Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Conception et étude d'un langage de coordination

Linden, Isabelle

Award date:
2002

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 03. Jul. 2025

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique

Année Académique 2001-2002

Conception et étude d'un langage de coordination

Isabelle LINDEN

Mémoire présenté en vue de l'obtention
du grade de Licencié en Informatique

VTLS 20006054

Résumé

Il est de plus en plus reconnu que les systèmes d'information modernes seront composés de nombreux éléments hétérogènes, en général, disséminés au travers de réseaux. Ceci induit un changement radical de conception de systèmes, perçus hier comme isolés, et reconnus aujourd'hui comme distribués, interactifs et hétérogènes. Au niveau de la programmation, cette évolution se traduit par un intérêt grandissant pour les langages dits de coordination.

Ce mémoire s'inscrit dans ce contexte. Il propose un ensemble de primitives de coordination, décrit une algèbre de processus concurrents, en donne une implémentation en utilisant le langage Java comme langage support, effectue une étude sémantique et propose différentes méthodologies de développement de systèmes d'information coordonnées.

Mots-clefs : coordination, langage, sémantiques, méthodologies de programmation

Abstract

It is widely acknowledged that modern information systems will be built from numerous heterogenous components, in general, distributed over networks. This induces a complete change in the conception of systems, conceived in the past as isolated and now as distributed, interactive and heterogenous. At the programming level, this evolution results in an increasing interest for so-called coordination languages.

This thesis proposes a set of coordination primitives, builds a process algebra therefrom, implements it using Java as support language, studies semantics and describes various programming methodologies aiming at developping coordinated information systems.

Keywords : coordination, language, semantics, programming methodologies

Introduction

Contexte

Longtemps réservées aux spécialistes de systèmes d'exploitation et de réseaux, les programmations concurrente et distribuée sont maintenant largement popularisées au point que l'on peut même les qualifier de réalités économiques. Plusieurs facteurs ont contribué à cette situation.

D'une part, l'apparition et la vente de machines à architecture parallèle de coût abordable ont nourri des espoirs de création de codes rapides dans l'esprit de beaucoup de programmeurs. Par suite, des applications impliquant des méthodes d'analyse numérique, de physique informatique, de graphisme, de traitement d'images et d'intelligence artificielle ont été développées.

D'autre part, les réseaux se sont considérablement développés. L'intérêt grandissant marqué pour le réseau de réseaux Internet et pour les réseaux locaux Intranet en sont des signes probants. Permettant à des ordinateurs autonomes de coordonner leurs activités et de partager des ressources, ces réseaux offrent aussi une possibilité d'exécuter en parallèle des fragments de logiciels et peuvent procurer, par suite, un gain d'efficacité tant au niveau de l'écriture de logiciels, en réutilisant des codes existants, qu'au niveau de l'exécution proprement dite en tirant profit de la puissance de différentes machines.

Enfin, l'intérêt récent porté pour la programmation orientée objet en Java, les systèmes multi-agents et encore les systèmes embarqués tels que ceux rencontrés dans l'industrie aérospatiale constituent autant de marques d'intérêt pour des systèmes distribués composés de multiples exécutions de codes soumis à des exigences de concurrence, d'interactivité et de compétition pour des ressources communes.

Les systèmes d'information modernes apparaissent donc comme des ensembles de nombreux composants concurrents, souvent dissiminés au travers de réseaux. Ces composants étant exécutés sur des sites différents, sont, en outre, codés dans des langages différents et, par suite, sont de nature très hétérogènes.

Ceci induit un changement radical de conception de systèmes, perçus hier comme isolés et, reconnus aujourd'hui comme distribués, interactifs et hétérogènes. Récemment, le Professeur P. Wegner ([29]) a d'ailleurs argumenté la thèse selon laquelle l'aspect interactif des systèmes est plus important que leur aspect algorithmique.

Au niveau de la programmation, cette évolution se traduit par un intérêt grandissant pour les langages dits de coordination, dont la caractérisitique commune est, à la suite de l'article [11], de vouloir séparer clairement les aspects de calcul des aspects d'interaction.

Mémoire

Séduit par cette idée de séparation des aspects calcul et interaction, notre mémoire s'est orienté vers une étude des fondements des langages de coordination. Il est essentiellement composé de trois parties abordant chacune trois aspects fondamentaux de la programmation : la conception d'un langage qui permet de décrire les exécutions, la définition d'une sémantique qui permet de préciser le sens à donner aux constructions et finalement des méthodologies qui permettent de guider le processus de conception de programmes corrects et fiables.

La première partie, consacrée aux langages de coordination, propose un nouvel ensemble de primitives de coordination. Tout comme le langage Linda ([11]), il repose sur l'utilisation d'un espace de données partagé par plusieurs processus concurrents. Toutefois, les primitives de Linda ont été remplacées par des primitives d'expressivité équivalente mais constituant un sur-ensemble des primitives proposées par la programmation concurrente par contraintes ([26]) dont l'objectif premier, très similaire au nôtre, était, précisément, de concevoir un paradigme de programmation concurrent dont la communication entre agents concurrents était basée sur la production et le test de disponibilité d'informations. Nous proposons ainsi deux primitives *tell* et *ask*. Toutefois, ces primitives ne sont pas suffisantes pour couvrir la totalité des interactions. En effet, une information présente, par exemple sur une page web, peut être supprimée, par exemple suite au retrait de la page web. De plus, il peut être adéquat d'agir non pas lorsqu'une quantité suffisante d'informations a été produite mais en l'absence d'information. Deux nouvelles primitives *get* et *nask* ont ainsi été introduites pour traiter ces problèmes. Enfin, les données partagées en Linda revêtent la forme de tuples et la sélection d'information s'effectue au moyen d'un "matching" des composants de tuples. Afin d'obtenir une sélection plus souple, nous avons introduit la notion de ψ-termes permettant de nommer les composants et, pour sélectionner des tuples, de n'en spécifier qu'une partie seulement et dans un ordre arbitraire.

Fidèle à notre désir de séparer les aspects de calcul des aspects de communication et d'interaction, les primitives *tell*, *ask*, *get* et *nask* sont indépendantes de tout langage de programmation. Ce fait induit une conséquence très intéressante : l'étude des propriétés de cet ensemble et de la programmation des interactions en l'utilisant peut être effectuée indépendamment de tout langage de programmation. A cet effet, nous définissons un langage abstrait $\mathcal{L}_\Psi$ qui prend la forme d'une algèbre de processus. Ce langage sera utilisé dans les parties 2 et 3 du mémoire.

Par ailleurs, le développement d'une application démontrant l'intérêt pratique de nos développements inclut tôt ou tard certains calculs et, par suite, le mariage d'un langage de programmation avec l'ensemble de primitives proposées. Dans ce but, nous proposons une implémentation de ce mariage avec le langage Java. En outre, des applications PingPong et Chantier sont développées pour illustrer l'intérêt de notre approche pour le codage d'applications distribuées.

La deuxième partie est consacrée à une étude sémantique du langage $\mathcal{L}_\Psi$. Une sémantique opérationnelle est d'abord présentée. Elle est basée sur un système de transition où les exécutions concurrentes sont modélisées par une approche d'entrelacements de pas élémentaires d'exécution. Cette sémantique, bien que simple et intuitive, n'est toutefois pas compositionnelle, une propriété pourtant manifestement suggérée par l'idée même de concevoir des systèmes d'information par composition de fragments de programmes existants.

Nous examinons donc ensuite une sémantique compositionnelle. Elle est obtenue en permettant à l'environnement d'effectuer des pas intermédiaires et, d'une manière équivalente, en insérant des trous dans des histoires sémantiques. La sémantique obtenue est établie compositionnelle. Elle l'est en fait grâce à l'incorporation d'informations non incluses dans la sémantique opérationnelle. Nous prouvons ensuite qu'il s'agit d'une incorporation minimale d'informations ou, en d'autres termes, que la sémantique obtenue est complètement abstraite. La preuve de ce caractère est inspirée de celle publiée en [8]. Toutefois, elle l'étend par la prise en compte de l'affectation de valeurs aux variables.

La dernière partie étudie deux aspects méthodologiques liés au développement de codes corrects. Dans un premier temps, la méthode de composition de spécifications proposée par M. Abadi et L. Lamport ([2]) est adaptée à notre contexte. Dans ce but, le système de transition sous-tendant la sémantique opérationnelle est étendu de façon à faire apparaître la notion d'agents en charge de l'exécution d'instructions. Cela permet de distinguer entre un pas d'exécution effectué par une instruction ou par son environnement. Ensuite, une logique de type Unity est développée et, nous inspirant des techniques de [16], un principe de composition de propriétés est établi.

Dans un deuxième temps, nous utilisons ce principe de composition de propriétés pour proposer une technique de raffinement de spécifications qui permet de dériver un ensemble d'agents concurrents satisfaisant une spécification générale donnée. Un exemple de construction de bâtiments est donné en guise d'illustration.

Remerciements

Qu'il me soit permis de remercier ici tous ceux qui ont rendu possible la réalisation de ce travail. Je voudrais particulièrement remercier mon promoteur, le Professeur J.-M. Jacquet pour ses conseils, son suivi et sa disponibilité, Madame A.-M. Breny et mon père qui ont accepté de relire le manuscrit. Je voudrais aussi mentionner tout ceux, parents, amis, collègues et proches qui, d'une façon ou d'une autre, m'ont apporté soutien et encouragements tout au long de ces deux dernières années d'études.

Table des matières

Introduction iii

I Langage 1

1 Définition du langage $\mathcal{L}_\Psi$ 3
- 1.1 Les langages de coordination . . . 3
- 1.2 Le Langage $\mathcal{L}_\Psi$ . . . 5

2 Implémentation du langage $\mathcal{L}_\Psi$ en Java 9
- 2.1 Affectation . . . 9
- 2.2 PsiTerme . . . 11
- 2.3 PsiTermeSpace . . . 12
- 2.4 Application . . . 13

II Sémantique 15

3 Sémantique 17
- 3.1 Sémantique historique . . . 17
- 3.2 Sémantiques compositionnelles . . . 18
- 3.3 Notations . . . 19
- 3.4 Sémantique dénotationnelle . . . 21
- 3.5 Sémantiques complètement abstraites . . . 28
- 3.6 Abstraction complète . . . 29
  - 3.6.1 Définitions . . . 29
  - 3.6.2 Intuition . . . 30
  - 3.6.3 Résultats Préliminaires . . . 31
  - 3.6.4 Preuves de l'abstraction complète . . . 34

III Méthodologie **39**

4 Composition de spécifications **41**
4.1 Une logique de programmation 41
4.2 Logique de spécification 44
4.2.1 Safety et Liveness 45
4.2.2 Unless, leads-to, initially 48
4.3 Un principe de composition 52

5 Construction de programmes **57**
5.1 Safety 57
5.2 Liveness 58
5.3 Systèmes hétérogènes 58
5.4 Application 61
5.4.1 La situation 61
5.4.2 Première spécification des agents 61
5.4.3 Spécification globale 64
5.4.4 Dérivation 64
5.4.5 Expression formelle des conditions 66
5.4.6 Spécifications complètes et implémentation 69

IV Conclusion **75**

6 Conclusions **77**
6.1 Travail effectué 77
6.2 Travaux futurs 77

V Annexes **79**

A Utilisation de Java-RMI **81**

B Preuve de la dérivation de l'application **83**

C Code $\mathcal{L}_\Psi$ **85**

D Code de l'application PingPong **109**

E Code de l'application Chantier **113**

Bibliographie **131**

Table des figures **133**

Index **135**

Première partie

Langage

Chapitre 1

Définition du langage $\mathcal{L}_\Psi$

1.1 Les langages de coordination

Ces dernières années, un intérêt grandissant est apparu pour les langages de coordination ainsi que les techniques de conception de logiciels à base de composants.

Certes, l'idée de résoudre un problème en le décomposant en sous-problèmes n'est pas neuve tout comme la méthodologie associée de programmation consistant à coder différentes procédures et à les agencer dans un corps principal. En outre, force est aussi de constater que des mécanismes tels que le "pipe" de Unix offrent depuis longtemps des moyens de composer des applications existantes.

Toutefois, ce qui est nouveau, c'est la perception de la nécessité, d'une part, de séparer clairement les concepts de calcul et d'interaction et, d'autre part, d'étudier des formes d'interactions. D. Gelernter et N. Carriero furent les premiers à mettre ces besoins en évidence en proposant en [18] l'équation suivante :

$$\text{Programming} = \text{Computation} + \text{Coordination}.$$

Ce travail fait suite à la proposition du premier langage de coordination appelé Linda ([11]). Depuis, de nombreuses extensions ont été proposées : Bauhaus Linda ([22]), Bonita ([25]), Gamma ([6]), Jada ([14]), KLAIM ([23]), Laura ([28]), Linear Objects ([4]), Manifold ([5]), *μLog* ([20]), PoliS ([13]), Shared Prolog ([7]), ...

Le langage Linda fut présenté comme envisageant la communication entre agents concurrents au moyen d'un espace partagé de tuples accessibles au moyen d'un ensemble de quatre primitives. En plus d'une primive permettant de créer un processus, cet ensemble inclut une primitive `out` pour ajouter un tuple, une primitive `in` pour enlever un tuple ainsi qu'une primitive `rd` pour tester l'existence d'un tuple.

Ces primitives sont, en fait, fort proches du paradigme de programmation concurrente par contraintes proposé par V. Saraswat en [26]. En effet, l'objectif de ce paradigme était de concevoir une communication et une synchronisation entre agents sur base du partage d'informations. Une primitive `tell` est ainsi proposée pour produire une information et une primitive `ask` pour tester l'existence de l'information.

En se basant sur cette analogie, A.Brogi et J.-M. Jacquet ont modélisé en [9] le langage Linda comme le langage $\mathcal{L}_L$(tell, ask, get, nask) généré par la grammaire de la figure 1.1.

$$A ::= C \mid A; A \mid A \parallel A \mid A + A$$
$$C ::= tell(t) \mid ask(t) \mid get(t) \mid nask(t)$$

FIG. 1.1 – Grammaire du langage $\mathcal{L}_L$(tell, ask, get, nask)

$$(T) \quad \langle tell(t) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \cup \{t\} \rangle$$

$$(A) \quad \langle ask(t) \mid \sigma \cup \{t\} \rangle \longrightarrow \langle E \mid \sigma \cup \{t\} \rangle$$

$$(N) \quad \frac{t \notin \sigma}{\langle nask(t) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \rangle}$$

$$(G) \quad \langle get(t) \mid \sigma \cup \{t\} \rangle \longrightarrow \langle E \mid \sigma \rangle$$

$$(S) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A \ ; \ B \mid \sigma \rangle \longrightarrow \langle A' \ ; \ B \mid \sigma' \rangle}$$

$$(P) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{l} \langle A \parallel B \mid \sigma \rangle \longrightarrow \langle A' \parallel B \mid \sigma' \rangle \\ \langle B \parallel A \mid \sigma \rangle \longrightarrow \langle B \parallel A' \mid \sigma' \rangle \end{array}}$$

$$(C) \quad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{l} \langle A \ + \ B \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle \\ \langle B \ + \ A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle \end{array}}$$

FIG. 1.2 – Règles de transition associées à la grammaire $\mathcal{L}_L$(tell, ask, get, nask)

Cette grammaire définit en fait les termes d'une algèbre dont les actions de base sont les primitives de communication C générées par la seconde règle. Les constructions A générées par la première règle sont les agents qui constituent le langage $\mathcal{L}_L$(tell, ask, get, nask).

Les règles de transition associées aux primitives sont formulées dans les règles T, A, N et G de la figure 1.2. La règle T exprime que l'agent atomique $tell(t)$ peut être exécuté quel que soit le contenu σ de l'espace de données et que son exécution se solde par l'ajout du jeton t au contenu de σ. Les règles A et N expriment que les agents atomiques $ask(t)$ et $nask(t)$ peuvent être exécutés respectivement si le jeton t est présent et absent dans l'espace de données et que leur exécution se termine sans en modifier le contenu. La règle G exprime que l'agent atomique $get(t)$ peut être exécuté si l'espace de données contient une occurence de t et que cette occurence de t est effacée de l'espace de données résultant. Enfin, le symbole E désigne l'agent vide.

La première règle de la grammaire de la figure 1.1 permet la construction d'agents qui font intervenir les opérateurs séquentiels, parallèles et de choix. Leur signification est exprimée par les règles S, P et C de la figure 1.2

1.2 Le Langage $\mathcal{L}_\Psi$

Le langage de coordination que nous nous proposons de considérer dans la suite de ce mémoire repose aussi sur le partage d'informations sur base de tuples. Toutefois, afin de faciliter la sélection d'informations et la définition d'informations partielles, nous enrichirons les composants des tuples d'une structure binaire permettant d'exprimer le nom du composant ainsi que sa valeur. De tels tuples sont appelés ψ-termes. Il est ici intéressant de relever que, bien qu'initialement proposés en [1], ils conviennent parfaitement bien pour modéliser les standards popularisés par le web tels que les pages html et xml.

Suite au langage $\mathcal{L}_L$, l'accès aux données du tableau s'effectue à l'aide d'un ensemble de primitives qui permettent de déposer un ψ-terme, de tester l'absence, de chercher ou de saisir un ψ-terme correspondant à un ψ-terme donné. Le langage $\mathcal{L}_\Psi$ repose sur ces primitives et est muni des opérateurs de composition séquentielle, parallèle et de choix.

Posons formellement les définitions des objets utiles à la présentation du langage $\mathcal{L}_\Psi$.

Définition 1.1 *Un ψ-terme est composé d'un foncteur et d'un ensemble d'arguments $\{champ_i = valeur_i : i = 1 \ldots n\}$ où*
- *$valeur_i$ désigne un entier, une chaîne de caractères, un ψ-terme ou une variable,*
- *les $champ_i$ sont tous distincts,*
- *les éventuelles variables sont toutes distinctes.*

Il est représenté par une expression de la forme

$$foncteur(champ_1 = valeur_1, ..., champ_n = valeur_n).$$

Par extension, nous parlerons du ψ-terme $f(ch_1 = v_1, ..., ch_n = v_n)$ pour désigner le ψ-terme représenté par cette expression. Un ψ-terme est dit clos s'il ne contient aucune variable.

Deux ψ-termes sont égaux s'ils ont même foncteur et même ensemble d'arguments. La suite de notre propos nous amènera, étant donné un ψ-terme, à rechercher un ψ-terme clos auquel il puisse être égalé moyennant une affectation de valeurs à ses variables. Nous introduisons ici la notion plus générale de correspondance.

Définition 1.2 *Soient $\psi_1 = f(ch_1 = v_1, \ldots, ch_n = v_n)$ et $\psi_2 = f'(ch'_1 = v'_1, \ldots, ch'_{n'} = v'_{n'})$ deux ψ-termes ne possédant aucune variable commune. Nous dirons que ψ_1 correspond à ψ_2 si*
- *les foncteurs f et f' sont identiques,*
- *$\{ch_i : i = 1 \ldots n\} \subseteq \{ch'_i : i = 1 \ldots n'\}$,*
- *pour chaque valeur v_i qui est un entier ou une chaîne de caractères, si $ch_i = ch'_j$ alors $v_i = v'_j$,*
- *pour chaque valeur v_i qui est un ψ-terme, si $ch_i = ch'_j$ alors v_i correspond à v'_j.*

Notons qu'avec cette définition deux ψ-termes égaux se correspondent. Dans la suite de notre travail, nous n'envisagerons cette relation asymétrique qu'avec un second argument clos. La question de la correspondance équivaut alors à celle de l'existence d'une affectation de valeurs aux variables qui rende égaux les ψ-termes considérés.

Si μ désigne l'affectation de valeurs qui associe aux variables du ψ-terme ψ les valeurs qui le rendent égal au ψ-terme clos ψ_c et $\perp$ aux autres variables, nous noterons μ par $\psi \lhd \psi_c$.

Définition 1.3 *L'ensemble des ψ-termes est appelé Spsiterme, l'ensemble des ψ-termes clos, Spsitermeclos et l'ensemble des multi-ensembles finis de ψ-termes clos est désigné par Setat.*

Muni de ces notions, nous pouvons définir le langage $\mathcal{L}_\Psi$ au moyen de la grammaire suivante.

Définition 1.4 *Si ψ désigne un ψ-terme, ψ_c un ψ-terme clos et instr une instruction primitive du langage ne dialoguant pas avec le tableau, le langage $\mathcal{L}_\Psi$ est l'ensemble des agents A définis par*

$$c ::= tell(\psi_c) \mid nask(\psi) \mid ask(\psi) \mid get(\psi)$$

$$A ::= c \mid A; A \mid A \parallel A \mid A + A \mid instr$$

Nous appellerons $Svar$ l'ensemble des variables qui apparaissent dans les ψ-termes de $\mathcal{L}_\Psi$. Nous ne faisons aucune hypothèse sur la syntaxe de l'instruction *instr*, le langage auquel elle appartient peut être impératif (Pascal, C,...), orienté objet (Pascal objet, Java,...) ou déclaratif (Prolog, Lisp,...). La seule contrainte est que les instructions de ce type n'interagissent pas avec l'espace de données partagé et ne modifient pas les valeurs des variables de $Svar$.

Définition 1.5 *Nous désignerons par Saffect l'ensemble des affectations de valeurs aux variables, une affectation de valeur θ aux variables de $Svar$ étant une fonction*

$$\theta : Svar \rightarrow (N \cup Sstring \cup Spsitermeclos \cup \{\perp\})$$

où Sstring désigne l'ensemble des chaînes de caractères.

L'ensemble *Saffect* est muni d'un ordre partiel $\prec$. Pour toute paire (θ, μ) d'éléments de *Saffect*, on a $\theta \prec \mu$ si pour toute variable x, $\theta(x) = \perp$ lorsque $\mu(x) = \perp$

On définit sur *Saffect* une loi de composition. Si θ et μ sont deux affectations de valeurs aux variables de $Svar$, $\theta\mu$ désigne l'affectation de valeurs qui associe à chaque variable la même valeur que μ si elle est distincte de $\perp$ et lui conserve la valeur que lui attribuait θ sinon. Elle est définie par

$$\theta\mu(x) = \begin{cases} \mu(x) & \text{si } \mu(x) \neq \perp \\ \theta(x) & \text{si } \mu(x) = \perp \end{cases}$$

Les instructions étant autorisées à consulter et utiliser les valeurs des variables de $Svar$, il se peut que leur exécution dépendent de l'affectation de valeurs à ces variables. Pour en tenir compte nous indroduisons le prédicat *executable*.

Définition 1.6 *Si Sinstr désigne l'ensemble des instructions, nous désignons par executable le prédicat défini sur les couples $(instr, \theta)$ de $Sinstr \times Saffect$ qui obtient la valeur vrai si l'instruction instr peut être exécutée sur θ, et faux sinon.*

Un agent obtenu par composition parallèle ne sera exécutable que si les agents qui le composent n'agissent que sur des ensembles distincts de variables. Ceci nous amène à définir la fonction Var_a qui associe à un agent l'ensemble des variables qu'il est susceptible de modifier.

Définition 1.7 *La fonction* $Var_a : \mathcal{L}_\psi \to \mathcal{P}(Svar)$ *est définie inductivement par*

- $Var_a(tell(\psi_c)) = Var_a(nask(\psi)) = Var_a(instr) = \emptyset$,
- $Var_a(ask(\psi)) = Var_a(get(\psi)) = \{v : v$ *variable apparaissant dans* $\psi\}$,
- $Var_a(A;B) = Var_a(A+B) = Var_a(A) \cup Var_a(B)$
- $Var_a(A \parallel B) = \begin{cases} \emptyset & si\ Var_a(A) \cap Var_a(B) \neq \emptyset \\ Var_a(A) \cup Var_a(B) & si\ Var_a(A) \cap Var_a(B) = \emptyset. \end{cases}$

Les primitives $tell$, $nask$ et $instr$ ne modifient pas l'affectation de valeurs aux variables. On attend des primitives ask et get qu'elles attribuent aux variables les valeurs qui vont faire coïncider leur argument avec le ψ-terme du tableau auquel il correspond, les valeurs des variables qui apparaissent dans leur argument sont donc modifiées. La composition séquentielle ou alternative de deux agents ne pose pas de difficultés. L'ensemble des variables susceptibles d'être modifiées par un agent ainsi composé est assez naturellement obtenu comme l'union des ensembles associés à ses composants. La mise en parallèle d'agents agissant sur des ensembles distincts de variables se comporte de la même façon. Si l'on tente de mettre en parallèle des agents modifiant des variables communes, l'agent résultant ne sera pas exécutable, c'est pourquoi l'ensemble vide de variables lui est associé.

Les transitions correspondant à l'exécution d'un agent entraîne la modification, à la fois de l'agent lui-même, de l'affectation de valeurs aux variables et de l'état du tableau. Nous appelons configuration un triplet formé d'un agent, d'une affectation de valeurs et d'un état du tableau. Les règles de transition expriment comment s'effectue le passage d'une configuration à une autre lors de l'exécution d'un agent.

Définition 1.8 *Une configuration est un triple* $\langle A \,\&\, \theta \mid \sigma \rangle$ *où* A *désigne un agent,* θ *une attribution de valeurs aux variables de* $Svar$ *et* σ *un tableau.*

L'exécution des instructions est définie par les règles de transition de la figure 1.3 où E est un symbole de terminaison aussi appelé agent vide. Les expressions $A \parallel E$, $E \parallel A$ et $E;A$ sont remplacées par A.

La primitive $tell(\psi)$ permet de placer un ψ-terme clos dans le multi-ensemble de ψ-termes σ. La primitive $nask(\psi)$ s'exécute en l'absence de tout ψ-terme clos correspondant à son argument et ne modifie ni les attributions de valeurs aux variables ni le contenu de l'espace partagé. Les primitives $ask(\psi)$, et $get(\psi)$ s'exécutent si l'espace partagé possède un ψ-terme clos ψ' correspondant à ψ, elles modifient l'ensemble des affectations en $\theta\mu$ avec μ l'affectation de valeurs qui établit la correspondance, l'instruction ask laisse inchangé l'espace partagé, tandis que l'instruction get y supprime une occurence de ψ'.

La règle P impose que les agents mis en parallèle ne partagent aucune variable. Aucune règle de transition n'est définie pour des agents parallèles ayant des variables communes.

$$(T) \quad \frac{clos(\psi_c)}{\langle tell(\psi_c)\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle E\ \&\ \theta \mid \sigma \cup \{\psi_c\}\rangle}$$

$$(A) \quad \frac{\psi \lhd \psi_c = \mu}{\langle ask(\psi)\ \&\ \theta \mid \sigma \cup \{\psi_c\}\rangle \longrightarrow \langle E\ \&\ \theta\mu \mid \sigma \cup \{\psi_c\}\rangle}$$

$$(N) \quad \frac{\nexists \psi_c : \psi_c \in \sigma, \psi \text{ correspond à } \psi_c}{\langle nask(\psi)\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle E\ \&\ \theta \mid \sigma\rangle}$$

$$(G) \quad \frac{\psi \lhd \psi_c = \mu}{\langle get(\psi)\ \&\ \theta \mid \sigma \cup \{\psi_c\}\rangle \longrightarrow \langle E\ \&\ \theta\mu \mid \sigma\rangle}$$

$$(S) \quad \frac{\langle A\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle A'\ \&\ \theta' \mid \sigma'\rangle}{\langle (A\ ;\ B)\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle (A'\ ;\ B)\ \&\ \theta' \mid \sigma'\rangle}$$

$$(P) \quad \frac{\langle A\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle A'\ \&\ \theta' \mid \sigma'\rangle, Var_a(A) \cap Var_a(B) = \emptyset}{\begin{array}{l}\langle (A \parallel B)\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle (A' \parallel B)\ \&\ \theta' \mid \sigma'\rangle \\ \langle (B \parallel A)\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle (B \parallel A')\ \&\ \theta' \mid \sigma'\rangle\end{array}}$$

$$(C) \quad \frac{\langle A\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle A'\ \&\ \theta' \mid \sigma'\rangle}{\begin{array}{l}\langle (A\ +\ B)\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle A'\ \&\ \theta' \mid \sigma'\rangle \\ \langle (B\ +\ A)\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle A'\ \&\ \theta' \mid \sigma'\rangle\end{array}}$$

$$(I) \quad \frac{executable(instr, \theta)}{\langle instr\ \&\ \theta \mid \sigma\rangle \longrightarrow \langle E\ \&\ \theta \mid \sigma\rangle}$$

FIG. 1.3 – Règles de transition associées à la grammaire $\mathcal{L}_\Psi$

Chapitre 2

Implémentation du langage $\mathcal{L}_\Psi$ en Java

Nous proposons ici une implémentation Java-RMI du langage exposé au chapitre précédent. La perspective adoptée est similaire à celle suivie par P. Ciancarini et D. Rossi dans leur définition du langage Jada [15]. Il ne s'agit pas d'augmenter la syntaxe de Java mais de développer des classes qui permettront une utilisation transparente des primitives tell, ask, get et nask. Notre travail se distingue de celui réalisé par les auteurs de Jada au niveau des objets déposés dans l'espace partagé et du choix des primitives considérées. L'implémentation s'articule autour de trois classes d'objets clés. La classe `PsiTermeSpaceImpl` représente l'espace de données partagé, la classe `PsiTerme` représente les ψ-termes et la classe `Affectation` représente les affectations de valeurs aux variables. Ces deux dernières classes utilisent respectivement les classes techniques `Item` et `Variable`. Les sections suivantes présentent les classes `Affectation`, `PsiTerme` et `PsiTermeSpace`. Le chapitre se termine par une application simple permettant de tester notre implémentation. La mise en oeuvre pratique de RMI s'est appuyée sur le cours de V. Englebert [17] et les documents de K. Noben [24]

2.1 Affectation

Une affectation est vue comme un ensemble de couples composés du nom de la variable et de la valeur de la variable. Seules sont représentées les variables qui ont une valeur significative (c'est-à-dire distincte de $\perp$). Les principales méthodes ont pour but d'ajouter/modifier une variable, ajouter un ensemble de variables et récupérer la valeur d'une variable.

Puisqu'une affectation doit pouvoir être transmise comme résultat d'une méthode (ask ou get) appelée sur le tableau, objet distant, la classe `Affectation` implémente l'interface `Serializable`.

Remarquons que d'un point de vue théorique, nous avons considéré l'affectation de valeurs aux variables comme un élément global au même titre que le tableau. Toutefois, nous avons imposé que des agents exécutés en parallèles ne partagent aucune variable. Ceci correspond à considérer les variables comme marquées par l'agent au sein duquel elles

sont utilisées. De façon pratique, au niveau de l'implémentation, nous maintiendrons avec chaque agent la partie de l'affectation qui le concerne. Ceci ne modifie en rien l'expressivité mais évite de nombreux accès distants.

Les méthodes publiques que nous utilisons pour gérer les variables et leurs valeurs sont les suivantes :

- `public void effacerVar(String)`
  Précondition. Cette Affectation représente une affectation de valeurs à des variables v_1,...,v_n. Le String argument est le nom d'une variable x.
  Postcondition. Si x est le nom d'une des variables v_1,..., v_n, la représentation de cette variable et sa valeur sont effacées de cette Affectation. Sinon, rien n'est modifié.
- `public void ajouterVar(String, String)`
  Précondition. Cette Affectation représente une affectation de valeurs à des variables v_1,...,v_n. Les String arguments représentent respectivement le nom d'une variable x et sa valeur vx.
  Postcondition. Si x est le nom d'une des variables v_1,..., v_n, la valeur de cette variable est remplacée par vx dans cette Affectation. Sinon, le couple (x, vx) est ajouté à cette Affectation.
- `public void ajouterVar(String, Integer)`
  Précondition. Cette Affectation représente une affectation de valeurs à des variables v_1,...,v_n. Le String et l'Integer arguments représentent respectivement le nom d'une variable x et sa valeur vx.
  Postcondition. Si x est le nom d'une des variables v_1,..., v_n, la valeur de cette variable est remplacée par vx dans cette Affectation. Sinon, le couple (x, vx) est ajouté à cette Affectation.
- `public void ajouterVar(String, PsiTerme)`
  Précondition. Cette Affectation représente une affectation de valeurs à des variables v_1,...,v_n. Le String et le PsiTerme arguments représentent respectivement le nom d'une variable x et sa valeur vx.
  Postcondition. Si x est le nom d'une des variables v_1,..., v_n, la valeur de cette variable est remplacée par vx dans cette Affectation. Sinon, le couple (x, vx) est ajouté à cette Affectation.
- `public void ajouterEnsemble(Affectation)`
  Précondition. Cette Affectation et l'Affectation argument représentent respectivement les affectations θ et μ.
  Postcondition. Cette Affectation représente l'affectation de valeurs $\theta\mu$.
- `public Integer valeurInteger(String x)`
  Précondition. Cette Affectation représente l'affectation θ, le String argument est une variable x pour laquelle $\theta(x)$ est un entier.
  Postcondition. Renvoie $\theta(x)$.
- `public String valeurString(String x)`
  Précondition. Cette Affectation représente l'affectation θ, le String argument est une variable x pour laquelle $\theta(x)$ est une chaîne de caractères.
  Postcondition. Renvoie $\theta(x)$.
- `public PsiTerme valeurPsiTerme(String x)`
  Précondition. Cette Affectation représente l'affectation θ, le String argument est une variable x pour laquelle $\theta(x)$ est un ψ-terme.

Postcondition. Renvoie $\theta(x)$.
- `public void affiche()`
  Précondition. Cette Affectation représente l'affectation θ.
  Postcondition. Affiche chaque nom de variable suivi de sa valeur.

2.2 PsiTerme

Les ψ-termes, l'information partagée, seront implémentés par des objets de la classe `PsiTerme`. Outre le constructeur, cette classe contient les méthodes permettant, de tester si un ψ-terme est clos, de récupérer la valeur d'un champ d'un ψ-terme clos et de tester si un ψ-terme correspond à un ψ-terme clos.

Puisqu'un ψ-terme doit pouvoir être transmis comme argument à une méthode appelée sur le tableau, objet distant, la classe `PsiTerme` implémente l'interface `Serializable`. Cela dégage le constructeur et les méthodes publiques suivants.

Constructeur
- `PsiTerme(String)`
  Précondition. La chaîne de caractères argument exprime un ψ-terme. Une chaîne de caractères exprime le ψ-terme $foncteur(champ_1 = valeur_1, ..., champ_n = valeur_n)$ si elle a pour préfixe cette expression dans laquelle les noms des variables sont précédés par un " ?".
  Postcondition. Ce PsiTerme a été créé et représente le ψ-terme exprimé par la chaîne de caractères argument.

Méthodes publiques
- `public boolean estClos ()`
  Précondition. Ce PsiTerme représente un ψ-terme.
  Postcondition. Renvoie vrai si le ψ-terme représenté est clos, faux sinon.
- `public Integer recupererEntier(String)`
  Précondition. Ce PsiTerme représente un ψ-terme ayant un champ représenté par la chaîne de caractères argument et possédant une valeur de type entier.
  Postcondition. Renvoie l'entier valeur du champ du ψ-terme représenté dont le nom est la chaîne de caractères argument.
- `public String recupererString(String)`
  Précondition. Ce PsiTerme représente un ψ-terme ayant un champ représenté par la chaîne de caractères argument et possédant une valeur de type chaîne de caractères.
  Postcondition. Renvoie la chaîne de caractères valeur du champ du ψ-terme représenté dont le nom est la chaîne de caractères argument.
- `public String recupererPsiTermeClos(String)`
  Précondition. Ce PsiTerme représente un ψ-terme ayant un champ représenté par la chaîne de caractères argument et possédant une valeur de type ψ-terme.
  Postcondition. Renvoie le PsiTerme représentant le ψ-terme valeur du champ du ψ-terme représenté dont le nom est la chaîne de caractères argument.
- `public boolean correspond(PsiTerme, Affectation)`
  Précondition. Ce PsiTerme et le PsiTerme argument représentent respectivement le

ψ-terme ϕ et le ψ-terme clos ψ_c, l'Affectation argument représente θ une affectation de valeurs à des variables.
Postcondition. Si $\phi \lhd \psi_c = \mu$, la méthode renvoie vrai et L'Affectation représente $\theta\mu$.
Si ϕ ne correspond pas à ψ_c, la méthode renvoie faux et l'Affectation est inchangée.

- `public String psiTermeToString()`
  Précondition. Ce PsiTerme représente un ψ-terme.
  Postcondition. Renvoie une chaîne de caractères qui exprime le ψ-terme représenté.

2.3 PsiTermeSpace

L'espace de données partagé est un multi-ensemble de ψ-termes, en quelque sorte le tableau sur lequel l'information est déposée, mise à disposition. Du point de vue de l'implémentation RMI, il s'agit d'un objet distant sur lequel les primitives tell, ask, nask et get vont être invoquées. Nous définissons donc d'une part, l'interface `TachePsiTermeSpace` qui étend `Remote` et d'autre part, la classe `PsiTermeSpaceImpl` qui implémente cet interface.

TachePsiTermeSpace

- void tell(PsiTerme PTC) throws RemoteException ;
- Affectation ask(PsiTerme PT) throws RemoteException ;
- void nask(PsiTerme PT) throws RemoteException ;
- Affectation get(PsiTerme PT) throws RemoteException ;

PsiTermeSpaceImpl. Constructeur

- `PsiTermeSpaceImpl()`
  Postcondition. Ce PsiTermeSpaceImpl a été créé et représente l'espace de ψ-termes vide.

Méthodes publiques

Nous souhaitons que les accès au PsiTermeSpace n'interfèrent pas entre eux, ce qui nous permet de considérer les primitives tell, ask, nask, get comme des actions atomiques. Ceci nous amène à déclarer *synchronized* les méthodes qui consultent ou modifient le PsiTermeSpace.

- `public synchronized void tell(PsiTerme)`
  Précondition. Ce PsiTermeSpaceImpl représente un espace de ψ-termes $\mathcal{E}$. Le PsiTerme argument représente un ψ-terme pt.
  Postcondition. Si pt est clos, ce PsiTermeSpaceImpl représente l'espace de ψ-termes $\mathcal{E} \uplus \{pt\}$ sinon il est inchangé.
- `public synchronized Affectation ask(PsiTerme)`
  Précondition. Ce PsiTermeSpaceImpl représente un espace de ψ-termes, le PsiTerme argument représente un ψ-terme ϕ.
  Postcondition. Ne modifie pas le contenu de ce PsiTermeSpaceImpl. Renvoie l'Affectation représentant μ dès qu'elle trouve un PsiTerme de ce PsiTermeSpaceImpl représentant un ψ-terme tel que $\phi \lhd \psi = \mu$ (se suspend en attendant d'en trouver un).

- `public synchronized void nask(PsiTerme)`
  Précondition. Ce PsiTermeSpaceImpl représente un espace de ψ-termes, le PsiTerme argument représente un ψ-terme ϕ.
  Postcondition. Ne modifie pas le contenu de ce PsiTermeSpaceImpl. Il n'y a pas de PsiTerme de ce PsiTermeSpaceImpl représentant un ψ-terme correspondant à ϕ (se suspend tant qu'il y en a un).
- `public synchronized PsiTerme get(PsiTerme)`
  Précondition. Ce PsiTermeSpaceImpl représente un espace de ψ-termes, le PsiTerme argument représente un ψ-terme ϕ.
  Postcondition. Renvoie l'Affectation représentant μ dès qu'elle trouve un PsiTerme de ce PsiTermeSpaceImpl représentant un ψ-terme tel que $\phi \lhd \psi = \mu$ (se suspend en attendant d'en trouver un). Supprime une représentation de ψ du PsiTermeSpaceImpl.
- `public static void main (String)`
  Précondition. L'annuaire sur le serveur est lancé.
  Postcondition. Un objet PsiTermeSpaceImpl représentant le tableau vide est initialisé et référencé sous le nom "MonTableau" dans l'annuaire du serveur.

2.4 Application

Nous proposons ici une application simple dont le seul objet est de tester et d'illustrer notre implémentation. Il s'agit de trois processus exécutés en parallèle. Le premier *PingPong* a pour but l'initialisation du tableau. Ensuite *Ping* et *Pong* se transmettent successivement une information via le tableau. Le code complet de ces agents se trouve en annexe. Afin d'illustrer l'utilisation des primitives spécifiées plus haut, nous n'exposons ici que la partie significative du code de ces agents.

Agent PingPong.

```
Registry repertoire;
TachePsiTermeSpace un_serveur;
Affectation AffectPong;

try{  repertoire =(Registry)LocateRegistry.getRegistry("localhost");
      un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
      un_serveur.tell(new PsiTerme("PingPong(etat=Ping,numero=12)"));
      }
catch (RemoteException e){e.printStackTrace();}
catch (NotBoundException e){e.printStackTrace();}
```

Agent Ping.

```
Registry repertoire;
TachePsiTermeSpace un_serveur;
Affectation AffectPing;

try{  repertoire = (Registry)LocateRegistry.getRegistry("localhost");
```

```
      un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
      while(true){
            AffectPing=un_serveur.get(new PsiTerme(
                                "PingPong(etat=Pong,numero=?x)"));
            int cptr= AffectPing.valeurInteger("x").intValue();
            String message=new String("PingPong(etat=Ping,numero=")
                 .concat(new Integer(cptr+1).toString()).concat(")");
            un_serveur.tell(new PsiTerme(message));
            System.out.println(message);
            AffectPing.affiche();
            try{
                Thread.sleep(1000);
                }
            catch (Exception e){};
            }
      }
catch (RemoteException e){e.printStackTrace();}
catch (NotBoundException e){e.printStackTrace();}
```

Agent Pong.

```
Registry repertoire;
TachePsiTermeSpace un_serveur;
Affectation AffectPong;

try{  repertoire = (Registry)LocateRegistry.getRegistry("localhost");
      un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
      while(true){
            AffectPong=un_serveur.get(new PsiTerme(
                                "PingPong(etat=Ping,numero=?x)"));
            int cptr= AffectPong.valeurInteger("x").intValue();
            String message=new String("PingPong(etat=Pong,numero=")
                 .concat(new Integer(cptr+1).toString()).concat(")");
            un_serveur.tell(new PsiTerme(message));
            System.out.println(message);
            AffectPong.affiche();
            try{
                Thread.sleep(1000);
                }
            catch (Exception e){};
            }
      }
catch (RemoteException e){e.printStackTrace();}
catch (NotBoundException e){e.printStackTrace();}
```

Les manipulations à effectuer pour tester cette application sont décrites dans l'annexe.

Une autre application sera développée au dernier chapitre en illustration de la méthode de développement que nous y proposerons.

Deuxième partie

Sémantique

Chapitre 3

Sémantique

Ce chapitre est consacré à une étude sémantique du langage $\mathcal{L}_\Psi$. Dans un premier temps, nous proposons une sémantique historique. Elle s'appuie sur les résultats de [10] et les adapte pour prendre en compte non seulement l'état du tableau mais aussi l'affectation de valeurs aux variables. La seconde étape de notre démarche consiste à fournir une sémantique complètement abstraite pour le langage $\mathcal{L}_\Psi$. Les définitions des notions de compositionalité et d'abstraction complète sont exposées par J.-M. Jacquet dans [19]. La définition de la sémantique complètement abstraite pour la sémantique historique exposée à la section 3.1. adapte au langage $\mathcal{L}_\Psi$ la démarche proposée par [10]. La prise en compte de l'affectation de valeurs dans les histoires introduit une série de difficultés propres dans les démonstrations, notamment celle de l'abstraction complète.

3.1 Sémantique historique

Nous avons exposé au chapitre 1.2 la syntaxe du langage $\mathcal{L}_\Psi$ et les règles de transition définissant sa sémantique opérationnelle. La sémantique historique proposée dans [10] pour $\mathcal{L}_L$, s'intéresse aux suites d'états produites par les exécutions d'agents. Pour notre langage $\mathcal{L}_\Psi$, outre les états, nous prenons en compte les affectations de valeurs aux variables. C'est pourquoi nous définissons la notion de situation, paire formée d'un état et d'une affectation de valeurs, et considérons les suites de situations.

Définition 3.1 *Nous appellerons situation une paire (θ, σ) formée d'une affectation de valeurs θ et d'un état σ. L'ensemble des situations sera désigné par $Ssituation$. Par suite,*

$$Ssituation = Saffect \times Setat$$

Etant donné un ensemble A, nous notons $A^{<\omega}$ l'ensemble de suites finies d'éléments de A. La concaténation de deux suites finies s_1 et s_2 est notée $s_1.s_2$ et si S_1 et S_2 sont deux ensembles de suites finies nous notons

$$S_1.S_2 \;=\; \{s_1.s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\}.$$

Définition 3.2 *Soient les symboles δ^+ et δ^- désignant respectivement le succès et l'échec. L'ensemble des histoires $Shist$ est l'ensemble des suites finies de situations se terminant*

par l'un des symboles de terminaison.

$$Shist = Ssituations^{<\omega}.\{\delta^+, \delta^-\}$$

Définition 3.3 *La sémantique historique $\mathcal{O}_h : \mathcal{L}_\psi \to \mathcal{P}(Shist)$ est la fonction suivante. Pour tout agent A,*

$$\begin{aligned}
&\mathcal{O}_h(A) = \\
&\{(\theta_0,\sigma_0).\cdots.(\theta_n,\sigma_n).\delta^+ : \quad \langle A_0 \,\&\, \theta_0 \mid \sigma_0\rangle \to \langle A_1 \,\&\, \theta_1 \mid \sigma_1\rangle \to \cdots \langle A_n \,\&\, \theta_n \mid \sigma_n\rangle, \\
&\qquad\qquad A_0 = A, \theta_0 = \emptyset, \sigma_0 = \emptyset, A_n = E, n \geq 0\} \\
&\cup \\
&\{(\theta_0\sigma_0).\cdots.(\theta_n,\sigma_n).\delta^- : \quad \langle A_0 \,\&\, \theta_0 \mid \sigma_0\rangle \to \langle A_1 \,\&\, \theta_1 \mid \sigma_1\rangle \to \cdots \langle A_n \,\&\, \theta_n \mid \sigma_n\rangle \not\to, \\
&\qquad\qquad A_0 = A, \theta_0 = \emptyset, \sigma_0 = \emptyset, A_n \neq E, n \geq 0\}
\end{aligned}$$

Il serait particulièrement intéressant de pouvoir connaître la sémantique d'un agent composé de deux agents reliés par un opérateur de composition séquentielle, parallèle ou de choix à partir de la sémantique de ses composants. C'est la question de la compositionalité du langage $\mathcal{L}_\psi$.

Examinons, sur base d'un exemple proposé en [10], le comportement de la sémantique historique que nous venons d'introduire. Considérons ψ et ϕ deux ψ-termes clos distincts. Nous avons :

$$\mathcal{O}_h(get(\psi)) \;=\; \mathcal{O}_h(get(\phi)) \;=\; \{(\emptyset,\emptyset).\delta^-\}$$

$$\mathcal{O}_h(tell(\psi)) \;= \{(\emptyset,\emptyset).(\emptyset,\{\psi\}).\delta^+\}$$

alors que

$$\begin{aligned}
\mathcal{O}_h(tell(\psi) \parallel get(\psi)) &= \{(\emptyset,\emptyset).(\emptyset,\{\psi\}).(\emptyset,\emptyset).\delta^+\} \\
\mathcal{O}_h(tell(\psi) \parallel get(\phi)) &= \{(\emptyset,\emptyset).(\emptyset,\{\psi\}).\delta^-\}
\end{aligned}$$

Il s'en suit que la sémantique $\mathcal{O}_h$ n'est pas compositionnelle.

3.2 Sémantiques compositionnelles

Considérons un langage L défini à partir d'expressions de base au moyen d'un ensemble d'opérateurs $\{\diamond_1, \cdots, \diamond_m\}$. En d'autres termes, les expressions du langage considéré sont définies récursivement de la façon suivante :

- toute expression de base est une expression.
- si s et t sont des expressions, alors, pour tout i, $s \diamond_i t$ est une expression.

Définition 3.4 *Soit $M : L \to S$ un modèle sémantique qui associe à chaque expression du langage L un membre d'un certain domaine sémantique S. Le modèle M est dit compositionnel s'il existe des opérateurs $\widetilde{\diamond}_1$, ..., $\widetilde{\diamond}_m$ tels que, pour toutes expressions s, t et tout i, on a*

$$M(s \diamond_i t) = M(s) \widetilde{\diamond}_i M(t).$$

La compositionalité d'un modèle sémantique peut être vérifiée au moyen de la notion de relation de congruence.

Définition 3.5 *Une relation d'équivalence sur L, disons $\equiv$, respecte l'opérateur $\diamond_i$ si pour toutes expressions s_1, s_2, t_1 et t_2, vérifiant $s_1 \equiv s_2$ et $t_1 \equiv t_2$, on a*

$$(s_1 \diamond_i t_1) \equiv (s_2 \diamond_i t_2).$$

Une relation d'équivalence est appelée relation de congruence sur L si elle respecte tous les opérateurs $\diamond_1, \ldots, \diamond_m$.

Les relations de congruence posssèdent la caractérisation suivante.

Théorème 3.1 *Désignons par $\equiv_M$ la relation d'équivalence induite par M, c'est-à-dire l'équivalence définie par*

$$s \equiv_M t \text{ ssi } M(s) = M(t)$$

quelles que soient les expressions s et t. Alors M est compositionnel ssi $\equiv_M$ est une relation de congruence sur L.

Preuve. La condition suffisante affirme que si les expressions s_1, s_2, t_1, t_2 vérifient $s_1 \equiv_M s_2$ et $t_1 \equiv_M t_2$ alors on a $(s_1 \diamond_i t_1) \equiv_M (s_2 \diamond_i t_2)$ pour chacun des opérateurs $\diamond_i$. Vérifier cette propriété revient à comparer $M(s_1 \diamond_i t_1)$ et $M(s_2 \diamond_i t_2)$. M étant compositionnel, on dispose pour chaque opérateur $\diamond_i$ de L d'un opérateur $\widetilde{\diamond}_i$ tel que $M(s \diamond_i t) = M(s) \widetilde{\diamond}_i M(t)$. Les deux expressions à comparer correspondent donc respectivement à $M(s_1) \widetilde{\diamond}_i M(t_1)$ et $M(s_2) \widetilde{\diamond}_i M(t_2)$ qui sont égales puisque par hypothèse, $M(s_1) = M(s_2)$ et $M(t_1) = M(t_2)$.

La preuve de la condition nécessaire demande de vérifier que l'opérateur

$$\widetilde{\diamond}_i : (M(s_1), M(s_2)) \mapsto M(s_1 \diamond_i s_2)$$

est une fonction. Considérons pour cela les expressions s_1, s_2, t_1, t_2 vérifiant $M(s_1) = M(s_2)$ et $M(t_1) = M(t_2)$. La congruence de M permet d'affirmer que $(s_1 \diamond_i t_1) \equiv_M (s_2 \diamond_i t_2)$ ce qui suffit.

3.3 Notations

Les histoires que considère la sémantique opérationnelle sont des suites de situations commençant dans la situation $(\emptyset, \emptyset)$. Il y a deux raisons principales pour lesquelles cette sémantique historique n'est pas compositionnelle. Tout d'abord, l'exécution d'une transition produit un état et une attribution de valeurs qui ne sont pas nécessairement vides. Une sémantique compositionnelle doit donc considérer des états initiaux et des affectations initiales non vides. De plus, l'exécution de l'agent $A \parallel B$ consiste en l'exécution imbriquée de pas de A et de B. Une sémantique compositionnelle doit donc tenir compte de modifications de la situation dues à des transitions réalisées par l'environnement d'un agent.

C'est pourquoi la sémantique dénotationnelle $\mathcal{D}_h$ que nous allons définir envisage les histoires comme des suites de transitions. Nous modélisons les transitions par des paires de situations input et output et prenons comme domaine de la sémantique les ensembles de suites de paires de ce type. Ces séquences peuvent contenir des trous tenant compte des actions de l'environnement. De plus, elles commencent dans une situation quelconque.

Définition 3.6 *Nous appelons Shhist l'ensemble*

$$\begin{aligned} Shhist \;=\; \{h = ((\theta_1,\sigma_1),(\kappa_1,\tau_1)).\cdots.((\theta_{n-1},\sigma_{n-1}),(\kappa_{n-1},\tau_{n-1})).((\theta_n,\sigma_n),\delta) : \\ \theta_i,\kappa_i \in Saffect, \sigma_i,\tau_i \in Setat \text{ pour } i = 1,\ldots,n, \delta \in \{\delta^+,\delta^-\} \\ \text{et } \kappa_i \prec \theta_{i+1} \text{ pour } i = 1,\ldots,n-1\} \end{aligned}$$

Remarquons que dans une histoire de *Shhist* rien n'impose à l'extrémité d'un couple de situations de coïncider avec l'origine du couple suivant. Si c'est le cas, l'histoire est qualifiée de continue. Lorsque l'extrémité d'un couple ne correspond pas à l'origine du suivant, nous disons que l'histoire présente un trou.

Définition 3.7 *Une histoire $h \in Shhist$ est* continue *ssi elle est de la forme*

$$(s_0,s_1).(s_1,s_2).\cdots.(s_{n-1},s_n).(s_n,\delta)$$

avec $\delta \in \{\delta^+,\delta^-\}$. Dans ce cas, $\bar{h}$ désigne la suite suivante de situations

$$\bar{h} = s_0.s_1.\cdots.s_n.\delta$$

A une histoire h est associée sa situation initiale, $init(h)$, et trois ensembles particuliers. Tout d'abord, l'ensemble $\mathit{diff}(h)$ des ψ-termes déposés ou saisis sur le tableau par l'une des transitions de l'histoire, ensuite, l'ensemble $Var(h)$ des variables modifiées par une des transitions de l'histoire et enfin, l'ensemble $Ext(h)$ des variables dont l'histoire a besoin qu'elles soient modifiées par l'environnement pour pouvoir se dérouler.

Définition 3.8

1. *Soit h une histoire de Shhist alors,*

$$init(h) = \begin{cases} s & si\ h = (s,t).h' \\ s & si\ h = (s,\delta), \delta \in \{\delta^+,\delta^-\}. \end{cases}$$

2. *Soient $n \geq 0$, $\delta \in \{\delta^+,\delta^-\}$ et*

$$h = ((\theta_1,\sigma_1),(\kappa_1,\tau_1)).\cdots.((\theta_{n-1},\sigma_{n-1}),(\kappa_{n-1},\tau_{n-1})).((\theta_n,\sigma_n),\delta),$$

une histoire de Shhist, alors

$$\mathit{diff}(h) = (\sigma_1 \setminus \tau_1) \cup (\tau_1 \setminus \sigma_1) \cup \cdots \cup (\sigma_{n-1} \setminus \tau_{n-1}) \cup (\tau_{n-1} \setminus \sigma_{n-1})$$

où $\cup$ and $\setminus$ désignent respectivement l'union et la différence de multi-ensembles. Nous élargissons cette définition aux ensembles d'histoires de la façon suivante : pour tout ensemble S d'histoires de Shhist,

$$\mathit{diff}(S) = \bigcup\{\mathit{diff}(h) : h \in S\}$$

3. *Soit $h = ((\theta_1,\sigma_1),(\kappa_1,\tau_1)).\cdots.((\theta_{n-1},\sigma_{n-1}),(\kappa_{n-1},\tau_{n-1})).((\theta_n,\sigma_n),\delta)$, une histoire de Shhist,*

$$Var(h) = \{x \in Svar : \exists i \in \{1,\ldots,n-1\} : \kappa_i(x) \neq \theta_i(x),\}$$

$$Ext(h) = \{x \in Svar : \exists i \in \{1, \ldots, n-1\} : \theta_{i+1}(x) \neq \kappa_i(x)\}$$

Par extension, si $S \subset Shhist$, *on définit*

$$Var(S) = \bigcup_{h \in S} Var(h)$$

$$Ext(S) = \bigcup_{h \in S} Ext(h)$$

A un ensemble S d'histoires de $Shhist$, outre les ensembles que nous venons de définir, est encore associé l'ensemble $S[p]$ des fins des histoires de S qui ont comme préfixe la suite p de couples de situations. Enfin, on désigne par S^a l'ensemble des histoires de S de longueur supérieure à 1, par S^+ l'ensemble des histoires de S de longueur 1 se terminant par δ^+ et par S^- l'ensemble de ses histoires de longueur 1 se terminant par δ^-.

Définition 3.9

1. *Soient* S *un ensemble d'histoires de* $Shhist$ *et* p *une suite de* $(Ssituation \times Ssituation)^{<\omega}$, *alors*
$$S[p] = \{h : p.h \in S\}.$$
2. *Soit* S *un ensemble d'histoires de* $Shhist$, *alors*
$$\begin{aligned} S^a &= \{h : h = (s,t).h' \in S\} \\ S^+ &= \{h : h = (s, \delta^+) \in S\} \\ S^- &= \{h : h = (s, \delta^-) \in S\}. \end{aligned}$$

3.4 Sémantique dénotationnelle

Définir une sémantique compositionnelle consiste à fournir, d'une part, la signification de chacune des primitives et, d'autre part, un opérateur sémantique pour chaque opérateur syntaxique. Dans cette sous-section, nous commençons par envisager les opérateurs, et définissons ensuite une sémantique dénotationnelle $/\!\!denh$ dont nous montrons qu'elle est correcte par rapport à la sémantique opérationnelle historique. La sous-section suivante établit qu'elle est complètement abstraite.

Avant d'aborder les définitions, il importe d'observer les faits suivants. Nous avons imposé que des agents ne puissent s'exécuter en parallèle que s'ils ne partagent aucune variable. Une histoire de $Shhist$ produite par l'exécution d'un agent ne peut donc attendre de l'environnement qu'il modifie la valeur d'une des variables qu'elle-même modifie. Quel que soit l'agent A et l'histoire h de la sémantique $\mathcal{D}_h(A)$, on aura donc

$$Var(h) \cap Ext(h) = \emptyset$$

La propriété est en fait plus exigente. Aucune des histoires associées à un agent ne peut attendre qu'une variable modifiée par l'une d'entre elles soit modifiée par l'environnement, c'est-à-dire que quel que soit l'agent A, on a

$$Var(\mathcal{D}_h(A)) \cap Ext(\mathcal{D}_h(A)) = \emptyset$$

Examinons successivement les trois opérateurs de compositions séquentielle, parallèle et de choix.

1. **Composition séquentielle.** Puisque la sémantique des histoires peut inclure des trous et commencer sur n'importe quelle situation initiale, composer la signification de deux agents qui sont composés séquentiellement revient à concaténer leurs histoires. Parmi les suites ainsi obtenues, certaines pourraient violer la condition de croissance des affectations de valeurs. On ne conservera donc que celles qui sont effectivement des histoires. Ceci est réalisé par l'opérateur $\widetilde{;}$.

   Définition 3.10 *La sémantique de la composition séquentielle est définie par la fonction* $\widetilde{;} : \mathcal{P}(Shhist) \times \mathcal{P}(Shhist) \to \mathcal{P}(Shhist)$ *telle que pour tous sous-ensembles* S_1, S_2 *de* $Shhist$,

$$\begin{aligned} S_1 \widetilde{;} S_2 = \; & \Big(\{h = h_1.h_2 : \quad h_1.(s,\delta^+) \in S_1, h_2 \in S_2, \\ & \qquad\qquad Ext(h) \cap (Var(S_1) \cup Var(S_2)) = \emptyset\} \cap Shhist\Big) \\ & \cup \\ & \{h = h_1.(s,\delta^-) : \quad h \in S_1 \text{ et } Ext(h) \cap (Var(S_1) \cup Var(S_2)) = \emptyset\} \end{aligned}$$

2. **Composition parallèle.** La composition parallèle permet l'exécution imbriquée de pas d'agents parallèles. La composition parallèle de deux histoires sémantiques consiste donc à les imbriquer en veillant ici aussi à ce que le résultat soit bien une histoire.

   Définition 3.11 *La composition parallèle de deux histoires est définie par la fonction* $\widetilde{||}_h : Shhist \times Shhist \to \mathcal{P}(Shhist)$ *définie inductivement par les égalités suivantes, où* $\delta_1, \delta_2 \in \{\delta^+, \delta^-\}$.

$$\begin{aligned} (s_1,t_1).h_1 \,\widetilde{||}_h\, (s_2,t_2).h_2 &= \{(s_1,t_1).h : h \in h_1 \,\widetilde{||}_h\, (s_2,t_2).h_2\} \cap Shhist \\ &\quad \cup \{(s_2,t_2).h : h \in (s_1,t_1).h_1 \,\widetilde{||}_h\, h_2\} \cap Shhist \\ (s_1,t_1).h_1 \,\widetilde{||}_h\, (s_2,\delta_2) &= (s_2,\delta_2) \,\widetilde{||}_h\, (s_1,t_1).h_1 \\ &= \{(s_1,t_1).h : h \in h_1 \,\widetilde{||}_h\, (s_2,\delta_2)\} \cap Shhist \end{aligned}$$

$$(s_1,\delta_1) \,\widetilde{||}_h\, (s_2,\delta_2) = \begin{cases} \{(s_1,\delta^+)\}, & si\ s_1 = s_2 \text{ et } \delta_1 = \delta_2 = \delta^+ \\ \{(s_1,\delta^-)\}, & si\ \ 1)\ s_1 = s_2 \text{ et} \\ & \qquad 2)\ \delta_1 = \delta^- \text{ ou } \delta_2 = \delta^- \\ \emptyset, & si\ s_1 \neq s_2 \end{cases}$$

   La composition parallèle $\widetilde{||}$ de deux ensembles d'histoires est obtenue par extension de la fonction $\widetilde{||}_h$. Toutefois, la composition parallèle d'ensembles d'histoires ayant des variables communes a pour sémantique l'ensemble vide.

   Définition 3.12 *La fonction* $\widetilde{||} : \mathcal{P}(Shhist) \times \mathcal{P}(Shhist) \to \mathcal{P}(Shhist)$ *est définie de la façon suivante. Pour tous sous-ensembles* S_1, S_2 *de* $Shhist$,

$$S_1 \,\widetilde{||}\, S_2 = \begin{cases} \bigcup \{h = h_1 \,\widetilde{||}_h\, h_2 : h_1 \in S_1, h_2 \in S_2 \\ \qquad et\ Ext(h) \cap (Var(S_1) \cup Var(S_2)) = \emptyset\} & si\ Var(S_1) \cap Var(S_2) = \emptyset \\ \emptyset & si\ Var(S_1) \cap Var(S_2) \neq \emptyset \end{cases}$$

3. **Choix.** Un agent formé du choix de deux agents se comporte comme l'un de ses composants. Notons que l'agent composé échoue si ses deux composants échouent mais qu'il réussit si l'un au moins de ses composants réussit.

Définition 3.13 *La fonction* $\widetilde{+} : \mathcal{P}(Shhist) \times \mathcal{P}(Shhist) \rightarrow \mathcal{P}(Shhist)$ *est définie de la façon suivante. Pour tous sous-ensembles* S_1, S_2 *de* $Shhist$,

$$S_1 \widetilde{+} S_2 = \{h \in S_1^a \cup S_2^a : Ext(h) \cap (Var(S_1) \cup Var(S_2)) = \emptyset\} \cup S_1^+ \cup S_2^+ \cup (S_1^- \cap S_2^-)$$

Etant donnés les opérateurs $\widetilde{;}$, $\widetilde{\|}$, et $\widetilde{+}$, définir la sémantique dénotationnelle revient à spécifier la sémantique des agents élémentaires *tell*, *ask*, *nask*, *get* et *instr*.

Définition 3.14 *La sémantique dénotationnelle est définie comme la fonction suivante* $\mathcal{D}_h : Sagent \rightarrow \mathcal{P}(Shhist)$. *Pour tout* ψ*-terme* ψ, *et tous agents* A_1, A_2,

$$\begin{aligned}
\mathcal{D}_h(tell(\psi)) &= \{((\theta,\sigma),(\theta,\sigma \cup \{\psi\})).((\kappa,\tau),\delta^+) : \sigma,\tau \in Setat, \theta,\kappa \in Saffect, \theta \prec \kappa\} \\
\mathcal{D}_h(ask(\psi)) &= \{((\theta,\sigma),(\theta\mu,\sigma)).((\kappa,\tau),\delta^+) : \sigma,\tau \in Setat, \psi_c \in \sigma, \psi \lhd \psi_c = \mu, \\
&\quad \theta,\kappa \in Saffect, \theta\mu \prec \kappa, \kappa\mu = \kappa\} \\
&\quad \cup \{((\theta,\sigma),\delta^-) : \sigma \in Setat, \theta \in Saffect, \nexists \psi_c \in \sigma, \psi \text{ correspond à } \psi_c\} \\
\mathcal{D}_h(nask(\psi)) &= \{((\theta,\sigma),(\theta,\sigma)).((\kappa,\tau),\delta^+) : \sigma,\tau \in Setat, \nexists \psi_c \in \sigma, \psi \text{ correspond à } \psi_c, \\
&\quad \theta,\kappa \in Saffect, \theta \prec \kappa\} \\
&\quad \cup \{((\theta,\sigma),\delta^-) : \sigma \in Setat, \theta \in Saffect, \exists \psi_c \in \sigma, \psi \text{ correspond à } \psi_c\} \\
\mathcal{D}_h(get(\psi)) &= \{((\theta,\sigma),(\theta\mu,\sigma \setminus \{\psi_c\})).((\kappa,\tau),\delta^+) : \sigma,\tau \in Setat, \psi_c \in \sigma, \psi \lhd \psi_c = \mu \\
&\quad \theta,\kappa \in Saffect, \theta\mu \prec \kappa, \kappa\mu = \kappa\} \\
&\quad \cup \{((\theta,\sigma),\delta^-) : \sigma \in Setat, \theta \in Saffect, \nexists \psi_c \in \sigma, \psi \text{ correspond à } \psi_c)\} \\
\mathcal{D}_h(instr) &= \{((\theta,\sigma),(\theta,\sigma)).((\kappa,\tau),\delta^+) : \sigma,\tau \in Setat, \theta,\kappa \in Saffect, \\
&\quad executable(instr,\theta), \theta \prec \kappa\} \\
&\quad \cup \{((\theta,\sigma),\delta^-) : \sigma \in Setat, \neg executable(instr,\theta)\} \\
\mathcal{D}_h(A_1 \ ; \ A_2) &= \mathcal{D}_h(A_1) \widetilde{;} \mathcal{D}_h(A_2) \\
\mathcal{D}_h(A_1 \ || \ A_2) &= \mathcal{D}_h(A_1) \widetilde{\|} \mathcal{D}_h(A_2) \\
\mathcal{D}_h(A_1 \ + \ A_2) &= \mathcal{D}_h(A_1) \widetilde{+} \mathcal{D}_h(A_2)
\end{aligned}$$

où les unions et différences sur les états s'entendent au sens multi-ensembliste.

Soulignons, dans la sémantique des primitives *ask* et *get*, la condition $\kappa\mu = \kappa$ qui impose que les variables modifiées par la primitive ne le soient pas par le contexte.

Pour être complet, la sémantique dénotationnelle de l'agent vide E est définie comme

$$\mathcal{D}_h(E) = \{((\theta,\sigma),\delta^+) : \sigma \in Setat, \theta \in Saffect\}$$

Il est intéressant d'observer la propriété suivante.

Proposition 3.1 *Soit* A *un agent et* h *une histoire de* $\mathcal{D}_h(A)$, *on a*

$$Ext(h) \cap Var(h) = \emptyset$$

$$Ext(\mathcal{D}_h(A)) \cap Var(\mathcal{D}_h(A)) = \emptyset$$

$$Var(\mathcal{D}_h(A)) = Var_a(A)$$

Preuve. Les deux premières égalités s'établissent par un raisonnement sur la structure à partir des définitions 3.10 à 3.14. La dernière inclusion s'obtient en mettant en parallèle les définitions 3.14 et 1.7

Avant d'examiner le lien entre sémantique dénotationnelle et sémantique opérationnelle, intéressons nous aux relations entre les transitions et la sémantique dénotationnelle.

Proposition 3.2 *Soient des affectations θ et κ, des états σ et τ et un agent A.*

1. *l'histoire $h = ((\sigma, \theta), \delta^+)$ est dans $\mathcal{D}_h(A)$ ssi $A = E$*
2. *l'histoire $h = ((\sigma, \theta), \delta^-)$ est dans $\mathcal{D}_h(A)$ ssi A n'est pas l'agent vide et $\langle A \,\&\, \theta \mid \sigma\rangle \not\rightarrow$*
3. *Si il existe une histoire h' telle que $h = ((\theta, \sigma), (\kappa, \tau)).h'$ soit dans $\mathcal{D}_h(A)$ alors il existe un agent B tel que $\langle A \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B \,\&\, \kappa \mid \tau\rangle$.*
   En outre, il y a un tel B pour lequel $h' \in \mathcal{D}_h(B)$
4. *Si il existe un agent B tel que $\langle A \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B \,\&\, \kappa \mid \tau\rangle$, alors quelle que soit l'histoire h' de $\mathcal{D}_h(B)$ si l'histoire $h = ((\theta, \sigma).(\kappa, \tau)).h'$ est un élément de $Shhist$ et vérifie $Ext(h) \cap Var\mathcal{D}_h(A) = \emptyset$, cette histoire h est dans $\mathcal{D}_h(A)$.*
   En outre, il existe une telle histoire h continue.

Preuve 1. La condition nécessaire découle directement de la définition de $\mathcal{D}_h(E)$. Démontrons la conditions suffisante par induction sur le nombre d'opérateurs de compositions intervenant dans l'agent A.

La définition de $\mathcal{D}_h$ n'introduit aucune histoire de ce type dans la sémantique des agents atomiques. Examinons les agents composés.

Cas 1. L'agent A est la composition séquentielle des agents A_1 et A_2. Pour que l'histoire $h = ((\sigma, \theta), \delta^+)$ appartienne à $\mathcal{D}_h(A_1 \ ; \ A_2)$, il doit y avoir une situation s pour laquelle $(s, \delta^+) \in \mathcal{D}_h(A_1)$ et h doit être dans $\mathcal{D}_h(A_2)$. Par hypothèse d'induction ceci impose que $A_1 = A_2 = E$ et donc $A = E$.

Cas 2. L'agent A est la composition parallèle des agents A_1 et A_2. Pour que l'histoire $h = ((\sigma, \theta), \delta^+)$ appartienne à $\mathcal{D}_h(A_1 \mid\mid A_2)$, il faut que h appartienne à la fois à $\mathcal{D}_h(A_1)$ et à $\mathcal{D}_h(A_2)$, ce qui impose que les agents A_1 et A_2 soient tous deux l'agent vide et donc $A = E$

Cas 3. L'agent A consiste en l'alternative entre les agents A_1 et A_2. Pour que l'histoire $h = ((\sigma, \theta), \delta^+)$ appartienne à $\mathcal{D}_h(A_1 \ + \ A_2)$, il faut que h appartienne soit à $\mathcal{D}_h(A_1)$ soit à $\mathcal{D}_h(A_2)$, ce qui impose que l'un des agents A_1 et A_2 soit l'agent vide. Ceci est impossible car la structure des agents ne permet pas à l'agent vide d'entrer dans une composition avec l'opérateur de choix.

Preuve 2. Pour un agent atomique, la propriété découle directement des définitions 3.14 et des règles de transition de la table 1.3. Examinons la condition suffisante le cas des agents composés par induction sur le nombre d'opérateurs qu'ils contiennent.

Cas 1. L'agent A est la composition séquentielle des agents A_1 et A_2. Puisque $E \ ; \ A$ est remplacé par A, on peut supposer que A_1 n'est pas l'agent vide. Par la propriété précédente, $\mathcal{D}_h(A_1)$ ne contient aucune histoire de type (s, δ^+). Pour que l'histoire $h = ((\sigma, \theta), \delta^-)$ appartienne à $\mathcal{D}_h(A_1 \ ; \ A_2)$, il faut donc que h soit dans $\mathcal{D}_h(A_1)$. L'induction nous permet d'affirmer que, dans ce cas, $\langle A_1 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$. Et dès lors, $\langle A_1 \ ; \ A_2 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$.

Cas 2. L'agent A est la composition parallèle des agents A_1 et A_2. Ici aussi nous pouvons supposer A_1 et A_2 distincts de l'agent vide. Pour que l'histoire $h = ((\sigma, \theta), \delta^-)$ appartienne à $\mathcal{D}_h(A_1 \parallel A_2)$, il faut donc que h soit dans $\mathcal{D}_h(A_1)$ et dans $\mathcal{D}_h(A_2)$. Ce qui nous permet d'affirmer que $\langle A_1 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$ et $\langle A_2 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$. Et dès lors, $\langle A_1 \parallel A_2 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$.

Cas 3. L'agent A consiste en l'alternative entre les agents A_1 et A_2. La structure de l'agent nous garantit que A_1 et A_2 sont distincts de l'agent vide. Pour que l'histoire $h = ((\sigma, \theta), \delta^-)$ appartienne à $\mathcal{D}_h(A_1 + A_2)$, il faut donc que h soit dans $\mathcal{D}_h(A_1)$ et dans $\mathcal{D}_h(A_2)$. Ce qui nous permet d'affirmer que $\langle A_1 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$ et $\langle A_2 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$. Et dès lors, $\langle A_1 + A_2 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$.

La preuve de la condition nécessaire s'obtient elle aussi par induction sur la composition de l'agent.

Cas 1. L'agent A est la composition séquentielle des agents A_1 et A_2 où A_1 n'est pas l'agent vide. Puisque $\langle A_1 \,;\, A_2 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$, les règles de transition nous permettent de déduire que $\langle A_1 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$. Par induction, ceci nous garantit que l'histoire $h = ((\sigma, \theta), \delta^-)$ appartient à $\mathcal{D}_h(A_1)$ et donc $\mathcal{D}_h(A_1 \,;\, A_2)$.

Cas 2. L'agent A est la composition parallèle des agents A_1 et A_2 où ni A_1 ni A_2 n'est l'agent vide. Puisque $\langle A_1 \parallel A_2 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$ les règles de transition nous permettent de déduire que $\langle A_1 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$ et $\langle A_2 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$. Par induction, ceci nous garantit que l'histoire $h = ((\sigma, \theta), \delta^-)$ appartient à la fois à $\mathcal{D}_h(A_1)$ et à $\mathcal{D}_h(A_1)$. Dès lors h est dans $\mathcal{D}_h(A_1 \parallel A_2)$.

Cas 3. L'agent A est l'alternative entre les agents A_1 et A_2 où ni A_1 ni A_2 n'est l'agent vide. Les règles de transitions nous permettent de déduire que si $\langle A_1 + A_2 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$ alors $\langle A_1 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$ et $\langle A_2 \,\&\, \theta \mid \sigma\rangle \not\rightarrow$. Par induction, ceci nous garantit que l'histoire $h = ((\sigma, \theta), \delta^-)$ appartient à la fois à $\mathcal{D}_h(A_1)$ et à $\mathcal{D}_h(A_1)$. Dès lors h est dans $\mathcal{D}_h(A_1 + A_2)$.

Preuve 3. Remarquons tout d'abord que si A est l'agent vide, $\mathcal{D}_h(A)$, ne peut contenir une telle histoire. Pour les autre cas, nous procédons à nouveau par induction sur la structure de l'agent A.

Cas 1. L'agent A est un agent atomique. La définition de $\mathcal{D}_h$ assure alors que h' est de la forme (s, δ^+). L'agent vide convient alors comme agent B et de fait $h' \in \mathcal{D}_h(E)$.

Cas 2. L'agent A est la composition séquentielle des agents A_1 et A_2 où A_1 n'est pas l'agent vide. L'histoire $h = ((\theta, \sigma), (\kappa, \tau)).h'$ peut être dans $\mathcal{D}_h A_1 \,;\, A_2)$ dans deux cas.

Sous-cas i. L'histoire $h \in \mathcal{D}_h(A_1)$ se termine par δ^-. L'induction nous fournit alors un agent B_1 tel que $\langle A_1 \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B_1 \,\&\, \kappa \mid \tau\rangle$ et $h' \in \mathcal{D}_h(B_1)$. L'agent $B_1 \,;\, A2$ est un agent B acceptable. En effet puisque l'histoire h' se termine par δ^- et est dans $\mathcal{D}_h(B_1)$, elle est aussi dans $\mathcal{D}_h(B_1 \,;\, A_2)$.

Sous-cas ii. L'histoire h peut s'écrire $((\theta, \sigma), (\kappa, \tau)).k.l$ avec $((\theta, \sigma), (\kappa, \tau)).k.(s, \delta^+) \in \mathcal{D}_h(A_1)$ et $l \in \mathcal{D}_h(A_2)$. L'induction nous fournit alors un agent B_1 tel que $\langle A_1 \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B_1 \,\&\, \kappa \mid \tau\rangle$ et $k.(s, \delta^+) \in \mathcal{D}_h(B_1)$. L'agent $(B_1 \,;\, A2)$ est un agent B acceptable. En effet puisque l'histoire $k.(s, \delta^+)$ est dans $\mathcal{D}_h(B_1)$ et l est dans $\mathcal{D}_h(A_2)$, l'histoire $h' = k.l$ est dans $\mathcal{D}_h(B_1 \,;\, A_2)$.

Cas 3. L'agent A est la composition parallèle des agents A_1 et A_2 où ni A_1 ni A_2 n'est l'agent vide. Pour que l'histoire $h = ((\theta, \sigma), (\kappa, \tau)).h'$ soit dans $\mathcal{D}_h A_1 \,;\, A_2)$ il faut que $h' \in h_1 \,\widetilde{\|}_h\, h_2$ où h_1 et h_2 vérifient l'une des conditions suivantes.

Sous-cas i. $((\theta,\sigma),(\kappa,\tau)).h_1 \in \mathcal{D}_h(A_1)$ et $h_2 \in \mathcal{D}_h(A_2)$. L'induction nous fournit alors un agent B_1 tel que $\langle A_1 \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B_1 \,\&\, \kappa \mid \tau\rangle$ et $h_1 \in \mathcal{D}_h(B_1)$. L'agent $B_1 \parallel A2$ est un agent B acceptable. En effet puisque l'histoire h_1 est dans $\mathcal{D}_h(B_1)$ et h_2 est dans $\mathcal{D}_h(A_2)$, l'histoire $h' \in h_1 \,\widetilde{\|}_h\, h_2$ est dans $\mathcal{D}_h(B_1 \parallel A_2)$.

Sous-cas ii. $h_1 \in \mathcal{D}_h(A_1)$ et $((\theta,\sigma),(\kappa,\tau)).h_2 \in \mathcal{D}_h(A_2)$. Se traite de la même façon.

Cas 4. L'agent A est l'alternative entre les agents A_1 et A_2 où ni A_1 ni A_2 n'est l'agent vide. Pour que l'histoire $h = ((\theta,\sigma),(\kappa,\tau)).h'$ soit dans $\mathcal{D}_h A_1 \;;\; A_2)$ il faut qu'elle appartienne soit à $\mathcal{D}_h(A_1)$ soit à $\mathcal{D}_h(A_2)$. Si $h \in \mathcal{D}_h(A_1)$, l'induction fournit un agent B_1 tel que $\langle A_1 \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B_1 \,\&\, \kappa \mid \tau\rangle$ et $h' \in \mathcal{D}_h(B_1)$. Cet agent B_1 convient comme agent B. L'alternative se traite de la même façon.

Preuve 4. L'agent vide ne peut vérifier les hypothèses de cette propriété. Pour les autres cas, nous procédons par induction sur le nombre d'opérateurs de l'agent A.

Cas 1. L'agent A est atomique. L'agent B est alors l'agent vide et $\langle A \,\&\, \theta \mid \sigma\rangle \rightarrow \langle E \,\&\, \kappa \mid \tau\rangle$. Quelle que soit l'histoire $((\theta',\sigma'),\delta^+)$ de $\mathcal{D}_h(E)$ vérifiant $\kappa \prec \theta'$, l'histoire $h = ((\theta,\sigma),(\kappa,\tau)).((\theta',\sigma'),\delta^+)$ est dans $\mathcal{D}_h(A)$. Une histoire continue est fournie par $h = ((\theta,\sigma),(\kappa,\tau)).((\kappa,\tau),\delta^+)$.

Cas 2. L'agent A est la composition séquentielle des agents A_1 et A_2 où A_1 n'est pas l'agent vide. Puisque $\langle A_1 \;;\; A_2 \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B \,\&\, \kappa \mid \tau\rangle$ l'examen des règles de transition, nous permet de conclure à l'existence d'un agent B_1 tel que $\langle A_1 \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B_1 \,\&\, \kappa \mid \tau\rangle$ et $B = B_1; A_2$. Toute histoire h' de $\mathcal{D}_h(B)$ est donc soit une histoire de $\mathcal{D}_h(B_1)$ se terminant par δ^-, soit une histoire de la forme $h' = h_1.h_2$ où $h_2 \in \mathcal{D}_h(A_2)$ et $h_1.(s,\delta^+) \in \mathcal{D}_h(A_1)$ pour une certaine situation s. Considérons une telle histoire vérifiant de plus $h = ((\theta,\sigma),(\kappa,\tau)).h' \in Shhist$ et $Ext(h) \cap Var(\mathcal{D}_h(A)) = \emptyset$. Dans le premier cas, l'induction nous garantit que h est dans $\mathcal{D}_h(A_1)$. Puisqu'elle se termine par δ^-, elle est aussi dans $\mathcal{D}_h(A_1 \;;\; A_2)$. Dans le second cas, l'induction nous garantit que $h^* = ((\theta,\sigma),(\kappa,\tau)).h_1.(s,\delta^+)$ est dans $\mathcal{D}_h(A_1)$. Ce qui nous permet ici aussi de conclure que h est dans $\mathcal{D}_h(A_1 \;;\; A_2)$.

Quant à l'existence d'une histoire continue l'induction nous fournit une histoire continue $h_1 = ((\theta,\sigma),(\kappa,\tau)).h_1'.((\theta',\sigma'),\delta)$ de $\mathcal{D}_h(A_1)$ telle que $h_1'.((\theta',\sigma'),\delta) \in \mathcal{D}_h(B_1)$. Si cette histoire ce termine par δ^-, elle est alors aussi dans $\mathcal{D}_h(A_1 \;;\; A_2)$. Si elle se termine par $((\theta',\sigma'),\delta^+)$, selon que $\langle A_2 \,\&\, \theta' \mid \sigma'\rangle \rightarrow$ ou $\langle A_2 \,\&\, \theta' \mid \sigma'\rangle \not\rightarrow$ l'induction ou la propriété 2, nous fournissent une histoire continue h_2 de $\mathcal{D}_h(A_)$ commençant en (θ',σ'). L'histoire $((\theta,\sigma),(\kappa,\tau)).h_1'.h_2$ est alors une histoire continue de $\mathcal{D}_h(A_1 \;;\; A_2)$.

Cas 3. L'agent A est la composition parallèle des agents A_1 et A_2 où ni A_1 ni A_2 n'est l'agent vide et $Var(\mathcal{D}_h(A_1)) \cap Var(\mathcal{D}_h(A_2)) = \emptyset$. Puisque $\langle A_1 \parallel A_2 \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B \,\&\, \kappa \mid \tau\rangle$ l'examen des règles de transition, nous permet de conclure soit à l'existence de B_1 tel que $\langle A_1 \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B_1 \,\&\, \kappa \mid \tau\rangle$ et $B = B_1 \parallel A_2$. Soit à celle B_2 tel que $\langle A_2 \,\&\, \theta \mid \sigma\rangle \rightarrow \langle B_2 \,\&\, \kappa \mid \tau\rangle$ et $B = A_1 \parallel B_2$. Envisageons la première situation, la seconde se traitant de façon tout à fait analogue. Pour toute histoire h' de $\mathcal{D}_h(B)$ il existe donc des histoires h_1 de $\mathcal{D}_h(B_1)$ et h_2 de $\mathcal{D}_h(A_2)$ telles que $h' \in h_1 \,\widetilde{\|}_h\, h_2$. Considérons une telle histoire vérifiant de plus $h = ((\theta,\sigma),(\kappa,\tau)).h' \in Shhist$ et $Ext(h) \cap Var(\mathcal{D}_h(A)) = \emptyset$. L'induction nous garantit que $((\theta,\sigma),(\kappa,\tau)).h_1$ est dans $\mathcal{D}_h(A_1)$. Dès lors l'histoire $h = ((\theta,\sigma),(\kappa,\tau)).h'$ qui est dans $((\theta,\sigma),(\kappa,\tau)).h_1 \,\widetilde{\|}_h\, h_2$, dans $Shhist$ et vérifie $Ext(h) \cap Var(\mathcal{D}_h(A)) = \emptyset$ est dans $\mathcal{D}_h(A_1 \parallel A_2)$.

Quant à l'existence d'une histoire continue, nous distinguerons deux cas.

Sous-cas i. $\langle B \,\&\, \kappa \mid \tau\rangle \not\rightarrow$. Nous avons démontré que, dans ce cas, l'histoire $((\kappa,\tau),\delta^-)$ est dans $\mathcal{D}_h(B)$. Elle est donc à la fois dans $\mathcal{D}_h(B_1)$ et $\mathcal{D}_h(A_2)$. Dès lors, l'histoire continue $((\theta,\sigma),(\kappa,\tau)).((\kappa,\tau),\delta^-)$ est une histoire de $\mathcal{D}_h(A_1)$ qui est aussi dans $\mathcal{D}_h(A_1 \mid\mid A_2)$.

Sous-cas ii. $\langle B \,\&\, \kappa \mid \tau\rangle \rightarrow \langle C \,\&\, \theta' \mid \sigma'\rangle$. L'induction nous fournit alors une histoire $h' = ((\kappa,\tau).(\theta',\sigma')).k$, histoire continue de $\mathcal{D}_h(B_1 \underset{\sim}{\mid\mid} A_2)$. Il existe donc des histoires h_1 de $\mathcal{D}_h(B_1)$ et h_2 de $\mathcal{D}_h(A_2)$ telles que $h' \in h_1 \underset{\sim}{\|}_h h_2$. Considérons l'histoire $k_1 = ((\theta,\sigma),(\kappa,\tau)).h_1$. Il s'agit bien d'une histoire de Shhist. De plus $Ext(k_1) \in Var(\mathcal{D}_h(A_2))$, en effet h' étant continue, les trous de l'histoire h_1 correspondent à des pas effectués par h_2. Dès lors, $Ext(k_1) \cap Var(\mathcal{D}_h(A_1)) = \emptyset$, et l'induction nous permet de conclure que k_1 est dans $\mathcal{D}_h(A_1)$. Nous pouvons maintenant construire l'histoire $h = ((\theta,\sigma),(\kappa,\tau)).h'$ qui est une histoire continue de $k_1 \underset{\sim}{\|}_h h_2$ et donc de $\mathcal{D}_h(A_1 \mid\mid A_2)$.

Il découle de ces propriétés quelques corollaires intéressants. Observons tout d'abord le résultat suivant.

Proposition 3.3 *Soit A un agent et (θ,σ) et (κ,τ) des situations telles que*

$$\mathcal{D}_h(A)[((\theta,\sigma),(\kappa,\tau))] \neq \emptyset.$$

Soient $B_1,\ldots,B_m$ tous les agents B tels que

$$\langle A\&\theta \mid \sigma\rangle \rightarrow \langle B\&\kappa \mid \tau\rangle,$$

alors

$$\mathcal{D}_h(A)[((\theta,\sigma),(\kappa,\tau))] \subseteq \mathcal{D}_h(B_1) \cup \cdots \cup \mathcal{D}_h(B_m).$$

Preuve. Par application directe de la troisième des propriétés de la proposition 3.2.

Notons que $\mathcal{D}_h(B_1) \cup \cdots \cup \mathcal{D}_h(B_m)$ est presque $\mathcal{D}_h(B_1 + \cdots + B_m)$. Ces deux ensembles diffèrent par le traitement des exécutions échouant immédiatement. Elles sont toutes présentes dans $\mathcal{D}_h(B_1) \cup \cdots \cup \mathcal{D}_h(B_m)$ alors que seules celles qui sont communes à $B_1,\ldots,B_m$ apparaissent dans $\mathcal{D}_h(B_1 + \cdots + B_m)$.

Une autre conséquence intéressante des résultats 3.2 est que pour tout agent A, la sémantique dénotationnelle $\mathcal{D}_h(A)$ est *extensible* au sens suivant.

Proposition 3.4 *Quels que soient l'agent A et les situations $s, s_1,\ldots,s_n$,*

1. *si $\mathcal{D}_h(A) \neq \emptyset$ alors il y a une histoire continue de $\mathcal{D}_h(A)$ qui commence en s*
2. *si $\mathcal{D}_h(A)[(s_1,s_2).\ldots.(s_{n-1},s_n)] \neq \emptyset$ alors il y a une histoire continue de $\mathcal{D}_h(A)$ de la forme $(s_1,s_2).\ldots.(s_{n-1},s_n).h'$*

Preuve. La preuve du premier point s'obtient par une application d'un des points 1, 2 ou 4 de la proposition 3.2 selon le cas. Examinons le second point. Soit h une histoire de $\mathcal{D}_h(A)[(s_1,s_2).\ldots.(s_{n-1},s_n)]$. Une application récursive du point 3 de la propriété 3.2 à l'histoire $(s_1,s_2).\ldots.(s_{n-1},s_n).h$, fournit les agents $A_1,\ldots,A_n$ $(A_1 = A)$ tels que

$$\langle A_i\&\theta_i \mid \sigma_i\rangle \rightarrow \langle A_{i+1}\&\theta_{i+1} \mid \sigma_{i+1}\rangle$$

où $(\theta_i, \sigma_i) = s_i$. L'application du point 4 de la propriété 3.2 à $\langle A_{n-1}\&\theta_{n-1} \mid \sigma_{n-1}\rangle \rightarrow \langle A_n\&\theta_n \mid \sigma_n\rangle$, fournit une histoire $(s_{n-1}, s_n).h'$ continue dans $\mathcal{D}_h(A_{n-1})$ qui permet de vérifier que $(s_1, s_2).....(s_{n-1}, s_n).h'$ est dans $\mathcal{D}_h(A)$.

3.5 Sémantiques complètement abstraites

Etant donné une relation d'équivalence $\equiv$, l'abstraction complète consiste essentiellement à trouver un modèle sémantique M qui vérifie les propriétés suivantes

- il est complet au sens ou $\equiv$ est contenu dans $\equiv_M$;
- il fait suffisamment de distinctions de façon à fournir une relation de congruence $\equiv_m$ contenue dans $\equiv$. En d'autres mots, $\equiv_M$ doit être la relation de congruence maximale contenue dans $\equiv$.

Une caractérisation plus précise peut être obtenue grâce à la notion de contexte.

Définition 3.15 *L'ensemble des contextes est défini inductivement de la façon suivante :*

- $\Box$ *est un contexte,*
- *toute expression de base est un contexte,*
- *si c et d sont des contextes alors, pour tout i, $c \diamond_i d$ est un contexte.*

Le symbole $\Box$ est appelé un trou. Des contextes contiendront en général plusieurs occurrences de ce trou. Ils peuvent être vus comme des fonctions de L dans L. Dans cette perspective, nous désignerons également un contexte c par c(.). Etant donné une expression s, c(s) désignera l'expression obtenue en substituant s à toutes les occurrences de $\Box$ dans c(.).

Définition 3.16 *Quelle que soit la relation d'équivalence $\equiv$ définie sur L, nous lui associons la relation $\equiv_c$ définie sur L par*

$$s \equiv_c t \text{ ssi pour tout contexte } c(.), c(s) \equiv c(t).$$

On a le résultat suivant.

Théorème 3.2 *Toute relation d'équivalence $\equiv$ vérifie les propriétés suivantes :*

1. *$\equiv_c$ est une relation de congruence.*
2. *$\equiv_c$ est incluse dans $\equiv$*
3. *toute relation de congruence incluse dans $\equiv$ est aussi incluse dans $\equiv_c$.*

Preuve. Pour démontrer que $\equiv_c$ est une relation de congruence, considérons des expressions s_1, s_2, t_1 et t_2 vérifiant $s_1 \equiv_c s_2$ et $t_1 \equiv_c t_2$. Quels que soient le contexte C, et l'opérateur $\diamond_i$, nous pouvons définir les contextes $C' = C(s_1 \diamond_i \Box)$ et $C" = C(\Box \diamond_i t_2)$. On a alors

$$C(s_1 \diamond_i t_1) = C'(t_1) = C'(t_2) = C"(s_1) = C"(s_2) = C(s_2 \diamond_i t_2)$$

L'inclusion de $\equiv_c$ dans $\equiv$ découle directement de l'existence du contexte $C = \Box$. En effet si deux expressions s_1 et s_2 vérifient $s_1 \equiv_c s_2$, elles vérifient alors $s_1 = C(s_1) = C(s_2) = s_2$

Soit maintenant une relation de congruence $\equiv_e$ incluse dans $\equiv$. Pour vérifier son inclusion dans $\equiv_c$, considérons deux expressions s_1 et s_2 qui vérifient $s_1 \equiv_e s_2$. Puisque $\equiv_e$ est une relation de congruence, elle respecte tous les opérateurs. Quel que soit le contexte C,

on a donc $C(s_1) \equiv_e C(s_2)$. La relation $\equiv_e$ étant incluse dans $\equiv$ ceci donne $C(s_1) \equiv C(s_2)$. On a donc aussi $s_1 \equiv_c s_2$.

Avec ces éléments, nous pouvons définir la notion d'abstraction complète.

Définition 3.17 *Soit $L \to S$ un modèle sémantique.*

1. *M est correct par rapport à $\equiv$ ssi $\equiv_M$ est inclus dans $\equiv_c$,*
2. *M est complet par rapport à $\equiv$ ssi $\equiv_c$ est inclus dans $\equiv_M$,*
3. *M est complètement abstrait par rapport à $\equiv$ ssi il est à la fois correct et complet par rapport à $\equiv$ c'est-à-dire si $\equiv_M$ est $\equiv_c$.*

Généralement, on définit d'abord une sémantique opérationnelle et on prend comme relation d'équivalence $\equiv$ la relation induite par la sémantique opérationnelle, c'est-à-dire l'équivalence $\equiv_O$ définie par

$$s \equiv_O t \text{ ssi } O(s) = O(t)$$

pour toutes expressions s et t. Alors, le problème est de construire une sémantique dénotationnelle compositionnelle M complètement abstraite par rapport à $\equiv_O$. Dans cette situation, une définition alternative de la propriété d'abstraction complète est la suivante.

Définition 3.18 *La sémantique M est complètement abstraite par rapport à O ssi pour toutes expressions s et t, on a*

$$M(s) = M(t) \text{ ssi, pour tout contexte } c,\ O(c(s)) = O(c(t))$$

C'est cette démarche que nous adoptons pour les sémantiques $\mathcal{D}_h$ et $\mathcal{O}_h$ dans la section suivante.

3.6 Abstraction complète

Avant d'aborder la démonstration de l'abstraction complète de la sémantique dénotationnelle $\mathcal{D}_h$ par rapport à la sémantique opérationnelle $\mathcal{O}_h$, réexprimons les définitions et propriétés à vérifier pour le langage $\mathcal{L}_\Psi$.

3.6.1 Définitions

Définition 3.19 *Soit $\Box$ un nouveau symbole. Nous définissons l'ensemble des contextes Scontext comme l'ensemble des contextes C définis par les règles suivantes où A désigne un agent*

$$C \quad ::= \quad \Box \mid A \mid C\,;\,A \mid A\,;\,C \mid C \,||\, A \mid A \,||\, C \mid C + A \mid A + C$$

Définition 3.20 *L'application d'un contexte C à un agent A est définie comme le nouvel agent obtenu en remplaçant les éventuelles occurrences du symbole $\Box$ dans C par A. On note $C[A]$ l'application du contexte C à l'agent A.*

Définition 3.21 *La sémantique $\mathcal{D}_h$ est complètement abstraite par rapport à la sémantique historique $\mathcal{O}_h$ ssi elle vérifie la propriété suivante. Quels que soient les agents A_1, A_2, les deux affirmations suivantes sont équivalentes :*

i) pour tout contexte C, $\mathcal{O}_h(C[A_1]) = \mathcal{O}_h(C[A_2])$;

ii) $\mathcal{D}_h(A_1) = \mathcal{D}_h(A_2)$.

Il est aisé d'observer que la sémantique $\mathcal{D}_h$ est compositionnelle par construction. Elle est également correcte par rapport à la sémantique $\mathcal{O}_h$, en effet, celle-ci peut être obtenue à partir de $\mathcal{D}_h$. Pour cela, il suffit de prendre parmi les histoires de $\mathcal{D}_h$, celles qui sont continues et commencent avec l'état et l'affectation vide pour obtenir celles produites par $\mathcal{O}_h$.

Proposition 3.5 *Soit la fonction* $\beta : \mathcal{P}(Shhist) \to \mathcal{P}(Shist)$ *définie de la façon suivante. Pour tout sous-ensemble* $S \subseteq Shhist$,

$$\beta(S) = \{\bar{h} : h \in S, h \text{ est continu}, init(h) = (\emptyset, \emptyset)\}.$$

Alors

$$\mathcal{O}_h = \beta \circ \mathcal{D}_h.$$

Preuve. Soit un agent A. Les points 1,2 et 4 de la proposition 3.2 permettent d'observer que si $h_O = s_1 \ldots . s_n . \delta$ est une histoire de $\mathcal{O}_h(A)$, l'histoire $h_D = (s_1, s_2) \ldots . (s_{n-1}, s_n) . (s_n, \delta)$ est une histoire continue de $\mathcal{D}_h(A)$ telle que $\bar{h_D} = h_O$. On a donc $\mathcal{O}_h \subseteq \beta \circ \mathcal{D}_h$.

Réciproquement, si $h_D = (s_1, s_2) \ldots . (s_{n-1}, s_n) . (s_n, \delta)$ est une histoire continue de $\mathcal{D}_h(A)$, le point 3, de la proposition 3.2 garantit que $\bar{h_D} \in \mathcal{O}_h(A)$.

3.6.2 Intuition

Donnons ici une intuition de la démarche qui permet de démontrer que les propriétés demandées par la définition 3.21 sont bien équivalentes. La compositionnalité de $\mathcal{D}_h$ et la fonction β introduite plus haut permettent d'établir que ii)⇒i). Pour démontrer la réciproque, on procède par contraposition. Supposons donc donnés deux agent A_1 et A_2 tels que

$$\mathcal{D}_h(A_1) \neq \mathcal{D}_h(A_2).$$

Nous allons construire un contexte C tel que

$$\mathcal{O}_h(C[A_1]) \neq \mathcal{O}_h(C[A_2]).$$

Il s'agit ainsi de construire à partir d'une histoire dénotationnelle h commençant en (θ, σ) d'un agent, disons A_1, qui n'est pas une histoire dénotationnelle de l'autre, A_2, un contexte C et une histoire opérationnelle de $C[A_1]$ qui ne soit pas dans $C[A_2]$. Etant donné la relation entre $\mathcal{O}_h$ et $\mathcal{D}_h$ exprimée plus haut par la fonction β, ceci revient à établir l'existence d'une histoire dénotationnelle continue commençant en $(\emptyset, \emptyset)$ qui soit dans $\mathcal{D}_h(C[A_1])$ mais pas dans $\mathcal{D}_h(C[A_2])$. Pour faire cela, l'idée est de construire à partir de h des agents S et T et des nouvelles histoires $h_s \in \mathcal{D}_h(S)$ et $h_t \in \mathcal{D}_h(A_1 \parallel T)$ qui soient tels que l'histoire $h_s.h_t$ soit une histoire continue commençant en $(\emptyset, \emptyset)$ contenue dans $\mathcal{D}_h(S; (A_1 \parallel T))$ mais pas dans $\mathcal{D}_h(S; (A_2 \parallel T))$.

La construction de S et de l'histoire h_s, histoire continue commençant en $(\emptyset, \emptyset)$ et se terminant en $(\theta, \emptyset)$, est prise en charge par un agent B. La construction de T et de l'histoire h_t, histoire continue commençant en $(\theta, \emptyset)$, repose sur une induction sur la longueur de h. Dans le cas de base, h est de la forme $((\theta, \sigma), \delta)$ où δ est soit δ^+, soit δ^-. Les éléments T du contexte test ont alors pour objectif de construire une suite continue amenant à l'état (θ, σ) à partir de l'état initial $(\theta, \emptyset)$ d'une façon qui, d'une part, empêche A_1 et A_2 de faire

un pas intermédiaire et, d'autre part, impose à A_1 et A_2 de faire le dernier pas $((\theta,\sigma),\delta)$. Par hypothèse, ceci est possible pour A_1 mais pas pour A_2.

Dans le cas inductif, h prend la forme $((\theta,\sigma),(\kappa,\tau)).h^*$ pour une histoire h^*. Deux cas sont possibles. Soit il n'y a aucune histoire de $\mathcal{D}_h(A_2)$ qui commence par $((\theta,\sigma),(\kappa,\tau))$, soit celles qui le font ne peuvent se terminer par h^*. Dans le premier cas, la preuve se construit comme dans le cas de base. Dans le second cas, la preuve utilise l'induction. Mais l'induction doit être appliquée à h^* de $\mathcal{D}_h(A_1)[((\theta,\sigma),(\kappa,\tau))]$ mais pas de $\mathcal{D}_h(A_2)[((\theta,\sigma),(\kappa,\tau))]$. Comme nous l'avons montré plus haut, ces sémantiques s'avèrent être quasiment mais pas exactement les sémantiques dénotationnelles $\mathcal{D}_h(A_1')$ et $\mathcal{D}_h(A_2')$ pour certains agents A_1' et A_2'. En conséquence, nous allons généraliser un peu l'induction aux ensembles d'histoires dénotationnelles. Par simplicité, contentons-nous dans cet exposé de l'intuition d'appliquer l'induction à h^*, A_1' et A_2'. Ceci nous fournit un agent T' et une histoire h" qui est dans $\mathcal{D}_h(A_1' \parallel T')$ mais pas dans $\mathcal{D}_h(A_2' \parallel T')$. A partir de là, nous pouvons construire l'agent T et une histoire h''' qui soit dans$\mathcal{D}_h(A_1 \parallel T)$ mais pas dans $\mathcal{D}_h(A_2 \parallel T)$. En fait, le pas $((\theta,\sigma),(\kappa,\tau))$ doit être effectué avant h'' et, puisque h''' doit être continu, h'' doit commencer dans n'importe quel état initial. Dès lors, nous devons généraliser le théorème et construire de façon générale à partir de h une histoire h' qui commence dans un état initial quelconque. Etant donné cette généralisation, l'agent T consiste essentiellement à faire tout d'abord les pas nécessaires pour produire (θ,σ) à partir d'un état initial donné, ensuite faire une transition auxiliaire de l'état τ vers un certain état τ' choisi de façon à imposer à A_1 et A_2 de faire le pas $((\theta,\sigma),(\kappa,\tau))$, modifier l'affectation de valeurs de façon à ce qu'elle soit celle de l'état initial de h'' et enfin terminer par T'.

3.6.3 Résultats Préliminaires

L'intuition exposée ci-dessus fait apparaître deux tâches auxiliaires. La première consiste à faire réaliser par un agent auxiliaire les pas nécessaires pour passer d'une affectation initiale de valeurs θ à une affectation de valeurs κ telle que $\theta \prec \kappa$ en gardant un état final égal à l'état initial σ. Nous allons construire un agent $B^W_{(\theta,\sigma)\to(\kappa,\sigma)}$ qui réalisera cette tâche. La seconde tâche a pour objet de faire réaliser par un agent auxiliaire les pas nécessaires pour passer d'un état initial σ à un état donné τ sans modifier l'affectation θ de valeurs aux variables. Ces pas sont réalisés au moyen de l'agent $Ag^V_{\sigma\to\tau}$ que nous définissons plus loin.

Construction. Soient les affectations de valeurs θ et κ telles que $\theta \prec \kappa$, σ un état et W un ensemble de ψ-termes. Construisons l'agent $B^W_{(\theta,\sigma)\to(\kappa,\sigma)}$ qui effectuera le passage de la situation (θ,σ) à la situation (κ,σ). Désignons par μ l'affectation de valeurs qui ne reprend que les variables modifiées entre θ et κ. L'affectation μ est définie par

$$\mu(x) = \begin{cases} \kappa(x) & \text{si } \kappa(x) \neq \theta(x) \\ \perp & \text{si } \kappa(x) = \theta(x). \end{cases}$$

Remarquons que l'on a $\kappa = \theta\mu$. Désignons par $fonct$ un foncteur qui n'apparaît dans aucun des ψ-termes de σ ni de W et par $x_i, i = 1,\ldots,n$ les variables pour lesquelles $\mu(x_i) \neq \perp$. Construisons les ψ-termes ψ_c et ψ de la façon suivante :

$$\psi_c = fonct(ch_1 = \mu(x_1),\ldots,ch_n = \mu(x_n))$$

$$\psi = fonct(ch_1 = x_1, \ldots, ch_n = x_n)$$

L'agent $B^W_{(\theta,\sigma)\to(\kappa,\sigma)}$ peut être défini comme

$$tell(\psi_c); get(\psi)$$

Observons en effet que l'histoire $((\theta,\sigma),(\theta,\sigma\cup\{\psi\})).((\theta,\sigma\cup\{\psi\}),(\theta\mu,\sigma))$ est une histoire de $\mathcal{D}_h(B^W_{(\theta,\sigma)\to(\kappa,\sigma)})$. Nous la noterons $\Gamma^W_{(\theta,\sigma)\to(\kappa,\sigma)}$

Il est important de noter que, cet agent manipulant des variables, il ne pourra pas être mis en parallèle avec un agent manipulant des variables communes sous peine de donner lieu à une sémantique dénotationnelle vide.

Définition 3.22 *Soient V un ensemble fini de ψ-termes, σ et τ deux états et θ une affectation de valeurs. Soient*

$$\begin{aligned} \sigma \setminus \tau &= \{g_1, \cdots, g_m\} \\ \tau \setminus \sigma &= \{t_1, \cdots, t_n\} \end{aligned}$$

avec $m, n \geq 0$. Soient $a_1, \ldots, a_{m+n}$ des ψ-termes clos qui ne soient ni dans V ni dans σ ni dans τ. Sans nous préoccuper d'avantage ici de ces a_i, nous désignons par $Ag^V_{\sigma\to\tau}$ l'agent

$$\begin{gathered} get(g_1); tell(a_1); \\ \cdots \\ get(g_m); tell(a_m); \\ tell(t_1); tell(a_{m+1}); \\ \cdots \\ tell(t_n); tell(a_{m+n}); \\ get(a_1); \cdots; get(a_{m+n}) \end{gathered}$$

Remarquons que, comme les primitives *get* ont toutes pour argument un ψ-terme clos, elles ne modifient pas l'affectation de valeurs, on a donc $Var(\mathcal{D}_h(Ag^V_{\sigma\to\tau})) = \emptyset$

Nous désignons par $\Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}$ la suite de situations commençant en (θ,σ) associée à $Ag^V_{\sigma\to\tau}$

$$\begin{gathered} ((\theta,\xi_0),(\theta,\gamma_1)).((\theta,\gamma_1),(\theta,\xi_1)). \\ \cdots \\ ((\theta,\xi_{m-1}),(\theta,\gamma_m)).((\theta,\gamma_m),(\theta,\xi_m)). \\ ((\theta,\xi_m),(\theta,\tau_1)).((\theta,\tau_1),(\theta,\xi_{m+1})). \\ \cdots \\ ((\theta,\xi_{m+n-1}),(\theta,\tau_n)).((\theta,\tau_n),(\theta,\xi_{m+n})). \\ ((\theta,\rho_0),(\theta,\rho_1)).\cdots.((\theta,\rho_{m+n-1}),(\theta,\rho_{m+n})) \end{gathered}$$

où

$$
\begin{array}{ll}
\xi_0 = \sigma & \\
\rho_0 = \xi_{m+n} & \\
\rho_{m+n} = \tau & \\
\gamma_i = \xi_{i-1} \setminus \{g_i\} & (1 \leq i \leq m) \\
\xi_i = \gamma_i \cup \{a_i\} & (1 \leq i \leq m) \\
\tau_j = \xi_{m+j-1} \cup \{t_j\} & (1 \leq j \leq n) \\
\xi_{m+j} = \tau_j \cup \{a_{m+j}\} & (1 \leq j \leq n) \\
\rho_k = \rho_{k-1} \setminus \{a_k\} & (1 \leq k \leq m+n)
\end{array}
$$

Evidemment, $Ag^V_{\sigma \to \tau}$ peut générer les histoires $\Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}.((\kappa,\gamma),\delta^+)$ pour toute affectation de valeurs κ telle que $\theta \prec \kappa$ et un état γ.

Proposition 3.6 *Soient σ et τ deux états et θ une affectation de valeur. Soit A un agent et V l'ensemble des ψ-termes présents dans les primitives tell, get et ask de A.*

1. *Toute histoire $h = \Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}.h'$ appartenant à l'ensemble $\mathcal{D}_h(Ag^V_{\sigma\to\tau} \mid\mid A)$ est dans l'ensemble $\Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}.((\kappa,\gamma),\delta^+) \mid\!\widetilde{\mid}_h h_a$ pour un certain état γ, une affectation κ, $\theta \prec \kappa$ et une histoire $h_a \in \mathcal{D}_h(A)$.*
2. *Pour tout agent B, toute histoire $h = \Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}.h'$ de $\mathcal{D}_h((Ag^V_{\sigma\to\tau} \ ; \ B) \mid\mid A)$ est dans l'ensemble $\Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}.h_b \mid\!\widetilde{\mid}_h h_a$ pour des histoires $h_a \in \mathcal{D}_h(A)$ et $h_b \in \mathcal{D}_h(B)$.*

Preuve. Etablissons la première partie de la proposition, la preuve de la seconde étant similaire. Par définition de la sémantique d'une composition parallèle, si h est une histoire de $\mathcal{D}_h(Ag^V_{\sigma\to\tau} \mid\mid A)$, alors, il existe des histoires $h_1 \in \mathcal{D}_h(Ag^V_{\sigma\to\tau})$ et $h_2 \in \mathcal{D}_h(A)$ telles que $h \in h_1 \mid\!\widetilde{\mid}_h h_2$. De plus, étant donné $Ag^V_{\sigma\to\tau}$, h_1 est nécessairement de la forme suivante :

$$
\begin{array}{c}
((\theta_1,\alpha_1),(\theta_1,\alpha_1 \setminus \{g_1\})).((\theta_2,\beta_1),(\theta_2,\beta_1 \cup \{a_1\})). \\
\cdots . \\
((\theta_{2m-1},\alpha_m),(\theta_{2m-1},\alpha_m \setminus \{g_m\})).((\theta_{2m},\beta_m),(\theta_{2m},\beta_m \cup \{a_m\})). \\
((\theta_{2m+1},\alpha_{m+1}),(\theta_{2m+1},\alpha_{m+1} \cup \{t_1\})).((\theta_{2m+2},\beta_{m+1}),(\theta_{2m+2},\beta_{m+1} \cup \{a_{m+1}\})). \\
\cdots \\
((\theta_{2m+2n-1},\alpha_{m+n}),(\theta_{2m+2n-1},\alpha_{m+n} \cup \{t_n\})).((\theta_{2m+2n},\beta_{m+n}),(\theta_{2m+2n},\beta_{m+n} \cup \{a_{m+n}\})). \\
((\theta_{2m+2n+1},\pi_1),(\theta_{2m+2n+1},\pi_1 \setminus \{a_1\})). \\
\cdots \\
((\theta_{3m+3n},\pi_{m+n}),(\theta_{3m+3n},\pi_{m+n} \setminus \{a_{m+n}\}))
\end{array}
$$

Etablissons progressivement que $h_1 = \Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}.h'_1$ pour une certaine histoire h'_1.

En utilisant les notations de la définition précédente, observons tout d'abord que h_2 ne peut pas être de la forme $((\theta,\xi_0),(\theta,\gamma_1)).h'_2$. En effet, si c'était le cas, alors, au vu de la définition de l'opérateur de parallélisme $\mid\!\widetilde{\mid}_h$, on aurait soit $h_1 = ((\theta,\gamma_1),(\theta,\xi_1)).h'_1$ soit $h'_2 = ((\theta,\gamma_1),(\theta,\xi_1)).h''_2$. Mais ces deux cas sont impossibles. Dans le premier cas, vu la forme de h_1, on devrait avoir $\xi_1 = \gamma_1 \setminus \{g_1\}$ et donc $a_1 \notin \xi_1$ puisque $a_1 \notin \gamma_1$ alors que par définition de ξ_1, $a_1 \in \xi_1$. Dans le second cas, puisque a_1 ne peut être déposé par A par choix de a_1, de nouveau $a_1 \notin \xi_1$ alors que par définition de ξ_1, $a_1 \in \xi_1$. Dès lors,

$$h_1 = ((\theta,\xi_0),(\theta,\gamma_1)).r_1$$

De plus, comme nous venons de le dire, par son choix, a_1 ne peut être déposé par A et par conséquent, h_2 ne peut être de la forme $h_2 = ((\theta,\gamma_1),(\theta,\xi_1)).h'_2$, quel que soit h'_2. Il s'en suit que

$$h_1 = ((\theta,\xi_0),(\theta,\gamma_1)).((\theta,\gamma_1),(\theta,\xi_1)).r_2$$

Par un raisonnement similaire, on prouve que h_1 est de la forme

$$\begin{aligned} h_1 = {} & ((\theta,\xi_0),(\theta,\gamma_1)).((\theta,\gamma_1),(\theta,\xi_1)).\cdots.((\theta,\xi_{m-1}),(\theta,\gamma_m)).((\theta,\gamma_m),(\theta,\xi_m)). \\ & ((\theta,\xi_m),(\theta,\tau_1)).((\theta,\tau_1),(\theta,\xi_{m+1})).\cdots((\theta,\xi_{m+n-1}),(\theta,\tau_n)).((\theta,\tau_n),(\theta,\xi_{m+n})).r_3 \end{aligned}$$

Maintenant, puisque par définition, A ne peut supprimer aucun a_i, h_1 doit être de la forme

$$\begin{gathered} ((\theta,\xi_0),(\theta,\gamma_1)).((\theta,\gamma_1),(\theta,\xi_1)). \\ \cdots. \\ ((\theta,\xi_{m-1}),(\theta,\gamma_m)).((\theta,\gamma_m),(\theta,\xi_m)). \\ ((\theta,\xi_m),(\theta,\tau_1)).((\theta,\tau_1),(\theta,\xi_{m+1})). \\ \cdots \\ ((\theta,\xi_{m+n-1}),(\theta,\tau_n)).((\theta,\tau_n),(\theta,\xi_{m+n})). \\ ((\theta,\rho_0),(\theta,\rho_1)). \\ \cdots \\ ((\theta,\rho_{m+n-1}),(\theta,\rho_{m+n})).r_4 \end{gathered}$$

En conclusion, l'agent $Ag^V_{\sigma\to\tau}$ a réalisé tous les 3*(m+n) pas et donc a été exécuté complètement avec succès. Il s'ensuit que r_4 doit être $((\kappa,\gamma),\delta^+)$, pour un certain état γ et une affectation κ telle que $\theta \prec \kappa$.

Définition 3.23 *Un ensemble d'histoires dénotationnelles est dit cohérent s'il est extensible (au sens défini dans la proposition 3.4), si son ensemble diff(S) est fini et s'il vérifie* $Var(S) \cap Ext(S) = \emptyset$.

Proposition 3.7 *Soient σ et τ deux états. Soit S un sous-ensemble cohérent de Shhist et V un ensemble contenant diff(S).*

1. *Toute histoire $h = \Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}.h'$ de l'ensemble $\mathcal{D}_h(Ag^V_{\sigma\to\tau}) \mathbin{\widetilde{\|}} S$ est dans l'ensemble $\Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}.((\kappa,\gamma),\delta^+) \mathbin{\widetilde{\|}_h} h_s$ pour un certain état γ, une affectation κ, $\theta \prec \kappa$ et une histoire $h_s \in S$.*
2. *Pour tout agent B, toute histoire $h = \Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}.h'$ de $\mathcal{D}_h(Ag^V_{\sigma\to\tau}\ ;\ B) \mathbin{\widetilde{\|}} S$ est dans l'ensemble $\Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}.h_b \mathbin{\widetilde{\|}_h} h_s$ pour des histoires $h_s \in S$ et $h_b \in \mathcal{D}_h(B)$.*

Preuve. La preuve s'obtient de façon analogue à celle de la proposition précédente.

3.6.4 Preuves de l'abstraction complète

Théorème 3.3 *Soient S_1 et S_2 deux sous-ensembles cohérents de Shhist qui vérifient $S_1 \setminus S_2 \neq \emptyset$. Soient $g \in S_1 \setminus S_2$ de longueur minimale et $init(g) = (\theta,\sigma)$. Alors, pour tout état α, il existe un agent T tel que $Var(\mathcal{D}_h(T)) \subseteq Ext(S_1)$ et une histoire continue $h \in (S_1 \mathbin{\widetilde{\|}} \mathcal{D}_h(T)) \setminus (S_2 \mathbin{\widetilde{\|}} \mathcal{D}_h(T))$ qui commence en (θ,α).*

Preuve. La preuve s'établit par induction sur la longueur minimale Lg d'une histoire qui est dans S_1 mais pas dans S_2.

Cas I : Lg=1. Dans ce cas, l'histoire $g \in S_1 \setminus S_2$ de longueur minimale est de la forme $((\theta,\sigma),\delta^+)$ ou $((\theta,\sigma),\delta^-)$.

Sous-cas i : $g = ((\theta,\sigma),\delta^+)$. Examinons d'abord le cas $g = ((\theta,\sigma),\delta^+)$. Par hypothèse, l'histoire $((\theta,\sigma),\delta^+) \notin S_2$. Soit V l'ensemble $\mathit{diff}(S_2)$. Considérons $T = Ag^V_{\alpha\to\sigma}$. Il vérifie $Var(\mathcal{D}_h(T)) = \emptyset$. Evidemment, $h = \Sigma^V_{(\theta,\alpha)\to(\theta,\sigma)}.((\theta,\sigma),\delta^+)$ est une histoire continue qui appartient à $(S_1 \mathbin{\widetilde{\|}} \mathcal{D}_h(T))$. Pour conclure ce cas, prouvons qu'il n'est pas dans $(S_2 \mathbin{\widetilde{\|}} \mathcal{D}_h(T))$. Si c'était le cas, par la proposition 3.7 h serait de la forme

$$h \in \Sigma^V_{(\theta,\alpha)\to(\theta,\sigma)}.((\kappa,\gamma),\delta^+) \mathbin{\widetilde{\|}}_h h_s$$

pour un état γ, une affectation de valeurs κ telle que $\theta \prec \kappa$ et une histoire $h_s \in S_2$. De plus, puisque h se termine, après $\Sigma^V_{(\theta,\alpha)\to(\theta,\sigma)}$, par $((\theta,\sigma),\delta^+)$, on devrait avoir, par définition de la composition parallèle, $\gamma = \sigma$, $\kappa = \theta$ et $h_s = ((\theta,\sigma),\delta^+)$. Dès lors, $((\theta,\sigma),\delta^+)$ appartiendrait à S_2 ce qui contredit notre hypothèse.

Sous-cas ii : $g = ((\theta,\sigma),\delta^-)$. Le cas où $g = ((\theta,\sigma),\delta^-)$ peut être traité de façon similaire, la fin de la preuve concluant que $h_s = ((\theta,\sigma),\delta^-)$ devrait appartenir à S_2, ce qui contredit l'hypothèse.

Cas II : Lg>1. Considérons maintenant le cas où le minimum des longueurs des histoires de $S_1 \setminus S_2$ est strictement supérieur à 1. Dans ce cas, l'histoire g de longueur minimale est de la forme $g = ((\theta,\sigma),(\kappa,\tau)).g'$ pour des états σ,τ, des affectations θ,κ et une histoire g' de $S_1[((\theta,\sigma),(\kappa,\tau))]$. Nous devons considérer deux cas : soit $S_2[((\theta,\sigma),(\kappa,\tau))] = \emptyset$, soit $S_2[((\theta,\sigma),(\kappa,\tau))] \neq \emptyset$ mais $g' \notin S_2[((\theta,\sigma),(\kappa,\tau))]$.

Sous-cas i : $S_2[((\theta,\sigma),(\kappa,\tau))] = \emptyset$. Si $S_2[((\theta,\sigma),(\kappa,\tau))] = \emptyset$, observons tout d'abord qu'il existe une histoire continue de la forme $((\theta,\sigma),(\kappa,\tau)).h_r$ dans S_1. Considérons V et $T = Ag^V_{\sigma\to\tau}$ comme dans le cas précédent. Il est évident que $h^* = \Sigma^V_{(\theta,\alpha)\to(\theta,\sigma)}.((\theta,\sigma),(\kappa,\tau)).h_r$ est une histoire continue commençant en (θ,α). De plus, si $((\rho,\omega),\delta)$ est la dernière paire de la suite h_r, alors, h^* appartient à la composition parallèle des histoires $\Sigma^V_{(\theta,\alpha)\to(\theta,\sigma)}.((\rho,\omega),\delta^+)$ et $((\theta,\sigma),(\kappa,\tau)).h_r$, et par conséquent, à $(S_1 \mathbin{\widetilde{\|}} \mathcal{D}_h(T))$. Pour conclure, il nous faut établir qu'elle n'appartient pas à $(S_2 \mathbin{\widetilde{\|}} \mathcal{D}_h(T))$. En effet, si c'était le cas, d'après la proposition précédente, h^* devrait s'écrire comme la composition parallèle de deux histoires de la forme $\Sigma^V_{(\theta,\alpha)\to(\theta,\sigma)}.((\phi,\gamma),\delta^+)$ et h_s avec $h_s \in S_2$. Selon la définition de la composition parallèle, on devrait alors avoir $h_s = ((\theta,\sigma),(\kappa,\tau)).h_r$ et donc $S_2[((\theta,\sigma),(\kappa,\tau))]$ ne serait pas vide ce qui contredit l'hypothèse.

Sous-cas ii : $S_2[((\theta,\sigma),(\kappa,\tau))] \neq \emptyset$ *mais* $g' \notin S_2[((\theta,\sigma),(\kappa,\tau))]$. Dans ce cas, par hypothèse $g' \in S_1[((\theta,\sigma),(\kappa,\tau))] \setminus S_2[((\theta,\sigma),(\kappa,\tau))]$ et le minimum des longueurs des histoires qui sont dans $S_1[((\theta,\sigma),(\kappa,\tau))]$ mais pas dans $S_2[((\theta,\sigma),(\kappa,\tau))]$ est strictement inférieur à Lg. Nous sommes donc en situation d'application de l'hypothèse d'induction. Désignons par (θ',σ') la situation $\mathit{init}(g')$. Remarquons que $Ext(S_1[((\theta,\sigma),(\kappa,\tau))]) \subseteq Ext(S_1)$ et $Var(S_1[((\theta,\sigma),(\kappa,\tau))]) \subseteq Var(S_1)$. L'application de l'induction, nous fournit, pour un état arbitraire α' (que nous spécifierons plus loin), un agent T' tel que $Var(\mathcal{D}_h(T')) \subseteq Ext(S_1[((\theta,\sigma),(\kappa,\tau))])$ et une histoire h^*_r commençant en (θ',α') qui est

dans $(S_1[((\theta,\sigma),(\kappa,\tau))] \widetilde{\|} \mathcal{D}_h(T')) \setminus (S_2[((\theta,\sigma),(\kappa,\tau))] \widetilde{\|} \mathcal{D}_h(T'))$. La preuve consiste maintenant à préfixer T' de certaines actions pour former T, et h_r^* d'une suite appropriée pour contruire h^*, de façon telle que h^* commence, comme demandé en (θ,α), soit continue et soit dans $(S_1 \widetilde{\|} \mathcal{D}_h(T))$ mais pas dans $(S_2 \widetilde{\|} \mathcal{D}_h(T))$. En appliquant la technique déjà utilisée, T peut commencer par $Ag^V_{\alpha\to\sigma}$ pour passer de la situation (θ,α) à (θ,σ), laisser alors S_1 et S_2 faire le pas $((\theta,\sigma),(\kappa,\tau))$, poursuivre avec $B^W_{(\kappa,\alpha')\to(\theta',\alpha')}$, où W désigne $diff(S_2)$ pour passer de la situation (κ,α') à la situation (θ',α') et terminer par T'. Pour forcer les S_i à faire ainsi, nous utilisons une astuce dont le principe consiste à ajouter dans h^*, après $((\theta,\sigma),(\kappa,\tau))$, un pas qui puisse être réalisé uniquement par T. Pour cela, choisissons t un nouveau ψ-terme qui n'apparaît ni dans $diff(S_1)$, ni dans $diff(S_2)$, ni dans les ψ-termes utilisés par $Ag^V_{\alpha\to\sigma}$ et posons $\alpha' = \tau \cup \{t\}$. Prenons maintenant

$$T = Ag^V_{\alpha\to\sigma} \ ; \ tell(t) \ ; \ B^W_{(\kappa,\alpha')\to(\theta',\alpha')} \ ; \ T'$$

et

$$h^* = \Sigma^V_{(\theta,\alpha)\to(\theta,\sigma)}.((\theta,\sigma),(\kappa,\tau)).((\kappa,\tau),(\kappa,\alpha')).\Gamma^W_{(\kappa,\alpha')\to(\theta',\alpha')}.h_r^*.$$

Notons que $Ag^V_{\alpha\to\sigma}$ et $tell(t)$ ne font intervenir aucune variable, $B^W_{(\kappa,\alpha')\to(\theta',\alpha')}$, ne modifient que des variables dont g attendait qu'elles soient affectées par l'environnement, c'est-à-dire $Var(B) \subseteq Ext(g) \subseteq Ext(S1)$ et l'agent T' ne fait intervenir que des variables de $Ext(S_1[((\theta,\sigma),(\kappa,\tau))]) \subseteq Ext(S_1)$. L'agent T ne fait donc intervenir aucune des variables de $Ext(S_1)$. Quant à l'histoire h^*, elle est dans $(S_1 \widetilde{\|} \mathcal{D}_h(T))$. En effet, puisque h_r^* est dans $S_1[((\theta,\sigma),(\kappa,\tau))] \widetilde{\|} \mathcal{D}_h(T'))$, il existe $h_1 \in S_1[((\theta,\sigma),(\kappa,\tau))]$ et $h_t \in \mathcal{D}_h(T'))$ tels que $h_r^* \in h_1 \widetilde{\|}_h h_t$. Dès lors, $((\theta,\sigma),(\kappa,\tau)).h_1 \in S_1$ et $\Sigma^V_{(\theta,\alpha)\to(\theta,\sigma)}.((\kappa,\tau),(\kappa,\alpha')).\Gamma^W_{(\kappa,\alpha')\to(\theta',\alpha')}.h_t \in \mathcal{D}_h(T)$. Nous avons donc obtenu que

$$h^* \in ((\theta,\sigma),(\kappa,\tau)).h_1 \widetilde{\|}_h \Sigma^V_{(\theta,\alpha)\to(\theta,\sigma)}.((\kappa,\tau),(\kappa,\alpha')).((\kappa,\alpha'),(\theta',\alpha')).h_t$$

et donc que h^* est dans $S_1 \widetilde{\|} \mathcal{D}_h(T)$.

Pour conclure, il reste à établir que $h^* \notin S_2 \widetilde{\|} \mathcal{D}_h(T)$. Il nous faut distinguer deux situations, selon que T fait intervenir ou non des variables de $Var(S_2)$.

Situation a. $Var(S_2) \cap Var(\mathcal{D}_h(T)) \neq \emptyset$. Dans ce cas, $S_2 \widetilde{\|} \mathcal{D}_h(T) = \emptyset$, et la démonstration est finie.

Situation b. $Var(S_2) \cap Var(\mathcal{D}_h(T)) = \emptyset$. Ici, nous procédons par contraposition. Si $h^* \in S_2 \widetilde{\|} \mathcal{D}_h(T)$, par la proposition précedente, h^* est dans l'ensemble $\Sigma^V_{(\theta,\alpha)\to(\kappa,\sigma)}.h_t \widetilde{\|}_h h_s$ pour certaines histoires $h_t \in \mathcal{D}_h(tell(t) \ ; \ B^W_{(\kappa,\alpha')\to(\theta',\alpha')} \ ; \ T')$ et $h_s \in S_2$. De plus, T ne peut être responsable du pas $((\theta,\sigma),(\kappa,\tau))$, en d'autres termes, h_t ne peut être de la forme $h_t = ((\theta,\sigma),(\kappa,\tau)).h_t'$. En effet, si c'était le cas, on aurait $\tau = \sigma \cup \{t\}$, alors que par définition $t \notin \tau$. Dès lors, $h_s = ((\theta,\sigma),(\kappa,\tau)).h_s'$ pour une certaine histoire h_s'. Notons que, puisque $h_s \in S_2$, $h_s' \in S_2[((\theta,\sigma),(\kappa,\tau))]$. Avançons d'un pas de plus dans h^*, de nouveau, grâce au choix de t, S_2 ne peut effectuer le pas $((\kappa,\tau),(\kappa,\alpha'))$ ce qui signifie que h_s' ne peut pas s'écrire $h_s' = ((\kappa,\tau),(\kappa,\alpha')).h_s''$. Vu le choix de W, aucun des pas de $\Gamma^W_{(\kappa,\alpha')\to(\theta',\alpha')}$ ne peut être effectué parh_s'. Dès lors, $h_t = ((\kappa,\tau),(\kappa,\alpha')).\Gamma^W_{(\kappa,\alpha')\to(\theta',\alpha')}.h_t'$.

En conclusion, $h_r^* \in h_t' \widetilde{\|}_h h_s'$ pour une histoire $h_t' \in \mathcal{D}_h(T')$ et une histoire $h_s' \in S_2[((\theta,\sigma),(\kappa,\tau))]$ et par conséquence, $h_r^* \in S_2[((\theta,\sigma),(\kappa,\tau))] \widetilde{\|} \mathcal{D}_h(T')$, ce qui contredit le fait que par construction h_r^* est dans $(S_1[((\theta,\sigma),(\kappa,\tau))] \widetilde{\|} \mathcal{D}_h(T')) \setminus (S_2[((\theta,\sigma),(\kappa,\tau))] \widetilde{\|} \mathcal{D}_h(T'))$.

Théorème 3.4 *La sémantique $\mathcal{D}_h$ est complètement abstraite par rapport à la sémantique $\mathcal{O}_h$.*

Preuve. Nous devons établir l'équivalence des deux propriétés suivantes :

i) pour tout contexte C, $\mathcal{O}_h(C[A_1]) = \mathcal{O}_h(C[A_2])$;

ii) $\mathcal{D}_h(A_1) = \mathcal{D}_h(A_2)$.

L'implication ii) $\Rightarrow$ i) découle directement de la fonction β définie précédemment.

L'autre implication i)$\Rightarrow$ ii) est démontrée par contraposition. Supposons $\mathcal{D}_h(A_1) \neq \mathcal{D}_h(A_2)$. Alors, il y a une histoire h qui est dans l'un des ensembles et pas dans l'autre. Nous pouvons supposer, sans perte de généralité, que $h \in \mathcal{D}_h(A_1)$, $h \notin \mathcal{D}_h(A_2)$ et $init(h) = (\theta, \sigma)$. Puisque $\mathcal{D}_h(A_1) \setminus \mathcal{D}_h(A_2) \neq \emptyset$, si l'on considère comme ensembles cohérents $S_1 = \mathcal{D}_h(A_1)$ et $S_2 = \mathcal{D}_h(A_2)$ et comme état initial $\emptyset$, le théorème 3.3 établit qu'il y a un agent T et une histoire continue $g \in (\mathcal{D}_h(A_1) \mathbin{\widetilde{||}} \mathcal{D}_h(T)) \setminus (\mathcal{D}_h(A_2) \mathbin{\widetilde{||}} \mathcal{D}_h(T))$ qui commence dans la situation $(\theta, \emptyset)$. En outre, $Var(\mathcal{D}_h(T)) \subseteq Ext(\mathcal{D}_h(A_1))$, c'est-à-dire $Var(\mathcal{D}_h(T)) \cap Var(\mathcal{D}_h(A_1)) = \emptyset$. Par la définition de la composition parallèle, nous savons que $g \in \mathcal{D}_h(A_1 \,||\, T) \setminus \mathcal{D}_h(A_2 \,||\, T)$. Considérons maintenant l'opérateur $S = B^{\emptyset}_{\emptyset \to \theta}$ et une histoire $((\emptyset, \emptyset), (\emptyset, \{\psi\})).((\emptyset, \{\psi\}), (\theta, \emptyset)).((\kappa, \tau), \delta^+) \in \mathcal{D}_h(B_{\emptyset \to \theta})$.

L'histoire $k = ((\emptyset, \emptyset), (\emptyset, \{\psi\})).((\emptyset, \{\psi\}), (\theta, \emptyset)).g$ est une histoire continue qui commence en $(\emptyset, \emptyset)$ et appartient à l'ensemble $\mathcal{D}_h(S) \mathbin{\widetilde{;}} (\mathcal{D}_h(A_1) \mathbin{\widetilde{||}} \mathcal{D}_h(T))$ mais pas à l'ensemble $\mathcal{D}_h(S) \mathbin{\widetilde{;}} (\mathcal{D}_h(A_2) \mathbin{\widetilde{||}} \mathcal{D}_h(T))$. La fonction β nous permet donc d'affirmer que $\bar{k}$ est une histoire opérationnelle de $\mathcal{O}_h(S; A_1 \,||\, T))$ mais pas de $\mathcal{O}_h(S; A_2 \,||\, T))$. Il y a donc un contexte $C = S; (\Box \,||\, T)$, tel que $\mathcal{O}_h(C[A_1]) \neq \mathcal{O}_h(C[A_2])$, ce qui termine la preuve.

Troisième partie

Méthodologie

Chapitre 4

Composition de spécifications

La définition de la sémantique dénotationnelle exposée à la section 3.4 a fait apparaître la nécessité, pour obtenir une sémantique compositionnelle, de tenir compte de toutes les exécutions possibles des agents, de les composer et de ne retenir que les histoires valides. Ceci représente une tâche redoutable ; une approche plus légère est souhaitable. Une première prespective proposée en [21] associe à chaque agent une trace constituée de la suite des interactions avec le tableau et de l'affectation de valeurs résultant de l'exécution de l'agent. Cette notion est très proche de la sémantique historique définie à la section 3.1 et pourrait être définie à partir de celle-ci. Pas plus que la sémantique historique, les traces ainsi obtenues ne pourraient être composées.

Dans le même article, une approche plus abstraite est proposée. Elle compose des spécifications écrites dans une logique temporelle de type Unity et s'appuie sur le principe de composition d'Abadi et Lamport [2]. C'est cette démarche que nous adaptons au langage $\mathcal{L}_\Psi$ dans ce chapitre. Au lieu de décrire le comportement d'un agent en terme d'histoire, nous allons maintenant le décrire en terme de spécifications. Celles-ci consistent essentiellement en formules du premier ordre pour une logique temporelle. Le comportement d'un ensemble d'agents est alors obtenu en combinant ces spécifications.

4.1 Une logique de programmation

Pour gagner en abstraction, nous considérons les modifications apportées au tableau à la fois par l'agent lui-même et par son environnement, c'est-à-dire l'ensemble des autres agents agissant sur le même tableau. Nous distinguons donc deux acteurs : l'agent considéré et son environnement. Puisque cet environnement est composé d'un certain nombre d'agents, il est techniquement adapté de considérer les deux acteurs comme des ensembles d'agents. Nous considérons dans la suite de ce travail un ensemble d'agents Sag infini dénombrable, nous désignerons par $\mathcal{P}_{ns}(Sag)$ l'ensemble des sous-ensembles stricts et non-vides de Sag. De tels ensembles sont désignés par la lettre α, éventuellement indicée, et leur complément dans Sag par $\tilde{\alpha}$.

Ceci nous amène à reformuler les règles de transition exposées dans la figure 1.3. Tout d'abord, nous introduisons des transitions ET, EA, EG, correspondant respectivement à l'exécution d'une primitive $tell$, ask ou get par le contexte de l'agent. En suite, nous associons à chaque règle un tag α ou $\tilde{\alpha}$ selon que l'agent ou son environnement est reponsable

de la transition. Les nouvelles règles ainsi formées sont les suivantes.

Définition 4.1 *Etant donné un ensemble d'agents $\alpha \in \mathcal{P}_{ns}(Sag)$, la relation de transition est définie comme la plus petite relation de $(Setat \times Sag \times Saffect) \times \mathcal{P}_{ns}(Sag) \times (Setat \times Sag \times Saffect)$ qui satisfait les règles de la figure 4.1. De façon pratique, la notation $C_1 \xrightarrow{\alpha} C_2$ est utilisée au lieu de $(C_1, \alpha, C_2) \in \rightarrow$. Comme précédemment, $A \parallel E$, $E \parallel A$ et $E; A$ sont remplacés par A.*

Les règles T à I réexpriment les règles déjà exposées sous les même noms. Les règles ET, EA et EG expriment les changements d'états dus à l'exécution par l'environnement, respectivement, d'une primitive *tell*, *ask* ou *get*. La condition qui apparait dans les règles EA et EG correspond à l'interdiction de mettre en parallèle des agents partageant des variables communes.

Cette relation de transition induit une modification dans la notion d'histoire en introduisant dans la transition l'ensemble d'agent responsable de cette transition. D'autre part, il n'est plus utile de considérer ici des histoires comportant des "trous". En effet, une exécution correspond à une succession de pas exécutés par un agent ou par son contexte. Puisque les uns et les autres sont pris en comptent par les nouvelles transitions, ils peuvent être présents au sein d'une histoire. Il n'y a plus lieu de considérer que des histoires continues.

Définition 4.2

1. *Nous désignons par ss, éventuellement indicé ou exposé, les situations de Ssituation.*
2. *L'ensemble des histoires d'exécution SEchist est l'ensemble des suites (éventuellement infinies) de la forme*

$$ss_1 \xrightarrow{\alpha_1} ss_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} ss_n \xrightarrow{\alpha_n} \cdots$$

   où $ss_1, ss_2, \ldots, ss_n \in Ssituation$, $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathcal{P}_{ns}(Sag)$.
3. *Quelle que soit l'histoire h de SEchist, nous désignerons par*
   (a) *$h[n]$, le préfixe de h de longueur n : $ss_1 \xrightarrow{\alpha_1} ss_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} ss_n$;*
   (b) *$lenght(h)$, la longueur de h ;*
   (c) *$h_{k,s}$, la k^{ieme} situation de h ;*
   (d) *$h_{k,a}$, le label α_k de la k^{ieme} transition de h.*
4. *L'ensemble SEchist peut être transformé en espace métrique complet en le dotant de la distance suivante $d : SEchist \times SEchist \rightarrow [0, 1]$: pour tout $h_1, h_2 \in SEchist$:*

$$d(h_1, h_2) = 2^{-sup\{n : h_1[n] = h_2[n]\}}$$

   avec la convention que $2^{-\infty} = 0$.
5. *Etant donné un ensemble $\alpha \in \mathcal{P}_{ns}(Sag)$, on dira que l'histoire $h = ss_1 \xrightarrow{\alpha_1} ss_2 \xrightarrow{\alpha_2} \cdots$ est α-normée si pour tout i, $\alpha_i \subseteq \alpha$ ou $\alpha_i \subseteq \tilde{\alpha}$*

Il nous sera utile de pouvoir comparer des histoires en ne prenant en compte que les pas qui modifient effectivement la situation ou de prolonger une histoire finie par des pas ne modifiant pas la situation. C'est pourquoi nous introduisons les notions suivantes.

$$(T) \quad \frac{clos(\psi_c)}{\langle tell(\psi_c)\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle E\ \&\ \theta \mid \sigma \cup \{\psi_c\}\rangle}$$

$$(A) \quad \frac{\psi \lhd \psi_c = \mu}{\langle ask(\psi)\ \&\ \theta \mid \sigma \cup \{\psi_c\}\rangle \xrightarrow{\alpha} \langle E\ \&\ \theta\mu \mid \sigma \cup \{\psi_c\}\rangle}$$

$$(N) \quad \frac{\nexists \psi_c : \psi_c \in \sigma, \psi \text{ correspond à } \psi_c}{\langle nask(\psi)\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle E\ \&\ \theta \mid \sigma\rangle}$$

$$(G) \quad \frac{\psi \lhd \psi_c = \mu}{\langle get(\psi)\ \&\ \theta \mid \sigma \cup \{\psi_c\}\rangle \xrightarrow{\alpha} \langle E\ \&\ \theta\mu \mid \sigma\rangle}$$

$$(S) \quad \frac{\langle A\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle A'\ \&\ \theta' \mid \sigma'\rangle}{\langle (A\ ;\ B)\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle (A'\ ;\ B)\ \&\ \theta' \mid \sigma'\rangle}$$

$$(P) \quad \frac{\langle A\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle A'\ \&\ \theta' \mid \sigma'\rangle, Var_a(A) \cap Var_a(B) = \emptyset}{\begin{array}{l}\langle (A \parallel B)\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle (A' \parallel B)\ \&\ \theta' \mid \sigma'\rangle \\ \langle (B \parallel A)\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle (B \parallel A')\ \&\ \theta' \mid \sigma'\rangle\end{array}}$$

$$(C) \quad \frac{\langle A\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle A'\ \&\ \theta' \mid \sigma'\rangle}{\begin{array}{l}\langle (A\ +\ B)\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle A'\ \&\ \theta' \mid \sigma'\rangle \\ \langle (B\ +\ A)\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle A'\ \&\ \theta' \mid \sigma'\rangle\end{array}}$$

$$(I) \quad \frac{executable(instr, \theta)}{\langle instr\ \&\ \theta \mid \sigma\rangle \xrightarrow{\alpha} \langle E\ \&\ \theta \mid \sigma\rangle}$$

$$(ET) \quad \frac{clos(\psi_c)}{\langle A\ \&\ \theta \mid \sigma\rangle \xrightarrow{\tilde{\alpha}} \langle A\ \&\ \theta \mid \sigma \cup \{\psi_c\}\rangle}$$

$$(EA) \quad \frac{\forall x \in Var(\alpha) : \theta\mu(x) = \theta(x)}{\langle A\ \&\ \theta \mid \sigma\rangle \xrightarrow{\tilde{\alpha}} \langle A\ \&\ \theta\mu \mid \sigma\rangle}$$

$$(EG) \quad \frac{\forall x \in Var(\alpha) : \theta\mu(x) = \theta(x)}{\langle A\ \&\ \theta \mid \sigma \cup \{\psi_c\}\rangle \xrightarrow{\tilde{\alpha}} \langle A\ \&\ \theta\mu \mid \sigma\rangle}$$

FIG. 4.1 – Règles de transition étiquetées associées à la grammaire *Sag*

Définition 4.3

1. *Quelle que soit l'histoire h de SEchist, on désigne par $\natural h$ l'histoire obtenue en remplaçant dans h toute séquence maximale $s \xrightarrow{\alpha_1} s \xrightarrow{\alpha_2} s \ldots$ par l'unique état s.*
2. *Quelles que soient les histoires h_1 et h_2 de SEchist, on dira qu'elles sont équivalentes à du bégaiement près si $\natural h_1 = \natural h_2$. On notera alors $h_1 \simeq h_2$.*
3. *Si h est une histoire de longueur m, on désigne par $\widehat{h}$ une histoire quelconque telle que $\widehat{h} \simeq h$ et $\widehat{h}[m] = h$.*

Cette nouvelle notion d'histoire amène une nouvelle sémantique qui associe à chaque agent un ensemble de telles histoires.

Définition 4.4 *Définissons la sémantique opérationnelle $\mathcal{O}_g^* : \mathcal{P}_{ns}(Sag) \times Sag \to \mathcal{P}(SEchist)$ comme suit : pour tout $\alpha \in \mathcal{P}_{ns}(Sag)$ et tout $A \in Sag$,*

$$
\begin{aligned}
\mathcal{O}_g^*(\alpha)(A) \;=\; & \{(\epsilon, \emptyset) \xrightarrow{\alpha_1} (\theta_2, \sigma_2) \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} (\theta_n, \sigma_n) : \\
& \quad \langle A \;\&\; \epsilon \mid \emptyset\rangle \xrightarrow{\alpha_1} \langle A_2 \;\&\; \theta_2 \mid \sigma_2\rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} \\
& \quad \langle A_n \;\&\; \theta_n \mid \sigma_n\rangle, \nrightarrow, \text{ et pour } i = 1, \ldots, n-1, \alpha_i \subseteq \alpha \text{ ou } \alpha_i \subseteq \tilde{\alpha}\} \\
& \cup \{(\epsilon, \emptyset) \xrightarrow{\alpha_1} (\theta_2, \sigma_2) \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} (\theta_n, \sigma_n) \cdots : \\
& \quad \langle A \;\&\; \epsilon \mid \emptyset\rangle \xrightarrow{\alpha_1} \langle A_2 \;\&\; \theta_2 \mid \sigma_2\rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} \\
& \quad \langle A_n \;\&\; \theta_n \mid \sigma_n\rangle, \xrightarrow{\alpha_n} \cdots, \text{ et pour } i = 1, \ldots, n, \alpha_i \subseteq \alpha \text{ ou } \alpha_i \subseteq \tilde{\alpha}\}
\end{aligned}
$$

Cette sémantique $\mathcal{O}_g^*$ ne diffère de la sémantique historique définie en 3.1 que par la possibilité de pas réalisés par l'environnement. Elle ne fournit guère plus d'abstraction. Après avoir élargi la sémantique opérationnelle à des histoires commençant dans une situation quelconque, nous allons y parvenir par les trois pas suivants :

- tout d'abord, en nous intéressant à des classes d'histoires
- ensuite, en identifiant certaines de ces classes
- enfin, en développant des règles pour raisonner sur ces classes.

Définition 4.5 *Définissons la sémantique opérationnelle $\mathcal{O}_g : \mathcal{P}_{ns}(Sag) \times Sag \to \mathcal{P}(SEchist)$ comme suit : pour tout $\alpha \in \mathcal{P}_{ns}(Sag)$ et tout $A \in Sag$,*

$$
\begin{aligned}
\mathcal{O}_g(\alpha)(A) \;=\; & \{(\theta_1, \sigma_1) \xrightarrow{\alpha_1} (\theta_2, \sigma_2) \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} (\theta_n, \sigma_n) : \\
& \quad \langle A \;\&\; \theta_1 \mid \sigma_1\rangle \xrightarrow{\alpha_1} \langle A_2 \;\&\; \theta_2 \mid \sigma_2\rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} \\
& \quad \langle A_n \;\&\; \theta_n \mid \sigma_n\rangle, \nrightarrow, \text{ et pour } i = 1, \ldots, n-1, \alpha_i \subseteq \alpha \text{ ou } \alpha_i \subseteq \tilde{\alpha}\} \\
& \cup \{(\theta_1, \sigma_1) \xrightarrow{\alpha_1} (\theta_2, \sigma_2) \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} (\theta_n, \sigma_n) \cdots : \\
& \quad \langle A \;\&\; \theta_1 \mid \sigma_1\rangle \xrightarrow{\alpha_1} \langle A_2 \;\&\; \theta_2 \mid \sigma_2\rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} \\
& \quad \langle A_n \;\&\; \theta_n \mid \sigma_n\rangle, \xrightarrow{\alpha_n} \cdots, \text{ et pour } i = 1, \ldots, n, \alpha_i \subseteq \alpha \text{ ou } \alpha_i \subseteq \tilde{\alpha}\}
\end{aligned}
$$

4.2 Logique de spécification

Un ensemble d'histoires clos pour l'équivalence $\simeq$ est considéré comme une propriété. Un agent est alors dit "satisfaire une propriété" si toutes les histoires d'exécution de sa

sémantique opérationnelle sont dans l'ensemble correspondant à la propriété.

Définition 4.6

1. *Une propriété est un ensemble d'histoire de SEchist clos pour l'équivalence* $\simeq$.
2. *Soient un agent* A *et l'ensemble* $\alpha \in \mathcal{P}_{ns}(Sag)$ *des ressources nécessaires à son exécution. L'agent* A *satisfait la propriété* P *ssi on a* $\mathcal{O}_g(\alpha)(A) \subseteq P$. *Ceci sera noté* $A\ sat_\alpha\ P$
3. *Bien qu'il s'agisse d'ensemble, nous commettrons l'abus de langage consistant, pour les propriétés* P *et* Q, *à parler de* $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$, $\neg P$, *pour signifier* $P \cap Q$, $P \cup Q$, $(Schist \setminus P) \cup Q$, *et* $Schist \setminus P$, *respectivement.*

4.2.1 Safety et Liveness

Nous allons à présent définir des propriétés de *safety* et *liveness*. De façon informelle, une propriété de *safety* est une propriété qui garantit que rien de mauvais n'arrive et une propriété de *liveness* assure que quelque chose de bien va arriver. En termes plus pragmatiques, les propriétés de *safety* correspondent à des invariants tandis que celles de *liveness* sont des propriétés d'évolution. Dans [3], Altern et Schneider montrent que ces définitions intuitives correspondent aux sous-ensembles clos et denses d'une certaine topologie. Il est intéressant ici d'observer qu'un corollaire à un raisonnement de la théorie générale de la topologie est que toute propriété est l'intersection d'une propriété de *safety* et d'une propriété de *liveness*. Nous présentons ces définitions et résultats pour les histoires d'exécution continues présentées dans la définition 4.2.

Considérons P, une propriété de *safety*, elle garantit que rien de mal n'arrive. Si une histoire h ne vérifie pas cette propriété, cela signifie qu'à un certain moment quelque chose de mal se passe. Ce "quelque chose de mal" doit être irrémédiable puisqu'une propriété de *safety* garantit que rien de mal n'arrive durant une exécution. Ceci suggère la définition suivante.

Définition 4.7 *Une propriété* P *est une propriété de safety si et seulement si*

$$\forall h \in SEchist, (h \notin P \Rightarrow (\exists n \geq 1 : (\forall k \in SEchist \text{ tel que } k[n] = h[n], \text{ on a } k \notin P)))$$

Cette définition permet d'observer certaines caractéristiques d'une propriété de *safety*. Tout d'abord, la définition ne pose aucune contrainte à propos du "quelque chose de mal" hormis qu'il doit être discret, c'est-à-dire, qu'il a lieu à un point précis de l'histoire. Ensuite, une propriété de *safety* ne peut jamais demander que quelque chose arrive parfois, par opposition à toujours. Une propriété de *safety* interdit inconditionnellement quelque chose de mal, et si ce quelque chose se réalise, il y a un point identifiable auquel ceci peut être repéré. Une propriété de *safety* est donc réfutable de façon finie. On peut encore noter que toute propriété qui ne dépend que de l'état initial est une propriété de *safety*.

Envisageons maintenant le cas d'une propriété de *liveness*, propriété qui garantit que quelque chose de bien se réalisera dans le futur. Une propriété de *liveness* ne peut interdire quelque chose de mal, elle peut juste imposer quelque chose de bien. seulement si tout préfixe fini peut être complété en une histoire de P.

Définition 4.8 *Une propriété P est une propriété de livenes si et seulement si*

$$\forall h \in SEchist \text{ et } \forall i \geq 1, \exists k \in SEchist : k[i] = h[i] \text{ et } k \in P$$

Observons ici quelques caractéristiques d'une propriété de *liveness*. Tout d'abord, la définition ne fait pas de restriction à propos du "quelque chose de bien", elle n'exige même pas qu'il s'agisse de quelque chose de discret. Le progrès peut consister en une suite infinie d'événements discrets. En cela, les "bonnes choses" diffèrent fondamentalement des "mauvaises choses". Ensuite, une propriété de *liveness* ne peut pas garantir que quelque chose de bien arrive toujours mais seulement qu'elle peut arriver. Une telle propriété n'est donc pas réfutable de façon finie.

Une propriété intéressante des définitions qui viennent d'être posées est qu'il y a une topologie de $SEchist$ pour laquelle les propriétés de *safety* sont exactement les ensembles clos et les propriétés des *liveness* sont exactement les ensembles denses. Les ensembles ouverts de base de cette topologie sont les ensembles de toutes les exécutions qui partagent un préfixe commun.

Définition 4.9 *L'ensemble $SEchist$ des histoires peut être muni de la topologie $\mathcal{T}$ définie de la façon suivante :*

- *les ensembles ouverts de base sont les ensembles de toutes les exécutions qui partagent un préfixe commun.*
- *un ensemble ouvert est l'union d'ensembles ouverts de base*
- *un ensemble fermé est le complémentaire d'un ensemble ouvert*
- *un ensemble dense est un ensemble qui intersecte tout ensemble ouvert non vide.*

Proposition 4.1 *Les propriétés de safety et de liveness sont respectivement les ouverts et les ensembles denses de cette topologie.*

Preuve. D'une part la définition d'une propriété de *safety* exprime exactement qu'elle est le complémentaire d'un ouvert. D'autre part, étant donné une propriété de *liveness*, quel que soit l'ouvert envisagé, le préfixe qui le caractérise peut être complété de façon à obtenir une histoire qui vérifie la propriété. C'est là la définition d'une propriété de *liveness*.

En outre, cette topologie fournit la propriété suivante.

Théorème 4.1 *Toute propriété P est l'intersection d'une propriété de safety et d'une propriété de liveness*

Preuve. Soient $\bar{P}$ la plus petite propriété de *safety* contenant P et L, le complémentaire de $\bar{P} \setminus P$. On a alors :

$$\begin{aligned} L \cap \bar{P} &= \complement(\bar{P} \setminus P) \cap \bar{P} \\ &= (\complement\bar{P} \cup P) \cap \bar{P} \\ &= (\complement\bar{P} \cap \bar{P}) \cup (P \cap \bar{P}) \\ &= (P \cap \bar{P}) \\ &= P. \end{aligned}$$

Il reste à montrer que L est dense et donc est une propriété de liveness. Par l'absurde, supposons qu'il y ait un ensemble O ouvert non vide contenu dans $\complement L$ c'est-à-dire que L ne soit pas dense. Alors $O \subseteq (\bar{P} \setminus P)$. Dès lors, $P \subseteq (\bar{P} \setminus O)$. L'intersection de deux ensembles fermés est un ensemble fermé, donc $P \setminus O$ est un ensemble fermé et par conséquent une propriété de *safety*. Ceci contredit l'hypothèse selon laquelle $\bar{P}$ est la plus petite propriété de *safety* contenant P.

Il est intéressant de noter que la topologie définie ici correspond à celle induite par la distance proposée dans la définition 4.2.

Proposition 4.2 *La topologie $\mathcal{T}$ est équivalente à celle induite par la distance d : $SEchist \times SEchist \rightarrow [0, 1]$: pour tout $h_1, h_2 \in SEchist$:*

$$d(h_1, h_2) = 2^{-sup\{n : h_1[n] = h_2[n]\}}$$

avec la convention que $2^{-\infty} = 0$.

Preuve. Nous allons démontrer que les deux topologies définissent les mêmes ensembles fermés.
Les fermés pour la topologie induite par la distance d sont des complémentaires des ouverts définis ci-dessus.
Considérons la suite $h_i (i \in N)$ d'histoires de $SEchist$ convergente au sens de la topologie des ouverts. Nous devons vérifier que si tous les éléments de la suite appartiennent à un fermé F de la topologie induite par d, alors sa limite h appartient elle aussi à F. Nous avons que

$$\forall n \geq 1, \exists i : h_j[n] = h[n] \text{ pour tout } j \geq i$$

c'est-à-dire

$$\forall n \geq 1, \exists i : d(h_j, h) \leq 2^{-n} \text{ pour tout } j \geq i.$$

Puisqu'en outre $h_i \in F$, la suite h_i est alors une suite d'éléments de F qui converge vers h au sens de la distance d, dès lors h est bien un élément de F.

Les complémentaires des ouverts définis ci-dessus sont des fermés pour la topologie induite par la distance d.
Considérons une suite $h_i (i \in N)$ d'histoires de $SEchist$ convergente au sens de la distance d. Nous devons vérifier que si tous ses éléments appartiennent à un fermé P, complémentaire d'un des ouverts définis ci-dessus, alors sa limite h appartient elle aussi à P. Nous avons que

$$\forall n \geq 1, \exists i : d(h_j, h) \leq 2^{-n} \text{ pour tout } j \geq i$$

c'est-à-dire

$$\forall n \geq 1, \exists i : h_i[n] = h[n].$$

Puisque $h_i \in P$ pour tout i, ceci suffit.

Cette propriété admet le corollaire suivant.

Proposition 4.3 *Une propriété P est une propriété de safety si et seulement si elle est fermée par rapport à la topologie induite par la distance de la définition 4.2.*

Dans [2] une introduction différente du concept de *safety* est proposée. Nous démontrons ici un critère qui établit l'équivalence de cette définition et de celle que nous avons exposée plus haut.

Critère 4.1 *Une propriété P est une propriété de safety si et seulement si elle satisfait la condition suivante :*

$$h \in P \text{ ssi pour tout } m \geq 1 \text{ on a } \widehat{h[m]} \in P.$$

Preuve Commençons par vérifier que cette condition est suffisante, nous montrerons en suite qu'elle est nécessaire.

Considérons une propriété P vérifiant la propriété du critère et une histoire h qui ne soit pas dans P. Par hypothèse, il existe un $m \geq 1$ tel que $\widehat{h[m]} \notin P$. Dès lors, pour toute histoire k vérifiant $k[m] = h[m]$, on a $k \notin P$. En effet, si k était dans P, on aurait $\widehat{k[m]} \in P$ avec $\widehat{k[m]} = \widehat{h[m]}$ ce qui est contraire à la définition de m. Ceci suffit à prouver que la propriété P est une propriété de *safety*.

Montrons maintenant que la condition est nécessaire. Considérons donc une propriété de *safety* P. Nous devons vérifier une condition nécessaire et suffisante. Nous procédons donc en deux étapes.

Etape 1. Condition suffisante. Soit une histoire h qui n'est pas dans P. Par définition d'une propriété de *safety*, nous savons qu'il existe un m pour lequel toute histoire $k \in SEchist$ vérifiant $k[m] = h[m]$ n'est pas dans P. L'histoire $\widehat{h[m]}$ en particulier, n'est pas dans P. Ce qui suffit.

Etape 2. Condition nécessaire. Soit une histoire h de P. Procédons par l'absurde en supposant qu'il existe un m pour lequel $\widehat{h[m]}$ ne soit pas dans P. La propriété P étant une propriété de *safety*, il existe alors un naturel $n \geq 1$ pour lequel toute histoire k de $SEchist$ vérifiant $k[n] = \widehat{h[m]}[n]$ ne soit pas dans P. Deux cas peuvent se présenter.

Cas 1. $n \leq m$. Dans ce cas $\widehat{h[m]}[n] = h[n]$ et l'on a une contradiction par le fait que h est dans P.

Cas 2. $n > m$. Dans ce cas $\widehat{h[m]}[n]$ peut s'écrire $h[m] \xrightarrow{\beta_1} h_{m,s} \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_{n-m}} h_{m,s}$ où les β nous importent peu. Nous nous intéressons alors à l'histoire h' définie par :

$$h' = h[m] \xrightarrow{\beta_1} h_{m,s} \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_{n-m}} h_{m,s} \xrightarrow{h_{m,a}} h_{m+1,s} \xrightarrow{h_{m+1,a}} h_{m+2,s} \xrightarrow{h_{m+2,a}} \ldots$$

Notons déjà que $h' \simeq h$. Les propriétés étant fermées pour $\simeq$ nous avons donc que h' est dans P comme l'est h. D'autre part, h' vérifie $h'[n] = \widehat{h[m]}[n]$, ce qui impose à h' de ne pas être dans P d'où la contradiction.

4.2.2 Unless, leads-to, initially

Les spécifications de type Unity proposées par Chandy et Misra [12] sont particulièrement adaptées à la description de programmes concurrents. Afin de combiner leurs propriétés avec le principe de composition d'Abadi et Lamport [2], Pierre Colette propose en [16] une reformulation qui introduit notamment la possibilité de distinguer un agent et son environnement. C'est cette proposition que nous adaptons ici au langage *Sag*.

L'opérateur $unless_\alpha$ fournit des formules qui imposent des contraintes uniquement sur les transitions exécutées par des agents de α. Une histoire h est dans $p\ unless_\alpha\ q$ si et seulement si toute transition de h due à un agent de α transforme une situation vérifiant $p \wedge \neg q$ en une situation vérifiant $p \vee q$. En d'autes mots, chaque fois que p est valide avant une transition due à un agent de α, p est valide après la transition à moins que q ne le soit. De façon formelle, on a la définition suivante.

Définition 4.10 *Soit $\alpha \in \mathcal{P}(Sag)$ un ensemble d'agents et p et q des assertions du premier ordre, l'opérateur $unless_\alpha$ est défini de la façon suivante :*

$$p\ unless_\alpha\ q \equiv \{h : \forall k \leq length(h) : (h_{k,s} \models p \wedge \neg q) \wedge (h_{k,a} \subseteq \alpha) \Rightarrow (h_{k+1,s} \models p \vee q)\}$$

A l'opérateur *unless* utilisé pour spécifier des propriétés de *safety*, s'ajoute un opérateur *leads to* permettant de spécifier des propriétés de *liveness*. La propriété p *leads to* q exprime que si p est valide un moment de l'histoire, alors q est ou sera vérifiée à un moment de la suite de l'histoire. Un opérateur *initialy* est utilisé pour spécifier les conditions initiales.

Définition 4.11 *Soit $\alpha \in \mathcal{P}(Sag)$ un ensemble d'agents et p et q des assertions du premier ordre, les opérateurs leadsto et initialy sont définis de la façon suivante :*

$$\begin{aligned} p\ leadsto\ q &\equiv \{h : \forall k \leq length(h) : \\ &\qquad \vee(h_{k,s} \models p) \Rightarrow (\exists j \in [k, length(h)] : h_{j,s} \models q)\} \\ initially\ p &\equiv \{h : h_{1,s} \models p\} \end{aligned}$$

Bien que ces trois opérateurs suffisent à notre propos, il est intéressant de mentionner encore les deux opérateurs classiques que sont *stable* et *invariant*.

Définition 4.12 *Soit $\alpha \in \mathcal{P}(Sag)$ un ensemble d'agents et p et q des assertions du premier ordre, les opérateurs stable et invariant sont définis de la façon suivante :*

$$\begin{aligned} stable_\alpha\ p &\equiv p\ unless_\alpha\ false \\ invariant\ p &\equiv \{h : h_{1,s} \models p, \forall k \leq length(h) : h_{k,s} \models p\} \end{aligned}$$

Il est évidemment possible de tenter de vérifier ces propriétés directement à partir des histoires d'exécutions. Toutefois, il est généralement plus simple et plus abstrait d'utiliser un système de preuve. Par exemple, on dispose des règles d'inférence suivantes. Une dérivation de ce système de preuve sera désignée par le symbole $\vdash\!\!\sim$.

Parmi les dérivations proposées par P. Colette [16] retenons les suivantes.

Proposition 4.4 *Les dérivations suivantes sont valides.*

1. *Affaiblissement de la conséquence.*

$$\frac{p\ unless_\alpha\ q \qquad invariant\ q \Rightarrow r}{p\ unless_\alpha\ r}$$

2. *Transitivité de leadsto.*

$$\frac{p \ leadsto \ q \qquad q \ leadsto \ r}{p \ leadsto \ r}$$

3. *Deux chemins.*

$$\frac{p \ leadsto \ q \vee r \qquad q \ leadsto \ r}{p \ leadsto \ r}$$

4. *Conjonction.*

$$\frac{p_1 \ unless_\alpha \ q_1 \qquad p_2 \ unless_\alpha \ q_2}{(p_1 \wedge p_2) \ unless_\alpha \ (p_1 \wedge q_2) \vee (p_2 \wedge q_1) \vee (q_1 \wedge q_2)}$$

5.

$$\frac{p \ unless_{\alpha_1} \ q \qquad p \ unless_{\alpha_2} \ q}{p \ unless_{\alpha_1 \cup \alpha_2} \ q}$$

6.

$$\frac{p \ unless_{\alpha_1} \ q \qquad \alpha_2 \subseteq \alpha_1}{p \ unless_{\alpha_2} \ q}$$

Preuve.

1. Par définition de $unless_\alpha$ toute histoire h de $p \ unless_\alpha \ q$ vérifie

$$\forall k \leq length(h) : (h_{k,s} \models p \wedge \neg q) \wedge (h_{k,a} \subseteq \alpha) \Rightarrow (h_{k+1,s} \models p \vee q).$$

L'hypothèse $q \Rightarrow r$ équivaut à $\neg r \Rightarrow \neg q$. On a donc $p \wedge \neg r \Rightarrow p \wedge \neg q$ et $p \vee q \Rightarrow p \vee r$. Les mêmes histoires vérifient donc

$$\forall k \leq length(h) : (h_{k,s} \models p \wedge \neg r) \wedge (h_{k,a} \subseteq \alpha) \Rightarrow (h_{k+1,s} \models p \vee r).$$

Ce qui suffit.

2. Considérons une histoire h qui appartienne à la fois à $p \ leadsto \ q$ et $q \ leadsto \ r$. Soit k tel que $h_{k,s} \models p \wedge \neg r$. Le fait que h appartient à $p \ leadsto \ q$ fournit un $j \in [k, length(h)]$ pour lequel $h_{j,s} \models q$. Le fait que h appartient à $q \ leadsto \ r$ fournit alors un $j' \in [j, length(h)] : h_{j',o} \models r$. Puisque $[j, length(h)] \subseteq [k, length(h)]$, ceci suffit pour conclure.

3. Considérons une histoire h qui appartienne à la fois à $p \ leadsto \ q \vee r$ et $q \ leadsto \ r$. Soit k tel que $h_{k,s} \models p \wedge \neg r$. Le fait que h appartient à $p \ leadsto \ q \vee r$ fournit un $j \in [k, length(h)] : h_{j,s} \models q \vee r$. Si $h_{j,s} \models r$, ce j permet de conclure, sinon $h_{j,s} \models q$ et le fait que h appartient à $q \ leadsto \ r$ fournit alors un $j' \in [j, length(h)] : h_{j',o} \models r$. Puisque $[j, length(h)] \subseteq [k, length(h)]$, ceci suffit pour conclure.

4. Toute histoire appartenant à la fois à $p_1 \ unless_\alpha \ q_1$ et à $p_2 \ unless_\alpha \ q_2$ vérifie

$$\forall k \leq length(h) :$$
$$(h_{k,s} \models ((p_1 \wedge \neg q_1) \wedge (p_2 \wedge \neg q_2)) \wedge (h_{k,a} \subseteq \alpha) \Rightarrow (h_{k+1,s} \models (p_1 \vee q_1) \wedge (p_2 \vee q_2)).$$

Or

$$(p_1 \wedge p_2) \wedge \neg((p_1 \wedge q_2) \vee (p_2 \wedge q_1) \vee (q_1 \wedge q_2)) = p_1 \wedge \neg q_1 \wedge p_2 \wedge \neg q_2$$

et

$$(p_1 \wedge p_2) \vee \neg((p_1 \wedge q_2) \vee (p_2 \wedge q_1) \vee (q_1 \wedge q_2)) = (p_1 \vee q_1) \wedge (p_2 \vee q_2)).$$

Ces histoires vérifient donc également la propriété :

$$(p_1 \wedge p_2)\ unless_{\alpha}\ (p_1 \wedge q_2) \vee (p_2 \wedge q_1) \vee (q_1 \wedge q_2).$$

5. Il est clair que toute histoire h appartenant à la fois à $p\ unless_{\alpha_1}\ q$ et à $p\ unless_{\alpha_2}\ q$, vérifie

$$\forall k \leq length(h) : (h_{k,s} \models p \wedge \neg q) \wedge (h_{k,a} \subseteq (\alpha_1 \cup \alpha_2)) \Rightarrow (h_{k+1,s} \models p \vee q).$$

Ce qui suffit.

6. Ici aussi il est évident que si une histoire h appartient à $p\ unless_{\alpha_1}\ q$ et que $\alpha_2 \subseteq \alpha_1$ alors, elle vérifie

$$\forall k \leq length(h) : (h_{k,i} \models p \wedge \neg q) \wedge (h_{k,a} \subseteq \alpha_2) \Rightarrow (h_{k,o} \models p \vee q).$$

Ce qui suffit.

Proposition 4.5 *Soit α un élément de $\mathcal{P}_{ns}(Sag)$. Les dérivations suivantes sont valides pour les histoires α-normées.*

1. *Progress-Safety-Progress*

$$\frac{p\ leadsto\ q \qquad r\ unless_{\alpha}\ b \qquad r\ unless_{\tilde{\alpha}}\ b}{p \wedge r\ leadsto\ (q \wedge r) \vee b}$$

2. *Invariant.*

$$\frac{initially\ p \qquad stable_{\alpha}\ p \qquad stable_{\tilde{\alpha}}\ p}{invariant\ p}$$

3. *Synchronisation.*

$$\frac{\begin{array}{lll} p\ leadsto\ q & q\ unless_{\alpha}\ s & q\ unless_{\tilde{\alpha}}\ s \\ p\ leadsto\ r & r\ unless_{\alpha}\ s & r\ unless_{\tilde{\alpha}}\ s \\ q \wedge r\ leadsto\ s & & \end{array}}{p \wedge r\ leadsto\ (q \wedge r) \vee b}$$

Preuve.

1. Considérons une histoire α-normée h vérifiant les trois hypothèses et un k tel que $h_{k,s} \models p \wedge r \wedge \neg(q \wedge r) \wedge \neg b$, c'est-à-dire $h_{k,s} \models p \wedge r \wedge \neg q \wedge \neg b$. L'histoire h étant α-normée, l'une des deux propriétés *unless* peut s'appliquer à chacune de ses transitions. Une application récursive des propriétés *unless* de l'hypothèse garantit que l'on est dans l'une des deux situations suivantes.
situation a. $\exists j \geq k : h_{j,s} \models b$, on a alors pour ce j, $h_{j,s} \models (q \wedge r) \vee b$.
situation b. $\forall j \geq k : h_{j,s} \models r$. La propriété *leadsto* de l'hypothèse fournit alors un $j \geq k$ tel que $h_{j,s} \models q$, et dès lors pour ce j, $h_{j,s} \models (q \wedge r) \vee b$
2. Toute histoire qui vérifie *initialy* p remplit évidemment la condition $h_{1,s} \models p$. Si, de plus, elle est α-normée et vérifie $stable_{\alpha}p$ et $stable_{\tilde{\alpha}}p$, on a pour tout k que $h_{k,s} \models p \Rightarrow h_{k+1,s} \models p$. Ceci suffit pour conclure en appliquant un raisonnement par récurrence que h vérifie $invariant\ p$.

3. Considérons une histoire h α-normée vérifiant les sept prémisses de la règle de synchronisation. Soit un $i \leq length(h)$ pour lequel $h_{i,s} \models p$, il nous faut trouver un $n \in [i, length(h)]$ tel que $h_{n,s} \models s$. Procédons par l'absurde, supposons qu'il n'existe pas de tel n. Puisque l'histoire h vérifie p *leadsto* q, il existe $j \in [i, length(h)]$ tel que $h_{j,s} \models q$. Puisqu'en outre h est α-normée, on peut appliquer à chacune de ses transitions, une des deux règles q $unless_{\alpha}$ s ou q $unless_{\tilde{\alpha}}$ s. Vu notre hypothèse qu'il n'existe aucun $n \in [i, length(h)]$ tel que $h_{n,s} \models s$, on a donc $h_{j+1,s} \models q$ et, en appliquant récursivement le même raisonnement, $h_{j',s} \models q$ pour tout $j' \in [j, length(h)]$. De la même façon, les prémisses p *leadstor*, r $unless_{\alpha}$ s et r $unless_{\tilde{\alpha}}$ s, permettent de conclure à l'existence d'un $k \in [i, length(h)]$ tel que $h_{k',s} \models q$ pour tout $k' \in [k, length(h)]$. Or $[j, length(h)] \cap [k, length(h)] \neq \emptyset$. Il existe donc $l \in [j, length(h)] \cap [k, length(h)]$ pour lequel $h_{l,s} \models q \wedge r$. Ce qui par la prémisse $q \wedge r$ *leadsto* s, impose l'existence d'un $n \in [l, lenght(h)]$, tel que $h_{n,s} \models s$, d'où l'absurdité.

4.3 Un principe de composition

Avant de composer les effets de plusieurs agents, nous devons d'abord spécifier les comportements des agents d'un manière abstraite adaptée. Les propriétés ci-dessus fournissent une bonne façon d'appréhender le comportement d'un agent. Toutefois, ce comportement dépend, d'une façon générale, du comportement de l'environnnement de cet agent. Nous sommes donc naturellement amenés à caractériser le comportement d'un agent au moyen d'une propriété vérifiée pour autant que l'environnement vérifie une autre propriété. En d'autres mots, le comportement d'un agent est spécifié par une formule de la forme $\mathcal{E} \Rightarrow \mathcal{G}$, où $\mathcal{E}$ est la propriété qui doit être vérifiée par l'environnement de l'agent et $\mathcal{G}$ la propriété qui sera alors vérifiée par l'agent.

Le pas suivant de la méthodologie de composition, consiste à caractériser les ensembles d'agents par des spécifications similaires en étendant la notion de satisfaire une propriété de façon directe à un ensemble d'agents.

Dans [2], Abadi et Lamport étudient la composition de spécifications indépendemment de tout langage de spécification et de toute logique. Considérant des systèmes qui interragissent avec leur environnement, le problème qu'ils envisagent est de prouver qu'un système composé satisfait ses spécifications si tous ses composants satisfont les leurs. Ils proposent le principe de composition suivant :

Principe 4.1 *Soient* Π *la composition de* $\Pi_1, \ldots \Pi_n$ *qui vérifient les conditions suivantes :*

1. Π *garantit* M *si tous les composants* Π_i *garantissent* M_i.
2. *Les hypothèses d'environnement* E_i *de chaque composant* Π_i *sont satisfaites si l'environnement de* Π *satisfait* E *et chaque* Π_j *satisfait* M_j.
3. *Chaque composant* Π_i *garantit* M_i *sous les hypothèses d'environnement* E_i.

Alors Π *garantit* M *sous les hypothèses d'environnement* E.

Ce principe, induisant un raisonnement circulaire, est valide sous certaines hypothèses appropriées dont le fait que les hypothèses d'environnement E et E_i soient limitées à des propriétés de *safety*. Dans [16], Pierre Collette a donné une caractérisation syntaxique

garantissant ces conditions. De façon suggérée par [21], cela donne, dans notre contexte, le principe suivant.

Principe 4.2 *Soient*
- $\Pi = \{A_1, \cdots, A_n\}$ *un ensemble d'agents exécutés en parallèle ;*
- $\mathcal{E}, \mathcal{E}_1, \ldots, \mathcal{E}_n$ *des propriétés de safety parmi lesquelles les propriétés unless sont désignées par* $\tilde{\alpha}, \tilde{\alpha_1}, \ldots, \tilde{\alpha_n}$*, respectivement ;*
- $\mathcal{G}, \mathcal{G}_1, \ldots, \mathcal{G}_n$ *des propriétés (générales) dont les parties safety ,* $\mathcal{G}^s, \mathcal{G}^s_1, \ldots, \mathcal{G}^s_n$*, ne contiennent aucune propriétés initially et ont* $\alpha, \alpha_1, \ldots, \alpha_n$ *pour label de leurs propriétés unless.*

Alors, la règle d'inférence suivante est valide :

$$\frac{\begin{array}{c} A_i \; sat_{\alpha_i} \; (\mathcal{E}_i \Rightarrow \mathcal{G}_i) \qquad i{=}1, \cdots, n \\ \mathcal{E}, \mathcal{G}^s_1, \cdots, \mathcal{G}^s_{i-1}, \mathcal{G}^s_{i+1}, \cdots, \mathcal{G}^s_n \vdash\!\sim \mathcal{E}_i \qquad i{=}1, \cdots, n \\ \mathcal{E}, \mathcal{G}^s_1, \cdots, \mathcal{G}^s_n \vdash\!\sim \mathcal{G}^s \\ \mathcal{E}, \mathcal{G}_1, \cdots, \mathcal{G}_n \vdash\!\sim \mathcal{G} \end{array}}{\Pi \; sat_{\alpha_1 \cup \cdots \cup \alpha_n} \; (\mathcal{E} \Rightarrow \mathcal{G})}$$

Notre contexte de travail, les règles de transition considérées nous permettent de simplifier de façon significative la preuve proposée dans [2] (cas $n = 2$). Commençons par établir deux lemmes.

Lemme 4.1 *Soient les agents* A *et* B *et les ensembles disjoints* α *et* $\beta \in \mathcal{P}_{ns}(Sag)$ *de ressources nécessaires à leurs exécutions respectives. Quelle que soit l'histoire*

$$h \in \mathcal{O}_g(\alpha \cup \beta)(A \parallel B)$$

on a

$$h \in \mathcal{O}_g(\alpha)(A) \qquad et \qquad h \in \mathcal{O}_g(\beta)(B)$$

Preuve. Au vu des transitions autorisées par les règles de la figure 4.1, la définition 4.5 nous permet d'affirmer qu'une histoire h de $\mathcal{O}_g(\alpha \cup \beta)(A \parallel B)$ peut s'écrire soit

$$h = (\theta_0, \sigma_0) \xrightarrow{\alpha_1} (\theta_1, \sigma_1) \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} (\theta_n, \sigma_n)$$

avec

$$\langle A_0 \parallel B_0 \;\&\; \theta_0 \mid \sigma_0 \rangle \xrightarrow{\alpha_1} \langle A_1 \parallel B_1 \;\&\; \theta_1 \mid \sigma_1 \rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} \langle A_n \parallel B_n \;\&\; \theta_n \mid \sigma_n \rangle, \not\rightarrow$$

et pour tout $i = 1, \ldots, n$

$$\alpha_i \subseteq \alpha \cup \beta \text{ ou } \alpha_i \subseteq \widetilde{\alpha \cup \beta}.$$

Ou bien

$$h = (\theta_0, \sigma_0) \xrightarrow{\alpha_1} (\theta_1, \sigma_1) \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} (\theta_n, \sigma_n) \xrightarrow{\alpha_{n+1}} \ldots$$

avec

$$\langle A_0 \parallel B_0 \;\&\; \theta_0 \mid \sigma_0 \rangle \xrightarrow{\alpha_1} \langle A_1 \parallel B_1 \;\&\; \theta_1 \mid \sigma_1 \rangle \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} \langle A_n \parallel B_n \;\&\; \theta_n \mid \sigma_n \rangle, \xrightarrow{\alpha_{n+1}} \ldots$$

et pour tout naturel i

$$\alpha_i \subseteq \alpha \cup \beta \text{ ou } \alpha_i \subseteq \widetilde{\alpha \cup \beta}.$$

Pour se convaincre qu'une telle histoire h est dans $h \in \mathcal{O}_g(\alpha)(A)$, il suffit d'observer une transition quelconque $\langle A_i \parallel B_i \;\&\; \theta_i \mid \sigma_i\rangle \xrightarrow{\alpha_{i+1}} \langle A_{i+1} \parallel B_{i+1} \;\&\; \theta_{i+1} \mid \sigma_{i+1}\rangle$ et de se convaincre qu'elle peut s'écrire $\langle A_i \;\&\; \theta_i \mid \sigma_i\rangle \xrightarrow{\alpha_{i+1}} \langle A_{i+1} \;\&\; \theta_{i+1} \mid \sigma_{i+1}\rangle$ avec $\alpha_{i+1} \subseteq \alpha$ ou $\alpha_i \subseteq \widetilde{\alpha}$. Il nous faut pour cela envisager trois cas.

cas 1. La transition est due à un agent du contexte. Dans ce cas $A_i = A_{i+1}$, $B_i = B_{i+1}$ et $\alpha_{i+1} \subseteq \widetilde{\alpha \cup \beta}$. Puisque $\widetilde{\alpha \cup \beta} \subseteq \widetilde{\alpha}$, on a la transition $\langle A_i \;\&\; \theta_i \mid \sigma_i\rangle \xrightarrow{\alpha_{i+1}} \langle A_{i+1} \;\&\; \theta_{i+1} \mid \sigma_{i+1}\rangle$ et le pas $(\theta_i, \sigma_i) \xrightarrow{\alpha_{i+1}} (\theta_{i+1}, \sigma_{i+1})$ est bien un pas d'une histoire de $\mathcal{O}_g(\alpha)(A)$.

cas 2. La transition est due à l'agent $A_i \parallel B_i$. Deux sous-cas se présentent

sous-cas i. La transition est due à l'agent A_i. Dans ce cas $B_i = B_{i+1}$ et $\alpha_{i+1} \subseteq \alpha$. On a alors la transition $\langle A_i \;\&\; \theta_i \mid \sigma_i\rangle \xrightarrow{\alpha_{i+1}} \langle A_{i+1} \;\&\; \theta_{i+1} \mid \sigma_{i+1}\rangle$ et le pas $(\theta_i, \sigma_i) \xrightarrow{\alpha_{i+1}} (\theta_{i+1}, \sigma_{i+1})$ est bien un pas d'une histoire de $\mathcal{O}_g(\alpha)(A)$.

sous-cas ii. La transition est due à l'agent B_i. Dans ce cas $A_i = A_{i+1}$ et $\alpha_{i+1} \subseteq \beta$. Puisque α et β sont disjoints, on a $\beta \subseteq \widetilde{\alpha}$, ce qui donne alors la transition $\langle A_i \;\&\; \theta_i \mid \sigma_i\rangle \xrightarrow{\alpha_{i+1}} \langle A_{i+1} \;\&\; \theta_{i+1} \mid \sigma_{i+1}\rangle$ et le pas $(\theta_i, \sigma_i) \xrightarrow{\alpha_{i+1}} (\theta_{i+1}, \sigma_{i+1})$ est bien un pas d'une histoire de $\mathcal{O}_g(\alpha)(A)$.

Lemme 4.2 *Soient un agent A, $\alpha \in \mathcal{P}_{ns}(Sag)$ l'ensemble des ressources nécessaires à son exécution, $\mathcal{E}$ et $\mathcal{G}$ des propriétés vérifiant*

- *$\mathcal{E}$ est une propriété de safety dont les propriétés unless ont pour label $\widetilde{\alpha}$.*
- *$\mathcal{G}$ est une propriété de safety ne contenant pas de propriété initialy et dont les propriétés unless ont pour label α.*
- *$A\ sat_\alpha\ (\mathcal{E} \Rightarrow \mathcal{G})$.*

Quelle que soit l'histoire h de $\mathcal{O}_g(\alpha)(A)$, si le naturel $i \geq 1$ vérifie $\widehat{h[i]} \notin \mathcal{G}$, alors $i > 1$ et $\widehat{h[i-1]} \notin \mathcal{E}$.

Preuve. Considérons une histoire h de $\mathcal{O}_g(\alpha)(A)$ qui vérifie $\widehat{h[i]} \notin \mathcal{G}$ et montrons que $i > 1$ et $\widehat{h[i-1]} \notin \mathcal{E}$.

Le fait que $i > 1$ est obtenu aisément. En effet, puisque $\mathcal{G}$ ne contient pas de propriété *initialy* mais uniquement des propriétés $unless_\alpha$, la propriété ne saurait être transgressée par l'état initial. Il faut qu'au moins un pas ait été exécuté.

Un résultat intermédiaire de notre démonstration consiste à vérifier que pour tout $j \geq 1$ si $\widehat{h[j]} \notin \mathcal{G}$ alors $\widehat{h[j]} \notin \mathcal{E}$. Puisque $\tilde{\alpha}$ est un ensemble non vide, la règle de transition EA de la figure 4.1 considérée pour un μ tel que $\theta\mu = \theta$, affirme que le pas $(h_{j,s} \xrightarrow{\tilde{\alpha}} h_{j,s})$ est un pas autorisé pour une histoire de $\mathcal{O}_g(\alpha)(A)$. Comme d'autre part $h \in \mathcal{O}_g(\alpha)(A)$, l'histoire $\widehat{h[j]} = h[j] \xrightarrow{\tilde{\alpha}} h_{j,s} \xrightarrow{\tilde{\alpha}} h_{j,s} \xrightarrow{\tilde{\alpha}} \ldots$ est donc une histoire de $\mathcal{O}_g(\alpha)(A)$. L'hypothèse $A\ sat_\alpha\ (\mathcal{E} \Rightarrow \mathcal{G})$, permet alors d'affirmer que $\widehat{h[j]} \in (\mathcal{E} \Rightarrow \mathcal{G})$, ce qui suffit pour établir le résultat intermédiaire.

Considérons maintenant le plus petit naturel j vérifiant $\widehat{h[j]} \notin \mathcal{G}$. L'hypothèse $\widehat{h[i]} \notin \mathcal{G}$ impose que $j \leq i$. Nous devons considérer deux cas.

Cas 1. $j < i$. La propriété $\mathcal{G}$ étant une propriété de $safety$, on a que pour tout $k \geq j$, $\widehat{h[k]} \notin \mathcal{G}$. Dès lors, $\widehat{h[i-1]} \notin \mathcal{G}$. Ce qui, par notre résultat intermédiaire, suffit pour affirmer que $\widehat{h[i-1]} \notin \mathcal{E}$

Cas 2. j=i. La définition de j et le fait que $i > 1$, garantissent que dans ce cas, $\widehat{h[i-1]} \in \mathcal{G}$. Puisque $\mathcal{G}$ ne contient que des propriétés $unless_\alpha$, nous déduisons de cette observation et de l'hypothèse $\widehat{h[i]} \notin \mathcal{G}$ que $h_{i-1,a} \subseteq \alpha$. D'autre part, l'application du résultat intermédiaire à cette hypothèse affirme que $\widehat{h[i]} \notin \mathcal{E}$. Puisque $\mathcal{E}$ est une propriété de $safety$ dont les propriétés $unless$ ont pour label $\widetilde{\alpha}$ et que $h_{i-1,a} \subseteq \alpha$ nous pouvons alors conclure que $\widehat{h[i]} \notin \mathcal{E}$.

Principe 4.3 *Soient*

- A_1 *et* A_2 *deux agents exécutés en parallèle et les ensembles disjoints* α_1 *et* $\alpha_2 \in \mathcal{P}_{ns}(Sag)$ *de ressources nécessaires à leurs exécutions respectives ;*
- $\mathcal{E}, \mathcal{E}_1, \mathcal{E}_2$ *des propriétés de safety dont les propriétés unless sont étiquetées respectivement par* $\widetilde{\alpha_1 \cup \alpha_2}, \widetilde{\alpha_1}, \widetilde{\alpha_n}$ *;*
- $\mathcal{G}, \mathcal{G}_1, \mathcal{G}_2$ *des propriétés (générales) dont les parties safety ,* $\mathcal{G}^s, \mathcal{G}_1^s, \mathcal{G}_2^s$*, ne contiennent aucune propriété initially et ont* $(\alpha_1 \cup \alpha_2), \alpha_1$*, et* α_2 *pour label de leurs propriétés unless.*

Alors, la règle d'inférence suivante est valide :

$$\frac{\begin{array}{ll} A_i \; sat_{\alpha_i} \; (\mathcal{E}_i \Rightarrow \mathcal{G}_i) \qquad i \in \{1,2\} & \\ \mathcal{E}, \mathcal{G}_1^s, \not\vdash\!\sim \mathcal{E}_2 \qquad \mathcal{E}, \mathcal{G}_2^s, \not\vdash\!\sim \mathcal{E}_1 & \\ \mathcal{E}, \mathcal{G}_1^s, \mathcal{G}_2^s \not\vdash\!\sim \mathcal{G}^s \qquad \mathcal{E}, \mathcal{G}_1, \mathcal{G}_2 \not\vdash\!\sim \mathcal{G} & \end{array}}{A_1 \;||\; A_2 \; sat_{\alpha_1 \cup \alpha_2} \; (\mathcal{E} \Rightarrow \mathcal{G})}$$

Preuve. Pour démontrer ce principe, il nous faut démontrer la validité de la règle d'inférence lorsque les hypothèses sont remplies. Pour cela nous considérons que les prémisses de la règle sont vérifiées et nous allons déduire sa conclusion.

Soit donc une histoire h de $\mathcal{O}_g(\alpha_1 \cup \alpha_2)(A_1 \;||\; A_2)$. Il nous faut prouver que cette histoire vérifie $(\mathcal{E} \Rightarrow \mathcal{G})$. Nous procédons pour cela par l'absurde.

Supposons donc que $h \notin (\mathcal{E} \Rightarrow \mathcal{G})$, c'est-à-dire $h \in \mathcal{E}$ et $h \notin \mathcal{G}$. De la prémisse $\mathcal{E}, \mathcal{G}_1, \mathcal{G}_2 \not\vdash\!\sim \mathcal{G}$, nous pouvons déduire que $h \notin \mathcal{G}_1 \cap \mathcal{G}_2$.

Le lemme 4.1, nous garantit que $h \in \mathcal{O}_g(\alpha_1)(A_1)$ et $h \in \mathcal{O}_g(\alpha_2)(A_2)$. Ce qui, au vu des prémisses, assure que $h \in (\mathcal{E}_1 \Rightarrow \mathcal{G}_1)$ et $h \in (\mathcal{E}_2 \Rightarrow \mathcal{G}_2)$. Puisque nous venons d'observer que $h \notin \mathcal{G}_1 \cap \mathcal{G}_2$, il en découle que

$$h \notin \mathcal{E}_1 \cap \mathcal{E}_2.$$

L'intersection $\mathcal{E}_1 \cap \mathcal{E}_2$ de deux propriétés de $safety$ est elle aussi une propriété de $safety$. Puisqu'elle n'est pas vérifiée par h, nous pouvons trouver un naturel i qui soit le plus petit naturel tel que $\widehat{h[i]} \notin \mathcal{E}_1 \cap \mathcal{E}_2$.

Des prémisses $\mathcal{E}, \mathcal{G}_1^s, \not\vdash\!\sim \mathcal{E}_2$ et $\mathcal{E}, \mathcal{G}_2^s, \not\vdash\!\sim \mathcal{E}_1$, nous pouvons alors déduire que $\widehat{h[i]} \notin \mathcal{E} \cap \mathcal{G}_1^s \cap \mathcal{G}_2^s$.

Par ailleurs, $\mathcal{E}$ étant une propriété de safety vérifiée par h, nous avons que $\widehat{h[i]} \in \mathcal{E}$. L'observation précédente devient donc $\widehat{h[i]} \notin \mathcal{G}_1^s \cap \mathcal{G}_2^s$.

Soit alors j dans $\{1,2\}$ tel que $\widehat{h[i]} \notin \mathcal{G}_j^s$. Le lemme 4.2 peut alors s'appliquer à $\mathcal{G}_j^s$ et $\mathcal{E}_j$ et conclut que $\widehat{h[i-1]} \notin \mathcal{E}_j^s$. Ce qui est en contradiction avec notre choix de i comme le plus petit naturel tel que $\widehat{h[i]} \notin \mathcal{E}_1^s \cap \mathcal{E}_2^s$.

Chapitre 5

Construction de programmes

L'objectif de ce chapitre est de fournir une méthodologie de construction de programmes comme composition d'agents. Nous cherchons à composer des agents satisfaisant certaines propriétés de façon à obtenir un système satisfaisant un ensemble de spécifications. La démarche proposée ici s'inspire de celle proposée par L. Semini en [27]. Toutefois, elle s'en distingue par la logique de spécification utilisée ainsi que par le langage considéré. Dans la première section, nous envisageons les propriétés de *safety* et montrons comment les composer pour obtenir la propriété plus générale d'un système satisfaisant un ensemble de contraintes de *safety*. Ensuite nous considérons les propriétés de *liveness* et nous terminons en introduisant la notion de système hétérogène et en proposant une technique de construction. Le chapitre se termine par une illustration de cette méthode. Il s'agit de dériver les spécifications des agents devant intervenir dans la construction d'un bâtiment.

5.1 Safety

Nous avons défini plus haut la notion de propriété de $safety$, propriété qui *garantit que rien de mal n'arrive.* Cette définition permet d'envisager des propriétés de formes très complexes. De façon pratique, nous nous limiterons ici à des formules de la forme $initialy\ p\ \wedge\ q\ unless_\alpha\ r$ auxquelles peuvent se ramener la plupart des propriétés de $safety$ utiles. Pour prouver qu'un système satisfait une telle propriété, il faut vérifier que Is vérifie la propriété attendue de l'état initial et que les comportements définis par les agents satisfont les propriétés $unless_\alpha$

Définition 5.1

1. *Un système* Σ *est un couple formé d'une situation initiale* Is *et d'un ensemble d'agents exécutés en parallèle* $A_1, \ldots, A_n$.
2. *Etant donné un ensemble d'agents* $\alpha \in \mathcal{P}_{ns}(\mathcal{L}_\Psi)$, *un système* $\Sigma = (Is, \{A_1, \ldots, A_n\})$ *satisfait la propriété* P *ssi on a* $\mathcal{O}_g(\alpha)(A) \cap \{h \ : \ h_{1,i} = Is\} \subseteq P$. *On notera* $\Sigma\ sat_\alpha\ P$

Proposition 5.1 *Soient* $\Sigma = (Is, \{A_1, \ldots, A_n\})$ *un système,* S *un ensemble de propriétés de* $safety$, $sf_i = initially\ p_i\ \wedge\ q_i\ unless_\alpha\ r_i$, *et* $\alpha \in \mathcal{P}_{ns}(\mathcal{L}_\Psi)$ *un ensemble d'agents. On a* $\Sigma\ sat_\alpha\ S$ *si :*

$$Is \vDash p_i\ \forall sf_i \in S$$

et il existe des α_j dont l'union rend α tels que

$$A_j \; sat_{\alpha_j} \; (q_i \; unless_{\alpha_j} \; r_i) \; \forall sf_i \in S \; and \; \forall j \in \{1, \ldots, m\}$$

Preuve. Puisque $\alpha = \cup\alpha_i$, les règles de dérivations mentionnées p. 49, nous permettent de déduire

$$q_i \; unless_{\alpha_1} \; r_i, \ldots, q_i \; unless_{\alpha_n} \; r_i \mid\sim q_i \; unless_{\alpha} \; r_i.$$

Avec ceci et les hypothèses $A_j \; sat_{\alpha_j} \; (q_i \; unless \; r_i)$, le principe de composition 4.2 permet de conclure que

$$(A_1 \parallel \ldots \parallel A_n) \; sat_{\alpha} \; q_i \; unless_{\alpha} \; r_i.$$

5.2 Liveness

Comme les propriétés de $safety$, les propriétés de *liveness*, *propriétés qui garantissent que quelque chose de bien arrive*, peuvent prendre des formes complexes. Nous envisageons ici les propriétés de la forme *p leadsto q*.

Proposition 5.2 *Soient $\Sigma = (Is, \{A_1, \ldots, A_n\})$ un système et $\alpha \in \mathcal{P}_{ns}(\mathcal{L}_\Psi)$, un ensemble d'agents. On a $\Sigma \; sat_\alpha \; (p \; leadsto \; q)$ si il existe des $\alpha_i (i = 1, ..., n)$ tels que les deux propositions suivantes soient vérifiées :*

1. $\cup\alpha_i = \alpha$
2. *il existe $k \in \{1, ..., n\}$ tel que $A_k \; sat_{\alpha_k} \; p \; leadsto \; q$*
   et pour tout $j \neq k$, $A_j \; sat_{\alpha_j} \; p \; unless_{\alpha_j} \; q$.

Preuve. Une application directe du principe de composition 4.2 donne

$$(A_1 \parallel \ldots \parallel A_n) \; sat_{\alpha} \; p \; leadsto \; q.$$

5.3 Systèmes hétérogènes

Partant des spécifications d'un programme, nous cherchons à le dériver comme composition d'agents vérifiant chacuns un ensemble de propriétés. L'hétérogénéïté consiste à envisager chacun des composants en supposant que les autres (considérés comme l'environnement) se comportent correctement.

Dans la perspective de l'hétérogénéïté, un système consiste en un ensemble de propriétés, un ensemble d'agents exécutés en parallèle, un état initial et un ensemble d'agents ressources : $(S, L, \{A_1, \ldots, A_n\}, Is, \alpha)$, où S et L sont respectivement des propriétés de $safety$ et de $liveness$.

Nous développons plus loin une technique de raffinements successifs d'un tel système hétérogène. A la première étape du processus, $n = 0$, $\alpha = \emptyset$ et S et L sont les spécifications du programme que l'on souhaite obtenir. A la dernière étape, $S = L = \emptyset$, et l'exécution parallèle des agents $A_1, \ldots, A_n$ vérifie les spécifications initialement posées sur le programme.

Définition 5.2

1. *Soit L un ensemble de propriétés de liveness et β un élément de $\mathcal{P}_{ns}(Sag)$. Nous désignons par $s_\beta(L)$ l'ensemble :*

$$s_\beta(L) = \{p \ unless_\beta \ q \mid p \ leadsto \ q \ \in \ L\}$$

2. *Soient S un ensemble de propriétés de safety et Is une situation. On écrira Is sat S si $Is \vDash p$ pour toute propriété initially $p \in S$.*

Définition 5.3 *Un système hétérogène est un quintuple $(S, L, \{A_1, \ldots, A_n\}, Is, \alpha)$ qui vérifie les deux propositions suivantes :*

1. *Is sat S*
2. *$(A_1 \parallel \ldots \parallel A_n)$ sat_α $S \cup s_\alpha(L)$*

Si à une étape donnée de la dérivation, le système $(S, L, \{A_1, \ldots, A_n\}, Is, \alpha)$ est hétérogène, nous exigeons non seulement que l'environnement satisfasse les propriétés de S et L, mais nous imposons aussi des contraintes aux agents déjà définis. En particulier, pour qu'il s'agisse d'un système hétérogène, nous imposons que $(A_1 \parallel \ldots \parallel A_n)$ sat_α S et $(A_1 \parallel \ldots \parallel A_n)$ sat_α p *unless* q pour toute propriété p *leadsto* q de L.

Proposition 5.3 *Soit $(S, L, \{A_1, \ldots, A_n\}, Is, \alpha)$ un système hétérogène, alors pour tout ensemble d'agents $\{A_{n+1}, \ldots, A_m\}$ vérifiant*

$$(A_{n+1} \parallel \ldots \parallel A_m) \ sat_\beta \ S \cup L \cup s_\beta(L)$$

on a

$$(A_1 \parallel \ldots \parallel A_m) \ sat_{\alpha \cup \beta} \ S \cup L \cup s_{\alpha \cup \beta}(L)$$

Preuve. Par application de la proposition 5.2.

Ce résultat exprime qu'un système hétérogène $(S, L, \{A_1, \ldots, A_n\}, Is, \alpha)$, satisfait les propriétés de S et L, si de nouveaux agents sont introduits pour "lui donner un coup de main".

La réflexion au sujet des systèmes hétérogènes nous amène à une méthodologie de dérivation. L'objectif est de pouvoir obtenir au départ d'un ensemble de spécifications, un ensemble d'agents $A_1, ..., A_n$ qui, exécutés en parallèle, vérifient ces spécifications. Nous obtenons ces agents par raffinements successifs d'un système hétérogène $(S, L, \emptyset, Is, \emptyset)$ où S est l'ensemble des conditions de *safety* des spécifications, L celui des conditions de *liveness* et Is un état initial qui satisfait les conditions de S. Les étapes de raffinement sont définies par les règles de dérivations exposées dans la définition suivante.

Quatre règles sont en fait appliquables. Dans le cas général, nous commençons avec un ensemble de propriétés qui peuvent aussi bien être des propriétés de *liveness* et des propriétés de *safety*. Un pas de dérivation d'"accomplisssement" est guidé par les propriétés de *liveness* (nous introduisons un agent qui "fait quelque chose de bien"), et doit satisfaire les propriétés de *safety* (l'agent introduit "ne doit rien faire de mal"). Cette règle de dérivation des systèmes hétérogènes va nous autoriser à annuler les propriétés de *liveness* qui sont satisfaites par l'agent introduit, en lui substituant les propriétés de *safety* appropriées.

Un pas de "spécification", permet d'ajouter des propriétés de *safety* et de *liveness* à un système hétérogène. Ceci ne peut se faire que pour autant que les règles introduites soient compatibles avec les agents déjà introduits dans le système et l'état initial. La dérivation "état initial" permet de modifier l'état initial tout en restant compatible avec les spécifications. Enfin, la dérivation "pas final" s'applique lorsque toutes les propriétés de *liveness* (initiales ou introduites en cours de dérivation) sont satisfaites. Le système hétérogène est alors de la forme $(S, \emptyset, \{A_1, \ldots, A_n\}, Is, \alpha)$ et le système exécutable $(\{A_1, \ldots, A_n\}, Is)$, remplit toutes les spécifications.

Définition 5.4 *Soit $SH = (S, L, \{A_1, \ldots, A_n\}, Is, \alpha)$ un système hétérogène, il peut être raffiné en un système SH' au moyen des quatre types de dérivations définis ci-dessous où $SH' \succ SH$ exprime que le système hétérogène SH' raffine SH.*

1. *spécification*

$$\frac{\begin{array}{ll} S' \cup L' \Rightarrow S \cup L & A_1 \parallel \ldots \parallel A_n sat_\alpha\ S' \setminus S \\ Is\ sat\ S' \setminus S & A_1 \parallel \ldots \parallel A_n sat_\alpha\ s(L' \setminus L) \end{array}}{(S', L', \{A_1, \ldots, A_n\}, Is, \alpha) \succ (S, L, \{A_1, \ldots, A_n\}, Is, \alpha)}$$

2. *état initial*

$$\frac{Is'\ sat\ S}{(S, L, \{A_1, \ldots, A_n\}, Is', \alpha) \succ (S, L, \{A_1, \ldots, A_n\}, Is, \alpha)}$$

3. *accomplissement*

$$\frac{B sat_\beta\ L' \cup s_\beta(L \setminus L') \cup S \qquad L' \subseteq L}{(S \cup s_{\tilde{\beta}}(L'), L \setminus L', \{A_1, \ldots, A_n, B\}, Is, \alpha \cup \beta) \succ (S, L, \{A_1, \ldots, A_n\}, Is, \alpha)}$$

4. *pas final*

$$\frac{L = \emptyset}{(\emptyset, \emptyset, \{A_1, \ldots, A_n\}, Is, \alpha) \succ (S, L, \{A_1, \ldots, A_n\}, Is, \alpha)}$$

Nous devons vérifier qu'il s'agit d'une bonne définition, c'est-à-dire que les systèmes introduits satisfont bien la définition 5.3 des systèmes hétérogènes. Pour les règles de spécification, état initial et accomplissement, ce résultat est une application de la proposition 5.2. Pour la règle de pas final, il est évident que $(\emptyset, \emptyset, \{A_1, \ldots, A_n\}, Is, \alpha)$ est un système hétérogène.

Proposition 5.4 *Soit $(\emptyset, \emptyset, \{A_1, \ldots, A_n\}, Is, \alpha)$ un système hétérogène obtenu par des dérivations (appliquant la définition 5.4) à partir de $(S, L, \emptyset, Is, \emptyset)$, alors $(A_1 \parallel \ldots \parallel A_m)$ satisfait les propriétés de liveness de L et préserve les conditions de safety de S*

Preuve. Par application de la définition 5.4 et de la propriété 5.3.

Proposition 5.5 *Soient $(S, L, \{A_1, \ldots, A_n\}, Is, \alpha)$ et $(S, L', \{A_{n+1}, \ldots, A_m\}, Is, \beta)$ des systèmes hétérogènes satisfaisant*

$$(S, L, \{A_1, \ldots, A_n\}, Is, \alpha) \succ (S_1, L_1, \emptyset, \emptyset)$$

et

$$(S, L', \{A_{n+1}, \ldots, A_m\}, Is, \beta) \succ (S_2, L_2, \emptyset, \emptyset)$$

pour certains S_1, L_1, α,S_2, L_2 *et* β. *Désignons par* L_n *l'ensemble des propriétés de liveness* $(L \setminus L_2) \cup (L' \setminus L_1)$, *alors*

$$\frac{A_1 \parallel \ldots \parallel A_n sat_\alpha\ s_\alpha(L' \setminus L) \qquad A_{n+1} \parallel \ldots \parallel A_m sat_\beta\ s_\beta(L \setminus L')}{(S, L_n, \{A_1, \ldots, A_m\}, Is, \alpha \cup \beta) \succ (S, L, \{A_1, \ldots, A_n\}, Is, \alpha)}$$

et

$$\frac{A_1 \parallel \ldots \parallel A_n sat_\alpha\ s_\alpha(L' \setminus L) \qquad A_{n+1} \parallel \ldots \parallel A_m sat_\beta\ s_\beta(L \setminus L')}{(S, L_n, \{A_1, \ldots, A_m\}, Is, \alpha \cup \beta) \succ (S, L, \{A_{n+1}, \ldots, A_m\}, Is, \beta)}$$

Preuve. Par application de la définition 5.4 et de la propriété 5.4

5.4 Application

Nous proposons ici une mise en oeuvre de la méthodologie que nous venons d'exposer. Le choix de l'exemple n'a pas été réalisé en fonction de son intérêt pratique mais selon des critères de type didactique. Il s'agit ici d'illustrer de la façon la plus détaillée possible les différentes étapes de la méthodologie. Nous avons aussi cherché à faire apparaître les principaux aspects du langage de coordination $\mathcal{L}_\Psi$, à savoir permettre la synchronisation d'une part et la communication d'informations d'autre part.

5.4.1 La situation

Nous envisageons la construction d'un bâtiment. Notre formalisation du problème met en scène 5 acteurs : un maître d'oeuvre, un maçon, un électricien, un couvreur, un menuisier. L'exécution d'un chantier se déroule de la façon suivante.

Le maître d'oeuvre définit les caractéristiques du chantier (pour l'exemple : largeur, longueur, nombre de portes et de fenêtres) et lance un appel à un maçon, un couvreur, un électricien et un menuisier. Le maçon est le premier à intervenir sur le chantier, il réalise le gros-oeuvre sur les prescriptions du maître d'oeuvre. Lorsqu'il a fini, il signale la fin de son travail. Dès que le gros-oeuvre a été reçu par le maître d'oeuvre, le couvreur, l'électricien et le menuisier peuvent commencer leurs tâches respectives selon les prescriptions établies par le maître d'oeuvre. Lorsque l'un d'eux a fini, il signale la fin de son travail. Chaque tâche est reçue par le maître d'oeuvre qui clôt le chantier.

5.4.2 Première spécification des agents

A chacun des acteurs de notre problème, nous allons associer un agent. Spécifions pour chacun la tâche qui lui correspond. Nous le faisons en terme de propriétés de *liveness* et *safety*. Nous considérons chaque agent comme isolé. Nous ne nous soucions pas des contraintes que doivent respecter les agents exécutés en parallèle pour permettre que leurs exécutions respectives se déroulent correctement. Nous utilisons dans un premier temps des expressions pour désigner les propositions. Nous examinerons plus loin les définitions formelles de chacune des propositions en terme de présence de ψ-terme(s) sur le tableau et de valeur d'affectation. Nous désignons par *myc* le nom du chantier à réaliser. La figure 5.1 propose une définition intuitive des propositions utilisées.

$sart$	=	Les conditions initiales nécessaires au début du chantier sont remplies
$Dma(myc)$	=	On demande un maçon pour le chantier myc
$Dc(myc)$	=	On demande un couvreur pour le chantier myc
$De(myc)$	=	On demande un électricien pour le chantier myc
$Dme(myc)$	=	On demande un menuisier pour le chantier myc
$Fma(myc)$	=	Le travail du maçon sur le chantier myc est fini
$Fc(myc)$	=	Le travail du couvreur sur le chantier myc est fini
$Fe(myc)$	=	Le travail de l'électricien sur le chantier myc est fini
$Fme(myc)$	=	Le travail du menuisier sur le chantier myc est fini
$Rma(myc)$	=	La travail du maçon sur le chantier myc est reçu
$Rmac(myc)$	=	Les travaux du maçon et du couvreur sur le chantier myc sont reçus
$Rmace(myc)$	=	Les travaux du maçon, du couvreur et de l'électricien sur le chantier myc sont reçus
$Rmaceme(myc)$	=	Les travaux du maçon, du couvreur, de l'électricien et du menuisier sur le chantier myc sont reçus
$F(myc)$	=	le chantier myc est terminé

FIG. 5.1 – Propositions

Le maître d'oeuvre. Six règles de *liveness* caractérisent le comportement du maître d'oeuvre.
La première correspond au lancement du chantier :

$$L_{mo1} = \text{start} \;\; leadsto\; Dma(myc) \wedge Dc(myc) \wedge De(myc) \wedge Dme(myc)$$

Les règles suivantes correspondent à la réception de chacune des tâches des corps de métiers. L'ordre est arbitrairement fixé (pour simplifier l'exposé).

$$\begin{aligned}
L_{mo2} &= Fma(myc)\; leadsto\; Rma(myc) \\
L_{mo3} &= Rma(myc) \wedge Fc(myc)\; leadsto\; Rmac(myc) \\
L_{mo4} &= Rmac(myc) \wedge Fe(myc)\; leadsto\; Rmace(myc) \\
L_{mo5} &= Rmace(myc) \wedge Fme(myc)\; leadsto\; Rmaceme(myc)
\end{aligned}$$

La sixième règle exprime la clôture du chantier.

$$L_{mo6} = Rmaceme(myc)\; leadsto\; F(myc)$$

Le maçon. Au lancement du chantier, sur base d'un appel à un maçon, le maçon réalise le gros-oeuvre selon les prescriptions du maître d'oeuvre. Lorsqu'il a fini, il signale la fin de son travail. Nous exprimons ce comportement par la règle de *liveness* suivante.

$$L_{ma} = Dma(myc)\; leadstoFma(myc)$$

Le couvreur. Dès que le maçon a fini le gros-oeuvre, le couvreur peut commencer la réalisation de la charpente et la pose du toit selon les prescriptions établies par le maître d'oeuvre. Lorsqu'il a fini, il signale la fin de son travail. Nous exprimons ce comportement par la règle de *liveness* suivante.

$$L_c = Dc(myc) \wedge Rma(myc)\; leadsto\; Fc(myc)$$

L'électricien. Dès que le maçon a fini le gros-oeuvre, l'électricien peut commencer l'installation électrique selon les prescriptions établies par le maître d'oeuvre. Lorsqu'il a fini, il signale la fin de son travail. Nous exprimons ce comportement par la règle de *liveness* suivante.

$$L_e = De(myc) \wedge Rma(myc)\; leadsto\; Fe(myc)$$

Le menuisier. Dès que le maçon a fini le gros-oeuvre, le menuisier peut commencer l'installation des portes et fenêtres selon les prescriptions établies par le maître d'oeuvre. Lorsqu'il a fini, il signale la fin de son travail. Nous exprimons ce comportement par la règle de *liveness* suivante.

$$L_{me} = Dme(myc) \wedge Rma(myc)\; leadsto\; Fme(myc)$$

5.4.3 Spécification globale

La résolution globale du problème se spécifie de façon simple par deux propriétés. La première consiste à imposer que les conditions initiales de travail soient vérifiées. La seconde exprime que si ces conditions sont remplies, le chantier doit être mené à son terme. Elles se formulent respectivement de la façon suivante.

$$initialy \text{ start}$$

$$\text{start } \textit{leadsto } F(myc)$$

5.4.4 Dérivation

Le système hétérogène de départ de notre dérivation est $(S_0, L_0, \emptyset, I_0, \emptyset)$où
- $S_0 = \{$ start$\}$
- $L_0 = \{$start $\textit{leadsto } F(myc)\}$
- $I_0 = (\emptyset, \emptyset)$.

Spécification en vue d'introduire *MO*. Aucun de nos agents ne peut satisfaire la condition de *liveness* telle qu'exprimée. Nous cherchons des ensembles S_1 et L_1 qui nous permettent d'appliquer la dérivation de spécifications et tels qu'une part de L_1 puisse être remplie par un de nos agents. Il serait agréable de pouvoir considérer l'ensemble $L_1 = \{L_{mo1}, \ldots, L_{mo6}, L_{ma}, L_c, L_e, L_{me}\}$

Ceci ne fournit malheureusement pas $L_1 \Rightarrow L_0$. En effet les règles L_{mo3},L_{mo4},L_{mo5}, L_c, L_e et L_{me} demandent que leurs prémisses soient vérifiées par un même état. Ceci nous amène à compléter l'ensemble des propriétés de *safety*. On définit l'ensemble $S_1 = S_0 \cup \{s_1^1, \ldots, s_1^9\}$ où

$$\begin{aligned}
s_1^1 &= Dc(myc) \; unless_{\tilde{\emptyset}} \; Fc(myc) \\
s_1^2 &= De(myc) \; unless_{\tilde{\emptyset}} \; Fe(myc) \\
s_1^3 &= Dme(myc) \; unless_{\tilde{\emptyset}} \; Fme(myc) \\
s_1^4 &= Fc(myc) \; unless_{\tilde{\emptyset}} \; Rmac(myc) \\
s_1^5 &= Fe(myc) \; unless_{\tilde{\emptyset}} \; Rmace(myc) \\
s_1^6 &= Fme(myc) \; unless_{\tilde{\emptyset}} \; Rmaceme(myc) \\
s_1^7 &= Rma(myc) \; unless_{\tilde{\emptyset}} \; Fini(myc) \\
s_1^8 &= Rmac(myc) \; unless_{\tilde{\emptyset}} \; Fini(myc) \\
s_1^9 &= Rmace(myc) \; unless_{\tilde{\emptyset}} \; Fini(myc)
\end{aligned}$$

où la notation $unless_{\tilde{\emptyset}}$ permet d'exprimer une propriété *unless* qui doit être vérifiée par toute transition. Les règles exposées aux propositions 4.4 et 4.5 permettent alors d'établir $S_1 \cup L_1 \Rightarrow S_0 \cup L_0$.

La preuve de ce résultat est détaillée en annexe.

Ceci nous permet d'appliquer la règle de dérivation par spécification et d'obtenir

$$(S_1, L_1, \emptyset, I_0, \emptyset) \succ (S_0, L_0, \emptyset, I_0, \emptyset)$$

Accomplissement par MO. Le choix de L_1 a été effectué de façon à nous permettre d'appliquer la règle d'accomplissement. Ceci demande toutefois l'ajout de condition de *safety* aux spécifications de l'agent MO associé au maître d'oeuvre. Si α_{mo} désigne l'ensemble des ressources nécessaires à l'exécution de l'agent MO, ces conditions sont $s_{\alpha_{mo}}\{L_{ma}, L_c, L_e, L_{me}\} \cup S_1$. L'application de la dérivation fournit alors

$$(S_2, L_2, \{MO\}, I_0, \{\alpha_{mo}\}) \succ (S_1, L_1, \emptyset, I_0, \emptyset)$$

où

- $S_2 = \{start, s_1^1, \ldots, s_1^9\} \cup s_{\widetilde{\alpha_{mo}}}(L_{mo1}, \ldots, L_{mo6})$
- $L_2 = \{L_{ma}, L_c, L_e, L_{me}\}$

Accomplissement par MA. La règle de *liveness* L_{ma} correspond à celle vérifiée par l'agent MA associé au maçon. Pour pouvoir appliquer la règle d'accomplissement, il nous faut ajouter des règles de *safety* à ses spécifications. Si α_{ma} désigne l'ensemble des ressources nécessaires à l'exécution de l'agent MA, ces conditions sont $s_{\alpha_{ma}}\{L_c, L_e, L_{me}\} \cup S_2$. L'application de la dérivation fournit alors

$$(S_3, L_3, \{MO, MA\}, I_0, \alpha_3) \succ (S_2, L_2, \{MO\}, I_0, \{\alpha_{mo}\})$$

où

- $S_3 = S_2 \cup s_{\widetilde{\alpha_{ma}}}(L_{ma})$
- $L_3 = \{L_c, L_e, L_{me}\}$
- $\alpha_3 = \alpha_{mo} \cup \alpha_{ma}$

Nous allons maintenant chercher à appliquer la dérivation d'accomplissement grâce aux agents E, C et ME successivement.

Accomplissement par E. L'agent E vérifie L_e, pour pouvoir l'introduire dans le système, nous devons imposer dans ses spécifications que

$$E \; sat_{\alpha_e} \; s_{\alpha_e}(L_c, L_{me}) \cup S_3.$$

Sous ces hypothèses, nous obtenons :

$$(S_4, L_4, \{MO, MA, E\}, I_0, \alpha_4) \succ (S_3, L_3, \{MO, MA\}, I_0, \{\alpha_3\})$$

où

- $S_4 = S_3 \cup s_{\widetilde{\alpha_e}}(L_e)$
- $L_4 = \{L_c, L_{me}\}$
- $\alpha_4 = \alpha_{mo} \cup \alpha_{ma} \cup \alpha_e$

Accomplissement par C. L'agent C vérifie L_c, pour pouvoir l'introduire dans le système, nous devons imposer dans ses spécifications que

$$C \; sat_{\alpha_c} \; s_{\alpha_c}(L_{me}) \cup S_4.$$

Sous ces hypothèses, nous obtenons :

$$(S_5, L_5, \{MO, MA, E, C\}, I_0, \alpha_5) \succ (S_4, L_4, \{MO, MA, E\}, I_0, \{\alpha_4\})$$

où

- $S_5 = S_5 \cup s_{\widetilde{\alpha_c}}(L_c)$
- $L_5 = \{L_{me}\}$
- $\alpha_5 = \alpha_{mo} \cup \alpha_{ma} \cup \alpha_e \cup \alpha_c$

Accomplissement par ME. L'agent ME vérifie L_{me}, pour pouvoir l'introduire dans le système, nous devons imposer dans ses spécifications que

$$ME\ sat_{\alpha_{me}}\ S_5.$$

Sous ces hypothèses, nous obtenons :

$$(S_6, L_6, \{MO, MA, E, C, ME\}, I_0, \alpha_6) \succ (S_5, L_5, \{MO, MA, E\}, I_0, \{\alpha_5\})$$

où

- $S_6 = S_5 \cup s_{\widetilde{\alpha_{me}}}(L_me)$
- $L_6 = \emptyset$
- $\alpha_6 = \alpha_{mo} \cup \alpha_{ma} \cup \alpha_e \cup \alpha_c \cup \alpha_{me}$

Reprenons ici l'ensemble des propriétés de *safety* présentes dans S_6. Ceci est réalisé dans la figure 5.2.

Pas final. L'ensemble des propriétés de *liveness* à établir étant maintenant vide, nous pouvons appliquer le pas final de la dérivation

$$(\emptyset, \emptyset, \{MO, MA, E, C, ME\}, I_0, \alpha_6) \succ (S_6, L_6, \{MO, MA, E, C, ME\}, I_0, \alpha_6)$$

5.4.5 Expression formelle des conditions

Avant de rédiger les agents, il nous faut maintenant exprimer les propositions que nous avons jusqu'ici laissées implicites en termes de contraintes sur les situations. Désignons par

- myc : le nom du chantier,
- *largeur* : la largeur de la construction,
- *longueur* : la longueur de la construction,
- np : le nombre de portes,
- nf : le nombre de fenêtre.

La base de la représentation du problème repose sur un ψ-terme associé au chantier. Il est de la forme :

$$\begin{aligned} chantier(&Cnom = myc, Cmacon = \ldots, Ccouvreur = \ldots, Celectricien = \ldots, \\ &Cmenuisier = \ldots, Clargeur = largeur, Clongueur = longueur, \\ &Cnbreporte = np, Cnbrefenetre = nf, Cetat = etat) \end{aligned}$$

où les ... peuvent prendre la valeur *demande* ou *recu* et *etat* peut prendre la valeur *encours* ou $fini$. Nous utilisons une série de ψ-termes pouvant lui correspondre. Ils sont définis sur la figure 5.3

start. La proposition start affirme que les conditions initiales requises pour le début du chantier sont vérifiées. De notre point de vue, il s'agit d'être certain qu'il n'y a pas un autre chantier du même nom déjà en cours de traitement sur le même tableau. Ceci se traduit par la proposition suivante.

$$start \quad = \quad \nexists\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi$$

$$
\begin{aligned}
s_0 &= start \\
\\
s_1^1 &= Dc(myc)\ unless_{\tilde{\emptyset}}\ Fc(myc) \\
s_1^2 &= De(myc)\ unless_{\tilde{\emptyset}}\ Fe(myc) \\
s_1^3 &= Dme(myc)\ unless_{\tilde{\emptyset}}\ Fme(myc) \\
s_1^4 &= Fc(myc)\ unless_{\tilde{\emptyset}}\ Rmac(myc) \\
s_1^5 &= Fe(myc)\ unless_{\tilde{\emptyset}}\ Rmace(myc) \\
s_1^6 &= Fme(myc)\ unless_{\tilde{\emptyset}}\ Rmaceme(myc) \\
s_1^7 &= Rma(myc)\ unless_{\tilde{\emptyset}}\ Fini(myc) \\
s_1^8 &= Rmac(myc)\ unless_{\tilde{\emptyset}}\ Fini(myc) \\
s_1^9 &= Rmace(myc)\ unless_{\tilde{\emptyset}}\ Fini(myc) \\
\\
s_2^1 &= \text{start}\ \ unless_{\widetilde{\alpha_{mo}}}\ Dma(myc) \wedge Dc(myc) \wedge De(myc) \wedge Dme(myc) \\
s_2^2 &= Fma(myc)\ unless_{\widetilde{\alpha_{mo}}}\ Rma(myc) \\
s_2^3 &= Rma(myc) \wedge Fc(myc)\ unless_{\widetilde{\alpha_{mo}}}\ Rmac(myc) \\
s_2^4 &= Rmac(myc) \wedge Fe(myc)\ unless_{\widetilde{\alpha_{mo}}}\ Rmace(myc) \\
s_2^5 &= Rmace(myc) \wedge Fme(myc)\ unless_{\widetilde{\alpha_{mo}}}\ Rmaceme(myc) \\
s_2^6 &= Rmaceme(myc)\ unless_{\widetilde{\alpha_{mo}}}\ F(myc) \\
\\
s_3 &= Dma(myc)\ unless_{\widetilde{\alpha_{ma}}}\ Fma(myc) \\
\\
s_4 &= Dc(myc) \wedge Rma(myc)\ unless_{\widetilde{\alpha_c}}\ Fc(myc) \\
\\
s_5 &= De(myc) \wedge Rma(myc)\ unless_{\widetilde{\alpha_e}}\ Fc(myc) \\
\\
s_6 &= Dme(myc) \wedge Rma(myc)\ unless_{\widetilde{\alpha_{me}}}\ Fc(myc)
\end{aligned}
$$

FIG. 5.2 – Propriétés de *safety* présentes dans S_6

$$
\begin{aligned}
\psi_{myc} &= chantier(Cnom = myc) \\
\psi_{dma} &= chantier(Cnom = myc, Cmacon = demande) \\
\psi_{dc} &= chantier(Cnom = myc, Ccouvreur = demande) \\
\psi_{de} &= chantier(Cnom = myc, Celectricien = demande) \\
\psi_{dme} &= chantier(Cnom = myc, Cmenuisier = demande) \\
\psi_{rma} &= chantier(Cnom = myc, Cmacon = recu) \\
\psi_{rmac} &= chantier(Cnom = myc, Cmacon = recu, Ccouvreur = recu) \\
\psi_{rmace} &= chantier(Cnom = myc, Cmacon = recu, Ccouvreur = recu, \\
&\qquad Celectricien = recu) \\
\psi_{rmaceme} &= chantier(Cnom = myc, Cmacon = recu, Ccouvreur = recu, \\
&\qquad Celectricien = recu, Cmenuisier = recu) \\
\psi_{fini} &= chantier(Cnom = myc, Cetat = fini)
\end{aligned}
$$

$$
\begin{aligned}
\phi_{tma} &= fini(Cnom = myc, Couvrier = macon) \\
\phi_{tc} &= fini(Cnom = myc, Couvrier = couvreur) \\
\phi_{te} &= fini(Cnom = myc, Couvrier = electricien) \\
\phi_{tme} &= fini(Cnom = myc, Couvrier = menuisier)
\end{aligned}
$$

FIG. 5.3 – Quelques ψ-termes particuliers.

Dma(myc). La proposition $Dma(myc)$ doit exprimer que le chantier myc est en attente d'un maçon. Le choix posé ici est de placer dans le ψ-terme de description du chantier un champ $Cmacon$ qui prend la valeur "demande" lorsque le chantier demande un maçon. Ceci se traduit par la proposition suivante.

$$\text{Dma(myc)} \quad = \quad \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi, \text{ on a } \psi_{dma} \text{ correspond à } \psi$$

Fma(myc). La proposition $Fma(myc)$ doit exprimer que le maçon a terminé la tâche relative au chantier myc. Le choix posé ici est de placer sur le tableau un ψ-terme de la forme $\phi = fini(Cnom = myc, Couvrier = macon)$. La proposition se traduit alors par

$$\text{Fma(myc)} \quad = \quad \exists\, \phi \in \sigma : \phi = \phi_{tma}$$

Rma(myc). La proposition $Rmamyc$ doit exprimer que le travail du maçon sur le chantier myc est reçu. Ceci est réalisé en donnant au champ $Cmacon$ du ψ-terme décrivant le chantier la valeur $recu$.

$$\text{Rma(myc)} \quad = \quad \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi \text{ on a } \psi_{rma} \text{ correspond à } \psi$$

Dc(myc), ..., Fc(myc),..., Rmac(myc),... On procède de façon analogue pour les autres corps de métiers. Ceci nous permet d'exprimer l'ensemble des propositions de la façon reprise dans la figure 5.4

F(myc). La fin du chantier est exprimée en plaçant la valeur $fini$ dans le champs $Cetat$ du ψ-terme décrivant le chantier.

$$\text{F(myc)} \quad = \quad \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi \text{ on a } \psi_{fini} \text{ correspond à } \psi$$

5.4.6 Spécifications complètes et implémentation

Il s'agit maintenant de reprendre un à un les agents, de lister l'ensemble des spécifications qu'ils doivent satisfaire et de proposer une implémentation.

Une difficuté particulière se révèle au moment de l'implémentation. Les agents du langage $\mathcal{L}_\Psi$ étant finis, il ne leur est pas possible de satisfaire une propriété *leadsto* en toute généralité. Une alternative s'offre à nous. La première possibilité consiste à fermer notre système une fois tous les agents déterminés et à nous appuyer sur quelques propriétés supplémentaires. La seconde à tolérer l'introduction d'agents infinis.

Considérons tout d'abord la première possibilité. Notre système est formé de cinq agents identifiés. Nous pouvons, sans modifier le déroulement du chantier, ajouter la propriété de *safety stable* $F(myc)$. Celle-ci combinée avec les règles de S_6, vérifiées par le

$$
\begin{aligned}
start &= \nexists\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi \\
Dma(myc) &= \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi, \text{ on a } \psi_{dma} \text{ correspond à } \psi \\
Dc(myc) &= \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi, \text{ on a } \psi_{dc} \text{ correspond à } \psi \\
De(myc) &= \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi, \text{ on a } \psi_{de} \text{ correspond à } \psi \\
Dme(myc) &= \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi, \text{ on a } \psi_{dme} \text{ correspond à } \psi \\
Fma(myc) &= \exists\, \phi \in \sigma : \phi = \phi_{tma} \\
Fc(myc) &= \exists\, \phi \in \sigma : \phi = \phi_{tc} \\
Fe(myc) &= \exists\, \phi \in \sigma : \phi = \phi_{te} \\
Fme(myc) &= \exists\, \phi \in \sigma : \phi = \phi_{tme} \\
Rma(myc) &= \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi \text{ on a } \psi_{rma} \text{ correspond à } \psi \\
Rmac(myc) &= \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi \text{ on a } \psi_{rmac} \text{ correspond à } \psi \\
Rmace(myc) &= \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi \text{ on a } \psi_{rmace} \text{ correspond à } \psi \\
Rmaceme(myc) &= \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi \text{ on a } \psi_{rmaceme} \text{ correspond à } \psi \\
\mathrm{F(myc)} &= \forall\, \psi \in \sigma : \psi_{myc} \text{ correspond à } \psi \text{ on a } \psi_{fini} \text{ correspond à } \psi
\end{aligned}
$$

FIG. 5.4 – Expression formelle des propositions

système, garantit en outre *stable Rma(myc)*, *stable Rmac(myc)*, *stable Rmace(myc)* et *stable Rmaceme(myc)*. Ce qui permet d'affirmer pour chaque agent qu'après son exécution les conditions de son exécution ne se présenteront plus. En s'appuyant sur ce résultat, fournir un agent qui réalise une fois les modifications exigées par la propriété *leadsto* suffit pour que la propriété soit effectivement satisfaite.

La seconde possibilité permettrait d'élargir l'application à plusieurs chantiers. Mais elle sort du cadre du langage $\mathcal{L}_\Psi$. Elle demanderait de revoir la définition de la sémantique associée au langage, pour permettre la prise en compte d'agents infinis.

Nous proposons ici une implémentation réalisée en suivant la première perspective.

Le maître d'oeuvre. Outre les propriétés de *liveness* L_{mo1} à L_{mo6}, l'agent MO associé au maître d'oeuvre doit vérifier les propriétés s_1^1 à s_1^9 (où l'on remplace $\tilde{\emptyset}$ par α_{ma}) et

$$\begin{aligned}
s_3^{mo} &= Dma(myc)\ unless_{\alpha_{mo}}\ Fma(myc)\\
s_4^{mo} &= Dc(myc) \wedge Rma(myc)\ unless_{\alpha_{mo}}\ Fc(myc)\\
s_5^{mo} &= De(myc) \wedge Rma(myc)\ unless_{\alpha_{mo}}\ Fc(myc)\\
s_6^{mo} &= Dme(myc) \wedge Rma(myc)\ unless_{\alpha_{mo}}\ Fc(myc)
\end{aligned}$$

Remarquons que $MO\ sat_{\alpha_{mo}}(s_1^1 \wedge \cdots \wedge s_1^9) \Rightarrow MO\ sat_{\alpha_{mo}}(s_4^{mo} \wedge s_5^{mo} \wedge s_6^{mo})$.

Le code complet de l'agent MO est fourni en annexe. Contentons-nous d'observer ici les éléments correspondant à l'application de l'agent. Le tableau parametre[ ] contient largeur, longueur, np, nf, myc.

```
String chantierold;
String chantiernew;
try{
     //verifier start
     repertoire = (Registry)LocateRegistry.getRegistry(hote);
     un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
     chantiernew= new String("chantier(Cnom="+parametre[4]+")");
     un_serveur.nask(new PsiTerme(chantiernew));

     //ouvrir le chantier
     chantiernew= new String("chantier(Cnom="+parametre[4]+
            ",Cmacon=demande,Ccouvreur=demande,Celectricien=demande,
            Cmenuisier=demande,Clargeur="+parametre[0]+",Clongueur="+
            parametre[1]+",Cnbreporte="+parametre[2]+",Cnbrefenetre="
            +parametre[3]+",etat=encours)");
     un_serveur.tell(new PsiTerme(chantiernew));

     //attendre le maçon
     String attendu=new String("fini(Cnom="+parametre[4]+",Couvrier=macon)");
     un_serveur.get(new PsiTerme(attendu));
```

```
//recevoir macon
chantierold = chantiernew;
chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=recu,
        Ccouvreur=demande,Celectricien=demande,Cmenuisier=demande,
        Clargeur="+parametre[0]+",Clongueur="+parametre[1]+",
        Cnbreporte="+parametre[2]+",Cnbrefenetre="+parametre[3]+
        ",etat=encours)");
un_serveur.tell(new PsiTerme(chantiernew));
un_serveur.get(new PsiTerme(chantierold));

//attendre le couvreur
attendu=new String("fini(Cnom="+parametre[4]+",Couvrier=couvreur)");
un_serveur.get(new PsiTerme(attendu));
//recevoir macon
chantierold = chantiernew;
chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=recu,
        Ccouvreur=recu,Celectricien=demande,Cmenuisier=demande,
        Clargeur="+parametre[0]+",Clongueur="+parametre[1]+",Cnbreporte="
        +parametre[2]+",Cnbrefenetre="+parametre[3]+",etat=encours)");
un_serveur.tell(new PsiTerme(chantiernew));
un_serveur.get(new PsiTerme(chantierold));

//attendre l'électricien
attendu=new String("fini(Cnom="+parametre[4]+",Couvrier=electricien)");
un_serveur.get(new PsiTerme(attendu));
//recevoir macon
chantierold = chantiernew;
chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=recu,
        Ccouvreur=recu,Celectricien=recu,Cmenuisier=demande,Clargeur="
        +parametre[0]+",Clongueur="+parametre[1]+",Cnbreporte="
        +parametre[2]+",Cnbrefenetre="+parametre[3]+",etat=encours)");
un_serveur.tell(new PsiTerme(chantiernew));
un_serveur.get(new PsiTerme(chantierold));

//attendre le menuisier
attendu=new String("fini(Cnom="+parametre[4]+",Couvrier=menuisier)");
un_serveur.get(new PsiTerme(attendu));
//recevoir menuisier
chantierold = chantiernew;
chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=recu,
        Ccouvreur=recu,Celectricien=recu,Cmenuisier=recu,Clargeur="+
        parametre[0]+",Clongueur="+parametre[1]+",Cnbreporte="+
        parametre[2]+",Cnbrefenetre="+parametre[3]+",etat=encours)");
un_serveur.tell(new PsiTerme(chantiernew));
un_serveur.get(new PsiTerme(chantierold));

//clore chantier
```

```
      chantierold = chantiernew;
      chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=recu,
               Ccouvreur=recu,Celectricien=recu,Cmenuisier=recu,Clargeur="+
               parametre[0]+",Clongueur="+parametre[1]+",Cnbreporte="+
               parametre[2]+",Cnbrefenetre="+parametre[3]+",etat=fini)");
      un_serveur.tell(new PsiTerme(chantiernew));
      un_serveur.get(new PsiTerme(chantierold));
      }//ftry

catch (RemoteException e){e.printStackTrace();}
catch (NotBoundException e){e.printStackTrace();}
```

Le maçon. Outre la propriété de *liveness* L_{ma}, l'agent MA associé au maçon doit vérifier les propriétés s_1^1 à s_1^9, s_2^1 à s_2^6, s_4, s_5 et s_6 où les indices de toutes les propriétés *unless* sont remplacés par α_{ma}.

Nous proposons ici une implémentation plus générique que celle nécessaire à la vérification des spécifications. La propriété de *liveness* L_{ma} est remplacée par L'_{ma}.

$$L'_{ma} = (\exists x : Dma(x))\ leadstoFma(x)$$

Cette modification n'a aucune influence sur la dérivation effectuée plus haut.

Le code complet de l'agent MO est fourni en annexe. Contentons-nous d'observer ici les éléments correspondant à l'application de l'agent.

```
try{
      //attendre demande macon
      repertoire = (Registry)LocateRegistry.getRegistry(hote);
      un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
      chantier= new String("chantier(Cnom=?VMAnom,Cmacon=demande,
               Clargeur=?VMAlargeur,Clongueur=?VMAlongueur,Cnbreporte=
               ?VMAnbreporte,Cnbrefenetre=?VMAnbrefenetre)");
      AffectMacon=un_serveur.ask(new PsiTerme(chantier));

      //realiser le travail...
      //les paramètres de description du chantier ont été récupérés dans
      //les variables VMAnom, VMAlargeur, VMAlongueur,VMAnbreporte et
      //VMAnbrefenetre


      //annoncer fin du travail
      String message=new String("fini(Cnom="+
               AffectMacon.valeurString("VMAnom")+",Couvrier=macon)");
      un_serveur.tell(new PsiTerme(message));

      }//ftry
```

```
catch (RemoteException e){e.printStackTrace();}
catch (NotBoundException e){e.printStackTrace();}
```

Le couvreur, l'électricien et le maçon. Ces trois agents se traitent de façon similaire au précédent. Donnons ici à titre d'exemple la partie significative du code du couvreur.

```
try{
     //attendre demande couvreur
     repertoire = (Registry)LocateRegistry.getRegistry(hote);
     un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
     chantier= new String("chantier(Cnom=?VCnom,Cmacon=recu,Ccouvreur=
               demande,Clargeur=?VClargeur,Clongueur=?VClongueur,
               Cnbreporte=?VCnbreporte,Cnbrefenetre=?VCnbrefenetre)");
     AffectCouvreur=un_serveur.ask(new PsiTerme(chantier));

     //realiser le travail
     //les paramètres de description du chantier ont été récupérés dans
     //les variables VCnom, VClargeur, VClongueur,VCnbreporte et
     //VCnbrefenetre


     //annoncer fin du travail
     String message=new String("fini(Cnom="+
           AffectCouvreur.valeurString("VCnom")+",Couvrier=couvreur)");
     un_serveur.tell(new PsiTerme(message));
     }

     catch (RemoteException e){e.printStackTrace();}
     catch (NotBoundException e){e.printStackTrace();}
```

Quatrième partie

Conclusion

Chapitre 6

Conclusions

6.1 Travail effectué

Notre mémoire a d'abord proposé la définition d'un langage de coordination $\mathcal{L}_\Psi$. Il repose sur quatre primitives *tell*, *ask*, *nask* et *get* qui permettent la consultation et l'interaction avec un tableau. Ce tableau est un lieu de dépôt, de consultation et de saisie d'information structurée sous la forme de ψ-termes. Nous avons proposé une implémentation en Java-RMI de ce langage.

Considérant une situation, état du système, comme un couple formé d'un état du tableau et d'une affectation de valeur aux variables utilisées pour la communication, nous avons développé une sémantique sur base d'histoires, successions de situations. Cette sémantique n'est pas compositionnelle, ce qui est une propriété particulièrement intéressante pour la mise en parallèles de différents agents. Nous avons alors proposé une sémantique dénotationnelle dont nous avons prouvé la compositionnalité ainsi que le caractère complètement abstrait.

Dans une perspective de développement, nous nous sommes en suite intéressés à la possibilité de spécifier des agents en termes de propriétés de *safety* et de *liveness*. Ceci nous a amené a envisager différentes approches de ces concepts. Nous les avons reformulé dans notre contexte et démontré un principe de composition de telles spécifications. Enfin, ce principe a servi de point d'appui à la méthodologie de développement que nous avons proposée et illustrée dans le dernier chapitre.

6.2 Travaux futurs

La mise en oeuvre de la méthodologie a mis en évidence l'impossibilité pour un agent fini de vérifier une propriété de *liveness* de type *leadsto* de façon absolue. Par le principe de composition 4.3, nous avons montré que la composition d'agents peut s'effectuer sur base de contraintes moins fortes. La propriété de *liveness* attendue d'un agent peut reposer sur la vérification de propriétés de *safety* par l'ensemble du système. Un pas de dérivation pourrait sans doute être proposé qui permette à l'agent introduit dans un système hétérogène de s'appuyer sur les conditions de *safety* imposées au système. Il serait alors également intéressant de formaliser une méthode permettant de déterminer si un agent

vérifie des spécifications d'un tel type.

Cinquième partie

Annexes

Annexe A

Utilisation de Java-RMI

Pour tester l'application PingPong :

- Créer un répertoire serveur et un répertoire client sur les machines respectives (éventuellement plusieurs clients). Il convient ici de s'assurer que le répertoire .../bin dans lequel se trouvent les exécutables de java (java, javac, rmiregistry...) se trouvent bien dans la variable d'environnement PATH et que les répertoires qui viennent d'être créés se trouvent dans le CLASSPATH de leurs machines respectives.
- Copier PsiTermeSpace.java et TachePsiTermeSpace.java dans le répertoire serveur (ainsi que les packages affectation, psiterme, item et variable)
- Compiler PsiTermeSpaceImpl.java (javac PsiTermeSpaceImpl.java). Deux nouveaux fichiers sont crées : PsiTermeSpaceImpl.class et TachePsiTermeSpace.class.
- Exécuter RMI sur PsiTermeSpaceImpl (rmic -v1.2 PsiTermeSpaceImpl). Un nouveau fichier est créé : PsiTermeSpaceImpl_Stub.class
- Copier PingPong.java (resp. Ping.java, Pong.java) (et les packages affectation, psiterme, item et variable), PsiTermeSpaceImpl_Stub.class et TachePsiTermeSpace.class dans le(s) répertoire(s) client(s).
- Compiler PingPong.java, Ping.java, Pong.java. Il convient de s'assurer préalablement que dans le code de ces fichiers, le *LocateRegistry* a comme paramètre l'adresse IP du serveur. De nouveaux fichiers sont créés : Ping.class, Pong.class et PingPong.class.
- Ouvrir une console sur le serveur et une sur chaque client.
- Lancer l'annuaire sur le serveur (rmiregistry & sous Unix/ rmiregistry sous Win).
- Lancer le tableau (java PsiTermeSpaceImpl). Il se lance et ne rend pas la main. Le message <- serveur operationnel -> s'affiche.
- Sur les machines clientes, lancer PingPong, Ping et Pong. Une fois lancés, la partie de pingpong commence entre Ping et Pong.

Pour tester l'application à la construction, on procède de façon analogue. Il est possible de transmettre quelques arguments aux différents agents.

L'exécutable MaitreOeuvre admet les arguments suivants

- -u=<adresse IP du Serveur>
- -l=<largeur du chantier>
- -L=<Longueur du chantier>
- -p=<nombre de portes du chantier>

- -f=<nombre de fenêtres du chantier>
- -n=<nom du chantier>

Les exécutables Macon, Couvreur, Electricien et Menuisier admettent l'argument -u=<adresse IP du Serveur>

Annexe B

Preuve de la dérivation de l'application

Nous avons
- $S_0 = \{$ start$\}$
- $L_0 = \{$start $\ leadsto\ F(myc)\}$
- $L_1 = \{L_{mo1}, \ldots, L_{mo6}, L_{ma}, L_c, L_e, L_{me}\}$
- $S_1 = S_0 \cup \{s_1^1, \ldots, s_1^9\}$ où

$$
\begin{aligned}
s_1^1 &= Dc(myc)\ unless_{\tilde{\emptyset}}\ Fc(myc)\\
s_1^2 &= De(myc)\ unless_{\tilde{\emptyset}}\ Fe(myc)\\
s_1^3 &= Dme(myc)\ unless_{\tilde{\emptyset}}\ Fme(myc)\\
s_1^4 &= Fc(myc)\ unless_{\tilde{\emptyset}}\ Rmac(myc)\\
s_1^5 &= Fe(myc)\ unless_{\tilde{\emptyset}}\ Rmace(myc)\\
s_1^6 &= Fme(myc)\ unless_{\tilde{\emptyset}}\ Rmaceme(myc)\\
s_1^7 &= Rma(myc)\ unless_{\tilde{\emptyset}}\ Fini(myc)\\
s_1^8 &= Rmac(myc)\ unless_{\tilde{\emptyset}}\ Fini(myc)\\
s_1^9 &= Rmace(myc)\ unless_{\tilde{\emptyset}}\ Fini(myc)
\end{aligned}
$$

Nous devons prouver

$$S_1 \cup L_1 \Rightarrow S_0 \cup L_0.$$

Supposons donc vérifiées les propriétés de S_1 et L_1, il est alors évident que S_0 est vérifié, montrons que L_0 l'est aussi. La preuve est une succession d'application des règles démontrées dans les propriétés 4.4 et 4.5.

La propriété suivante est également utilisée de multiples fois

$$\frac{p\ leadsto\ q}{p \wedge r\ leadsto\ q}.$$

Ce résultat est évident. Chaque fois que nous l'utiliserons nous placerons r entre parenthèses.

Observons tout d'abord que

$$\frac{L_{ma} \qquad L_{mo2}}{Dma(myc)\ leadsto\ Rma(myc)}\ [4.4:2]$$

Montrons en suite, en utilisant ce résultat que $Dma(myc) \wedge Dc(myc)\ leadsto\ Fc(myc)$.

$$\frac{Dma(myc)(\wedge Dc(myc))\ leadsto\ Rma(myc) \qquad s_1^1}{Dma(myc) \wedge Dc(myc)\ leadsto\ (Rma(myc) \wedge D_c(myc)) \vee F(myc)}\ [4.5:1]$$

$$\frac{Dma(myc) \wedge Dc(myc)\ leadsto\ (Rma(myc) \wedge D_c(myc)) \vee F(myc) \qquad L_c}{Dma(myc) \wedge Dc(myc)\ leadsto\ Fc(myc)}\ [4.4:3]$$

De façon analogue on montre que

$$Dma(myc) \wedge De(myc)\ leadsto\ Fe(myc)$$

$$Dma(myc) \wedge Dme(myc)\ leadsto\ Fme(myc).$$

En utilisant ces résultats, on a successivement

$$\frac{\begin{array}{ll} Dma(myc)(\wedge Dc(myc))\ leadsto\ Rma(myc) & s_1^7 \\ Dma(myc) \wedge Dc(myc)\ leadsto\ Fc(myc) & s_1^4 \\ L_{mo3} & \end{array}}{Dma(myc) \wedge Dc(myc)\ leadsto\ Rmac(myc)}\ [4.5:3]$$

$$\frac{\begin{array}{ll} Dma(myc) \wedge Dc(myc)(\wedge De(myc))\ leadsto\ Rmac(myc) & s_1^8 \\ Dma(myc)(\wedge Dc(myc)) \wedge De(myc)\ leadsto\ Fe(myc) & s_1^5 \\ L_{mo4} & \end{array}}{Dma(myc) \wedge Dc(myc) \wedge De(myc)\ leadsto\ Rmace(myc)}\ [4.5:3]$$

$$\frac{\begin{array}{ll} Dma(myc) \wedge Dc(myc) \wedge De(myc)(\wedge Dme(myc))\ leadsto\ Rmace(myc) & s_1^9 \\ Dma(myc)(\wedge Dc(myc) \wedge De(myc)) \wedge Dme(myc)\ leadsto\ Fme(myc) & s_1^6 \\ L_{mo5} & \end{array}}{Dma(myc) \wedge Dc(myc) \wedge De(myc) \wedge Dme(myc)\ leadsto\ Rmaceme(myc)}\ [4.5:3]$$

$$\frac{L_{mo1} \qquad Dma(myc) \wedge Dc(myc) \wedge De(myc) \wedge Dme(myc)\ leadsto\ Rmaceme(myc)}{Start\ leadsto\ Rmaceme(myc)}\ [4.4:2]$$

$$\frac{Start\ leadsto\ Rmaceme(myc) \qquad L_{mo6}}{Start\ leadsto\ F(myc)}\ [4.4:2]$$

Ce qui suffit.

Annexe C

Code $\mathcal{L}_\Psi$

```
1   ////////////////////////////////////////////////////////
    ////  LpsiRMI                                       ////
    ////  package item         class Item               ////
    ////////////////////////////////////////////////////////
5
    package item;
    import psiterme.*;

    public class Item implements java.io.Serializable{
10  private String champ;

    //CONSTRUCTEUR
    public Item(String s){
        champ=s;
15      }

    //METHODE PUBLIQUE
    public String getNomChamp(){
        return(champ);
20      }
     }
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI                                       ////
   ////  package item      class ItemInteger           ////
   ////////////////////////////////////////////////////////
5
   package item;
   import psiterme.*;

   public class ItemInteger extends Item{
10 private Integer valeur;

   //CONSTRUCTEUR
   public ItemInteger(String strchamp,Integer intvaleur){
       super(strchamp);
15     valeur=intvaleur;
       }

   //METHODE PUBLIQUE
   public Integer getValeur(){
20     return valeur;
       }
   }
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI                                       ////
   ////  package item      class ItemString            ////
   ////////////////////////////////////////////////////////
5
   package item;
   import psiterme.*;

   public class ItemString extends Item{
10 private String valeur;

   //CONSTRUCTEUR
   public ItemString(String strchamp,String strvaleur){
       super(strchamp);
15     valeur=strvaleur;
       }

   //METHODE PUBLIQUE
   public String getValeur(){
20     return valeur;
       }
    }
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI                                       ////
   ////  package item         class ItemPsiTerme       ////
   ////////////////////////////////////////////////////////
5
   package item;
   import psiterme.*;

   public class ItemPsiTerme extends Item{
10 private PsiTerme valeur;

   //CONSTRUCTEUR
   public ItemPsiTerme(String strchamp,PsiTerme psitvaleur){
       super(strchamp);
15     valeur=psitvaleur;
       }

   //METHODE PUBLIQUE
   public PsiTerme getValeur(){
20     return valeur;
       }
   }
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI                                       ////
   ////  package item      class ItemString            ////
   ////////////////////////////////////////////////////////
5
   package item;
   import psiterme.*;

   public class ItemString extends Item{
10 private String valeur;

   //CONSTRUCTEUR
   public ItemString(String strchamp,String strvaleur){
       super(strchamp);
15     valeur=strvaleur;
       }

   //METHODE PUBLIQUE
   public String getValeur(){
20     return valeur;
       }
    }
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI                                       ////
   ////  package psiterme        class PsiTerme        ////
   ////////////////////////////////////////////////////////
5

   package psiterme;

   import item.*;
10 import java.util.*;
   import affectation.*;

   public class PsiTerme extends HashSet implements java.io.Serializable{
   String foncteur;
15
   //CONSTRUCTEUR
   //pre: la String argument a un prefixe dont la structure est correcte

   public PsiTerme (String str){
20     super();
       int separeItem=str.indexOf('(');
       foncteur=str.substring(0,separeItem);
       boolean fini=(str.indexOf(')')==separeItem+1);
       while (!fini){
25     //la boucle complete le psiterme item par item.
       //Invariant: separeItem contient l'index du caractère suivant le
       //dernier item completement traite.
           int egal=str.indexOf('=',separeItem);
           String champItem= new String(str.substring(separeItem+1,egal));
30         int cptr=egal+1;
           char suivant= str.charAt(cptr);
```

```
//on parcourt str à la recherche du prochain séparateur
while((suivant!='(')&&(suivant!=')')&&(suivant!=',')){
    cptr++;
    suivant=str.charAt(cptr);
    }
if (suivant=='('){
    //l'item a comme valeur un psiterme.
    String sousstr=new String(str.substring(egal+1));
    PsiTerme valeurItem=new PsiTerme(sousstr);
    ItemPsiTerme newItem= new ItemPsiTerme(champItem,valeurItem);
    add(newItem);
    int cptrparenthese=1;
    while (cptrparenthese!=0){
    //recherche la ) correspondant a la fin du psiterme-valeur que l'on a traite.
        if ((str.indexOf(")",cptr+1)<str.indexOf("(",cptr+1))||(str.indexOf("(",cptr+1)<0)){
              cptr=str.indexOf(")",cptr+1);
              cptrparenthese=cptrparenthese-1;
              }
        else{
              cptr=str.indexOf("(",cptr+1);
              cptrparenthese=cptrparenthese+1;
              }
        }
    separeItem=cptr+1;
    if (str.charAt(separeItem)==')'){
        fini=true;
        }
    }

if (suivant==','||suivant==')'){
    //l'item a comme valeur un entier ou un string ou ?.
```

```
                if (suivant==')'){
                    fini=true;
                    }
                try{
                    //l'item a comme valeur un entier
                    String Val=str.substring(egal+1,cptr);
                    Integer valeurItem = new Integer(Val);
                    ItemInteger newItem= new ItemInteger(champItem,valeurItem);
                    add(newItem);
                    }
                catch (Exception e){
                    if (str.charAt(egal+1)=='?'){//l'item est une variable
                        String nomVariable=str.substring(egal+2,cptr);
                        ItemVar newItem = new ItemVar(champItem,nomVariable);
                        add(newItem);
                        }
                    else {//l'item a comme valeur un string
                        String valeurItem=str.substring(egal+1,cptr);
                        ItemString newItem = new ItemString(champItem,valeurItem);
                        add(newItem);
                        }
                    }
                separeItem=cptr;
                }
            }
        }


    //METHODES AUXILIAIRES
```

```
protected Item recupererItem(String nomchamp){
    //Precondition. nomchamp contient le nom de champ d'un des items de ce PsiTerme
    boolean trouve=false;
    Item valeur= new Item("");
    Iterator itemIterator= this.iterator();
    while(itemIterator.hasNext()&& !trouve){
        Item itemExamine=(Item)itemIterator.next();
        if (itemExamine.getNomChamp().equals(nomchamp)){
            trouve=true;
            valeur=(itemExamine);
            }
        }
    if (trouve==true) {
        return(valeur);
        }
        else{return(null);}
    }


protected boolean possedeLeChamp(String nomChamp){
    boolean trouve=false;
    Iterator itemIterator= this.iterator();
    while(itemIterator.hasNext()&& !trouve){
        Item itemExamine=(Item)itemIterator.next();
        if (itemExamine.getNomChamp().equals(nomChamp)){
            trouve=true;
            }
        }
    return(trouve);
    }
```

```
protected boolean possedeLItem(Item itemCherche, Affectation affect){//Pre this est clos
    boolean trouve=false;
    Iterator itemIterator= this.iterator();
    while(itemIterator.hasNext()&& !trouve){
        Item itemExamine=(Item)itemIterator.next();
        boolean memeChamp=itemExamine.getNomChamp().equals(itemCherche.getNomChamp());
        boolean memeValeur=(itemCherche.getClass()==new ItemVar("","").getClass());
        if (memeValeur){//Si l'item cherche a une valeur variable
              String var = ((ItemVar)itemCherche).getValeur();
              if (itemExamine.getClass()== new ItemInteger("",new Integer(0)).getClass()){
                  Integer valeur=((ItemInteger)itemExamine).getValeur();
                  affect.ajouterVar(var,valeur);
                  }
              if (itemExamine.getClass()== new ItemString("","").getClass()){
                  String valeur=((ItemString)itemExamine).getValeur();
                  affect.ajouterVar(var,valeur);
                  }
              if (itemExamine.getClass()== new ItemPsiTerme("",new PsiTerme("f()")).getClass()){
                  PsiTerme valeur=((ItemPsiTerme)itemExamine).getValeur();
                  affect.ajouterVar(var,valeur);
                  }
              }
        if (itemCherche.getClass()==itemExamine.getClass()){
              if (itemCherche.getClass()== new ItemString("","").getClass()){
                  memeValeur=(((ItemString)itemExamine).getValeur())
                                              .equals(((ItemString)itemCherche).getValeur());
                  }
              if (itemCherche.getClass()== new ItemInteger("",new Integer(0)).getClass()){
                  memeValeur=(((ItemInteger)itemExamine).getValeur())
                                               .equals(((ItemInteger)itemCherche).getValeur());
```

```
                    }
                if (itemCherche.getClass()== new ItemPsiTerme("",new PsiTerme("f()")).getClass()){
                    memeValeur=(((ItemPsiTerme)itemCherche).getValeur())
                                        .correspond(((ItemPsiTerme)itemExamine).getValeur(),affect);
                    }
                }
            if ((memeChamp) &&(memeValeur)){
                trouve=true;
                }
            }
        return(trouve);
        }


    //METHODES PUBLIQUES

    public boolean estClos(){
        boolean fermeture=true;
        Iterator itemIterator= this.iterator();
        while ((fermeture)&&(itemIterator.hasNext())){
            Item ItemExamine=(Item)itemIterator.next();
            if ( ItemExamine.getClass()==new ItemVar("","").getClass()){
                fermeture=false;
                }
            if ( ItemExamine.getClass()==new ItemPsiTerme("",new PsiTerme ("f()")).getClass()){
                fermeture=((ItemPsiTerme)ItemExamine).getValeur().estClos();
                }
            }
        return fermeture;
        }
```

```
    public boolean correspond(PsiTerme PTC, Affectation affect){
        //!?signification si ptc est pt non clos
190     boolean correspondance=true;
        Affectation affectProv= new Affectation();
        if (!this.foncteur.equals(PTC.foncteur)){
            correspondance=false;
            }
195     Iterator itemIterator= this.iterator();
        while ((correspondance)&&(itemIterator.hasNext())){
            correspondance=PTC.possedeLItem((Item)itemIterator.next(),affectProv);
            }
          if (correspondance){
200           affect.ajouterEnsemble(affectProv);
              }
          return(correspondance);
        }


205
    public Integer recupererEntier(String nomchamp){
        //pre!!!
        Item ItemTrouve=recupererItem(nomchamp);
        return(((ItemInteger)ItemTrouve).getValeur());
210     }

    public String recupererString(String nomchamp){
        //pre!!!
        Item ItemTrouve=recupererItem(nomchamp);
215     return(((ItemString)ItemTrouve).getValeur());
        }
```

```
    public PsiTerme recupererPsiTerme(String nomchamp){
        //pre!!!
220     Item ItemTrouve=recupererItem(nomchamp);
        return(((ItemPsiTerme)ItemTrouve).getValeur());
        }

    public String psiTermeToString(){
225     String ToString = new String("");
        ToString=ToString.concat(this.foncteur);
        ToString=ToString.concat("(");
        Iterator itemIterator= this.iterator();
        while (itemIterator.hasNext()){
230         Item ItemExamine=(Item)itemIterator.next();
            ToString=ToString.concat(ItemExamine.getNomChamp());
            ToString=ToString.concat("=");
            if ( ItemExamine.getClass()==new ItemVar("","").getClass()){
                ToString=ToString.concat("?");
235             }
            if ( ItemExamine.getClass()==new ItemString("","").getClass()){
                ToString=ToString.concat(((ItemString)ItemExamine).getValeur());
                }
            if ( ItemExamine.getClass()==new ItemInteger("",new Integer(0)).getClass()){
240             ToString=ToString.concat(((ItemInteger)ItemExamine).getValeur().toString());
                }

            if ( ItemExamine.getClass()==new ItemPsiTerme("",new PsiTerme ("f()")).getClass()){
                  ToString=ToString.concat(((ItemPsiTerme)ItemExamine).getValeur().psiTermeToString());
245             }
            if (itemIterator.hasNext()){
                ToString=ToString.concat(",");
                }
```

```
            }
250
        ToString=ToString.concat(")");
        return ToString;
        }
    }
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI                                       ////
   ////  package variable         class Variable       ////
   ////////////////////////////////////////////////////////
5
   package variable;
   import psiterme.*;

   public class Variable implements java.io.Serializable{
10 private String nom;

   //CONSTRUCTEUR
   public Variable(String s){
       nom=s;
15     }

   //METHODE PUBLIQUE
   public String getNomVar(){
       return(nom);
20     }
    }
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI                                       ////
   ////  package variable      class VariableInteger   ////
   ////////////////////////////////////////////////////////
5
   package variable;
   import psiterme.*;

   public class VariableInteger extends Variable{
10 private Integer valeur;

   //CONSTRUCTEUR
   public VariableInteger(String strnom,Integer intvaleur){
       super(strnom);
15     valeur=intvaleur;
       }

   //METHODE PUBLIQUE
   public Integer getValeur(){
20     return valeur;
       }
   }
```

```
////////////////////////////////////////////////////////
////  LpsiRMI                                       ////
////  package variable         class VariableString ////
////////////////////////////////////////////////////////

package variable;
import psiterme.*;

public class VariableString extends Variable{
private String valeur;

//CONSTRUCTEUR
public VariableString(String strnom,String strvaleur){
    super(strnom);
    valeur=strvaleur;
    }

//METHODE PUBLIQUE
public String getValeur(){
    return valeur;
    }
 }
```

```
////////////////////////////////////////////////////////
////  LpsiRMI                                       ////
////  package variable     class VariablePsiTerme   ////
////////////////////////////////////////////////////////

package variable;
import psiterme.*;

public class VariablePsiTerme extends Variable{
private PsiTerme valeur;

//CONSTRUCTEUR
public VariablePsiTerme(String strnom,PsiTerme psitvaleur){
    super(strnom);
    valeur=psitvaleur;
    }

//METHODE PUBLIQUE
public PsiTerme getValeur(){
    return valeur;
    }
}
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI                                       ////
   ////  package affectation     class Affectation     ////
   ////////////////////////////////////////////////////////
5
   package affectation;
   import psiterme.*;
   import variable.*;
   import java.util.*;
10
   public class Affectation extends HashSet implements java.io.Serializable{


   //METHODES PUBLIQUES
15
   public void effacerVar(String strNom){
      Iterator varIterator= this.iterator();
      boolean efface = false;
      while((varIterator.hasNext()) &&(!efface)){
20       Variable suivant= (Variable)varIterator.next();
         if(suivant.getNomVar()==strNom){
               remove(suivant);
               efface=true;
               }
25       }
     }

   public void ajouterVar(String strNom,String strValeur){
       effacerVar(strNom);
30     add(new VariableString(strNom,strValeur));
```

```
    }

public void ajouterVar(String strNom,Integer intValeur){
    effacerVar(strNom);
    add(new VariableInteger(strNom,intValeur));
    }

 public void ajouterVar(String strNom,PsiTerme pstValeur){
    effacerVar(strNom);
    add(new VariablePsiTerme(strNom,pstValeur));
    }

public void ajouterEnsemble(Affectation affect){
    Iterator varIterator= affect.iterator();
    while (varIterator.hasNext()){
        Variable suivant =(Variable)varIterator.next();
        effacerVar(suivant.getNomVar());
        add(suivant);
        }
    }

public Integer valeurInteger (String x){
    Iterator varIterator= this.iterator();
    boolean trouve = false;
    Integer valeur =new Integer(0);
    while((varIterator.hasNext()) &&(!trouve)){
        Variable suivant= (Variable)varIterator.next();
        if(suivant.getNomVar().equals(x)){
            valeur=((VariableInteger)suivant).getValeur();
            trouve=true;
            }
```

```
        }
    return valeur;
    }

public String valeurString (String x){
    Iterator varIterator= this.iterator();
    boolean trouve = false;
    String valeur =new String("");
    while((varIterator.hasNext()) &&(!trouve)){
        Variable suivant= (Variable)varIterator.next();
        if(suivant.getNomVar().equals(x)){
            valeur=((VariableString)suivant).getValeur();
            trouve=true;
            }
        }
    return valeur;
    }

public PsiTerme valeurPsiTerme (String x){
    Iterator varIterator= this.iterator();
    boolean trouve = false;
    PsiTerme valeur =new PsiTerme("f()");
    while((varIterator.hasNext()) &&(!trouve)){
        Variable suivant= (Variable)varIterator.next();
        if(suivant.getNomVar().equals(x)){
            valeur=((VariablePsiTerme)suivant).getValeur();
            trouve=true;
            }
        }
    return valeur;
    }
```

```
    public void affiche(){
95      Iterator varIterator= this.iterator();
        while(varIterator.hasNext()){
            Variable suivant= (Variable)varIterator.next();
            System.out.print(suivant.getNomVar());
            if (suivant.getClass()== new VariableInteger("",new Integer(0)).getClass()){
100             String valeur=((VariableInteger)suivant).getValeur().toString();
                System.out.println(valeur);
                }
            if (suivant.getClass()== new VariableString("","").getClass()){
                String valeur=((VariableString)suivant).getValeur();
105             System.out.println(valeur);
                }
            if (suivant.getClass()== new VariablePsiTerme("",new PsiTerme("f()")).getClass()){
                String valeur=((VariablePsiTerme)suivant).getValeur().psiTermeToString();
                System.out.println(valeur);
110             }
            }
        }
    }
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI - Serveur                              ////
   ////  Interface TachePsiTermeSpace                   ////
   ////////////////////////////////////////////////////////
5
   import java.rmi.*;

   import item.*;
   import psiterme.*;
10 import java.util.*;
   import affectation.*;

   public interface TachePsiTermeSpace extends Remote {

15 //METHODES PUBLIQUES

    void tell(PsiTerme PTC) throws RemoteException;

    Affectation ask(PsiTerme PT) throws RemoteException;
20
    void nask(PsiTerme PT) throws RemoteException;

    Affectation get(PsiTerme PT) throws RemoteException;

25 }
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI - Serveur                             ////
   ////  class PsiTermeSpaceImpl                       ////
   ////////////////////////////////////////////////////////
5
   import java.rmi.*;
   import java.rmi.server.*;
   import java.rmi.registry.*;

10 import item.*;
   import psiterme.*;
   import java.util.*;
   import affectation.*;

15
   public class PsiTermeSpaceImpl extends UnicastRemoteObject implements TachePsiTermeSpace {

   private ArrayList ListePsiTerme;

20 //CONSTRUCTEUR

   public PsiTermeSpaceImpl() throws java.rmi.RemoteException{
       super();
       ListePsiTerme=new ArrayList();;
25     }


   //METHODES PUBLIQUES

30 public synchronized void tell(PsiTerme PTC){
```

```
        if (PTC.estClos()){//Si le ptc n'est pas clos rien n'est fait
            ListePsiTerme.add(PTC);//pas besoin de clone car recoit une copie
            notifyAll();
            }
    }

public synchronized Affectation ask(PsiTerme PT){
    boolean trouve=false;
    Affectation correspondance= new Affectation();

    while (!trouve){
        int compteur=0;
            while((compteur<(ListePsiTerme.size()))&&(!trouve)){
            compteur++;
            correspondance.clear();
            trouve=PT.correspond((PsiTerme)ListePsiTerme.get(compteur-1),correspondance);
            }

        if (!trouve){
            try{
                wait();
                }
                catch (Exception e) {}
            }
        }
    notifyAll();
    return correspondance;
    }

public synchronized void nask(PsiTerme PT){
    boolean trouve=true;
```

```
        Affectation correspondance= new Affectation();
        while (trouve){
            int compteur=0;
            trouve=false;
                while((compteur<(ListePsiTerme.size()))&&(!trouve)){
                compteur++;
                correspondance.clear();
                trouve=PT.correspond((PsiTerme)ListePsiTerme.get(compteur-1), correspondance);
                }

            if (trouve){
                try{
                    wait();
                    }
                    catch (Exception e) {}
                }
            }
        notifyAll();
        }


    public synchronized Affectation get(PsiTerme PT){
        boolean trouve=false;
        Affectation correspondance=new Affectation();
        int compteur;
        while (!trouve){
                compteur=0;
                while((compteur<(ListePsiTerme.size()))&&(!trouve)){
                compteur++;
                correspondance.clear();
                trouve=PT.correspond((PsiTerme)ListePsiTerme.get(compteur-1),correspondance);
```

```
                }

            if (trouve){
                ListePsiTerme.remove(compteur-1);
                }
            else{
                try{
                    wait();
                    }
                    catch (Exception e) {}
                }
            }
        notifyAll();
        return correspondance;
        }

    public static void main(String args[]){
        Registry repertoire;
        try{  PsiTermeSpaceImpl espace =new PsiTermeSpaceImpl();
              repertoire=LocateRegistry.getRegistry("localhost");
              repertoire.bind("MonTableau",espace);
              }
        catch (RemoteException e){
              e.printStackTrace();
              }
        catch (AlreadyBoundException e){
              e.printStackTrace();
              }
        System.out.println("<--serveur opérationnel-->");
        }
    }
```

Annexe D

Code de l'application PingPong

```
1   ////////////////////////////////////////////////////////
    ////  LpsiRMI - Client                                ////
    ////  class PingPong                                  ////
    ////////////////////////////////////////////////////////
5
    import java.rmi.*;
    import java.rmi.registry.*;

    import item.*;
10  import psiterme.*;
    import affectation.*;


    public class PingPong{
15      public static void main(String[] args) {
            Registry repertoire;
            TachePsiTermeSpace un_serveur;

            Affectation AffectPong;
20
            try{
            repertoire = (Registry)LocateRegistry.getRegistry("localhost");
            un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
            un_serveur.tell(new PsiTerme("PingPong(etat=Ping,numero=12)"));
25          }
            catch (RemoteException e){e.printStackTrace();}
            catch (NotBoundException e){e.printStackTrace();}
            }
        }
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI - Client                                ////
   ////  class Ping                                      ////
   ////////////////////////////////////////////////////////
5
   import java.rmi.*;
   import java.rmi.registry.*;

   import item.*;
10 import psiterme.*;
   import affectation.*;


   public class Ping {
15     static public void main(String args[]){
           Registry repertoire;
           TachePsiTermeSpace un_serveur;

           Affectation AffectPing;
20
           try{  repertoire = (Registry)LocateRegistry.getRegistry("localhost");
                 un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
                 while(true){
                       AffectPing=un_serveur.get(new PsiTerme("PingPong(etat=Pong,numero=?x)"));
25                     int cptr= AffectPing.valeurInteger("x").intValue();
                       String message=new String("PingPong(etat=Ping,numero=")
                                            .concat(new Integer(cptr+1).toString()).concat(")");
                       un_serveur.tell(new PsiTerme(message));
                       System.out.println(message);
30                     AffectPing.affiche();
                       try{
```

```
                        Thread.sleep(1000);
                        }
                    catch (Exception e){};
                }
            }
        catch (RemoteException e){e.printStackTrace();}
        catch (NotBoundException e){e.printStackTrace();}
        }
    }
```

```
////////////////////////////////////////////////////////
////  LpsiRMI - Client                                ////
////  class Pong                                      ////
////////////////////////////////////////////////////////

import java.rmi.*;
import java.rmi.registry.*;

import item.*;
import psiterme.*;
import affectation.*;


public class Pong {
    static public void main(String args[]){
        Registry repertoire;
        TachePsiTermeSpace un_serveur;

        Affectation AffectPong;

        try{  repertoire = (Registry)LocateRegistry.getRegistry("localhost");
```

```
                un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
                while(true){
                      AffectPong=un_serveur.get(new PsiTerme("PingPong(etat=Ping,numero=?x)"));
                      int cptr= AffectPong.valeurInteger("x").intValue();
                      String message=new String("PingPong(etat=Pong,numero=")
                                          .concat(new Integer(cptr+1).toString()).concat(")");
                      un_serveur.tell(new PsiTerme(message));
                      System.out.println(message);
                      AffectPong.affiche();
                      try{
                            Thread.sleep(1000);
                            }
                        catch (Exception e){};
                      }
              }
          catch (RemoteException e){e.printStackTrace();}
          catch (NotBoundException e){e.printStackTrace();}
          }
    }
```

Annexe E

Code de l'application Chantier

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI - Client                                ////
   ////  class MaitreOeuvre                              ////
   ////////////////////////////////////////////////////////
5
   import java.rmi.*;
   import java.rmi.registry.*;

   import item.*;
10 import psiterme.*;
   import affectation.*;

   public class MaitreOeuvre {
   //champs
15         Registry repertoire;
           TachePsiTermeSpace un_serveur;

           Affectation AffectMaitreOeuvre;
           String chantier;
20
   //constructeur
   public MaitreOeuvre(String hote,String parametre[]){

   String chantierold;
25 String chantiernew;
           try{
               //verifier start
               repertoire = (Registry)LocateRegistry.getRegistry(hote);
               un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
30             chantiernew= new String("chantier(Cnom="+parametre[4]+")");
               un_serveur.nask(new PsiTerme(chantiernew));
```

```
System.out.println(chantiernew);
System.out.println();

//ouvrir le chantier

chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=demande,Ccouvreur=demande,
   Celectricien=demande,Cmenuisier=demande,Clargeur="+parametre[0]+",Clongueur="+parametre[1]+
   ",Cnbreporte="+parametre[2]+",Cnbrefenetre="+parametre[3]+",etat=encours)");
un_serveur.tell(new PsiTerme(chantiernew));

System.out.println(chantiernew);
System.out.println();

//attendre le maçon
String attendu=new String("fini(Cnom="+parametre[4]+",Couvrier=macon)");
un_serveur.get(new PsiTerme(attendu));
//recevoir macon
chantierold = chantiernew;
chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=recu,Ccouvreur=demande,
   Celectricien=demande,Cmenuisier=demande,Clargeur="+parametre[0]+",Clongueur="+parametre[1]+
   ",Cnbreporte="+parametre[2]+",Cnbrefenetre="+parametre[3]+",etat=encours)");
un_serveur.tell(new PsiTerme(chantiernew));
un_serveur.get(new PsiTerme(chantierold));


System.out.println(chantiernew);
System.out.println();

//attendre le couvreur
attendu=new String("fini(Cnom="+parametre[4]+",Couvrier=couvreur)");
un_serveur.get(new PsiTerme(attendu));
```

```
//recevoir macon
chantierold = chantiernew;
chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=recu,Ccouvreur=recu,
   Celectricien=demande,Cmenuisier=demande,Clargeur="+parametre[0]+",Clongueur="+parametre[1]+
   ",Cnbreporte="+parametre[2]+",Cnbrefenetre="+parametre[3]+",etat=encours)");
un_serveur.tell(new PsiTerme(chantiernew));
un_serveur.get(new PsiTerme(chantierold));

System.out.println(chantiernew);
System.out.println();

//attendre l'électricien
attendu=new String("fini(Cnom="+parametre[4]+",Couvrier=electricien)");
un_serveur.get(new PsiTerme(attendu));
//recevoir macon
chantierold = chantiernew;
chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=recu,Ccouvreur=recu,
   Celectricien=recu,Cmenuisier=demande,Clargeur="+parametre[0]+",Clongueur="+parametre[1]+
   ",Cnbreporte="+parametre[2]+",Cnbrefenetre="+parametre[3]+",etat=encours)");
un_serveur.tell(new PsiTerme(chantiernew));
un_serveur.get(new PsiTerme(chantierold));

System.out.println(chantiernew);
System.out.println();

//attendre le menuisier
attendu=new String("fini(Cnom="+parametre[4]+",Couvrier=menuisier)");
un_serveur.get(new PsiTerme(attendu));
//recevoir menuisier
chantierold = chantiernew;
chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=recu,Ccouvreur=recu,
```

```
                Celectricien=recu,Cmenuisier=recu,Clargeur="+parametre[0]+",Clongueur="+parametre[1]+
                ",Cnbreporte="+parametre[2]+",Cnbrefenetre="+parametre[3]+",etat=encours)");
            un_serveur.tell(new PsiTerme(chantiernew));
            un_serveur.get(new PsiTerme(chantierold));

            System.out.println(chantiernew);
            System.out.println();

            //clore chantier
            chantierold = chantiernew;
            chantiernew= new String("chantier(Cnom="+parametre[4]+",Cmacon=recu,Ccouvreur=recu,
                Celectricien=recu,Cmenuisier=recu,Clargeur="+parametre[0]+",Clongueur="+parametre[1]+
                ",Cnbreporte="+parametre[2]+",Cnbrefenetre="+parametre[3]+",etat=fini)");
            un_serveur.tell(new PsiTerme(chantiernew));
            un_serveur.get(new PsiTerme(chantierold));

            System.out.println(chantiernew);
            System.out.println();
        }//ftry

        catch (RemoteException e){e.printStackTrace();}
        catch (NotBoundException e){e.printStackTrace();}
        }    //fin constructeur


public static void main(String args[]){

      String hote=new String("localhost");
      String[] parametre= new String[5];
      parametre[0]="0";//longueur
      parametre[1]="0";//largeur
```

```
125         parametre[2]="0";//nbreporte
            parametre[3]="0";//nbrefenêtre
            parametre[4]="myc";//nomchantier

        // un petit parseur de paramètres...
130
            for (int i=0 ; i < args.length;i++ ) {
            char argType = args[i].charAt(1);

            if ((argType == 'U')||(argType == 'u')) {
135             if (args[i].length() < 4) {
                      System.out.println("ERROR : URL ["+args[i]+"] invalide ");
                      System.exit(0);
                      }//fi
                hote = args[i].substring(3);
140             System.out.println(" INFO : naming service URL is [ "+hote+" ].");
                 }//fi

            if (argType == 'l') {
                if (args[i].length() < 4) {
145                   System.out.println("ERROR : largeur ["+args[i]+"] invalide ");
                      System.exit(0);
                      }//fi
                parametre[0] = args[i].substring(3);
                System.out.println(" INFO : la largeur est  "+parametre[0]+".");
150              }//fi


            if (argType == 'L') {
                if (args[i].length() < 4) {
155                   System.out.println("ERROR : longueur ["+args[i]+"] invalide ");
```

```
                System.exit(0);
                }//fi
            parametre[1] = args[i].substring(3);
            System.out.println(" INFO : la longueur est  "+parametre[1]+".");
             }//fi


        if ((argType == 'p')||(argType == 'P')) {
            if (args[i].length() < 4) {
                System.out.println("ERROR : nombre portes ["+args[i]+"] invalide ");
                System.exit(0);
                }//fi
            parametre[2] = args[i].substring(3);
            System.out.println(" INFO : le nombre de portes est  "+parametre[2]+".");
             }//fi


        if ((argType == 'f')||(argType == 'F')) {
            if (args[i].length() < 4) {
                System.out.println("ERROR : nombre fenetres  ["+args[i]+"] invalide ");
                System.exit(0);
                }//fi
            parametre[3] = args[i].substring(3);
            System.out.println(" INFO : le nombre de fenetres est  "+parametre[3]+".");
             }//fi

        if ((argType == 'n')||(argType == 'N')) {
            if (args[i].length() < 4) {
                System.out.println("ERROR : nom  ["+args[i]+"] invalide ");
                System.exit(0);
                }//fi
```

```
            parametre[4] = args[i].substring(3);
            System.out.println(" INFO : le nom du chantier est  "+parametre[4]+".");
             }//fi
      }//fin parser

    new MaitreOeuvre(hote,parametre);

    }//fin main

}
```

```
////////////////////////////////////////////////////////
////  LpsiRMI - Client                                ////
////  class Macon                                     ////
////////////////////////////////////////////////////////

import java.rmi.*;
import java.rmi.registry.*;

import item.*;
import psiterme.*;
import affectation.*;


public class Macon {
//champs
        Registry repertoire;
        TachePsiTermeSpace un_serveur;

        Affectation AffectMacon;
```

```
20          String chantier;


    //constructeur
        public Macon(String hote){
25        try{  repertoire = (Registry)LocateRegistry.getRegistry(hote);
                un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
                      chantier= new String("chantier(Cnom=?VMAnom,Cmacon=demande,Clargeur=?VMAlargeur,
                        Clongueur=?VMAlongueur,Cnbreporte=?VMAnbreporte,Cnbrefenetre=?VMAnbrefenetre)");
                      AffectMacon=un_serveur.ask(new PsiTerme(chantier));
30
                      //realiser le travail
                      System.out.println("Le macon travaille sur ".concat(AffectMacon.valeurString("VMAnom")));
                      int duree=(AffectMacon.valeurInteger("VMAlargeur").intValue()*
                                                AffectMacon.valeurInteger("VMAlongueur").intValue())/2;
35                    for (int compteur=1;compteur<duree;compteur++ )
                      {System.out.println("Il reste au maçon "+String.valueOf(duree-compteur)+
                                                " jours de travail sur "+(AffectMacon.valeurString("VMAnom")));
                      try{
                            Thread.sleep(100);
40                          }
                        catch (Exception e){};
                      }


45                    String message=new String("fini(Cnom=".concat(AffectMacon.valeurString("VMAnom"))
                                                                        .concat(",Couvrier=macon)"));
                      un_serveur.tell(new PsiTerme(message));
                      System.out.println(message);

50            }
```

```
            catch (RemoteException e){e.printStackTrace();}
            catch (NotBoundException e){e.printStackTrace();}
        }//fin constructeur

55

    //main
        static public void main(String args[]){
            String hote=new String("localhost");

60      // un petit parseur de paramètres...

                for (int i=0 ; i < args.length;i++ ) {
              char argType = args[i].charAt(1);

65            if ((argType == 'U')||(argType == 'u')) {
                  if (args[i].length() < 4) {
                            System.out.println("ERROR : URL ["+args[i]+"] too short ");
                            System.exit(0);
                            }//fi
70                hote = args[i].substring(3);
                  System.out.println(" INFO : naming service URL is [ "+hote+" ].");
                   }//fi
              }//fin parser

75          new Macon(hote);

            }
      }

80
```

```
1  ////////////////////////////////////////////////////////
   ////  LpsiRMI - Client                                ////
   ////  class Couvreur                                  ////
   ////////////////////////////////////////////////////////
5
   import java.rmi.*;
   import java.rmi.registry.*;

   import item.*;
10 import psiterme.*;
   import affectation.*;


   public class Couvreur {
15 //champs
          Registry repertoire;
          TachePsiTermeSpace un_serveur;

          Affectation AffectCouvreur;
20        String chantier;


   //constructeur
       public Couvreur(String hote){
25
          try{  repertoire = (Registry)LocateRegistry.getRegistry(hote);
                un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
                      chantier= new String("chantier(Cnom=?VCnom,Cmacon=recu,Ccouvreur=demande,
                         Clargeur=?VClargeur,Clongueur=?VClongueur,Cnbreporte=?VCnbreporte,
30                       Cnbrefenetre=?VCnbrefenetre)");
                      AffectCouvreur=un_serveur.ask(new PsiTerme(chantier));
```

```
                    //realiser le travail
                    System.out.println("Le couvreur travaille sur "
                                                .concat(AffectCouvreur.valeurString("VCnom")));
                    int duree=AffectCouvreur.valeurInteger("VClargeur").intValue()*
                                        AffectCouvreur.valeurInteger("VClongueur").intValue();
                    for (int compteur=1;compteur<duree;compteur++ )
                    {System.out.println("Il reste au couvreur "+String.valueOf(duree-compteur)+
                              " jours de travail sur "+(AffectCouvreur.valeurString("VCnom")));
                    try{
                          Thread.sleep(100);
                          }
                      catch (Exception e){};
                    }

                    String message=new String("fini(Cnom=".concat(AffectCouvreur.valeurString("VCnom"))
                                                                        .concat(",Couvrier=couvreur)"));
                    un_serveur.tell(new PsiTerme(message));
                    System.out.println(message);

                }
            catch (RemoteException e){e.printStackTrace();}
            catch (NotBoundException e){e.printStackTrace();}

        }//fin constructeur


    //main
        static public void main(String args[]){
            String hote=new String("localhost");
```

```
        // un petit parseur de paramètres...

65              for (int i=0 ; i < args.length;i++ ) {
            char argType = args[i].charAt(1);

            if ((argType == 'U')||(argType == 'u')) {
                if (args[i].length() < 4) {
                        System.out.println("ERROR : URL ["+args[i]+"] too short ");
70                      System.exit(0);
                        }//fi
                hote = args[i].substring(3);
                System.out.println(" INFO : naming service URL is [ "+hote+" ].");
75               }//fi
            }//fin parser

           new Couvreur(hote);

80         }
        }
```

```
 1  ////////////////////////////////////////////////////
    ////  LpsiRMI - Client                          ////
    ////  class Electricien                         ////
    ////////////////////////////////////////////////////
 5
    import java.rmi.*;
    import java.rmi.registry.*;

    import item.*;
10  import psiterme.*;
```

```
import affectation.*;


public class Electricien {
//champs
        Registry repertoire;
        TachePsiTermeSpace un_serveur;

        Affectation AffectElectricien;
        String chantier;


//constructeur
    public Electricien(String hote){

        try{  repertoire = (Registry)LocateRegistry.getRegistry(hote);
              un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
                    chantier= new String("chantier(Cnom=?VEnom,Cmacon=recu,Celectricien=demande,
                       Clargeur=?VElargeur,Clongueur=?VElongueur,Cnbreporte=?VEnbreporte,
                       Cnbrefenetre=?VEnbrefenetre)");
                    AffectElectricien=un_serveur.ask(new PsiTerme(chantier));

                    //realiser le travail
                    System.out.println("Le electricien travaille sur "
                                              .concat(AffectElectricien.valeurString("VEnom")));
                    int duree=AffectElectricien.valeurInteger("VElargeur").intValue()*
                                    AffectElectricien.valeurInteger("VElongueur").intValue()/5;
                    for (int compteur=1;compteur<duree;compteur++ )
                    {System.out.println("Il reste au electricien "+String.valueOf(duree-compteur-1)+
                                " jours de travail sur "+(AffectElectricien.valeurString("VEnom")));
                    try{
```

```
                        Thread.sleep(100);
                        }
                    catch (Exception e){};
                }

                String message=new String("fini(Cnom=".concat(AffectElectricien.valeurString("VEnom"))
                                                                .concat(",Couvrier=electricien)"));
                un_serveur.tell(new PsiTerme(message));
                System.out.println(message);

            }
        catch (RemoteException e){e.printStackTrace();}
        catch (NotBoundException e){e.printStackTrace();}

    }//fin constructeur


//main
    static public void main(String args[]){
        String hote=new String("localhost");

    // un petit parseur de paramètres...

          for (int i=0 ; i < args.length;i++ ) {
        char argType = args[i].charAt(1);

        if ((argType == 'U')||(argType == 'u')) {
            if (args[i].length() < 4) {
                    System.out.println("ERROR : URL ["+args[i]+"] too short ");
                    System.exit(0);
                    }//fi
```

```
                hote = args[i].substring(3);
                System.out.println(" INFO : naming service URL is [ "+hote+" ].");
                 }//fi
            }//fin parser

        new Electricien(hote);

        }
    }
```

```
////////////////////////////////////////////////////////
////  LpsiRMI - Client                              ////
////  class Menuisier                               ////
////////////////////////////////////////////////////////

import java.rmi.*;
import java.rmi.registry.*;

import item.*;
import psiterme.*;
import affectation.*;


public class Menuisier {
//champs
        Registry repertoire;
        TachePsiTermeSpace un_serveur;

        Affectation AffectMenuisier;
        String chantier;
```

```
//constructeur
    public Menuisier(String hote){

        try{  repertoire = (Registry)LocateRegistry.getRegistry(hote);
              un_serveur=(TachePsiTermeSpace)repertoire.lookup("MonTableau");
                    chantier= new String("chantier(Cnom=?VMEnom,Cmacon=recu,Cmenuisier=demande,
                       Clargeur=?VMElargeur,Clongueur=?VMElongueur,Cnbreporte=?VMEnbreporte,
                       Cnbrefenetre=?VMEnbrefenetre)");
                    AffectMenuisier=un_serveur.ask(new PsiTerme(chantier));

                    //realiser le travail
                    System.out.println("Le menuisier travaille sur "
                                                .concat(AffectMenuisier.valeurString("VMEnom")));
                    int duree=AffectMenuisier.valeurInteger("VMEnbreporte").intValue()+
                                   AffectMenuisier.valeurInteger("VMEnbrefenetre").intValue()/2;
                    for (int compteur=1;compteur<duree;compteur++ )
                    {System.out.println("Il reste au menuisier "+String.valueOf(duree-compteur-1)+
                                " jours de travail sur "+(AffectMenuisier.valeurString("VMEnom")));
                    try{
                          Thread.sleep(100);
                          }
                      catch (Exception e){};
                    }

                    String message=new String("fini(Cnom=".concat(AffectMenuisier.valeurString("VMEnom"))
                                                                        .concat(",Couvrier=menuisier)"));
                    un_serveur.tell(new PsiTerme(message));
                    System.out.println(message);
```

```
            }
        catch (RemoteException e){e.printStackTrace();}
        catch (NotBoundException e){e.printStackTrace();}

    }//fin constructeur


//main
    static public void main(String args[]){
        String hote=new String("localhost");

    // un petit parseur de paramètres...

          for (int i=0 ; i < args.length;i++ ) {
        char argType = args[i].charAt(1);

        if ((argType == 'U')||(argType == 'u')) {
            if (args[i].length() < 4) {
                      System.out.println("ERROR : URL ["+args[i]+"] too short ");
                      System.exit(0);
                      }//fi
            hote = args[i].substring(3);
            System.out.println(" INFO : naming service URL is [ "+hote+" ].");
             }//fi
        }//fin parser

        new Menuisier(hote);

        }
    }
```

Bibliographie

[1] A.-K. and A. Podelski. Towards a meaning of life. *Journal of Logic Programming*, 16(3-4) :195–234, 1993.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1) :73–132, 1993.

[3] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21 :181–185, 1985.

[4] J.M. Andreoli and R. Pareschi. Linear objects : logical processes with builtin inheritance. *New Generation Computing*, 9(3-4) :445–473, 1991.

[5] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency : practice and experience*, 5(1) :23–70, 1993.

[6] J. Banatre and D. LeMetayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1) :98–111, 1991.

[7] A. Brogi and P. Ciancarini. The Concurrent Language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1) :99–123, 1991.

[8] A. Brogi and J.-M. Jacquet. Modeling Coordination via Asynchronous Communication. In D. Garlan and D. Le Métayer, editors, *Proceedings of the Second International Conference on Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*, pages 238–255, Berlin, Germany, 1997.

[9] A. Brogi and J.-M. Jacquet. On the expressiveness of Coordination Models. In P. Ciancarini and A. Wolf, editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594, pages 134–149, Amsterdam, Netherland, 1999. Springer-Verlag, Berlin.

[10] A. Brogi and J.-M. Jaquet. Modeling Coordination via Asynchronous Communication. In *Coordination Models and Languages*, pages 238–255, 1997.

[11] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4) :444–458, 1989.

[12] K.M Chandy and J. Misra. *Parallel Program Design : a Foundation*. Addison-Wesley, 1988.

[13] P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, (3) :251–284, 1994.

[14] P. Ciancarini and D. Rossi. Jada : coordination and communication for java agents. In *Proc. Second Int. Workshop on mobile object systems*, volume 1222 of *LNCS*, pages 213–228. Springer-Verlag, 1996.

[15] P. Ciancarini and D. Rossi. Jada : A Coordination Toolkit for Java. Technical Report C, Department of Computer Science, University of Bologna, Italy, 1997.

[16] P. Collette. Application of the composition principle to unity-like specifications. In M. C. Gaudel and J. P. Jouannaud, editors, *TAPSOFT'93 : Theory and Practice of Software Development / 4th International Joint Conference CAAP/FASE*, pages 230–242. Springer, Berlin, Heidelberg, 1993.

[17] V. Englebert. Conception des systèmes d'information coopératifs. Cours Lihd-2, 2001.

[18] D. Gelertner and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 37(2) :97 – 107, 1992.

[19] J.-M. Jacquet. About compositionality and full abstractness. 1997.

[20] J.-M. Jacquet and K. De Bosschere. On the Semantics of μlog. *Future Generation Computer Systems*, 10 :93–135, 1994.

[21] J.-M. Jacquet and K. De Bosschere. On Composing Concurrent Logic Processes. In *International Conference on Logic Programming*, pages 531–545, 1995.

[22] D. Gelernter N. Carriero and L. Zuck. Bauhaus Linda. In In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object based models and languages for concurrent systems*, volume 924 of *Lecture Notes in Computer Science*, pages 66–76. Springer-Verlag, 1994.

[23] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim : a kernel language for agents interaction and mobility. *IEEE Trans. on Software Engineering*, 1998.

[24] K. Noben. Exemple complet d'utilisation de RMI. http ://www.info.fundp.ac.be/~kno/, 2001.

[25] A. Rowstron and A. Wood. Bonita : A set of tuple space primitives for distributed coordination. In *Proc. 30th Hawaii Int. Conf. on System Sciences*, pages 379–388. IEEE Press, 1997.

[26] V.A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, 1993.

[27] L. Semini. *Refinement in Tuple Space Languages.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1996.

[28] R. Tolksdorf. Coordinating services in open distributed systems with laura. In P. Ciancarini and C. Hankin, editors, *Proc. Coordination'96 : Proceedings of The First International Conference on Coordination Models and Languages*, volume 1061 of *LNCS*, pages 386–402. Springer-Verlag, 1996.

[29] P. Wegner. Why Interaction Is More Powerful Than Algorithms. *Communications of the ACM*, 1997.

Table des figures

1.1 Grammaire du langage $\mathcal{L}_L$(tell, ask, get, nask) 4
1.2 Règles de transition associées à la grammaire $\mathcal{L}_L$(tell, ask, get, nask) 4
1.3 Règles de transition associées à la grammaire $\mathcal{L}_\Psi$ 8

4.1 Règles de transition étiquetées associées à la grammaire Sag 43

5.1 Propositions 62
5.2 Propriétés de *safety* présentes dans S_6 67
5.3 Quelques ψ-termes particuliers. 68
5.4 Expression formelle des propositions 70

Index

$A^{<\omega}$, 17
$Ext(S)$, 21
$Ext(h)$, 21, 23
Is, 57
$S[p]$, 21
S^+, 21
S^-, 21
S^a, 21
$Saffect$, 6
Sag, 41
$Setat$, 6
$Shhist$, 20
$Shist$, 18
$Sinstr$, 6
$Spsiterme$, 6
$Spsitermeclos$, 6
$Ssituation$, 17
$Svar$, 6
$Var(S)$, 21
$Var(h)$, 20, 23
Var_a, 7
α, 41
$\bar{h}$, 20
β, 30
δ^+, 17
δ^-, 17
$SEhist$, 42
$\Box$, 28
$\mathcal{P}_{ns}(Sag)$, 41
ψ-terme, 5
    clos, 5
    correspondance, 5
$\Sigma^V_{(\theta,\sigma)\to(\theta,\tau)}$, 32
$\tilde{\alpha}$, 41
$diff(S)$, 20
$diff(h)$, 20
$h[n]$, 42
$h_{k,a}$, 42
$h_{k,i}$, 42
$init(h)$, 20
$lenght(h)$, 42
$s_\beta(L)$, 59
sat_α, 57
ss, 42

agent
    $Ag^V_{\sigma\to\tau}$, 32
    $B^W_{(\theta,\sigma)\to(\kappa,\sigma)}$, 31

cohérent, 34
contexte, 28, 29

distance, 42

instuction executable, 6

langage
    $\mathcal{L}_\Psi$, 5

opérateur
    $\widetilde{+}$, 23
    $\diamond_i$, 18
    $\widetilde{\diamond}_i$, 18
    $\widetilde{;}$, 22
    $\widetilde{||}$, 22
    $\widetilde{||}_h$, 22
    initialy, 49
    invariant, 49
    leadsto, 49
    stable, 49
    $unless_\alpha$, 49

propriété, 45
propriété
    liveness, 45
    safety , 45

raffinement, 60
relation de congruence, 19

situation, 17
système
    Σ, 57
    hétérogène, 59
    raffinement, 60
sémantique
    $\mathcal{D}_h$, 19, 22, 23
    $\mathcal{O}_g$, 44
    $\mathcal{O}_g^*$, 44
    $\mathcal{O}_h$, 18
    complète, 29
    complètement abstraite, 29
    compositionnelle, 18
    correcte, 29
    extensible, 27