/\ell$.

La preuve est effectuée par induction sur la structure de l'arbre de dérivation pour l'application du système de types. L'idée est d'appliquer, pour chaque règle du système de types, une règle correspondante de la sémantique annotée.

Démonstration. [Démonstration du lemme 3] Par induction sur la dernière règle du système de types spatiaux appliquée.

Cas de la règle Imm : $e = i$. Soit s tel que $H|G \vdash i : b \texttt{ at } s/\{s\}$. Quels que soient R et $\hat{R}$ tels que $|\hat{R}| = R$, nous avons $R \vdash i \xrightarrow{i} i \Rightarrow \hat{R} \overset{\{s\}}{\Vdash} i \xrightarrow{i \texttt{ at } s} i$ et $i \texttt{ at } s \in I_H(b \texttt{ at } s)$.

Cas de la règle Inst : $e = x$. Soient H, t, ℓ tels que $H|G \vdash e : t/\ell$. Soient R, v, e' tels que $R \vdash e \xrightarrow{v} e'$.

- Pour tout $\hat{R}$ tel que $|\hat{R}| = R$ et $\hat{R} : H$, alors il existe $\hat{v}$ tel que $\hat{R}(x) = \hat{v}$. De plus, $|\hat{v}| = v$ puisque $|\hat{R}| = R$, et $\hat{v} \in I_H(t)$ puisque $\hat{R} : H$. Par conséquent, $\text{locations}(t) = \text{loc}(\hat{v}) = \ell$. Alors la règle Inst de la sémantique annotée s'applique : $\hat{R}, [\hat{v}/x] \overset{\ell}{\Vdash} x \xrightarrow{\hat{v}} x$.

Cas de la règle Pair : $e = (e_1, e_2)$.

- $H|G \vdash e : t/\ell$ ssi $\exists \ell_1, \ell_2, t_1, t_2$ tels que $H|G \vdash e_1 : t_1/\ell_1$, $H|G \vdash e_2 : t_2/\ell_2$, $t = t_1 \times t_2$ et $\ell = \ell_1 \cup \ell_2$.
- $R \vdash (e_1, e_2) \xrightarrow{v} e'$ ssi $\exists v_1, v_2, e'_1, e'_2$ tels que $v = (v_1, v_2)$, $e' = (e'_1, e'_2)$, $R \vdash e_1 \xrightarrow{v_1} e'_1$ et $R \vdash e_2 \xrightarrow{v_2} e'_2$.
- Par hypothèse d'induction, il existe $\hat{R}_1, \hat{R}_2, \hat{v}_1, \hat{v}_2$ tels que :
  - $\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{\hat{v}_1} e'_1$, $\hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{\hat{v}_2} e'_2$,
  - $|\hat{R}_1| = R$, $\hat{R}_1 : H$, $|\hat{R}_2| = R$, $\hat{R}_2 : H$,
  - $|\hat{v}_1| = v_1$, $\hat{v}_1 \in I_H(t_1)$, $|\hat{v}_2| = v_2$, $\hat{v}_2 \in I_H(t_2)$,
  - $H|G \vdash e'_1 : t_1/\ell_1$ et $H|G \vdash e'_2 : t_2/\ell_2$.
- D'après le lemme 2, $\hat{R}_1 = \hat{R}_2$. Soit $\hat{R} = \hat{R}_1$.

– La règle Pair de la sémantique annotée s'applique :

$$\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} (e_1, e_2) \xrightarrow{(\hat{v}_1, \hat{v}_2)} (e'_1, e'_2),$$

avec $(\hat{v}_1, \hat{v}_2) \in I_H((t_1 \times t_2))$ et $|(\hat{v}_1, \hat{v}_2)| = (|\hat{v}_1|, |\hat{v}_2|) = (v_1, v_2)$.
– La règle Pair du système de types s'applique : $H|G \vdash e' : t/\ell$.

Cas de la règle Op : $e = \mathrm{op}(e_1)$.
– $H|G \vdash e : t/\ell$ ssi il existe $\ell_1, s, t1$ tels que $\ell = \ell_1 \cup \{s\}$,
$H|G \vdash \mathrm{op} : t_1 \prec\!\{s\}\!\succ t/\{s\}$ et $H|G \vdash e_1 : t_1/\ell_1$.
– D'après la règle Op de la sémantique centralisée, $R \vdash \mathrm{op}(e) \xrightarrow{v} e'$ ssi $\exists v_1, e'_1, \mathrm{op}'$ tels que $e' = \mathrm{op}'(e'_1)$, $R \vdash e_1 \xrightarrow{v_1} e'_1$ et $\mathrm{op}(v_1) \xrightarrow{v} \mathrm{op}'$.
– Par hypothèse d'induction, il existe $\hat{R}, s$ tels que $\hat{R} \overset{\ell}{\Vdash} e_1 \xrightarrow{v_1 \texttt{ at } s} e'_1$ et
$H|G \vdash e'_1 : t_1/\ell_1$.
– Par application de la règle Op de la sémantique annotée, nous avons :
$\hat{R} \overset{\ell}{\Vdash} \mathrm{op}(e_1) \xrightarrow{v \texttt{ at } s} \mathrm{op}'(e'_1)$.
– Par application de la règle Op du système de types, nous avons : $H|G \vdash e' : t/\ell$.

Cas de la règle At : $e = e_1 \texttt{ at } A$.
– $H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash e_1 \texttt{ at } A : t/\ell$ ssi $H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash e_1 : t/\ell$, $\ell = \{A\}$ et $A \in \mathcal{S}$.
– $R \vdash e_1 \texttt{ at } A \xrightarrow{v} e'$ ssi $\exists e'_1$ tel que $e' = e'_1 \texttt{ at } A$ et $R \vdash e_1 \xrightarrow{v} e'_1$.
– Par hypothèse d'induction, il existe $\hat{R}, \hat{v}, s$ tels que $\hat{R} \overset{\{A\}}{\Vdash} e_1 \xrightarrow{\hat{v}} e'_1$, $|\hat{R}| = R$,
$|\hat{v}| = v$, $\hat{R} : H$, $\hat{v} \in I_H(t)$ et $H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash e'_1 : t/\ell$.
– Par application de la règle At de la sémantique annotée, nous avons :
$\hat{R} \overset{\{A\}}{\Vdash} e_1 \texttt{ at } A \xrightarrow{\hat{v}} e'_1 \texttt{ at } A$.
– Par application de la règle At du système de types, nous avons :
$H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash e' : t/\ell$

Cas de la règle Comm :
– $H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash e : t \texttt{ at } s'/\ell$ ssi il existe ℓ' tel que $\ell = \ell' \cup \{s\}$,
$H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash e : t \texttt{ at } s/\ell'$ et $(s, s') \in \mathcal{L}$.
– Par hypothèse du lemme 3, $R \vdash e \xrightarrow{v} e'$.
– Par hypothèse d'induction, il existe $\hat{R}, \hat{v}, s$ tels que $\hat{R} \overset{\ell}{\Vdash} e_1 \xrightarrow{\hat{v}} e'_1$, $|\hat{R}| = R$,
$|\hat{v}| = v$, $\hat{R} : H$, $\hat{v} \in I_H(t \texttt{ at } s)$, $H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash e' : t \texttt{ at } s/\ell'$.
– D'après la définition de la relation $(\hat{v} \in I_H(t))$, $\hat{v} \in I_H(t \texttt{ at } s)$ ssi il existe vl telle que $\hat{v} = vl \texttt{ at } s$.
– Par application de la règle Comm de la sémantique annotée, nous avons donc :
$\hat{R} \overset{\ell \cup \{s\}}{\Vdash} e \xrightarrow{\hat{v}'} e'$, avec $\hat{v}' = vl \texttt{ at } s'$, d'où $\hat{v}' \in I_H(t \texttt{ at } s')$.

– Par application de la règle COMM du système de types, nous avons $H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e' : t \texttt{ at } s'/\ell$.

□

Le lemme 4 établit que, si un ensemble d'équation réagit avec la sémantique centralisée, et qu'il est accepté par le système de types spatiaux, alors il existe une exécution répartie telle que les valeurs réparties en entrées et en sorties de cette exécution sont égales aux valeurs centralisées. D'autre part, les types spatiaux de cet ensemble d'équations sont préservés par l'exécution répartie.

Lemme 4 (Correction (nœuds et équations))**.**

1. *Pour tous* H, H', G, D, e, ℓ, R, *si* $H|G \vdash \texttt{node } f(x) = e \texttt{ where } D : H'/\ell$,
   – *pour tout* R *tel qu'il existe* D', e', R', v, v', $R, [v/x] \vdash D \xrightarrow{R'} D'$ *et* $R, [v/x], R' \vdash e \xrightarrow{v'} e'$,
   – *il existe* $\hat{R}$ *tel que* $\hat{R} : H$, $|\hat{R}| = R$ *et* $(\lambda x.e \texttt{ where } D) \in I_H(H'(f))$.
2. *Pour tous* D, D', H, H', R, R', G,
   si $H|G \vdash D : H'/\ell$ *et* $R \vdash D \xrightarrow{R'} D'$, *alors il existe* $\hat{R}$ *et* $\hat{R}'$ *tels que :*
   – $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$,
   – $\hat{R} : H$,
   – $\hat{R}' : H'$,
   – $|\hat{R}| = R$,
   – $|\hat{R}'| = R'$,
   – *et* $H|G \vdash D' : H'/\ell$.

Démonstration. Par induction sur la structure de D.

Cas des nœuds :

– $H|G \vdash \texttt{node } f(x) = e \texttt{ where } D : H'/\ell$ ssi il existe $t_1, t_2, H_1, \ell_1, \ell_2$ tels que :
  – $\ell = \ell_1 \cup \ell_2$, $H' = [\sigma/f]$, $\sigma = \text{gen}_H(t_1 \prec\langle\ell\rangle\rightarrow t_2)$,
  – $H, x : t_1, H_1|G \vdash D : H_1/\ell_1$,
  – $H, x : t_1, H_1|G \vdash e : t_2/\ell_2$.

– Soit R, R', v, v' tel que $R(f) = \lambda x.e \texttt{ where } D$, $R, [v/x], R' \vdash D \xrightarrow{R'} D'$ et $R, [v/x], R' \vdash e \xrightarrow{v'} e'$.

– Par hypothèse d'induction, il existe $\hat{R}, \hat{R}', \hat{v}$ tels que :
  – $\hat{R} : H$, $|\hat{R}| = R$, $\hat{R}' : H_1$, $|\hat{R}'| = R'$,
  – $\hat{v} \in I_H(t_1)$, $|\hat{v}| = v$,
  – $\hat{R}, [\hat{v}/x], \hat{R}' \overset{\ell_1}{\Vdash} D \xrightarrow{\hat{R}'} D'$,
  – $H, x : t_1, H_1|G \vdash D' : H_1/\ell_1$.

– Par application du lemme 3, il existe $\hat{v}'$ tel que
  – $\hat{v}' \in I_H(t_2)$, $|\hat{v}'| = v'$,

  - $\hat{R}, [\hat{v}/x], \hat{R}' \overset{e}{\Vdash} \hat{v}' \xrightarrow{e'}$,
  - $H, x : t_1, H_1 | G \vdash e' : t_2/\ell_2$.
- Par conséquent, $\hat{R}(f) \in I_H(\sigma)$.

Cas $D = ((x_1, \ldots, x_n) = e)$ **:**
- $H|G \vdash (x_1, \ldots, x_n) = e : H'/\ell$ ssi $\exists s, t_1, \ldots, t_n$, tels que $H' = [t_1/x_1, \ldots, t_n/x_n]$ et $H|G \vdash e : t_1 \times \ldots \times t_n/\ell$.
- $R \vdash (x_1, \ldots, x_n) = e \xrightarrow{R'} D'$ ssi $\exists e', v_1, \ldots, v_n$, $D' = ((x_1, \ldots, x_n) = e')$, $R' = [v_1/x_1, \ldots, v_n/x_n]$, et $R \vdash e \xrightarrow{(v_1, \ldots, v_n)} e'$.
- Par application du lemme 3, $\exists \hat{R}, \hat{v}_1, \ldots, \hat{v}_n$ tels que $\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{(\hat{v}_1, \ldots, \hat{v}_n)} e'$, $|\hat{R}| = R$, $\hat{R} : H$, et pour tout $i \in \{1, \ldots, n\}$, $|\hat{v}| = v$ et $\hat{v}_i \in I_H(t_i)$.
- Par application de la règle Def, nous avons

$$\hat{R} \overset{\ell}{\Vdash} (x_1, \ldots, x_n) = e \xrightarrow{[\hat{v}_1/x_1, \ldots, \hat{v}_n/x_n]} (x_1, \ldots, x_n) = e'.$$

  Pour tout i, comme $\forall i, \hat{v} \in I_H(t \texttt{ at } s)$, nous avons donc $|[\hat{v}_1/x_1, \ldots, \hat{v}_n/x_n]| = [v_1/x_1, \ldots, v_n/x_n] = R$ et $[\hat{v}_1/x_1, \ldots, \hat{v}_n/x_n] : H'$.

Cas $D = (x = f(e))$ **:**
- $H|G \vdash x = f(e) : H'/\ell$ ssi $\exists s, t_1, t_2, \ell_1, \ell_2, \ell_3$ tels que $\ell = \ell_1 \cup \ell_2 \cup \ell_3$, $H' = [t_2/x]$, $H|G \vdash f : t_1 \prec\!\langle \ell_1 \rangle\!\succ t_2/\ell_2$ et $H|G \vdash e : t_1/\ell_3$.
- $R \vdash x = f(e) \xrightarrow{R'} D'$ ssi $\exists e_1, D_1$ tels que $R(f) = \lambda y.e_1 \texttt{ where } D_1$ et $R \vdash \texttt{let } y = e \texttt{ and } D_1 \texttt{ in } x = e_1 \xrightarrow{R'} D'$.
- $R \vdash \texttt{let } y = e \texttt{ and } D_1 \texttt{ in } x = e_1 \xrightarrow{R'} D'$ ssi il existe $v, v_1, e', e'_1, D'_1, R_1$ tels que :
  - $R' = [v/x]$,
  - $R \vdash e \xrightarrow{v} e'$,
  - $R, [v/y] \vdash D_1 \xrightarrow{R_1} D'_1$,
  - $R, [v/y], R_1 \vdash e_1 \xrightarrow{v_1} e'_1$
- Par hypothèse d'induction, $(\lambda y.e_1 \texttt{ where } D_1) \in I_H(t_1 \prec\!\langle \ell_1 \rangle\!\succ t_2)$. Par conséquent, il existe $H_1, \ell_{11}, \ell_{12}, \hat{R}, \hat{v}, \hat{R}_1, \hat{v}_1$ tels que :
  - $\ell_1 = \ell_{11} \cup \ell_{12}$,
  - $|\hat{R}| = R$, $\hat{R} : H$, $|\hat{R}_1| = R_1$, $\hat{R}_1 : H_1$,
  - $|\hat{v}| = v$, $\hat{v} \in I_H(t_1)$, $|\hat{v}_1| = v_1$, $\hat{v}_1 \in I_H(t_2)$,
  - $\hat{R}, [\hat{v}/y] \overset{\ell_{11}}{\Vdash} D_1 \xrightarrow{\hat{R}_1} D'_1$, $\hat{R}, [\hat{v}/y], \hat{R}_1 \overset{\ell_{12}}{\Vdash} e_1 \xrightarrow{\hat{v}_1} e'$,
  - $H, y : t_1 | G \vdash D'_1 : H_1/\ell_{11}$ et $H, y : t_1, H_1 | G \vdash e'_1 : t_2/\ell_{12}$.
- Par application de la règle Let de la sémantique annotée,

$$\hat{R} \overset{\ell}{\Vdash} \texttt{let } y = e \texttt{ and } D_1 \texttt{ in } x = e_1 \xrightarrow{\hat{R}'} D',$$

$|\hat{R}| = R$, $|\hat{R}'| = \hat{R}'$, $\hat{R} : H$ et $\hat{R}' : H'$.
- Par application de la règle App de la sémantique annotée, nous avons donc $\hat{R} \overset{\ell}{\Vdash} x = f(e) \xrightarrow{\hat{R}'} D'$.
- D'autre part, puisque $H, y : t_1 | G \vdash D'_1 : H_1/\ell_1$ et $H, y : t_1, H_1 | G \vdash e'_1 : t_2/\ell_2$, par application de la règle Let du système de types, on a bien $H|G \vdash D' : H'/\ell$.

Cas $D = D_1$ `and` D_2 **et** $D =$ `let` D_1 `in` D_2 **:** induction directe.

□

Démonstration du théorème 1. Soient R, S, R', S', H, H', et $P = \mathcal{A}; d; D$, tels que :
- $R.S \vdash P : R'.S'$,
- et $H \vdash P : H'$.

D'après la règle d'exécution :

$$\frac{R, R' \vdash D \xrightarrow{R'} D' \qquad S \vdash (d_1; \ldots; d_n; D') : S'}{R.S \vdash (d_1; \ldots; d_n; D) : R'.S'}$$

On a donc $R, R' \vdash D \xrightarrow{R'} D'$ et $S \vdash (d_1; \ldots; d_n; D') : S'$.
D'après la règle Prog du système de types :

(Prog)
$$\frac{\langle \emptyset, \emptyset \rangle \vdash \mathcal{A} : \langle \mathcal{S}, \mathcal{L} \rangle \qquad H, H_0 | \langle \mathcal{S}, \mathcal{L} \rangle \vdash d : H_1/\ell_1 \qquad H, H_1, H' | \langle \mathcal{S}, \mathcal{L} \rangle \vdash D : H'/\ell \qquad \ell \subseteq \mathcal{S}}{H \vdash \mathcal{A}; d; D : H'}$$

Donc, il existe $H_1, G = \langle \mathcal{S}, \mathcal{L} \rangle$ tels que $\langle \emptyset, \emptyset \rangle \vdash \mathcal{A} : \langle \mathcal{S}, \mathcal{L} \rangle$, $H, H_0 \vdash d : H_1/\ell_1$, $H, H_1, H' \vdash D : H'/\ell$ et $\ell \subseteq \mathcal{S}$.
Par application du lemme 4, il existe $\hat{R}, \hat{R}'$ tels que :
- $\hat{R}, \hat{R}' \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$,
- $\hat{R} : H, H_1$,
- $\hat{R}' : H'$,
- $|\hat{R}| = R$,
- $|\hat{R}'| = R'$,
- et $H, H_1 \vdash D' : H'/\ell$.

Par hypothèse, soient $\hat{S}$ et $\hat{S}'$ tels que :
- $\hat{S} \Vdash P : \hat{S}'$,
- $\hat{S} : H$,
- $\hat{S}' : H'$,
- $|\hat{S}| = S$,

– et $|\hat{S}'| = S'$.

Par conséquent, la règle d'exécution des programmes de la sémantique synchrone prenant en compte les annotations de répartition s'applique :

$$\frac{\langle \emptyset, \emptyset \rangle \vdash \mathcal{A} : \langle \mathcal{S}, \mathcal{L} \rangle \qquad \hat{R}, \hat{R}' \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D' \qquad \ell \subseteq \mathcal{S} \qquad \hat{S} \Vdash \mathcal{A}; d_1; \ldots; d_n; D' : \hat{S}'}{\hat{R}.\hat{S} \Vdash \mathcal{A}; d_1; \ldots; d_n; D : \hat{R}'.\hat{S}'}$$

On a donc :
– $\hat{R}.\hat{S} \Vdash P : \hat{R}'.\hat{S}'$,
– $\hat{R} : H$ et $\hat{S} : H$, alors $\hat{R}.\hat{S} : H$,
– $\hat{R}' : H'$ et $\hat{S}' : H'$, alors $\hat{R}'.\hat{S}' : H'$,
– $|\hat{R}.\hat{S}| = |\hat{R}|.|\hat{S}| = R.S$,
– et $|\hat{R}'.\hat{S}'| = |\hat{R}'|.|\hat{S}'| = R'.S'$.

□

Le système de types ainsi défini est donc correct, mais pas complet : la structure des types spatiaux, contrainte par la forme de l'opération de projection guidée par ces types, ne permet en effet ni la complétude du système ni sa principalité.

5.7 Implémentation

Le système de types présenté dans ce chapitre repose sur un mécanisme de sous-typage. Ce mécanisme est utile dans le cas où les communications peuvent avoir lieu à n'importe quel endroit du programme.

Cette situation soulève deux problèmes. Le premier est que l'implémentation de systèmes de types avec mécanisme de sous-typage est coûteux, en raison de la complexité des algorithmes usuels impliquant le calcul de contraintes entre les types apparaissant dans le programme par application systématique de la règle de sous-typage à chaque point du programme. Le second est que l'utilisation systématique du sous-typage revient à ne laisser aucun contrôle au programmeur quant à la localisation des communications.

Nous pouvons donc aborder ces deux problèmes orthogonaux en restreignant les points où les communications peuvent avoir lieu, et ainsi restreindre les points du programme où le sous-typage est appliqué. Nous montrons dans cette section comment modifier le système de types afin de mettre en œuvre cette restriction.

Nous restreignons ici les valeurs communiquées aux variables définies par les équations ($x = e$). Ainsi, pour le programme de la section 5.2.1, seules les valeurs définissant `y` et `z` peuvent être communiquées d'un site à l'autre.

Le système de types modifié, noté $H \vdash_i e : t/\ell$, consiste en la suppression, du système de types initial, de la règle de sous-typage Comm, et en la modification

de la règle Def. Cette nouvelle règle contraint les valeurs portées par les variables à être centralisées, et les localisations de la valeur émise et de la valeur définie sont soit égales, soit contraintes par l'existence d'un lien de communication dans l'architecture. Nous notons cette dernière contrainte $s \triangleright s'$:

$$s \triangleright s' \Leftrightarrow s = s' \vee (s, s') \in \mathcal{L}$$

La règle Def modifiée est donnée ci-dessous :

$$\text{(Def)}\quad \frac{H|G \vdash_i e : tc_1 \texttt{ at } s_1 \times \ldots \times tc_n \texttt{ at } s_n/\ell \qquad \forall i \in \{1, \ldots, n\}, s_i \triangleright s'_i}{H|G \vdash_i (x_1, \ldots, x_n) = e : [tc_1 \texttt{ at } s'_1/x_1, \ldots, tc_n \texttt{ at } s'_n]/\ell}$$

Typer une équation $x = e$ consiste à contraindre l'expression e à être d'un type centralisé tc sur un site s. L'environnement de typage produit par l'équation associe à x le type tc at s', tel que soit $s = s'$, soit il existe un lien de communication de s à s' dans l'architecture.

L'implémentation repose ensuite sur l'utilisation d'environnements de typage localisés, notés H at A. Toutes les instances des types issus d'un environnement localisé sur A représentent des valeurs entièrement localisées sur A. Ces environnements seront utilisés pour localiser une expression e sur un site A.

$$H ::= H \texttt{ at } A \mid [x_1 : \sigma_1, \ldots, x_n : \sigma_n]$$

La relation d'instanciation est définie à partir d'un environnement localisé sur s de manière à ce que tout type instancié à partir d'un tel environnement représente une valeur entièrement localisée sur le site s :

$$tc \texttt{ at } s \leq (H \texttt{ at } s)(x) \Leftrightarrow tc \texttt{ at } s \leq H(x)$$

Les règles Imm et At du système de types deviennent alors :

$$\text{(Imm)}\quad \frac{\forall s' \in \mathcal{S}, H \neq H' \texttt{ at } s' \qquad s \in \mathcal{S}}{H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash_i i : b \texttt{ at } s/\{s\}}$$

$$\text{(Imm-Loc)}\quad H \texttt{ at } A|G \vdash_i i : b \texttt{ at } A/\{A\}$$

$$\text{(At)}\quad \frac{H \texttt{ at } A|\langle \mathcal{S}, \mathcal{L} \rangle \vdash_i e : t/\ell \qquad A \in \mathcal{S}}{H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash_i e \texttt{ at } A : t/\ell}$$

Les autres règles du système de types sont identiques au système de types initial.

Le résultat suivant est la correction de ce système de types modifié, c'est-à-dire, que tout programme accepté par le système de types modifié est aussi accepté par le système de types initial.

Ce résultat repose sur le lemme 5, qui établit que l'ensemble des sites nécessaires à l'évaluation d'une expression typée dans un environnement de typage localisé sur le site A est le singleton $\{A\}$.

Lemme 5. *Pour tous H, G, A, D, H', ℓ, si H `at` $A|G \vdash D : H'/\ell$, alors $\ell = \{A\}$.*

Démonstration. Par la définition de la relation d'instanciation, pour tous H, A, x, t tels que $t \leq (H \texttt{ at } A)(x)$, il existe t', tel que $t = t' \texttt{ at } A$. Par conséquent, $\text{locations}(t) = \{A\}$.

Le lemme 5 est ensuite prouvé par induction sur la structure de e. Les autres règles de typage ne calculent l'effet d'une expression qu'à partir de l'union des sites des effets des composants de l'expression e : si elle est typée dans un environnement de typage localisé de la forme $H \texttt{ at } A$, alors la propriété est vérifiée. □

Théorème 2 (Correction du système de types modifié)**.** *Pour tous H, H', D, ℓ, si $H|G \vdash_i D : H'/\ell$, alors $H|G \vdash D : H'/\ell$.*

Démonstration. Par induction sur la structure de D et e. À chaque application de la règle Def du système de types modifié, on peut appliquer la règle de sous-typage Comm puis la règle Def du système de types initial.

La règle At est applicable du fait du lemme 5. □

5.8 Discussion

Le système de types modifié a été implémenté dans le compilateur Lucid Synchrone. L'algorithme de typage consiste en la génération, à partir du typage de chaque équation $(x_i = e_i)_{i \in \{1,\ldots,n\}}$, d'un ensemble de contraintes entre variables de sites $C = \{s_1 \triangleright s'_1, \ldots, s_n \triangleright s'_n\}$, accompagné des contraintes d'égalité usuelles de l'inférence de types. Ces contraintes sont ensuite résolues en affectant à chaque s_i et s'_i des valeurs de sites concrets de l'architecture, telles que les contraintes soient vérifiées. L'algorithme de résolution des contraintes est donc ici indépendant de la méthode de typage, et peut être remplacé par n'importe quel algorithme de la littérature (par exemple, l'algorithme AAA de SynDex [55]). Il est aussi possible de considérer un langage de contraintes plus général, permettant par exemple l'expression d'exclusivité de sites pour des fonctionnalités de tolérance aux fautes. L'apport de cette méthode par typage vis-à-vis de ces algorithmes est un gain de modularité, au prix cependant d'une perte de généralité, l'algorithme de résolution de contraintes devant être évalué localement. Cette dernière contrainte est due à l'opération de projection ; nous verrons au chapitre 8 comment elle peut être relâchée, si cette opération ne s'applique pas.

Chapitre 6

Répartition

Ce chapitre présente, sur la base du système de types spatiaux présenté au chapitre précédent, une opération de projection guidée par les types. Cette opération permet d'obtenir, à partir d'un programme centralisé dont les valeurs sont annotées par leur type spatial, un nouveau programme réparti constitué d'un fragment par site de l'architecture. Chaque fragment de ce programme réparti est une projection du programme initial, au sens où ce fragment est un programme comprenant uniquement les calculs à effectuer sur un site donné. Cette opération de projection ajoute les communications entre sites sous forme de variables représentant le flot de données de chacune de ces communications. Enfin, cette opération est modulaire : le résultat de la projection d'un nœud est un nœud par site de l'architecture.

6.1 Principe

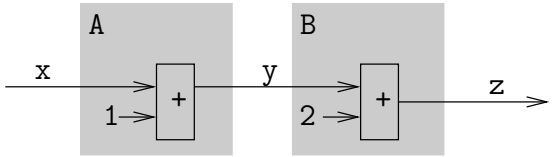
6.1.1 Projection d'ensembles d'équations

Une fois l'analyse de type spatial effectuée sur un programme, les expressions et équations de ce programme sont annotées avec leur effet, c'est-à-dire un ensemble de sites nécessaires à leur évaluation. La projection d'un ensemble d'équations sur un site A consiste en :

1. la suppression des équations et expressions non évaluées sur A,
2. l'ajout de la définition de flots portant les valeurs communiquées depuis A vers d'autres sites de l'architecture,
3. le remplacement des expressions évaluées sur d'autres sites, et dont la valeur est communiquée vers A, par la variable définie comme portant cette valeur communiquée sur ces sites.

Considérons par exemple la projection des deux équations de la section 5.2.1, en supposant que le typage a placé l'évaluation de `x + 1` sur le site A :

```
    y = x + 1 at A
and z = y + 2 at B
```



La communication entre `A` et `B` est représentée par l'ajout d'une variable `c`, portant la valeur du résultat de l'évaluation de l'expression `x + 1`. D'une part, la définition de cette variable est placée sur le site `A`. D'autre part, cette variable remplace l'expression `x + 1` sur le site `B`. Le résultat de la projection est donné ci-dessous :

A	B
$c = x_A + 1$	$y_B = c$ and $z_B = y_B + 2$

Les trois variables d'origines `x`, `y` et `z` sont indicées par le site sur lequel elles sont projetées.

La projection de ces équations sur `A` consiste en la suppression de l'équation définissant `z` (qui est entièrement exécutée sur `B`), et la définition de la nouvelle variable `c`, définissant un canal de communication de `A` à `B`, et portant la valeur du résultat de l'expression `x + 1`.

La projection sur `B` consiste en la suppression de l'expression `x + 1` calculée sur `A`. Cette expression est remplacée par la variable `c`, définie sur `A`.

Le produit synchrone des deux programmes obtenus par projection (noté ci-dessus $\|$) est un programme dont la sémantique est équivalente à celle du programme initial.

Le fait d'indicer les variables par le site sur lequel elles sont projetées permet de gérer correctement le cas où ces variables sont définies avec une valeur répartie. Par exemple, la projection sur `A` et `B` de l'équation :

```
y = (1 at A, 2 at B)
```

a pour résultat :

A	B
$y_A = (1, \perp)$	$y_B = (\perp, 2)$

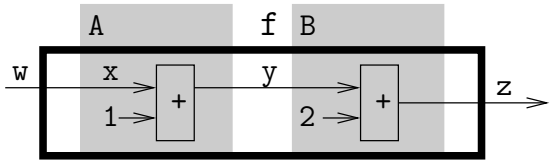
Ici, la variable `y` est définie avec la paire répartie $(1, 2)$, dont la première composante est sur le site `A` et la deuxième sur le site `B`. On note $\perp$ l'expression spéciale « absente » : cette valeur ne sera utilisée dans aucun calcul, et est du type spécial « `abs` » comportant la seule valeur $\perp$:

```
type abs = ⊥
```

6.1.2 Projection de nœuds

La projection d'un nœud consiste en la projection des équations et expressions constituant son corps. Les variables portant les valeurs communiquées dans ce corps sont ensuite ajoutées en entrées et en sorties du nœud projeté. Considérons ainsi la projection sur `A` et `B` du nœud suivant :

```
node f(w) = z where
      x = w at A
  and y = x + 1
  and z = y + 2 at B
```



La projection de ce nœud sur `A` et `B` a pour résultat un unique nœud par site :

A	B
node $f_A(w_A)$ = $(\perp, c)$ where x_A = w_A and c = x_A + 1	node $f_B(w_B, c)$ = z_B where y_B = c and z_B = y_B + 2

La projection de `f` sur le site `A` consiste en la projection du corps de `f` comme précédemment, et de la modification de la sortie :

- en supprimant la sortie `z`, remplacée par la valeur spéciale $\perp$ dénotant la suppression de l'expression,
- en ajoutant une deuxième sortie, prenant la valeur du canal de communication `c`.

La projection de l'instanciation de cette fonction permet d'ajouter la définition d'un canal de communication `c1`, défini comme sortie du programme sur `A`, et comme entrée sur `B`. Le programme suivant :

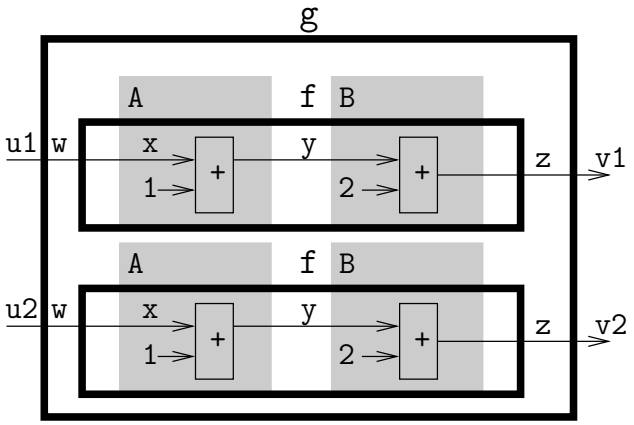
```
v = f(u)
```

est ainsi projeté en :

A	B
v_A,c1 = f_A(u_A)	v_B = f_B(u_B,c1)

Un nouveau canal de communication doit être créé pour *chaque* instanciation du nœud `f` : ainsi, si le programme contient deux instanciations, deux canaux seront créés, représentés par deux variables différentes ajoutées en entrée et en sortie des programmes projetés. Par exemple :

```
node g(u1,u2) = (v1,v2) where
      v1 = f(u1)
  and v2 = f(u2)
```



A	B
node g_A($u1_A$,$u2_A$) = (⊥,⊥,c1,c2) where $v1_A$,c1 = f_A($u1_A$) and $v2_A$,c2 = f_A($u2_A$)	node g_B($u1_B$,$u2_B$,c1,c2) = ($v1_B$,$v2_B$) where $v1_B$ = f_B($u1_B$,c1) and $v2_B$ = f_B($u2_B$,c2)

6.2 Inférence des canaux de communication

La section précédente permet de se rendre compte que, pour la répartition par projection de l'instanciation d'un nœud `f`, plusieurs informations sont nécessaires :

1. l'ensemble des sites impliqués dans le calcul de `f`, afin de supprimer cette instanciation des sites où elle n'est pas nécessaire ;

2. l'ensemble des communications internes à `f`, afin d'ajouter à l'instanciation de `f` les entrées et les sorties nécessaires.

Afin d'inférer ces informations, nous allons étendre le système de types à effets présenté au chapitre 5 : les effets inférés des équations et des nœuds seront une paire ℓ/T :

- ℓ est comme précédemment l'ensemble des sites nécessaires à l'évaluation de l'expression ou de l'équation ;
- T est une séquence de *canaux de communication* utilisés pour cette évaluation.

Il est nécessaire de souligner la différence faite ici entre les *liens* et les *canaux* de communication. Les liens de communications sont déclarés par le programmeur par `link` A `to` B, et définissent une relation entre les sites de l'architecture. Cette relation signifie la *possibilité* de communications entre deux sites, et est utilisée comme *contrainte* lors du typage. Les canaux de communication sont inférés par le système de types étendu, et représentent les communications réellement utilisées par le programme réparti obtenu par projection.

Un canal est une paire de sites, à laquelle un nom c est associé. D'une part ce nom sert à différencier les canaux, et d'autre part il correspond au nom de la variable ajoutée lors de la projection. On suppose pour la suite que les noms de canaux de communication appartiennent à un espace de nom différents des variables x utilisées dans le langage. Les noms de canaux sont ordonnés par un ordre total $<$.

Un canal de communication est noté $A_s \overset{c}{\mapsto} A_d$, où c est le nom du canal, A_s son site source, et A_d son site de destination. Les séquences de canaux sont notées T, la concaténation de deux séquences T_1 et T_2 est notée T_1, T_2. Un nom de canal ne peut apparaître qu'une fois dans une séquence de canaux. La syntaxe des séquences de canaux est donc :

$$\begin{array}{rl} T ::= & [A_1 \overset{c_1}{\mapsto} A'_1, \ldots, A_n \overset{c_n}{\mapsto} A'_n] \\ & \text{si } \forall i \neq j, c_i \neq c_j \end{array}$$

À titre d'exemple, reprenons le nœud `f` de la section précédente :

```
node f(w) = z where
      x = w at A
  and y = x + 1
  and z = y + 2 at B
```

L'effet de l'ensemble des trois équations composant le corps de ce nœud est la paire $\{A, B\}/[A \overset{c}{\mapsto} B]$, ce qui signifie que l'évaluation de ces équations nécessite les sites A et B et un canal de communication de A à B. Le type spatial de `f` est alors :

$$b \texttt{ at } A \prec\!\langle \{A, B\}/[A \overset{c}{\mapsto} B] \rangle\!\succ b \texttt{ at } B.$$

L'information donnée par ce type permet, à la projection d'une instanciation de `f`, d'ajouter à cette instanciation le bon nombre d'entrées et de sorties, et de garder la projection de l'instanciation cohérente avec la projection du nœud lui-même.

On note $\mathrm{dom}(T)$ l'ensemble des noms de canaux de T. Il est nécessaire de pouvoir renommer les canaux d'une séquence, afin de permettre l'instanciation multiple des nœuds projetés. Ce renommage doit conserver l'ordre des canaux dans la séquence renommée. $T' \cong T$ signifie que les séquences de canaux T et T' sont identiques à un renommage près. La relation $\cong$ est définie par :

$$T' \cong T \Leftrightarrow T' = T[c'_1/c_1, \ldots, c'_n/c_n] \qquad \begin{array}{l} \text{où } \mathrm{dom}(T) = \{c_1, \ldots, c_n\} \\ \text{et } \mathrm{dom}(T') = \{c'_1, \ldots, c'_n\} \end{array}$$

Remarque 4. *L'exemple suivant montre l'importance de l'ordre des canaux pour la projection des instanciations de nœuds comportant plusieurs communications.*

```
node f(x,y) = (z,t) where
      z1 = f1(x) at A
  and z = f2(z1) at B
  and t1 = g1(y) at A
  and t = g2(t1) at B
```

Le type spatial de `f` *sera ici :*

$$\mathtt{f} : (b\ \mathbf{at}\ A \times b\ \mathbf{at}\ A) \prec\!\{A, B\}/[A \overset{c_1}{\mapsto} B, A \overset{c_2}{\mapsto} B]\!\succ\!\rightarrow (b\ \mathbf{at}\ B \times b\ \mathbf{at}\ B)$$

La figure 6.1 représente ce nœud sous la forme de schémas-blocs. Ce nœud prend deux entrées `x` *et* `y`*, et renvoie une paire composée des valeurs* `f2(f1(x))` *et* `g2(g1(y))`*.* `f1` *et* `g1` *sont exécutées sur le site A, et* `f2` *et* `g2` *sur B. Deux canaux de communications* `c1` *et* `c2` *sont donc nécessaires pour l'exécution de ce nœud.*

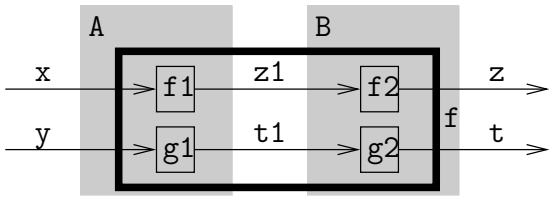


FIG. 6.1 – Nœud comprenant deux canaux de communications.

La projection de ce nœud est donnée ci-dessous :

A	*B*
node $f_A(x_A,y_A)$ = $(\perp,\perp,c1,c2)$ where	node $f_B(x_B,y_B,c1,c2)$ = (z_B,t_B) where
$c1 = f1_A(x_A)$ and $c2 = g1_A(y_A)$	$z_B = f2_B(c1)$ and $t_B = g2_B(c2)$

Considérons maintenant la projection d'une instanciation de `f` *:*

```
(z1,t1) = f(x1,y1)
```

En supposant que les canaux `c1` *et* `c2` *de* `f`*, à son instanciation, ont été renommés respectivement en* `c3` *et* `c4`*, la projection de cette instanciation est alors :*

A	*B*
$(z1_A,t1_A,c3,c4) = f_A(x1_A,y1_A)$	$(z1_B,t1_B) = f_B(x1_B,y1_B,c3,c4)$

Cette projection est correcte : elle associe, comme voulu, la valeur calculée de `c1` *à la variable* `c3` *sur A, puis la valeur* `c3` *reçue à la variable* `c1` *en paramètre de* `f` *sur B. La projection intervertissant* `c3` *et* `c4` *sur les deux sites est de la même manière correcte.*

On peut souligner ici que le fait d'utiliser une méthode de répartition automatique à partir d'un unique programme permet de se prémunir d'incohérences possible dans le cas où la répartition est effectuée de manière manuelle. Ainsi, une analyse causale globale permet de définir un ordre cohérent de communications entre deux sites. Pour la projection des nœuds et de leurs instanciations, cette cohérence est conservée par l'ordre des valeurs communiquées, ajoutées en entrées et en sorties. Par exemple, intervertir `c3` *et* `c4` *sur un seul des sites modifie le sens du programme réparti :*

A	*B*
$(z1_A,t1_A,c3,c4) = f_A(x1_A,y1_A)$	$(z1_B,t1_B) = f_B(x1_B,y1_B,c4,c3)$

Sur le site B, la valeur `c3` *reçue est ici affectée au paramètre* `c2` *du nœud* f_B*. Les deux valeurs communiquées sont interverties, ce qui mène le programme réparti à calculer la paire composée des valeurs* `f2(g1(y))` *et* `g2(f1(x))` *: le sens du programme réparti n'est alors pas le même que celui du programme initial.*

Pour le système de types étendu, la syntaxe des types est modifiée en ajoutant

les séquences de canaux aux effets des types de nœuds :

$$\begin{array}{l} \vdots \\ t ::= t \prec\!\langle \ell/T \rangle\!\succ t \mid t \times t \mid tc \ \texttt{at} \ s \\ \vdots \end{array}$$

Les égalités de types sont modifiées : le type des nœuds centralisés ne comporte aucun canal de communication.

$$(tc_1 \rightarrow tc_2) \ \texttt{at} \ s = (tc_1 \ \texttt{at} \ s) \prec\!\langle \{s\}/\emptyset \rangle\!\succ (tc_2 \ \texttt{at} \ s)$$

De même, l'environnement de typage initial H_0, contenant le type des opérations, est défini en tenant compte du caractère centralisé des opérations :

$$H_0 = \left[\begin{array}{rl} \cdot \ \texttt{fby} \ \cdot : & \forall\alpha.\forall\delta.\alpha \ \texttt{at} \ \delta \times \alpha \ \texttt{at} \ \delta \prec\!\langle \{\delta\}/\emptyset \rangle\!\succ \alpha \ \texttt{at} \ \delta, \\ \texttt{if} \ \cdot \ \texttt{then} \ \cdot \ \texttt{else} \ \cdot : & \forall\alpha.\forall\delta.(b \ \texttt{at} \ \delta \times \alpha \ \texttt{at} \ \delta \times \alpha \ \texttt{at} \ \delta) \prec\!\langle \{\delta\}/\emptyset \rangle\!\succ \delta \ \texttt{at} \ \delta, \\ (+) : & \forall\delta.b \ \texttt{at} \ \delta \times b \ \texttt{at} \ \delta \prec\!\langle \{\delta\}/\emptyset \rangle\!\succ b \ \texttt{at} \ \delta, \\ & \cdots \end{array} \right]$$

Le système de types étendu est défini par les prédicats :

$$H|G \vdash e : t/\ell/T \qquad\qquad H|G \vdash D : H'/\ell/T$$

- $H|G \vdash e : t/\ell/T$ signifie que dans l'environnement de typage H, et l'architecture G, l'expression e est de type t, et son évaluation implique l'ensemble de sites ℓ, et les canaux de communications T.
- $H|G \vdash D : H'/\ell/T$ signifie que dans l'environnement de typage H, et l'architecture G, l'équation D produit l'environnement de typage H', et son évaluation implique l'ensemble de sites ℓ, et les canaux de communications T.

Ces prédicats sont définis par les règles de la figure 6.2. La règle de sous-typage Comm est modifiée en la règle Comm-C, qui ajoute un canal à chaque point de communication du programme. La règle App-C permet de renommer, lors de l'instanciation d'un nœud, les canaux utilisés dans le nœud appliqué. Les autres règles étendent les règles du système de types initial, en définissant la séquence de canaux comme étant la concaténation des séquences de canaux des composants de l'expression ou de l'équation typée.

La figure 6.3 montre l'application du système étendu aux équations de la figure 5.3.

(IMM-C)
$$H|G \vdash i : b \texttt{ at } s/\{s\}/\emptyset$$

(INST-C)
$$\frac{t \leq H(x)}{H|G \vdash x : t/\operatorname{locations}(t)/\emptyset}$$

(PAIR-C)
$$\frac{H|G \vdash e_1 : t_1/\ell_1/T_1 \qquad H|G \vdash e_2 : t_2/\ell_2/T_2}{H|G \vdash (e_1, e_2) : t_1 \times t_2/\ell_1 \cup \ell_2/T_1, T_2}$$

(OP-C)
$$\frac{H|G \vdash \mathrm{op} : t_1 \prec\!\{s\}/\emptyset\!\succ\!\rightarrow t_2/\{s\}/\emptyset \qquad H|G \vdash e : t_1/\ell/T}{H|G \vdash \mathrm{op}(e) : t_2/\{s\} \cup \ell/T}$$

(AT-C)
$$\frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : t/\{A\}/T \qquad A \in \mathcal{S}}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e \texttt{ at } A : t/\{A\}/T}$$

(COMM-C)
$$\frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc \texttt{ at } s/\ell/T \qquad (s, s') \in \mathcal{L}}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc \texttt{ at } s'/\ell \cup \{s'\}/T, [s \overset{c}{\mapsto} s']}$$

(NODE-C)
$$\frac{H, x : t_1, H_1|G \vdash D : H_1/\ell_1/T_1 \qquad H, x : t_1, H_1|G \vdash e : t_2/\ell_2/T_2}{\begin{array}{c} H|G \vdash \texttt{node } f(x) = e \texttt{ where } D : \\ [\mathrm{gen}_H(t_1 \prec\!\ell_1 \cup \ell_2/T_1, T_2\!\succ\!\rightarrow t_2)/f]/\ell_1 \cup \ell_2/\emptyset \end{array}}$$

(DEF-C)
$$\frac{H|G \vdash e : t_1 \times \ldots \times t_n/\ell/T}{H|G \vdash (x_1, \ldots, x_n) = e : [t_1/x_1, \ldots, t_n/x_n]/\ell/T}$$

(APP-C)
$$\frac{H|G \vdash f : t_1 \prec\!\ell_1/T_1\!\succ\!\rightarrow t_2/\ell_2/T_2 \qquad H|G \vdash e : t_1/\ell_3/T_3 \qquad T_1' \cong T_1}{H|G \vdash x = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3/T_1', T_2, T_3}$$

(AND-C)
$$\frac{H|G \vdash D_1 : H_1/\ell_1/T_1 \qquad H|G \vdash D_2 : H_2/\ell_2/T_2}{H|G \vdash D_1 \texttt{ and } D_2 : H_1, H_2/\ell_1 \cup \ell_2/T_1, T_2}$$

(LET-C)
$$\frac{H|G \vdash D_1 : H_1/\ell_1/T_1 \qquad H, H_1|G \vdash D_2 : H_2/\ell_2/T_2}{H|G \vdash \texttt{let } D_1 \texttt{ in } D_2 : H_2/\ell_1 \cup \ell_2/T_1, T_2}$$

FIG. 6.2 – Système de types étendu avec canaux de communications.

$$\cfrac{\cfrac{\cfrac{\cfrac{\begin{array}{c}b\ \mathtt{at}\ A \times b\ \mathtt{at}\ A \prec\!\langle\{A\}/\emptyset\rangle\!\rightarrowtail b\ \mathtt{at}\ A\\ \leq \forall\delta.b\ \mathtt{at}\ \delta \times b\ \mathtt{at}\ \delta \prec\!\langle\{\delta\}/\emptyset\rangle\!\rightarrowtail b\ \mathtt{at}\ \delta\end{array}}{H_1|G \vdash (+) : b\ \mathtt{at}\ A \times b\ \mathtt{at}\ A \prec\!\langle\{A\}/\emptyset\rangle\!\rightarrowtail b\ \mathtt{at}\ A/\{A\}/\emptyset}\ (\textsc{Inst-C}) \quad \vdots \quad H_1|G \vdash x : b\ \mathtt{at}\ A/\{A\}/\emptyset \qquad H_1|G \vdash 1 : b\ \mathtt{at}\ A/\{A\}/\emptyset}{H_1|G \vdash x+1 : b\ \mathtt{at}\ A/\{A\}/\emptyset}\ (\textsc{Op-C})}{H_1|G \vdash x+1 : b\ \mathtt{at}\ B/\{A,B\}/[A \overset{c}{\mapsto} B]}\ (\textsc{Comm-C})}{H_1|G \vdash y = x+1 : [y : b\ \mathtt{at}\ B]/\{A,B\}/[A \overset{c}{\mapsto} B]}\ (\textsc{Def-C})$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\begin{array}{c}b\ \mathtt{at}\ B \times b\ \mathtt{at}\ B \prec\!\langle\{B\}/\emptyset\rangle\!\rightarrowtail b\ \mathtt{at}\ B\\ \leq \forall\delta.b\ \mathtt{at}\ \delta \times b\ \mathtt{at}\ \delta \prec\!\langle\{\delta\}/\emptyset\rangle\!\rightarrowtail b\ \mathtt{at}\ \delta\end{array}}{H_1|G \vdash (+) : b\ \mathtt{at}\ B \times b\ \mathtt{at}\ B \prec\!\langle\{B\}/\emptyset\rangle\!\rightarrowtail b\ \mathtt{at}\ B/\{B\}/\emptyset}\ (\textsc{Inst-C}) \quad \vdots \quad H_1|G \vdash y : b\ \mathtt{at}\ B/\{B\}/\emptyset \qquad H_1|G \vdash 2 : b\ \mathtt{at}\ B/\{B\}/\emptyset}{H_1|G \vdash y+2 : b\ \mathtt{at}\ B/\{B\}/\emptyset}\ (\textsc{Op-C})}{H_1|G \vdash y+2\ \mathtt{at}\ B : b\ \mathtt{at}\ B/\{B\}/\emptyset}\ (\textsc{At-C})}{H_1|G \vdash z = y+2\ \mathtt{at}\ B : [z : b\ \mathtt{at}\ B]/\{B\}/\emptyset}\ (\textsc{Def-C})$$

$$\cfrac{\vdots \qquad \vdots}{H_1|G \vdash \begin{array}{rl} & y = x+1\\ \mathtt{and} & z = y+2\ \mathtt{at}\ B\end{array} : \left[\begin{array}{c} y : b\ \mathtt{at}\ B,\\ z : b\ \mathtt{at}\ B\end{array}\right]/\{A,B\}/[A \overset{c}{\mapsto} B]}\ (\textsc{And-C})$$

Avec $H_1 = H_0, [x : b\ \mathtt{at}\ A, y : b\ \mathtt{at}\ B, z : b\ \mathtt{at}\ B]$ et $G = \langle\{A,B\}, \{(A,B)\}\rangle$.

FIG. 6.3 – Exemple d'application du système de types étendu.

6.3 Opération de projection

Nous définissons maintenant, à partir de ce système de types étendu, une opération de *projection sur un site* A, définie sur les expressions et les équations. Cette projection calcule, à partir d'une expression ou d'une équation, l'expression ou l'équation à évaluer sur le site A, telle que le produit synchrone des équations résultant de la projection sur chacun des sites de l'architecture ait la même sémantique que l'équation initiale projetée.

Cette opération est définie séparément, et effectuée sur des programmes entièrement annotés par leur types spatiaux et leur canaux de communications utilisés.

On note ϵ l'équation vide et $\perp$ une expression supprimée sur le site sur lequel s'effectue la projection. Pour toute équation D, on a $(D \texttt{ and } \epsilon) = (\epsilon \texttt{ and } D) = D$. D'autre part, on a $(\perp, \perp) = \perp$.

À partir d'une séquence de canaux T, on note $T \uparrow A$ la séquence de canaux extraite de T composée des canaux de source A. De même, $T \downarrow A$ est la séquence extraite de T composée des canaux de destination A.

$$\begin{cases} \emptyset \uparrow A = \emptyset \\ ([A_1 \overset{c}{\mapsto} A_2], T) \uparrow A = \begin{cases} [A_1 \overset{c}{\mapsto} A_2], (T \uparrow A) & \text{si } A_1 = A \\ (T \uparrow A) & \text{sinon} \end{cases} \end{cases}$$

$$\begin{cases} \emptyset \downarrow A = \emptyset \\ ([A_1 \overset{c}{\mapsto} A_2], T) \downarrow A = \begin{cases} [A_1 \overset{c}{\mapsto} A_2], (T \downarrow A) & \text{si } A_2 = A \\ (T \downarrow A) & \text{sinon} \end{cases} \end{cases}$$

La projection d'une équation ou d'un ensemble d'équations D sur le site A est définie par le prédicat $H|G \vdash D : H'/\ell/T \overset{A}{\Longrightarrow} D'$. Le résultat de la projection est une nouvelle équation ou ensemble d'équations D'.

L'opération de projection d'une expression e sur le site A est définie par le prédicat $H|G \vdash e : t/\ell/T \overset{A}{\Longrightarrow} e'/D$. Le résultat de la projection est une paire e'/D, où e' est l'expression remplaçant e sur le site A, et D un ensemble d'équations définissant les canaux de communications partant de A et utilisés par l'évaluation de e.

Les règles définissant ces deux prédicats sont données en figures 6.4, 6.5, 6.6 et 6.7.

Projection des expressions (figures 6.4 et 6.5)

La projection d'une valeur immédiate sur un site A est cette même valeur, si le type de cette valeur la définit comme étant localisée sur A (règle Imm-P).

Sinon, cette valeur immédiate est supprimée et remplacée par la valeur spéciale $\perp$ (règle Imm-P-Suppr).

La projection de l'instanciation d'une variable x sur un site A consiste en le remplacement de cette variable par la variable annotée x_A, si le type de cette instanciation contient le site A (règle Inst-P). Sinon, cette instanciation est supprimée (règle Inst-P-Suppr).

La projection d'une paire consiste en la projection de ses composantes. Les équations résultant de ces projections sont composées en parallèle (règle Pair-P).

La projection d'une opération sur A consiste en la projection de ses composantes si le type de cette opération spécifie qu'elle est effectuée sur le site A (règle Op-P). Sinon, cette opération est supprimée (règle Op-P-Suppr). Dans les deux cas, les équations résultant de la projection des composantes de l'opération sont composées en parallèle.

Le résultat de la projection d'une expression annotée par le programmeur est la projection de sa composante (règle At-P).

Les canaux de communications sont utilisés, à la projection, aux points du programme déterminés par l'application de la règle de sous-typage Comm-C. Cette règle infère l'utilisation d'un canal de communication $A \overset{c}{\mapsto} A'$. Trois règles de projection sont définies pour les points de communication :

- Concernant la projection sur A, source de la communication, le résultat de la projection est l'expression spéciale $\perp$. Le résultat e' de la projection avant la communication permet l'ajout d'une équation définissant le canal c associé à cette communication (règle Comm-P-From).
- Concernant la projection sur le site A', destination de la communication, l'expression est remplacée par le nom du canal (règle Comm-P-To).
- Sur tous les autres sites, la projection supprime l'expression (règle Comm-P-Suppr).

(Imm-P)
$$H|G \vdash i : b \texttt{ at } A/\{A\}/\emptyset \overset{A}{\Longrightarrow} i/\epsilon$$

(Imm-P-Suppr)
$$\frac{A \neq A'}{H|G \vdash i : b \texttt{ at } A'/\{A'\}/\emptyset \overset{A}{\Longrightarrow} \perp/\epsilon}$$

(Inst-P)
$$\frac{t \leq H(x) \qquad A \in \text{locations}(t)}{H|G \vdash x : t/\text{locations}(t)/\emptyset \overset{A}{\Longrightarrow} x_A/\epsilon}$$

(Inst-P-Suppr)
$$\frac{t \leq H(x) \qquad A \notin \text{locations}(t)}{H|G \vdash x : t/\text{locations}(t)/\emptyset \overset{A}{\Longrightarrow} \perp/\epsilon}$$

Fig. 6.4 – Définition de l'opération de projection (règles pour les expressions, 1/2).

(Pair-P)

$$\frac{H|G \vdash e_1 : t_1/\ell_1/T_1 \stackrel{A}{\Longrightarrow} e_1'/D_1 \qquad H|G \vdash e_2 : t_2/\ell_2/T_2 \stackrel{A}{\Longrightarrow} e_2'/D_2}{H|G \vdash e_1, e_2 : t_1 \times t_2/\ell_1 \cup \ell_2/T_1, T_2 \stackrel{A}{\Longrightarrow} e_1', e_2'/D_1 \texttt{ and } D_2}$$

(Op-P)

$$\frac{H|G \vdash \mathrm{op} : t_1 \prec\!\{A\}/\emptyset\!\succ\!\rightarrow t_2/\{A\}/\emptyset \stackrel{A}{\Longrightarrow} f'/D_1 \qquad H|G \vdash e : t_1/\ell/T \stackrel{A}{\Longrightarrow} e'/D_2}{H|G \vdash \mathrm{op}(e) : t_2/\{s\} \cup \ell/T \stackrel{A}{\Longrightarrow} f'(e')/D_1 \texttt{ and } D_2}$$

(Op-P-Suppr)

$$\frac{\begin{array}{c} H|G \vdash \mathrm{op} : t_1 \prec\!\{s\}/\emptyset\!\succ\!\rightarrow t_2/\{s\}/\emptyset \stackrel{A}{\Longrightarrow} \bot/D_1 \\ H|G \vdash e : t_1/\ell/T \stackrel{A}{\Longrightarrow} \bot/D_2 \qquad s \neq A \end{array}}{H|G \vdash \mathrm{op}(e) : t_2/\{s\} \cup \ell/T \stackrel{A}{\Longrightarrow} \bot/D_1 \texttt{ and } D_2}$$

(At-P)

$$\frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : t/\{A'\}/T \stackrel{A}{\Longrightarrow} e'/D \qquad A' \in \mathcal{S}}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e \texttt{ at } A' : t/\{A'\}/T \stackrel{A}{\Longrightarrow} e'/D}$$

(Comm-P-From)

$$\frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc \texttt{ at } A/\ell/T \stackrel{A}{\Longrightarrow} e'/D \qquad (A, s') \in \mathcal{L}}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc \texttt{ at } s'/\ell \cup \{s'\}/T, [A \stackrel{c}{\mapsto} s'] \stackrel{A}{\Longrightarrow} \bot/D \texttt{ and } c = e'}$$

(Comm-P-To)

$$\frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc \texttt{ at } s/\ell/T \stackrel{A'}{\Longrightarrow} \bot/D \qquad (s, A') \in \mathcal{L}}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc \texttt{ at } A'/\ell \cup \{A'\}/T, [s \stackrel{c}{\mapsto} A'] \stackrel{A'}{\Longrightarrow} c/D}$$

(Comm-P-Suppr)

$$\frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc \texttt{ at } s/\ell/T \stackrel{A}{\Longrightarrow} \bot/D \qquad (s, s') \in \mathcal{L} \qquad A \notin \{s, s'\}}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : tc \texttt{ at } s'/\ell \cup \{s'\}/T, [s \stackrel{c}{\mapsto} s'] \stackrel{A}{\Longrightarrow} \bot/D}$$

Fig. 6.5 – Définition de l'opération de projection (règles pour les expressions, 2/2).

Projection des nœuds (figure 6.6)

La règle Node-P définit la projection d'un nœud sur le site A participant à son évaluation ($A \in \ell_1 \cup \ell_2$, ℓ_1 et ℓ_2 étant les effets respectifs de l'expression et de l'équation définissant le nœud). Le résultat de cette projection est un nœud dont le nom et les paramètres sont annotés par A. L'ensemble d'équations D_2, issu de la projection de e, est composé en parallèle avec le résultat de la projection de D. Les canaux de communications utilisés au sein du nœud projeté et dont la source est A sont ajoutés en sortie du nœud. Ceux dont la destination est A sont ajoutés en entrée.

La règle Node-P-Suppr définit la projection d'un nœud sur un site ne participant pas à son évaluation. Ce nœud est alors supprimé.

La règle Node-P-Gen définit la projection des nœuds présents sur tous les sites, c'est-à-dire dont le schéma de types inféré est de la forme $\forall \alpha_1, \ldots, \alpha_n . \forall \delta . tc$ `at` δ. Le contenu de ce nœud est dupliqué sur tous les sites.

(Node-P)

$$\frac{\begin{array}{c} H, x : t_1, H_1 | G \vdash D : H_1/\ell_1/T_1 \stackrel{A}{\Longrightarrow} D_1 \qquad H, x : t_1, H_1 | G \vdash e : t_2/\ell_2/T_2 \stackrel{A}{\Longrightarrow} e'/D_2 \\ (T_1, T_2) \uparrow A = [A \stackrel{c_1}{\mapsto} A_1, \ldots, A \stackrel{c_n}{\mapsto} A_n] \qquad (T_1, T_2) \downarrow A = [A'_1 \stackrel{c'_1}{\mapsto} A, \ldots, A'_p \stackrel{c'_p}{\mapsto} A] \\ A \in \ell_1 \cup \ell_2 \qquad \exists t. \operatorname{gen}_H(t_1 \prec\!\langle \ell_1 \cup \ell_2/T_1, T_2 \rangle\!\rightarrow t_2) = \forall \alpha_1, \ldots, \alpha_n . t \end{array}}{\begin{array}{c} H | G \vdash \texttt{node}\ f(x) = e\ \texttt{where}\ D : \\ [\operatorname{gen}_H(t_1 \prec\!\langle \ell_1 \cup \ell_2/T_1, T_2 \rangle\!\rightarrow t_2)/f]/\ell_1 \cup \ell_2/\emptyset \\ \stackrel{A}{\Longrightarrow} \texttt{node}\ f_A(x_A, c'_1, \ldots, c'_p) = (e', c_1, \ldots, c_n)\ \texttt{where}\ D_1\ \texttt{and}\ D_2 \end{array}}$$

(Node-P-Suppr)

$$\frac{\begin{array}{c} H, x : t_1, H_1 | G \vdash D : H_1/\ell_1/T_1 \qquad H, x : t_1, H_1 | G \vdash e : t_2/\ell_2/T_2 \\ A \notin \ell_1 \cup \ell_2 \qquad \exists t. \operatorname{gen}_H(t_1 \prec\!\langle \ell_1 \cup \ell_2/T_1, T_2 \rangle\!\rightarrow t_2) = \forall \alpha_1, \ldots, \alpha_n . t \end{array}}{\begin{array}{c} H | G \vdash \texttt{node}\ f(x) = e\ \texttt{where}\ D : \\ [\operatorname{gen}_H(t_1 \prec\!\langle \ell_1 \cup \ell_2/T_1, T_2 \rangle\!\rightarrow t_2)/f]/\ell_1 \cup \ell_2/\emptyset \stackrel{A}{\Longrightarrow} \epsilon \end{array}}$$

(Node-P-Gen)

$$\frac{\begin{array}{c} H, x : t_1, H_1 | G \vdash D : H_1/\ell_1/T_1 \qquad H, x : t_1, H_1 | G \vdash e : t_2/\ell_2/T_2 \\ \exists tc. \operatorname{gen}_H(t_1 \prec\!\langle \ell_1 \cup \ell_2/T_1, T_2 \rangle\!\rightarrow t_2) = \forall \alpha_1, \ldots, \alpha_n . \forall \delta . tc\ \texttt{at}\ \delta \end{array}}{\begin{array}{c} H | G \vdash \texttt{node}\ f(x) = e\ \texttt{where}\ D : \\ [\operatorname{gen}_H(t_1 \prec\!\langle \ell_1 \cup \ell_2/T_1, T_2 \rangle\!\rightarrow t_2)/f]/\ell_1 \cup \ell_2/\emptyset \\ \stackrel{A}{\Longrightarrow} \texttt{node}\ f_A(x_A) = e\ \texttt{where}\ D \end{array}}$$

Fig. 6.6 – Définition de l'opération de projection (règles pour les nœuds).

Projection des équations (figure 6.7)

La projection d'une équation simple consiste en l'annotation des variables définies, et l'ajout en parallèle des équations définies par la projection de l'expression définissant cette variable (règle DEF-P). L'équation est supprimée si le résultat de la projection de l'expression qui la compose est l'expression spéciale $\perp$ (règle DEF-P-SUPPR).

La projection d'une application consiste en la projection des expressions la composant, et l'ajout en entrée et en sortie des canaux de communications utilisés au sein du nœud appliqué ($A \in \ell_1$, règle APP-P). Les équations additionnelles issues de la projection des expressions composant l'application sont ajoutées en parallèle de l'application projetée. L'application est supprimée si le site sur lequel la projection est effectuée ne participe pas à l'évaluation du nœud appliqué ($A \notin \ell_1$, règle APP-P-SUPPR).

La projection d'une composition d'équations en parallèle consiste en la composition parallèle de la projection de ses composantes (règle AND-P).

La projection d'équations dont la portée est restreinte par `let/in` requiert de sortir la définition des canaux sortants de cette portée. Les canaux sont donc immédiatement redéfinis, avec la même valeur (règle LET-P).

Reprenons par exemple les équations de la section 6.1.1, restreintes par un `let/in` :

```
let y = x + 1
and z = y + 2 at B in
t = z
```

La dernière équation utilisant la valeur de `z` calculée sur B, elle sera placée sur le site B. Si on procède maintenant de façon naïve, sans redéfinir le canal `c` en dehors de la portée du `let/in`, on obtient la projection suivante :

A	B
let c = x_A + 1 in ϵ	let y_B = c and z_B = y_B + 2 in t_B = z_B

La composition parallèle de ces deux programmes est alors incorrecte, car la variable `c`, utilisée sur B, n'est pas définie. Nous obtenons, par application de la règle LET-P, le résultat correct de la projection de ces équations :

A	B
let c = x_A + 1 in c = c	let y_B = c and z_B = y_B + 2 in t_B = z_B

Le fonctionnement de cette opération de projection est illustré sur les équations de la section 6.1.1 : la figure 6.8 montre leur projection sur le site A et la figure 6.9 leur projection sur le site B.

6.4 Correction de l'opération de projection

La correction de l'opération de projection sera démontrée en deux étapes : nous allons d'abord montrer que cette opération peut être appliquée sur n'importe quel programme bien typé au sens des types spatiaux (complétude). Ensuite, l'équivalence sémantique du programme réparti avec le programme initial sera démontrée (correction).

Dans cette section, on suppose que tout programme, expression ou équation est typé dans l'architecture $G = \langle \mathcal{S}, \mathcal{L} \rangle$. Pour des raisons de clarté, les équations et expressions de cette section sont évaluées implicitement dans l'environnement composé des déclarations de nœuds associées.

6.4.1 Complétude

Le premier résultat utilise le lemme suivant, qui établit que le résultat de la projection sur un site A d'une expression e est l'expression absente $\perp$ si et seulement si A n'apparaît pas dans le type spatial de e.

Lemme 6. *Pour tout H, G, e, t, ℓ, T, D, et pour tout site A,*

$$H|G \vdash e : t/\ell/T \overset{A}{\Longrightarrow} \perp/D \Leftrightarrow A \notin \text{locations}(t)$$

Démonstration. Par induction sur la structure de l'arbre de dérivation issu de l'application de la projection. Les seules règles pouvant produire l'expression $e' = \perp$ sont les règles Imm-P-Suppr, Inst-P-Suppr, Pair-P, Op-P-Suppr, At-P, Comm-P-From et Comm-P-Suppr. La conclusion des autres règles vérifie simultanément $e' \neq \perp$ et $A \in \text{locations}(t)$.
Soit $H, G = \langle \mathcal{S}, \mathcal{L} \rangle, e, t, \ell, T, D, A$ tels que $H|G \vdash e : t/\ell/T \overset{A}{\Longrightarrow} \perp/D$.

Cas de la règle Imm-P-Suppr : On a $e = i$, $t = b$ `at` A', $\ell = \{A'\}$ avec $A' \neq A$. Donc $A \notin \text{locations}(t) = \{A'\}$.

(Def-P)

$$\frac{H|G \vdash e : t_1 \times \ldots \times t_n/\ell/T \stackrel{A}{\Longrightarrow} e'/D}{H|G \vdash (x_1, \ldots, x_n) = e : [t_1/x_1, \ldots, t_n/x_n]/\ell/T \stackrel{A}{\Longrightarrow} (x_{1A}, \ldots, x_{nA}) = e' \texttt{ and } D}$$

(Def-P-Suppr)

$$\frac{H|G \vdash e : t_1 \times \ldots \times t_n/\ell/T \stackrel{A}{\Longrightarrow} \bot/D}{H|G \vdash (x_1, \ldots, x_n) = e : [t_1/x_1, \ldots, t_n/x_n]/\ell/T \stackrel{A}{\Longrightarrow} D}$$

(App-P)

$$\frac{\begin{array}{c} H|G \vdash f : t_1 \prec\!\langle \ell_1/T_1 \rangle\!\succ t_2/\ell_2/T_2 \stackrel{A}{\Longrightarrow} f'/D_1 \\ H|G \vdash e : t_1/\ell_3/T_3 \stackrel{A}{\Longrightarrow} e'/D_2 \qquad T_1' \cong T_1 \\ T_1' \uparrow A = [A \stackrel{c_1}{\mapsto} A_1, \ldots, A \stackrel{c_n}{\mapsto} A_n] \qquad T_1' \downarrow A = [A_1' \stackrel{c_1'}{\mapsto} A, \ldots, A_p' \stackrel{c_p'}{\mapsto} A] \qquad A \in \ell_1 \end{array}}{\begin{array}{c} H|G \vdash x = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3/T_1', T_2, T_3 \\ \stackrel{A}{\Longrightarrow} (x_A, c_1, \ldots, c_n) = f'(e', c_1', \ldots, c_p') \texttt{ and } D_1 \texttt{ and } D_2 \end{array}}$$

(App-P-Suppr)

$$\frac{\begin{array}{c} H|G \vdash f : t_1 \prec\!\langle \ell_1/T_1 \rangle\!\succ t_2/\ell_2/T_2 \stackrel{A}{\Longrightarrow} \bot/D_1 \\ H|G \vdash e : t_1/\ell_3/T_3 \stackrel{A}{\Longrightarrow} \bot/D_2 \qquad T_1' \cong T_1 \qquad A \notin \ell_1 \end{array}}{H|G \vdash x = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3/T_1', T_2, T_3 \stackrel{A}{\Longrightarrow} D_1 \texttt{ and } D_2}$$

(And-P)

$$\frac{H|G \vdash D_1 : H_1/\ell_1/T_1 \stackrel{A}{\Longrightarrow} D_1' \qquad H|G \vdash D_2 : H_2/\ell_2/T_2 \stackrel{A}{\Longrightarrow} D_2'}{H|G \vdash D_1 \texttt{ and } D_2 : H_1, H_2/\ell_1 \cup \ell_2/T_1, T_2 \stackrel{A}{\Longrightarrow} D_1' \texttt{ and } D_2'}$$

(Let-P)

$$\frac{\begin{array}{c} H|G \vdash D_1 : H_1/\ell_1/T_1 \stackrel{A}{\Longrightarrow} D_1' \\ H, H_1|G \vdash D_2 : H_2/\ell_2/T_2 \stackrel{A}{\Longrightarrow} D_2' \qquad T_2 \uparrow A = [A \stackrel{c_1}{\mapsto} A_1, \ldots, A \stackrel{c_n}{\mapsto} A_p] \end{array}}{\begin{array}{c} H|G \vdash \texttt{let } D_1 \texttt{ in } D_2 : H_1, H_2/\ell_1 \cup \ell_2/T_1, T_2 \\ \stackrel{A}{\Longrightarrow} \texttt{let } D_1' \texttt{ in } (D_2' \texttt{ and } c_1 = c_1 \texttt{ and } \ldots \texttt{ and } c_n = c_n) \end{array}}$$

Fig. 6.7 – Définition de l'opération de projection (règles pour les équations).

$$
\begin{array}{c}
\dfrac{\begin{array}{c} b \texttt{ at } A \times b \texttt{ at } A \prec\!\{A\}/\emptyset\!\succ\!\rightarrow b \texttt{ at } A \\ \leq \forall\delta.b \texttt{ at } \delta \times b \texttt{ at } \delta \prec\!\{\delta\}/\emptyset\!\succ\!\rightarrow b \texttt{ at } \delta \qquad A \in \{A\} \end{array}}{H_1|G \vdash (+) : b \texttt{ at } A \times b \texttt{ at } A \prec\!\{A\}/\emptyset\!\succ\!\rightarrow b \texttt{ at } A/\{A\}/\emptyset \overset{A}{\Longrightarrow} (+)_A/\epsilon}\ (\textsc{Inst-P}) \\
\vdots \\
\dfrac{\dfrac{\dfrac{\begin{array}{c} \vdots \\ H_1|G \vdash x : b \texttt{ at } A/\{A\}/\emptyset \overset{A}{\Longrightarrow} x_A/\epsilon \qquad H_1|G \vdash 1 : b \texttt{ at } A/\{A\}/\emptyset \overset{A}{\Longrightarrow} 1/\epsilon \end{array}}{H_1|G \vdash x+1 : b \texttt{ at } A/\{A\}/\emptyset \overset{A}{\Longrightarrow} x_A + 1/\epsilon}\ (\textsc{Op-P})}{H_1|G \vdash x+1 : b \texttt{ at } B/\{A,B\}/[A \overset{c}{\mapsto} B] \overset{A}{\Longrightarrow} \bot/(c = x_A + 1)}\ (\textsc{Comm-From})}{H_1|G \vdash y = x+1 : [y : b \texttt{ at } B]/\{A,B\}/[A \overset{c}{\mapsto} B] \overset{A}{\Longrightarrow} c = x_A + 1}\ (\textsc{Def-P-Suppr}) \\
\dfrac{\begin{array}{c} \vdots \\ (\textsc{Def-P-Suppr})\ \dfrac{(\textsc{Op-P-Suppr})\ \dfrac{(\textsc{At-P})\ \dfrac{\dfrac{(\textsc{Inst-P-Suppr})\ \dfrac{\begin{array}{c} b \texttt{ at } B \times b \texttt{ at } B \prec\!\{B\}/\emptyset\!\succ\!\rightarrow b \texttt{ at } B \\ \leq \forall\delta.b \texttt{ at } \delta \times b \texttt{ at } \delta \prec\!\{\delta\}/\emptyset\!\succ\!\rightarrow b \texttt{ at } \delta \qquad A \notin \{B\} \end{array}}{H_1|G \vdash (+) : b \texttt{ at } B \times b \texttt{ at } B \prec\!\{B\}/\emptyset\!\succ\!\rightarrow b \texttt{ at } B/\{B\}/\emptyset \overset{A}{\Longrightarrow} \bot/\epsilon}}{H_1|G \vdash y : b \texttt{ at } B/\{B\}/\emptyset \overset{A}{\Longrightarrow} \bot/\epsilon} \quad H_1|G \vdash 2 : b \texttt{ at } B/\{B\}/\emptyset \overset{A}{\Longrightarrow} \bot/\epsilon}{H_1|G \vdash y+2 : b \texttt{ at } B/\{B\}/\emptyset \overset{A}{\Longrightarrow} \bot/\epsilon}}{H_1|G \vdash y+2 \texttt{ at } B : b \texttt{ at } B/\{B\}/\emptyset \overset{A}{\Longrightarrow} \bot/\epsilon}}{H_1|G \vdash z = y+2 \texttt{ at } B : [z : b \texttt{ at } B]/\{B\}/\emptyset \overset{A}{\Longrightarrow} \bot/\epsilon} \end{array}}{\begin{array}{c} H_1|G \vdash \begin{array}{c} y = x+1 \\ \texttt{and} \quad z = y+2 \texttt{ at } B \end{array} : \left[\begin{array}{c} y : b \texttt{ at } B, \\ z : b \texttt{ at } B \end{array}\right] /\{A,B\}/[A \overset{c}{\mapsto} B] \\ \overset{A}{\Longrightarrow} c = x_A + 1 \end{array}}\ (\textsc{And-P})
\end{array}
$$

Avec $H_1 = H_0, [x : b \texttt{ at } A, y : b \texttt{ at } B, z : b \texttt{ at } B]$ et $G = \langle\{A,B\},\{(A,B)\}\rangle$.

FIG. 6.8 – Exemple de projections d'équations (projection sur A).

$$
\begin{array}{c}
\dfrac{\begin{array}{c} b \ \texttt{at}\ A \times b \ \texttt{at}\ A \prec\!\{A\}/\emptyset\!\succ\!\!\rightarrow b \ \texttt{at}\ A \\ \leq \forall\delta.b \ \texttt{at}\ \delta \times b \ \texttt{at}\ \delta \prec\!\{\delta\}/\emptyset\!\succ\!\!\rightarrow b \ \texttt{at}\ \delta \qquad B \notin \{A\} \end{array}}{H_1|G \vdash (+) : b \ \texttt{at}\ A \times b \ \texttt{at}\ A \prec\!\{A\}/\emptyset\!\succ\!\!\rightarrow b \ \texttt{at}\ A/\{A\}/\emptyset \overset{B}{\Longrightarrow} \bot/\epsilon}\ (\textsc{Inst-P-Suppr}) \\
\vdots \\
\dfrac{\begin{array}{c} \dfrac{\dfrac{\dfrac{\dots \quad H_1|G \vdash x : b \ \texttt{at}\ A/\{A\}/\emptyset \overset{B}{\Longrightarrow} \bot/\epsilon \qquad H_1|G \vdash 1 : b \ \texttt{at}\ A/\{A\}/\emptyset \overset{B}{\Longrightarrow} \bot/\epsilon}{H_1|G \vdash x+1 : b \ \texttt{at}\ A/\{A\}/\emptyset \overset{B}{\Longrightarrow} \bot/\epsilon}\ (\textsc{Op-Suppr})}{H_1|G \vdash x+1 : b \ \texttt{at}\ B/\{A,B\}/[A \overset{c}{\mapsto} B] \overset{B}{\Longrightarrow} c/\epsilon}\ (\textsc{Comm-P-To})}{H_1|G \vdash y = x+1 : [y : b \ \texttt{at}\ B]/\{A,B\}/[A \overset{c}{\mapsto} B] \overset{B}{\Longrightarrow} y_B = c}\ (\textsc{Def-P}) \\
\vdots \\
\dfrac{\begin{array}{c} (\textsc{Inst-P}) \\ \dfrac{\begin{array}{c} b \ \texttt{at}\ B \times b \ \texttt{at}\ B \prec\!\{B\}/\emptyset\!\succ\!\!\rightarrow b \ \texttt{at}\ B \\ \leq \forall\delta.b \ \texttt{at}\ \delta \times b \ \texttt{at}\ \delta \prec\!\{\delta\}/\emptyset\!\succ\!\!\rightarrow b \ \texttt{at}\ \delta \qquad B \in \{B\} \end{array}}{H_1|G \vdash (+) : b \ \texttt{at}\ B \times b \ \texttt{at}\ B \prec\!\{B\}/\emptyset\!\succ\!\!\rightarrow b \ \texttt{at}\ B/\{B\}/\emptyset \overset{B}{\Longrightarrow} (+)_B/\epsilon} \\ H_1|G \vdash y : b \ \texttt{at}\ B/\{B\}/\emptyset \overset{B}{\Longrightarrow} y_B/\epsilon \\ \dfrac{\dfrac{\dfrac{H_1|G \vdash 2 : b \ \texttt{at}\ B/\{B\}/\emptyset \overset{B}{\Longrightarrow} 2/\epsilon}{H_1|G \vdash y+2 : b \ \texttt{at}\ B/\{B\}/\emptyset \overset{B}{\Longrightarrow} y_B+2/\epsilon}\ (\textsc{Op-P})}{H_1|G \vdash y+2 \ \texttt{at}\ B : b \ \texttt{at}\ B/\{B\}/\emptyset \overset{B}{\Longrightarrow} y_B+2/\epsilon}\ (\textsc{At-P})}{H_1|G \vdash z = y+2 \ \texttt{at}\ B : [z : b \ \texttt{at}\ B]/\{B\}/\emptyset \overset{B}{\Longrightarrow} z_B = y_B+2}\ (\textsc{Def-P}) \end{array}}{H_1|G \vdash \begin{array}{c} y = x+1 \\ \texttt{and}\ \ z = y+2 \ \texttt{at}\ B \end{array} : \left[\begin{array}{c} y : b \ \texttt{at}\ B, \\ z : b \ \texttt{at}\ B \end{array}\right] /\{A,B\}/[A \overset{c}{\mapsto} B] \overset{B}{\Longrightarrow} \begin{array}{c} y_B = c \\ \texttt{and}\ \ z_B = y_B + 2 \ \texttt{at}\ B \end{array}}\ (\textsc{And-P})
\end{array}
$$

Avec $H_1 = H_0, [x : b \ \texttt{at}\ A, y : b \ \texttt{at}\ B, z : b \ \texttt{at}\ B]$.

FIG. 6.9 – Exemple de projections d'équations (projection sur B).

Cas de la règle Inst-P-Suppr : D'après la règle considérée, $A \notin \text{locations}(t)$.

Cas de la règle Pair-P : On a $e = (e_1, e_2)$, $t = t_1 \times t_2$, $\ell = \ell_1 \cup \ell_2$, $T = T_1, T_2$, $D = D_1$ `and` D_2, avec $H|G \vdash e_1 : t_1/\ell_1/T_1 \overset{A}{\Longrightarrow} e_1'/D_1$ et $H|G \vdash e_2 : t_2/\ell_2/T_2 \overset{A}{\Longrightarrow} e_2'/D_2$. $H|G \vdash e : t/\ell/T \overset{A}{\Longrightarrow} \bot/D$ si et seulement si $e_1' = e_2' = \bot$. Par hypothèse d'induction, $A \notin \text{locations}(t_1)$ et $A \notin \text{locations}(t_2)$, donc $A \notin \text{locations}(t)$.
Réciproquement, si $A \notin \text{locations}(t)$, alors $A \notin \text{locations}(t_1)$ et $A \notin \text{locations}(t_2)$. Par hypothèse d'induction, $e_1' = e_2' = \bot$, donc $e' = (e_1', e_2') = \bot$.

Cas de la règle Op-P-Suppr : Il existe $f, e_1, \ell_1, t_1, s, D_1$ et D_2 tels que $e = f(e_1)$, $\ell = \ell_1 \cup \{s\}$, $D = D_1$ `and` D_2,
$H|G \vdash f : t_1 \prec\!\{s\}/\emptyset\!\succ\!\rightarrow t/\{s\}/\emptyset \overset{A}{\Longrightarrow} \bot/D_1$ et $H|G \vdash e_1 : t_1/\ell_1/T \overset{A}{\Longrightarrow} \bot/D_2$. Par hypothèse d'induction,

$$A \notin \text{locations}(t_1 \prec\!\{s\}/\emptyset\!\succ\!\rightarrow t).$$

Donc, $A \notin \text{locations}(t)$.
Réciproquement, si $A \notin \text{locations}(t)$, $A \notin \text{locations}(t_1 \prec\!\{s\}/\emptyset\!\succ\!\rightarrow t)$, puisque d'après la définition de H_0, il existe tc_1, tc_2 tels que

$$t_1 \prec\!\{s\}/\emptyset\!\succ\!\rightarrow t = (tc_1 \rightarrow tc_2) \texttt{ at } s,$$

d'où $\text{locations}(t_1 \prec\!\{s\}/\emptyset\!\succ\!\rightarrow t) = \text{locations}(t)$. Donc $A \notin \text{locations}(t_1)$.

Cas de la règle At-P : Induction directe.

Cas des règles Comm-P-From et Comm-P-Suppr : On a $t = tc$ `at` s', avec $A \neq s'$. Donc $A \notin \text{locations}(t) = \{s'\}$.

□

Le lemme 7 établit que la projection peut être appliquée sur toute expression bien typée.

Lemme 7. *Pour tout H, e, t, ℓ, T, si $H|G \vdash e : t/\ell/T$ alors pour tout site A de $\mathcal{S}$, il existe une expression e' et un ensemble d'équations D tels que :*

$$H|G \vdash e : t/\ell/T \overset{A}{\Longrightarrow} e'/D.$$

Démonstration. Par induction sur la structure de l'arbre de dérivation du système de types sur l'expression. L'induction est effectuée selon la dernière règle de typage appliquée.

Soient H, e, t, ℓ et T tels que $H|G \vdash e : t/\ell/T$. Soit $A \in \mathcal{S}$.

Cas de la règle Imm-C : Il existe s et i tels que $e = i$, $t = b$ `at` s, $\ell = \{s\}$ et $T = \emptyset$.

- Si $s = A$, alors d'après la règle Imm-P, $H|G \vdash i : b$ `at` $A/\{A\}/\emptyset \overset{A}{\Longrightarrow} i/\epsilon$.
- Sinon, d'après la règle Imm-P-Suppr, $H|G \vdash i : b$ `at` $A'/\{A'\}/\emptyset \overset{A}{\Longrightarrow} \bot/\epsilon$.

Cas de la règle Inst-C : On a $e = x$, $T = \emptyset$ et $\ell = \text{locations}(t)$.

- Si $A \in \text{locations}(t)$, alors d'après la règle Inst-P,
  $H|G \vdash x : t/\text{locations}(t)/\emptyset \overset{A}{\Longrightarrow} x_A/\epsilon$.
- Sinon, d'après la règle Inst-P-Suppr, $H|G \vdash x : t/\text{locations}(t)/\emptyset \overset{A}{\Longrightarrow} \bot/\epsilon$.

Cas de la règle Pair-C : Induction directe, et application de la règle Pair-P.

Cas de la règle Op-C : Il existe f, e_1, t_1, t_2, s et ℓ_1 tels que $e = f(e_1)$, $t = t_2$, $\ell = \ell_1 \cup \{s\}$, $H|G \vdash f : t_1 \prec\!\{\{s\}/\emptyset\}\!\succ t_2/\{s\}/\emptyset$ et $H|G \vdash e_1 : t_1/\ell_1/T$.

Par hypothèse d'induction, il existe f', D_1, e'_1 et D_2 tels que :

$$H|G \vdash f : t_1 \prec\!\{\{A\}/\emptyset\}\!\succ t_2/\{A\}/\emptyset \overset{A}{\Longrightarrow} f'/D_1 \text{ et } H|G \vdash e : t_1/\ell/T \overset{A}{\Longrightarrow} e'/D_2$$

- Si $s = A$, alors d'après la règle Op-P,

$$H|G \vdash f(e) : t_2/\{s\} \cup \ell/T \overset{A}{\Longrightarrow} f'(e')/D_1 \texttt{ and } D_2.$$

- Sinon, d'après le lemme 6, $f' = \bot$ et $e'_1 = \bot$. Alors, d'après la règle Op-P-Suppr,

$$H|G \vdash f(e) : t_2/\{s\} \cup \ell/T \overset{A}{\Longrightarrow} \bot/D_1 \texttt{ and } D_2.$$

Cas de la règle At-C : Induction directe et application de la règle At-P.

Cas de la règle Comm-C : Il existe tc, s, s', c, ℓ' et T' tels que $t = tc$ `at` s', $\ell = \ell' \cup \{s'\}$, $T = T', [s \overset{c}{\mapsto} s']$, $(s, s') \in \mathcal{L}$ et $H|G \vdash e : tc$ `at` $s/\ell'/T'$.

Par hypothèse d'induction, il existe e' et D tels que :

$$H|G \vdash e : tc \texttt{ at } s/\ell'/T' \overset{A}{\Longrightarrow} e'/D.$$

- Si $s = A$, alors d'après la règle Comm-P-From,

$$H|G \vdash e : t/\ell \cup \{s'\}/T, [A \overset{c}{\mapsto} s'] \overset{A}{\Longrightarrow} \bot/D \texttt{ and } c = e'.$$

- Si $s' = A$, d'après le lemme 6, $e' = \bot$. Alors, d'après la règle Comm-P-To,

$$H|G \vdash e : tc \texttt{ at } A'/\ell \cup \{A'\}/T, [s \overset{c}{\mapsto} A'] \overset{A'}{\Longrightarrow} c/D.$$

– Sinon, d'après le lemme 6, $e' = \bot$. Alors, d'après la règle COMM-P-SUPPR,

$$H|G \vdash e : tc \texttt{ at } s'/\ell \cup \{s'\}/T, [s \overset{c}{\mapsto} s'] \overset{A}{\Longrightarrow} \bot/D.$$

□

Le théorème 3 établit que la projection peut être appliquée sur tout nœud ou tout ensemble d'équations bien typé.

Théorème 3.

1. *Pour tout H, H', D, e, ℓ et T, si $H|G \vdash$ **node** $f(x) = e$ **where** $D : H'/\ell/T$ alors pour tout A de $\mathcal{S}$, il existe d tel que :*

$$H|G \vdash \textbf{node } f(x) = e \textbf{ where } D : H'/\ell/T \overset{A}{\Longrightarrow} d.$$

2. *Pour tout H, H', D, ℓ et T, si $H|G \vdash D : H'/\ell/T$ alors pour tout A de $\mathcal{S}$, il existe un ensemble d'équations D' tel que $H|G \vdash D : H'/\ell/T \overset{A}{\Longrightarrow} D'$.*

Démonstration. Par induction sur la structure de l'arbre de dérivation du système de types sur l'expression. L'induction est effectuée selon la dernière règle de typage appliquée.
Soit $A \in \mathcal{S}$.

Cas des nœuds :

Soient H, D, e, H', ℓ et T tels que $H|G \vdash \texttt{node } f(x) = e \texttt{ where } D : H'/\ell/T$.
D'après la règle NODE-C, il existe t, t_1, ℓ_1, ℓ_2, T_1, T_2 et H_1 tels que :
– $\ell = \ell_1 \cup \ell_2$,
– $T = T_1, T_2$,
– $H' = [\text{gen}_H(t \prec\!\langle \ell/T \rangle\!\succ t_1)/f]$,
– $H, x : t, H_1|G \vdash D : H_1/\ell_1/T_1$ et
– $H, x : t, H_1|G \vdash e : t/\ell_2/T_2$.

– Si il existe tc tel que $\text{gen}_H(t \prec\!\langle \ell/T \rangle\!\succ t_1) = \forall \alpha_1, \ldots, \alpha_n . \forall \delta . tc \texttt{ at } \delta$, alors la règle NODE-P-GEN s'applique :

$$\begin{aligned} &H|G \vdash \texttt{node } f(x) = e \texttt{ where } D : [\text{gen}_H(t \prec\!\langle \ell_1 \cup \ell_2/T_1, T_2 \rangle\!\succ t_1)/f]/\ell_1 \cup \ell_2/\emptyset \\ &\overset{A}{\Longrightarrow} \texttt{node } f_A(x_A, c'_1, \ldots, c'_p) = (e', c_1, \ldots, c_n) \texttt{ where } D_1 \texttt{ and } D_2 \end{aligned}$$

– Sinon, d'après la définition de gen_H, il existe t' tel que $\text{gen}_H(t \prec\!\langle \ell/T \rangle\!\succ t_1) = \forall \alpha_1, \ldots, \alpha_n . t'$. Si $A \notin \ell$, alors la règle NODE-P-SUPPR s'applique :
$H|G \vdash \texttt{node } f(x) = e \texttt{ where } D : H'/\ell/T \overset{A}{\Longrightarrow} \epsilon$.

– Si $A \in \ell$, par hypothèse d'induction, il existe D_1 tel que $H, x : t, H_1|G \vdash D : H_1/\ell_1/T_1 \overset{A}{\Longrightarrow} D_1$. D'après le lemme 7, il existe e' et D_2 tels que $H, x_t, H_1|G \vdash e : t/\ell_2/T_2 \overset{A}{\Longrightarrow} e'/D_2$. La règle NODE-P s'applique : soient $T \uparrow A = [A \overset{c_1}{\mapsto} A_1, \ldots, A \overset{c_n}{\mapsto} A_p]$ et $T \downarrow A = [A \overset{c'_1}{\mapsto} A_1, \ldots, A \overset{c'_p}{\mapsto} A_q]$, alors on a :

$$\begin{aligned} &H|G \vdash \texttt{node}\ f(x) = e\ \texttt{where}\ D : [\mathrm{gen}_H(t \prec\!\langle \ell_1 \cup \ell_2/T_1, T_2\rangle\!\succ t_1)/f]/\ell_1 \cup \ell_2/\emptyset \\ &\overset{A}{\Longrightarrow} \texttt{node}\ f_A(x_A, c'_1, \ldots, c'_p) = (e', c_1, \ldots, c_n)\ \texttt{where}\ D_1\ \texttt{and}\ D_2 \end{aligned}$$

Cas des équations :

Soient H, D, H', ℓ et T tels que $H|G \vdash D : H'/\ell/T$.

Cas de la règle Def-C : Il existe $x_1, \ldots, x_n, t = (t_1 \times \ldots \times t_n)$ et e tels que $D = ((x_1, \ldots, x_n) = e)$, $H' = [t_1/x_1, \ldots, t_n/x_n]$ et $H|G \vdash e : t/\ell/T$. D'après le lemme 7, il existe e' et D' t. q. $H|G \vdash e : t/\ell/T \overset{A}{\Longrightarrow} e'/D'$.
– Si $e' = \bot$, la règle DEF-P-SUPPR s'applique : $H|G \vdash (x_1, \ldots, x_n) = e : [t/x]/\ell/T \overset{A}{\Longrightarrow} D'$.
– Sinon, la règle DEF-P s'applique : $H|G \vdash (x_1, \ldots, x_n) = e : [t/x]/\ell/T \overset{A}{\Longrightarrow} (x_{1A}, \ldots, x_{nA}) = e'\ \texttt{and}\ D'$.

Cas de la règle App-C : Il existe x, f, e, t_1, t_2, ℓ_1, ℓ_2, ℓ_3, T_1, T'_1, T_2 et T_3 tels que $D = (x = f(e))$, $H' = [t_2/x]$, $\ell = \ell_1 \cup \ell_2 \cup \ell_3$, $T = T'_1, T_2, T_3$, $T'_1 \cong T_1$, $H|G \vdash f : t_1 \prec\!\langle \ell_1/T_1\rangle\!\succ t_2/\ell_2/T_2$ et $H|G \vdash e : t_1/\ell_3/T_3$. Par hypothèse d'induction, il existe f', D_1, e' et D_2 tels que

$$H|G \vdash f : t_1 \prec\!\langle \ell_1/T_1\rangle\!\succ t_2/\ell_2/T_2 \overset{A}{\Longrightarrow} f'/D_1 \text{ et } H|G \vdash e : t_1/\ell_3/T_3 \overset{A}{\Longrightarrow} e'/D_2.$$

– Si $A \in \ell_1$, alors la règle APP-P s'applique : soient $T'_1 \uparrow A = [A \overset{c_1}{\mapsto} A_1, \ldots, A \overset{c_n}{\mapsto} A_p]$ et $T'_1 \downarrow A = [A_1 \overset{c'_1}{\mapsto} A, \ldots, A_q \overset{c'_p}{\mapsto} A]$, alors on a :

$$\begin{aligned} &H|G \vdash x = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3/T'_1, T_2, T_3 \\ &\overset{A}{\Longrightarrow} (x, c'_1, \ldots, c'_p) = f'(e', c_1, \ldots, c_n)\ \texttt{and}\ D_1\ \texttt{and}\ D_2 \end{aligned}$$

– Sinon, $A \notin \mathrm{locations}(t_1 \prec\!\langle \ell_1/T_1\rangle\!\succ t_2)$. Donc, $A \notin \mathrm{locations}(t_2)$. D'après le lemme 6, $f' = \bot$ et $e' = \bot$. Alors la règle APP-P-SUPPR s'applique :

$$H|G \vdash D_1\ \texttt{and}\ D_2 : H_1, H_2/\ell_1 \cup \ell_2/T_1, T_2 \overset{A}{\Longrightarrow} D'_1\ \texttt{and}\ D'_2$$

Cas de la règle And-C : Induction directe et application de la règle AND-P.

Cas de la règle Let-C : Induction directe et application de la règle LET-P.

□

6.4.2 Équivalence sémantique

Montrons maintenant l'équivalence sémantique du programme réparti avec le programme initial. La sémantique du programme réparti est définie ici par le produit synchrone des équations résultant de la projection de ce programme.

On appelle dans la suite $\mathcal{S} = \{A_1, \ldots, A_n\}$ l'ensemble des sites déclarés de l'architecture. La sémantique d'un ensemble d'équations D, projeté sur l'architecture $\langle \mathcal{S}, \mathcal{L} \rangle$, est définie par :

$$\begin{array}{l} D_1 \texttt{ and } \ldots \texttt{ and } D_n \\ \text{où } \forall i, H|G \vdash D : H'/\ell/T \stackrel{A_i}{\Longrightarrow} D_i \end{array}$$

Les valeurs émises par les équations d'un programme réparti peuvent être des composantes de valeurs réparties. Ces valeurs ne seront donc en général pas liées par une relation d'égalité avec les valeurs émises par le programme initial. Nous définissons donc une relation, d'après le type spatial des expressions, afin de lier ces valeurs partielles émises par le programme réparti avec les valeurs initiales. Cette relation, notée $v' \preccurlyeq_t^A v$, signifie que la valeur partielle v', émise par une expression de type spatial t projetée sur le site A, est la composante de la valeur v émise par le programme initial. Cette relation est définie par les prédicats ci-dessous :

$$v \preccurlyeq_{tc \texttt{ at } A}^A v \qquad \frac{A \neq A'}{\bot \preccurlyeq_{tc \texttt{ at } A'}^A v} \qquad \frac{v_1' \preccurlyeq_{t_1}^A v_1 \qquad v_2' \preccurlyeq_{t_2}^A v_2}{(v_1', v_2') \preccurlyeq_{t_1 \times t_2}^A (v_1, v_2)}$$

Une valeur émise par une expression de type localisé sur le site A doit être égale à la valeur émise par le programme initial. Une valeur émise par une expression de type localisé sur un site différent de A doit être la valeur absente. Le caractère partiel des valeurs émises apparaît pour les paires, dont les composantes peuvent être émises sur des sites différents.

Par exemple, si une expression du programme initial émet la valeur $((1, \texttt{true}), 2)$, l'expression étant de type spatial $(b \texttt{ at } A \times b \texttt{ at } B) \times b \texttt{ at } A$, alors la projection de cette expression sur le site A émettra la valeur partielle $((1, \bot), 2)$: la relation ci-dessous est vérifiée :

$$((1, \bot), 2) \preccurlyeq_{(b \texttt{ at } A \times b \texttt{ at } B) \times b \texttt{ at } A}^A ((1, \texttt{true}), 2)$$

Cette relation permet de définir une autre relation entre environnements de réaction. Si l'exécution d'un programme centralisé émet un environnement de réaction R dont le domaine est $\{x_1, \ldots, x_n\}$, alors l'exécution du programme réparti issu de la projection de ce programme sur les sites $\mathcal{S}$ émet un environnement de réaction R_p dont le domaine est un sous-ensemble de :

$$\{x_{iA} \mid i \in \{1, \ldots, n\}, A \in \mathcal{S}\} \cup \mathrm{dom}(T)$$

où T est la séquence de canaux de communication nécessaires pour l'exécution du programme réparti.

On dit que R_p est *compatible avec R selon l'environnement de typage H*, noté $R_p \preccurlyeq_H R$, si les valeurs partielles définies par les variables annotées par un site A dans R_p représentent les valeurs définies par R sur le site A :

$$R_p \preccurlyeq_H R \text{ ssi } \forall x \in \mathrm{dom}(R), \forall A \in \mathcal{S}, v \preccurlyeq^A_{H(x)} R(x),$$
$$\text{où } v = \begin{cases} R_p(x_A) & \text{si } x_A \in \mathrm{dom}(R_p) \\ \bot & \text{sinon.} \end{cases}$$

Le lemme 8 établit la correction de la projection des expressions. Le résultat de la projection d'une expression e est un ensemble de paires e_i/D_i, D_i comprenant la définition des canaux de communication utilisés pour l'évaluation de e. Pour ce résultat, la sémantique de chaque expression e_i est évaluée dans un environnement R_p, R'_p, où R_p est compatible avec l'environnement de réaction dans lequel est évaluée e, et R'_p est l'environnement de réaction émis par l'ensemble d'équations D_1 `and` $D_2 \ldots$ `and` D_n. De plus, en réagissant, les e_i/D_i se réécrivent en e'_i/D'_i, résultat de la projection de l'expression e', réécriture de l'expression e réagissant avec la sémantique centralisée.

Lemme 8. *Pour tous H, G, e, v, e', t, ℓ, T, e_i, D_i et R, si*
- $R \vdash e \xrightarrow{v} e'$ *et*
- $\forall i, H|G \vdash e : t/\ell/T \overset{A_i}{\Longrightarrow} e_i/D_i$,

*alors pour tout environnement de réaction R_p compatible avec R selon H, soit $D = D_1$ **and** $\ldots$ **and** D_n, il existe v_i, e'_i, D' et R' tels que :*
- $R_p \vdash D \xrightarrow{R'} D'$,
- $\mathrm{dom}(R') = \mathrm{dom}(T)$,
- $\forall i, R_p, R' \vdash e_i \xrightarrow{v_i} e'_i$ *avec* $v_i \preccurlyeq^{A_i}_t v$,
- $D' = D'_1$ **and** $\ldots$ **and** D'_n *et*
- $\forall i, H|G \vdash e' : t/\ell/T \overset{A_i}{\Longrightarrow} e'_i/D'_i$.

Démonstration. Par induction sur la structure de l'arbre de dérivation du système de types sur e.
Soient H, e, v, e', t, ℓ, T, e_i, D_i, R et R_p, tels que

– $R \vdash e \xrightarrow{v} e'$,
– $\forall i \in \{1, \ldots, n\}, H|G \vdash e : t/\ell/T \overset{A_i}{\Longrightarrow} e_i/D_i$ et
– $R_p \preccurlyeq_H R$.
L'induction est effectuée selon la dernière règle du système de types étendu appliquée pour le typage de e.

Cas de la règle Imm-C : On a $e = e' = i$ et $R \vdash i \xrightarrow{i} i$. L'expression e est bien typée et de type spatial t, ssi il existe un site A tel que $t = b$ `at` A. D'autre part, $\ell = \{A\}$ et $T = \emptyset$.
Soit $\forall j, D_j = D = D' = \epsilon$, $R' = \epsilon$.
Soit $j \in \{1, \ldots, n\}$:
– Si $A_j = A$, soient $e_j = e'_j = i$ et $v_j = i$. Alors, $H|G \vdash e : t/\ell/T \overset{A_j}{\Longrightarrow} e_j/D_j$, $R_p \vdash e_j \xrightarrow{v_j} e'_j$ et $i \preccurlyeq^{A_j}_{b \texttt{ at } A} i$.
– Sinon, soient $e_j = e'_j = \bot$ et $v_j = \bot$. Ces valeurs vérifient $R_p \vdash e_j \xrightarrow{v_j} e'_j$. $A_j \neq A$, donc $H|G \vdash e : t/\ell/T \overset{A_j}{\Longrightarrow} e_j/D_j$ et $v_j \preccurlyeq^{A_j}_{b \texttt{ at } A} i$.

Cas de la règle Inst-C : On a $e = e' = x$ et $R \vdash x \xrightarrow{v} x$, avec $v = R(x)$.
Soit $\forall i, D_i = D = D' = \epsilon$, $R' = \epsilon$.
Soit $i \in \{1, \ldots, n\}$:
– si $A_i \in \text{locations}(t)$, soit $e_i = e'_i = x_{A_i}$: la règle Inst-P s'applique. Soit $v_i = R_p(x_{A_i})$: $R_p \vdash e_i \xrightarrow{v_i} e'_i$ est vérifiée. Par hypothèse, R_p est compatible avec R selon l'environnement H, donc $v_i \preccurlyeq^{A_i}_t v$.
– Sinon, soient $e_i = e'_i = \bot$ et $v_i = \bot$: la règle Inst-P-Suppr s'applique. Puisque $A_i \notin \text{locations}(t)$, alors $v_i \preccurlyeq^{A_i}_t v$.

Cas de la règle Pair-C : Soit $e = (e_a, e_b)$.
D'après la sémantique centralisée, $R \vdash e \xrightarrow{v} e'$ ssi il existe e'_a, e'_b, v_a et v_b tels que $e' = (e'_a, e'_b)$ et $v = (v_a, v_b)$. On a alors $R \vdash e_a \xrightarrow{v_a} e'_a$ et $R \vdash e_b \xrightarrow{v_b} e'_b$.
L'expression e est bien typée ssi il existe ℓ_a, ℓ_b, T_a, T_b, t_a et t_b tels que $\ell = \ell_a \cup \ell_b$, $T = T_a \uplus T_b$, $t = t_a \times t_b$, $H|G \vdash e_a : t_a/\ell_a/T_a$ et $H|G \vdash e_b : t_b/\ell_b/T_b$.
D'après la règle Pair-P, pour tout i, la projection de l'expression e sur le site A_i est la paire e_i/D_i ssi il existe e_{ai}, e_{bi}, D_{ai} et D_{bi} tels que $e_i = (e_{ai}, e_{bi})$,
$D_i = D_{ai}$ `and` D_{bi}, $H|G \vdash e_a : t_a/\ell_a/T_a \overset{A_i}{\Longrightarrow} e_{ai}/D_{ai}$ et
$H|G \vdash e_b : t_b/\ell_b/T_b \overset{A_i}{\Longrightarrow} e_{bi}/D_{bi}$.
Soit $D_a = D_{a1}$ `and` $D_{a2} \ldots$ `and` D_{an} et $D_b = D_{b1}$ `and` $D_{b2} \ldots$ `and` D_{bn}.
Par hypothèse d'induction, pour tout i, il existe v_{ai}, v_{bi}, e'_{ai}, e'_{bi}, D'_a, D'_b, R'_a et R'_b tels que :
– $R_p \vdash D_a \xrightarrow{R'_a} D'_a$, $R_p \vdash D_b \xrightarrow{R'_b} D'_b$,

– $\mathrm{dom}(R'_a) = \mathrm{dom}(T_a)$ et $\mathrm{dom}(R'_b) = \mathrm{dom}(T_b)$,
– $R_p, R'_a \vdash e_{ai} \xrightarrow{v_{ai}} e'_{ai}$ et $R_p, R'_b \vdash e_{bi} \xrightarrow{v_{bi}} e'_{bi}$ et
– $\forall i, v_{ai} \preccurlyeq^{A_i}_{t_a} v_a$ et $v_{bi} \preccurlyeq^{A_i}_{t_b} v_b$.

$T = T_a, T_b$, donc les domaines de T_a et de T_b sont disjoints. Soient $R' = R'_a, R'_b$, $D = D_a$ `and` $D_b = D_1$ `and` $D_2 \ldots$ `and` D_n et $D' = D'_a$ `and` D'_b. On a alors $R_p \vdash D \xrightarrow{R'} D'$ et $\mathrm{dom}(R') = \mathrm{dom}(T)$.

Soient pour tout i, $v_i = (v_{ai}, v_{bi})$ et $e'_i = (e'_{ai}, e'_{bi})$. Alors, $R_p, R' \vdash e_i \xrightarrow{(v_{ai}, v_{bi})} e'_i$ et $(v_{ai}, v_{bi}) \preccurlyeq^{A_i}_t (v_a, v_b)$.

Cas de la règle Op-C : Soit $e = \mathrm{op}(e_a)$.
D'après la règle Op-C, $H|G \vdash e : t/\ell/T$ ssi il existe ℓ', s, et t_1 tels que $\ell = \ell' \cup \{s\}$, $H|G \vdash \mathrm{op} : t_1 \prec\!\{s\}\!\succ t/\ell'/\emptyset$ et $H|G \vdash e_a : t_1/\ell'/T$. D'autre part, d'après la définition de H_0, il existe tc et tc_1 tels que $t = tc$ `at` s et $t_1 = tc_1$ `at` s.
D'après le lemme 7, pour tout i, il existe e_{ai} et D_i tels que $H|G \vdash e_a : t_1/\ell'/T \overset{A_i}{\Longrightarrow} e_{ai}/D_i$.
D'après la règle Op de la sémantique centralisée, $R \vdash e \xrightarrow{v} e'$ ssi il existe v, v_a, op' et e'_a tels que $e' = \mathrm{op}'(e'_a)$, $R \vdash e_a \xrightarrow{v_a} e'_a$, et $\mathrm{op}(v_a) \xrightarrow{v} \mathrm{op}'$.
Soit $D = D_1$ `and` $\ldots$ `and` D_n.
Par hypothèse d'induction, il existe $D', D'_1, \ldots, D'_n$ tels que $R_p \vdash D \xrightarrow{R'} D'$, $D' = D'_1$ `and` $\ldots$ `and` D'_n, et pour tout i, il existe v_{ai} et e'_{ai} tels que $R_p, R' \vdash e_{ai} \xrightarrow{v_{ai}} e'_{ai}$ avec $v_{ai} \preccurlyeq^{A_i}_{t_1} v_a$, et $H|G \vdash e'_a : t/\ell/T \overset{A_i}{\Longrightarrow} e'_{ai}/D'_i$.
Soit $i \in \{1, \ldots, n\}$.

– Si $A_i = s$, étant donné que $v_{ai} \preccurlyeq^{A_i}_{tc_1 \texttt{ at } s} v_a$, alors $v_{ai} = v_a$, et par application de la règle Op-P,
$$H|G \vdash \mathrm{op}(e_a) : t/\ell/T \overset{A_i}{\Longrightarrow} \mathrm{op}(e_{ai})/D_i.$$
Par application de la règle Op de la sémantique, $R_p, R' \vdash \mathrm{op}(e_{ai}) \xrightarrow{v} \mathrm{op}'(e'_{ai})$ avec $v \preccurlyeq^{A_i}_t v$.
De plus, on a :
$$H|G \vdash \mathrm{op}'(e'_a) : t/\ell/T \overset{A_i}{\Longrightarrow} \mathrm{op}'(e'_{ai})/D_i.$$
– Sinon, $v_i = \bot$. Par application de la règle Op-P-Suppr,
$$H|G \vdash \mathrm{op}(e_a) : t/\ell/T \overset{A_i}{\Longrightarrow} \bot/D_i.$$

Cas de la règle At-C : Induction directe.

Cas de la règle Comm-C : L'expression e est bien typée ssi il existe tc, A, A', T', c et ℓ' tels que l'architecture autorise les communications de A vers A', $t = tc$ `at` A', $\ell = \ell' \cup \{A'\}$, $T = T', [A \overset{c}{\mapsto} A']$, et $H|G \vdash e : tc$ `at` $A/\ell'/T'$.

D'après le lemme 7, pour tout i, il existe une expression e_i et une équation D_i telles que $H|G \vdash e : tc \texttt{ at } A/\ell'/T' \overset{A_i}{\Longrightarrow} e_i/D_i$.
Soit $D_p = D_1 \texttt{ and } D_2 \ldots \texttt{ and } D_n$.
Par hypothèse d'induction, $R_p \vdash D_p \xrightarrow{R'_p} D'_p$, et pour tout i, $R_p, R'_p \vdash e_i \xrightarrow{v_i} e'_i$ avec $v_i \preccurlyeq^{A_i}_{tc \texttt{ at } A} v$.
Soit j tel que $A_j = A$.
Pour tout $i \in \{1, \ldots, n\}$, soit D'_i tel que $D'_i = (D_i \texttt{ and } (c = e_i))$ si $i = j$, $D'_i = D_i$ sinon.
Soit $D = (D'_1 \texttt{ and } D'_2 \ldots \texttt{ and } D'_n) = (D_p \texttt{ and } (c = e_j))$. Soit $R' = R'_p, [v_j/c]$.
Alors, $R_p \vdash D \xrightarrow{R'} D'$, et $\mathrm{dom}(R') = \mathrm{dom}(R'_p) \cup \{c\} = \mathrm{dom}(T') \cup \{c\} = \mathrm{dom}(T)$.
Soit $i \in \{1, \ldots, n\}$.

- Si $A_i = A$, par application de la règle Comm-P-From de l'opération de projection, $H|G \vdash e : t/\ell/T \overset{A_i}{\Longrightarrow} \bot/D'_i$, avec $D'_i = D_i \texttt{ and } c = e_i$. Alors, puisque $A_i \neq A'$, $\bot \preccurlyeq^{A_i}_{t' \texttt{ at } A'} v$.
- Si $A_i = A'$, par application de la règle Comm-P-To, $H|G \vdash e : t/\ell/T \overset{A_i}{\Longrightarrow} c/D'_i$, avec $D'_i = D_i$. D'après la définition de R', $R_p, R' \vdash c \xrightarrow{v_j} c$, avec $v_j \preccurlyeq^{A_i}_{tc \texttt{ at } A'} v$, puisque d'après $v_j \preccurlyeq^{A_j}_{tc \texttt{ at } A} v$, $v_j = v$.
- Sinon, par application de la règle Comm-P-Suppr, $H|G \vdash e : t/\ell/T \overset{A_i}{\Longrightarrow} \bot/D'_i$ avec $D'_i = D_i$ et $\bot \preccurlyeq^{A_i}_{t} v$.

□

Le théorème 4 établit la correction de la projection des équations. La projection d'un ensemble d'équations D définit un nouveau programme $D_1 \texttt{ and } D_2 \ldots \texttt{ and } D_n$, dont la sémantique est équivalente à celle de D, en prenant en compte les types spatiaux des variables définies par D.

Théorème 4. *Pour tous H, H', D, D', ℓ, T, D_i, R et R', si*

- $R \vdash D \xrightarrow{R'} D'$ *et*
- $\forall i, H|G \vdash D : H'/\ell/T \overset{A_i}{\Longrightarrow} D_i$,

*alors pour tout environnement de réaction R_p compatible avec R selon H, soit $D_p = D_1$ **and** $D_2 \ldots$ **and** D_n, il existe R'_p et D'_p tels que :*

- $R_p \vdash D_p \xrightarrow{R'_p} D'_p$,
- $R'_p \preccurlyeq_{H'} R'$, *et*
- $\forall i, H|G \vdash D' : H'/\ell/T \overset{A_i}{\Longrightarrow} D'_i$

Démonstration. Par induction sur la structure de D.

Soient H, H', D, D', ℓ, T, D_i, R et R' tels que $R \vdash D \xrightarrow{R'} D'$ et pour tout i, $H|G \vdash D : H'/\ell/T \overset{A_i}{\Longrightarrow} D_i$.

Soit R_p tel que $R_p \preccurlyeq_H R$.

Cas $D = ((x_1, \ldots, x_n) = e)$**.** D'après la règle DEF de la sémantique, $R \vdash D \xrightarrow{R'} D'$ ssi il existe $e', v_1, \ldots, v_n$, $D' = ((x_1, \ldots, x_n) = e')$, $R' = [v_1/x_1, \ldots, v_n/x_n]$ et $R \vdash e \xrightarrow{(v_1,\ldots,v_n)} e'$.

L'équation D est bien typée ssi il existe $t = t_1 \times \ldots \times t_n$ tel que $H' = [t_1/x_1, \ldots, t_n/x_n]$ et $H|G \vdash e : t/\ell/T$.

D'après le lemme 7, pour tout i, il existe e_i et D'_i t.q. $H|G \vdash e : t/\ell/T \overset{A_i}{\Longrightarrow} e_i/D'_i$.

Soit $D_e = D'_1$ `and` $D'_2 \ldots$ `and` D'_n. Par application du lemme 8, il existe $v_{1i}, \ldots, v_{ni}$, e'_i, D_e, D'_e et R_e tels que $R_p \vdash D_e \xrightarrow{R_e} D'_e$, $R_p, R_e \vdash e_i \xrightarrow{(v_{1i},\ldots,v_{ni})} e'_i$ et $\forall j, v_{ji} \preccurlyeq^{A_i}_{t_j} v_j$.

Pour tout i, $D_i = ((x_{1A_i}, \ldots, x_{nA_i}) = e_i)$ `and` D'_i si $e_i \neq \bot$, $D_i = D'_i$ sinon. Soit $D_p = D_1$ `and` $\ldots$ `and` D_n et R'_p tel que $R_p \vdash D_p \xrightarrow{R'_p} D'_p$. D'après la définition des équations D_i, il existe R'' tel que $R'_p = R_e, R''$. Donc, si $x_{jA_i} \in \mathrm{dom}(R'_p)$, alors $R'_p(x_{jA_i}) = v_{ji}$.

D'après le lemme 6, $e_i = \bot$ ssi $A_i \notin \mathrm{locations}(t)$. Donc, pour tout i,

- Si $e_i = \bot$, alors :
  - la règle DEF-P-SUPPR s'applique :
    $H|G \vdash (x_1, \ldots, x_n) = e : H'/\ell/T \overset{A_i}{\Longrightarrow} D'_i$, et
  - $\forall j, x_{jA_i} \notin \mathrm{dom}(R'_p)$.
- Sinon :
  - la règle DEF-P s'applique :
    $H|G \vdash (x_1, \ldots, x_n) = e : H'/\ell/T \overset{A_i}{\Longrightarrow} ((x_{1A_i}, \ldots, x_{nA_i}) = e_i)$ `and` D'_i, et
  - $\forall j, R'_p(x_{jA_i}) = v_{ji}$, donc $R'_p(x_{jA_i}) \preccurlyeq^{A_i}_{t_j} R'(x_j)$.

Donc dans tous les cas $R'_p \preccurlyeq_H R'$.

Cas $D = (x = f(e))$**.** D'après la règle APP de la sémantique, $R \vdash D \xrightarrow{R'} D'$ ssi il existe e_f et D_f tels que $R(f) = \lambda y.e_f$ `where` D_f et $R \vdash$ `let` $y = e$ `and` D_f `in` $x = e_f \xrightarrow{R'} D'$. D'après les règles LET, AND et DEF de la sémantique, l'équation `let` $y = e$ `and` D_f `in` $x = e_f$ réagit ssi il existe e', v, R_f, D'_f, e'_f et v_f tels que :

- $R, [v/y] \vdash D_f \xrightarrow{R_f} D'_f$,
- $R, R_f \vdash e \xrightarrow{v} e'$,
- $R, R_f, [v/y] \vdash e_f \xrightarrow{v_f} e'_f$,
- $D' =$ `let` $y = e'$ `and` D'_f `in` $x = e'_f$ et
- $R' = [v_f/x]$.

L'équation D est bien typée ssi il existe t_1, t_2, ℓ_1, ℓ_2, ℓ_3, T_1, T'_1, T_2 et T_3 tels que :

- $\ell = \ell_1 \cup \ell_2 \cup \ell_3$,
- $T = T'_1, T_2, T_3$,

– $T_1 \cong T'_1$,
– $H|G \vdash f : t_1 \prec\!\langle \ell_1/T_1 \rangle\!\rightarrow t_2/\ell_2/T_2$,
– $H|G \vdash e : t_1/\ell_3/T_3$ et
– $H' = [t_2/x]$.

D'après le lemme 7, pour tout i, il existe f_i, e_i, D_{fi} et D'_i tels que :
– $H|G \vdash f : t_1 \prec\!\langle \ell_1/T_1 \rangle\!\rightarrow t_2/\ell_2/T_2 \overset{A_i}{\Longrightarrow} f_i/D_{fi}$, d'autre part, d'après les règles Inst-P et Inst-P-Suppr, $D_{fi} = \epsilon$, et $f_i = f_{A_i}$ ou $f_i = \perp$.
– $H|G \vdash e : t_1/\ell_3/T_3 \overset{A_i}{\Longrightarrow} e_i/D'_i$.

Soit $D_e = D'_1$ `and` $D'_2 \ldots$ `and` D'_n. Par application du lemme 8, il existe v_i, e'_i, D'_e et R_e tels que $R_p \vdash D_e \xrightarrow{R_e} D'_e$, $R_p, R_e \vdash e_i \xrightarrow{v_i} e'_i$ et $v_i \preccurlyeq^{A_i}_{t_1} v$.

Soit c_{ij}, c'_{ij}, $\hat{c}_{ij}$ et $\hat{c}'_{ij}$ tels que pour tout i :
– $T'_1 \uparrow A_i = [A_i \overset{c_{i1}}{\mapsto} A_{i1}, \ldots, A_i \overset{c_{ip_i}}{\mapsto} A_{ip_i}]$,
– $T_1 \uparrow A_i = [A_i \overset{\hat{c}_{i1}}{\mapsto} A_{i1}, \ldots, A_i \overset{\hat{c}_{ip_i}}{\mapsto} A_{ip_i}]$,
– $T'_1 \downarrow A_i = [A_{i1} \overset{c'_{i1}}{\mapsto} A_i, \ldots, A_{iq_i} \overset{c'_{iq_i}}{\mapsto} A_i]$ et
– $T_1 \downarrow A_i = [A_{i1} \overset{\hat{c}'_{i1}}{\mapsto} A_i, \ldots, A_{iq_i} \overset{\hat{c}'_{iq_i}}{\mapsto} A_i]$.

Soit pour tout i, D_i tel que $H|G \vdash D : H'/\ell/T \overset{A_i}{\Longrightarrow} D_i$:

$$\begin{cases} D_i = D'_i & \text{si } A_i \notin \ell_1 \\ D_i = (x_{A_i}, c_{i1}, \ldots, c_{ip_i}) = f_{A_i}(e_i, c'_{i1}, \ldots, c'_{iq_i}) \texttt{ and } D'_i & \text{sinon.} \end{cases}$$

Soit $D_p = D_1$ `and` $\ldots$ `and` D_n.

Considérons le cas d'application d'un nœud dupliqué sur tous les sites :

$$H(f) = \forall \alpha_1, \ldots, \alpha_n . \forall \delta . tc_1 \texttt{ at } \delta \prec\!\langle \{\delta\}/\emptyset \rangle\!\rightarrow tc_2 \texttt{ at } \delta.$$

Dans ce cas, $t_2 = tc_2$ `at` A_j, $\ell_1 = \{A_j\}$ et $T_1 = T'_1 = \emptyset$. D'après la règle Node-P-Gen, on a $R_p(f_{A_j}) = R(f) = \lambda y.e_f$ `where` D_f.

On a donc D_p de la forme :

$$\begin{array}{rll} D_p = & & D'_1 \\ & \texttt{and} & D'_2 \\ & & \vdots \\ & \texttt{and} & x_{A_j} = f_{A_j}(e_j) \texttt{ and } D'_j \\ & & \vdots \\ & \texttt{and} & D'_n \end{array}$$

Soit $R'_p = R_e, [v_f/x_{A_j}]$. On a $R_p \vdash D_p \xrightarrow{R'_p} D'_p$ et pour tout $i \in \{1, \ldots, n\}$:
– si $i = j$, alors $x_{A_i} \in \mathrm{dom}(R'_p)$, $R'_p(x_{A_i}) = v_f$ et $v_f \preccurlyeq^{A_i}_{tc_2 \texttt{ at } A_i} v_f$.

– Sinon, $x_{A_i} \notin \mathrm{dom}(R'_p)$ et $\bot \preccurlyeq^{A_i}_{tc_2 \texttt{ at } A_j} v_f$.

Donc, $R'_p \preccurlyeq_H R'$.

Considérons maintenant le cas :

$$H(f) = \forall \alpha_1, \ldots, \alpha_n . t_1 \prec\!\langle \ell_1 / T_1 \rangle\!\succ t_2.$$

D'après les règles Node-P et Node-P-Suppr, il existe ℓ_{a1}, ℓ_{b1}, T_{a1} et T_{b1} tels que $\ell_1 = \ell_{a1} \cup \ell_{b1}$, $T_1 = T_{a1}, T_{b1}$ et

$$\begin{cases} f_{A_i} \notin \mathrm{dom}(R_p) & \text{si } A_i \notin \ell_1 \\ R_p(f_{A_i}) = \lambda(y_A, \hat{c}'_{i1}, \ldots, \hat{c}'_{iq_i}).(e_{fi}, \hat{c}_{i1}, \ldots, \hat{c}_{ip_i}) \texttt{ where } D_{fi} & \text{sinon.} \\ \qquad \text{où} \quad H|G \vdash D_f : H_f/\ell_{a1}/T_{a1} \stackrel{A_i}{\Longrightarrow} D_{ai}, & \\ \qquad\qquad H, y : t_1, H_f|G \vdash e_f : t_2/\ell_{b1}/T_{b1} \stackrel{A_i}{\Longrightarrow} e_{fi}/D_{bi}, & \\ \qquad \text{et} \quad D_{fi} = D_{ai} \texttt{ and } D_{bi} & \end{cases}$$

D'après le lemme 8, et par hypothèse d'induction, pour tout i tel que $A_i \in \ell_1$, on a $R_p \vdash (D_{f1} \texttt{ and } \ldots \texttt{ and } D_{fn}) \xrightarrow{R_{fp}} D'_{fp}$, $R_p, R_{fp} \vdash e_{fi} \xrightarrow{v_{fi}} e'_{fi}$ et $v_{fi} \preccurlyeq^{A_i}_{t_2} v_f$.

On a D_p de la forme :

$$D_p = \begin{array}{lll} & \vdots & \\ \texttt{and} & D'_i & \text{si } A_i \notin \ell_1 \\ & \vdots & \\ \texttt{and} & D'_j \texttt{ and } (x_{A_j}, c_{j1}, \ldots, c_{jp_j}) = f_{A_j}(e_j, c'_{j1}, \ldots, c'_{jq_j}) & \text{si } A_j \in \ell_1 \\ & \vdots & \end{array}$$

D'après la règle App de la sémantique centralisée, les équations D_p réagissent dans R_p en émettant R'_p ssi les équations suivantes réagissent dans R_p en émettant R'_p :

$$\begin{array}{ll} & \vdots \\ \texttt{and} & D'_i \\ & \vdots \\ \texttt{and} & D'_j \texttt{ and } \texttt{let } (y_{A_j}, \hat{c}'_{j1}, \ldots, \hat{c}'_{jq_j}) = (e_j, c'_{j1}, \ldots, c'_{jq_j}) \texttt{ and } D_{fj} \texttt{ in} \\ & \qquad (x_{A_j}, c_{j1}, \ldots, c_{jp_j}) = (e_{fj}, \hat{c}_{j1}, \ldots, \hat{c}_{jp_j}) \\ & \vdots \end{array}$$

Soit R'_p tel que :

$$R'_p = R_e, [(v_{fi}/x_{A_i})_{i \in \{1,\ldots,n\}}], [(v_{ij}/c_{ij})_{i \in \{1,\ldots,n\}, j \in \{1,\ldots,p_i\}}],$$

où $\forall i, j, v_{ij} = R_{fp}(\hat{c}_{ij})$.

Soit $i \in \{1, \ldots, n\}$.

– Si $A_i \notin \ell_1$, alors $x_{A_i} \notin \mathrm{dom}(R'_p)$, et $\perp \preccurlyeq_{t_2}^{A_i} v_f$.
– Sinon, soit $R_i = R_{1i}, R_{2i}$ tel que $R_{1i} = [v_i/y_{A_i}, (v_{ij}/\hat{c}'_{ij})_{j\in\{1,\ldots,p_i\}}]$ où $v_{ij} = R_{fp}(\hat{c}'_{ij})$, et $R_{2i} = [R_{fp}(x_{A_ij})/x_{A_ij}], [(v_{ij}/\hat{c}_{ij})_{j\in\{1,\ldots,p_i\}}]$ où $v_{ij} = R_{fp}(\hat{c}_{ij})$ et $\{x_{A_i1}, \ldots, x_{A_in}\} = \{x_{A_i} | x_{A_i} \in \mathrm{dom}(R_{fp})\}$. Alors il existe D_{yi} tel que :

$$R_p, R'_p, R_{2i} \vdash (y_{A_i}, \hat{c}'_{i1}, \ldots, \hat{c}'_{ip_i}) = (e', c'_{i1}, \ldots, c'_{ip_i}) \xrightarrow{R_{pi}} D_{yi}$$

$$R_p, R'_p, R_{1i} \vdash D_{fi} \xrightarrow{R_{2i}} D'_{fi}$$

D'après la définition de R_{1i} et R_{2i}, toutes les variables instanciées dans e_{fi} sont dans le domaine de R_i. D'autre part, pour tout x du domaine de R_i, on a $R_i(x) = R_{fp}(x)$. Donc, $R_p, R'_p, R_i \vdash e_{fi} \xrightarrow{v_{fi}} e'_{fi}$, et

$$R_p, R'_p, R_i \vdash (x_{A_i}, c_{i1}, \ldots, c_{ip_i}) = (e_{fi}, \hat{c}_{i1}, \ldots, \hat{c}_{ip_i}) \xrightarrow{[v_{fi}/x_{A_i}, (v_{ij}/c_{ij})_{j\in\{1,\ldots,p_i\}}]} D_{xi}$$

On a alors $x_{A_i} \in \mathrm{dom}(R'_p)$, avec $R'_p(x_{A_i}) \preccurlyeq_{t_2}^{A_i} v_f$.

Donc, $R'_p \preccurlyeq_H R'$.

Cas $D = D_1$ `and` D_2 **et** $D =$ `let` D_1 `in` D_2 **:** Induction directe.

□

6.5 Application au canal de radio logicielle

Nous reprenons, à titre d'exemple, le canal de radio logicielle présenté en section 2.5. En l'absence de structure de contrôle, on suppose les deux canaux de réception exécutés en parallèle : la figure 6.10 montre le programme considéré.

Le nœud `channel` prend en argument deux nœuds centralisés, un exécutable sur le site `FPGA`, un sur `DSP`. L'entrée `x` doit être disponible sur `FPGA`, la sortie est émise sur `DSP`. Ce nœud comprend une communication de `FPGA` à `DSP` (la valeur filtrée `f`). Le type spatial de `channel` sera donc :

$$\begin{array}{l}\texttt{channel} : \forall \alpha, \beta, \gamma. \Big(\quad (\alpha \texttt{ at FPGA} \prec\!\langle \{\texttt{FPGA}\}/\emptyset \rangle\!\rightarrow \beta \texttt{ at FPGA}) \\ \qquad\qquad\qquad \times \ (\beta \texttt{ at DSP} \prec\!\langle \{\texttt{DSP}\}/\emptyset \rangle\!\rightarrow \gamma \texttt{ at DSP}) \\ \qquad\qquad\qquad \times \ \alpha \texttt{ at FPGA} \qquad\qquad\qquad\qquad \Big) \\ \qquad\qquad\qquad\qquad\qquad \prec\!\langle \{\texttt{FPGA}, \texttt{DSP}\}/[\texttt{FPGA} \overset{c}{\mapsto} \texttt{DSP}] \rangle\!\rightarrow \gamma \texttt{ at DSP}\end{array}$$

Supposons maintenant que les nœuds de filtrage et de démodulation aient reçu

```
loc FPGA
loc DSP
link FPGA to DSP

node channel(filter,demod,x) = y where
      f = filter(x) at FPGA
  and y = demod(f) at DSP

node multichannel_sdr(x1,x2) = (y1,y2) where
      y1 = channel(filter_1800,demod_gmsk,x1)
  and y2 = channel(filter_2000,demod_qpsk,x2)
```

FIG. 6.10 – Deux canaux de réception de radio logicielle en parallèle.

les types spatiaux suivants :

$$\begin{array}{ll}
\texttt{filter_1800} & : b \texttt{ at FPGA} \prec\!\{\texttt{FPGA}\}/\emptyset\!\succ\!\rightarrow b \texttt{ at FPGA} \\
\texttt{filter_2000} & : b \texttt{ at FPGA} \prec\!\{\texttt{FPGA}\}/\emptyset\!\succ\!\rightarrow b \texttt{ at FPGA} \\
\texttt{demod_gmsk} & : b \texttt{ at DSP} \prec\!\{\texttt{DSP}\}/\emptyset\!\succ\!\rightarrow b \texttt{ at DSP} \\
\texttt{demod_qpsk} & : b \texttt{ at DSP} \prec\!\{\texttt{DSP}\}/\emptyset\!\succ\!\rightarrow b \texttt{ at DSP}
\end{array}$$

Le nœud `multichannel_sdr` prend deux entrées disponibles sur le site `FPGA`, et émet ses deux sorties sur le site `DSP`. Ce nœud comporte deux communications de `FPGA` à `DSP` : une communication par instance du nœud `channel`. Son type spatial sera donc :

$$\begin{array}{l}
\texttt{multichannel_sdr} : (b \texttt{ at FPGA} \times b \texttt{ at FPGA}) \\
\qquad \prec\!\{\texttt{FPGA}, \texttt{DSP}\}/[\texttt{FPGA} \overset{c_1}{\mapsto} \texttt{DSP}, \texttt{FPGA} \overset{c_2}{\mapsto} \texttt{DSP}]\!\succ\!\rightarrow (b \texttt{ at DSP} \times b \texttt{ at DSP})
\end{array}$$

Le résultat de la projection de ces deux nœuds est donné en figures 6.11 et 6.12.

La figure 6.11 montre la projection sur le site `FPGA`. L'application du nœud `demod` est supprimée du corps du nœud `channel`, et une sortie additionnelle `c` est ajoutée, définie par la valeur issue de l'application du nœud `filter`. Deux sorties additionnelles `c1` et `c2` sont par conséquent ajoutées au nœud `multichannel_sdr`.

La figure 6.12 montre la projection sur le site `DSP`. L'application du nœud `filter` est supprimée du corps de `channel`, et les canaux de communication sont cette fois ajoutés en entrée des deux nœuds.

6.6 Répartition sur des systèmes GALS

L'opération de projection présentée dans ce chapitre permet d'obtenir, à partir d'un programme synchrone, n programmes synchrones exprimés dans le même

```
node channel(filter,demod,x) = (⊥,c) where
      c = filter(x)

node multichannel_sdr(x1,x2) = (⊥,⊥,c1,c2) where
      (y1,c1) = channel(filter_1800,⊥,x)
  and (y2,c2) = channel(filter_2000,⊥,x)
```

FIG. 6.11 – Résultat de la projection sur le site FPGA.

```
node channel(filter,demod,x,c) = y where
      f = c
  and y = demod(f)

node multichannel_sdr(x1,x2,c1,c2) = (y1,y2) where
      y1 = channel(⊥,demod_gmsk,⊥,c1)
  and y2 = channel(⊥,demod_qpsk,⊥,c2)
```

FIG. 6.12 – Résultat de la projection sur le site DSP.

langage que le programme source. Ces programmes synchrones sont destinés à être exécutés sur chacun des sites de l'architecture. La correction de cette opération de projection prouve que le produit synchrone de ces n programmes est sémantiquement équivalent au programme source. Cette opération donne donc un cadre général pour la répartition, indépendant de la méthode de compilation utilisée (code séquentiel, circuit) et du type de communications (tant que celles-ci préservent l'ordre des messages : communications par rendez-vous, par tampon de taille bornée, par exemple).

En effet, le cadre donné par cette opération de répartition permet de donner une sémantique de Kahn au programme réparti [54]. On peut donc effectuer sur ce programme toute opération de compilation ou de transformation conservatrice du point de vue de cette sémantique. En particulier, l'insertion de tampons de taille bornée aux points de communications permet d'obtenir des systèmes GALS. Cette insertion conserve la sémantique de Kahn, et permet donc d'obtenir un système sémantiquement équivalent au programme source.

Cette opération de projection peut aussi aisément être reportée après la compilation sous forme de code séquentiel. La définition et l'utilisation de canaux de communications peut alors prendre la forme d'envoi et de réception de valeurs. L'analyse de causalité étant effectuée à la compilation, le programme ainsi compilé et réparti ne contient aucun interblocage. Cette méthode correspond alors à celle implantée dans ocrep [21], opérant sur des programmes réactifs compilés

sous la forme de code séquentiel. La preuve de correction décrite dans [16] permet d'assurer la correction du système réparti en utilisant cette méthode.

À titre d'illustration, le programme de la figure 6.10 peut être compilé sous la forme de fonctions séquentielles décrites par le pseudo-code ci-dessous :

```
let channel(filter,demod,x) =
  f <- filter(x);
  y <- demod(f);
  return y

let multichannel_sdr(x1,x2) =
  y1 <- channel(filter_1800,demod_gmsk,x1);
  y2 <- channel(filter_2000,demod_qpsk,x2);
  return (y1,y2)
```

Cette compilation sous forme de code séquentiel n'est pas aussi directe en général : en effet, la compilation d'équations récursives comprenant l'accès à des valeurs en mémoire implique une analyse de causalité permettant de construire un graphe acyclique de dépendances entre les variables à calculer, et le code séquentiel produit doit comporter la mise à jour de l'état représenté par les valeurs mises en mémoire et accédées à l'état suivant.

À partir du code séquentiel obtenu, l'opération de projection peut être appliquée, en remplaçant les définitions de canaux de communications par une opération `send`, prenant en argument un site de destination et la valeur à envoyer. L'utilisation de ces mêmes canaux est compilée sous la forme d'une opération `recv`, prenant en argument le site source d'où la valeur est reçue. Ces deux opérations peuvent être par ailleurs généralisées, en distinguant non pas les sites sources et destination, mais les canaux utilisés, afin de permettre, entre deux mêmes sites, plusieurs techniques de communication, selon les valeurs communiquées à différents points du programme. Dans ce cas, les canaux de communications deviennent systématiquement des entrées des fonctions séquentielles, et deviennent les arguments des opérations d'envoi et de réception.

Les figures 6.13 et 6.14 montrent le code séquentiel réparti respectivement sur les sites `FPGA` et `DSP`.

Ce code conserve la modularité fonctionnelle initiale, et la résolution de dépendances entre les équations sous forme d'un ordre total d'exécution étant effectuée avant la répartition, chaque opération `send` correspond sur son site de destination à une opération `recv`. Ainsi, sur les deux sites, la fonction séquentielle `channel` est appelée deux fois en séquence. L'analyse de causalité a placé, sur `multichannel_sdr`, la définition de `y1` avant celle de `y2`. La séquence effectuée sur `FPGA` et sur `DSP` est la même, par conséquent, l'opération `send` sur le canal `c1` est effectuée avant l'opération `send` sur le canal `c2`, de même que l'opération `rcv(c1)`

```
let channel(filter,demod,x,c) =
  f <- filter(x);
  send(c,f);
  return ⊥

let multichannel_sdr(x1,x2,c1,c2) =
  y1 <- channel(filter_1800,⊥,x,c1);
  y2 <- channel(filter_2000,⊥,x,c2);
  return (⊥,⊥)
```

FIG. 6.13 – Code séquentiel exécuté sur le site FPGA.

```
let channel(filter,demod,x,c) =
  f <- recv(c);
  y <- demod(f);
  return y

let multichannel_sdr(x1,x2,c1,c2) =
  y1 <- channel(⊥,demod_gmsk,⊥,c1);
  y2 <- channel(⊥,demod_qpsk,⊥,c2);
  return (y1,y2)
```

FIG. 6.14 – Code séquentiel exécuté sur le site DSP.

est effectuée avant `rcv(c2)`. Il n'y a donc pas d'interblocage, ce qui aurait été le cas si l'analyse de causalité n'avait pas été effectuée de manière centralisée : dans un tel cas, l'ordre d'évaluation des deux équations du nœud `multichannel_sdr` n'est pas déterminé a priori ; un interblocage est alors susceptible d'apparaître par exemple en échangeant les deux appels de `channel` sur un seul des deux sites.

6.7 Conclusion

Nous avons défini dans ce chapitre une méthode de répartition au moyen d'une opération de projection, dirigée par les types spatiaux définis au chapitre 5. Dans un premier temps, le système de types spatiaux a été étendu, afin que les effets comportent l'ensemble des canaux de communications nécessaires pour l'évaluation d'une expression ou d'un ensemble d'équations. L'opération de projection, à partir de ce système de types étendu, permet d'obtenir, à partir du programme initial typé, un fragment de programme par site de l'architecture. Ce fragment ne comporte que les calculs du programme initial à effectuer sur ce site, ainsi que la définition des canaux de communications sortants. Cette opération de projection est analogue à une opération de « slicing » [78]. Nous avons donné la preuve de la correction de cette opération de projection : la sémantique du programme réparti, définie par le produit synchrone des fragments obtenus par projection, est identique à la sémantique du programme initial.

La méthode de répartition présentée dans ce chapitre est modulaire, ce qui correspond à l'objectif initial de compilation modulaire dans un cadre réparti. De plus, cette méthode est indépendante de la méthode de compilation choisie, et de la technique réellement utilisée pour les communications entre les sites. Elle peut ainsi être appliquée avant ou après toute transformation du programme initial préservant la sémantique de Kahn. Elle peut aussi être appliquée sur le programme séquentiel issu de la compilation du programme synchrone. On obtient ainsi de manière automatique, à partir du programme source centralisé, un système GALS sémantiquement équivalent au programme centralisé décrit par le programmeur.

Chapitre 7

Horloges et introduction du contrôle

Le langage traité jusqu'ici ne comporte aucun contrôle, que ce soit sous forme d'horloge ou de structures de contrôle. Les programmes décrits dans un tel langage sont des graphes d'opérateurs, tous évalués à chaque instant. Dans le cadre synchrone, l'ajout du contrôle sous forme d'horloge permet de restreindre les instants où certaines opérations seront effectivement évaluées. L'utilisation d'horloges permet ainsi d'augmenter l'efficacité du programme à l'exécution, en réduisant le nombre d'opérations effectuées à chaque cycle. Dans un cadre réparti, l'utilisation d'horloges et de structures de contrôle pose un problème, dans la mesure où le contrôle doit rester cohérent d'un site à l'autre. Ce chapitre montre donc de quelle manière le langage et la méthode de répartition peuvent être étendus de manière sûre avec des structures de contrôle.

7.1 Horloges en Lucid Synchrone

Les horloges sont des flots booléens indiquant la présence ou l'absence de valeurs sur les flots. Un flot `x` d'horloge `c` portera une valeur aux instants où `c` est vrai, et sera absent aux autres instants. Il n'est pas nécessaire de décrire, au sein d'un programme, l'horloge de tous les flots : de même que pour les types de données, les horloges sont inférées à partir de celles décrites dans le programme. Ainsi, si un flot `x` est d'horloge α, l'horloge de l'expression `(x when c) + y` sera α `on` c, et le flot `y` devra être de même sur l'horloge α `on` c. Cette analyse est effectuée par une inférence de types appelée calcul d'horloges [30] : les horloges sont le type « temporel » des flots du programme, et l'analyse est modulaire, les nœuds étant « typés » par l'horloge de leurs entrées et de leurs sorties.

La méthode de répartition décrite jusqu'à présent est correcte du point de vue

des types de données : en effet, l'analyse de types est effectuée sur le programme initial, et est indépendante de la méthode de répartition. Cela garantit donc que le type d'une valeur portée par un canal de communication est le même sur le site de définition et d'utilisation de ce canal.

Cependant, le cas des horloges nécessite un peu plus d'attention : le calcul d'horloge comprend des types dépendants, c'est-à-dire des horloges définies par des valeurs dynamiques du programme. Si `x` est d'horloge α, on peut par exemple écrire le programme Lucid Synchrone suivant :

```
let clock pos = (x > 0) in
z = merge pos y 0
```

Ce programme définit une horloge `pos`, vraie aux instants où `x` est positif. Cette horloge est elle-même de la même horloge α que `x`. La construction `merge` définit ensuite un flot `z` d'horloge α et égal à `y` lorsque `x` est positif, 0 sinon. `y` est inféré d'horloge α `on pos`, et est donc calculé uniquement lorsque `x` est positif.

Ces types dépendants posent un problème pour la répartition : supposons par exemple que le calcul de `y` dépende de deux sites A et B :

```
let clock pos = (x > 0) in
let y2 = (y1 when pos) + 1 at A in
let y = y2 + 2 at B in
z = merge pos y 0
```

Le calcul d'horloge infère ici que `y2` a pour horloge α `on pos`. Les expressions `(y1 when pos) + 1` et `y2 + 2` ne sont calculées que lorsque l'horloge `pos` est vraie. Pour répartir ce programme, il est donc nécessaire que cette horloge calculée localement soit rendue disponible sur les sites A et B. Dans le cas général, le système de types spatial ne permet pas d'inférer cette communication de `pos` sur les sites où l'horloge doit être disponible : en effet, il faudrait que l'information de type spatial attachée à `y` contienne l'ensemble des sites nécessaires au calcul de `y` et de ses dépendances, ce qui rendrait la répartition dépendante du calcul d'horloge.

Afin de préserver l'indépendance entre le calcul d'horloges et la répartition, nous nous restreignons dans cette thèse à deux cas : le cas des horloges globales disponibles sur tous les sites, qui peuvent donc être traitées de manière indépendantes du calcul d'horloge, et le cas des horloges locales qui seront restreintes à la construction `match/with`. Ces deux cas permettent de traiter de problèmes complémentaires : les horloges globales permettent de définir, sur un site donné, un contrôle indépendant des valeurs présentes sur les autres sites, et ne nécessitant donc aucune communication entre sites. La construction `match/with`, au contraire, définit une structure de contrôle dépendante des communications entre les sites impliqués dans le calcul de ses branches.

Remarquons que le programme ci-dessus est sémantiquement équivalent à :

```
match (x > 0) with
| true -> let y2 = y1 + 1 at A in
          let y = y2 + 2 at B in
          do z = y done
| false -> do z = 0 done
end
```

Dans cet exemple, le `match` définit implicitement une horloge à partir de l'expression `(x>0)`, et échantillonne de manière implicite les variables libres sur cette horloge (ici, `y1`). Un `merge` implicite est ensuite effectué sur le flot `z`. Cette restriction permet, à la répartition, de connaître l'ensemble des sites sur lesquels la valeur de l'expression `(x>0)` doit être disponible : ce sont les sites impliqués dans le calcul de chaque composante du `match`, afin que chacun de ces sites connaisse la branche à exécuter. Dans notre exemple, ce sont les sites `A` et `B`.

7.2 Horloges globales

Pour déclarer des horloges globales, on introduit la construction `clock` $c = e$, au même niveau que les définitions de nœuds. Cette construction définit une horloge c par le flot booléen e. Ce flot booléen peut être instancié sur toute horloge α du programme, ce qui signifie que toute expression e' d'horloge α peut être échantillonnée par l'horloge c ; l'expression e' `when` c est d'horloge α `on` c.

Pour que l'analyse d'horloge et la répartition soient indépendantes, il est nécessaire qu'une expression définissant une horloge puisse être exécutée sur n'importe quel site. Dans ce cas, l'horloge ainsi définie est disponible sur tous les sites de l'architecture.

Prenons par exemple le programme de la figure 7.1, définissant une horloge globale `three` à partir d'un flot booléen vrai tous les 3 instants. Cette horloge est utilisée, dans l'exemple de radio logicielle, pour exécuter alternativement les deux canaux de réception, à partir de la même entrée, et définissant une seule sortie pour le système ; ainsi, le canal de réception GSM est exécuté deux fois plus souvent que le canal de réception UMTS.

Supposons, pour cet exemple, que les entrées et les sorties des fonctions de filtres et de démodulation sont sur la même horloge[1]. L'horloge de ces nœuds est donc $\forall\alpha.\alpha \rightarrow \alpha$. Aucune indication d'horloge n'est spécifiée pour le nœud `channel` : le calcul d'horloge infère l'horloge la plus générale, signifiant que ce nœud prend en

[1]En réalité, ce n'est pas le cas : la fonction de démodulation renvoie un symbole démodulé tous les SR/f_s échantillons du signal reçu, SR étant le taux de symboles et f_s la fréquence d'échantillonnage du signal.

```
loc FPGA
loc DSP
link FPGA to DSP

node sample(n) = ok where
      ok = (count = n)
  and count = 1 fby (if t then 1 else count + 1)

clock three = sample 3

node channel(filter,demod,x) = y where
      f = filter(x) at FPGA
  and y = demod(f) at DSP

node multichannel_sdr(x) = y where
      y1 = channel(filter_1800,demod_gmsk,(x whenot three))
  and y2 = channel(filter_2000,demod_qpsk,(x when three))
  and y = merge three y1 y2
```

FIG. 7.1 – Deux canaux de réception de radio logicielle en parallèle, contrôlés par une horloge.

entrée deux nœuds d'horloges $\alpha \to \beta$ et $\beta \to \gamma$, un flot sur l'horloge α, et la sortie est sur l'horloge γ :

$$\texttt{channel} : \forall \alpha, \beta, \gamma. \Big((\alpha \to \beta) \times (\beta \to \gamma) \times \alpha \Big) \to \gamma$$

Concernant le nœud `multichannel_sdr`, `x` est d'horloge α, les expressions `(x when three)` et `(x whenot three)` sont respectivement d'horloge α `on three` et α `on not three`. Les instances des fonctions de filtre et de démodulation reçoivent alors respectivement les horloges α `on three` $\to \alpha$ `on three` et α `on not three` $\to$ α `on not three`. `y1` est d'horloge α `on not three`, et `y2` d'horloge α `on three`.

Les figures 7.2 et 7.3 montrent le résultat obtenu de la répartition de ce programme. L'échantillonnage du flot `x` est effectué sur le site `FPGA` (figure 7.2), et la combinaison de `y1` et `y2` sur le site `DSP` (figure 7.3). La déclaration de l'horloge globale `three`, ainsi que la définition du nœud `sample`, sont dupliquées sur les deux sites.

La compilation des horloges, en Lucid Synchrone, consiste à générer, à partir des horloges des flots, des gardes booléennes sur les valeurs et les opérations. Ces gardes représentent les horloges et permettent, en fonction de leur valeur à l'exécution, de produire ou non les valeurs, ou d'effectuer ou non les opérations gardées.

```
node sample(n) = ok where
      ok = (count = n)
  and count = 1 fby (if t then 1 else count + 1)

clock three = sample 3

node channel(filter,demod,x) = (⊥,c) where
      c = filter(x)

node multichannel_sdr(x) = (⊥,c1,c2) where
      (y1,c1) = channel(filter_1800,⊥,(x whenot three))
  and (y2,c2) = channel(filter_2000,⊥,(x when three))
```

FIG. 7.2 – Projection du programme de la figure 7.1 sur le site FPGA.

```
node sample(n) = ok where
      ok = (count = n)
  and count = 1 fby (if t then 1 else count + 1)

clock three = sample 3

node channel(filter,demod,x,c) = y where
      f = c
  and y = demod(f)

node multichannel_sdr(x,c1,c2) = y where
      y1 = channel(⊥,demod_gmsk,⊥,c1)
  and y2 = channel(⊥,demod_qpsk,⊥,c2)
  and y = merge three y1 y2
```

FIG. 7.3 – Projection du programme de la figure 7.1 sur le site DSP.

L'inférence et la compilation des horloges étant indépendantes de la répartition, l'ajout de ces gardes peut être effectué sur chacun des programmes issus des projections, au sein desquels les informations issues du calcul d'horloges peuvent être transmises.

7.3 Structures de contrôle

7.3.1 Principe et exemples

Une construction `match/with` est ajoutée au langage. On donne à cette construction la même sémantique que celle exposée en section 2.4.1, à la différence qu'elle est ici restreinte aux valeurs booléennes pour plus de clarté (la généralisation aux types énumérés est directe). Dans le langage considéré, cette construction définit un ensemble d'équations, et s'écrit :

$$\begin{array}{l} \texttt{match}\ e\ \texttt{with} \\ \mid \texttt{true} \rightarrow D_1 \\ \mid \texttt{false} \rightarrow D_2 \end{array}$$

L'expression e est un flot booléen. Aux instants où e est vraie, seul l'ensemble d'équations D_1 réagit. Réciproquement, seul D_2 réagit aux instants où e est fausse. D_1 et D_2 définissent le même ensemble de variables.

Dans un cadre réparti, l'évaluation de D_1 et D_2 peut impliquer plusieurs sites. Dans ce cas, sur chacun de ces sites et à chaque instant, le choix doit être fait entre l'évaluation de la portion locale de D_1 ou de D_2. La structure de contrôle `match/with` doit donc être dupliquée sur ces sites, et la valeur de l'expression e doit y être rendue disponible par communication.

Considérons par exemple deux nœuds `f` et `g`, définis par `f(x)=f2(f1(x))` et `g(x)=g2(g1(x))`, `f1` et `g1` devant être exécutés sur A, et `f2` et `g2` sur B, l'architecture permettant les communications de A vers B.

```
node f(x) = z where
      y = f1(x) at A
  and z = f2(y) at B

node g(x) = z where
      y = g1(x) at A
  and z = g2(y) at B
```

On a vu au chapitre 6 que la projection de ces deux nœuds donne le code suivant :

A	B

node $f_A(x_A)$ = (⊥,c) where
  c = $f1_A(x_A)$
 	
node $g_A(x_A)$ = (⊥,c) where
  c = $g1_A(x_A)$
	node $f_B(x_B$,c) = y_B where
	  z_B = $f2_B$(c)
	
	node $g_B(x_B$,c) = y_B where
	  z_B = $g2_B$(c)

On veut maintenant exécuter ces deux nœuds de manière alternative avec un flot `x` en entrée, en fonction de la valeur d'un deuxième flot `y`.

```
match y with
| true -> z = f(x)
| false -> z = g(x)
```

Insistons tout d'abord sur le fait que ce programme a un comportement différent de `z = if y then f(x) else g(x)` : dans ce dernier cas, `f` et `g` sont évalués et progressent *à chaque instant*. L'opération `if/then/else` n'a donc pas à être dupliquée : il suffit de la placer sur le site où le flot `z` est défini.

Le cas du `match/with` est différent : `f(x)` n'est évaluée *qu'aux seuls instants* où `y` est vrai. Le programme réparti devra donc n'évaluer, sur A et sur B, les composantes de `f` qu'aux seuls instants où `y` est vrai. Cela suppose :

1. de dupliquer la structure `match/with` sur A et sur B ;
2. d'ajouter la communication de la valeur de `y` de A à B, afin d'évaluer, à chaque instant et sur chaque site, la même branche des deux structures `match/with` obtenues.

Le programme réparti utilisera donc trois canaux de communication : un pour chaque instance de `f` et `g`, et un pour `y`. Par ailleurs, dans chaque cas du `match`, il est nécessaire de compléter les équations projetées par la définition des canaux non utilisés avec la valeur absente :

A	B

c1 = y_A
and match y_A with
  | true -> (z_A,c2) = $f_A(x_A)$
  and c3 = ⊥
  | false -> (z_A,c3) = $g_A(x_A)$
  and c2 = ⊥
	match c1 with
	| true -> z_B = f_B(⊥,c2)
	| false -> z_B = g_B(⊥,c3)

Considérons maintenant un nœud centralisé `f`, disponible sur A et B, et que l'on veut exécuter alternativement sur A et sur B selon la valeur de `y` :

```
match y with
| true -> z = f(x) at A
| false -> z = f(x) at B
```

Les flots `x` et `y` seront localisés sur A, `z` sur B. Dans le cas `true` du `match`, `f` sera exécuté sur A, puis le résultat sera communiqué de A vers B pour la définition de `z`. Dans le cas `false`, la valeur de `x` sera communiquée de A vers B avant l'exécution de `f`. La projection de ce programme est donc :

A	B
c1 = y_A and match y_A with | true -> c2 = $f_A(x_A)$ and c3 = $\perp$ | false -> c3 = x_A and c2 = $\perp$	match c1 with | true -> z_B = c2 | false -> z_B = f_B(c3)

7.3.2 Application à la radio logicielle

Reprenons maintenant notre exemple de canaux de réception de radio logicielle. On veut maintenant exécuter les deux canaux de réception de manière exclusive. De plus, le canal de réception à exécuter est calculé avec un nœud `gsm_umts` à partir du symbole démodulé précédent, nœud qu'on suppose être localisé sur le site `DSP`. La figure 7.4 montre ce nouveau programme. Notons qu'il est nécessaire d'ajouter un lien de communication de `DSP` vers `FPGA`, afin de communiquer le symbole démodulé et de permettre le contrôle sur le site `FPGA`.

Les figures 7.5 et 7.6 montrent le résultat de la répartition de ce programme, respectivement sur `FPGA` et sur `DSP`. Le canal de communication `c1` porte la valeur de `gsm`, permettant de choisir le canal de réception à exécuter à chaque instant sur chacun des sites. `c2` et `c3` sont les canaux utilisés à l'intérieur de chacun de ces canaux de réception.

```
loc FPGA
loc DSP
link FPGA to DSP
link DSP to FPGA

node channel(filter,demod,x) = y where
      f = filter(x) at FPGA
  and y = demod(f) at DSP

node multichannel_sdr(x) = y where
      gsm = true fby (gsm_umts(y)) at DSP
  and match gsm with
      | true -> y = channel(filter_1800,demod_gmsk,x)
      | false -> y = channel(filter_2000,demod_qpsk,x)
```

FIG. 7.4 – Canaux de réception de radio logicielle contrôlés par une structure `match/with`.

7.4 Extension du langage

Afin d'incorporer les horloges globales et la structure `match/with`, le langage est étendu comme suit :

$$
\begin{array}{rcl}
d & ::= & \ldots \mid \texttt{clock}\ c = e \\
D & ::= & \ldots \mid \texttt{match}\ e\ \texttt{with} \\
 & & \quad \mid \texttt{true} \rightarrow D \\
 & & \quad \mid \texttt{false} \rightarrow D \\
e & ::= & \ldots \mid e\ \texttt{when}\ c \mid \texttt{merge}\ c\ e\ e
\end{array}
$$

Les définitions sont complétées avec les définitions d'horloges globales (`clock` $c = e$), les équations avec la construction `match/with` limitée aux expressions booléennes, et les expressions avec l'opération d'échantillonnage sur une horloge (e `when` c) et de combinaison de deux flots complémentaires (`merge` c e e). Les horloges c sont considérées comme d'un espace de nom différent des variables : elles ne peuvent pas être définies par des équations, ni utilisées comme expression en dehors des opérations d'échantillonnage et de combinaison.

7.5 Sémantique synchrone

Nous définissons maintenant la sémantique synchrone du langage étendu. Cette sémantique est définie par extension de la sémantique définie en section 2.4.3. Les

```
node channel(filter,demod,x) = (⊥,c) where
      c = filter(x)

node multichannel_sdr(x,c1) = (⊥,c2,c3) where
  match c1 with
  | true -> (y1,c2) = channel(filter_1800,⊥,x)
        and c3 = ⊥
  | false -> (y2,c3) = channel(filter_2000,⊥,x)
        and c2 = ⊥
```

FIG. 7.5 – Projection du programme de la figure 7.4 sur le site FPGA.

```
node channel(filter,demod,x,c) = y where
      f = c
  and y = demod(f)

node multichannel_sdr(x,c2,c3) = (y,c1) where
      gsm = true fby (gsm_umts(y))
  and c1 = gsm
  and match gsm with
      | true -> y = channel(⊥,demod_gmsk,⊥,c1)
      | false -> y = channel(⊥,demod_qpsk,⊥,c2)
```

FIG. 7.6 – Projection du programme de la figure 7.4 sur le site DSP.

valeurs considérées sont étendues aux valeurs absentes, notées [], et aux horloges ; une horloge c définie par `clock` $c = e$ est représentée par une valeur notée $\langle e \rangle$:

$$\begin{array}{lcl} v & ::= & [] \mid i \mid (v, v) \\ rv & ::= & v \mid \lambda x.e \texttt{ where } D \\ cv & ::= & \langle e \rangle \\ R & ::= & [rv_1/x_1, \ldots, rv_n/x_n, cv_1/c_1, \ldots, cv_p/c_p] \end{array}$$

De même que pour les nœuds, les horloges globales définies par `clock` $c_i = e_i$ définissent un environnement global $R_0 = [\langle e_1 \rangle / c_1, \ldots, \langle e_n \rangle / c_n]$.

On note `var` (D) l'ensemble des variables définies par les équations D :

$$\begin{array}{lcl} \texttt{var}\ ((x_1, \ldots, x_n) = e) & = & \{x_1, \ldots, x_n\} \\ \texttt{var}\ ((x_1, \ldots, x_n) = f(e)) & = & \{x_1, \ldots, x_n\} \\ \texttt{var}\ (D_1 \texttt{ and } D_2) & = & \texttt{var}\ (D_1) \cup \texttt{var}\ (D_2) \\ \texttt{var}\ (\texttt{let } D_1 \texttt{ in } D_2) & = & \texttt{var}\ (D_2) \\ \texttt{var} \begin{pmatrix} \texttt{match } e \texttt{ with} \\ \mid \texttt{true} \rightarrow D_1 \\ \mid \texttt{false} \rightarrow D_2 \end{pmatrix} & = & \texttt{var}\ (D_1) = \texttt{var}\ (D_2) \end{array}$$

Les règles de la figure 7.7 définissent le prédicat $R \vdash e \xrightarrow{v} e'$. Les règles suffixées par « -[] » permettent de gérer les valeurs absentes.

- Une valeur immédiate peut ne pas émettre de valeur dans l'instant considéré (règle Imm-[]).
- Un délai, ou une opération, ne peuvent être effectués que si les expressions en argument émettent toutes deux une valeur. Sinon, aucune de ces expressions n'émet de valeur, et le résultat du délai ou de l'opération est la valeur absente (règle Op-[]). Les opérations synchronisent donc leurs entrées, qui doivent être présentes aux mêmes instants. Le rôle du calcul d'horloge est de rejeter les programmes ne permettant pas cette synchronisation.

La figure 7.8 montre la sémantique des opérations d'échantillonnage et de combinaison.

- Les règles When-Init et Merge-Init initialisent les horloges avec leur valeur à partir de l'environnement.
- L'échantillonnage impose la synchronisation du flot à échantillonner et de l'horloge d'échantillonnage (règle When-[]).
- Si l'expression de l'horloge émet la valeur `true`, le flot échantillonné est présent et émet la valeur du flot en argument (règle When-True) ; sinon, le flot échantillonné est absent, mais le flot initial progresse (règle When-False).
- La combinaison de deux flots échantillonnés est absente si l'horloge est absente (règle Merge-[]).

$$\text{(Imm-[])}\quad R \vdash i \xrightarrow{[]} i \qquad \text{(Imm)}\quad R \vdash i \xrightarrow{i} i \qquad \text{(Inst)}\quad R, [v/x] \vdash x \xrightarrow{v} x$$

$$\text{(Op-[])}\quad \frac{R \vdash e \xrightarrow{[]} e'}{R \vdash \mathrm{op}(e) \xrightarrow{[]} \mathrm{op}(e')} \qquad \text{(Op)}\quad \frac{R \vdash e \xrightarrow{v} e' \qquad \mathrm{op}(v) \xrightarrow{v'} \mathrm{op}'}{R \vdash \mathrm{op}(e) \xrightarrow{v'} \mathrm{op}(e')}$$

$$\text{(Pair)}\quad \frac{R \vdash e_1 \xrightarrow{v_1} e_1' \qquad R \vdash e_2 \xrightarrow{v_2} e_2'}{R \vdash (e_1, e_2) \xrightarrow{(v_1,v_2)} (e_1', e_2')}$$

Fig. 7.7 – Sémantique avec horloges : règles pour les expressions.

– Les règles Merge-True et Merge-False vérifient que les deux flots combinés sont complémentaires, et prend la valeur du premier si l'horloge émet la valeur `true` (le deuxième flot étant absent), la valeur du deuxième sinon (le premier flot étant absent).

Les règles Def, App, And et Let pour les équations sont identiques à celles de la sémantique initiale (cf section 2.4.3).

La figure 7.9 donne les règles concernant le `match/with`. Si la valeur de l'expression conditionnelle est absente, alors aucune équation n'est évaluée, et l'équation `match/with` émet l'environnement donnant la valeur absente à toutes les variables définies dans le `match` (règle Match-[]). Si la valeur de l'expression conditionnelle est présente, selon sa valeur, la branche `true` ou `false` est évaluée (règles Match-True et Match-False).

Un calcul d'horloge peut ensuite être utilisé pour rejeter les programmes n'admettant pas d'exécution synchrone. Nous renvoyons le lecteur au travaux de Grégoire Hamon [47, 48], un tel calcul n'étant pas le sujet du travail présent. On considère ici uniquement l'ensemble des programmes admettant une sémantique synchrone.

L'extension de la sémantique répartie consiste en l'ajout des règles des figures 7.10 et 7.11.

La figure 7.10 concerne la sémantique répartie des opérations d'échantillonnage et de combinaison. Ces règles assurent que ces opérations sont effectuées sur des flots localisés sur le même site ; le résultat est alors sur ce site. Les autres règles sont identiques, en ajoutant les règles « -[] » pour l'absence de valeurs, annotées

(WHEN-INIT)

$$\frac{R(c) = \langle e_2 \rangle \qquad R \vdash e_1 \texttt{ when } \langle e_2 \rangle \xrightarrow{v} e'}{R \vdash e_1 \texttt{ when } c \xrightarrow{v} e'}$$

(WHEN-[])

$$\frac{R \vdash e_1 \xrightarrow{[]} e'_1 \qquad R \vdash e_2 \xrightarrow{[]} e'_2}{R \vdash e_1 \texttt{ when } \langle e_2 \rangle \xrightarrow{[]} e'_1 \texttt{ when } \langle e'_2 \rangle}$$

(WHEN-TRUE)

$$\frac{R \vdash e_1 \xrightarrow{v_1} e'_1 \qquad R \vdash e_2 \xrightarrow{\texttt{true}} e'_2}{R \vdash e_1 \texttt{ when } \langle e_2 \rangle \xrightarrow{v_1} e'_1 \texttt{ when } \langle e'_2 \rangle}$$

(WHEN-FALSE)

$$\frac{R \vdash e_1 \xrightarrow{v_1} e'_1 \qquad R \vdash e_2 \xrightarrow{\texttt{false}} e'_2}{R \vdash e_1 \texttt{ when } \langle e_2 \rangle \xrightarrow{[]} e'_1 \texttt{ when } \langle e'_2 \rangle}$$

(MERGE-INIT)

$$\frac{R(c) = \langle e_1 \rangle \qquad R \vdash \texttt{merge } \langle e_1 \rangle\ e_2\ e_3 \xrightarrow{v} e'}{R \vdash \texttt{merge } c\ e_2\ e_3 \xrightarrow{v} e'}$$

(MERGE-[])

$$\frac{R \vdash e_1 \xrightarrow{[]} e'_1 \qquad R \vdash e_2 \xrightarrow{[]} e'_2 \qquad R \vdash e_3 \xrightarrow{[]} e'_3}{R \vdash \texttt{merge } \langle e_1 \rangle\ e_2\ e_3 \xrightarrow{[]} \texttt{merge } \langle e'_1 \rangle\ e'_2\ e'_3}$$

(MERGE-TRUE)

$$\frac{R \vdash e_1 \xrightarrow{\texttt{true}} e'_1 \qquad R \vdash e_2 \xrightarrow{v_2} e'_2 \qquad R \vdash e_3 \xrightarrow{[]} e'_3}{R \vdash \texttt{merge } \langle e_1 \rangle\ e_2\ e_3 \xrightarrow{v_2} \texttt{merge } \langle e'_1 \rangle\ e'_2\ e'_3}$$

(MERGE-FALSE)

$$\frac{R \vdash e_1 \xrightarrow{\texttt{false}} e'_1 \qquad R \vdash e_2 \xrightarrow{[]} e'_2 \qquad R \vdash e_3 \xrightarrow{v_3} e'_3}{R \vdash \texttt{merge } \langle e_1 \rangle\ e_2\ e_3 \xrightarrow{v_3} \texttt{merge } \langle e'_1 \rangle\ e'_2\ e'_3}$$

FIG. 7.8 – Sémantique avec horloges : règles pour les opérations d'échantillonnage et de combinaison.

(MATCH-[])

$$\frac{R \vdash e \xrightarrow{[]} e' \qquad \texttt{var}\ (D_1) = \{x_1, \ldots, x_n\}}{R \vdash \begin{array}{l}\texttt{match}\ e\ \texttt{with}\\ \mid \texttt{true} \rightarrow D_1\\ \mid \texttt{false} \rightarrow D_2\end{array} \xrightarrow{[[]/x_1,\ldots,[]/x_n]} \begin{array}{l}\texttt{match}\ e'\ \texttt{with}\\ \mid \texttt{true} \rightarrow D_1\\ \mid \texttt{false} \rightarrow D_2\end{array}}$$

(MATCH-TRUE)

$$\frac{R \vdash e \xrightarrow{\texttt{true}} e' \qquad R \vdash D_1 \xrightarrow{R_1} D_1'}{R \vdash \begin{array}{l}\texttt{match}\ e\ \texttt{with}\\ \mid \texttt{true} \rightarrow D_1\\ \mid \texttt{false} \rightarrow D_2\end{array} \xrightarrow{R_1} \begin{array}{l}\texttt{match}\ e'\ \texttt{with}\\ \mid \texttt{true} \rightarrow D_1'\\ \mid \texttt{false} \rightarrow D_2\end{array}}$$

(MATCH-FALSE)

$$\frac{R \vdash e \xrightarrow{\texttt{false}} e' \qquad R \vdash D_2 \xrightarrow{R_2} D_2'}{R \vdash \begin{array}{l}\texttt{match}\ e\ \texttt{with}\\ \mid \texttt{true} \rightarrow D_1\\ \mid \texttt{false} \rightarrow D_2\end{array} \xrightarrow{R_2} \begin{array}{l}\texttt{match}\ e'\ \texttt{with}\\ \mid \texttt{true} \rightarrow D_1\\ \mid \texttt{false} \rightarrow D_2'\end{array}}$$

FIG. 7.9 – Sémantique avec horloge : règle pour le `match`/`with`.

comme leur homologues.

La figure 7.11 montre les règles applicables à la structure de contrôle `match/with`. Ces règles assurent que, si l'expression conditionnelle émet une valeur sur le site s, alors cette valeur peut être communiquée à tout site nécessaire à l'évaluation de D_1 et D_2.

7.6 Système de types

Nous étendons maintenant le système de types du chapitre 5, afin de traiter le langage ainsi étendu.

Un type spécial pour les horloges, non localisé, est ajouté aux types spatiaux ; il est noté « clock » :

$$tc ::= \ldots \mid \text{clock}$$

Les opérations d'échantillonnage sont ajoutées à l'environnement initial H_0 :

$$H_0 = \left[\begin{array}{c} \ldots \\ \cdot \ \texttt{when} \ \cdot : \forall\alpha.\forall\delta.\alpha \ \texttt{at} \ \delta \times \text{clock} \ \texttt{at} \ \delta \prec\{\delta\}\succ\!\rightarrow \alpha \ \texttt{at} \ \delta, \\ \texttt{merge} \ \cdot \ \cdot \ \cdot : \forall\alpha.\forall\delta.(\text{clock} \ \texttt{at} \ \delta \times \alpha \ \texttt{at} \ \delta \times \alpha \ \texttt{at} \ \delta) \prec\{\delta\}\succ\!\rightarrow \alpha \ \texttt{at} \ \delta \end{array} \right]$$

Le type de ces opérations assure qu'elles seront effectuées sur un unique site s, sur lequel leurs arguments sont présents.

Les règles de la figure 7.12 sont ensuite ajoutées.

- La règle Clock vérifie que la définition d'une horloge peut être exécutée sur n'importe quel site unique. Cette horloge est disponible partout, et est donc de type spatial $\forall\delta.\,\text{clock} \ \texttt{at} \ \delta$.
- Une structure `match/with` ne peut être répartie que si, le résultat de l'expression conditionnelle étant localisé sur un site s, tous les sites impliqués dans l'exécution de chacune de ses branches sont accessibles depuis s (règle Match).

7.7 Projection

La figure 7.13 montre les règles d'inférence des canaux de communications pour les constructions ajoutées. Tout d'abord, le calcul d'une horloge globale ne nécessite aucun canal de communication (règle Clock-C). Afin de permettre de communiquer le résultat de l'expression conditionnelle vers tous les sites où la structure de contrôle sera dupliquée, un canal de communication est ajouté par site participant à l'exécution des équations de chaque cas du `match` (règle Match-C).

Enfin, la figure 7.14 montre la projection des définitions d'horloges, dupliquées sur tous les sites (règle Clock-P), et la projection des structures de contrôle.

(When-Init)

$$\frac{\hat{R}(c) = \langle e_2 \rangle \qquad \hat{R} \overset{\ell}{\Vdash} e_1 \texttt{ when } \langle e_2 \rangle \xrightarrow{\hat{v}} e'}{\hat{R} \overset{\ell}{\Vdash} e_1 \texttt{ when } c \xrightarrow{\hat{v}} e'}$$

(When-[])

$$\frac{\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{[] \texttt{ at } s} e_1' \qquad \hat{R} \overset{\{s\}}{\Vdash} e_2 \xrightarrow{[] \texttt{ at } s} e_2'}{\hat{R} \overset{\ell_1}{\Vdash} e_1 \texttt{ when } \langle e_2 \rangle \xrightarrow{[] \texttt{ at } s} e_1' \texttt{ when } \langle e_2' \rangle}$$

(When-True)

$$\frac{\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{vl_1 \texttt{ at } s} e_1' \qquad \hat{R} \overset{\{s\}}{\Vdash} e_2 \xrightarrow{\texttt{true at } s} e_2'}{\hat{R} \overset{\ell_1}{\Vdash} e_1 \texttt{ when } \langle e_2 \rangle \xrightarrow{vl_1 \texttt{ at } s} e_1' \texttt{ when } \langle e_2' \rangle}$$

(When-False)

$$\frac{\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{dv_1 \texttt{ at } s} e_1' \qquad \hat{R} \overset{\{s\}}{\Vdash} e_2 \xrightarrow{\texttt{false at } s} e_2'}{\hat{R} \overset{\ell_1}{\Vdash} e_1 \texttt{ when } \langle e_2 \rangle \xrightarrow{[] \texttt{ at } s} e_1' \texttt{ when } \langle e_2' \rangle}$$

(Merge-Init)

$$\frac{\hat{R}(c) = \langle e_1 \rangle \qquad \hat{R} \overset{\ell}{\Vdash} \texttt{merge } \langle e_1 \rangle\ e_2\ e_3 \xrightarrow{\hat{v}} e'}{\hat{R} \overset{\ell}{\Vdash} \texttt{merge } c\ e_2\ e_3 \xrightarrow{\hat{v}} e'}$$

(Merge-[])

$$\frac{\hat{R} \overset{\{s\}}{\Vdash} e_1 \xrightarrow{[] \texttt{ at } s} e_1' \qquad \hat{R} \overset{e_2}{\Vdash} \ell_2 \xrightarrow{[] \texttt{ at } s} e_2' \qquad \hat{R} \overset{e_3}{\Vdash} \ell_3 \xrightarrow{[] \texttt{ at } s} e_3'}{\hat{R} \overset{\ell_2 \cup \ell_3}{\Vdash} \texttt{merge } \langle e_1 \rangle\ e_2\ e_3 \xrightarrow{[] \texttt{ at } s} \texttt{merge } \langle e_1' \rangle\ e_2'\ e_3'}$$

(Merge-True)

$$\frac{\hat{R} \overset{\{s\}}{\Vdash} e_1 \xrightarrow{\texttt{true at } s} e_1' \qquad \hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{dv_2 \texttt{ at } s} e_2' \qquad \hat{R} \overset{\ell_3}{\Vdash} e_3 \xrightarrow{[] \texttt{ at } s} e_3'}{\hat{R} \overset{\ell_2 \cup \ell_3}{\Vdash} \texttt{merge } \langle e_1 \rangle\ e_2\ e_3 \xrightarrow{dv_2 \texttt{ at } s} \texttt{merge } \langle e_1' \rangle\ e_2'\ e_3'}$$

(Merge-False)

$$\frac{\hat{R} \overset{\{s\}}{\Vdash} e_1 \xrightarrow{\texttt{false at } s} e_1' \qquad \hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{[] \texttt{ at } s} e_2' \qquad \hat{R} \overset{\ell_3}{\Vdash} e_3 \xrightarrow{dv_3 \texttt{ at } s} e_3'}{\hat{R} \overset{\ell_2 \cup \ell_3}{\Vdash} \texttt{merge } \langle e_1 \rangle\ e_2\ e_3 \xrightarrow{dv_3 \texttt{ at } s} \texttt{merge } \langle e_1' \rangle\ e_2'\ e_3'}$$

Fig. 7.10 – Sémantique pour la répartition des opérations d'échantillonnage et de combinaison.

(Match-[])

$$\frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{[]\ \texttt{at}\ s} e' \qquad \texttt{var}\ (D_1) = \{x_1, \ldots, x_n\}}{\hat{R} \overset{\ell}{\Vdash} \begin{array}{l} \texttt{match}\ e\ \texttt{with} \\ \mid \texttt{true} \rightarrow D_1 \\ \mid \texttt{false} \rightarrow D_2 \end{array} \xrightarrow{[[]/x_1,\ldots,[]/x_n]} \begin{array}{l} \texttt{match}\ e'\ \texttt{with} \\ \mid \texttt{true} \rightarrow D_1 \\ \mid \texttt{false} \rightarrow D_2 \end{array}}$$

(Match-True)

$$\frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\texttt{true at}\ s} e' \qquad \hat{R} \overset{\ell_1}{\Vdash} D_1 \xrightarrow{\hat{R}_1} D_1' \qquad \forall s' \in \ell_1, (s, s') \in \mathcal{L}}{\hat{R} \overset{\ell \cup \ell_1}{\Vdash} \begin{array}{l} \texttt{match}\ e\ \texttt{with} \\ \mid \texttt{true} \rightarrow D_1 \\ \mid \texttt{false} \rightarrow D_2 \end{array} \xrightarrow{\hat{R}_1} \begin{array}{l} \texttt{match}\ e'\ \texttt{with} \\ \mid \texttt{true} \rightarrow D_1' \\ \mid \texttt{false} \rightarrow D_2 \end{array}}$$

(Match-False)

$$\frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\texttt{false at}\ s} e' \qquad \hat{R} \overset{\ell_2}{\Vdash} D_2 \xrightarrow{R_2} D_2' \qquad \forall s' \in \ell_2, (s, s') \in \mathcal{L}}{\hat{R} \overset{\ell \cup \ell_2}{\Vdash} \begin{array}{l} \texttt{match}\ e\ \texttt{with} \\ \mid \texttt{true} \rightarrow D_1 \\ \mid \texttt{false} \rightarrow D_2 \end{array} \xrightarrow{R_2} \begin{array}{l} \texttt{match}\ e'\ \texttt{with} \\ \mid \texttt{true} \rightarrow D_1 \\ \mid \texttt{false} \rightarrow D_2' \end{array}}$$

Fig. 7.11 – Sémantique pour la répartition du `match`/`with`.

$$\text{(Clock)}\quad \frac{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : b \texttt{ at } s/\{s\} \qquad s \notin \mathcal{S}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash \texttt{clock } c = e : [\forall\delta.\,\text{clock } \texttt{ at } \delta/c]/\emptyset}$$

$$\text{(Match)}\quad \frac{\begin{array}{c} H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : b \texttt{ at } s/\ell \qquad H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_1 : H'/\ell_1 \\ H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_2 : H'/\ell_2 \qquad \forall s' \in \ell_1 \cup \ell_2, s = s' \vee (s,s') \in \mathcal{L} \end{array}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash \begin{array}{l} \texttt{match } e \texttt{ with} \\ \mid \texttt{true} \rightarrow D_1 \\ \mid \texttt{false} \rightarrow D_2 \end{array} : H'/\ell \cup \ell_1 \cup \ell_2}$$

Fig. 7.12 – Règles de typage des horloges et du `match`/`with`.

La projection d'une structure de contrôle sur A est définie par les trois règles suivantes :

- Si A ne participe pas à l'exécution des cas du `match`, et n'est pas le site sur lequel se trouve le résultat de l'expression conditionnelle, alors le `match` est supprimé (règle Match-P-Suppr).
- Si A est le site sur lequel se trouve le résultat de l'expression conditionnelle, alors le résultat de la projection de cette expression est utilisé pour définir une variable fraîche x, qui est utilisée pour la communication de la condition vers les sites où le `match` est dupliqué (règle Match-P-From).
- Si A participe à l'exécution des cas du `match`, alors celui-ci est dupliqué sur A. La condition est remplacée par le canal de communication utilisé pour sa définition sur le site où cette condition est calculée. Chaque cas du `match` contient les définitions additionnelles, avec la valeur absente, des canaux non utilisés dans le cas considéré (règle Match-P-To).

7.8 Discussion

Soulignons tout d'abord que la complexité de la solution proposée est le reflet des problèmes posés à la répartition manuelle de programmes. De ce point de vue, le caractère automatique de notre méthode permet d'enlever au programmeur une charge réelle de travail systématique et potentiellement source d'erreurs, telle que la duplication de code de contrôle (par exemple ici les horloges et les structures `match`/`with`), duplication pouvant compromettre la modularité du système.

Ce chapitre a montré l'attention particulière à porter aux structures de contrôle dans le cadre de la répartition. En particulier, l'utilisation d'horloges locales dé-

$$\text{(Clock-C)}\quad \frac{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : b \texttt{ at } s/\{s\}/\emptyset \qquad s \notin \mathcal{S}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash \texttt{clock } c = e : [\forall\delta.\,\text{clock } \texttt{ at } \delta/c]/\emptyset/\emptyset}$$

$$\text{(Match-C)}\quad \frac{\begin{array}{c} H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : b \texttt{ at } s/\ell/T \qquad H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_1 : H'/\ell_1/T_1 \\ H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_2 : H'/\ell_2/T_2 \qquad \forall s' \in \ell_1 \cup \ell_2, s = s' \vee (s,s') \in \mathcal{L} \\ T' = [s \overset{c_1}{\mapsto} s_1, \ldots, s \overset{c_n}{\mapsto} s_n] \text{ où } \{s_1,\ldots,s_n\} \in (\ell_1 \cup \ell_2) \setminus \{s\} \end{array}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash \begin{array}{l} \texttt{match } e \texttt{ with} \\ \mid \texttt{true} \rightarrow D_1 \\ \mid \texttt{false} \rightarrow D_2 \end{array} : H'/\ell \cup \ell_1 \cup \ell_2/T, T_1, T_2, T'}$$

Fig. 7.13 – Règles de typage et inférence des canaux de communications des horloges et du `match/with`.

finies dynamiquement pose un problème, car elle rend la répartition dépendante du calcul d'horloge. Les restrictions imposées ici au langage (horloges globales et expressions `match/with`) permettent de traiter la répartition indépendamment du calcul d'horloge. Néanmoins, il serait intéressant d'examiner l'intégration de notre méthode de répartition au calcul d'horloge. Le profit peut dans ce cas être double : d'une part lever les restrictions sur les horloges utilisées, et d'autre part permettre par répartition de séparer les parties du programme s'exécutant à différents rythmes [41].

(Clock-P)

$$\frac{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : b \texttt{ at } s/\{s\}/\emptyset \qquad s \notin \mathcal{S}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash \texttt{clock } c = e : [\forall\delta.\,\text{clock } \texttt{ at } \delta/c]/\emptyset/\emptyset \overset{A}{\Longrightarrow} \texttt{clock } c_A = e}$$

(Match-P-Suppr)

$$\frac{\begin{array}{c} H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : b \texttt{ at } s/\ell/T \overset{A}{\Longrightarrow} \bot/D \qquad H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_1 : H'/\ell_1/T_1 \\ H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_2 : H'/\ell_2/T_2 \qquad \forall s' \in \ell_1 \cup \ell_2, s = s' \vee (s,s') \in \mathcal{L} \\ T' = [s \overset{c_1}{\mapsto} s_1, \ldots, s \overset{c_n}{\mapsto} s_n] \text{ où } \{s_1,\ldots,s_n\} \in (\ell_1 \cup \ell_2) \setminus \{s\} \qquad A \notin \ell_1 \cup \ell_2 \cup \{s\} \end{array}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash \texttt{match } e \texttt{ with } | \texttt{ true} \to D_1 \mid \texttt{false} \to D_2 : H'/\ell \cup \ell_1 \cup \ell_2/T, T_1, T_2, T' \overset{A}{\Longrightarrow} D}$$

(Match-P-From)

$$\frac{\begin{array}{c} H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : b \texttt{ at } A/\ell/T \overset{A}{\Longrightarrow} e'/D \qquad H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_1 : H'/\ell_1/T_1 \overset{A}{\Longrightarrow} D'_1 \\ H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_2 : H'/\ell_2/T_2 \overset{A}{\Longrightarrow} D'_2 \qquad \forall s' \in \ell_1 \cup \ell_2, A = s' \vee (A,s') \in \mathcal{L} \\ T' = [A \overset{c_1}{\mapsto} s_1, \ldots, A \overset{c_n}{\mapsto} s_n] \text{ où } \{s_1,\ldots,s_n\} \in (\ell_1 \cup \ell_2) \setminus \{A\} \\ \text{dom}(T_1) = \{c_{11},\ldots,c_{1n_1}\} \qquad \text{dom}(T_2) = \{c_{21},\ldots,c_{2n_2}\} \qquad x \notin \text{dom}(H) \end{array}}{\begin{array}{l} H|\langle\mathcal{S},\mathcal{L}\rangle \vdash \texttt{match } e \texttt{ with } | \texttt{ true} \to D_1 \mid \texttt{false} \to D_2 : H'/\ell \cup \ell_1 \cup \ell_2/T, T_1, T_2, T' \\ \quad \overset{A}{\Longrightarrow} D \;\; \texttt{and } x = e' \texttt{ and } c_1 = x \texttt{ and } \ldots \texttt{ and } c_n = x \\ \qquad\qquad \texttt{and match } x \texttt{ with} \\ \qquad\qquad\qquad | \texttt{ true} \to D'_1 \texttt{ and } c_{21} = \bot \texttt{ and } \ldots \texttt{ and } c_{2n_2} = \bot \\ \qquad\qquad\qquad | \texttt{ false} \to D'_2 \texttt{ and } c_{11} = \bot \texttt{ and } \ldots \texttt{ and } c_{1n_1} = \bot \end{array}}$$

(Match-P-To)

$$\frac{\begin{array}{c} H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : b \texttt{ at } s/\ell/T \overset{A}{\Longrightarrow} \bot/D \qquad H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_1 : H'/\ell_1/T_1 \overset{A}{\Longrightarrow} D'_1 \\ H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_2 : H'/\ell_2/T_2 \overset{A}{\Longrightarrow} D'_2 \qquad \forall s' \in \ell_1 \cup \ell_2, s = s' \vee (s,s') \in \mathcal{L} \\ T' = [s \overset{c_1}{\mapsto} s_1, \ldots, s \overset{c_i}{\mapsto} A, \ldots, s \overset{c_n}{\mapsto} s_n] \text{ où } \{s_1,\ldots,s_n\}(\ell_1 \cup \ell_2) \setminus \{s\} \\ \text{dom}(T_1) = \{c_{11},\ldots,c_{1n_1}\} \qquad \text{dom}(T_2) = \{c_{21},\ldots,c_{2n_2}\} \end{array}}{\begin{array}{l} H|\langle\mathcal{S},\mathcal{L}\rangle \vdash \texttt{match } e \texttt{ with } | \texttt{ true} \to D_1 \mid \texttt{false} \to D_2 : H'/\ell \cup \ell_1 \cup \ell_2/T, T_1, T_2, T' \\ \quad \overset{A}{\Longrightarrow} D \texttt{ and match } c_i \texttt{ with} \\ \qquad\qquad\qquad | \texttt{ true} \to D'_1 \texttt{ and } c_{21} = \bot \texttt{ and } \ldots \texttt{ and } c_{2n_2} = \bot \\ \qquad\qquad\qquad | \texttt{ false} \to D'_2 \texttt{ and } c_{11} = \bot \texttt{ and } \ldots \texttt{ and } c_{1n_1} = \bot \end{array}}$$

Fig. 7.14 – Règle de projection des horloges et du `match`/`with`.

Chapitre 8

Architectures locales

Nous avons vu que, l'opération de répartition étant dirigée par les types, il est nécessaire que ceux-ci respectent certaines propriétés. Notamment, le schéma de type d'un nœud ne peut comporter plus d'une variable de sites, sous peine de ne pouvoir, dans le cas général, obtenir un unique nœud pour chaque site réel de l'architecture. Ce chapitre montre comment des schémas de type plus généraux peuvent être calculés, grâce à l'introduction de la notion d'architecture locale aux nœuds.

8.1 Motivation

Le besoin d'architectures locales correspond à un besoin de modularité de l'architecture. Particulièrement, ce besoin de modularité se rencontre lorsqu'un code est écrit non pas pour une architecture définie, mais pour un type d'architecture générique qui sera instancié avec une architecture concrète lors de l'instanciation de ce code.

Reprenons l'exemple du canal de réception de la section 3.2 :

```
node channel(filter,demod,x) = y where
      m = filter(x) at FPGA
  and y = demod(m) at DSP
```

Supposons maintenant que le nœud `channel` soit décrit non pas comme réparti de manière définitive sur les sites concrets `FPGA` et `DSP`, mais réparti sur une architecture quelconque, comprenant deux sites liés par un lien de communication.

Cette situation correspond à l'introduction de « polymorphisme de sites » dans le système de types. Dans les chapitres précédents, le seul polymorphisme utilisé correspondait aux nœuds recevant un type de la forme $\forall\delta.tc$ `at` $\delta \prec\!\langle\{\delta\}\rangle\!\rightarrow tc'$ `at` δ, c'est-à-dire un nœud non réparti disponible sur tous les sites. Le but est donc

d'étendre le système de types afin d'exprimer le fait qu'un nœud puisse être réparti sur plusieurs sites, spécifiés non plus à la définition du nœud, mais à son instanciation. On veut donc pouvoir écrire un programme de la forme :

```
node f(x) = y where
      x1 = x + 1 at δ1
  and y = x1 + 2 at δ2
```

où δ_1 et δ_2 sont deux sites locaux au nœud `f`. Le type de ce nœud serait alors :

$$\texttt{f} : \forall \delta_1, \delta_2 : \{\delta_1 \triangleright \delta_2\}.b \texttt{ at } \delta_1 \prec\!\{\delta_1, \delta_2\}\!\succ\!\!\rightarrow b \texttt{ at } \delta_2$$

Dans un schéma de type, les variables de sites quantifiées sont accompagnées d'un ensemble de contraintes de la forme $s \triangleright s'$, signifiant que ou bien $s = s'$, ou bien il existe un lien de communication de s vers s'. À l'instanciation de `f`, la contrainte $\delta_1 \triangleright \delta_2$ sera résolue par le compilateur, qui prendra des valeurs pour δ_1 et δ_2 telles que cette contrainte soit vérifiée. Une instanciation de `f` peut en particulier être exécutée entièrement sur un même site : la contrainte $\delta_1 \triangleright \delta_2$ est alors résolue en $\delta_1 = \delta_2$.

8.2 Extension du langage

Nous proposons ici une extension simple du langage, permettant d'exprimer l'existence d'une architecture locale lors de la définition des nœuds. Cette extension correspond à un compromis entre une notation légère et transparente au programmeur et la simplicité des types inférés.

La première préoccupation, poussée à l'extrême, viserait à permettre au programmeur de ne fournir, pour un nœud donné, aucune directive de répartition, et d'inférer le type « le plus général », en insérant le plus de communications possible. Les limites de cette approche sont évidentes : les communications pouvant être insérées par sous-typage à n'importe quel endroit, le type inféré peut devenir arbitrairement complexe, et le comportement réparti d'un tel nœud échappe à la compréhension du programmeur.

De plus, l'extension doit être conservatrice : le type des nœuds tels qu'exposés dans les chapitres précédents doit être le même une fois l'extension définie. Ainsi, sans aucune directive de répartition, le schéma de type ne comprendra toujours qu'une seule variable de site.

L'extension proposée consiste ici donc à permettre au programmeur de donner une liste de *sites locaux* $\delta_1, \ldots, \delta_n$ à un nœud `f`, selon la syntaxe :

```
node f[δ1,...,δn](x) = ...
```

Ces sites locaux δ_i peuvent ensuite être utilisés pour les directives de répartition du corps de `f`. Les communications entre ces sites ne sont pas contraintes :

l'ensemble des liens de communication nécessaires sera inféré à partir des liens effectivement utilisés. Cet ensemble permettra de définir les contraintes entre les variables de sites du schéma de type, contraintes qui seront résolues à chaque instanciation de `f`. Ces instanciations gardent la même syntaxe :

```
y = f(x)
```

Les contraintes inférées sur les variables $\delta_1, \ldots, \delta_n$ lors du typage de `f` sont alors résolues à partir des types spatiaux de `x` et de `y`.

Le canal de réception de notre radio logicielle peut donc être défini avec une architecture locale composée de deux sites δ_1 et δ_2 :

```
node channel[δ1,δ2](filter,demod,x) = y where
      m = filter(x) at δ1
  and y = demod(m) at δ2
```

Son type spatial devient alors :

$$\texttt{channel} : \forall \alpha, \beta, \gamma . \forall \delta_1, \delta_2 : \{\delta_1 \triangleright \delta_2\}. \left(\begin{array}{ll} & (\alpha \texttt{ at } \delta_1 \prec\!\{\delta_1\}\!\succ\!\rightarrow \beta \texttt{ at } \delta_1) \\ \times & (\beta \texttt{ at } \delta_2 \prec\!\{\delta_2\}\!\succ\!\rightarrow \gamma \texttt{ at } \delta_2) \\ \times & \alpha \texttt{ at } \delta_1 \end{array} \right) \prec\!\{\delta_1, \delta_2\}\!\succ\!\rightarrow \gamma \texttt{ at } \delta_2$$

Supposons maintenant que l'architecture globale du système soit composée de plusieurs FPGAs et plusieurs DSPs ; il est alors possible d'instancier le nœud `channel` avec plusieurs paires de sites différents. Le code ci-dessous montre la définition d'un système composé de deux canaux de réception en parallèle, sur deux paires d'éléments FPGA et DSP différents :

```
loc FPGA1
loc FPGA2
loc DSP1
loc DSP2
link FPGA1 to DSP1
link FPGA2 to DSP2

node multichannel_sdr(x1,x2) = (y1,y2) where
      y1 = channel(filter_1800 at FPGA1, demod_gmsk at DSP1, x1)
  and y2 = channel(filter_2000 at FPGA2, demod_qpsk at DSP2, x2)
```

8.3 Système de types modifié

La syntaxe du langage devient :

$$
\begin{array}{lll}
P & ::= & d_1\,;\ldots;d_n\,;D \\
d & ::= & \epsilon \mid \texttt{node}\ f[\delta_1,\ldots,\delta_n](p) = e\ \texttt{where}\ D \\
D & ::= & \epsilon \mid p = e \mid p = x(e) \mid D\ \texttt{and}\ D \mid \texttt{let}\ D\ \texttt{in}\ D \\
e & ::= & i \mid x \mid (e,e) \mid \mathrm{op}(e,e) \mid e\ \texttt{fby}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ \texttt{at}\ s \\
s & ::= & A \mid \delta \\
p & ::= & p,p \mid x \\
i & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid \ldots
\end{array}
$$

Cette syntaxe permet la déclaration locale d'une liste de variables de sites lors de la définition des nœuds, et l'utilisation de variables de sites pour la localisation d'expressions. On suppose, pour que la syntaxe soit conservatrice, que $(\texttt{node}\ f[](p) = e\ \texttt{where}\ D) = (\texttt{node}\ f(p) = e\ \texttt{where}\ D)$.

Un schéma de type est un type spatial quantifié avec un ensemble $\{\alpha_1,\ldots,\alpha_n\}$ de variable de types, et un ensemble $\{\delta_1,\ldots,\delta_p\}$ de variable de sites, auxquelles on associe un ensemble de contraintes C :

$$
\begin{array}{ll}
\sigma & ::= \forall\alpha_1,\ldots,\alpha_n.\forall\delta_1,\ldots,\delta_p : C.t \\
\vdots & \\
C & ::= \{s_1 \triangleright s'_1,\ldots,s_n \triangleright s'_n\}
\end{array}
$$

Un schéma de type peut être instancié en un couple (t, C), où C est l'ensemble des contraintes devant être vérifiées par les variables de sites instanciées.

$$
\Big(t[t_1/\alpha_1,\ldots,t_n/\alpha_n, s_1/\delta_1,\ldots,s_p/\delta_p], C[s_1/\delta_1,\ldots,s_p/\delta_p]\Big) \leq \forall\alpha_1\ldots\alpha_n.\forall\delta_1\ldots\delta_p : C.t
$$

$$
(t\ \texttt{at}\ s, C) \leq (H\ \texttt{at}\ s)(x) \Leftrightarrow (t\ \texttt{at}\ s, C) \leq H(x)
$$

Un ensemble de contraintes C est dit *compatible* avec un ensemble de liens de communication $\mathcal{L}$, noté $\mathcal{L} \models C$, si et seulement si $\forall s$ et s', $s \triangleright s' \in C \wedge s \neq s' \Rightarrow (s, s') \in \mathcal{L}$.

La généralisation d'un type t en schéma de type, dans l'environnement de typage H et l'architecture locale $\langle\mathcal{S},\mathcal{L}\rangle$, est notée $\mathrm{gen}_{H|\mathcal{S}|\mathcal{L}}(t)$. Dans le cas où $\mathcal{S} = \emptyset$, cette fonction de généralisation est identique à celle définie au chapitre 5. Sinon, les variables quantifiées sont celles de l'ensemble $\mathcal{S}$, et les contraintes proviennent

de l'ensemble des liens de communication utilisés :

$$\begin{cases} \mathrm{gen}_{H|\emptyset|\emptyset}(t) = \forall \alpha_1, \ldots, \alpha_n . \forall \delta : \emptyset . tc \ \texttt{at} \ \delta & \text{ssi } t = tc \ \texttt{at} \ \delta \text{ et } \delta \notin \mathrm{FLV}(H) \\ \mathrm{gen}_{H|\emptyset|\emptyset}(t) = \forall \alpha_1, \ldots, \alpha_n . t & \text{ssi } \mathrm{FLV}(t) = \emptyset \\ \mathrm{gen}_{H|\mathcal{S}|\mathcal{L}}(t) = \forall \alpha_1, \ldots, \alpha_n . \forall \delta_1, \ldots, \delta_p : C . t & \text{où } \{\delta_1, \ldots, \delta_p\} = \mathcal{S} \\ & \text{et } C = \{s \triangleright s' | (s, s') \in \mathcal{L}\} \end{cases}$$

$$\text{où } \{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(t) - \mathrm{FTV}(H)$$

Les règles de définition de ces prédicats sont données en figure 8.1. La règle NODE permet d'étendre l'architecture courante avec une architecture locale. L'ensemble des liens de communication utilisé est élargi par un sous-ensemble des liens entre variables de sites introduites, et entre ces variables et les sites de l'architecture globale. Ce sous-ensemble peut être calculé *a posteriori*, à partir des liens de communication effectivement utilisés : il peut par exemple être calculé à partir de l'ensemble des canaux de communications inférés. Les autres règles sont similaires aux règles de la figure 5.2.

8.4 Discussion

L'extension proposée ici est légère, et permet une réelle amélioration de l'expressivité du langage, par la possibilité d'exprimer des fonctionnalités réparties instanciées dans différents contextes d'architecture. Cette préoccupation rejoint la notion de conception orientée plateforme (« platform-based design » [18]), dans laquelle la conception du système est faite en tenant compte, non pas de l'architecture précise et entièrement définie, mais d'une abstraction de celle-ci. Cette notion vise à permettre d'instancier cette abstraction avec l'architecture réelle le plus tard possible dans le déroulement de la conception du système.

La répartition d'un nœud associé à une architecture locale n'est pas aisée. Dans tous les cas, cette répartition ne peut consister en la projection d'un unique nœud spécialisé pour chaque site de l'architecture globale, tel que proposé dans les chapitres précédents. En effet, cette spécialisation pour un site donné suppose la suppression des parties du nœud non concernées par ce site, ce qui n'est pas possible quand la valeur réelle d'un site n'est connue qu'à l'instanciation du nœud. La méthode de répartition présentée au chapitre 6 ne s'applique donc pas à cette extension.

Plusieurs pistes alternatives peuvent être explorées. La première consiste en l'*inlining* des nœuds avec architectures locales, c'est-à-dire le remplacement de leurs instances par le corps du nœud. Cette solution est la plus simple à mettre en œuvre, mais a l'inconvénient de la perte de modularité lors de la répartition. La deuxième possibilité se rapproche de la compilation des horloges en Lucid Synchrone [24] : des gardes booléennes sont ajoutées aux calculs des expressions et

des équations afin de permettre de déterminer dynamiquement, à partir du site sur lequel le programme est exécuté, quelles expressions ou équations sont effectivement à calculer. L'ajout des canaux de communication entre variables de sites devient dans ce cas un problème délicat, puisque ceux-ci apparaissent à la fois en tant qu'entrée et en tant que sortie du nœud produit par répartition. Cependant, cette solution est modulaire et plus raisonnable en terme de passage à l'échelle pour son application à des programmes industriels. De plus, c'est une solution qui peut être rapprochée de l'option, considérée au chapitre 7, d'intégration de la méthode de répartition avec le calcul d'horloge. Elle rejoint de plus la vision GALS des systèmes répartis, dans le sens où un site serait alors vu comme une horloge autonome du système ; les liens de communication entre sites comme l'existence de moyens techniques, abstraits lors de la conception du système global, permettant de transmettre des valeurs d'une horloge à une autre.

$$\text{(Prog)}\quad \frac{\langle\emptyset,\emptyset\rangle \vdash \mathcal{A} : G \qquad H_0|G \vdash d : H/\ell \qquad H, H_1|G \vdash D : H_1/\ell'}{H \vdash \mathcal{A}; d; D : H_1}$$

$$\text{(Imm)}\quad H|G \vdash i : b \texttt{ at } s/\{s\}$$

$$\text{(Inst)}\quad \frac{(t, C) \leq (H(x)) \qquad \mathcal{L} \models C}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash x : t/\operatorname{locations}(t)}$$

$$\text{(Pair)}\quad \frac{H|G \vdash e_1 : t_1/\ell_1 \qquad H|G \vdash e_2 : t_2/\ell_2}{H|G \vdash (e_1, e_2) : t_1 \times t_2/\ell_1 \cup \ell_2}$$

$$\text{(Op)}\quad \frac{H|G \vdash \mathrm{op} : t_1 \prec\!\{s\}\!\succ t_2/\{s\} \qquad H|G \vdash e : t_1/\ell}{H|G \vdash \mathrm{op}(e) : t_2/\{s\} \cup \ell}$$

$$\text{(At)}\quad \frac{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : t/\{s\} \qquad s \in \mathcal{S}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e \texttt{ at } s : t/\{s\}}$$

$$\text{(Comm)}\quad \frac{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : tc \texttt{ at } s/\ell \qquad \mathcal{L} \models s \triangleright s'}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : tc \texttt{ at } s'/\ell \cup \{s'\}}$$

$$\text{(Node)}\quad \frac{\begin{array}{c} H, x_i : t_i, H_1|\langle\mathcal{S}\cup\mathcal{S}',\mathcal{L}\cup\mathcal{L}'\rangle \vdash D : H_1/\ell_1 \qquad H, x_i : t_i, H_1|\langle\mathcal{S}\cup\mathcal{S}',\mathcal{L}\cup\mathcal{L}'\rangle \vdash e : t/\ell_2 \\ \mathcal{S}' = \{\delta_1, \ldots, \delta_n\} \qquad \mathcal{L}' \subseteq (\mathcal{S}' \times (\mathcal{S}\cup\mathcal{S}')) \cup (\mathcal{S} \times \mathcal{S}') \end{array}}{\begin{array}{c} H|G \vdash \texttt{node } f[\delta_1, \ldots, \delta_n](x) = e \texttt{ where } D : \\ [\mathrm{gen}_{H|\mathcal{S}'|\mathcal{L}'}((t_1 \times \ldots \times t_n) \prec\!\ell_1 \cup \ell_2\!\succ t)/f]/\ell_1 \cup \ell_2 \end{array}}$$

$$\text{(Def)}\quad \frac{H|G \vdash e : t/\ell}{H|G \vdash x = e : [t/x]/\ell}$$

$$\text{(App)}\quad \frac{H|G \vdash f : t_1 \prec\!\ell_1\!\succ t_2/\ell_2 \qquad H|G \vdash e : t_1/\ell_3}{H|G \vdash x = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3}$$

$$\text{(And)}\quad \frac{H|G \vdash D_1 : H_1/\ell_1 \qquad H|G \vdash D_2 : H_2/\ell_2}{H|G \vdash D_1 \texttt{ and } D_2 : H_1, H_2/\ell_1 \cup \ell_2}$$

$$\text{(Let)}\quad \frac{H|G \vdash D_1 : H_1/\ell_1 \qquad H, H_1|G \vdash D_2 : H_2/\ell_2}{H|G \vdash \texttt{let } D_1 \texttt{ in } D_2 : H_2/\ell_1 \cup \ell_2}$$

$$\text{(Match)}\quad \frac{\begin{array}{c} H|\langle\mathcal{S},\mathcal{L}\rangle \vdash e : b \texttt{ at } s/\ell \\ H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_1 : H'/\ell_1 \qquad H|\langle\mathcal{S},\mathcal{L}\rangle \vdash D_2 : H'/\ell_2 \qquad \forall s' \in \ell_1 \cup \ell_2, \mathcal{L} \models s \triangleright s' \end{array}}{H|\langle\mathcal{S},\mathcal{L}\rangle \vdash \texttt{match } e \texttt{ with } | \texttt{ true} \rightarrow D_1 \mid \texttt{false} \rightarrow D_2 : H'/\ell \cup \ell_1 \cup \ell_2}$$

Fig. 8.1 – Système de types spatial avec architecture locale.

Chapitre 9

Implémentation et discussion

Ce chapitre présente en premier lieu l'implémentation de notre méthode de répartition modulaire dans le compilateur Lucid Synchrone. Nous discutons ensuite des limites de l'approche proposée, en terme d'expressivité et de domaines d'application visés. Ces limites sont relativisées par l'existence de choix restrictifs initiaux guidés par un certain type d'application, et dont l'extension ne remet pas en cause la méthode elle-même. Enfin, nous réexprimons l'exemple de radio logicielle en Lucid Synchrone, en utilisant l'ordre supérieur pour exprimer la reconfiguration dynamique. Cet exemple prospectif permet de montrer l'intérêt de la méthode présentée dans un langage flot de données avec ordre supérieur dynamique.

9.1 Implémentation

Le système d'inférence de types spatiaux présenté au chapitre 5, la méthode de répartition automatique du chapitre 6, ainsi que l'introduction du contrôle présentée au chapitre 7, ont été implémentées dans la version 3 du compilateur Lucid Synchrone. L'inférence de types est composée de 2000 lignes de code OCaml, dont 330 dédiées à l'algorithme naïf de résolution des contraintes de répartition. L'implémentation de la méthode de répartition automatique comporte 300 lignes de code.

La compilation de programmes Lucid Synchrone vers du code séquentiel dans un langage de programmation généraliste s'effectue, dans ce compilateur, en deux étapes. La première étape produit à partir du code en langage source un code dans un langage intermédiaire d'équations sous forme SSA[1] et munies de gardes issues de la compilation des horloges. Cette première étape est précédée de l'analyse de types, d'horloge et de causalité (permettant de rejeter respectivement les

[1]Static Single Assignment

programmes mal typés, non synchrones et non causaux), ainsi qu'une analyse d'initialisation qui permet de rejeter les programmes dont certaines mémoires ne sont pas initialisées. La deuxième étape effectue une série de transformations successives sur ce code intermédiaire : compilation des automates sous forme d'équations, ordonnancement séquentiel des équations, élimination de code mort. Cette étape se termine par la production, à partir de ces équations, de code séquentiel dans le langage cible (actuellement le langage OCaml [58]).

L'inférence de types spatiaux s'insère naturellement dans la première étape. Cette insertion est très peu intrusive : la seule modification opérée sur les autres analyses est l'ajout dans la syntaxe du langage source de la construction $\cdot$ `at` A. Cette modification est très légère, puisque cette construction ne modifie pas la sémantique synchrone du programme.

La répartition elle-même est effectuée sur le code intermédiaire. Deux solutions différentes ont été expérimentées :

1. La première correspond à la méthode de répartition pour les systèmes GALS décrite en section 6.6. Les canaux de communication sont ajoutés en entrée, et sont des références sur des sockets Unix sur lesquelles les données peuvent être envoyées et reçues. Cette méthode est appliquée après l'ordonnancement des équations, assurant ainsi la correction et l'absence de blocage du système réparti ainsi produit [16].
2. La deuxième solution consiste à appliquer la répartition sur le code intermédiaire avant toute autre transformation. On obtient ainsi autant de codes intermédiaires que de sites déclarés, sur lesquels les autres transformations sont appliquées de manière indépendante. Cette deuxième solution est plus proche de la formalisation proposée : le résultat, bien que sous forme de code intermédiaire, est un ensemble de programmes synchrones dont les communications ont été ajoutées en entrées et en sorties. Cependant, le caractère contraignant de la compilation modulaire du langage ne permet de traiter de cette manière que les architectures ne comprenant pas de cycles de communication. En effet, il est alors nécessaire, pour l'évaluation d'un nœud sur un site donné, de connaître les valeurs de toutes ses entrées. Or, contrairement à la première solution où les entrées additionnelles sont des références permettant d'obtenir ces valeurs, les entrées ajoutées pour la mise en œuvre de cette deuxième solution portent directement les valeurs communiquées.

Dans les deux cas, un mode d'exécution du système réparti a été implémenté. Ce mode d'exécution produit une socket Unix par canal de communication inféré. Le passage d'une implémentation à l'autre n'a été l'objet que de légères modifications ; la structure mise en place pour la première implémentation a été facilement réutilisée. La seule modification induite par ces deux solutions sur les autres transformations effectuées sur le code intermédiaire est la propagation des types

spatiaux vers le code intermédiaire transformé.

Ces deux implémentations confirment donc le caractère modulaire de notre méthode de répartition automatique. Le fait qu'elle soit applicable au niveau de diverses phases de la compilation tend à démontrer son indépendance du langage ou du format considéré.

9.2 Limites d'expressivité

Les limites de l'expressivité de notre méthode se trouvent, d'une part dans la syntaxe des extensions du langage, et d'autre part dans le langage et les contraintes d'architectures considérées.

Ces deux considérations sont liées : l'objectif étant la répartition automatique sur une architecture donnée, les extensions de syntaxe sont liées à la nature des contraintes induites par cette architecture. Le système de types permet, à partir des contraintes données par les annotations du programmeur, de vérifier la cohérence de ces annotations, et d'utiliser ces contraintes pour donner une localisation aux flots et aux opérations du programme. Nous avons vu dans les chapitres précédents que cette dernière résolution des contraintes pouvait être faite soit de manière globale, soit de manière modulaire, la méthode de répartition exposée au chapitre 6 ne s'appliquant qu'aux nœuds dont les contraintes sont entièrement résolues.

Les contraintes et le langage d'architecture considérés sont ici :

- un langage de graphe pour l'architecture, dans lequel les nœuds sont les sites et les arcs les liens de communications physiques entre ces sites ;
- le sous-typage introduit des contraintes de la forme $s \triangleright s'$, signifiant que, ou bien s et s' sont égaux, ou bien il existe un lien de communication de s vers s' ;
- la construction e `at` s introduit des contraintes d'égalité entre les sites rencontrés dans e et le site s.

Ces choix initiaux peuvent toutefois être remis en question sans remettre en cause la méthode proposée.

Il a par exemple été question, à l'occasion de cette thèse, d'étendre le langage d'architecture pour permettre d'exprimer des architectures *hiérarchiques*, où les sites comprennent eux-mêmes d'autres sous-sites. L'objectif était de fournir un moyen d'expression plus riche de la structure de l'architecture, de permettre d'exprimer certaines contraintes de communication, et de permettre au programmeur d'exprimer la localisation d'opérations sous la forme d'un ensemble de sites.

Toutefois, on peut faire remarquer que l'usage d'un langage simple de graphe d'architecture force l'expression explicite de toutes les ressources de calcul, et des liens de communication entre elles. Au contraire, la mise en œuvre d'une extension du langage d'architecture avec l'expression de hiérarchies de sites ne peut faire

l'économie d'une réflexion plus exigeante des besoins réellement exprimés par de telles architectures. Autrement dit, l'introduction de hiérarchie peut être un moyen de rendre implicite certaines caractéristiques de l'architecture, dont l'expression sous forme de simple graphe était forcément explicite :

Masquage : l'inclusion d'un sous-site à l'intérieur d'un autre site père peut être l'occasion de masquer l'existence du site inclus vis-à-vis des autres sites. Dans ce cas, les liens de communication ne peuvent traverser les frontières du site père.

Liens de communication implicites : le fait de rendre certains liens de communication implicites peut être un confort d'écriture pour le programmeur. Il peut s'agir des liens entre sites pères et fils, ou entre sites inclus dans le même site père. Cela correspond, par exemple, aux MPSoCs[2] munis d'un NoC[3].

Regroupement de sites : l'inclusion de plusieurs sous-sites à l'intérieur d'un site père peut permettre d'exprimer la localisation d'une opération sur un de ces sites, la charge d'en choisir un en particulier étant laissée à la phase de compilation/répartition automatique. Dans ce cas, la construction e `at` s introduit, en lieu et place des contraintes d'égalité, des contraintes d'inclusion des sites utilisés par l'expression e dans l'ensemble de sites composé de s et de ses descendants.

Ces préoccupations sont fortement dépendantes du contexte et notamment du type d'architectures considérées. Elles peuvent de plus être contradictoires : si l'expression de la hiérarchie est utilisée pour exprimer des ensemble de sites, alors il ne peut être automatiquement question de liens de communication implicites à l'intérieur de cet ensemble. De plus, les ensembles considérés peuvent être d'intersection non nulle (on peut vouloir par exemple utiliser l'ensemble des processeurs, et l'ensemble des ressources situées géographiquement à un endroit donné — certains processeurs inclus), ce qui compromet l'utilisation de la notion même de hiérarchie pour cette considération.

Ces questions restent ouvertes, et illustrent les limites du langage d'architecture considéré, qui suffit cependant à démontrer l'intérêt de l'approche présentée dans cette thèse. D'autres types de contraintes, essentielles dans le contexte des systèmes embarqués, n'ont pas non plus été considérées : ce sont les contraintes de coût, notamment de temps d'exécution au pire cas des opérations, en fonction de leur localisation. L'inclusion de telles contraintes dans les types spatiaux des nœuds, combinée à l'approche proposée dans cette thèse, permettrait de traiter de manière modulaire et intégrée au langage source des problèmes d'ordonnancement traités habituellement de manière spécifique à un format ou un langage donné.

[2]Multi-Processor System on Chips
[3]Network on Chip

9.3 Application à d'autres cadres de répartition

Nous souhaitons maintenant montrer comment la méthode présentée dans cette thèse peut être utilisée, ou à tout le moins adaptée, dans d'autres contextes de répartition. Les trois exemples considérés ont été introduits dans le chapitre de motivation : il s'agit des programmes comprenant plusieurs rythmes de calcul, des systèmes tolérants aux fautes, et de l'application de la synthèse de contrôleurs discrets.

9.3.1 Asynchronie des rythmes

Les systèmes embarqués comportent souvent des calculs coûteux et longs, à exécuter sur une horloge plus lente, et éventuellement en arrière-plan pour permettre aux calculs plus rapides et plus urgents de préempter ces tâches lentes.

La notion d'horloge du paradigme synchrone permet de traiter de tels problèmes, à condition que le programmeur séquentialise à la main les tâches lentes, afin qu'une partie de ces tâches soit calculée sur l'horloge de base, le résultat final étant sur une horloge plus lente. Les travaux d'Alain Girault, Xavier Nicollin et Marc Pouzet [41] montrent comment utiliser la répartition automatique pour exécuter de telles tâches lentes sur un processeur différent, puis par désynchronisation, faire en sorte que l'exécution de ces tâches ne ralentisse pas le reste du système.

Dans cette sous-section, nous allons considérer un exemple tiré du cadre de l'outil ORCCAD [13] : les « tâches-robots » sont des graphes flot de données, composées d'opérations de base (appelées « modules algorithmiques ») elles-mêmes programmées en C. La période d'exécution de ces opérations de base est précisée par le programmeur, ainsi que le mode de communication (synchrone ou asynchrone) entre deux modules du graphe. L'outil permet alors d'obtenir automatiquement une tâche temps-réel par période d'exécution ; ces tâches sont exécutées comme processus légers (« threads ») sur un processeur temps-réel [75]. Le mode de communication permet de spécifier le mode de synchronisation entre ces tâches : le plus souvent, des communications asynchrones sont utilisées entre deux tâches de périodes différentes. Dans le contexte des systèmes robotiques, dans lequel ORCCAD est utilisé, on considère en général qu'à chaque instant, le résultat des tâches lentes à utiliser est le dernier résultat rendu, quel que soit l'état courant de leur calcul [74]. Par exemple, ces tâches lentes peuvent être le calcul de matrices d'inertie, ou de compensation de gravité. La valeur de ces matrices évolue avec la position du robot, mais il n'est cependant pas critique d'utiliser une valeur datant par exemple d'un dixième de seconde, alors même qu'une nouvelle valeur est en cours de calcul. Ces valeurs sont utilisées pour le calcul d'une nouvelle commande à envoyer au robot, dont la période la plus critique doit être de l'ordre de quelques millisecondes. Les périodes d'exécution spécifiées permettent donc ici uniquement de regrouper

les calculs effectués à la même période dans une même tâche temps-réel : il n'est pas fait d'analyse de temps d'exécution, ni de calcul de date limite d'exécution à partir de ces périodes.

La méthode proposée dans cette thèse peut être utilisée dans ce contexte. Les sites sont alors les périodes d'exécution, et la répartition automatique permet d'obtenir une tâche temps-réel par période utilisée. Il ne s'agit alors pas de répartition au sens physique du terme, mais de répartition logique : on obtient plusieurs programmes synchrones, exécutés sur le même processeur à des rythmes différents. Ces tâches temps-réel peuvent être composées de manière asynchrone, en utilisant un tampon par valeur communiquée d'une tâche à une autre. On se place alors dans un contexte où, comme on l'a vu, la sémantique synchrone centralisée n'a pas à être préservée : c'est même une condition, autorisée par le contexte considéré, permettant l'exécution de tâches en parallèle et à des rythmes différents. Un exemple d'un tel contexte est la communication entre tâches de priorités différentes, exécutées dans un environnement préemptif. Cet exemple a été décrit par Norman Scaife et Paul Caspi dans [72].

9.3.2 Tolérance aux fautes

La tolérance aux fautes permet de garantir la non défaillance du système en présence de fautes logicielles ou matérielles. Les stratégies de tolérance aux fautes, qui sont l'objet d'un vaste domaine de recherche, utilisent en général la redondance matérielle ou logicielle [38].

Une stratégie simple de tolérance aux fautes matérielles consiste en la duplication d'une opération sur trois processeurs différents, suivie d'un vote par comparaison des valeurs produites par ces trois processeurs (« Triple Modular Redundancy »). Cette stratégie permet de tolérer une faute d'un processeur pendant le calcul de cette opération, et peut être programmée comme suit :

```
node tmr[δ1,δ2,δ3](f1,f2,f3,x) = y where
      y1 = f1(x) at δ1
  and y2 = f2(x) at δ2
  and y3 = f3(x) at δ3
  and y = if (y1 = y2) or (y1 = y3) then y1 else y2
```

Notons que si deux défaillances ou plus se produisent parmi δ_1, δ_2 et δ_3, alors la fonction `tmr` ci-dessus fournit un résultat erroné, mais cette situation est en dehors de son domaine de spécification.

Ce nœud peut ensuite être instancié avec trois nœuds et une entrée. Ces trois nœuds doivent être trois instances d'un même nœud disponible sur ces trois sites. Le langage n'est cependant pas suffisant : dans le contexte de la tolérance aux

fautes, il est nécessaire de contraindre les trois sites δ_1, δ_2 et δ_3 à être différents, et donc d'étendre le langage de contraintes utilisées.

Les bénéfices attendus d'une telle extension sont ceux exposés pour la répartition modulaire. Elle pourrait permettre d'exprimer certaines stratégies de tolérance aux fautes sous la forme de modules paramétrés par les nœuds dupliqués, rendant ainsi leur mise en œuvre elle-même modulaire. Il serait donc intéressant d'évaluer quels types de stratégies peuvent être programmées de cette manière ; étant entendu cependant que cette application ne recouvre certainement pas l'étendue du domaine de la tolérance aux fautes.

9.3.3 Synthèse de contrôleurs

La synthèse de contrôleurs est une méthode automatique permettant d'obtenir statiquement, à partir d'un système non déterministe et d'une propriété temporelle non assurée *a priori* par ce système, un contrôleur qui interdit les comportements du système d'origine qui violent la propriété temporelle désirée.

Du point de vue de la répartition, cette méthode est intéressante du fait qu'elle n'opère que sur des systèmes d'équations booléennes mis à plat, ce qui interdit les calculs numériques et l'utilisation d'ordre supérieur dynamique : dans le cas général, il faut donc extraire une abstraction strictement booléenne du système, effectuer la synthèse de contrôleurs dessus, puis être en mesure de recomposer le système contrôlé avec la partie non booléenne du système d'origine.

De manière plus générale, il s'agit ici de traiter des systèmes sur une partie desquels une transformation *quelconque* doit être faite. La méthode de répartition automatique permet de séparer le système en deux parties, dont une sur laquelle sera effectuée la transformation en question. L'automatisation de cette séparation permet d'être en mesure de recomposer la partie transformée avec l'autre partie du système, et d'assurer l'équivalence sémantique de cette recomposition.

L'application de cette méthode à la synthèse de contrôleurs est présentée en détail dans [32]. La figure 9.1 résume cette application.

L'application de la synthèse de contrôleurs nécessite dans un premier temps la partition des entrées du système en deux sous-ensembles, les entrées incontrôlables I_u et les entrées contrôlables I_c. Ces dernières permettent d'exprimer le caractère non déterministe du système. Le contrôleur synthétisé calcule, à chaque instant, quelles valeurs donner à ces entrées contrôlables pour que le système vérifie la propriété temporelle à assurer. L'utilisation de la répartition dans ce contexte consiste à considérer deux « sites », un sur lequel les opérations booléennes sont effectuées, et un comprenant toutes les autres opérations. La répartition automatique permet d'obtenir deux programmes P_b (booléen) et P_d (données), dont le premier est une abstraction booléenne du programme d'origine. Cette répartition ajoute des entrées incontrôlables I_{db} au programme P_b : il s'agit par exemple des opérations de

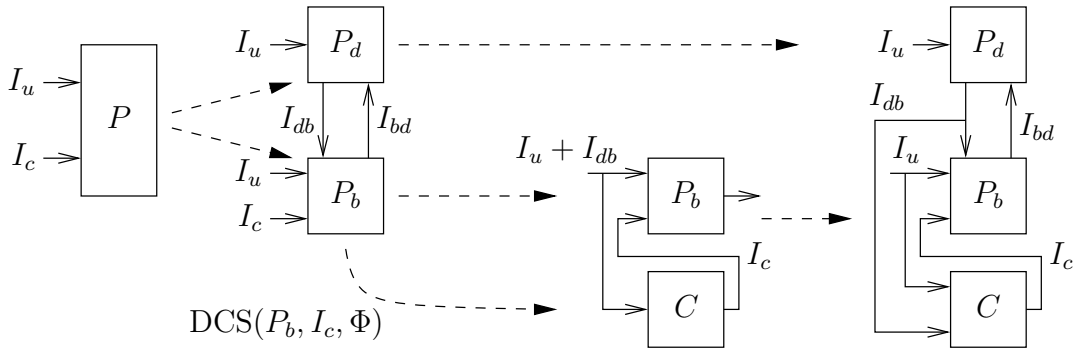


FIG. 9.1 – Application de la répartition automatique pour la synthèse de contrôleurs appliquée à une partie du programme.

comparaison numériques, qui sont effectuées par P_d, et dont le résultat booléen est communiqué à P_b. Les entrées additionnelles I_{bd} du programme P_d sont les valeurs booléennes calculées par l'abstraction booléenne, entrées contrôlables comprises.

La synthèse de contrôleurs est ensuite effectuée sur le programme P_b. La composition de ce programme avec le contrôleur calculé est ensuite recomposée de manière synchrone avec le programme P_d. Cette recomposition conserve la sémantique du programme original ; la preuve est basée sur l'équivalence sémantique du produit synchrone de P_b et P_d, d'une part, et sur la préservation par tout produit synchrone des propriétés de sûreté assurées par le contrôleur synthétisé. De plus, si la propriété à assurer par synthèse Φ est une propriété de sûreté, alors elle est conservée par produit synchrone [45].

Cette application montre de manière générale que la méthode de répartition proposée permet d'appliquer des transformations spécifiques à une partie du programme, cette partie n'étant pas forcément modulaire d'un point de vue fonctionnel. La compilation séparée pour des architectures hétérogènes est un autre exemple de ce type de transformation.

9.4 Ordre supérieur et reconfiguration dynamique

Nous terminons enfin ce chapitre de discussion par un exemple prospectif, que nous espérons motivant. Nous souhaitons ici reprendre l'exemple de canaux de réception multiples de la radio logicielle de la figure 7.4, et d'utiliser l'ordre supérieur pour exprimer la *reconfiguration dynamique*, cette fois non pas sous la forme de structures de contrôle dupliquées, mais de téléchargement de code d'une ressource à l'autre. Nous reprenons pour l'occasion la syntaxe concrète de Lucid Synchrone, la syntaxe formelle introduite dans la section 2.4.2 ne permettant pas d'exprimer l'ordre supérieur dynamique.

Nous supposons ici que les ressources FPGA et DSP ne peuvent pas accueillir en même temps les nœuds de tous les canaux de réception disponibles. Ces nœuds seront donc envoyés dynamiquement, lors de la reconfiguration du canal de réception. Nous utilisons pour cet exemple la combinaison des signaux et des automates à états paramétrés, constructions introduites dans [28]. Un signal permet d'encapsuler une valeur et son horloge, et permet ainsi de définir des flots de nœuds : les valeurs du signal sont alors les nœuds, dont les horloges sont synchronisées sur l'horloge de base du signal. L'utilisation des états paramétrés permet de récupérer les valeurs émises sur un signal, et de les utiliser comme paramètres statiques d'un état.

```
let node channel reconfigure x = y where
  rec automaton
      | Init ->
          do y = x
          until reconfigure(filter,demod)
          then Configure(filter,demod)
      | Configure(filter,demod) ->
          let f = run filter x in
          do y = run demod f
          until reconfigure(filter',demod')
          then Configure(filter',demod')
      end

let node multichannel_sdr x = y where
  rec y = channel switch_channel x
  and automaton
      | Reconfigure_GSM ->
          do emit switch_channel = (filter_1800,demod_gmsk)
          then GSM
      | GSM ->
          do until (umts y) then Reconfigure_UMTS
      | Reconfigure_UMTS ->
          do emit switch_channel = (filter_2000,demod_qpsk)
          then UMTS
      | UMTS ->
          do until (gsm y) then Reconfigure_GSM
      end
```

FIG. 9.2 – Canaux de réception multiples d'une radio logicielle programmés à l'aide de l'ordre supérieur dynamique.

Le programme Lucid Synchrone de ces canaux de réception avec reconfiguration dynamique se trouve en figure 9.2. Le nœud `channel` est un automate, qui attend l'occurrence d'un signal d'entrée `reconfigure`. Ce signal porte un couple de nœuds `(filter,demod)` (le mot-clé `run` permet d'indiquer au compilateur que `filter` et `demod` sont des nœuds, et non des fonctions combinatoires). L'occurrence de ce signal permet d'initialiser, ou de réinitialiser, l'état `Configure` avec deux nouvelles occurrences de ces deux nœuds. La cohérence des nœuds portés par le signal du point de vue de leur entrées/sorties est assurée par typage. Entre deux occurrences du signal, l'état interne des nœuds est conservé ; il est réinitialisé à chaque fois qu'un nouveau couple de nœuds est reçu.

Le nœud `multichannel_sdr` instancie le nœud `channel`, en définissant le signal d'entrée au moyen d'un automate. Cet automate émet ce signal dans deux états « transitoires », `Reconfigure_GSM` et `Reconfigure_UMTS`, qui permettent de passer de l'état `GSM`, dans lequel le canal de réception GSM est utilisé, à l'état `UMTS`. Les fonctions booléennes `gsm` et `umts` permettent de déterminer les transitions d'un état à l'autre, en fonction de la sortie `y`.

Les perspectives offertes par un tel exemple dans le cadre de la répartition sont de pouvoir exprimer la reconfiguration dynamique d'une ressource physique par une autre, au moyen de l'envoi du nouveau code à exécuter par un canal de communication [27]. Il s'agirait ici d'annoter l'émission du signal `switch_channel`, ou l'exécution entière de l'automate, sur un troisième site. Un canal de communication serait alors ajouté, permettant l'envoi des valeurs portées par ce signal vers les ressources FPGA et DSP. Dans cette perspective, le système de types doit nécessairement être modifié, pour permettre d'exprimer l'existence de nœuds présents sur un site, mais exécutables sur un autre. Une telle fonctionnalité pose alors des problèmes intéressants en terme de répartition automatique, notamment en vue d'architectures hétérogènes.

9.5 Conclusion

Nous avons discuté au cours de ce chapitre de l'implémentation, puis des limites d'expressivité de la méthode proposée. Nous avons proposé trois autres domaines d'application pour notre méthode. Bien que ces domaines ne soient pas forcément traitables directement, l'adaptation nécessaire de la méthode pour ces problèmes spécifiques est aisée et concerne des points orthogonaux à l'approche générale considérée, tels que le langage de contraintes ou d'architecture. Enfin, un exemple prospectif, programmé dans le langage Lucid Synchrone, démontre l'intérêt de l'application d'une méthode de répartition modulaire à un langage flots-de-données d'ordre supérieur, permettant d'exprimer des fonctionnalités de reconfiguration dynamique sous la forme de flots de nœuds.

Chapitre 10

Conclusion

10.1 Résumé

Nous avons présenté une nouvelle méthode de répartition automatique intégrée à la compilation d'un langage flots de données synchrones. Cette méthode est basée sur des extensions du langage permettant l'ajout d'annotations de répartition à un programme flots de données. Un système de types à effets dédié à la répartition permet de vérifier la cohérence de ces annotations et d'inférer la localisation de tous les calculs d'un programme partiellement annoté. Nous avons défini, à l'aide de ce système de types, une opération de projection permettant d'obtenir un programme spécialisé par ressource de l'architecture.

Le fait d'utiliser un système de types rend possible la modularité de notre méthode de répartition. La structure fonctionnelle du programme réparti est donc la même que celle du programme centralisé. Cette modularité permet d'intégrer notre méthode de manière naturelle dans un processus global de compilation modulaire, et donc de bénéficier des fonctionnalités et de l'expressivité fournie par le compilateur au sein duquel cette intégration est réalisée. Ainsi, l'implémentation de cette méthode dans le compilateur du langage flots de données Lucid Synchrone permet de bénéficier de l'expression de l'ordre supérieur, fourni par ce langage, dans un cadre réparti.

Avec la formalisation de cette méthode, nous avons fourni une sémantique répartie, destinée à fournir au programmeur le sens le plus précis possible des annotations de répartition utilisées. Le système de types proposé permet de vérifier la cohérence des annotations vis-à-vis de cette sémantique répartie. La sémantique du programme réparti par projection est donnée par le produit synchrone, dans le langage source, des fragments issus de la projection. Cette sémantique est équivalente, pour tout programme accepté par le système de types, à la sémantique du programme original. La conservation de cette sémantique, et le fait d'utiliser

la même sémantique pour le programme original et le programme réparti, permet d'effectuer de manière orthogonale d'autres analyses et transformations, soit sur le programme original avant répartition, soit sur les fragments issus de la projection. Le système de types ainsi que l'opération de projection ont été implémentés dans le compilateur de Lucid Synchrone.

L'extension du langage proposé permet de déclarer l'architecture sous la forme d'un graphe, dont les sommets sont les ressources disponibles, et les arêtes les liens de communication possibles entre ces ressources. Cette extension permet de plus d'annoter certaines opérations du programme, contraignant ainsi leur localisation. Ce cadre particulier définit un langage de contraintes, lui-même indépendant de la méthode de répartition proposée. Nous avons discuté de la modification de ce langage de contraintes, en vue de l'adaptation de notre méthode dans des domaines d'application différents que celui initialement considéré, tels que la tolérance aux fautes, ou l'application d'une méthode de transformation, par exemple la synthèse de contrôleurs, sur une partie définie du programme.

Ce travail se place dans le cadre d'une volonté d'unification de langages, et d'intégration de méthodes et d'outils indépendants. À travers l'objectif de la répartition vers des architectures hétérogènes, nous avons enfin souhaité montrer qu'une telle intégration n'est pas un obstacle à la mise en œuvre d'analyses spécifiques, sur des parties du programme indépendantes de sa structure fonctionnelle. L'utilisation de systèmes de types spécifiques est de ce point de vue un outil bien adapté, du fait qu'ils permettent de bénéficier à la fois d'une intégration modulaire de la méthode considérée, et d'un retour vers le programmeur du résultat de l'application de cette méthode, ici sous la forme du type spatial des nœuds du programme. Cette méthode convient donc particulièrement à la mise en œuvre de méthodes formelles transparentes pour le programmeur.

10.2 Perspectives

Les perspectives à court terme concernent l'expressivité du langage et du système de types. L'intérêt de l'utilisation d'architectures locales, permettant d'accroître la modularité du système, a été démontrée. Nous avons montré comment étendre le système de types, les perspectives de ce point de vue concernent donc l'extension de la méthode de répartition en tenant compte de telles architectures locales. Nous souhaitons, dans cette perspective, considérer l'intégration du système de types spatiaux au sein du calcul d'horloge, afin de bénéficier de la compilation sous forme de code séquentiel des horloges en Lucid Synchrone. La répartition modulaire de nœuds utilisant plusieurs variables de sites nous paraît en effet relever de la même démarche que la compilation des horloges, tant en concept qu'en pratique. On peut en effet considérer que la répartition d'un programme synchrone

revient à considérer une horloge par ressource physique, les canaux de communication représentant un moyen technique pour projeter une valeur d'une horloge à une autre. Il est alors nécessaire, pour représenter ces communications, d'ajouter un mécanisme de sous-typage au calcul d'horloge.

L'expressivité du langage d'architecture peut aussi être étendue, en particulier concernant la possibilité d'exprimer l'architecture de manière hiérarchique ou modulaire. Une telle extension permettrait de représenter des architectures matérielles plus complexes, telles que les MPSoCs[1] ou les NoCs[2]. Il serait aussi intéressant de considérer l'ajout à ce langage d'architecture de propriétés temps-réel, par exemple la latence associée aux liens de communication ou à certaines ressources, et leur impact sur la méthode proposée. Un tel ajout peut alors être l'occasion de l'évaluation de l'indépendance effective du traitement des contraintes et de la méthode de répartition. L'extension du langage de contraintes pour l'application de cette méthode à d'autres problèmes incluant la répartition de programme, tel que le domaine de la tolérance aux fautes, s'inscrit dans la même démarche.

Enfin, une des perspectives les plus motivantes est l'extension de notre méthode à l'ordre supérieur dynamique, qui n'est pas traité en l'état. Une telle extension permettrait l'expression de flots de fonctions dans un cadre réparti, et ainsi de la reconfiguration de ressources physiques par l'envoi de leur nouvelle fonctionnalité depuis la ressource reconfigurante. Nous avons montré l'utilité d'une telle technique pour la conception de systèmes embarqués reconfigurables, par exemple dans le contexte de la radio logicielle. Le cadre formel, ainsi que le cadre technique d'une telle extension restent à définir. Mais nous pensons que cette perspective reste particulièrement pertinente, au regard des besoins croissants de sécurité, de flexibilité et de reconfigurabilité des systèmes embarqués.

[1] Multi-Processor System on Chips
[2] Network on Chips

Bibliographie

[1] AUTOSAR consortium. `http://www.autosar.org`.

[2] Lucid synchrone v3. `http://www.lri.fr/~pouzet/lucid-synchrone`.

[3] Architecture analysis & design language (AADL). SAE Standard (AS5506), Nov. 2004.

[4] R. M. Amadio. On modelling mobility. *Journal of Theoretical Computer Science*, 240 :147–176, 2000.

[5] R. M. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed pi-calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5) :549–577, Sept. 2003.

[6] P. Amagbégnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language Signal. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation (PLDI'95)*, pages 163–173, New York, NY, USA, 1995. ACM.

[7] C. André. Representation and analysis of reactive behaviors : A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille, France, July 1996. IEEE-SMC.

[8] C. André, F. Boulanger, and A. Girault. Software implementation of synchronous programs. In *International Conference on Application of Concurrency to System Design, ICACSD'01*, pages 133–142, Newcastle, UK, June 2001. IEEE.

[9] P. Aubry, P. Le Guernic, and S. Machard. Synchronous distribution of signal programs. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, pages 656–665, Honolulu, HW, USA, Jan. 1996.

[10] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of the IEEE, Special issue on embedded systems*, 91(1) :64–83, Jan. 2003.

[11] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction : Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.

[12] N. Blackwell, S. Leinster-Evans, and S. Dawkins. Developing safety cases for integrated flight systems. In *Proceedings of the IEEE Aerospace Conference*, volume 5, pages 225–240, 1999.

[13] J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The Orccad architecture. *Int. J. of Robotics Research*, 17(4), 1998.

[14] G. Boudol. ULM : A core programming model for global computing. In *European Symposium on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 234–248, Barcelona, Spain, Apr. 2004.

[15] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy : A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2) :155–182, 1994.

[16] B. Caillaud, P. Caspi, A. Girault, and C. Jard. Distributing automata for asynchronous networks of processors. *European Journal of Automated Systems*, 31(3) :503–524, 1997. Research report Inria 2341.

[17] L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Proc. FOSSACS'98, International Conference on Foundations of Software Science and Computation Structures*, volume 1378 of *LNCS*, pages 140–155, Lisbon, Portugal, 1998. Springer-Verlag.

[18] L. Carloni, F. De Bernardinis, C. Pinello, A. L. Sangiovanni-Vincentelli, and M. Sgroi. Platform-based design for embedded systems. *The Embedded Systems Handbook*, 2005.

[19] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94 :125–140, 1992.

[20] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to Scade/Lustre to TTA : A layered approach for distributed embedded applications. In *International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES'03*, pages 153–162, San Diego, USA, June 2003. ACM.

[21] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Trans. on Software Engineering*, 25(3) :416–427, May 1999.

[22] P. Caspi, C. Mazuet, and N. Reynaud Paligot. About the design of distributed control systems : the quasi-synchronous approach. In *SAFECOMP 2001*, volume 2187. LNCS, 2001.

[23] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ICFP '96 : Proceedings of the first ACM SIGPLAN International Conference on Functional Programming*, pages 226–238, New York, NY, USA, 1996. ACM Press.

[24] P. Caspi and M. Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998. Extended version available as a VERIMAG tech. report no. 97–07 at `www.lri.fr/~pouzet`.

[25] D. M. Chapiro. *Globally Asynchronous Locally Synchronous Systems.* PhD thesis, Stanford University, Oct. 1984.

[26] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. *SIGOPS Oper. Syst. Rev.*, 41(6) :31–44, 2007.

[27] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, september 2004.

[28] J.-L. Colaço, G. Hamon, and M. Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006.

[29] J.-L. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[30] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In R. Alur and I. Lee, editors, *International Conference on Embedded Software (EMSOFT'03)*, volume 2855 of *LNCS*, pages 134–155, Philadelphia, USA, Oct. 2003. Springer-Verlag.

[31] P. Cuoq and M. Pouzet. Modular causality in a synchronous stream language. In *European Symposium on Programming (ESOP'01)*, volume 2028 of *LNCS*, pages 237–251, Genova, Italy, April 2001. Springer-Verlag.

[32] G. Delaval. Modular distribution and application to discrete controller synthesis. In *International Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P'08)*, Budapest, Hungary, Apr. 2008.

[33] G. Delaval, A. Girault, and M. Pouzet. A type system for the automatic distribution of higher-order synchronous dataflow programs. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, Tucson, Arizona, USA, June 2008.

[34] S. A. Edwards. SHIM : A language for hardware/software integration. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'05)*, ENTCS, Edinburgh, UK, Apr. 2005. Elsevier Science.

[35] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia : Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[36] P. H. Feiler, B. A. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL), a standard for engineering performance critical systems. In *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design*, Munich, Germany, Oct. 2006.

[37] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '96)*, pages 372–385, New York, NY, USA, 1996. ACM Press.

[38] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1) :1–26, 1999.

[39] A. Girault. *Contribution à la conception sûre des systèmes embarqués sûrs.* Habilitation à diriger des recherches, INPG, Grenoble, France, Sept. 2006.

[40] A. Girault, B. Lee, and E. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(6) :742–760, June 1999. Research report UCB/ERL M97/57.

[41] A. Girault, X. Nicollin, and M. Pouzet. Automatic rate desynchronization of embedded reactive programs. *ACM Trans. on Embedded Computing Systems*, 5(3) :687–717, Aug. 2006.

[42] G. Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones ; application à Esterel.* PhD thesis, École des Mines de Paris, Mar. 1988.

[43] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic, 1993.

[44] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. of the IEEE*, 79(9) :1305–1320, Sept. 1991.

[45] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third International Conference on Algebraic Methodology and Software Technology (AMAST'93)*, Twente, Netherland, June 1993. Workshops in Computing, Springer Verlag.

[46] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from dataflow programs. In *Third International Symposium on Programming Language*

Implementation and Logic Programming (PLILP'91), Passau, Germany, Aug. 1991.

[47] G. Hamon. *Calcul d'horloge et Structures de Contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML.* PhD thesis, Université Pierre et Marie Curie, Paris, France, novembre 2002.

[48] G. Hamon. Synchronous data-flow pattern matching. In *Synchronous Languages, Applications, and Programming (SLAP'04)*, Barcelona, Spain, 2004. Electronic Notes in Theoretical Computer Science.

[49] D. Harel. Statecharts : A visual approach to complex systems. *Science of Computer Programming*, 8(3) :231–275, June 1987.

[50] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems, NATO.* Springer-Verlag, 1985.

[51] S. Haridi, P. Van Roy, and G. Smolka. An overview of the design of distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA, July 1997. ACM Press.

[52] G.-D. Jo, M.-J. Sheen, S.-H. Lee, and K.-R. Cho. A DSP-based reconfigurable SDR platform for 3G systems. In *IEICE Transactions on Communications*, volume E88-B, pages 678–686, 2005.

[53] F. Jondral. Software-defined radio — basics and evolution to cognitive radio. *EURASIP Journal on Wireless Communications and Networking*, 3 :275–283, 2005.

[54] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74 : Proceedings of the IFIP Congress*, pages 471–475, New York, NY, 1974. North-Holland.

[55] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SynDEx software environment for real-time distributed systems, design and implementation. In *Proceedings of European Control Conference, ECC'91*, Grenoble, France, July 1991.

[56] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Lemaire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9) :1321–1336, Sept. 1991.

[57] X. Leroy. Unboxed objects and polymorphic typing. In *19th Symposium on Principles of Programming Languages (POPL'92)*, pages 177–188, Albuquerque, New Mexico, USA, 1992. ACM Press.

[58] X. Leroy. The Objective Caml system release 3.08. Documentation and user's manual. Technical report, INRIA, 2005.

[59] C. Lhoussaine. Type inference for a distributed pi-calculus. In *12th European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 253–268, Warsaw, Poland, 2003. Springer-Verlag.

[60] B. Liblit and A. Aiken. Type systems for distributed data structures. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 199–213, New York, NY, USA, 2000. ACM Press.

[61] F. Maraninchi and Y. Rémond. Mode-automata : About modes and states for reactive systems. In *European Symposium On Programming (ESOP'98)*, Lisbon, Portugal, Mar. 1998. Springer-Verlag.

[62] F. Maraninchi and Y. Rémond. Argos : An automaton-based synchronous language. *Computer Languages*, 27(1–3) :61–92, April–October 2001.

[63] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System : Theory and Applications*, 10(4), October 2000.

[64] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, USA, 1997.

[65] J. Mitola. The software radio architecture. *IEEE Communications Magazine*, 33(5) :26–38, May 1995.

[66] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL '05 : Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–232, New York, NY, USA, 2005. ACM Press.

[67] J. Ouy. A survey of desynchronisation in a polychronous model of computation. In *Electronic Notes in Theoretical Computer Science*, pages 151–167, 2006.

[68] J. Ouy, J.-P. Talpin, L. Besnard, and P. Le Guernic. Separate compilation of polychronous specifications. In *Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS'07)*, volume 200 of *ENTCS*. Elsevier, 2007.

[69] L. Pautet and S. Tardieu. Inside the Distributed Systems Annex. In *Proceedings of 3rd International Conference on Reliable Software Conference*, Uppsala, Sweden, June 1998. Springer-Verlag.

[70] D. Potop-Bucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in Systems Design*, 28(2), Mar. 2006.

[71] E. Rutten and P. Le Guernic. Sequencing data flow tasks in Signal. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando, Florida, June 1994.

[72] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, pages 119–126, Catania, Italy, June 2004.

[73] P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute : High-level programming language design for distributed computation. In *Proceedings of ICFP 2005 : International Conference on Functional Programming*, Tallinn, Estonia, Sept. 2005.

[74] D. Simon and F. Benattar. Design of real-time periodic control systems through synchronisation and fixed priorities. *Int. Journal of Systems Science*, 36(2) :57–76, Feb. 2005.

[75] D. Simon, R. Pissard-Gibollet, K. Kapellos, and B. Espiau. Synchronous composition of discretized control actions : design, verification and implementation with Orccad. In *6th Int. Con f. on Real-Time Control Systems and Application*, Hong-Kong, Dec. 1999.

[76] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3), 1992.

[77] S. Vinoski. CORBA : Integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2) :46–55, Feb. 1997.

[78] M. Ward and H. Zedan. Slicing as a program transformation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(2) :7, 2007.