Modèle et langage de coordination pour les systèmes multi-agents ouverts. Application au problème du Transport À la Demande

Mahdi Zargayouna

▶ **To cite this version:**

Mahdi Zargayouna. Modèle et langage de coordination pour les systèmes multi-agents ouverts. Application au problème du Transport À la Demande. Intelligence artificielle [cs.AI]. Université Paris Dauphine PSL, 2007. Français. NNT : . tel-02878991

HAL Id: tel-02878991
https://hal.science/tel-02878991v1

Submitted on 23 Jun 2020

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

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

UNIVERSITÉ PARIS DAUPHINE
EDDIMO
LAMSADE (UMR 7024)

N° attribué par la bibliothèque

Modèle et langage de coordination pour les Systèmes Multi-Agents Ouverts.

Application au problème du Transport À la Demande

THESE

Présentée publiquement le 7 décembre 2007
Pour l'obtention du titre de
DOCTEUR EN INFORMATIQUE
(Arrêté du 7 Août 2006)

Mahdi Zargayouna

JURY

Directeurs de thèse :	**Monsieur Serge Haddad** Professeur à l'Université Paris-Dauphine
	Monsieur Gérard Scémama Directeur de recherche à l'Inrets
Responsable scientifique :	**Monsieur Flavien Balbo** Maître de conférences à l'Université Paris-Dauphine
Rapporteurs :	**Madame Amal El Fallah-Seghrouchni** Professeur à l'Université Paris-VI Pierre et Marie Curie
	Monsieur Philippe Mathieu Professeur à l'Université Lille-I
Examinateurs :	**Madame Béatrice Bérard** Professeur à l'Université Paris-Dauphine
	Monsieur Michel Occello Professeur à l'Université Pierre Mendes France - Valence

Table des matières

Liste des tableaux

Table des figures

Introduction générale

Motivations

L'expansion constante des réseaux d'ordinateurs et le développement continu d'applications distribuées oriente la conception des systèmes logiciels modernes à se concentrer de plus en plus sur la réutilisation et l'intégration de composants logiciels. Ainsi, nous avons assisté à un passage des applications autonomes à des réseaux distribués en interaction, qui appellent à des méthodologies et des outils bien définis, visant à intégrer des composants logiciels hétérogènes. Dans ce contexte, Gelernter and Carriero [Gelernter and Carriero, 1992] appellent à une séparation claire entre les aspects interactionnels et les aspects computationnels des composants logiciels. Selon eux, un langage de programmation est constitué d'un langage de calcul et d'un langage de coordination. Les auteurs proposèrent un modèle de coordination, Linda [Carriero *et al.*, 1986], présenté comme un ensemble de primitives sur un espace de données, pouvant être théoriquement ajoutés à n'importe quel langage de programmation. Outre la création de processus, les primitives permettent d'ajouter, lire et retrancher des données d'un espace partagé. Récemment, les modèles et langages de coordination ont connu une explosion de propositions dans des domaines multiples et variés. Par exemple, Klaim [De Nicola *et al.*, 1998] propose un langage de coordination, extension de Linda pour les agents mobiles ; Lime [Picco *et al.*, 1999], Limone [Fok *et al.*, 2004] et ObjectPlaces [Schelfthout and Holvoet, 2004] sont des modèles pour les réseaux ad hoc mobiles ; PagesSpaces [Ciancarini *et al.*, 1998], XMLSpaces [Tolksdorf and Glaubitz, 2001] et Semantic Web Spaces [Nixon *et al.*, 2007] sont des propositions de développement de modèles à la Linda spécifiquement pour Internet et les applications Web etc.

D'autre part, depuis les années 1990, les systèmes multi-agents (SMA) sont l'un des domaines les plus actifs en recherche informatique. Dans un SMA, le contrôle est distribué sur un réseau d'agents autonomes, et le fonctionnement du système émerge des comportements locaux des agents le composant. L'un des sujets qui a reçu le plus d'attention en SMA est celui de la coordination. La coordination a pour objectif de garantir un comportement cohérent d'un réseau d'agents.

C'est dans ce contexte que se situe le travail de cette thèse. Notre premier objectif est de bâtir un modèle et un langage de coordination à la Linda capable de coordonner des agents dans un SMA ouvert. Notre application support est issue du domaine du transport, et modélise un mode de transport novateur se situant entre les services de bus collectifs et les taxis individuels : le transport à la demande (TAD). Il s'agit d'un problème où les voyageurs demandent un service temps-réel de transport personnalisé, en acceptant de partager le véhicule avec d'autres personnes. Bien que facile à énoncer, ce problème est de grande complexité et difficile à résoudre.

Systèmes Multi-Agents

Les avantages de la décentralisation d'un système qu'il soit biologique, politique ou informatique sont indéniables. En effet, la minimisation du contrôle central allège le fonctionnement d'un système et colle au plus près aux intérêts individuels des entités le composant. Les SMA transposent ces idées sur des systèmes informatiques composés d'entités intelligentes appelées *agents*. Grâce au développement de l'informatique distribuée, les SMA sont un paradigme offrant un cadre abstrait permettant une approche conceptuelle simple et distribuée dans l'appréhension de problèmes complexes. Les travaux en SMA couvrent plusieurs disciplines, pas seulement informatiques, puisqu'ils s'inspirent notamment des idées et concepts issus de l'économie, de la psychologie et de la sociologie.

Un SMA est un système composé d'*agents* autonomes qui interagissent afin de résoudre des problèmes. Un agent est un système informatique évoluant dans un environnement et qui exécute des actions autonomes dans le but de satisfaire ses propres objectifs. Ainsi, un agent est capable d'agir sans intervention directe d'un humain ou d'un autre agent, et d'avoir un contrôle sur ses propres actions et son état interne. De plus, un agent est situé dans un environnement et peut agir dessus et en recevoir des signaux qu'il interprète. Enfin, un agent interagit avec les autres agents ou avec un humain afin de satisfaire ses objectifs ou aider les autres à satisfaire les leurs. La catégorie de SMA qui nous intéresse particulièrement dans le cadre de cette thèse concerne les SMA ouverts. Un SMA ouvert est un SMA dans lequel le nombre d'agent évolue à travers le temps, des agents pouvant rejoindre et quitter le système librement. Le principal problème posé dans cette catégorie de systèmes est le « problème de connexion » [Davis and Smith, 1983], ou comment trouver les bons interlocuteurs d'un nouvel agent rejoignant le SMA. Il faut également garantir une exécution cohérente du SMA lorsqu'un agent quitte le système. Les questions que se posent les chercheurs en SMA sont si multiples et variées qu'il serait vain d'essayer de les synthétiser ici. Les aspects qui nous intéressent particulièrement relèvent d'un sous-domaine des SMA, la coordination multi-agent.

Coordination

Dans la communauté multi-agent, une grande partie des travaux traitent de près ou de loin de la coordination, et répondent aux questions suivantes. D'abord, comment faire en sorte que les comportements des agents ne conduisent pas à un comportement chaotique du système. Ensuite, comment faire converger les comportements individuels des agents vers un objectif global, lorsqu'il existe, que chaque agent isolé ne peut atteindre tout seul. Aussi, comment garantir un comportement correct des agents. Un comportement correct peut concerner une ressource partagée, qui ne peut être affectée à plusieurs agents en même temps, ou une contrainte de dépendance entre les actions des agents. Enfin, comment améliorer l'efficacité du système en accélérant la résolution d'un problème global grâce à l'entraide des agents.

Dans le domaine des langages de coordination, les agents sont des programmes séquentiels, et l'objectif de la coordination est de les faire se comporter ensemble comme un seul programme. Un modèle de coordination est « la glu qui relie des activités séparées dans un ensemble » alors qu'un langage de coordination est « l'incarnation linguistique d'un modèle de coordination » [Gelernter and Carriero, 1992]. Le modèle de coordination fournit le cadre dans lequel l'interaction entre programmes peut être exprimée. Les aspects relatifs à la création et de destruction de processus, leur communication, les aspects relatifs à leur distribution spatiale, ainsi que la synchronisation de leurs actions sont des aspects qui sont du ressort du modèle de coordination. Un langage de

coordination est composé des constructeurs linguistiques et d'opérateurs permettant la création de programmes, ainsi qu'une sémantique opérationnelle décrivant l'évolution d'un système sous l'action de ces opérateurs, dans le cadre du modèle de coordination.

Il est admis qu'il existe deux familles d'approches pour la coordination dans le domaine des langages de coordination : les modèles et langages de coordination orientées-contrôle et orientées-données [Papadopoulos and Arbab, 1998]. Dans la première, la coordination est réalisée par des coordinateurs : des programmes responsables d'établir les connexions entre les flux sortants et les flux entrant des autres programmes. Dans la seconde catégorie, la coordination est facilitée par un espace de données partagé, accessible par tous, et dans lequel l'accès aux données se fait par contenu.

Comme on le voit, la coordination multi-agent est considérée à un niveau d'abstraction plus élevé que la coordination en langages de coordination, son objectif est de garantir une interaction cohérente et non conflictuelle entre agents autonomes. À la différence de la coordination dans le domaine des langages de coordination, la coordination en SMA prend en compte les objectifs individuels des agents et les objectifs globaux du système.

Les avantages relatifs aux modèles de coordination orientés-données sont d'un intérêt certain pour la coordination multi-agent, et plus particulièrement pour les SMA ouverts. En effet, la communication dans les modèles de coordination orientés-données est « générative » : un agent génère une donnée, et elle devient dès lors indépendante de celui-ci. La communication générative est anonyme, un agent peut interagir avec un autre sans le connaître ; elle est découplée dans le temps et dans l'espace : deux agents ne sont pas contraints d'établir des rendez-vous afin d'interagir. D'ailleurs, certaines approches [Cabri *et al.*, 1999; Omicini and Zambonelli, 1999; Schumacher, 2001] tentent de faire converger les travaux de ces deux communautés.

Transport À la Demande

Le transport est une activité essentielle dans l'économie d'un pays. En 1978, elle était déjà de l'ordre de 15% du PNB[1] des États-Unis [Fisher, 1997], et cette proportion ne faiblit pas. Au Danemark par exemple, cette proportion était de 13 % en 1981 et de 15 % en 1994 [Larsen, 1999]. Optimiser l'utilisation la gestion des réseaux de transport peut donc résulter en des gains substantiels. De plus, dans le cadre des politiques de développement durable, les considérations écologiques tendent à encourager l'utilisation des transports en commun au détriment de l'usage des voitures privés. Pour encourager cette tendance, et vu la rigidité de l'offre de transport en commun traditionnel, le concept de TAD a émergé depuis les années 1970 aux États-Unis en réaction au choc pétrolier, et depuis, des centaines d'expériences ont vu le jour partout dans le monde, en particulier dans les pays développés. Le principe est de bâtir une offre de transport structurée par la demande, et réactive à celle-ci. Le TAD est un service qui se situe entre les transports en commun et les taxis individuels. Il est plus flexible que la première catégorie et moins coûteux que la seconde.

Du point de vue de la recherche informatique, le problème du TAD a intéressé plusieurs communautés. D'abord, et depuis les années 1960, les chercheurs en optimisation combinatoire ont intensivement étudié le problème du voyageur de commerce et ses variantes, dont fait partie le TAD, et des solutions exactes lui ont été proposées. Ensuite, au vu de la complexité des contraintes de ces problèmes, les chercheurs en intelligence artificielle ont proposé des méthodes heuristiques et méta-heuristiques capables de traiter des versions du problème de taille de plus en plus grande. Enfin, l'évolution technologique, le développement et la démocratisation des

[1] Produit National Brut

moyens de communication ont permis d'envisager des problèmes dynamiques, i.e. où les voyageurs s'annoncent au système pendant son exécution, auxquels il doit répondre rapidement. Dans ce contexte, les SMA offrent des possibilités de distribution des traitements pour ces problèmes de telle manière que les délais de réponses peuvent être substantiellement réduits.

Ce problème nous intéresse particulièrement pour les raisons suivantes :

- Il s'agit d'un problème difficile, i.e. les méthodes exactes ne peuvent traiter que des versions de petite taille, qui ne peuvent représenter des applications réelles,
- Il s'agit d'un problème dynamique, i.e. les données ne sont pas disponibles avant le démarrage du système, ce qui nécessite le développement de systèmes ouverts, et le problème demande des temps de réponses courts face à des demandes client,
- Avec le développement technologique, il est raisonnable de considérer des véhicules avec des capacités calculatoires embarquées, et des entreprises disposant de plusieurs serveurs. Dans ce contexte, le problème est, de fait, distribué. Si nous ne disposons pas d'une modélisation distribuée du problème, nous ne pouvons tirer profit de la distribution des processeurs,
- La considération d'un point de vue multi-agent permet d'imaginer de nouvelles mesures, de nouvelles heuristiques, non-envisagées par les approches centralisées.

Contributions

Le travail présenté dans le cadre de cette thèse est divisé en trois principales contributions.

Au niveau des modèles de coordination, nous proposons le modèle Acios[2], fondé sur une nouvelle représentation originale des données et des agents. Nous proposons un nouveau mécanisme d'appariement (pour la lecture et l'extraction des données), fondé sur des expressions, intégrant des opérateurs et des variables, tirant profit de la richesse de la structure de données et pouvant exprimer une interaction contextuelle. Une interaction est contextuelle lorsque la lecture d'une donnée peut être gardée par la présence où l'absence d'une ou de plusieurs autres données dans le système.

Au niveau des langages de coordination, nous proposons le langage Lacios[3] permettant d'écrire un SMA adhérant à Acios. Une sémantique opérationnelle lui est associée. Elle spécifie l'évaluation des expressions ainsi que la réduction des constructeurs du langage et son effet sur l'état du SMA. Nous avons implémenté un langage au dessus de Java, permettant une écriture proche du langage formel défini, et dont l'exécution garantit le respect de la sémantique opérationnelle que nous avons donné au langage. Nous proposons également une représentation des données échangées dans le SMA de manière à optimiser le processus d'appariement.

Au niveau du problème du Transport à la Demande, nous proposons une conception multi-agent du problème permettant de modéliser plusieurs variantes du problèmes d'une manière simple, ainsi qu'une nouvelle mesure pouvant être utilisée dans le cadre d'autres heuristiques de la littérature. Les solutions que nous proposons adhérent au modèle Acios, et sont écrites dans Lacios. Notre contribution est présenté en deux phases. Dans la première, nous traitons un problème moins contraint que le problème TAD, celui du problème de tournée de véhicules avec fenêtres temporelles euclidien (problème où le réseau est un plan euclidien). L'objectif est d'abord, de montrer une modélisation basique du problème et comment notre langage permet d'exprimer ses contraintes, ensuite d'introduire l'intuition de notre mesure et son application dans le contexte d'un problème euclidien. Dans la seconde phase, nous étendons notre modélisation au problème plus contraint du TAD, et notre mesure au cas des problèmes quelconques

[2] *Agent Contextual Interaction in Open Systems*
[3] *Language for Agent Contextual Interaction in Open Systems*

(travaillant avec des réseaux et non des plans). Nos propositions sont validés avec des expérimentations sur des *benchmarks* de la littérature.

Organisation du rapport

Cette thèse est divisée en deux parties. Chaque partie est composée de trois chapitres. Les trois premiers chapitres présentent l'état de l'art des trois différents domaines sur lesquelles nous travaillons, à savoir : les SMA et la coordination, les algèbres de processus et le Transport à la Demande. Notre présentation n'est pas exhaustive, et a pour objectif de donner les notions essentielles pour la compréhension de notre travail.

Première partie. État de l'art

La première partie sert à positionner notre travail et à présenter les travaux antérieurs qui nous inspirent.

Chapitre 1. La coordination et les systèmes muti-agents

Le chapitre 1 présente un état de l'art de la coordination en SMA et en langages de coordination. Un intérêt particulier est porté aux modèles de coordination orientés-données qui inspirent notre travail.

Chapitre 2. Algèbres de processus pour la coordination

Le chapitre 2 présente les algèbres de processus avant de s'étendre sur la sémantique des langages de coordination orientée-données qui ont été proposés et les principaux résultats auxquels ils ont donné lieu.

Chapitre 3. Le problème du transport à la demande

Le chapitre 3 donne l'état de l'art du domaine dans lequel nous appliquons le modèle et le langage. Il s'agit des problèmes de tournées de véhicules dynamiques et du problème du Transport À la Demande.

Deuxième partie. Contributions

Nous proposons le modèle de coordination Acios, le langage de coordination Lacios, ainsi que le système Lacios-TAD, une réalisation en Lacios pour les problèmes de tournées de véhicules dynamiques et leur extension TAD. Le modèle Acios définit une partie des constructeurs linguistiques du langage et les aspects conceptuels s'y afférant. Le langage Lacios complète la définition de la syntaxe et définit la sémantique du langage. Enfin, le système Lacios-TAD est un système coordonné via Acios, écrit dans Lacios et qui modélise une solution pour le problème du Transport À la demande.

Chapitre 4. Modèle de coordination Acios

Le chapitre 4 présente le modèle de coordination Acios. Étendant les modèles de coordination orientés-données présentés dans le chapitre 1, Acios se fonde sur une structure de données utilisant des couples *propriétés-valeurs* permettant un mécanisme d'appariement expressif et gardé

par l'état des agents. L'échange de données dans Acios est sécurisé autant par par le concepteur du système que par les agents qui y évoluent. Le modèle Acios permet la conception de SMA ouverts dans le sens où les agents peuvent rejoindre et quitter le système librement, et dans le sens où il peut modéliser l'action d'un système externe sur le comportement des agents du SMA.

Chapitre 5. Langage de coordination Lacios

Dans le chapitre 5, nous proposons un langage de correspondant à notre modèle de coordination, nous y spécifions le comportement d'un système adhérant au modèle en en proposant une sémantique opérationnelle. Une implémentation du modèle au dessus de Java est proposée. Elle permet de programmer un SMA dans une syntaxe concise, proche de la syntaxe formelle, en garantissant son adhésion au modèle de coordination. Nous y proposons également plusieurs solutions pour limiter la complexité de l'appariement en utilisant des structures d'index et de treillis.

Chapitre 6. Le système Lacios-TAD

Le chapitre 6 propose d'appliquer le modèle Acios pour la modélisation du problème du TAD. Nous introduisons une nouvelle mesure pouvant être utilisée avec différentes heuristiques d'insertion de la littérature. La mesure est appliquée au problème euclidien de tournée de véhicules avec fenêtres de temps. Le système est étendu afin de traiter le problème du Transport À la Demande, en y intégrant les contraintes additionnelles du problème.

Première partie

État de l'art

Chapitre 1

La coordination et les Systèmes Multi-Agents

Sommaire

1.1 Introduction

La coordination est une nécessité dans tout système où le contrôle est distribué en général et dans les systèmes multi-agents en particulier. Elle consiste en la limitation des interactions possibles entre les composants d'un système à celles qui permettent de forcer le comportement global désiré. L'acception du terme coordination dans la communauté des systèmes multi-agents (SMA) et dans celle des langages de coordination (LC) n'est pas la même. Différentes classifications essaient de concilier les deux acceptions, ce qui a eu pour résultat une proposition de les classifier en coordination subjective et coordination objective [Omicini *et al.*, 2004; Schumacher, 2001]. Dans les modèles de coordination subjective (vision SMA), les agents seraient des entités « coordinatrices », ils seraient les « sujets » de la coordination. La coordination serait assurée en agissant sur les comportements individuels des agents, en faisant en sorte qu'ils soient en concordance les uns avec les autres. Dans les modèles de coordination objective (vision LC), ils seraient des entités « coordonnées », ils seraient les « objets » de la coordination. La coordination est considérée comme une activité normative facilitée par un composant dédié du système.

Pour notre part, nous considérons que les problématiques de la coordination en SMA et en LC se situent à deux niveaux d'abstraction différents. La différence essentielle réside dans la représentation des buts des agents. En SMA, elle détermine le modèle et le mécanisme de coordination, alors qu'en langage de coordination, elle n'est simplement pas considérée. Les langages de coordination cherchent à doter les langages de calcul d'un langage de coordination qui permettrait à des entités indépendantes de se comporter ensemble d'une manière cohérente, et ce quelque soit leurs objectifs, ou l'éventuel objectif recherché du système. Les travaux des deux communautés sont en réalité complémentaires : les modèles de coordination multi-agent se fondent sur des modèles et langages de coordination, choisis selon le type de problème traité. Les deux approches de la coordination nous intéressent dans le cadre de cette thèse. D'une part, nous proposons un modèle et langage le développement de SMA, ils étendent ceux développés dans la communauté des langages de coordination. D'autre part, nous traitons un problème spécifique, qui nécessite l'utilisation de mécanismes de coordination développés dans la communauté des SMA.

Ce chapitre présente un état de l'art des SMA et des différents modèles et langages de coordination de la littérature. Cette présentation ne couvre évidemment pas tous les aspects de la recherche en SMA, ou en langages de coordination. Dans ce chapitre, nous introduisons les éléments nécessaires afin de positionner notre travail, le lecteur intéressé est invité à se référer à [Ferber, 1995; Wooldridge, 2002; Briot and Demazeau, 2001] pour un état de l'art des SMA et [Papadopoulos and Arbab, 1998] pour un état de l'art des langages de coordination. Ce chapitre est structuré comme suit. La section 1.2 introduit le domaine des systèmes multi-agents. La section 1.3 présente l'interaction dans les systèmes multi-agents. La section 1.4 présente les modèles et mécanismes de coordination en SMA. La section 1.5 présente les langages de coordination orientée-contrôle et orientée-données, avec une attention particulière portée sur la seconde catégorie, qui inspire notre travail.

1.2 Les Systèmes Multi-Agents

L'intelligence Artificielle (IA) tente de résoudre des problèmes complexes en se fondant sur des entités cognitives. Face à la nature distribuée de certains problèmes et leur grande complexité, l'IA a été confrontée à un certain nombre de limites conceptuelles et matérielles. Grâce au développement de l'informatique distribuée, l'IA a donné lieu à un nouveau domaine : l'Intelligence Artificielle Distribuée (IAD) [Lesser, 1995]. Les systèmes Multi-Agents (SMA) sont un paradigme ayant pour objectif de canaliser l'IAD dans le cadre d'une approche conceptuelle simple dans l'appréhension de problèmes complexes [Adams, 2001]. Aujourd'hui, les termes IAD et SMA sont généralement utilisés d'une manière interchangeable. Les sources d'inspiration des recherches en SMA sont très larges, allant de l'économie et la théorie des jeux jusqu'aux sciences humaines et sociales, en passant par les théories des organisations. Sycara donne cette définition des SMA :

Définition Système Multi-Agent [Sycara, 1998]

> Un SMA est un réseau d'agents faiblement couplés qui interagissent afin de résoudre des problèmes qui dépassent les capacités ou les connaissances de chacun[4].

[4] *A multiagent system can be defined as a loosely coupled network of problem solvers that interact to solve problems that are beyond the individual capabilities or knowledge of each problem solver*

Deux aspects fondamentaux émergent de cette définition. D'une part, les capacités et les connaissances sont distribuées. D'autre part, la résolution du problème par un seul des agents n'est pas envisageable, ce qui rend impératif leur interaction. Différentes définitions de ce que c'est qu'un agent existent dans la littérature. Ferber propose une définition regroupant neuf caractéristiques pour la définition d'un agent :

Définition Agent [Ferber, 1995]

On appelle agent une entité physique ou virtuelle

1. qui est capable d'agir dans un *environnement*,
2. qui peut *communiquer* directement avec d'autres agents,
3. qui est mue par un ensemble de tendances (sous la forme d'*objectifs* individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser),
4. qui possède des ressources propres,
5. qui est capable de *percevoir* (mais de manière limitée) son environnement,
6. qui ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune),
7. qui possède des *compétences* et offre des services,
8. qui peut éventuellement se reproduire,
9. dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit.

Les termes mis en italique sont les plus communs dans les définitions d'agent. À savoir qu'un agent évolue dans un environnement, qu'il en a une perception dessus, qu'il peut communiquer et qu'il a des objectifs et des compétences. Une définition plus concise est donnée par Wooldridge and Jennings :

Définition Agent [Wooldridge and Jennings, 1995]

Un agent est un système informatique qui est *situé* dans un *environnement* et qui est capable d'appliquer des actions *autonomes* dans le but de satisfaire ses *buts*.

D'une manière générale, les propriétés communément admises pour les agents peuvent être synthétisées comme suit [Jennings *et al.*, 1998] :

- **Autonomie**
  Un agent devrait être capable d'agir sans intervention directe d'un humain ou d'un autre agent. Il doit avoir un contrôle sur ses propres actions et son état interne.
- **Caractère situé**
  Un agent est situé dans un environnement. Il peut agir sur son environnement et il en reçoit des signaux qu'il interprète. Toutefois, Weyns et Holvoet [Weyns and Holvoet, 2004] observent que généralement, cette acception de la situation des agents n'implique pas l'existence d'une entité « explicite » appelé environnement. Certains travaux arguent l'existence d'une telle entité d'une manière explicite [Balbo, 2000b; Platon *et al.*, 2005; Weyns, 2006].

- **Réactivité**
  Un agent devrait pouvoir réagir rapidement à une perception d'un changement de son environnement.
- **Proactivité**
  En phase avec son autonomie, un agent ne devrait pas simplement réagir à des événements dans son environnement, mais également pouvoir initier des actions en concordance avec ses objectifs.
- **Aptitudes sociales**
  Un agent devrait pouvoir interagir avec ses pairs ou avec un humain afin de satisfaire ses objectifs ou aider les autres à satisfaire les leurs.

L'aptitude sociale est certainement l'un des points centraux dans la définition d'un SMA comme étant un système distribué. Si les agents vivent en autarcie, sans interaction avec le monde extérieur, ils ne forment pas ensemble un même système, mais plusieurs systèmes isolés. L'interaction est en effet l'un des points clés dans les SMA, et mobilise un effort de recherche considérable.

1.3 L'interaction dans les SMA

L'interaction est un processus dans lequel les agents s'engagent afin de s'assurer que, prises ensemble, leurs actions s'exécutent d'une manière cohérente. Ferber propose la définition suivante de l'interaction :

Définition Interaction [Ferber, 1995]

> Une interaction est une mise en relation dynamique de deux ou plusieurs agents par le biais d'un ensemble d'actions réciproques.

Les agents interagissent à travers un ensemble d'événements pendant lesquels les agents sont en relation les uns avec les autres soit directement soit par le biais de l'environnement. La notion d'interaction suppose :

- La présence d'agents capables d'agir et/ou de communiquer
- Des situations susceptibles de servir de point de rencontre entre agents : collaboration, déplacement de véhicules amenant à une collision, utilisation de ressources limitées, régulation de la cohésion d'un groupe.
- Des éléments dynamiques permettant des relations locales et temporaires entre agents : communication, choc, champ attractif ou répulsif, etc.
- Un certain jeu dans les relations entre les agents leur permettant à la fois d'être en relation, mais aussi de pouvoir se séparer de cette relation, c'est-a-dire de disposer d'une certaine autonomie.

1.3.1 L'interaction directe

La communication est un outil permettant à des agents d'interagir directement. L'interaction est directe dans le cas où un agent envoie intentionnellement un message vers un autre agent. Ce type d'interaction est appelé communication « point à point » ou dyadique. La diffusion ou *broadcast* est aussi une communication directe et consiste en l'envoi d'un message par un agent vers tous les autres agents du système. Le *multicast* consiste en l'envoi d'un message par un agent vers un sous-ensemble des agents du système.

L'un des champs de recherche en SMA est le développement de langages de communication agent. Les langages de communication les plus connus sont KQML et le langage FIPA-ACL.

Un langage de communication agent fournit un ensemble d'actes de communication que les agents peuvent utiliser. Le but de ces actes est d'acheminer des informations sur l'état mental de l'agent avec l'objectif d'affecter l'état mental du partenaire. Les actions de communication des langages de communication agent sont constitués d'un ensemble de couches distinctes. Par exemple, KQML est composé de trois couches. La première concerne le contenu informationnel de l'action de communication. Le contenu est exprimé dans un langage communément agréé, e.g. logique propositionnelle, du premier ordre, etc. La seconde couche exprime une attitude particulière envers le contenu sous forme d'un « acte de langage ». Par exemple, l'acte `tell` exprime que le contenu est tenu pour valide, `untell` exprime que le contenu est tenu pour non valide ou `ask` afin de demander l'avis du partenaire concernant le contenu. Enfin, la troisième couche traite du mécanisme de communication, impliquant des aspects tels que le canal sur lequel la communication est établie ainsi que sa direction (envoi ou réception).

1.3.2 L'interaction indirecte

L'interaction indirecte divise un acte d'interaction en deux parties, en intercalant un troisième protagoniste qui s'en fait l'intermédiaire. Le besoin d'avoir une troisième entité se fait sentir dans plusieurs situations. Par exemple, dans un système ouvert, un nouvel agent ne peut connaître les autres agents, ni ses possibilités de communication. Ce problème est connu sous le nom de « problème de connexion » [Davis and Smith, 1983] ou comment « trouver les autres agents qui ont l'information dont tu as besoin »[5] [Wong and Sycara, 2000]. C'est dans ce but que les agents intermédiaires ont été introduits. Dans [Sycara *et al.*, 1997], les auteurs définissent un agent intermédiaire comme une entité qui n'est ni demandeur ni fournisseur d'un service mais qui participe à l'interaction entre un demandeur et un fournisseur. Le demandeur a des préférences et le fournisseur a des capacités, et l'agent intermédiaire se charge de l'appariement entre les préférences de l'un et les capacités de l'autre.

L'interaction indirecte englobe également l'utilisation « tableau noir » [Hayes-Roth, 1985] et l'utilisation de l'environnement (voir e.g. [Balbo, 2000b; Weyns, 2006]). La « stigmergie » permet aux agents de laisser des traces dans l'environnement qui peuvent peut être utilisés par les autres agents afin de guider leurs actions. L'interaction indirecte englobe aussi l'écoute flottante et l'attention mutuelle (*mutual awareness*) [Ricci *et al.*, 2006; Saunier *et al.*, 2006; Zargayouna *et al.*, 2006a]. Dans ces travaux, de nouveaux modes d'interaction indirecte sont envisagés qui permettent à des agents, qui ne sont pas protagonistes de la communication, de participer, en recevant certains messages qui ne leur sont pas destinés.

Étant donné des agents en interaction, les modèles et mécanismes de coordination permettent un SMA d'avoir un comportement cohérent, i.e. où les interactions ne mènent pas à des situations non désirées.

1.4 Modèles et mécanismes de coordination multi-agent

Selon [Nwana, 1994; Jennings, 1996], il y a différentes raisons pour lesquelles des agents ont besoin de se coordonner :

– Empêcher un comportement chaotique du système. Par définition, aucun agent n'a de vue globale de tout le système. Dans cette situation, il est assez simple de tomber dans des configurations chaotiques, où le comportement du système en tant que tout n'est pas prévisible.

[5] *finding the other agents that have the information or capabilities that you need*

- Satisfaire à des contraintes globales. Ces contraintes existent très souvent, et un groupe d'agents est considéré comme ayant réussi sa tâche s'il ne les viole pas. Un budget ou une date butoir à ne pas dépasser sont autant de contraintes globales à satisfaire par un système multi-agent, et pour y parvenir, les agents doivent interagir.
- Atteindre un objectif global, que chaque agent isolé ne peut atteindre tout seul. Différents agents peuvent avoir des expertises et/ou sources d'information différentes, la mutualisation de ces expertises et sources d'information est nécessaire afin de parvenir à l'objectif global.
- Respecter les dépendances entre les actions individuelles des agents.
- Améliorer l'efficacité du système. Quand des agents découvrent des informations que les autres peuvent utiliser pour accomplir leurs tâches, les partager fait gagner du temps à tous, accélérant la résolution du problème traité.

Nous présentons dans ce qui suit les modèles et mécanismes de coordination dans un SMA.

1.4.1 Modèles de coordination

Modèles orientés-tâches

Dans les modèles de coordination orientée-tâches, on présuppose la présence d'un but global partagé par tous les agents. Ainsi, tous les agents ont pour tâche de satisfaire l'objectif global du problème. Un agent *gestionnaire* alloue des sous-tâches de la tâche correspondant à l'objectif global aux autres agents. Les agents dans ce cas sont dans une situation de coopération, puisque le but est d'optimiser l'efficacité du groupe d'agents. Ces modèles sont applicables dans les cas de résolution distribuée de problèmes (*Distributed Problem Solving*).

Modèles orientés-agent

Dans les modèles de coordination orientée-agents les agents sont autonomes ou semi-autonomes, et ne partagent donc pas forcément un objectif global souhaité. Dans cette configuration, chaque agent cherche à optimiser ses propres objectifs. La coordination est nécessaire pour qu'un agent ne pouvant satisfaire une tâche par ses propres moyens puisse augmenter son gain en accomplissant cette tâche avec d'autres agents.

1.4.2 Mécanismes de coordination multi-agents

Les modèles de coordination orientée-agent se focalisent sur le comportement des agents afin d'aboutir à un système coordonné. Plusieurs approches existent pour la coordination des agents dans un SMA. Dans [El Fallah-Seghrouchni, 2001], Fallah-Seghrouchni les classe dans six catégories :

- Résolution distribuée de problèmes
- Structuration organisationnelle
- Protocoles
- Négociation
- Formation de coalitions
- Planification multi-agent

Résolution distribuée de problèmes

Dans la résolution distribuée de problèmes, les agents sont coopératifs et essaient d'aboutir collectivement à une solution au problème global. Notre problème du transport à la demande se

situe dans cette catégorie. Lesser [Lesser and Corkill, 1988] a proposé le modèle FA/C (*Functionally Accurate Model*) (modèle fonctionnellement exact) où les activités des agents sont interdépendantes, ces derniers échangeant des informations de haut niveau (niveau *méta*) tels que leurs choix, buts, etc. Le niveau *méta* est défini ici d'une manière statique. Durfee et Lesser [Durfee and Montgomery, 1991] ont proposé le modèle PGP (*Partial Global Planning*) (planification globale partielle) où le niveau méta est dynamique. Les agents coordonnent leurs plans, en échangeant les informations pertinentes à leur exécution. Trois niveaux de représentation de plans sont gérés par un agent. Le plan local définissant les actions de l'agent, le plan nodal est une abstraction du plan local destinée à être communiquée avec les autres agents et le plan global partiel contenant les buts des autres agents susceptibles d'interagir avec l'agent propriétaire du plan.

Decker [Decker and Lesser, 1992] a proposé une généralisation de ce modèle appelée GPGP (*Generalized Global Partial Planing*) où la coordination doit respecter des contraintes d'exécution temps réel. Dans GPGP, les buts sont organisés en hiérarchie et incluent des informations telles que l'ordre des buts, leurs durées, leurs priorités etc. GPGP offre l'avantage de permettre aux agents de prédire voire d'anticiper les comportements des autres agents [El Fallah-Seghrouchni, 2001].

Structuration organisationnelle

C'est le scénario le plus simple car il exploite une catégorisation préexistante. Cette catégorie exploite le fait que, lorsqu'elles existent, les structures organisationnelles définissent (implicitement) les responsabilités, les capacités, la connectivité et les flux de contrôle de chaque agent, selon le rôle qu'il joue dans le système. L'organisation y est vue comme une relation à long terme, prédéfinie, entre les agents [Durfee *et al.*, 1987]. Trois dimensions permettent de définir une typologie des structures organisationnelles [El Fallah-Seghrouchni, 2001] :

- fonctionnelle : consiste à organiser les agents selon leurs rôles ou leurs fonctions au sein du système, en prenant en compte leurs capacités et la nature des tâches qui leur incombent ;
- spatiale : consiste à affecter des agents à des régions bien délimitées de l'environnement dans lequel ils évoluent ;
- temporelle : consiste à des organisations dynamiques, c'est à dire qui évoluent dans le temps, dans les cas où les agents évoluent, apprennent ou acquièrent des connaissances nouvelles, influençant ainsi la structure de l'organisation.

Dans [Durfee *et al.*, 1987], l'organisation est hiérarchique (cas le plus traité) et dans [Ferber *et al.*, 2000], chaque **agent** joue un **rôle** dans un **groupe** (modèle AGR - Agent, Groupe, Rôle). Dans ce dernier cas, la communication est un *multicast* qui dépend des groupes et des rôles des agents à contacter. Dans le premier cas, celui de la structuration hiérarchique, l'implémentation est effectuée de deux manières :

- **Configuration maître-esclave**. L'agent maître planifie et distribue des fragments de plans aux esclaves. Les esclaves peuvent - ou pas - communiquer entre eux mais doivent à la fin de leur exécution renvoyer leurs résultats à l'agent maître,
- **Tableau noir**. Cette implémentation exploite l'architecture en tableau noir [Hayes-Roth, 1985]. Les sources de connaissances sont remplacés par des agents qui écrivent et lisent depuis le tableau noir. L'agent *scheduler* ordonnance les lectures/écritures.Ce schéma est utilisé par des systèmes tels que DFI[6] [Werkman, 1990] et SMAK[7] [Kearney *et al.*, 1994].

[6] *Designer Fabricator Interpreter*
[7] *Sharp Multi-Agent Kernel*

Protocoles

Le principe de la coordination par protocoles est de définir un ordonnancement de messages échangés, qui doit être respecté par les agents participants. Une technique désormais classique pour l'allocation de tâches et des ressources à un ensemble d'agents est le *Contract Net Protocol* (CNP) [Smith, 1980], que les auteurs dans [Aknine *et al.*, 2004a] ont étendu au cas où les agents sont impliqués dans plusieurs négociations. Dans cette approche, un agent peut assumer un des deux rôles suivants :

- Un *manager* qui transforme un problème en sous-problèmes et cherche des « sous-traitants » pour les résoudre, il assume également le pilotage du processus de résolution du problème initial,
- Un sous-traitant qui accomplit une sous-tâche. Il peut à son tour devenir un *manager* pour son sous-problème et chercher des sous-traitants pour en accomplir les sous-tâches.

Afin de localiser les sous-traitants, les *managers*, procèdent comme suit :

- Le *manager* annonce la tâche,
- Les sous-traitants évaluent la tâche par rapport à leurs capacités et engagements,
- Les sous-traitants proposent une offre au *manager*,
- Le *manager* évalue les offres reçues, choisit un sous-traitant et lui alloue le contrat,
- Enfin, le *manager* suit l'exécution et attend le résultat du contrat.

Bien que très simple, le CNP offre plusieurs avantages. D'une part, il convient aux systèmes ouverts puisque les agents peuvent être introduits et extraits dynamiquement du système. Par conséquent, il offre une forte tolérance aux pannes. D'autre part, il offre un mécanisme naturel d'équilibrage de charges, puisqu'un agent déjà occupé ne proposera pas d'offres.

Cependant, le mécanisme de coordination par les contrats n'est applicable que dans un nombre restreint d'applications. Il peut être utilisé lorsque [Huhns and Singh, 1994] :

- Le problème a une structure hiérarchique bien définie,
- Le problème peut être décomposé en des sous-problèmes de grande granularité,
- Il existe un couplage minimal entre sous-tâches.

Fondée sur un schéma de communication par diffusion, la coordination par contrats est fortement gourmande en bande passante, et est donc irréaliste pour nombre d'applications réelles.

Négociation

Bussman et Miller [Bussmann and Müller, 1993] définissent la négociation comme « le processus de communication d'un groupe d'agents afin d'atteindre un accord mutuellement accepté sur un certain sujet » [8], elle constitue une étape importante dans un processus de coordination [Davis and Smith, 1983]. Sycara [Sycara, 1990] met l'accent sur le fait que, afin de négocier effectivement, les agents doivent raisonner sur les croyances, désirs et intentions des autres agents. Ainsi, différents travaux traitent de la représentation et la maintenance des modèles de croyance, du raisonnement sur les croyances des autres agents et sur la manière d'influencer ces intentions et ces croyances. Ainsi, toutes sortes de techniques d'Intelligence Artificielle ont été utilisées telles que le raisonnement à partir de cas, la révision de croyances, le raisonnement à partir de modèles, etc. [Nwana *et al.*, 1996]. Dans les approches fondées sur la négociation, les agents possèdent une stratégie de prise de décision, leur objectif est de maximiser leur « fonction d'utilité locale ». Parmi les travaux les plus significatifs dans ce domaine figurent les modèles de négociation inspirés de la théorie des jeux et de la théorie d'aide à la décision.

[8] *Negotiation is the communication process of a group of agents in order to reach a mutually accepted agreement on some matter*

Les approches fondées sur la théorie d'aide à la décision se fondent essentiellement sur des modèles de décision multicritères [El Fallah-Seghrouchni *et al.*, 2000; Moraitis and Tsoukiàs, 1996; Moraitis and Tsoukiàs, 1999]. Par exemple, [Moraitis and Tsoukiàs, 1999] proposent une représentation des plans des agents fondée sur un graphe multicritère. Dans les approches fondées sur la théorie des jeux (e.g. [Beaufils *et al.*, 1996]), l'objectif est de trouver une solution « consensus » acceptable par tous les participants au jeu. La solution trouvée doit satisfaire des critères tels que le critère d'efficacité Pareto. Une solution est dite « Pareto optimale » si elle n'est dominée par aucune autre solution, i.e. aucune autre solution possible ne donne plus à un agent sans en donner moins à un autre. Les modèles issus de la théorie des jeux appliqués aux SMA modélisent la négociation comme un jeu où les intérêts des joueurs (agents) est représenté par une fonction d'utilité. Sous les conditions théoriques fixées par la théorie des jeux, il est possible de garantir la convergence d'une négociation vers une situation où aucune fonction d'utilité n'est affaiblie. Néanmoins, ces conditions théoriques restreignent grandement le cadre applicatif des modèles issus de la théorie des jeux, soit car trop complexes pour être appliqués, soit car les conditions réelles ne satisfont pas aux conditions théoriques du modèle (telles que la rationalité des agents ou la symétrie de l'information, e.g. équilibre de Nash [Nash, 1950]).

Formation de coalitions

Dans [El Fallah-Seghrouchni, 2001], El Fallah définit une coalition comme une organisation à court terme fondée sur des engagements spécifiques et contextuels, qui permet aux agents de coexister tout en profitant de leurs compétences respectives. Les recherches sur la formation de coalitions essaient d'optimiser la formation de coalition (maximisation de profit total généralement), qui nécessite des algorithmes de grande complexité. Des hypothèses sont donc émises afin de trouver des solutions en des temps raisonnables. Les hypothèses formulées sont notamment l'altruisme des agents, leur coopération, avoir la même rationalité pour tous les agents, avoir la même stratégie [Shehory and Kraus, 1996] ou une stratégie imposée [Zlotkin and Rosenschein, 1994], l'utilisation d'intégrations incrémentales des agents à une coalition (ne prenant pas en compte les éventuels futurs participants) [Ketchpel, 1994] ou la contrainte d'appartenir à une seule coalition.

[Vauvert and El Fallah-Seghrouchni, 2000a] n'impose pas l'application d'une même stratégie par tous les agents, ni le partage des modèles de préférence ou de critères à optimiser. Chaque agent dispose de sa propre rationalité économique individuelle, ses critères ne sont pas connus par les autres et seules ses préférences sont communiquées. Il peut changer de stratégie, changer ses critères, leurs pondération ou ses préférences. L'algorithme proposé garantie la convergence vers un consensus grâce à l'introduction du concept d'alliances : coalitions minimales regroupant deux agents ayant les préférences les plus proches (par rapport à une distance entre préférences).

Dans [Aknine *et al.*, 2004b], les auteurs proposent un algorithme qui permet de former des coalitions à partir des préférences des agents. Dans [Caillou *et al.*,], ils proposent un algorithme qui permet de restructurer des coalitions dans les systèmes ouverts, sans avoir a relancer l'algorithme lors de l'apparition de nouveaux agents.

Planification multi-agent

Afin d'éviter des actions conflictuelles ou inconsistantes, une approche majeure en coordination multi-agent est la planification multi-agent, et dès le début des années 1980, plusieurs travaux ont adopté la planification multi-agent comme approche pour la coordination [Georgeff, 1983; Durfee, 1988]. Un plan détaille les actions et interactions futures des agents du système,

nécessaires à la réalisation de leurs buts. La plupart de ces travaux considèrent des agents capable d'assurer une post-planification. En effet, les plans doivent souvent être reconsidérés, exécution et replanification sont alors alternées.

La synthèse de plans exige un raisonnement sur la manière dont les actions des différents agents peuvent interférer. La synchronisation de plans peut être effectuée lors de différentes phases de la résolution du problème : durant la phase de décomposition du problème [Corkill, 1979], lors de la construction des plans [Corkill, 1979], après la construction des plans [Georgeff, 1983], ou alors en introduisant, dans la phase de planification, un processus de raisonnement sur les dépendances possibles entre plans [Ephrati *et al.*, 1995; El Fallah-Seghrouchni and Haddad, 1996b]. La planification peut être centralisée ou distribuée. Dans la planification centralisée, il existe généralement un agent « coordinateur » qui génère un plan partiellement ordonné, où les actions parallèles sont exécutées par des agents différents de manière concurrente. Le coordinateur veille à la détection d'une inconsistance et de conflits.

Dans la planification multi-agent distribuée, deux situations sont distinguées : la planification pour un plan centralisé et la planification pour un plan distribué. Dans la première situation, toutes les techniques utilisées en résolution distribuée de problèmes peuvent être utilisées. En effet, l'objectif est, pour un ensemble d'agents de pouvoir fusionner leurs plans partiels d'une manière distribuée en un plan centralisé, correspondant généralement à une tâche complexe à réaliser par le groupe d'agents. Dans la deuxième situation, plusieurs agents peuvent planifier et exécuter leurs plans d'une manière concurrente et autonome, et ce indépendamment d'un plan ou d'un éventuel but global du système.

Dans cette catégorie, Von Martial [von Martial, 1992] propose une approche fondée sur une taxonomie des relations existant entre les différents plans, où les agents négocient pour résoudre les relations existant entre leurs plans. Cependant, il ne coordonne que deux agents à la fois. Alami [Alami *et al.*, 1994] propose une approche fondée sur la fusion de plans qui coordonne n agents à la fois. El Fallah-Seghrouchni et Haddad [El Fallah-Seghrouchni and Haddad, 1996a] proposent une approche fondée sur la notion d'ordre partiel entre actions, où les actions sont connectées par des liens causaux, qui permet de résoudre les interactions positives (actions redondantes ou facilitant le travail d'autres agents) et les situations négatives (situations de conflit). Dans [El Fallah-Seghrouchni and Haddad, 1996b], les auteurs enrichissent le formalisme de représentation de plan, se fondant sur une extension des réseaux de Petri appelée Réseaux de Petri Récursif [El Fallah-Seghrouchni and Haddad, 1995]. Ceci résulte, entre autres, en la possibilité de vérifier la consistance des plans produits à l'aide d'algorithmes efficaces [El Fallah-Seghrouchni, 2001].

L'approche centralisée partage les mêmes inconvénients que la communication maître-esclave, en produisant un goulet d'étranglement au niveau de l'agent qui prend la responsabilité de gérer les plans locaux. Elle a en revanche l'avantage d'être plus simple à mettre en place que l'approche distribuée.

Comme on vient de le voir, les modèles et mécanismes de coordination multi-agents permettent à un groupe d'agents de se comporter d'une manière cohérente, en prenant en compte l'éventuel objectif du système et les buts des agents. Dans la communauté des langages de coordination, l'objectif du système et ceux des processus n'est pas pris en compte. L'objectif de la coordination est de permettre l'implantation de systèmes parallèles et distribués en composant des processus séquentiels, via un autre processus coordinateur ou un espace de données. Les travaux en langages de coordination sont complémentaires avec ceux des SMA. Notre objectif dans le cadre de cette thèse est de proposer un langage de coordination dans le sens de la communauté des langages de coordination, qui soit assez riche pour décrire des agents et par ce fait, de permettre la mise en place de mécanismes de coordination au sens SMA.

1.5 Les langages de coordination

Dans le domaine des langages de coordination, il est maintenant admis qu'une application parallèle est principalement divisée en deux parties : le *calcul* et la *coordination* [Gelernter and Carriero, 1992]. Un langage de coordination est orthogonal à un langage de calcul, c'est à dire qu'un langage de coordination étend un langage de calcul avec des fonctionnalités facilitant l'implémentation d'applications distribuées. On dit également qu'un langage de coordination forme le pendant linguistique d'un modèle de coordination [Gelernter and Carriero, 1992], c'est à dire qu'un langage de coordination doit offrir des constructeurs linguistiques, soit sous forme d'appels de librairies, soit sous forme d'extensions (des primitives), afin de matérialiser le modèle de coordination. Dans ces domaines, la définition la plus largement acceptée de la notion de coordination est la suivante.

Définition Coordination [Gelernter and Carriero, 1992]

> La coordination est le processus de construire des programmes en collant ensemble des composantes actives[9].

Un modèle de coordination peut être vu comme un triplet $\langle E, M, L\rangle$ où E représente les entités coordonnées, M le média utilisé afin de les coordonner, et L le cadre sémantique auquel le modèle adhère [Wegner, 1996]. Deux principales catégories de modèles de coordination ont été identifiées : les modèles de coordination orientée-contrôle et les modèles de coordination orientée-données [Papadopoulos and Arbab, 1998]. Dans la suite de ce chapitre, nous donnons une brève présentation des modèles de coordination orientée-contrôle, puis détaillons les modèles de coordination orientée-données. Dans cette famille des langages de coordination, les termes agents et processus sont utilisés d'une manière interchangeable. Le lecteur intéressé peut trouver une étude qui couvre une grande partie de ces deux types de modèles dans [Papadopoulos and Arbab, 1998].

1.5.1 La coordination orientée-contrôle

Dans les modèles de coordination orientée-contrôle, un système coordonné évolue en observant des changements dans les états des processus, et éventuellement en diffusant des événements, les processus sont vus comme des boîtes noires. Les processus communiquent avec leur environnement avec des interfaces bien définies, généralement appelées des « ports d'entrée » et « ports de sortie ». Les relations producteur-consommateur sont formées par des canaux entre les ports de sortie des producteurs et les ports d'entrée des consommateurs. Ces canaux sont point-à-point, ou parfois ils forment une relation un-à-plusieurs, dans le cadre d'une diffusion restreinte. Outre les ports, les processus génèrent des événements qu'ils envoient à leur environnement, avec pour objectif d'informer les processus à propos de leur état actuel, ou d'un changement de leur état. La figure 1.1 illustre ces concepts, elle présente une configuration avec un producteur et deux consommateurs. Le producteur (processus1) a un port d'entrée et deux ports de sortie. Le premier consommateur (processus2) a un port d'entrée et un port de sortie, tandis que le deuxième consommateur (processus3) a deux ports d'entrée et un port de sortie. Les canaux (représentés par des flèches) connectent les ports de sortie du producteur aux ports d'entrée des consommateurs avec éventuellement plusieurs canaux sortant ou entrant du même port. De plus, processus1 et processus3 déclenchent et/ou observent des événements (Ev1, Ev2 et Ev3). La plupart des modèles de coordination instancient ce modèle, et se différencient par la

[9] *Coordination is the process of building programs by gluing together active pieces.*

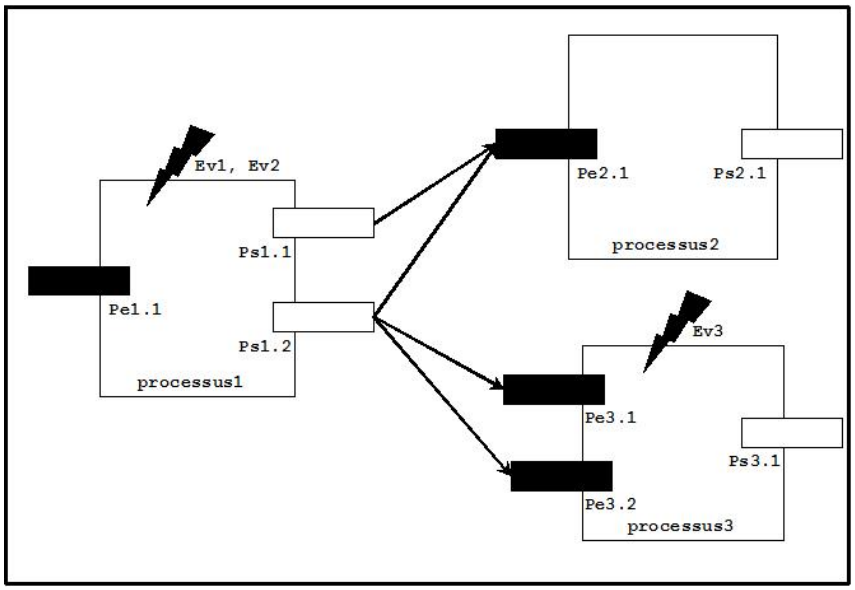

FIG. 1.1 – Principe de la coordination orientée-contrôle

manière de le mettre en oeuvre. Un processus exécute généralement un programme séquentiel, envoie des données sur son port de sortie et en reçoit sur ses ports d'entrées. Cette famille de langages introduit une entité appelée coordinateur, qui gère dynamiquement les canaux entre les producteurs et consommateurs en exécution. Ainsi, le coordinateur orchestre les actions du système en réagissant à des événements déclenchés par les entités qui sont sous son contrôle. Des événements typiques sont des opérations d'entrée-sortie, ou l'atteinte d'un état particulier par l'entité. Les modèles les plus connus dans cette catégorie sont Manifold [Arbab *et al.*, 1993], ConCoord [Holzbacher, 1996] et RAPIDE [Shaw *et al.*, 1995].

1.5.2 La coordination orientée-données

Linda [Gelernter, 1985; Carriero *et al.*, 1986], le premier modèle qui a été proposé illustre le principe général des modèles de coordination orientée-données. Linda est un modèle et un langage de coordination, dans lequel des processus parallèles communiquent en échangeant des tuples via une abstraction d'une mémoire partagée appelée *tuplespace*. Un *tuplespace* est un espace partagé composé d'un multi-ensemble de tuples (la duplication de tuples est permise). Chaque tuple est une séquence d'une ou de plusieurs valeurs typées, et n'a de sens que pour le producteur et le consommateur du tuple. La communication dans Linda est dite *communication générative* : un processus génère un tuple et la durée de vie de celui-ci devient indépendante de celle du processus l'ayant créé. L'accès aux données du *tuplespace* est associatif i.e. par contenu. Comme le montre la figure 1.2, relative à l'une des première implémentations de Linda [Carriero *et al.*, 1986], la mémoire partagée consiste en une clique des mémoires locales des processus, mais elle est considérée comme une seule mémoire pour un processus donné. Linda est une alternative aux deux méthodes traditionnelles de programmation parallèle, viz. le passage de messages entre processus et l'usage d'une mémoire partagée.

La différence de Linda avec l'échange dyadique de messages est double. D'une part, lors d'un échange de message point à point, les processus doivent être synchronisés pour que l'émission d'un message par le premier processus corresponde à sa réception par le second. D'autre part,

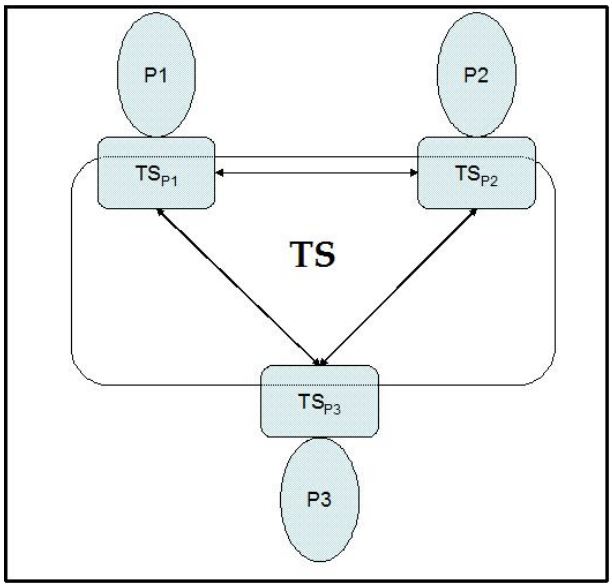

FIG. 1.2 – Modèle Linda

l'émetteur doit connaître l'emplacement physique du processus récepteur. Alors qu'avec un *tuplespace*, la communication est découplée dans le temps et dans l'espace. Le découplage dans le temps offre la possibilité aux processus de communiquer à travers le temps i.e., leurs temps d'exécution n'ont pas à se chevaucher (les processus n'ont pas à être synchronisés par un mécanisme de rendez-vous) afin d'établir une communication. Le découplage dans l'espace est relatif à la génération et au retrait anonymes de tuples i.e. les processus n'ont pas à connaître l'emplacement des autres processus pour communiquer, la communication a donc lieu indépendamment de l'emplacement des processus impliqués.

La différence entre l'usage des *tuplespaces* et l'utilisation d'une mémoire partagée est que la première est une mémoire associative i.e. l'accès à la mémoire se fait par contenu et non par adresse. Dans Linda, la récupération d'un tuple n'est pas nominative mais résulte d'un appariement avec un *template*, et le premier tuple satisfaisant le *template* est retourné, d'une manière non déterministe.

Le langage pionnier : Linda

Comme nous l'avons exposé plus haut, Linda est un modèle de coordination qui propose de relaxer la synchronisation entre deux processus, pour la remplacer par deux synchronisations entre chaque processus et le *tuplespace*. Le langage associé à ce modèle est composé d'un ensemble minimal de primitives, ce qui ne constitue pas un langage *stricto sensu*, mais une extension d'un langage de programmation. Aucune sémantique opérationnelle n'est associée au langage Linda original. Cependant, quelques langages qui étendent Linda (e.g. Klaim [De Nicola *et al.*, 1998]) en proposent une. La présentation de la sémantique de ces langages de coordination fait l'objet du chapitre 2.

Les primitives du langage

Le langage Linda offre trois primitives aux processus afin d'interagir avec le *tuplespace*.

- $out(t)$ permet à un processus d'insérer le tuple t dans le *tuplespace*,
- $in(template)$ permet de retirer le premier tuple satisfaisant le template. C'est une opération bloquante i.e. si aucun tuple ne satisfait le *template*, le processus ayant invoqué le processus

appelant est suspendu jusqu'à ce qu'un tuple approprié est créé par un autre processus,
– $rd(template)$ se comporte comme in, à la différence que rd lit une copie du premier tuple satisfaisant le *template*, mais ne le retire pas. C'est une opération bloquante également.

Un *template* est un tuple composé partiellement ou totalement de paramètres formels i.e. non instanciés. Il représente et est capable de s'apparier avec les tuples de même arité i.e. ayant le même nombre d'éléments que lui. Chaque élément du *template* est donc soit une valeur (élément instancié), soit un type de champ (élément formel ou libre). Par exemple, soit la séquence d'instructions $string\ s$; $float\ x$; $x \leftarrow 10.01$; Dans ce contexte, avec l'instruction $in(\langle s, x\rangle)$; le *template* en paramètre de in sera appareillé avec n'importe quel tuple composé de deux éléments, à condition que son premier élément soit de type chaîne de caractères et que son second élément soit un réel évalué à 10.01.

Lorsque deux processus sont suspendus avec deux opérations de in/rd et qu'un tuple satisfaisant leur *template* devient disponible, l'un des deux processus est choisi d'une manière non déterministe pour continuer son traitement. Lorsque plusieurs tuples sont appariables avec le *template* d'un in/rd, seul l'un des tuples est sélectionné.

Une primitive additionnelle est généralement implémentée, mais elle ne fait pas partie de Linda, c'est la primitive $eval$; $eval(t)$ permet d'évaluer des expressions contenues dans le tuple t avant de l'ajouter. Elle prend en paramètre un tuple contenant une ou plusieurs expressions, une expression étant un traitement à effectuer sur une valeur avant de la mettre dans le tuple à ajouter au *tuplespace*. Par exemple, $eval(\langle \text{“}carre\text{”}, sqrt(4), sqrt(9)\rangle)$ crée d'abord deux processus concurrents qui évalueront les deux racines carrés de 4 et de 9 avant de mettre le tuple $\langle \text{“}carre\text{”}, 2, 3\rangle$ dans le *tuplespace* pour qu'il soit disponible aux autres. Tant que ce calcul n'est pas effectué, aucun appariement (*pattern matching*) ne peut être effectué sur ce tuple.

Cependant, la primitive *eval* est désormais utilisée afin de lancer un nouveau processus (e.g dans Klaim [De Nicola *et al.*, 1998] : l'opération $eval(P)$ permet d'exécuter le processus P qui peut désormais interagir avec le *tuplespace*.

D'un point de vue applicatif, les primitives précitées sont des extensions pour des langages de programmation séquentielle, permettant à ces derniers de se transformer en des langages de programmation parallèle (e.g. C-Linda, Fortran-Linda, pour n'en citer que quelques-uns).

Exemple

Nous choisissons l'exemple traditionnel du dîner de philosophes, pour illustrer l'utilisation des primitives de Linda (il y a Num philosophes). Chaque processus exécute le code relatif à l'algorithme 1. Le programme est instancié par l'algorithme 2. Comme on le voit, les instructions de calcul (manger et réfléchir) sont mixées avec les instructions de coordination (in et out). Ce fait a été reproché à Linda par certains auteurs [Papadopoulos and Arbab, 1998], puisque le langage n'offre pas de séparation claire entre instructions de calcul et instructions de coordination.

Afin d'éviter une situation d'interblocage, des « tickets » sont utilisés. Leur nombre est égal au nombre de philosophes présents sur la table moins un, un philosophe ne pouvant manger que s'il dispose d'un ticket. Ce dernier est libéré par le philosophe quand il a fini de manger.

Limites de Linda

À l'origine, le modèle Linda était destiné à être utilisé dans des applications étroitement intégrées i.e. dans un environnement clos. En effet, un tuple n'a de sens que pour son émetteur et son récepteur, le partage d'un *tuplespace* entre des processus codés indépendamment n'est pas géré et le caractère bloquant des primitives in et rd n'est plausible que dans une configuration où le récepteur d'un tuple est sûr que son temps de suspension est borné.

Néanmoins, le découplage dans le temps et dans l'espace du modèle et la simplicité du

Algorithme 1 Un philosophe phil(i) avec Linda

```
int i ;
tant que vrai faire
  réfléchir() ;
  in(⟨ "ticket"⟩) ;
  in(⟨"fourchette″, i⟩) ;
  in(⟨"fourchette", (i+l)%Num⟩) ;
  manger() ;
  out(⟨"fourchette, i⟩) ;
  out(⟨"fourchette″, (i + i)%Num⟩) ;
  out(⟨"ticket"⟩) ;
fin tant que
```

Algorithme 2 Processus principal

```
int i ;
pour tout i = 0 à Num-1 faire
  out(⟨"fourchette", i⟩) ;
  eval(phil(i)) ;
  si i < (Num-1) alors
    out(⟨"ticket"⟩) ;
  fin si
fin pour
```

langage Linda font de lui un candidat de choix dans le développement d'applications ouvertes. Pour ce faire, le modèle initial a été remanié, tout en gardant ses principaux avantages : la communication anonyme et sans rendez-vous. Les principales lacunes de Linda, qui constituent autant de verrous pour son utilisation dans des applications ouvertes sont :

(i) la sécurité : un tuple disponible dans le *tuplespace* est potentiellement accessible à tous les processus connectés. En l'état, Linda est incapable de garantir la confidentialité de certaines données,

(ii) la mobilité : conjointement au problème de sécurité, le problème de mobilité. Un *tuplespace* est globalement accessible à tous. Aucun mécanisme n'est prévu pour le mouvement des processus d'un hôte à un autre, et les conséquences sur les données partagées,

(iii) la sémantique des primitives : les primitives initiales de Linda s'avèrent insuffisantes pour des scénarios d'exécution où un processus ne désire pas bloquer sur un échec de récupération d'un tuple, ou s'il veut exécuter une séquence d'actions atomiquement.

(iv) la réactivité du *tuplespace* : un *tuplespace* dans Linda est un espace passif, seuls les processus peuvent changer son état. On ne peut pas prévoir des réactions du *tuplespace* face à l'ajout ou le retrait de tuples, afin de garantir la cohérence des données par exemple.

Les paragraphes suivants reprennent chacune des limites de Linda et présente les solutions qui y ont été apportées dans la littérature.

Assurer la sécurité

Deux moyens ont été introduit pour assurer la sécurité dans un modèle de coordination orientée-données.

Multiple Spaces

Plusieurs raisons ont motivé l'introduction de plusieurs *tuplespaces* [Carriero *et al.*, 1995; De Nicola *et al.*, 1998] au lieu d'un seul. Les *multiple spaces* satisfont au besoin de sécurité des données en les plaçant dans un *tuplespace* particulier, restreignant ainsi leur visibilité, puisqu'une action de lecture depuis un *tuplespace* n'a accès qu'aux données qui y figurent. Ceci permet également d'utiliser les *tuplespaces* comme des entités de première classe (*first-class entities*) i.e. ils peuvent être produits, consommés, dupliqués, déplacés ou passés en paramètre.

***Law Governed Linda* (LGI)**

Avec l'utilisation de *multiple spaces*, la sécurité est forcée par la limitation du partage de données. Cependant, dans [Minsky *et al.*, 2001], les auteurs observent que « isoler la communication dans des *tuplespaces* multiples augmente la sûreté seulement dans la mesure où cela élimine le partage, mais [que] les *tuplespaces* sont plus utiles quand plusieurs agents partagent l'accès à un seul *tuplespace* »[10]. Ainsi, Minsky *et al.* arguent que, comme la communication dans Linda se fait par contenu (accès associatif), la sécurité doit aussi être assurée par contenu. Ils présentent les lois, un mécanisme qui utilise le contenu des tuples afin de savoir si oui ou non, l'opération peut réussir. Ainsi, la sécurité est satisfaite tout en tirant un profit maximal du partage des données.

Introduire la mobilité

Le regain d'intérêt pour les modèles et langages de coordination à mémoire associative a en partie été stimulé par la popularité des applications mobiles. Quelques travaux ont repris le modèle Linda pour en faire un langage pour agents mobiles. Nous en présentons deux : Lime et Klaim.

Lime[11]

Lime [Picco *et al.*, 1999; Picco and Buschini, 2002] propose d'étendre Linda aux environnements mobiles. Les auteurs y introduisent la notion de *tuplespaces* provisoirement partagés. Un *tuplespace* provisoirement partagé est un *tuplespace* contenant l'union des *tuplespaces* locaux de chaque agent connecté. Dès qu'un agent est déconnecté, le *tuplespace* partagé est recalculé en prenant en compte l'absence des tuples que possède l'agent. Réciproquement, quand un agent se connecte à un hôte, son espace de tuples est fusionné (uni) avec ceux des autres agents mobiles présents sur la machine. Tous ces calculs des nouveaux états des *tuplespaces* sont faits d'une manière transparente à l'utilisateur (c.f figure 1.3).

Klaim[12]

La deuxième proposition pour étendre Linda aux environnements mobiles est Klaim [De Nicola *et al.*, 1998]. Ses auteurs proposent d'introduire les localités dans l'expression des primitives de langage. Cela signifie qu'un processus peut ajouter, lire ou prendre un tuple depuis n'importe quel *tuplespace* du moment qu'il détient son identifiant, et ce en préfixant l'identifiant de la localité par @. Mais ce qui permet la mobilité des processus, c'est l'usage des localités avec la primitive *eval* qui, dans Klaim, peut prendre un processus en paramètre. Ainsi, nous pouvons faire migrer un processus d'un *tuplespace* vers un autre. Les auteurs font une distinction entre mobilité physique et mobilité logique. Pour ce faire, dans la définition du langage, ils introduisent deux variables L (pour localités, des emplacements logiques, manipulés par les processus) et S (pour sites, des emplacements physiques, non accessibles aux processus), ainsi qu'une application associant une localité logique à une localité physique. Les auteurs étendent

[10] *Segregating communication into multiple tuplespaces increases safety only insofar as it eliminates sharing, but [that] tuplespaces are most useful when diverse agents share access to a single tuplespace*

[11] *LInda in a Mobile Environment*

[12] *Kernel Language for Agent Interaction and Mobility*

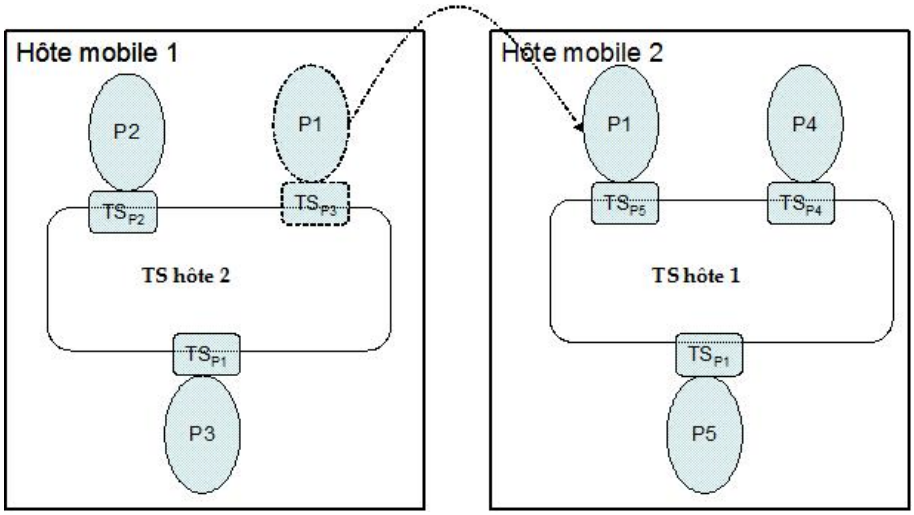

FIG. 1.3 – Principe de Lime

le modèle avec un mécanisme de typage permettant la gestion des droits d'accès des processus aux différents *tuplespaces*. Des librairies Java sont associées à Klaim. Baptisées Klava, le langage (Java + librairies) permet le déploiement d'une application Java au dessus d'un *tuplespace* avec les classes nécessaires à la création d'un *tuplespace*, des tuples, et les méthodes correspondant aux primitives du langage.

Ci-après un exemple type de code écrit dans Klaim exprimant un appel de procédure à distance (RPC[13]). Un processus *caller* envoie une requête à un autre processus distant *callee*, ensuite il attend sa réponse. La requête (un tuple) contient, outre le nom et les paramètres de la procédure invoquée, la localité privée de *caller* où la réponse doit être envoyée. La primitive $newloc(u)$ crée une nouvelle localité logique qui a pour identifiant u.

$$caller = newloc(u).out(procid, e_1, \ldots, e_n, u)@callee.in(!y_1, \ldots, !y_k)@u.0$$

De l'autre côté, le processus *callee* attend constamment une requête. Dès que celle-ci est reçue, *callee* exécute la procédure demandée et envoie son résultat vers la localité spécifiée dans le tuple reçu de la part de *caller*.

$$callee = in(!pid, !x_1, \ldots, !x_n, !u)@self.(callee \mid \langle pid(x_1, \ldots, x_n)\rangle.out(r_1, \ldots, r_k)@u.0)$$

La barre verticale | est l'opérateur de composition parallèle, i.e. *callee* est prêt immédiatement à recevoir de nouvelles requêtes, il n'a donc pas à attendre l'exécution de la procédure *pid* avant d'être à nouveau disponible. Le caractère 0 représente un processus terminé.

Étendre les primitives de Linda

L'extension des primitives de Linda concerne plusieurs points. D'abord, elle concerne la possibilité qu'a un processus d'exprimer des opérations de transactions, comme pour les systèmes de gestion de bases de données, i.e. des opérations possédant les quatre propriétés : atomicité, consistance, isolation et durabilité (ACID). JavaSpaces [Freeman *et al.*, 1999], un produit de Sun Microsystem™, permet de démarrer une transaction avant de lancer l'exécution de ses opérations.

[13] *Remote Procedure Call*

Chaque fois qu'une transaction est lancée, un « gestionnaire de transactions » est désigné comme responsable pour la préservation des propriétés ACID.

Ensuite, l'extension des primitives concerne l'utilisation d'une vision globale du *tuplespace* lorsqu'il exprime son besoin en interaction avec un *rd* ou un *in*. En effet, les *templates* expriment des conditions devant être satisfaites par un tuple particulier, alors qu'un processus pourrait désirer conditionner son exécution par l'absence d'un certain tuple par exemple. Pour ce faire, il a besoin d'avoir un instantané de l'état global du *tuplespace*. Dans [Busi *et al.*, 2000b] les auteurs proposent d'introduire un branchement conditionnel gardé par l'absence d'un tuple dans le *tuplespace*. L'approche du système proposé par IBM™TSpaces [Wyckoff *et al.*, 1998] est d'étendre l'ensemble des primitives du langage, en ajoutant des opérations qui ajoutent et/ou consomment des multi-ensembles de données. Par exemple, La primitive *multiwrite* ajoute l'ensemble de tuples existant dans un tableau passé en paramètres, la primitive *collect* récupère tous les tuples satisfaisant un *template* et la primitive *copy-collect* lit tous les tuples satisfaisant un *template*.

Enfin, l'extension des primitives concerne la relaxation de la contrainte de blocage d'un processus sur une action de lecture sur le *tuplespace*, en introduisant un *in* et un *rd* non bloquant (*inp* et *rdp*) [Busi *et al.*, 1998]. Ces derniers retournent vrai ou faux, mais ne bloquent pas en attendant un tuple qui s'apparie avec le *template*.

Programmer le *tuplespace*

Afin d'ajouter plus de réactivité au *tuplespace*des travaux proposent d'étendre le système par des règles de comportement du *tuplespace*. Les modèles qui permettent au système d'avoir un comportement événementiel sont souvent appelés modèles de coordination « hybrides » parce qu'ils allient l'utilisation d'un espace de données partagé avec un comportement réactif tel que dans les modèles de coordination orientée-contrôle. Le système Tucson [Omicini and Zambonelli, 1999] et Mars [Cabri *et al.*, 1999] proposent de permettre d'ajouter des expressions sous la forme événement-condition-action au *tuplespace* pour en faire un système de gestion des interactions plutôt qu'une structure de données passive. Le système LGI [Minsky *et al.*, 2001] présenté plus haut est déjà un exemple de *tuplespace* programmable. Les événements concernent les actions des autres agents, i.e. l'exécution d'une opération de coordination. Cependant, il est connu que ce type de comportement est potentiellement générateur de non-terminaison, tel que l'ont déjà expérimentés les chercheurs dans le domaine des bases de données actives [Paton and Diaz, 1999]. En effet, si la réaction d'une règle tombe elle-même dans une autre condition de déclenchement d'une autre règle, nous pouvons avoir une chaîne infinie d'actions-réactions. Le problème est que la terminaison de l'exécution d'une règle ne peut être vérifiée en *off-line*, puisqu'elle dépend de l'état du *tuplespace*, qui est par définition dynamique. Dans LGI, il n'existe pas de moyen automatique afin de garantir la terminaison du déclenchement des règles. C'est la responsabilité du concepteur de la règle d'assurer la terminaison, en un temps raisonnable. Si tel n'est pas le cas, le contrôleur définit un temps limite pour l'évaluation d'un événement. S'il est dépassé, la terminaison de l'évaluation est forcée sans aucune réaction [Minsky, 2004, p.18]. Mars résout le problème de récursion infinie en définissant deux niveaux dans le *tuplespace*, et en interdisant le niveau de base d'avoir une réaction. Enfin, la termination n'est pas abordée dans les articles traitant de Tucson.

1.6 Conclusion

Dans ce chapitre, nous avons présenté la problématique de la coordination dans le domaine des SMA et des langages de coordination. Nous avons porté une attention particulière aux modèles de coordination orientée-données dans le cadre desquels s'inscrit notre proposition. En effet, nous proposons un modèle de coordination orientée-donné pour l'implémentation de SMA ouverts (chapitre 4). La coordination dans le sens des SMA est traitée dans le chapitre 6 où nous proposons un SMA pour la résolution d'un problème particulier : le problème du transport à la demande.

Nous avons axé notre présentation des modèles et langages de coordination sur les problèmes conceptuels et les cadres d'application des différents modèles présentés. Néanmoins, lors de l'implémentation d'un système, des ambiguïtés demeurent. En effet, afin de s'assurer qu'un système particulier adhère effectivement à un modèle de coordination, il est nécessaire d'en avoir une représentation formelle. Les modèles et langages de coordination que nous avons présentés n'ont pas tous une sémantique opérationnelle associée, la majorité étant présentés d'une manière informelle. Nous n'avons donc pas de spécification claire qui guiderait une éventuelle implémentation de ces langages. Le chapitre suivant présente ces travaux, qui sont effectués dans le domaine de la spécification formelle des langages de coordination. Ils se fondent sur la tradition des algèbres de processus, e.g. CCS [Milner, 1989]. Nous nous inspirons de ces approches dans le chapitre 5 afin de formaliser un langage de coordination correspondant à notre modèle de coordination.

Chapitre 2

Algèbres de processus pour la coordination

Sommaire

2.1 Introduction

Un effort considérable a été engagé depuis les années 1970 afin de définir des modèles formels pour la spécification de systèmes. Cet effort a également été conduit dans le cadre strict des modèles de coordination. En effet, il est important d'associer un langage de coordination doté d'une sémantique opérationnelle à un modèle de coordination. Ceci permet de s'assurer qu'un système écrit dans ce langage adhère bien au modèle de coordination présenté. De plus, ceci permet de raisonner sur ces systèmes, les « vérifier » i.e. établir qu'un système satisfait à certaines propriétés. Dans ce chapitre, nous donnons la définition formelle des principales propositions dans le cadre des langages de coordination orientée-données. Nous reprenons le cadre formel proposé dans [Busi *et al.*, 2001], qui a l'intérêt d'avoir une présentation uniforme des différentes propositions. Dans la section 2.2, nous introduisons les algèbres de processus. Dans la section 2.3, nous donnons une algèbre du langage Linda original. Ensuite, nous étendons le langages avec des primitives additionnelles (section 2.4), des *tuplespaces* multiples (section 2.5), nous transformant le *tuplespace* en un espace de données réactifs (section 2.6) avant d'introduire des constructeurs temporels (section 2.7). Dans la section 2.8, nous résumons les principaux résultats de l'introduction des algèbres de processus dans les modèles de coordination orientée-données.

2.2 Introduction aux algèbres de processus

Le terme processus se réfère au *comportement* d'un *système*. Un comportement est l'ensemble des événements ou des actions qu'un système peut exécuter, leur ordre d'exécution et différents autres aspects tels que le temps associé. Généralement, les actions sont considérées comme étant discrètes : leur occurrence survient à un instant donné, et différentes actions sont séparées dans le temps. C'est la raison pour laquelle un processus est quelquefois appelé un « système à événements discrets » [Baeten, 2005]. Le terme algèbre quant à lui dénote que nous suivons une approche algébrique, i.e. utilisant des méthodes de l'algèbre universelle, pour traiter des comportements. Le terme algèbre de processus dénote donc l'utilisation d'un approche algébrique dans l'étude du comportement d'un système. Actuellement, il se réfère le plus souvent à l'étude algébrique du comportement de systèmes parallèles/distribués. Afin de faire face à la complexité de la conception de systèmes parallèles/distribués, les développements théoriques relatifs à une algèbre de processus consistent généralement en quatre étapes [Lee *et al.*, 1994]. Premièrement, définir un ensemble d'opérateurs et de règles syntaxiques pour construire des processus. Un processus est défini en termes d'un ensemble d'opérateurs basiques, comme la composition séquentielle, le choix et la composition parallèle. Deuxièmement, associer à chaque opérateur un ensemble de règles sémantiques qui affectent à un processus une interprétation comportementale. La sémantique associée est en général une sémantique opérationnelle structurée dans le style SOS[14] [Plotkin, 1981]. Un exemple de règle est celui spécifiant une composition parallèle :

$$\frac{P \xrightarrow{l} P'}{P \| Q \xrightarrow{l} P' \| Q}$$

La règle spécifie que, s'il est possible pour un processus P d'exécuter une action l et se transformer en P', alors il est également possible pour la composition parallèle de P et Q d'exécuter l'action l et de se transformer en une composition parallèle entre P' et Q. Troisièmement, afin de comparer différents processus, introduire quelques relations d'équivalence et de congruence qui expriment que deux processus ont un comportement similaire. Généralement, les processus sont considérés par rapport à un certain type de comportement observable. Deux processus sont considérés équivalents s'ils exhibent le même comportement observable. Les équivalences peuvent être prouvées formellement en utilisant des systèmes de réécriture, qui consistent en des axiomes et règles d'inférence sous la forme d'équivalences algébriques. Un exemple d'équivalence algébrique est :

$$P \| Q \equiv Q \| P$$

Enfin, développer des algorithmes qui décident de l'équivalence de deux processus. Les principales approches algébriques pour la concurrence sont CCS [Milner, 1989], CSP [Hoare, 1985] et ACP [Bergstra and Klop, 1984]. Différentes extensions ont été depuis introduites, telles que l'introduction du temps, le passage de canal de communication en paramètres, etc.

2.3 Une algèbre de processus pour Linda

La première étape est de proposer une algèbre basique pour le modèle Linda original. Cette dernière est enrichie au fur et à mesure suivant les approches présentées dans le chapitre 1. Un agent est décrit en tant que terme d'une algèbre où les actions sont l'une des primitives du langages, viz. *out*, *rd*, *in* ou *eval*. Un ensemble de messages est considéré, appelé *Data*, que

[14] *Structured Operational Semantics*

$$(out(a).A \oplus Ag, DS) \rightarrow (A \oplus Ag, DS \oplus a)$$

$$(in(a).A \oplus Ag, DS \oplus a) \rightarrow (A \oplus Ag, DS)$$

$$(rd(a).A \oplus Ag, DS \oplus a) \rightarrow (A \oplus Ag, DS \oplus a)$$

$$(eval(A').A \oplus Ag, DS) \rightarrow (A' \oplus A \oplus Ag, DS)$$

$$\frac{(A_j, DS) \rightarrow (A', DS')}{(\sum_{i \in I} A_i \oplus Ag, DS) \rightarrow (A' \oplus Ag, DS')} \text{ si } j \in I$$

$$\frac{(A, DS) \rightarrow (A', DS')}{(K \oplus Ag, DS) \rightarrow (A' \oplus Ag, DS')} \text{ si } K = A$$

TAB. 2.1 – Les règles de transition de Linda

nous parcourons avec $a, b, \ldots$. L'ensemble *Agent* d'expression d'agents, que nous dénotons par $A, A', \ldots$ est l'ensemble des termes généré par la grammaire suivante :

$$A ::= 0 \mid \mu.A \mid \sum_{i \in I} A_i \mid K$$
$$\mu ::= out(a) \mid in(a) \mid rd(a) \mid eval(A)$$

μ est l'une des quatre primitives de coordination, et K un élément de $Name$ de noms d'agents. Nous supposons que toutes les occurrences des noms d'agent sont associés à une définition correspondante sous la forme $K = A$. Les noms des agents sont utilisés pour supporter une définition récursive. Par exemple, $Ren_{ab} = in(a).out(b).Ren_{ab}$ représente un agent qui renomme les messages du type a en des messages du type b.

L'agent 0 est un agent qui fait rien. L'agent $\mu.A$ est un agent qui effectue μ et se comporte ensuite comme A. L'agent $\sum_{i \in I} A_i$ est un agent qui peut se comporter comme n'importe lequel des agents A_i, définissant une composition alternative. Notons qu'il n'existe pas d'opérateur de composition parallèle. Comme nous le verrons, cet opérateur est implicite, car nous considérons toujours un multi-ensemble d'agents agissant sur le *tuplespace*, qui s'exécutent en parallèle sans synchronisation.

La configuration d'un système est composée d'un multi-ensemble d'agents et d'un multi-ensemble de messages disponibles. Une configuration d'un système est un couple $(Ag, DS) \in \mathcal{M}(Agent) \times \mathcal{M}(Data)$, où $\mathcal{M}(ensemble)$ est utilisé pour dénoter l'ensemble de tous les multi-ensembles d'éléments pris dans *ensemble*. Un multi-ensemble est généralement représenté comme un ensemble i.e. avec des accolades (omises dans le cas d'un singleton). L'union de multi-ensembles est notée $\oplus$. Par abus de notation, $\oplus$ est également utilisé pour une union d'ensembles également.

Une configuration d'un système évolue suivant l'exécution d'opérations de coordination. Les règles sont reportées dans la table 2.1. L'exécution d'un $out(a)$ a l'effet d'ajouter le tuple a au *tuplespace*. Un $in(a)$ retire a du *tuplespace*, s'il existe, sinon l'opération est bloquée. Une action $rd(a)$ se comporte pareillement mais laisse a dans le *tuplespace*. L'exécution de $eval(A')$ a pour

effet de rajouter A' au multi-ensemble d'agents présents dans le système. Le choix entre plusieurs A_i est résolu en choisissant l'un deux pouvant exécuter son action. Enfin, Un agent K peut faire ce que peut faire l'agent A qu'il définit.

Il existe deux sémantiques alternatives pour le *out*. En effet, dans [Busi *et al.*, 2000a], les auteurs proposent trois sémantiques différentes pour *out* : instantanée, ordonnée ou non-ordonnée. Dans la sémantique instantanée, *(out(a).A,DS)* et *(A,DS* $\oplus$ *a)* sont équivalents, i.e. a est déjà dans le *tuplespace*. La sémantique ordonnée est celle qui est présentée ici : en un seul pas, l'exécution de l'opération et l'addition du tuple sont effectuées. Le terme ordonnée se réfère au fait que l'ordre d'émission est cohérent avec l'ordre d'arrivée des tuples. Dans la sémantique non-ordonnée, l'émission du tuple et son arrivée au *tuplespace* sont deux étapes distinctes. Ainsi, un tuple peut être émis, et arriver avec un retard non prévisible. Ceci est effectué simplement en remplaçant la règle du *out* par les deux règles qui suivent :

$$(out(a).A \oplus Ag, DS) \rightarrow (\langle a \rangle \oplus A \oplus Ag, DS)$$

$$(\langle a \rangle \oplus Ag, DS) \rightarrow (Ag, DS \oplus a)$$

où $\langle a \rangle$ dénote un tuple émis mais qui n'a pas encore atteint le *tuplespace*.

2.4 Extension des primitives du langage

Il est très simple d'étendre les primitives du langage basique Linda, en introduisant des opérations de transaction. En effet, considérons une opération $rew(m_1, m_2)$ qui remplace un multi-ensemble de données m_1 par un multi-ensemble m_2. Il suffit d'ajouter la primitive à la grammaire des agents :

$$\mu ::= \cdots \mid rew(m_1, m_2)$$

La règle correspondante est :

$$(rew(m_1, m_2).A \oplus Ag, DS \oplus m_1) \rightarrow (A \oplus Ag, DS \oplus m_2)$$

Il est possible également d'introduire des opérations globales, i.e. qui nécessitent une vision globale du *tuplespace* afin d'être exécutées. Par exemple, une primitive de test d'absence vérifie qu'il n'y a pas de données satisfaisant un *template* dans le *tuplespace*. L'introduction de cette primitive (*tfa* pour *test-for-absence*) se fait comme suit :

$$\mu ::= \cdots \mid tfa(a)$$

et la sémantique associée est :

$$(tfa(a).A \oplus Ag, DS) \rightarrow (A \oplus Ag, DS) \text{ si } a \notin DS$$

La nécessité d'une vision globale du *tuplespace* est reflétée par la condition de bord vérifiant que a n'existe pas dans le multi-ensemble DS. Plusieurs propositions d'extension des primitives de Linda considèrent également des primitives qui ne sont pas bloquantes. Les versions non bloquantes pour *in* et *rd* sont respectivement *inp* et *rdp*. Elles sont non-bloquantes car, si le tuple recherché n'est pas trouvé, ces primitives se terminent avec un échec. Dans [Busi *et al.*, 1998], les auteurs introduisent les primitives $inp(a)?A\_B$ et $rdp(a)?A\_B$, qui consistent en un branchement conditionnel entre A et B gardé par la présence de a. Si a est présent, alors A est

exécuté, sinon B est exécuté. Ces mêmes primitives peuvent être écrites avec l'opérateur tfa, comme suit :

$$inp(a)?A\_B = in(a) + tfa(a).B \qquad rdp(a)?A\_B = rd(a) + tfa(a).B$$

D'autres extensions englobent des opérations globales de transactions. Ce genre de primitives est celui qui effectue un ensemble d'actions atomiques (transaction) et qui nécessite une vue globale du *tuplespace*. Par exemple, les primitives *collect* et *copy_collect* qui prennent ou lisent un ensemble de tuples satisfaisant un *template*. Il s'agit d'une solution au problème des $rd - multiples$ de Linda [Rowstron and Wood, 1998]. Ce problème consiste en la récupération d'un même tuple par deux rd différents d'un même agent. Dans Linda, il n'y a aucun moyen de pouvoir différencier la lecture de deux tuples distincts de la double-lecture du même tuple. Ainsi, la grammaire des actions de coordination de Linda peut être étendue par :

$$\mu ::= \cdots \mid collect(a)$$

et la sémantique associée est :

$$(collect(a).A \oplus Ag, DS) \rightarrow (A \oplus Ag, \{b \in DS \mid b \neq a\})$$

2.5 Multiplier les *tuplespaces*

Avec les *tuplespaces* multiples, chaque *tuplespace* est identifié par un nom unique. Soit $Space$ un ensemble dénombrable de noms, que nous dénotons par $r, s, \ldots$. Une configuration d'un système devient un sous-ensemble de $Conf = Space \times \mathcal{M}(Agent) \times \mathcal{M}(Data)$ avec comme élément typique $s.(Ag, DS)$, il représente un *tuplespace* ayant pour nom s qui contient les données de DS et les agents de Ag. Nous parcourons les noms de *tuplespaces* par $C, D, \ldots$. La syntaxe de la formulation basique est étendue avec les nouvelles versions des primitives de coordination ayant maintenant un *tuplespace* associé. L'idée est que la primitive considérée est exécutée au niveau du *tuplespace* spécifié, quand le nom est omis, la primitive est exécutée dans le *tuplespace* depuis lequel l'opération a été exécuté (dans Klaim [De Nicola *et al.*, 1998], ceci est effectué en spécifiant explicitement la localité spécifique self).

$$\mu ::= \cdots \mid out(a)@s \mid in(a)@s \mid rd(a)@s \mid eval(A)@s \mid newloc(s)$$

$newloc(s)$ est une primitive qui crée dynamiquement un nouveau *tuplespace* avec pour nom unique s. Le nom s est une variable libre, la valeur qui lui est associée (le nom) est choisie durant l'exécution de telle sorte que l'unicité des éléments de $Space$ est préservée. La sémantique du nouveau langage est donnée dans la table 2.2. $A[u/s]$ désigne l'agent A ou chaque occurrence de s est remplacée par u. Dans [Busi *et al.*, 2001], les auteurs observent que la vraie nouveauté des *tuplespaces* multiples réside dans l'introduction de la primitive qui permet la création dynamique de *tuplespaces* (viz. *newloc*). Sans cette primitive, il existe un encodage du langage avec *tuplespaces* multiples en utilisant seulement les primitives de Linda et en remplaçant $in(a)@s$ par $in(\langle a, s\rangle)$. Une autre alternative utilisant des *tuplespaces* multiples est celle utilisée dans e.g. Bauhaus Linda [Carriero *et al.*, 1995]. Dans de telles approches, les *tuplespaces* sont organisés hiérarchiquement. Dynamiquement, un *tuplespace* peut être transféré dans un autre. Les espaces frères (dans la hiérarchie) peuvent interagir en déposant et en retirant un tuple de leur espace père commun. Les primitives du langage basique peuvent être étendues avec le signe $\uparrow$ signifiant que la primitive doit s'exécuter, non pas dans son espace, mais dans l'espace père (e.g. $eval(out(a)\uparrow)\uparrow$).

$$r.(out(a)@s.A \oplus Ag, DS) \oplus s.(Ag', DS') \oplus C \rightarrow r.(A \oplus Ag, DS) \oplus s.(Ag', DS' \oplus a) \oplus C$$

$$r.(in(a)@s.A \oplus Ag, DS) \oplus s.(Ag', DS' \oplus a) \oplus C \rightarrow r.(A \oplus Ag, DS) \oplus s.(Ag', DS') \oplus C$$

$$r.(rd(a)@s.A \oplus Ag, DS) \oplus s.(Ag', DS' \oplus a) \oplus C \rightarrow r.(A \oplus Ag, DS) \oplus s.(Ag', DS' \oplus a) \oplus C$$

$$r.(eval(A')@s.A \oplus Ag, DS) \oplus s.(Ag', DS') \oplus C \rightarrow r.(A \oplus Ag, DS) \oplus s.(A' \oplus Ag', DS') \oplus C$$

$$r.(newloc(s).A \oplus Ag, DS) \oplus C \rightarrow r.(A[u/s] \oplus Ag, DS) \oplus u.(\varnothing, \varnothing) \oplus C \quad u \text{ est un nouveau nom}$$

$$\frac{r.(\mu.A \oplus Ag, DS) \oplus C \rightarrow r.(Ag', DS') \oplus C}{r.(\mu@r.A \oplus Ag, DS) \oplus C \rightarrow r.(Ag', DS') \oplus C}$$

$$\frac{r.(A_j, DS) \oplus C \rightarrow r.(A', DS') \oplus C'}{r.(\sum_{i \in I} A_i \oplus Ag, DS) \oplus C \rightarrow r.(A' \oplus Ag, DS') \oplus C'} \text{ si } j \in I$$

$$\frac{r.(A, DS) \oplus C \rightarrow r.(A', DS') \oplus C'}{r.(K \oplus Ag, DS) \oplus C \rightarrow r.(A' \oplus Ag, DS') \oplus C'} \text{ si } K = A$$

TAB. 2.2 – Les règles de transition avec des *tuplespaces* multiples

2.6 *Tuplespaces* réactifs

Les *tuplespaces* réactifs incluent les propositions de Tucson [Omicini and Zambonelli, 1999], Mars [Cabri *et al.*, 1999] et les langages introduisant des mécanismes de notification, tels que JavaSpaces [Freeman *et al.*, 1999], TSpaces [Wyckoff *et al.*, 1998] et leur contrepartie formelle donnée dans [Busi and Zavattaro, 1999]. L'introduction d'un mécanisme de notification se fait comme suit :

$$\mu ::= \cdots \mid notify(a, A)$$

qui spécifie que le processus A sera exécuté à chaque fois qu'un tuple a est présent dans le *tuplespace*. D'une manière générale, la réactivité des *tuplespaces* a pour principal objectif d'ajouter des capacités de contrôle d'accès au *tuplespace*. Par exemple, un espace peut être programmé de telle manière que des agents inconnus ne peuvent qu'ajouter des tuples, mais pas en retirer ni en lire. Dans [Busi *et al.*, 2001], les auteurs proposent un formalisme qui permet d'exprimer d'une manière générale cette famille de langages. Il s'agit d'associer à chaque primitive de coordination une règle de comportement. Plus précisément, ils définissent Op comme l'ensemble de tous les préfixes μ qu'ils utilisent comme identifiants aux règles. Soit $\mathcal{R} = Op \times \mathcal{M}(Data) \times \mathcal{M}(Agent) \times \mathcal{M}(Data)$ l'ensemble des règles possibles (parcourues avec R). Les éléments de $\mathcal{R}$ sont dénotés $\mu : DS \triangleright [Ag', DS']$ où μ est l'identifiant de la règle, DS la partie du *tuplespace* qui est consommée, Ag' l'ensemble des agents créés et DS' la partie du *tuplespace* qui est créée. Pour chaque opération, il existe une et une seule règle. Par exemple, les règles relatives aux primitives du langage reportées dans la table 2.1 peuvent être décrites comme suit :

$$out(a) : \varnothing \triangleright [\varnothing, a]$$
$$in(a) : a \triangleright [\varnothing, \varnothing]$$
$$rd(a) : a \triangleright [\varnothing, a]$$
$$eval(A) : \varnothing \triangleright [A, \varnothing]$$

Comme les règles peuvent être changées durant l'exécution, la description d'une configuration doit être augmentée avec les règles du *tuplespace*. Formellement, la configuration d'un système devient un triplet $(Ag, DS, R) \in \mathcal{M}(Agent) \times \mathcal{M}(Data) \times \mathcal{P}(\mathcal{R})$, où $\mathcal{P}(\mathcal{R})$ dénote l'ensemble de tous les ensembles possibles de règles. La règle pour l'exécution d'un μ donné devient :

$$\frac{(\mu : DS \triangleright [Ag', DS'])}{(\mu.A \oplus Ag, DS \oplus DS_c, R) \rightarrow (A \oplus Ag' \oplus Ag_c, DS' \oplus DS_c, R)}$$

L'ajout d'une nouvelle règle est simple, une nouvelle primitive $newRule$ est définie, responsable de la modification de l'ensemble des règles présentes :

$$\mu ::= \cdots \mid newRule(\mu : DS \triangleright [Ag', DS'])$$

dont la sémantique est :

$$(newRule(\mu : DS \triangleright [Ag', DS']).A \oplus Ag_c, DS_c, (\mu : DS_a \triangleright [Ag'_o, DS'_o]) \oplus R)$$
$$\rightarrow (A \oplus Ag_c, DS_c, (\mu : DS \triangleright [Ag', DS']) \oplus R)$$

Il est intéressant de noter que la sémantique de la primitive $notify$ présentée plus haut peut être écrite dans cette formulation, comme suit :

$$(notify(a, A').A \oplus Ag_c, DS_c, (out(a) : DS \triangleright [Ag', DS']) \oplus R)$$
$$\rightarrow (A \oplus Ag_c, DS_c, (out(a) : DS \triangleright [A' \oplus Ag', DS']) \oplus R)$$

2.7 Introduction du temps

JavaSpaces [Freeman *et al.*, 1999] et TSpaces [Wyckoff *et al.*, 1998] ont des constructeurs temporels. Cependant, comme aucune sémantique n'est associée au langage, aucune description formelle du comportement opérations temporisées n'a été proposée. Des travaux académiques tels que [Linden *et al.*, 2006; Busi and Zavattaro, 2003] ont proposé de formaliser l'extension du langage Linda original avec des constructeurs temporels. Deux approches typiques existent afin d'introduire le temps [Busi and Zavattaro, 2003]. Dans la première, nous supposons que les actions de coordination ne consomment pas de temps. De plus, nous considérons le temps comme une action à part, passée d'une manière synchrone sur tous les tuples du *tuplespace* et tous les processus ; il s'agit du « fonctionnement à deux phases »[15]. Cette approche a été utilisée avec succès dans le cadre des systèmes réactifs.

La deuxième alternative est de supposer que les actions de coordination peuvent consommer du temps, et il n'y a pas de synchronisation globale sur le passage du temps ; il s'agit de « l'approche paresseuse »[16]. Dans [Busi and Zavattaro, 2003], les auteurs plaident en faveur de l'utilisation de l'approche paresseuse dans le contexte de Linda, puisqu'elle est plus appropriée aux systèmes distribuées. En effet, dans un système distribué, il n'existe pas d'horloge globale sur laquelle les processus du système se synchronisent. De plus, supposant qu'un agent génère un premier tuple avec $out(a, \delta t)$ spécifiant que a a une durée de vie de δt, suivi d'un $in(a)$. Si aucun autre agent ne veut retirer ce tuple, l'agent ayant exécuté *out* devrait intuitivement récupérer le tuple a avec *in*. Néanmoins, la primitive *out* pouvant avoir été exécutée depuis un *tuplespace* distant, le temps d'arriver au *tuplespace* destination, il se pourrait que δt soit écoulé et que donc, *in* devrait rester bloqué. Ce scénario n'est pas considéré par l'approche à deux phases.

Dans le cadre du fonctionnement à deux phases, trois alternatives sont possibles afin d'introduire le temps :

1. Soit le temps est introduit sous forme de retards, stipulant qu'une action de coordination doit être exécutée après une période de temps,
2. soit le temps est introduit en spécifiant que les tuples dans le *tuplespace* sont valides pendant une période de temps ; D'une manière similaire, les requêtes pour un tuple ne peuvent être retardées indéfiniment,
3. soit le temps est introduit sous forme d'intervalles de temps durant lesquelles les actions doivent être exécutées.

Par exemple, pour introduire les retards dans l'exécution des primitives, une nouvelle primitive est introduite :

$$\mu ::= \cdots \mid delay(d)$$

$delay(d).A$ fait passer d unités de temps avant de se comporter comme A. Un nouveau type de transition, relatif au passage du temps est introduit (dénoté $\rightsquigarrow$). La sémantique associée est composée de deux règles :

$$\frac{(Ag, DS) \not\rightarrow}{(Ag, DS) \rightsquigarrow (Ag^{-}, DS)}$$

$$(delay(0).A \oplus Ag, DS) \rightarrow (A \oplus Ag, DS)$$

Où $\not\rightarrow$ exprime le fait qu'aucune transition non temporelle est possible ; avec Ag^{-} décrivant les processus A de Ag auxquels on applique les règles suivantes :

$$\text{Si } A = \mu.A' \text{ et } \mu \in \{out(a), in(a), rd(a), eval(A), delay(0)\} \text{ alors } A^{-} = A$$

[15] *two-phase functioning*
[16] *lazy approach*

$$\text{Si } A = delay(d).A' \text{ et } d > 0 \text{ alors } A^- = delay(d-1).A'$$

$$\text{Si } A = \sum_{i \in I} A_i \text{ alors } A^- = \sum_{i \in I} A_i^-$$

Dans l'approche paresseuse, $out(a)$ est remplacée par $out(a, \delta t)$:

$$\mu ::= \cdots \mid out(a, \delta t)$$

$TimedData$ est introduit comme $\{a_t \mid a \in Data, t \in \mathbb{N}\}$. Le temps courant t est ajouté à la configuration, et les règles relatives aux primitives du langage deviennent (Δ_μ dénote le temps nécessaire à μ afin de s'exécuter) :

$$(out(a, \delta t').A \oplus Ag, DS, t) \rightarrow (A \oplus Ag, DS \oplus a_{t+\delta t+\delta t'}, t + \delta t) \quad \text{avec } \delta_t \in \Delta_{out}$$

$$(in(a).A \oplus Ag, DS \oplus a, t) \rightarrow (A \oplus Ag, DS, t + \delta t) \quad \text{avec } \delta t \in \Delta_{in}$$

$$(rd(a).A \oplus Ag, DS \oplus a, t) \rightarrow (A \oplus Ag, DS \oplus a, t + \delta t) \quad \text{avec } \delta t \in \Delta_{rd}$$

2.8 Principaux résultats

La spécification formelle d'un langage permet d'assurer que son implémentation sur différentes architecture résulte en des systèmes équivalents. Mais aussi, avoir une sémantique opérationnelle permet de vérifier certaines propriétés intéressantes. Certains travaux traitent de la composition sûre de composantes évoluant dans Linda [Roldan *et al.*, 2003]. Mais les travaux les plus abondants ont été effectués dans le domaine de l'expressivité du langage Linda. Ci-après quelques résultats parmi les plus significatifs [Busi *et al.*, 2001].

D'abord, dans le langage Linda basique, *rd* est sémantiquement redondant par rapport au *in* dans la définition basique du langage. Ce résultat n'est plus vérifié dans les systèmes où il existe une synchronisation entre plusieurs *rd* en un *rd* global, car cette synchronisation n'est pas possible avec plusieurs *in* (puisque *in* détruit l'objet et ne permet pas aux autres agents d'y avoir accès). Ensuite, les trois sémantiques pour la primitive *out* (viz. simultanée, ordonnée et non-ordonnée) sont faiblement bisimilaires dans la définition basique du langage Linda. Le résultat n'est plus vérifié lorsqu'on introduit des tests d'absence. De plus, le langage Linda basique n'est pas Turing-complet [Busi *et al.*, 2000b]. Cependant, en étendant le langage avec *inp* (un branchement conditionnel avec un *in* non bloquant), il le devient, mais seulement avec les sémantiques ordonnée et instantanée du *out*. Ensuite, la primitive de notification $notify$ augmente strictement l'expressivité du langage Linda basique, puisqu'il n'existe pas d'encodage permettant de l'exprimer avec *out*, *in* et *rd*. Cependant, il est possible de simuler $notify$ avec *inp* moyennant l'activation d'un protocole spécial à chaque exécution d'un *out*. Le contraire n'étant pas vrai, le langage Linda basique avec *inp* est strictement plus expressif que Linda avec $notify$. La combinaison *inp* et $notify$ rend la sémantique non-ordonnée aussi expressive que les sémantiques ordonnée et instantanée.

Les opérations globales de récupération de tuples (e.g. $rew(m_1, \varnothing)$ dans le langage défini plus haut) augmentent strictement l'expressivité du langage. Ceci est démontré en prouvant qu'il n'existe pas d'encodage de cette primitive en utilisant les autres primitives du langage

(*eval*, *out*, *in* et *rd*). Cependant, en présence de primitives demandant une vision globale, tel que le tfa présenté plus haut, la monotonicité est perdue. La monotonicité concerne le fait que l'ajout d'autres données dans le *tuplespace* n'altère pas les calculs précédemment effectués. Formellement, la monotonicité est satisfaite si la règle suivante est satisfaite :

$$\text{si } (Ag, DS) \rightarrow (Ag', DS') \text{ alors } (Ag, DS \oplus DS'') \rightarrow (Ag', DS' \oplus DS'')$$

Dans un système à multiple *tuplespaces* distribués sur un réseau, si la monotonicité est satisfaite, alors l'exécution d'une opération de coordination peut s'exécuter avec l'ensemble de tuples impliqués dans l'opération, sans avoir à parcourir l'ensemble des *tuplespaces* existants.

2.9 Conclusion

Dans ce chapitre, nous avons présenté les algèbres de processus, et leur utilisation afin de spécifier des langages de coordination orientée-données. Nous croyons que cet effort est nécessaire lors de la proposition d'un modèle de coordination afin de s'assurer que la définition du modèle est non-ambiguë et qu'un système donné adhère bien à un modèle de coordination. Les résultats sur l'expressivité des différents langages sont d'une grande utilité lors du choix du modèle pour une application particulière, et de l'implémentation des différents langages correspondants. La formalisation des langages associés à Linda souffre néanmoins de quelques faiblesses. D'une part, la majorité des travaux se focalisant sur la vérification de l'expressivité des langages, ils ne proposent pas une formalisation complète des types du langage et la manière avec laquelle l'appariement est effectué, à l'exception notable de [Ciancarini *et al.*, 1995]. D'autre part, l'amalgame entre agents et processus donne une définition simpliste d'un agent (un agent = un comportement), ce qui empêche la considération d'un système adhérant à un modèle de coordination orienté-données comme un SMA.

Chapitre 3

Le problème du transport à la demande

Sommaire

3.1 Introduction

Le transport de personnes et de marchandises est un problème que rencontrent beaucoup d'entreprises, non seulement dans le secteur des transports, mais également dans d'autres secteurs. Par exemple, l'acheminement de produits ou d'employés entre usines, ou l'acheminement de courrier dans des grandes entreprises ayant un service de courrier interne. Avec la mondialisation de l'économie, le problème du transport devient de plus en plus important. Dans les années 1980 déjà, approximativement 400 milliards U.S. $ étaient utilisés en coûts de distribution aux États Unis, et aux environs de 15 milliards £ au Royaume Uni [Bodin *et al.*, 1983]. En 1989, 76.5% de tous les transports de marchandises étaient effectués avec des véhicules [Halse, 1992]. Le problème du Transport à la Demande (TAD) est un problème impliquant une offre de transport (utilisant des véhicules) tirée par la demande, i.e. qui répondent à la demande, et n'obéissent pas à des tables de marche (plans) théoriques.

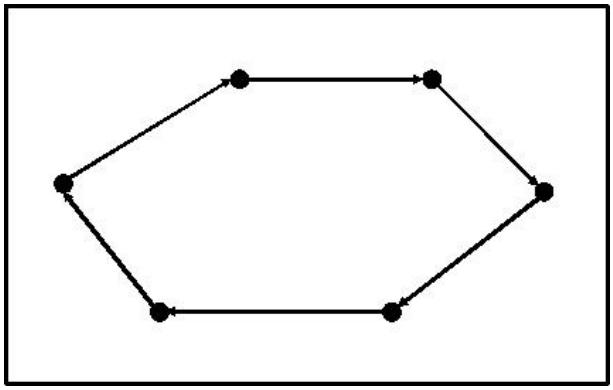

FIG. 3.1 – Problème du voyageur de commerce

Les problèmes traités en recherche sont souvent plus simples que les problèmes réels, mais même si un nombre de contraintes réelles sont ignorées (e.g. contraintes dues à la législation ou aux syndicats), les modèles qui sont développés couvrent les propriétés basiques et offrent des résultats pouvant être utilisés dans l'analyse et l'implémentation de systèmes réels. Le modèle théorique correspondant au problème du TAD fait partie de la grande famille des problèmes de tournées de véhicules, eux-mêmes étant des extensions du problème de voyageur de commerce (TSP[17]). Dans le TSP, un nombre de villes doit être visité par un voyageur de commerce qui doit revenir à la ville depuis laquelle il est parti. La tournée doit être construite de telle manière qu'elle minimise la distance parcourue. La figure 3.1 montre une solution typique au problème de voyageur de commerce.

Le m-TSP est un problème de m voyageurs de commerce devant visiter un ensemble donné de villes. Chaque ville doit être visitée exactement une seule fois, et chaque voyageur doit commencer et finir sa tournée à la même ville (appelée dépôt). La nécessité d'avoir recours à plusieurs voyageurs résulte généralement de la limitation de la longueur maximale d'une tournée. Car sinon, et si on suppose l'inégalité triangulaire, une solution à un véhicule donnera toujours le meilleur résultat.

Le problème de tournées de véhicules (VRP[18]) est un m-TSP où une quantité est associée à chaque ville et où chaque véhicule a une capacité limitée. La somme des quantités associées aux villes parcourues par chaque véhicule ne doit pas excéder sa capacité maximale. Le problème n'est plus seulement géographique (comme le m-TSP) puisque la contrainte de capacité rend des solutions non faisables malgré leur minimisation des distances parcourues. Remarquons que dans la littérature, le m-TSP est quelquefois appelé VRP, et le VRP appelé CVRP (pour VRP avec contrainte de capacité). La figure 3.2 montre une solution typique du problème de tournées de véhicules. Une version à plusieurs dépôts existe pour le VRP (MDVRP[19]), dans laquelle les clients doivent être desservis par un véhicule depuis l'un des dépôts, et chaque véhicule part et revient à son dépôt initial.

Dans le problème de tournées de véhicules avec fenêtres temporelles (VRPTW[20]), en plus des quantités, un intervalle de temps est associé à chaque ville, et un temps de service pendant lequel le véhicule est obligé de stationner avant de pouvoir repartir. Le tuple constitué par une ville (ou noeud), une quantité, un temps de service et une fenêtre de temps est dorénavant appelé

[17] *Travelling Salesman Problem*
[18] *Vehicle Routing Problem*
[19] *Multiple Depots VRP*
[20] *Vehicle Routing Problem with Time Windows*

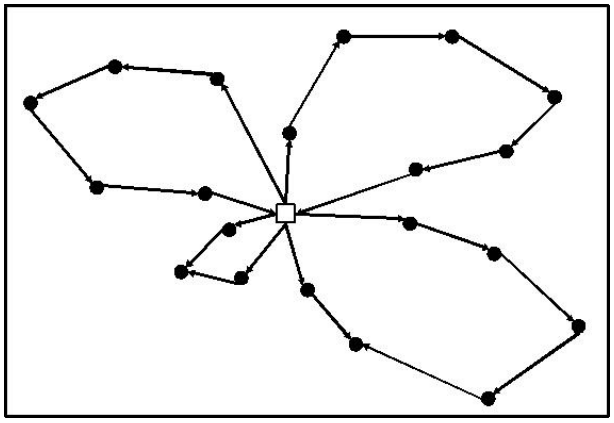

FIG. 3.2 – Problème de tournées de véhicules (4 tournées). Le carré est le dépôt

client. Chaque véhicule ne peut visiter un client que pendant l'intervalle de temps qui lui est associé, et ne peut le quitter avant d'y avoir passé le temps de service associé. Dans certaines versions, un véhicule peut commencer son service s'il arrive trop tôt ou trop tard, et une pénalité est associée à une solution si le véhicule visite trop tard ou trop tôt un client (e.g. [Balakrishnan, 1993]). Mais dans la majorité des versions, et dans notre travail également, les fenêtres de temps sont rigides, dans le sens où une solution est considérée comme non valide si un véhicule visite un client en dehors de sa fenêtre temporelle. Un véhicule est autorisé à visiter un client trop tôt, sans pénalité, mais il ne lui est pas permis de le visiter trop tard. Si le véhicule arrive trop tôt, il doit attendre jusqu'à la borne inférieure de la fenêtre temporelle associée au client avant de commencer son service (comme dans, e.g. [Larsen, 1999; Desaulniers *et al.*, 2002]).

Dans le problème de tournées de véhicules avec ramassage et livraison (PDPTW[21]), les clients demandent d'être transportés d'un point vers un autre par le même véhicule, et non d'être simplement visités par celui-ci. Le point de ramassage comme le point de livraison sont décrits par un tuple comme pour le VRPTW. Une solution est considérée valide s'il s'agit, d'abord, d'une solution valide en considération des contraintes de capacité et de fenêtres de temps, et si chaque couple de noeuds d'un même client est desservi par le même véhicule et enfin, si tout point de ramassage d'un client est visité avant son point de livraison. Une observation est à relever pour le problème PDPTW, par rapport au VRPTW, au regard des contraintes de capacités. Le nombre minimal de véhicules utilisés en VRPTW est égal à la somme des quantités demandées par les clients divisée par la capacité maximale d'un véhicule. Cette borne inférieure n'est plus valide pour le PDPTW, puisque la somme des quantités demandées est égale à zéro (la quantité demandée au point de départ est annulée par la quantité négative du point d'arrivée). Le résultat est qu'une solution à un véhicule est théoriquement (si les fenêtres temporelles le permettent) toujours possible dans le problème TAD alors que ce n'est pas le cas en VRPTW.

Dans la communauté anglophone, le PDPTW concerne le transport de marchandises [Desaulniers *et al.*, 2001]. Le problème de transport à la demande de personnes est quant à lui appelé *Dial-A-Ride Problem* (DARP) [Fu and Teply, 1999]. Dans la communauté de l'ingénierie des transports, les systèmes de transport à la demande de personnes s'appellent *Demand-Responsive Transportation systems* (DRTS) [Horn, 2002] ou encore *Paratransit* dans la communauté états-unienne [Cervero, 1997]. Le terme *Transportation On Demand* désigne quant à lui toute sorte d'offres de transport structurées par la demande, ne se limitant pas aux véhicules

[21] *Pickup & Delivery Problem with Time Windows*

(e.g. avions) [Cordeau *et al.*, 2004]. Dans la communauté francophone, le terme « Transport A la Demande » désigne le problème de transport de personnes [Garaix *et al.*, 2006].

Les systèmes de transport à la demande de personnes se distinguent par la présence de deux objectifs conflictuels : minimiser les coûts opérationnels et minimiser le désagrément causé à l'usager. Les coûts opérationnels sont en général relatifs à la taille de la flotte de véhicules mobilisée et à la distance parcourue. Le désagrément de l'utilisateur est relatif à la distance et au temps additionnels par rapport à un chemin direct entre les points de ramassage et de livraison. Une manière maintenant classique de balancer ces deux objectifs est de traiter la minimisation des coûts opérationnels comme objectif premier et d'imposer une qualité de service minimale devant être observée [Cordeau, 2006].

Tous les problèmes de tournées de véhicules sont subdivisés en deux catégories concernant la configuration du problème : les problèmes statiques et les problèmes dynamiques. Dans les problèmes statiques, le système dispose de tous les paramètres du problème, et ces données ne changent pas jusqu'à la fin du traitement. Dans les problèmes dynamiques, les paramètres du problème sont reçus au fur et à mesure de l'exécution, et doivent être incorporés dans la résolution. La principale source de dynamicité étudiée dans la littérature concerne l'arrivée des clients. Ces derniers apparaissent au fur et à mesure de la résolution. Tant et si bien que le terme VRP dynamique et ses variantes désigne désormais ce problème : un VRP où tous les clients ne sont pas connus au début du calcul. Les autres sources de dynamicité telles que le changement dans les temps de parcours sont désignés par d'autres noms (e.g. dans [Ichoua *et al.*, 2003][22]ou dans [Potvin *et al.*, 2006]).

Les idées sous-jacentes aux propositions traitant du TAD sont issues en grande majorité de celles déjà utilisées dans les problèmes de tournées de véhicules. Aussi, les contraintes temporelles sont celles qui augmentent considérablement la difficulté du problème. Ce chapitre présente l'état de l'art des approches pour la résolution des VRP statiques et dynamiques, avec une attention particulière aux deux problèmes VRPTW et TAD. Ce chapitre est structuré comme suit. Dans la section 3.2, nous exposons les expériences opérationnelles d'implantation de systèmes TAD. Dans la section 3.3, nous donnons une formulation mathématique du problème TAD. Dans la section 3.4, nous décrivons brièvement les résolutions exactes du problème. La section 3.5 présentent les approches centralisées, autant pour le cas statique que pour le cas dynamique, et la section 3.6 décrit les approches distribuées .

3.2 Expériences opérationnelles

3.2.1 Expériences aux États-Unis

Aux États-Unis, plusieurs expériences de systèmes TAD ont été implantés, et ce depuis les années 1970 en réaction au choc pétrolier. Les systèmes les plus importants sont Haddonfield dans le New Jersey, Rochester à New York and Ann Arbor dans le Michigan. Dans aucun de ces systèmes, la rentabilité économique n'a été atteinte, mais ils ont servis comme base d'expérimentationafin de tester des algorithmes d'ordonnancement de véhicules [Diana, 2002]. D'autres systèmes ont été implémentés depuis, les plus grands étant El Cajon en Californie, Danville dans l'Illinois et Davenport dans l'Iowa. Mais la rentabilité économique n'était toujours pas au rendez-vous, et une augmentation des tarifs empirait la situation. Par exemple, il est reporté dans [Cervero, 1997] que lorsque dans El Cajon, le tarif par voyage a doublé, passant de 2$ à 4$, le déficit par voyage est passé de 3$ à 10$. En général, les difficultés financières

[22]Le problème traité y est appelé *Vehicle dispatching with time-dependent travel times*

finissaient par avoir raison de ces systèmes.

Dans les années 1990, le TAD a connu une renaissance grâce au *American with Disabilities Act* qui obligea toutes les sociétés de transport à fournir un service de transport à la demande pour les personnes à mobilité réduite (PMR) qui ne peuvent utiliser les systèmes de transport existants. À la fin des années 1990, il existait des centaines de tels systèmes. Même si ces systèmes ont connu un vif succès (en termes de fréquentation), les caractéristiques spartio-temporelles des demandes faisaient en sorte que les possibilités de covoiturage étaient minimes, et le coût exorbitant, i.e. équivalent à un taxi. De nouveaux crédits fédéraux ont été donc alloués à des recherches visant à améliorer la rentabilité de ces systèmes, en automatisant les tâches effectuées jusque là manuellement. Cependant, le niveau d'équipement technologique demeurait relativement bas, même pour des tâches typiquement automatisables telles que le *dispatching* des véhicules.

3.2.2 Expériences Européennes

En Europe, les projets européens SAMPO[23] et SAMPLUS[24] (1995-1999) ont stimulé l'implantation de systèmes de petites ou moyennes tailles en Belgique, en Italie, en Finlande et en Suède, dont la majorité sont toujours en service (voir [Mageean and Nelson, 2003] pour une évaluation des systèmes TAD en Europe). En suisse, le service *Car Postal* offre ses services sur tout le territoire national. Le système allemand *Ruf-Bus* a été implémenté dans les années 1970 et a connu différentes remises à niveau et est toujours en service. Il peut passer d'un service avec des itinéraires fixes pour devenir un système TAD pendant le week-end. Les types de véhicules utilisés varient selon les circonstances.

L'expérience britannique rejoint les expériences européennes en terme d'historique, de défis et de barrières au développement. Une étude spécifique à cette expérience a été conduite [Enoch *et al.*, 2004], dont le but était de regarder le potentiel pour le TAD comme système alternatif pour les transports en commun, en termes de marché ou d'espace de demande, du point de vue du service public et des opérateurs commerciaux. L'étude vise à déterminer comment le TAD pourrait être développé pour servir les voyageurs qui ne sont pas bien servis par les transports en communs (TC) et explore les raisons pour lesquelles le TAD n'ont pas eu beaucoup d'impact jusqu'alors, et comment le gouvernement et d'autres services publics pourraient rectifier le tir [Enoch *et al.*, 2004].

Différentes catégories de TAD ont été identifiées. La catégorie des TAD dits « de rechange » survient quand, au lieu de compléter les services de bus conventionnels, un système de TAD remplace - totalement ou partiellement - ceux-ci. La catégorie des TAD « d'échange » vise à assurer l'échange avec les réseaux de transports en commun (TC). Pour cette dernière catégorie, il est approprié d'avoir des prix modérément au dessus des taux tarifaires des bus, mais avec des concessions pour les groupes et des rabais pour les dessertes pré-réservées à des arrêts fixes. Une autre catégorie de TAD est celle à « destination spécifique » : ces services ne tendent pas à assurer des chaînes de voyage (inter-connectivité), et donc les systèmes de prix, les billets et/ou les horaires peuvent être indépendants.

En France, le rapport du Certu [Certu, 2002] dresse un bilan de 15 expériences dans le domaine des systèmes TAD en France (plus 4 expériences Européennes). Les types de dessertes sont très hétérogènes, allant des services qui se substituent au réseau régulier de TC à certaines heures ou dans certaines zones, jusqu'aux services dédiés aux Personnes à Mobilité Réduite

[23] `http://cordis.europa.eu/telematics/tap_transport/research/projectsum/sampo.html`
[24] `http://cordis.europa.eu/telematics/tap_transport/research/projectsum/samplus.html`

(PMR), en passant par les services spécialement organisés pour les trajets domicile-travail. Plusieurs catégories de véhicules sont utilisés, tels que les taxis avec des véhicules standard, les monospaces et les minibus spécialement aménagés pour les PMR. Plusieurs mode de fonctionnement existent, des statiques (des lignes virtuelles avec itinéraires, arrêts et horaires fixes activées à la demande) aux dynamiques (des services sur mesure : porte à porte mis au point avec chaque client, mais qui reste inchangé pendant la période d'abonnement), mais aucun ne propose de systèmes complètement flexible et automatisé.

La clientèle utilisatrice de ces systèmes est majoritairement constituée des scolaires, représentant plus de la moitié des usagers des systèmes de TAD étudiés, mais aussi les plus de 65 ans, essentiellement de sexe féminin, les travailleurs (trajets domicile - travail ou affaires) et les PMR. Les taux de fréquentation observés des systèmes TAD sont de 1,2 à 1.5 personnes transportées par course pour les taxis dits collectifs, le taux fréquentation des TAD par rapport aux TCs varie de 1 pour mille à 5 pour cent selon les sites.

La majorité des services de TAD pratiquent la même tarification que les TCs et acceptent les mêmes abonnements. Quelques services TAD pratiquent des tarifications de l'ordre de deux à quatre fois le prix des tickets TC. Enfin, certains services affichent des prix forfaitaires élevés pour des destinations particulières (90 euros pour un aller-retour par des navettes vers les aéroports parisiens, depuis Reims). Il est à noter ici que la pratique de la même tarification que les TCs constitue un plus quant à la complémentarité avec eux, qui est un critère essentiel dans l'évaluation de la viabilité d'un système TAD opérationnel.

Les systèmes de gestion dans les expériences recensées par le Certu intégrée se caractérisent en trois groupes : les systèmes propriétaires, les systèmes avec logiciels spécifiques et les systèmes d'exploitation simples.

Les systèmes propriétaires sont développés en interne après étude des besoins du TAD à mettre en place. C'est le cas de Routair à Reims. Ce système possède les grandes fonctions de logiciels dits spécifiques, à savoir : réservation, planification et gestion. Le système intègre un module de calcul de prix de revient de chaque tournée.

Dans les systèmes avec logiciels spécifiques, les exploitants utilisent ici des logiciels qui ont été développés spécifiquement pour optimiser les transports à la demande. Les fonctions de planification, optimisation des courses, suivi de clientèle, cartographie voire localisation, facturation pour certains se rencontrent fréquemment.

Dans le cas d'utilisation d'un système d'exploitation simple, les exploitants n'utilisent pas de logiciels dédiés aux TAD. C'est une simple gestion des courses avec utilisation de logiciels bureautiques de type tableur et base de données. La prise en charge de la clientèle est effectuée via une centrale de réservation ou par appel téléphonique. Certains sites n'utilisent pas de systèmes de gestion intégrée et travaillent seulement avec un centre d'appel qui se charge de répercuter la demande vers le transporteur. Il est à remarquer que l'offre sur le marché souvent ne répond pas aux besoins spécifiques de l'exploitant. Cette offre ne lui est souvent pas visible, il ne connaît donc pas l'offre réelle disponible sur le marché.

3.2.3 Bilan

Les services de TAD sont typiquement plus coûteux par passager que les bus conventionnels. Cependant, étant plus flexibles, ils peuvent assurer efficacement un service « pilote » de bus dans une zone jusqu'à ce que le niveau de demande sur certaines routes ou dans des arrêts particuliers peut être totalement assuré avec des ressources allouées à un service à route fixe [Enoch *et al.*, 2004]. Ce passage entre TAD et TC est très important, et il est bidirectionnel dans le sens où un système TAD ne cherche pas seulement à combler des manques à gagner (pour raisonner en

termes économiques seulement), mais aussi en traquant le gaspillage sur les lignes fixes (de TC à TAD).

La majorité des services TAD ont été mis en place pour des considérations sociales et se sont donc concentrés sur des usagers cibles, qui ont par définition un choix de transport limité, et en particulier ceux qui ont un accès limité aux voitures personnelles. En revanche, nombre de services TAD ont comme cible des voyageurs de choix, dont beaucoup pourrait faire le trajet en voiture. Ce dernier groupe est particulièrement intéressant dans le cadre d'une politique environnementale [Enoch *et al.*, 2004].

Dans tous les systèmes étudiés en Europe et aux États-Unis, la proportion relativement faible de l'utilisation de solutions informatiques génériques à la problématique de TAD est frappante, et le gap entre recherche et industrie demeure assez net. Dans [Enoch *et al.*, 2006], d'autres conclusions ont été dégagées (72 expériences recensées). Il est conclu que les projets TAD sont souvent trop coûteux ou alors conçus sans une compréhension totale du marché qu'ils vont servir. La tentation d'offrir un service trop flexible et d'introduire des systèmes technologiques coûteux, quand il n'y en a pas réellement besoin, est dangereuse. Les systèmes TAD requièrent également un effort de *marketing* plus important que pour les services de bus réguliers, mais plus que tout, ils nécessitent des capacités importantes de travail en partenariat avec les opérateurs en place. Ce dernier point a été identifié comme la source principale de l'échec d'une grande partie des expériences opérationnelles de systèmes TAD.

3.3 Formulation du problème TAD

Dans [Cordeau, 2006], Cordeau *et al.* donnent une formulation du problème du TAD, nous renvoyons le lecteur à [Desaulniers *et al.*, 2002] pour une formulation du VRPTW. Soit n le nombre de clients du problème. Le problème TAD peut être défini comme un graphe complet dirigé $G = (N, A)$ où $N = P \cup D \cup \{0, 2n+1\}$. $P = \{1, \ldots, n\}$ et $D = \{n+1, \ldots, 2n\}$. Les sous-ensembles de P et D contiennent respectivement les noeuds de départ et d'arrivée, tandis que les noeuds 0 et $2n+1$ représentent le dépôt. Avec chaque client sont associés le noeud de départ i et le noeud d'arrivée $n+i$. Soit K l'ensemble des véhicules. Chaque véhicule $k \in K$ a une capacité Q_k et la durée totale de sa tournée ne peut dépasser T_k. Avec chaque noeud $i \in N$ est associée une quantité q_i et une durée de service positive s_i telles que $q_0 = q_{2n+1} = 0$, $q_i = -q_{n+i}$ $(i = 1, \ldots, n)$ et $s_0 = s_{2n+1} = 0$. Une fenêtre de temps $[e_i, l_i]$ est également associée avec les noeuds $i \in N$ où e_i et l_i représentent respectivement le temps au plus tôt et au plus tard auquel le service doit commencer au niveau du noeud i. Avec chaque arc $(i, j) \in A$ sont associés un coût c_{ij} et un temps de parcours t_{ij}. Enfin, L est le temps de parcours maximal du client.

Pour chaque arc $(i, j) \in A$ et chaque véhicule $k \in K$, soit $x_{ij}^k = 1$ si le véhicule k se déplace du le noeud i au noeud j directement, et 0 sinon. Pour chaque noeud $i \in N$ et chaque véhicule $k \in K$, soit B_{ki} le temps auquel le véhicule k commence son service au niveau du noeud i, et Q_i^k la charge du véhicule k après avoir terminé son service au niveau du noeud i. Finalement, pour chaque client i, soit L_i^k le temps passé par le client i à bord du véhicule k. La fonction objectif minimise le coût total et est définie ainsi :

$$\min \sum_{k \in K} \sum_{i \in N} \sum_{j \in N} c_{ij} x_{ij}^k \tag{3.1}$$

elle est soumise aux contraintes suivantes :

$$\sum_{k \in K} \sum_{j \in N} x_{ij}^k = 1 \quad \forall i \in P \cup D \tag{3.2}$$

Les contraintes (3.2) restreignent l'affectation de chaque client à exactement un seul véhicule.

$$\sum_{j \in N} x_{ij}^k - \sum_{j \in N}^{N} x_{n+i,j}^k = 0 \quad \forall i \in P, \forall k \in K \tag{3.3}$$

Les contraintes (3.3) s'assurent que le noeud de départ et d'arrivée sont desservis par le même véhicule.

$$\sum_{j \in N} x_{j0}^k = 1 \qquad \forall k \in K \tag{3.4}$$

$$\sum_{i \in N} x_{ij}^k - \sum_{i \in N} x_{ji}^k = 0 \quad \forall j \in P \cup D, \forall k \in K \tag{3.5}$$

$$\sum_{j \in N} x_{j,2n+1}^k = 1 \qquad \forall k \in K \tag{3.6}$$

Les contraintes (3.4) à (3.6) caractérisent le chemin à suivre par un véhicule k : k doit quitter le dépôt une seule fois (3.4), s'il dessert un autre client, il doit le quitter (3.5) et finalement retourner au dépôt une seule fois (3.6).

$$Q_j^k \geq (Q_i^k + q_j) x_{ij}^k \quad \forall i \in N, \forall j \in N, \forall k \in K \tag{3.7}$$

$$\max(0, q_i) \leq Q_i^k \leq \min(Q_k, Q_k + q_i) \qquad \forall i \in N, k \in K \tag{3.8}$$

Les contraintes (3.7) et (3.8) garantissent la non-violation des limites de capacité de chaque véhicule. Si le véhicule k se déplace de i vers j ($x_{ij}^k = 1$), sa capacité après la desserte de i plus la quantité demandée par j est égale à sa capacité après la desserte de j (Q_j^k) (3.7). De plus, après avoir desservi n'importe quel noeud i, le véhicule k a une capacité supérieure à 0 (ou à q_i si q_i est positif) et inférieure à sa capacité maximale Q_k (ou a $Q_k - qi$ si q_i est positif) (3.8).

$$e_i \leq B_i^k \leq l_i \qquad \forall i \in N, k \in K \tag{3.9}$$

$$B_j^k \geq (B_i^k + s_i + t_{ij}) x_{ij}^k \quad \forall i \in N, \forall j \in N, \forall k \in K \tag{3.10}$$

$$B_{2n+1}^k - B_0^k \leq T_k \qquad \forall k \in K \tag{3.11}$$

Les contraintes (3.9), (3.10) et (3.11) s'assurent de la non-violation des contraintes temporelles. Tous les temps de service à n'importe quel noeud i doivent commencer à l'intérieur des fenêtres temporelles correspondantes à i (3.9). Si le véhicule k visite j après i, il ne peut pas commencer son service au niveau de j avant $B_i^k + s_i + t_{ij}$ (3.10). Chaque tournée ne peut durer plus longtemps que la durée maximale du parcours du véhicule (3.11).

$$L_i^k = B_{n+i}^k - (B_i^k + s_i) \quad \forall i \in P, \forall k \in K \tag{3.12}$$

$$t_{i,n+i} \leq L_i^k \leq L \quad \forall i \in P, \forall k \in KF \tag{3.13}$$

Les contraintes (3.12) et (3.13) sont relatives à la qualité du service offert aux clients. Son temps de parcours est calculé en soustrayant son temps de départ et son temps de service de son temps d'arrivée (3.12). Ce temps de parcours ne doit pas dépasser le temps de parcours maximal

d'un client, et ne peut pas être inférieur au temps de parcours direct entre son noeud de départ et son noeud d'arrivée (3.13).

$$x_{ij}^{k} \in \{0,1\}, \quad \forall i \in N, \forall j \in N, \forall k \in K \tag{3.14}$$

Enfin, les contraintes (3.14) restreignent les variables à des valeurs binaires.

3.4 Approches exactes et complexité

Trouver une solution faisable pour le TAD est NP-Difficile puisqu'il généralise le TSP avec fenêtres temporelles [Cordeau, 2006]. En effet, le TAD contient différents problèmes d'optimisation NP-Difficiles impliquant qu'il est également NP-difficile. Parmi ces problèmes NP-difficiles, on a le VRP [Lenstra and Kan, 1981], lui-même contenant le TSP ([Garey and Johnson, 1979; Lenstra and Kan, 1981]) et le *Bin Packing* [Garey and Johnson, 1979].

Presque tous les algorithmes exacts utilisent l'un de ces trois principes :

1. Programmation dynamique
2. Méthodes fondées sur la relaxation Lagrangienne.
3. Génération de colonnes.

Les premières recherches sur le TAD ont été effectuées par Psaraftis [Psaraftis, 1980; Psaraftis, 1983] qui a développé des algorithmes de programmation dynamique pour le cas avec un seul véhicule. Une version améliorée a été proposée par Desrosiers *et al* [Desrosiers *et al.*, 1986] qui était capable de résoudre des instances allants jusqu'à 40 clients. Dumas et al. [Dumas *et al.*, 1991] ont présenté un algorithme exact fondé sur la génération de colonnes. Récemment, Cordeau [Cordeau, 2006] a proposé une méthode *Branch & Cut* pour la résolution du TAD, il y traite des problèmes de taille allant jusqu'à 32 clients, avec des temps de calcul allant de 30 minutes à 2 heures. Huit problèmes sur trente n'avaient pas pu être résolus. Dans sa thèse, Diana [Diana, 2002] a modélisé le problème du TAD. Le nombre de variables induit par sa formulation est $NV = 4m(n+1)^2 + 8n + 2m$, avec n le nombre de requêtes et m le nombre de véhicules ; le nombre de contraintes est égal à $NC = 22n^2m + 17nm + 15n + 6m$. La taille des problèmes pouvant être résolus en quelques minutes ne dépassent pas cinq requêtes et deux véhicules, ce qui ne peut représenter un problème réaliste, ou d'un quelconque intérêt expérimental.

Les VRP et leurs variantes étant des problèmes NP-Difficiles, les VRP dynamiques sont également NP-Difficiles, puisqu'un nouveau problème statique est à résoudre à chaque apparition d'un nouveau client.

Le lecteur intéressé par les approches d'optimisation peut se reporter à [Desrochers *et al.*, 1988], et le lecteur intéressé par un état de l'art spécifique au problème TAD à [Cordeau and Laporte, 2003; Desaulniers *et al.*, 2001].

3.5 Approches centralisées

Notre présentation porte sur les méthodes heuristiques et métaheuristiques qui essaient de donner de bons résultats, en des temps d'exécution raisonnables. Dans ces méthodes, la taille de la flotte de véhicules n'est pas fixée. Elle devient le premier critère à minimiser, le second critère demeurant le coût global, fonction de la distance totale parcourue par l'ensemble des véhicules. Notre présentation est structurée selon deux axes : les approches statiques et les approches dynamiques, les premières considérant l'ensemble des clients comme connus avant le démarrage

de la résolution, alors que les secondes considèrent des clients arrivant au fur et à mesure de la résolution.

3.5.1 Approches statiques

Les premiers travaux ont considéré uniquement le problème statique, à cause de la difficulté technologique de mettre en place un système de transport en temps-réel. Par conséquent, la littérature abonde de propositions pour les versions statiques du problème, et le cas dynamique est relativement moins étudié. Ces approches se subdivisent en heuristiques et méta-heuristiques. La différence entre les heuristiques et les méta-heuristiques est que les premières sont proposées spécifiquement pour les problèmes de tournées de véhicules, alors que les secondes sont proposées indépendamment du problème, mais sont calibrées pour y être appliquées. Le résultat est que l'intuition des heuristiques par rapport au problème traité est plus claire que celle des méta-heuristiques.

Heuristiques

La résolution des problèmes de tournées de véhicules étant d'une grande complexité, il est important d'avoir de bonnes stratégies heuristiques de résolution [Aronson, 1996]. Bowerman *et al.* [Bowerman *et al.*, 1994] décrivent plusieurs classes de stratégies pour résoudre le VRP. Nous les présentons ci-après, avec une attention particulière sur les heuristiques d'insertion qui inspirent notre travail.

1. **Programmation mathématique** La programmation mathématique consiste en général à l'exécution des étapes suivantes :
   (a) formuler un problème relaxé d'une ou plusieurs contraintes à partir de la formulation initiale du problème, telles que les restrictions sur l'intégrité des variables,
   (b) résoudre le problème relaxé (par programmation linéaire),
   (c) ajuster la solution de sorte qu'elle satisfasse les contraintes éliminées, ou alors ajouter quelques contraintes et revenir à la deuxième étape.
2. **Amélioration / échange**
   L'idée centrale des algorithmes d'amélioration / échange est de trouver une solution initiale et ensuite d'essayer de l'améliorer en échangeant des noeuds, des arcs ou des véhicules. Un algorithme connu de ce type est celui décrit dans [Clarke and Wright, 1964]. L'idée de base est que tous les arcs qui ne sont pas dans la solution courante sont essayés. Si deux noeuds sont- dans la solution courante - desservis par différents véhicules, nous pouvons construire une nouvelle solution en créant un arc entre ces deux noeuds.
3. **Gains / insertion**
   L'idée des algorithme de gains / insertion est de trouver rapidement une solution initiale et ensuite, de l'améliorer vers une solution moins coûteuse. La différence entre cette méthode et la méthode d'amélioration / échange est que nous ne sommes pas obligés de maintenir des solutions faisables tout le temps, mais on doit bien évidemment en avoir une en fin d'exécution. Souvent, la solution initiale est construite en créant des itinéraires qui contiennent seulement un client. Ces itinéraires peuvent alors être fusionnés ensemble tant que ceci améliore les coûts [Paessens, 1988].
4. ***Route first / cluster second***
   L'idée générale de cette classe d'algorithmes est la suivante :

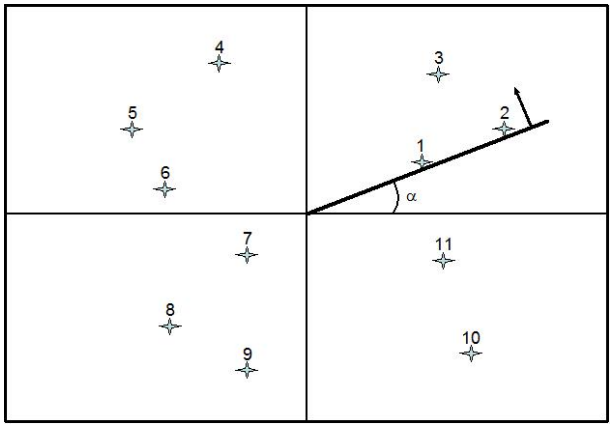

FIG. 3.3 – Le principe de l'algorithme SWEEP

– construire un seul itinéraire TSP pour tous les noeuds excepté le dépôt,
– fragmenter l'itinéraire en plusieurs tel que chacun peut être confié à un véhicule (« problème de partitionnement »).

Bowerman *et al.* [Bowerman *et al.*, 1994] utilisent les courbes de recouvrement de l'espace (*space filling curves*) afin de trouver la solution initiale au TSP initial (avec tous les noeuds), et des techniques de programmation dynamique afin de la fragmenter en M solutions réalisables. Le VRP traité est un problème euclidien.

Définition VRP EUCLIDIEN

Un VRP euclidien est défini de telle manière que chaque noeud i est décrit par ses coordonnées (x_i, y_i), et où la distance entre deux noeuds i et j est égale au temps de parcours et est calculée selon la métrique euclidienne :

$$d_{ij} = t_{ij} = \sqrt{(x_i - y_i)^2 + (x_j - y_j)^2}$$

Puisque cette classe d'approches divise la résolution en deux étapes, pour chaque étape, différentes approches peuvent être utilisées.

5. ***Cluster first / route second***

Il s'agit de grouper les noeuds, affecter chaque groupe à un véhicule, et pour chaque véhicule, résoudre le problème de voyageur de commerce correspondant. Dans cette catégorie, Gillet et Miller [Gillett and Miller, 1974] ont proposé un algorithme appelé SWEEP pour le VRP euclidien qui se fonde sur la direction géographique des clients.

Les auteurs ont considéré que tous les véhicules ont la même capacité. La position des clients peut être donnée en coordonnées polaires et le dépôt est considéré comme se trouvant à l'origine. Les noeuds sont ordonnés selon l'ordre croissant de leurs angles. Si deux noeuds ont le même angle, alors celui qui a le rayon le plus petit vient avant l'autre. Supposons qu'on ait ordonné et numéroté les noeuds (le noeud 0 est le dépôt), l'algorithme est le suivant (c.f. figure 3.3) :

(a) commencer par le dépôt,

(b) ajouter le noeud avec le numéro le plus petit à l'itinéraire du véhicule courant ; si la capacité du véhicule est dépassée, choisir un nouveau véhicule et retourner à la première étape,

	$client_1$	$\underline{client_2}$	$client_3$
$vehicule_1$	4.5	∞	20
$\underline{vehicule_2}$	3.5	$\underline{1.5}$	6
$vehicule_3$	∞	2	∞

TAB. 3.1 – Heuristique de Solomon. Le client 2 est inséré dans le tournée du véhicule 2

(c) répéter la deuxième étape jusqu'à ce que tous les clients soient desservis.

Après la construction de ces tournées, les auteurs essaient d'améliorer l'ordonnancement en permutant des noeuds entre les tournées.
Fisher and Jaikumar [Fisher and Jaikumar, 1981] ont décomposé le problème de regroupement en deux étapes :

(a) D'abord, affecter un client « graine » (*seed*) à chaque véhicule.
(b) Ensuite, affecter tous les clients non encore affectés au véhicule « le moins cher » selon une fonction objectif.

Ces deux étapes forment désormais une classe classique de méthodes appelées heuristiques d'insertion, et ont connu un grand succès. Les heuristiques d'insertion sont, dans leur version originale, des algorithmes gloutons, dans la mesure où une décision d'allouer tel véhicule à tel client est irrévocable. Deux versions sont envisageables : l'insertion séquentielle ou parallèle. « Parallèle » dans ce contexte ne veut pas dire concurrente, mais simplement que plusieurs plans partiels sont créés au fur et à mesure, en opposition à l'insertion séquentielle qui planifie la tournée d'un seul véhicule avant d'en considérer un nouveau (un seul plan partiel à la fois). Dans [Liu and Shen, 1999], les auteurs montrent que les procédures d'insertion parallèle surpassent les approches séquentielles. Nous décrivons leur principe général.
D'abord, une fonction objectif est définie, généralement une somme pondérée des distances, temps de parcours et temps d'attente. Deux étapes sont alors réalisées : le choix de clients graines et le processus d'insertion. Les clients graines sont choisis de telle sorte qu'ils peuvent difficilement coexister dans un même plan (même véhicule). Le processus d'insertion du client $u^* \in C$ - l'ensemble des clients -, dans le plan du véhicule $v^* \in V$ - l'ensemble des véhicules - est effectué de telle manière que l'accroissement de la fonction objectif est minimal (heuristique de Solomon [Solomon, 1987]). Afin de fixer le nombre de clients graines (le nombre de véhicules donc), plusieurs choix sont possibles. On peut fixer un nombre très grand, et faire tourner le système avec de moins en moins de véhicules jusqu'à ce que le système commence à rejeter des clients. On peut aussi estimer le nombre de véhicules, et le revoir à la hausse ultérieurement si une requête client se trouve rejetée.
La table 3.1 illustre l'heuristique de Solomon. En lignes, figurent les véhicules et en colonnes les clients. Pour chaque client et chaque véhicule, le coût d'insertion est calculé. Lorsque le client ne peut pas être inséré dans la tournée du véhicule, un coût infini est renseigné. L'insertion du client 2 dans la tournée du véhicule 2 est la moins chère, c'est donc cette insertion qui est effectuée à cette étape de l'exécution. Une fois le client inséré, il est éliminé du tableau, et les coûts d'insertion des autres clients dans la tournée du véhicule 2 sont recalculés. La complexité de l'heuristique de Solomon est de l'ordre de $O(n^2)$, avec n le nombre de clients.
Supposons qu'un client u ne soit insérable que dans la tournée d'un seul véhicule v mais à un grand coût, et supposons que ce véhicule soit choisi, selon l'heuristique de Solomon,

	$client_1$	$client_2$	$\underline{client_3}$
$vehicule_1$	4.5	100	20
$\underline{vehicule_2}$	3.5	1.5	$\underline{6}$
$vehicule_3$	100	2	100
Regret	96.5	99	$\underline{108}$

TAB. 3.2 – Heuristique de regret. Le client 3 est inséré dans le tournée du véhicule 2

pour desservir un autre client u' de telle manière que u ne devienne plus insérable dans la tournée de v. Dans ce cas, le choix de u' s'avère être un choix malheureux, bien que minimisant le coût additionnel, car il implique la mobilisation d'un nouveau véhicule pour la desserte de u. Afin de pallier cette myopie de l'heuristique de Solomon, l'heuristique de regret tente de se projeter un pas dans le futur en choisissant le véhicule v^* et le client u^* tels que le coût prévu d'une insertion ultérieure de u^* est maximal. C'est à dire que le client choisi pour être inséré est celui qui risque de voir le coût de son insertion s'envoler s'il n'est pas inséré immédiatement (voir e.g [Liu and Shen, 1999] pour l'utilisation de l'heuristique de regret pour le VRPTW et [Diana, 2002] pour le TAD).

La table 3.2 illustre l'heuristique de regret. Une nouvelle ligne illustrant le regret par client est ajoutée. Lorsqu'un client ne peut pas être inséré dans la tournée d'un véhicule, le coût associé est fixé à une grande valeur (100 dans notre exemple). Pour chaque client u, le véhicule pour lequel son insertion est minimale est identifié (véhicule 2 pour le client 1, véhicule 2 pour le client 2 etc.), appelons le coût associé à ce véhicule $min(u)$. Le regret d'un client u est égal à la somme des différences entre le coût de l'insertion de u dans tous les autres véhicules moins $min(u)$. Le client 3 a le regret maximal, il est donc inséré dans la tournée du véhicule 2. La complexité de l'heuristique de regret est de $O(n^3)$.

Méta-heuristiques

Une grande quantité de solutions proposées dans la littérature sont des méta-heuristiques. Les méta-heuristiques ont présenté de meilleurs résultats que les heuristiques, en moyenne et avec les problèmes *benchmark*. Par exemple, la recherche locale [Rochat and Taillard, 1995], le recuit simulé [Czech and Czarnas, 2002] et les colonies de fourmis [Gambardella *et al.*, 1999]. Le passage d'une solution à une solution voisine peut se faire notamment en échangeant un ou plusieurs noeuds entre différents véhicules (*city-swap*), ou en échangeant un ou plusieurs arcs entre véhicules (*k-opt*), en gardant toujours la meilleure des deux solutions. Toutes les approches méta-heuristiques fonctionnent avec de nombreux paramètres, dont les valeurs influencent grandement les résultats, et qui, pour être calibrées correctement, nécessitent généralement plusieurs exécutions du système.

Les algorithmes génétiques introduits par Holland [Holland, 1975] ont été également largement utilisés pour la résolution des VRPs (voir e.g. [Homberger and Gehring, 1999]). Les algorithmes génétiques simulent le processus d'évolution naturelle suivant le modèle Darwinien dans un environnement donné. Pour un problème d'optimisation donné, un individu représente une solution potentielle, auquel on associe la valeur du critère à optimiser. On génère ensuite de façon itérative des populations d'individus sur lesquelles on applique des processus de sélection, de croisement et de mutation. La sélection a pour but de favoriser les meilleurs éléments de la population pour le critère considéré (les mieux adaptés), le croisement et la mutation assurent l'exploration de l'espace d'états. Le lecteur intéressé est renvoyé à [Bräysy *et al.*, 2004] pour un

état de l'art des approches évolutionnistes pour le VRPTW (algorithmes génétiques et stratégies d'évolution).

La recherche tabou est l'une des techniques les plus utilisées pour sa capacité à éviter les minima locaux lors du processus d'amélioration (voir e.g. [Taillard *et al.*, 1997]). Ceci est fait en acceptant une détérioration de la solution courante durant un certain nombre d'itérations. Une recherche avec tabou part d'une solution initiale x_1 et visite à chaque itération le meilleur voisin x_{t+1} jusqu'à atteindre un critère d'arrêt. Si $f(x_t)$ est le coût de la solution x_t à un minimum local, $f(x_t+1)$ n'est pas forcément inférieure à $f(x_t)$, mais il est possible de trouver une solution de meilleur coût si le minimum local x_t n'est pas le minimum global. Afin d'éviter de boucler sur les solutions déjà explorées, les solutions récemment examinées sont dites « tabou », et ajoutées dans une liste de solutions interdites.

Plusieurs travaux combinent des heuristiques d'insertion avec des méta-heuristiques. Par exemple, Garaix *et al.* [Garaix *et al.*, 2006] proposent d'utiliser une heuristique d'insertion pour le problème TAD, suivie d'une recherche locale, consistant à réannoncer les clients dans le même ordre d'une manière répétitive jusqu'à aboutir à une solution stable, i.e. qui ne s'améliore plus.

3.5.2 Approches dynamiques

Les problèmes opérationnels de tournées de véhicules sont rarement statiques, et nous pouvons dire qu'aujourd'hui, un système statique n'a pas de chance de satisfaire les besoins en mobilité des utilisateurs. En effet, dans un contexte opérationnel, et même si l'ensemble des requêtes est connu avant l'exécution, il existe toujours quelques éléments qui rendent le problème dynamique. Ces éléments englobent les pannes, les retards, les voyageurs ne se présentant pas, etc. Il est ainsi toujours utile de considérer un problème qui n'est pas totalement statique. Dans [Ichoua *et al.*, 2000; Gendreau and Potvin, 1998] et dans [Larsen, 2000], un état de l'art des approches pour les versions dynamiques pour le problème VRP et VRPTW respectivement sont présentés.

Dans la version dynamique du problème, le besoin de trouver rapidement une solution est bien plus important que dans le cas statique. C'est la raison pour laquelle les approches de réoptimisation ne sont pas adaptées. Afin de répondre à l'explosion combinatoire, il est possible de limiter l'horizon d'ordonnancement, i.e. ne considérer que les événements survenant dans une certaine période de temps dans le futur [Savelsbergh and Sol, 1998; Colorni and Righini, 2001; Mitrovic-Minic and Laporte, 2004]. Psaraftis [Psaraftis, 1988] énonce que le problème dynamique de tournées de véhicules nécessite des algorithmes en ligne qui fonctionnent en temps réel, du moment où les requêtes immédiates doivent être traitées. Il nuance le cas dynamique du cas statique en énumérant douze points clés :

1. **La dimension temps est essentielle**
   Dans le cas statique, la dimension temps peut être ou ne pas être importante ; dans le cas dynamique, le temps l'est toujours ; le système doit connaître en permanence la position de tous les véhicules, spécialement lors de la réception de requêtes.
2. **Le problème peut être ouvert**
   Le processus de résolution est temporellement borné dans le problème statique. Un véhicule quitte son dépôt avec un plan complet et immuable, partant et revenant au dépôt. Dans une configuration dynamique, le processus peut très bien être non borné, les véhicules pouvant disposer de chemins à suivre, qui changent dynamiquement au lieu d'avoir un plan complet.
3. **L'information future peut être imprécise ou inconnue**

Dans le problème statique, l'information est connue et de même qualité. Le futur dans le problème dynamique n'est jamais connu avec assurance. Au meilleur des cas, on peut en avoir une information probabiliste.

4. **Les événements à court terme sont plus importants**
   Toutes les requêtes ont le même poids dans le cas du VRP statique, alors qu'il est inadéquat d'allouer, immédiatement, des ressources pour les requêtes long-terme dans le cas dynamique ; les requêtes court-terme sont donc privilégiées.
5. **La mise à jour des informations est essentielle**
   Toutes les données utilisées par un problème dynamique sont théoriquement sujettes à des changements durant l'exécution. Il est par conséquent essentiel que les mécanismes de mise à jour des informations soient intégrés dans la méthode de résolution. Naturellement, la mise à jour des informations est sans objet dans le cas statique.
6. **Le ré-ordonnancement et la ré-affectation des décisions sont importants**
   Dans le cas dynamique, les nouvelles entrées font que les décisions prises auparavant par le système deviennent sous-optimales et une reconstruction des tournées devient nécessaire afin de répondre à la nouvelle situation.
7. **Des temps de calcul plus courts sont nécessaires**
   Dans un problème statique, le système peut attendre une longue durée pour obtenir de meilleurs résultats. Ceci n'est plus possible dans le cas dynamique, car le système doit répondre au problème courant le plus tôt possible (de l'ordre des dizaines de secondes).
8. **Éviter le renvoi infini de requêtes est essentiel**
   Il faut éviter de retarder le service d'une demande particulière indéfiniment, à cause de sa difficulté à être desservi par rapport aux autres.
9. **La fonction objectif peut être différente**
   Les objectifs traditionnels statiques tels que la minimisation de la distance totale ou la durée totale d'exécution peuvent ne pas être pertinents dans une configuration dynamique.
10. **Les contraintes temporelles peuvent être moins rigides**
    Les contraintes telles que la borne supérieure d'une fenêtre temporelle tendent à être plus flexibles dans le cas dynamique. Ceci est dû au fait que le refus d'un service à une demande immédiate - car la contrainte temporelle n'est pas satisfaite - est généralement moins attractif que de violer cette contrainte temporelle. Il est à rappeler ici que nous faisons le choix dans notre problème d'avoir des fenêtres temporelles rigides, afin de garantir une qualité de service chez le client.
11. **La flexibilité dans la variation de la taille de la flotte de véhicules est plus faible**
    Dans une configuration statique, la marge entre l'exécution des algorithmes et l'exécution des plans permet généralement des ajustements dans la taille de la flotte. Cependant, dans le cas dynamique, le gestionnaire peut ne pas avoir d'accès immédiat à des véhicules libres et opérationnels. Les implications de cela peuvent vouloir dire que quelques clients recevraient un service de plus mauvaise qualité.
12. **La gestion de la file d'attente des clients à traiter est importante**
    Si le nombre de clients demandant un service devient important, le système devient congestionné et les algorithms utilisés condamnés à donner des résultats de moindre qualité. Psarfatis note que même si l'ordonnancement des véhicules et la théorie des files d'attente sont des disciplines très étudiées, l'effort de les combiner reste limité.

Le problème avec le cas dynamique c'est que la résolution est myope puisque nous ne savons pas quelles requêtes vont être soumises au système une fois que nous avons affecté telles requêtes à

tels véhicules. Et même si nous arrivons à avoir une affectation optimale avec un sous-ensemble de requêtes et de véhicules, une nouvelle requête pourrait rendre l'ancienne affectation sous-optimale, et pourrait nécessiter dans le pire des cas un re-calcul de toutes les solutions.

Généralement, la plupart des travaux traitant du cas dynamique sont plus ou moins des adaptations directes des méthodes statiques. Dans ce contexte, les heuristiques d'insertion ont été adaptés pour travailler dans un environnement dynamique (e.g. [Madsen *et al.*, 1995; Fu and Teply, 1999; Horn, 2002; Diana, 2006]). Dans ces travaux, le critère d'insertion est la minimisation du coût additionnel de l'insertion de la nouvelle requête, critère initialement suggéré par [Solomon, 1987] comme décrit plus haut.

Les méta-heuristiques ont également été adaptées pour travailler dans un contexte dynamique (e.g. recherche locale à voisinage large [Gendreau *et al.*, 2006]). Dans [Housroum *et al.*, 2006], les auteurs proposent d'adapter les algorithmes génétiques pour traiter des VRPTW dynamiques. L'algorithme génétique proposé commence par créer une population de solutions initiales et essaie continuellement d'en améliorer la qualité. Lorsqu'un nouveau client apparaît, il est inséré dans toutes les solutions courantes dans les positions minimisant le coût additionnel.

Dans un contexte dynamique, les heuristiques d'insertion sont également combinés avec des méta-heuristiques afin d'améliorer la qualité des solutions. Zhu and Ong [Zhu and Ong, 2000] proposent une approche pour un VRP dynamique, dans laquelle un solveur central constitué de réacteurs gèrent les événements survenant dans le réseau. Lorsqu'un client apparaît, il est inséré dans la tournée d'un véhicule à la manière des heuristiques d'insertion. Après chaque insertion, une procédure d'optimisation est lancée tentant de réduire le nombre de véhicules et la distance totale parcourue, elle est répétée jusqu'à ce que la solution courante ne s'améliore plus. Les clients sont traités séquentiellement selon un ordre de priorité décroissant, fonction de leurs distances respectives et de l'ordre décroissant de leur fenêtre temporelle ouvrante.

Dans un récent travail, Kiechle *et al.* [Kiechle *et al.*, 2007] traitent un problème de transport de patients avec des ambulances dans un service d'urgences Autrichien devant traiter les urgences qui se manifestent comme des « disparitions » de véhicules qui doivent servir les demandes urgentes. Le reste des demandes (pré-planifiées) doivent être desservies par la flotte de véhicules diminuée de celui qui doit desservir la demande urgente. La méthode utilisée consiste en une heuristique d'insertion suivie d'une recherche locale en échangeant des clients (des patients) entre différentes tournées et à l'intérieur d'une même tournée. La procédure est répétée pour les nouvelles demandes (urgentes), le véhicule sélectionné pour les traiter se débarrasse des clients qui lui étaient alloués (les demandes pré-planifiées) qui doivent être ré-optimisées et allouées aux autres véhicules.

Dans [Bent and Hentenryck, 2003], les auteurs traitent un VRP dynamique et stochastique. Dans les VRP stochastiques, des données sur la distribution prévue des demandes existent et peuvent être utilisées pour guider la résolution. L'approche à plans multiples (MPA[25]) proposée génère continuellement des plans compatibles avec l'état courant des informations existantes et supprime ceux qui ne le sont plus. A tout moment, un plan privilégié est maintenu, et qui sera retenu si la situation ne change pas. Les auteurs utilisent une recherche locale afin d'améliorer leurs solutions.

3.6 Approches distribuées

Plusieurs travaux proposent de paralléliser des méthodes exactes pour la résolution des VRP et leurs variantes. L'une des raisons invoquées est de pousser les limites de taille des problèmes

[25] *Multiple Plan Approach*

pouvant être résolus par des méthodes exactes, ou de les résoudre plus rapidement [Larsen, 1999]. En effet, si la résolution prend des semaines ou des mois avant de donner une solution à un problème, il est difficile d'utiliser les résultats dans des recherches nécessitant plusieurs expérimentations, afin de calibrer les algorithmes par exemple. Mais si la résolution est parallélisée de sorte que le processus prend quelques jours ou quelques heures, il devient possible de faire tourner plusieurs tests. [Ralphs, 1995] propose un algorithme *Branch & Cut* parallèle pour le VRP et [Larsen, 1999] propose un algorithme *Branch & Price* parallèle pour la résolution du VRPTW. Ceci étant dit, l'essentiel des travaux pour des approches parallèles et distribuées ont été des adaptations distribuées d'heuristiques et de métaheuristiques et en SMA. Dans le contexte des SMA, les approches pour le VRP et ses variantes s'apparentent à des problèmes de coordination SMA, dans la catégorie de la résolution de problèmes distribuées (voir chapitre 1). Plusieurs directions ont été suivies dans la littérature afin de distribuer la résolution de ces problèmes. La configuration maître-esclave a été utilisée pour paralléliser la méta-heuristique de recherche tabou et pour une procédure de post-optimisation (à partir d'une solution initiale) en SMA. Afin de distribuer les heuristiques d'insertion, généralement dans une configuration dynamique, le protocole CNP et *Extended* CNP ont été utilisés dans lesquelles les clients s'annoncent séquentiellement, les véhicules leurs proposent des offres et le client choisit le meilleur d'entre eux. Enfin, la formation de coalitions a été présentées dans le contexte d'un problème statique, où les clients choisissent d'appartenir à la coalition maximisant une fonction de compromis.

3.6.1 Configuration maître-esclave

Dans [Gendreau *et al.*, 1999], les auteurs proposent d'adapter la recherche tabou à une configuration dynamique avec une implémentation parallèle pour le problème VRPTW. A tout moment, une solution courante est maintenue et un ensemble des meilleures solutions déjà explorées est sauvegardé dans une mémoire partagée. Les nouvelles solutions sont créées à partir de la mémoire partagée, en appliquant des méthodes de recherche locale, transférant un ensemble de clients adjacents d'une tournée vers une autre. La parallélisation est effectuée selon un schéma maître-esclave, où le maître maintient la mémoire partagée et propose les prochaines solutions initiales aux esclaves, et où les esclaves appliquent la recherche tabou sur les solutions qui leur sont proposées.

L'adaptation au problème dynamique se fait en interrompant les processus de recherche au niveau de chaque esclave à chaque arrivée d'un nouveau client, et en insérant le nouveau venu dans chaque solution de la mémoire à la manière des heuristiques d'insertion. Les solutions où le client ne peut pas être inséré sont éliminées de la mémoire. Cependant, si le client n'est insérable dans aucune solution, le système le rejette et continue à travailler avec les mêmes solutions dans la mémoire. Dans [Attanasio *et al.*, 2004], une recherche Tabu parallèle est proposée pour le TAD.

Dans [Leong and Liu, 2006], les auteurs proposent une solution multi-agent pour un VRPTW statique. Deux étapes sont effectuées. Dans la première, une solution initiale générée selon l'heuristique de Solomon est générée. Ensuite, pour chaque client et chaque véhicule est créé un agent représentant. Un agent planificateur interagit avec les deux types d'agents. Les agents client et véhicule sont informés continuellement de l'état actuel de la situation et envoient à l'agent planificateur des listes d'opérations désirées. Ces listes comprennent, pour un client, la meilleure position d'une tournée où il désire être inséré, le véhicule le plus proche (par rapport au centre de gravité de la tournée du véhicule) à la tournée duquel il veut appartenir et le client avec lequel il veut être échangé. Les véhicules proposent des listes d'opérations désirées, comprenant le meilleur client (moins coûteux à insérer) qu'il désire insérer et le meilleur couple de clients (un

de sa tournée et un autre de la tournée d'un autre véhicule) qu'il désire échanger avec un autre véhicule. L'agent planificateur choisit itérativement les meilleures propositions depuis celles proposées par les agents client et véhicule, et exécute périodiquement des procédures d'optimisation globales telles que 2-opt, afin d'améliorer l'ordonnancement des clients dans une tournée. Il essaie également d'éliminer les véhicules avec des tournées de mauvaises qualités. La qualité d'une solution est fonction du nombre de clients dans la tournée et du taux de remplissagedu véhicule.

3.6.2 Segmentation *a priori*

Le système ADART [Dial, 1995] se fonde sur une segmentation géographique *a priori* du réseau, en allouant chaque segment à quelques véhicules. Cette proposition a pour principal intérêt d'être la première à avoir envisagé un système de transport personnalisé, et totalement automatisé. Le contrôle y est local, dans les véhicules, les conducteurs n'ayant qu'à obéir aux ordres de leur ordinateur embarqué. Le service propose un service à l'avance, ou répétitif e.g. maison-travail-maison (version statique du problème) et un service *ad hoc* (version dynamique du problème).

L'auteur propose une taxonomie des système de transport à la demande. Il en existerait trois types :

1. plusieurs (emplacements) à un : il s'agit d'un service qui transporte des voyageurs depuis plusieurs endroits (n'importe quel noeud du réseau) vers un seul endroit (aéroport). C'est la version VRPTW du problème.
2. plusieurs à plusieurs : les requêtes sont composées d'un noeud de départ et d'un noeud d'arrivée. L'auteur indique que ce type de systèmes n'offre pas de grande rentabilité puisque le partage de courses entre voyageurs est plus difficile à effectuer. Il s'agit d'un service de transport à la demande tel que nous l'avons défini plus haut.
3. plusieurs à quelques-uns : il s'agit d'un service qui transporte des voyageurs depuis n'importe quel noeud du réseau vers quelques endroits spécifiques (aéroport et centre-ville par exemple). Pour l'auteur, ce type de systèmes est le plus rentable, et donc le plus intéressant à mettre en place. Puisque les possibilités de covoiturage sont plus grandes que dans le cas plusieurs à plusieurs, mais que le système est plus attractif que le cas plusieurs à un. Ce cas correspond à la version VRPTW avec dépôts multiples.

La couverture d'un réseau avec ADART se fait en divisant la zone géographique couverte en différentes sous-zones, et d'allouer plusieurs véhicules à chaque sous-zone. Pour la gestion de l'arrivée dynamique des requêtes de transport, la communication est établie en point à point entre le voyageur, ayant appelé le service par téléphone, et l'ordinateur embarqué dans le véhicule. L'appel du client aura été préalablement aiguillé vers un véhicule couvrant sa région. L'ordinateur embarqué rappelle l'utilisateur quelques minutes avant l'arrivée du véhicule son point de départ, afin de l'informer de l'arrivée imminente du véhicule. Aucune méthode de résolution n'a été formellement décrite, mais un ensemble d'intuitions ont été exposées par l'auteur. D'abord, l'ordinateur à bord se concentre sur les événements proches dans le futur. Ensuite, le conducteur est informé de la prise de décision concernant une requête le plus tard possible (lorsqu'aucune autre requête ne peut être insérée avant elle). Enfin, chaque véhicule ne considère que ses propres requêtes.

La communication inter-véhiculaire s'effectue entre véhicules de la même sous-région. Pour une requête donnée, une enchère est lancée à l'initiative du véhicule ayant reçu l'appel téléphonique et le véhicule qui a le coût minimal d'insertion est retenu. Néanmoins, l'auteur ne précise pas comment les véhicules peuvent effectuer leur *multicast* vers les autres véhicules, puisque ces

derniers ne se connaissent pas *a priori*. Durant le voyage, chaque véhicule dispose d'un ensemble de requêtes, le temps libre peut être passé à résoudre le TSP local. Ceci correspond aux méthodes heuristiques *Cluster first / route second*. Dans [Cordeau *et al.*, 2004], les auteurs observent que ADART met en avant nombre d'idées et de concepts intéressants, mais que malheureusement, il n'a jamais été implémenté.

3.6.3 Protocoles

CNP

Dans [Thangiah *et al.*, 2001] et dans [Kohout and Erol, 1999], les auteurs proposent une architecture multi-agent pour résoudre un VRP et un MDVRP pour le premier et un TAD pour le second. Le principe est le même, celui de distribuer une heuristique d'insertion parallèle, suivi d'une étape de post-optimisation. Dans [Thangiah *et al.*, 2001], les clients sont traités séquentiellement, diffusés à tous les véhicules, ces derniers proposent des offres et le meilleur est retenu. Dans la seconde étape, les véhicules échangent des clients afin d'améliorer leurs solutions, chaque véhicule connaissant tous les autres agents véhicule du système. Lors du passage à la version MDVRP, seuls les agents véhicules connaissent leurs dépôts, le système et l'algorithme suivis restent les mêmes. Et étant concurrents, les auteurs envisagent d'appliquer des heuristiques et des méta-heuristiques différentes pour chaque véhicule, sans changer l'architecture.

In-Time [Kohout and Erol, 1999] est un système réalisé dans le cadre d'une application réelle d'ordonnancement de véhicules d'une société de transport en aéroport. Le système est composé d'agents client et d'agents véhicule. L'agent *GUI* est responsable de la récupération des requêtes de transport client. L'agent client s'annonce et tous les agents véhicule calculent son coût d'insertion dans leurs itinéraires et renvoient des offres à l'agent client. Ce dernier sélectionne l'offre la moins chère et pose sa requête à l'agent véhicule associé. Si l'offre est toujours valable, le client est inséré dans la tournée de ce véhicule et il est informé que le contrat est accepté ; sinon, si l'offre est désormais plus chère probablement à cause de l'insertion d'un autre client entre temps, l'agent véhicule propose son nouveau prix, et l'agent client réitère ce processus jusqu'à ce qu'il soit ordonnancé.

L'algorithme (voir Algorithme 3) commence par localiser une position où insérer le point de cueillette et continue à la recherche dans l'ordonnancement du véhicule à la recherche d'une position où insérer le point de livraison, en respectant les implications du point de cueillette. Le coût final de l'insertion d'un client est une somme pondérée des coûts d'insertion des points de cueillette et de livraison. Comme dans toute heuristique d'insertion, l'ordre d'arrivée des clients est important dans cet algorithme. En l'absence d'une amélioration locale, cet algorithme agent présente des résultats de mauvaise qualité quand les clients sont ordonnancés dans un ordre aléatoire.

Le mécanisme proposé par les auteurs représente en fait une recherche locale distribuée. En effet, ils permettent à un client de demander d'une manière stochastique d'annuler son ordonnancement actuel et de se ré-annoncer au système dans l'objectif d'avoir un meilleur marché. Le processus peut être décrit comme suit :

1. chaque agent véhicule a un intervalle de ré-ordonnancement défini selon une méthode stochastique. Cette période est définie quand un agent véhicule est initialisé et est réinitialisée en une nouvelle valeur après chaque ré-ordonnancement.
2. durant un intervalle de ré-ordonnancement, un agent véhicule choisit une variable aléatoire dont la borne supérieure est le nombre de clients dans le véhicule pouvant être ré-ordonnancés.

3. Les clients sont choisis et chacun est sélectionné pour être ré-ordonnancé avec une probabilité qui tend vers zéro lorsque le temps de début de service du client approche. Expérimentalement [Kohout and Erol, 1999], la probabilité de ré-ordonnancement atteint zéro 30 minutes avant le début de service du client.

Algorithme 3 Algorithme d'insertion agent

```
Créer un itinéraire
tant que il reste des clients non ordonnancés faire
  Récupérer le coût dinsertion du prochain client dans chacun des itinéraires actifs des véhicules
  si le client ne peut être inséré dans aucun des itinéraires des agents véhicule alors
    créer un nouvel agent véhicule et un nouvel itinéraire
  sinon
    Insérer le client dans litinéraire de plus faible
  fin si
fin tant que
```

CNP étendu

MARS [Fischer *et al.*, 1994; Fischer *et al.*, 1995] modélise un ordonnancement coopératif dans une compagnie maritime sous la forme d'un SMA. La solution du problème de l'ordonnancement global émerge de la prise de décision locale et des stratégies de résolution de problèmes. Le système utilise une extension du *Contract Net Protocol*(CNP) [Davis and Smith, 1983] et montre qu'il peut être utilisé pour obtenir de bonnes solutions initiales aux problèmes complexes d'allocation de ressources.

Deux catégories d'agents existent au sein du système : les sociétés de transport maritime (SCA[26]) et les camions (TA[27]). Il s'agit d'un VRPTW dynamique avec possibilité de diviser une demande d'un client sur plusieurs véhicules (chacun transportant une sous-partie de la quantité totale demandée). Le SMA profite d'une structuration organisationnelle *a priori*, puisque chaque TA est associé à une société particulière. Un SCA n'effectue pas l'ordonnancement des transports lui-même, il ne fait que transmettre les demandes de transport à des TAs. Ces derniers maintiennent des plans locaux d'ordonnancement.

L'interaction des agents au sein d'une société maritime (appelée coopération verticale) est totalement coopérative. Ceci veut dire qu'un agent TA donné va accepter de coopérer même s'il n'en tire pas profit. La coopération entre les agents sociétés maritimes (SCA) est appelée coopération horizontale.

– Trouver une solution initiale
  Si une commande de transport est soumise à un agent SCA par un client (qui peut être aussi un autre SCA), il doit calculer une offre pour la satisfaire. Afin de déterminer les coûts, le SCA transfère la commande à ses TAs. Chaque TA A_i, $1 \leq i \leq n \in \mathbb{N}$ calcule une offre $(A_i, cout(T_i, o), a)$ où T_i est l'itinéraire courant de A_i et a la quantité de o de transport que peut satisfaire A_i.
  Pour chaque demande de transport o annoncée par un SCA à ses TAs, elle reçoit un ensemble d'offres :
  $$B = \{(A_1, c_1, a_1), \ldots, (A_n, c_n, a_n)\}, n \in \mathbb{N}$$

[26] *Shipping Company Agent*
[27] *Truck Agent*

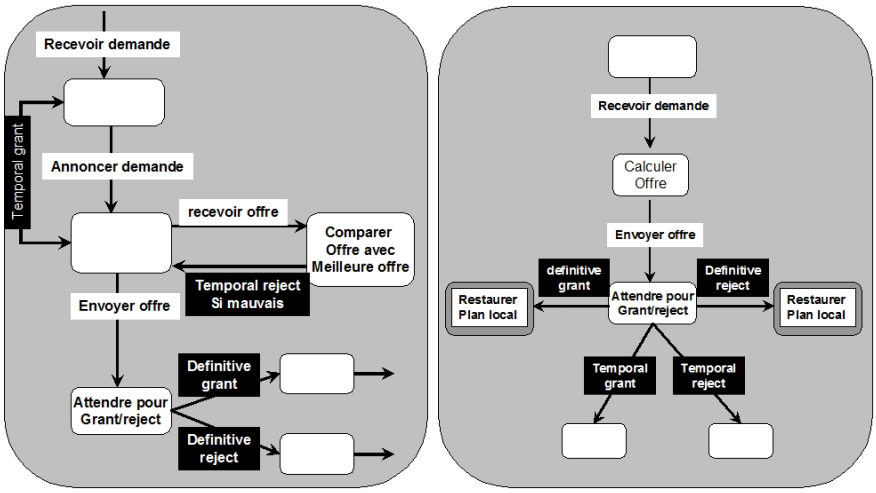

FIG. 3.4 – Protocole ECNP (à droite *manager*, à gauche offreur)

où c_i spécifie le coût engendré pour A_i par l'exécution de la quantité a_i de la demande o. Le SCA sélectionne $(A_{min}, c_{min}, a_{min}) \in B$ avec :

$$\forall (A, c, a) \in B : \frac{c_{min}}{a_{min}} \leq \frac{c}{a}$$

et envoie un accord à l'agent TA A_{min}, lui notifiant qu'il lui sera alloué la quantité a_{min} pourvu que SCA reçoive un accord pour o du client.
Le CNP basique pose des problèmes si les tâches excèdent la capacité d'un seul TA, i.e. $a_{min} -$ *quantité à transporter*$(o) < 0$. Dans ce cas, le SCA a un « problème de sac à dos » à résoudre, qui est lui-même NP-Difficile [Fischer *et al.*, 1994]. Pour surmonter ce problème, les auteurs ont décentralisé la décomposition des tâches en développant une extension du CNP : ECNP (*Extended* CNP). Dans ce nouveau protocole, les deux actes de langages *grant* et *reject* sont éclatés en : *temporal grant*, *temporal reject*, *definitive grant* et *definitive reject* (voir figure 3.4).
Dans ENCP, le manager (SCA) annonce une demande ou une commande à ses TAs. Il reçoit les offres correspondantes et choisit la meilleure comme décrit plus haut. Au meilleur TA est envoyé un *temporal grant* et aux autres des *temporal rejects*. Si la meilleure offre ne couvre pas le service complet demandé, la partie qui reste est ré-annoncée par le SCA. Cette procédure est répétée jusqu'à ce qu'il y ait un ensemble d'offres qui couvre la totalité de la commande initiale o. A partir de cet ensemble, le SCA calcule une offre qu'il soumet au client. A partir de la réponse du client, il envoie un *definitive grant* (resp. *definitive reject*) à tous les TAs qui avaient reçu des *temporal grants* (resp. *temporal reject*).
Quand un TA reçoit un *temporal grant* pour la première fois, il doit sauvegarder une copie de sa situation actuelle, i.e. le plan valide courant, pour qu'il puisse le restaurer dans le cas où il recevrait un *definitive reject* ultérieurement. Si un TA reçoit un *definitive grant* pour une commande, il enlève la copie créée précédemment et migre vers le nouveau plan ; et s'il reçoit un *definitive reject*, il restaure son ancien plan.

– Le *simulated trading* (le commerce simulé)
  En utilisant l'ECNP, le SCA distribue les demandes reçues vers son ensemble de TAs. Cependant, parce que la situation change à cause de la réception de nouvelles demandes reçues, et parce que les TAs en restent aux décisions prises dans le passé, la solution

trouvée n'est pas forcément la meilleure en prenant en compte les changements survenus. Pour pallier cette carence, les auteurs utilisent un mécanisme d'enchères appelé simulated trading (ST) de [Bachem *et al.*, 1996]. L'idée principale est de laisser le SCA simuler une bourse de valeurs(*stock exchange*) où ses TAs peuvent offrir leurs demandes courantes à un certain « prix de revient » et en acheter d'autres à un « prix d'insertion ». En obtenant des offres d'achat et de vente de la part de ses TAs, le SCA essaie de trouver un échange de demandes qui puisse optimiser la solution globale (voir [Fischer *et al.*, 1994]).

– Coopération horizontale
  En raison de la distribution spatio-temporelle des demandes client, la coopération inter-SCA (appelé coopération horizontale) peut être une opération bénéfique en vue d'optimiser l'utilisation des capacités de transport, but ultime pour un SCA. Bien qu'il serait possible d'utiliser l'approche du *simulated trading* aussi pour une optimisation globale, en échangeant les demandes entre SCAs, les auteurs affirment qu'une telle approche est inadéquate et ce pour les raisons suivantes :
  – Au sein d'une compagnie maritime, le processus ST peut être aisément installé ; cependant, même dans ce cas, les plans courants des TAs doivent être gelés et chaque TA évoluant dans ce processus peut seulement accepter de nouvelles demandes après la fin du processus de ST. Un ST inter-SCA pourrait demander un immense effort de synchronisation et de considérables coûts en communications.
  – Contrairement à la coordination entre une compagnie et ses TAs, la coopération inter-compagnies est un processus de pair à pair (peer to peer) où une solution ne peut être trouvée que si tous les participants acceptent et où les conditions de la solution doivent être négociées entre les compagnies. Ainsi, il n'existe pas d'autorité de décision globale pour contrôler le processus de négociation.
  – Les SCAs vont se comporter plus individuellement que les TAs dans la négociation. Par conséquent, l'optimalité globale de l'ordonnancement global qui émerge des résolutions locales de problèmes n'est plus le critère clé qui guide la négociation. Les SCAs vont plutôt essayer de maximiser leur profit personnel en vendant et en achetant les demandes entre eux. L'individualité est aussi la raison pour laquelle, en général, les informations concernant les demandes, les coûts et les prix nécessaires pour l'algorithme de ST, ne sont pas censés être disponibles publiquement.

  Les auteurs choisissent un modèle fournissant une bourse pour les demandes de transport entre les SCAs, organisée comme un tableau noir dans lequel les SCAs peuvent déposer les demandes qu'ils veulent vendre en spécifiant le prix qu'il désire obtenir. Si un autre SCA veut acheter une commande, une négociation bilatérale entre eux est engagée déterminant le prix actuel qui doit être payé pour la commande d'une manière décentralisée.

3.6.4 Formation de coalitions

Dans [Kefi and Ghedira, 2004; Boudali *et al.*, 2004; Boudali *et al.*, 2005], les auteurs proposent un modèle multi-agent - appelé coal-VRP - pour traiter le VRPTW. Ils considèrent un VRPTW euclidien et statique. Coal-VRP est un système multi-agent fondé sur l'approche de formation de coalitions. Une coalition consiste à partitioner les agents capables de communiquer et de négocier en groupes, appelés coalitions [Sichman, 1998]. Dans [Vauvert and El Fallah-Seghrouchni, 2000b], une coalition est définie comme une organisation à court terme fondée sur des engagements spécifiques et contextuels, qui permet aux agents de coexister tout en profitant de leurs compétences respectives. Le principe de Coal-VRP est divisé en deux étapes. Lors de la première étape, les coalitions possibles sont créés entre les agents, en prenant en compte leur

contraintes spatio-temporelles respectives. La formation de coalition est limitée à un voisinage spatial α afin de limiter l'explosion combinatoire du nombre de coalitions envisageables. Lors de la deuxième étape, les agents client essaient de créer leurs coalitions parmi leurs coalitions possibles. Les agents négocient leurs coalitions par échange de messages et la coalition à laquelle un agent choisit d'appartenir est celle qui minimise une fonction de compromis égale à :

$$\frac{\text{distance totale du plan de la coalition}}{\text{nombre de clients dans la coalition}}$$

La complexité du processus de résolution est corrélée avec la valeur de α et les résultats deviennent significatifs lorsque sa valeur est élevée. Le schéma de communication est donc en point à point et la recherche de partenaires de coalition se fait par diffusion dans cette proposition.

3.7 Conclusion

Dans ce chapitre, nous avons présenté le problème de transport à la demande comme variante du problème de tournée de véhicules. La grande majorité des travaux effectués dans ce domaine ont été concentrés sur les heuristiques et les méta-heuristiques pour résoudre des problèmes statiques. La variante du problème avec contraintes temporelles a reçu beaucoup d'attention, et les travaux traitant d'extensions de ce problème, telles que le transport à la demande, adaptent en général des méthodes déjà utilisées dans le problème avec fenêtres temporelles basiques. Le problème dynamique est moins étudié, et les approches distribuées pour résoudre le problème dynamique encore moins.

Les heuristiques d'insertion sont les méthodes les plus utilisées dans tous les travaux de la littérature. Elles sont utilisées dans les approches à deux phases afin de trouver des solutions initiales, ou des populations de solutions initiales (dans les algorithmes génétiques). Dans les approches distribuées, surtout celles qui traitent des problèmes dynamiques, elles sont utilisées dans une version distribuée, généralement avec le protocole CNP, afin de choisir le meilleur véhicule pour le client considéré.

Deuxième partie

Contributions

Chapitre 4

Modèle de coordination Acios

Sommaire

4.1 Introduction

Les modèles de coordination orientée-données présentés dans le chapitre 1 proposent des solutions pour la coordination de processus séquentiels à travers l'utilisation d'un espace de données partagé. Les principaux avantages relatifs à ces modèles de coordination, et qui ont un intérêt pour la conception de SMA, sont l'interaction anonyme et le style de communication découplé dans le temps et dans l'espace. Appelé communication générative, ce mode de communication permet de concevoir des systèmes ouverts, où les processus peuvent rejoindre et quitter le système librement, puisque tous ont un interlocuteur commun, l'espace de données, et n'ont pas à maintenir et à mettre à jour les adresses des autres.

Cependant, en l'état, l'utilisation de ces modèles dans le contexte des SMA se heurte à plusieurs verrous. D'une part, un système adhérant à un modèle de coordination orientée-données est constitué d'une composition de processus et d'un espace de données. Or, le comportement d'un agent ne peut être réduit à un processus séquentiel, ni le SMA à une composition de ces comportements. D'autre part, la structure de données de référence dans ces modèles est celle des tuples ; lors de la conception et l'implémentation d'un SMA, le programmeur doit aligner la représentation des données échangées entre agents à cette structure, ce qui augmente considérablement l'effort d'expression de besoins complexes impliquant des opérateurs et des variables.

Notre proposition est structurée en deux parties complémentaires, fondées sur ce chapitre et le suivant. Ce chapitre définit un modèle de coordination appelé Acios[28] (*Agent Contextual Interaction in Open Systems*), et le chapitre suivant lui associe un langage, appelé Lacios (*Language for Agent Contextual Interaction in Open Systems*). La présentation du modèle se concentre sur la présentation des aspects conceptuels relatifs au modèle tels que la structure de données, l'appariement et la sécurité. Nous y présentons des éléments syntaxiques du langage ainsi que les principales définitions s'y rattachant. Le langage de coordination Lacios fait l'objet du chapitre suivant, nous y complétons la définition de la syntaxe du langage, et nous lui associons une sémantique et une implémentation.

Dans Acios, les agents du SMA ont un état, observable de l'espace de données, qui conditionne leur interaction dans le SMA. Dans ce modèle, nous utilisons une structure de données comportant des couples *propriété-valeur*, qui permet une représentation des données échangées entre les agents plus riche que les tuples et un mécanisme d'appariement plus expressif que les *templates*, fournissant à un agent le moyen d'exprimer une interaction complexe. Le mécanisme d'appariement permet à un agent de conditionner son interaction par l'état de plusieurs entités de l'espace de données (ce que nous appelons un contexte d'interaction). La sécurité est garantie par un mécanisme de règles de sécurité et de restrictions de perception et de réception associées aux données ajoutées dans l'espace de données. Les restrictions de perception et de réception sont définies par les agents de telle manière que chaque agent gère l'observabilité de ses propres données.

Ce chapitre est structuré comme suit. Dans la section 4.2, nous donnons les motivations de notre travail. Dans la section 4.3, nous présentons l'exemple des agents voyageurs dans une gare, qui sert d'illustration aux notions que nous introduisons. Nous utilisons cet exemple tout au long de ce chapitre et du chapitre suivant. La section 4.4 présente les définitions de base du modèle Acios. Il s'agit du modèle basique qui est complété au fur et à mesure que nous avançons dans la description de notre proposition. La section 4.5 augmente le modèle avec les constructeurs linguistiques permettant la prise en compte du contexte d'interaction. La section 4.6 enrichit la syntaxe pour que les agents puissent s'assurer de la sécurité de leurs données. Nous discutons le modèle Acios par rapport à ceux de la littérature dans la section 4.9. La section 4.10 conclut le chapitre.

4.2 Motivations

Ce travail est en continuité et complémentarité des travaux effectués au sein de notre équipe concernant la modélisation des interactions dans les SMA [Balbo, 2000b; Balbo, 2004; Zargayouna *et al.*, 2006b]. Dans nos travaux, les interactions dans un SMA sont vues comme une

[28]Le travail relatif à ce chapitre est effectué au sein de l'équipe du projet Modèles d'interaction (pôle Agents Intelligents et Modèles Coopératifs) au laboratoire Lamsade - Université Paris-Dauphine

mise en correspondance des besoins d'agents ne se connaissant pas *a priori*. Ces besoins sont exposés par des propriétés observables publiées, par exemple, dans l'environnement du SMA. Ce travail de thèse partage ces mêmes objectifs, et propose le modèle de coordination Acios, le langage Lacios utilisant une formulation en algèbres de processus, et une implémentation du langage permettant la réalisation d'un SMA adhérant au modèle en garantissant le respect de sa sémantique.

Afin d'interagir avec les autres agents du SMA lorsque le nombre d'agents du système est relativement stable, il est pertinent que les agents du système gèrent des carnets d'adresses des autres agents. Ces adresses peuvent être mises à jour des nouveaux arrivants et des agents ayant quitté le système. Des compétences et des préférences peuvent être également associées aux adresses des agents. Ainsi, lorsqu'un agent désire communiquer avec un autre ayant une capacité ou une préférence particulière, il lui suffit d'accéder à son propre carnet d'adresses, et de découvrir les agents ayant cette capacité ou cette préférence, pour ensuite leur adresser ses messages.

Si les capacités et les préférences des agents changent dynamiquement, la diffusion de messages semble être la solution appropriée. En effet, chaque agent envoie ses messages à tous les agents du SMA, et ces derniers peuvent lui répondre s'ils sont intéressés par sa demande. Cependant, lorsque les agents quittent et rejoignent le SMA librement et fréquemment, i.e. dans un SMA ouvert, les connaissances des agents des autres deviennent rapidement obsolètes, et des mises à jour fréquentes s'avèrent nécessaires afin de garantir que la connaissance des agents soit cohérente avec l'état réel du système.

C'est dans ce dernier contexte, celui des SMA ouverts, que l'adoption d'un modèle de coordination orientée-données est pertinente. En effet, il est admis que ces modèles sont parmi les plus pertinents dans le contexte de systèmes ouverts [Ciancarini, 1990]. Comme nous l'avons montré dans le chapitre 1, les modèles de coordination orientée-données proposent un seul interlocuteur pour les agents du système, l'espace de données partagé, la connexion entre deux agents étant réalisée par son biais, les agents n'ont donc pas à maintenir une connaissance des autres agents. Le second aspect des SMA ouverts est relatif à l'interaction de tout le SMA avec un système externe. Si le passage par un espace de données partagé résout le problème de l'entrée et sortie d'agent du SMA, il ne permet pas d'exprimer l'action d'un autre système, non modélisé, sur le SMA.

Cependant, les modèles de coordination de la littérature considèrent un espace de données sur lequel agissent des processus, et non des agents. Les principales différences de notre modèle d'avec ceux de la littérature sont que : d'abord, les agents dans Acios ont un état qu'ils peuvent utiliser dans la définition de leurs situations d'interaction. Par exemple, un agent peut décider de ne recevoir des requêtes que s'il est libre, i.e. n'effectuant aucune autre action. D'autre part, les agents Acios ne sont pas des processus séquentiels, mais une composition de processus, notamment parallèle. Ainsi, un agent ne se bloque pas nécessairement dans l'attente des informations recherchées, mais peut lancer un processus à la recherche de cette information alors que ses autres processus continuent leur traitement. De plus, le modèle Acios offre une structure de données et un mécanisme d'appariement plus en concordance avec des besoins interactionnels complexes tels que l'interaction avec une base de données ou la manipulation d'objets en programmation orientée-objet. Enfin, il permet à un agent de gérer la sécurité de ses données sans les cloisonner dans des espaces de données multiples.

La modélisation de l'action d'un système externe sur le comportement des agents du SMA est réalisée avec l'introduction des variables dans le comportement d'un agent. Une variable désigne une valeur découverte en cours d'exécution, instanciée par un système externe interagissant en point à point avec l'agent considéré.

4.3 Scénario des agents voyageurs dans une gare

Dans cette section, nous introduisons le scénario des agents voyageurs dans une gare pour illustrer les concepts introduits par le modèle Acios. Il sera évoqué tout au long de ce chapitre et le suivant afin de motiver l'introduction de nouveaux concepts et pour illustrer la syntaxe et la sémantique du langage. Il s'agit d'une gare où des agents humains, des services de planification, de réservation, de paiement etc. et des sources d'information coexistent. Nous considérons des agents humains représentant des voyageurs, des trains qui génèrent des informations sur les départs, les arrivées, les retards etc. et deux agents additionnels : café et salle d'attente.

Nous envisageons que tous les voyageurs désirant utiliser les services à l'intérieur de la gare disposent d'un moyen de communication, tel qu'un téléphone portable avec un navigateur, un PDA ou un ordinateur portable avec une technologie d'accès à distance (*bluetooth*, *Wi-Fi*, etc.). Tous les voyageurs et services de la gare disposent d'agents les représentant dans le SMA. Ces agents interagissent par échange de données dans un espace partagé à la manière des modèles de coordination orientée-données.

4.4 Modèle basique

Dans cette section, nous définissons un modèle basique pour notre proposition. Nous introduisons la structure de données que nous avons retenue, ainsi qu'une syntaxe minimale du langage et ses primitives.

4.4.1 Structure de données

La structure de tuples sur laquelle se fondent les modèles de coordination à la Linda ne nous satisfait pas, car elle dispose seulement de deux leviers afin de discriminer un tuple : l'ordre des champs et leur type. La structure de données que nous retenons est celle d'un système d'information standard, où les données sont représentées par un ensemble de couples *propriété-valeur*. En effet, lorsqu'on modélise un système d'information (e.g. [Komorowski *et al.*, 1998]), on s'appuie généralement sur cette représentation (un enregistrement dans une table par exemple). Dorénavant, nous désignons toute donnée échangée dans le système par le terme « entité », et l'espace de données est désigné par le terme « environnement » du SMA, noté Ω_{ENV}.

Considérons une entité *voyageur*. Cette entité pourrait être décrite ainsi :

$$\{id \leftarrow \text{“}v1\text{”}, age \leftarrow 25, fumeur \leftarrow \text{vrai}, budget \leftarrow 120\}$$

Chaque donnée échangée dans le système est décrite par une « description », i.e. un ensemble de couples *propriété*$\leftarrow$ *valeur*, et toutes les propriétés dans le langage sont typées.

Définition Types

Les types du langage sont définis comme $type_1, \ldots, type_{nbt}$. Chaque $type_i$ est un ensemble tel que $\forall (i,j) \in \{1, \ldots, nbt\}^2, i \neq j, type_i \cap type_j = \{\text{nil}\}$

Notation 1 *Nous définissons l'ensemble des valeurs supportées par le langage comme* $\mathcal{T} = \bigcup_{i=1}^{nbt} type_i$.

Remarque 1 *Nous supposons que les types du langage sont disjoints deux à deux, en dehors de* nil*, qui appartient à tous les types du langage. Ceci est nécessaire afin de pouvoir, à partir d'une valeur, déduire le type qui lui est associé, et ainsi vérifier la validité syntaxique d'une expression.*

Remarque 2 *Nous supposons l'existence du type* booléen *dans le langage, i.e.* $\exists i \in \{1, \ldots, nbt\}$, $type_i = \{\text{vrai}, \text{faux}, \text{nil}\}$

La valeur particulière nil a un double emploi dans notre syntaxe. D'une part, elle représente toute erreur sémantique du langage. Comme nous le verrons dans le chapitre suivant, lors de l'évaluation d'une expression syntaxiquement valide, si elle génère une erreur sémantique, elle est évaluée à nil. D'autre part, elle représente l'absence d'une propriété. En effet, une propriété dont la valeur est égale à nil est considérée comme inexistante. Les propriétés sont utilisées afin de décrire chaque entité du SMA, et chaque propriété est définie par son type.

Définition Propriétés

$\mathcal{N}$ est l'espace de propriétés, il s'agit d'un ensemble dénombrable de propriétés. Une propriété $\pi \in \mathcal{N}$ est définie par un type $type(\pi) \in \{type_1, \ldots, type_{nbt}\}$.

Notation 2 *Nous notons* $unknown_\pi$ *une valeur de type* $type(\pi)$ *mais qui n'a pas de valeur. Par exemple* $unknown_{age}$ *est une propriété de type* $type(age) = \mathbb{N}$*, mais dont la valeur est (temporairement) inconnue.*

Une description est une collection de propriétés avec leur valeur correspondante.

Définition Descriptions

DS est l'ensemble des descriptions. Une description est une fonction faisant correspondre des propriétés à des valeurs, i.e. $d \equiv \{\pi \leftarrow v_\pi \mid v_\pi \in type(\pi)\}_{\pi \in \mathcal{N}}$. La correspondance est omise quand $v_\pi = \text{nil}$. Nous utilisons $d(\pi)$ afin d'accéder à la valeur v_π. Pour chaque description, l'ensemble de propriétés $\{\pi \mid d(\pi) \neq \text{nil}\}$ est fini.

Ainsi donc, une propriété dont la valeur est nil désigne une propriété non définie, i.e. qui n'existe pas. Une description représente un ensemble fini de couples *propriété-valeur* tel que *valeur* est différente de nil.

Définition Entités

Ω est l'ensemble des entités du SMA. Chaque entité ω a une description comme décrite plus haut notée d_ω. La valeur de la propriété π de l'entité ω est notée $d_\omega(\pi)$.

Remarque 3 *Nous supposons l'existence du type* référence *dans le langage, une variable de type référence désigne une entité dans* Ω*, i.e.* $\exists i \in \{1, \ldots, nbt\}$, $type_i = \Omega \cup \{\text{nil}\}$.

Prenons l'exemple des voyageurs dans une gare. Une entité (voyageur) v_1 peut avoir la description suivante :

$d_{v_1} \equiv \{id \leftarrow$ "$v1$", $catégorie \leftarrow$ "$voyageur$", $fumeur \leftarrow$ vrai, $budget \leftarrow 120$, $destination \leftarrow$ "$Rennes$", $train \leftarrow unknown_{train}\}$

Une entité train t peut avoir la description suivante :

$d_{t_1} \equiv \{id \leftarrow$ "t_1", $catégorie \leftarrow$ "$train$", $capacité \leftarrow 125$, $destination \leftarrow$ "$Rennes$", $prix_{ind} \leftarrow 80$, $prix_{groupe} \leftarrow 50\}$,

avec $prix_{ind}$ désignant le prix pour une seule personne, et $prix_{groupe}$ désignant le prix pour un groupe de personnes. Par convention, un groupe est formé d'au moins trois personnes.

Les types définis pour cet exemple ne sont pas détaillés car intuitifs, par exemple, $type(fumeur = booléen, type(age) = \mathbb{N}^*_+ \cup \text{nil}$, etc.

Dans Acios, un agent est une entité, il a donc une description. Dorénavant, nous distinguons entre les agents et les objets (des entités non agents). Soit $\mathcal{A}$ l'ensemble des agents et $\mathcal{O}$ l'ensemble des objets. Les agents sont les entités actives du système. En effet, outre leur description, ils ont également un comportement, que nous introduisons ultérieurement. Les agents ajoutent, lisent et extraient des objets de l'environnement, comme dans tout modèle de coordination orientée-données.

4.4.2 Expressions et appariement

Reprenons notre exemple des agents voyageurs dans une gare. Nous désirons décrire le comportement des agents voyageurs, de telle sorte que chaque agent voyageur commence par ajouter un objet le décrivant dans l'environnement, i.e. un objet ayant la même description (couples *propriétés-valeurs*) que lui. S'agissant d'un comportement générique, i.e. adopté par n'importe quel agent voyageur, il n'est pas possible de créer cet objet en le décrivant par des couples *propriété-valeur*, puisque ces valeurs sont différentes selon l'instance d'agent considéré. En l'occurrence, nous désirons associer aux propriétés de l'objet généré les propres propriétés de l'agent, et non les valeurs de ces propriétés. Dans cet exemple, l'agent voudrait générer un objet o ayant la description suivante : $d_o \equiv \{age \leftarrow age, fumeur \leftarrow fumeur, budget \leftarrow budget, destination \leftarrow destination, train \leftarrow train\}$, avec $age, budget, fumeur, destintion, train$ à droite du signe d'affectation désignant les propres propriétés de l'agent.

Plus généralement, un agent doit pouvoir créer des descriptions, sans se circonscrire à des couples *propriété-valeur*, mais avoir la possibilité d'utiliser des expressions au lieu des valeurs, utilisant des combinaisons d'opérateurs, de propriétés et de valeurs dans la descriptions des objets qu'il génère. Par exemple, l'agent v_1 pourrait ne pas publier son budget réel mais seulement $budget-20$. *In fine*, une expression est destinée à être transformée en une valeur typée, lors de son évaluation. Afin d'atteindre cet objectif, nous commençons par introduire la notion d'opérateur.

Définition Opérateurs

Chaque opérateur *op* du langage est défini par :
- (i) *arité*(*op*) Le nombre de paramètres de l'opérateur,
- (ii) $par(op) : \{1, \ldots, \text{arité}(op)\} \rightarrow \{1, \ldots, nbt\}$, $par(op)(i)$ donne l'indice du type du $i^{ème}$ paramètre de l'opérateur *op*,
- (iii) $ret(op) \in \{1, \ldots, nbt\}$, l'indice du type de la valeur resultant de l'évaluation de *op*.

Par exemple, soit $type_1 = booléen \cup \{\text{nil}\}$. L'opérateur `et` est défini comme suit :
$arite(\texttt{et}) = 2$, $par(\texttt{et})(1) = par(\texttt{et})(2) = 1$ et $ret(\texttt{et}) = 1$.

Notation 3 *Les opérateurs booléens* `et`, `ou` *et* `not` *seront dorénavant notés* $\wedge, \vee, \neg$, *respectivement.*

Les expressions sont introduites comme une alternative à l'utilisation exclusive de valeurs dans la description des objets et des agents. Une expression peut être simplement une valeur, mais aussi une propriété, auquel cas, elle se réfère à une propriété de l'agent qui est en train de l'évaluer, comme dans l'exemple ci-haut. Si une propriété de l'agent est de type *référence*,

et qu'il désire accéder à une valeur de l'une de ses propriétés, il peut le faire en utilisant un point, comme ceci : $\pi.e$ (π étant une propriété se référant à l'entité considérée). Par exemple, supposons que l'agent v_1 a une propriété *adr* de type référence, qui pointe vers un objet *adresse*, décrit ainsi :

$d_{adresse} \equiv \{numéro \leftarrow 54, rue \leftarrow$ "av. de Choisy"$, ville \leftarrow$ "$Paris$"$\}$.

Si l'agent v_1 désire accéder à la propriété *ville* de son adresse (la référence *adr* dans l'exemple), il peut le faire ainsi : *adr.ville*, qui sera évaluée à "Paris". Une expression peut être également un opérateur ayant comme arguments des expressions. Par exemple, v_1 peut vérifier s'il rentre chez lui ainsi : $adr.ville = destination$.

Notation 4 *Pour alléger la présentation, nous utilisons dorénavant la notation précédente pour les opérateurs binaires, i.e. e op e′ au lieu de* $op(e, e')$.

Remarque 4 *Par abus de langage, les mêmes opérateurs d'égalité* $(=, \neq)$ *et de comparaison* $(>, <, \leq, \geq)$ *sont définis avec des paramètres de types différents.*

Définition Expressions

Exp est l'ensemble des expressions. Une expression $e \in Exp$ est générée via la grammaire du tableau 4.1.

$e ::=$ nil	
$\mid v$	, avec $v \in \mathcal{T} \backslash$nil
$\mid \pi$	, avec $\pi \in \mathcal{N}$
$\mid op(e, \ldots, e)$	, avec *op* un opérateur du langage, et nil n'apparaît dans aucun e
$\mid \pi.e$	, avec $\pi \in \mathcal{N}$ et $type(\pi) = \Omega$

Tab. 4.1 – Syntaxe d'une expression

Comme nous le voyons dans la définition précédente, nous faisons le choix de considérer la présence de nil comme paramètre d'un opérateur comme une erreur syntaxique. Nous verrons lors de la définition de la sémantique des expressions qu'un paramètre évalué à nil fait que l'opérateur est également évalué à nil. Dans ce cas, il s'agit d'une erreur sémantique.

Il nous reste maintenant à définir les « modèles d'appariement », l'équivalent des *templates* dans Linda. Rappelons que l'appariement associatif dans Linda est fondé sur l'ordre des champs et la correspondance valeur-valeur ou valeur-type. Utilisant une structure de données plus riche, nos « *templates* » ne doivent pas se limiter à un appariement à la Linda.

Dans son modèle ESAC[29], Balbo a proposé la notion de *filtre* publié dans l'environnement, formé d'une conjonction de conditions [Balbo, 1999; Balbo, 2000a; Balbo, 2000c; Balbo, 2001; Balbo, 2004]. Au cours des développements ultérieurs du modèle, nous nous sommes fondés sur un langage spécifique pour l'appariement, développé en Analyse de Données Symboliques (ADS) [Bock and Diday, 2000], utilisé afin de décrire symboliquement une collection d'objets. Les objets analysés en ADS sont décrits par des couples *propriété-valeur*, nous avions donc choisi

[29]Environnement comme Support Actif aux Communications

d'utiliser ce même langage afin de construire un modèle d'appariement [Zargayouna *et al.*, 2006a; Saunier *et al.*, 2006; Zargayouna, 2006]. Un filtre f était défini comme une conjonction d'objets symboliques, i.e. des conditions sur les valeurs prises par les propriétés. Néanmoins, ayant introduit la notion d'expression, nous pouvons aisément exprimer un filtre sans changer de notation. Pour ce faire, il suffit d'augmenter la syntaxe des expressions d'un mot clé spécifique (nous avons choisi *that* comme mot clé), pour spécifier que cet objet n'est pas connu de l'agent, mais qui sera découvert en cours d'exécution, lors de l'appariement avec un objet de l'environnement. La syntaxe d'une expression devient donc :

$$\begin{array}{|l|}\hline e ::= \ldots \\ \mid that.e \\ \hline\end{array}$$

Par exemple, prenons l'expression suivante :
$that.destination =$ "$Rennes$" $\wedge\, that.prix_{ind} < budget$

Dans cette expression, *that* désigne un objet, inconnu pour le moment, qui doit avoir pour destination "Rennes" et un prix individuel inférieur au budget de l'agent pour que l'expression soit évaluée à vrai. Si l'expression était évaluée avec l'objet t_1 décrit plus-haut, l'expression serait évaluée à vrai.

En considérant les définitions présentées jusqu'ici, nous pouvons déterminer si une expression est bien typée ou non, et si elle l'est, nous pouvons déduire son type (à l'exception de nil, qui appartient à tous les types du langage). Nous pouvons ainsi vérifier la correction syntaxique d'une expression.

Définition Expressions bien typées et leurs types

Une expression bien typée e et son type $type(e)$ sont définis d'une manière inductive suivant les règles suivantes :
- $e =$ nil est une expression bien typée ; $type$(nil) est non défini
- $e = v$ est une expression bien typée ; $type(e) = type_i$, avec $v \in type_i$, étant donné un certain $i \in \{1, \ldots, nbt\}$
- $e = \pi$ est une expression bien typée ; $type(e) = type(\pi)$
- $e = \pi.e' \mid that.e'$ est une expression bien typée ssi e' est une expression bien typée ; $type(e) = type(e')$
- $op(e_1, ..., e_l)$ est une expression bien typée ssi $arité(op) = l$ et $\forall i \in \{1, \ldots, l\}, e_i$ est une expression bien typée et $type(e_i) = type_{par(op)(i)}$; $type(e) = type_{ret(op)}$

Par exemple, cette expression : $(that.budget > 200) \wedge (budget + 10)$ n'est pas bien typée, puisque l'opérateur $\wedge$ prend deux paramètres de type booléen alors que dans cette expression, son second paramètre est un réel. Dans la suite de ce chapitre, nous ne considérons que les expressions bien typées, les autres sont considérées comme syntaxiquement incorrectes.

Comme décrit plus haut, une description est un ensemble de couples *propriété-valeur*. Désormais, nous pouvons associer une expression à chaque propriété, et non plus une valeur. Le résultat est une « description symbolique ». Une description symbolique est transformée en une description lors de l'évaluation des expressions la composant.

Définition Descriptions symboliques

SDS est l'ensemble des descriptions symboliques. Une description symbolique est une description qui fait correspondre des propriétés π à des expressions e_π, i.e. $sds \equiv \{\pi \leftarrow e_\pi \mid type(e_\pi) = type(\pi)\}_{\pi \in \mathcal{N}}$.

4.4.3 Primitives du langage

Les agents peuvent interagir avec l'environnement en utilisant des primitives dans μ définis ci-après, qui ont, généralement parlant, la même fonction que ceux proposés par Linda (ajout, lecture, et retrait de données) :

$$\mu ::= spawn(P, sds) \mid add(sds) \mid perceive(\pi, e) \mid receive(\pi, e)$$

avec e une expression et sds une description symbolique.

La primitive $spawn(P, sds)$ crée un nouvel agent qui a un comportement P et a comme description le résultat de l'évaluation de la description symbolique $sds \in SDS$ (sa transformation en une description $ds \in DS$). Nous détaillons les comportements des agents dans le chapitre suivant, il suffit pour le moment de savoir qu'un comportement P est construit avec les primitives de μ. La primitive $add(sds)$ ajoute une entité décrite par l'évaluation de sds. Un $perceive(\pi, e)$ se bloque jusqu'à ce qu'il y'ait un objet dans l'environnement tel que l'expression e est évaluée à vrai (ladite entité remplace la mot clé *that* dans e), lorsqu'une telle entité est trouvée, π devient une référence vers cette entité. La primitive $receive(\pi, e)$ se comporte de la même manière mais enlève l'objet apparié de l'environnement. Voici quelques exemples :

$$add(\{catégorie \leftarrow \text{“}promotion\text{”}, destination \leftarrow destination, prix_{ind} \leftarrow prix_{ind-10}\})$$

ajoute dans l'environnement un objet ayant comme *catégorie* une *promotion*, comme *destination* la *destination* de l'agent exécutant le *add* et comme prix individuel ($prix_{ind}$) le prix individuel de l'agent diminué de 10.

$spawn(add(\{catégorie \leftarrow \text{“}promotion\text{”}, destination \leftarrow destination, prix_{ind} \leftarrow prix_{ind} - 10\}))$, $\{destination \leftarrow \text{“}Rennes\text{”}, prix_{ind} \leftarrow 50\})$

lance un agent qui a comme comportement d'ajouter un objet comme dans l'exemple précédent, et qui a comme *destination* “Rennes” et comme $prix_{ind}$ 50.

$$perceive(concurrent, that.destination = destination \wedge that.prix_{ind} < prix_{ind})$$

exécutée par un train à la recherche de concurrents plus compétitifs, cherche à lire depuis l'environnement un objet ayant la même *destination* que lui et un prix individuel inférieur au sien. Une fois un appariement trouvé, la propriété *concurrent* du train pointera sur l'objet apparié.

Remarque 5 *Les propriétés d'un agent sont utilisées dans les actions qu'il exécute et sont évaluées lors de l'exécution. Cependant, ces propriétés ne sont pas accessibles aux autres agents du système (elles ne sont pas considérées lors de l'exécution des perceive ou des receive des autres agents), à moins que l'agent ne décide de les publier en exécutant un* add*, et ajoute un objet spécifique décrit avec ses propres propriétés. Dans le dernier exemple, cela signifie qu'un train ne percevra son concurrent que si ce dernier a décidé de publier ses propriétés sous forme d'un objet (avec un add). En revanche, les propres propriétés de l'agent (destination et* $prix_{ind}$*) sont évaluées par l'environnement, même si l'agent ne les a pas publiés.*

Avec les primitives que nous venons d'introduire, nous avons une difficulté relative au changement de valeur d'une propriété d'un agent. En effet, nos agents ne peuvent pas changer directement la valeur d'une de leurs propriétés. Par exemple, supposons qu'un agent veuille changer

sa propriété *budget*, en le diminuant de 20. En l'état, le moyen permettant à un agent de changer la valeur d'une de ses propriétés est inélégant et contre-intuitif. Il s'agit de l'utilisation de *spawn*. En effet, l'agent désirant changer sa propriété *fumeur* est obligé de créer un agent qui partage exactement les mêmes propriétés que lui, sauf la propriété *budget*, qui est diminuée de 20. Ce moyen pose le problème de la co-existence de deux agents représentant la même entité, et qui vont désormais s'exécuter en parallèle. Pour pallier ce problème, nous introduisons une primitive additionnelle *update* qui permet à un agent de mettre à jour une ou plusieurs de ses propriétés.

$$\nu ::= \cdots \mid update(sds)$$

Lors de l'exécution de $update(sds)$, avec $sds \in SDS$ une description symbolique, chaque propriété π dans sds prend la valeur de l'évaluation de l'expression correspondante e_π. Il suffit donc à l'agent d'exécuter $update(\{budget \leftarrow budget - 20\})$ pour voir sa propriété *budget* changée.

4.5 Prise en compte du contexte

4.5.1 Motivation

Reprenons l'exemple des agents voyageurs. Nous avons vu qu'un train proposait des prix individuels et des prix de groupe. Soit l'agent voyageur v qui veut percevoir les trains, seulement si leur prix de groupe ne dépassent pas son budget. Seulement voilà, v ne peut profiter des prix de groupe à moins qu'il ne trouve deux autres voyageurs pour profiter, ensemble, du prix de groupe.

Avec notre syntaxe, le seul moyen de pouvoir réaliser cet objectif est pour v de percevoir un train ayant la même destination avec des prix de groupe ne dépassant pas son budget, ensuite d'exécuter deux *perceive* à la recherche d'agents ayant la même destination que lui, et non affectés à un train (leur propriété *train* doit être égale à $unknown_{train}$), et un budget suffisant. La syntaxe correspondante aux trois *perceive* serait la suivante.

$perceive(candidat, that.destination = destination \wedge prix_{groupe} \leq budget)$
ensuite :
$perceive(compagnon_1, that.destination = destination \wedge that.train = unknown_{train} \wedge$
$that.budget \geq candidat.prix_{groupe})$
et :
$perceive(compagnon_2, that.destination = destination \wedge that.train = unknown_{train} \wedge$
$that.budget \geq candidat.prix_{groupe})$

Le problèmes posés avec cette solution sont les suivants. D'une part, le coût réseau est proportionnel au nombre d'objets que l'agent veut percevoir, puisque pour chacune, une action de perception spécifique est exécutée. D'autre part, et c'est ce qui rend cette solution non valide, lorsque l'agent perçoit l'un ou l'autre des deux voyageurs, perception qui aura lieu dans un temps non déterminé dans le futur, il est probable que la situation du train qu'il a perçu auparavant ait changée ; la première perception n'aura servi à rien, et il faut tout recommencer, puisque le nouveau train perçu aura peut être un prix de groupe ($prix_{groupe}$) supérieur au budget de l'un ou l'autre des voyageurs. Nous désirons rendre atomique la perception du train et la vérification de l'état des objets gardant cette perception.

Plus généralement, nous désirons doter les agents d'une syntaxe leur permettant d'exprimer une interaction contextuelle. C'est à dire une interaction guidée par l'état, non plus de l'entité perçue uniquement, mais aussi d'autres objets de l'environnement. Cette syntaxe doit également permettre à l'agent de percevoir tout ou une partie de l'état de son contexte. Dans notre exemple,

l'agent désirerait avoir connaissance des deux autres agents afin de pouvoir démarrer un protocole de demande de places collectives ultérieurement.

4.5.2 Syntaxe

La première étape consiste à enrichir de la syntaxe d'une expression afin d'exprimer un contexte d'interaction. Pour ce faire, une expression peut contenir des « variables » dont le type est Ω.

Définition VARIABLES

$\mathcal{X}$ est un ensemble (potentiellement infini) de variables. Une variable $x \in \mathcal{X}$ est définie par son type $type(x) \in \{type_1, \ldots, type_{nbt}\}$.

Une variable d'entité x' est une variable dans $\mathcal{X}$ avec $type(x') = \Omega$. Les variables d'entités sont les variables faisant référence à une entité. La syntaxe d'une expression est augmentée ainsi :

$e ::= \ldots$	
$\mid x$	avec $x \in \mathcal{X} \wedge type(x) = \Omega$
$\mid x.e$	avec $x \in \mathcal{X} \wedge type(x) = \Omega$

Le mot clé *that* est donc un cas particulier des variables d'entité. Les variables d'entité dans une expression e possèdent un quantificateur existentiel implicite. Par exemple, l'expression $x.position = y.position$ est interprétée comme $\exists x, y \in \Omega_{ENV} / x.position = y.position$.

Une expression bien typée est définie d'une manière inductive suivant les règles définies plus haut augmentées de la règle suivante :

- $e = x.e'$ est une expression bien typée ssi e' est une expression bien typée ; $type(e) = type(e')$;
- $e = x$ est une expression bien typée ; $type(e) = \Omega$.

Nous définissons la fonction suivante :

$$varent : Exp \rightarrow 2^{\mathcal{X}}$$

$varent(e)$ renvoie l'ensemble des variables d'entité contenues dans e, y compris *that*.

Lors de l'évaluation d'une expression e, nous décidons de permettre l'unification de deux variables d'entité x et x' dans $varent(e)$ avec le même objet de l'environnement. Ainsi, avec cette expression :

$$x.destination = destination \wedge y.budget = budget,$$

l'agent désire percevoir deux voyageurs, l'un ayant la même destination que lui, et l'autre ayant le même budget, mais il accepte également de percevoir un seul voyageur s'il satisfait aux deux conditions. L'avantage de ce choix est que nous pouvons interdire l'unification de deux variables avec le même objet explicitement (en ajoutant $\wedge x \neq y$), alors que si nous décidons d'empêcher l'unification de deux variables avec le même objet, nous ne pouvons pas permettre leur unification.

Définition UNIFICATION

Étant donné une expression e, Σ_e est l'ensemble des tuples de taille $varent(e)$ d'objets de l'environnement devant être testés pour unification avec e : $\Sigma_e = \Omega_{ENV}^{varent(e)}$. Nous parcourons les éléments de Σ_e par σ_e, σ'_e etc. Afin d'accéder à l'objet unifié avec la variable d'entité x de $varent(e)$ dans un σ_e particulier, nous utilisons $\sigma_e(x)$, qui renvoie un objet dans Ω_{ENV}.

Reprenons l'exemple précédent. L'expression exprimant le besoin de v devient la suivante.

$e_{context} = (t.destination = destination) \wedge (t.prix_{groupe} \leq budget) \wedge (x.destination = destination) \wedge (x.train = unknown_{train}) \wedge (x.budget \geq t.prix_{groupe}) \wedge (y.destination = destination) \wedge (y.train = unknown_{train}) \wedge (y.budget \geq t.prix_{groupe}) \wedge (x \neq y)$

Vu notre choix concernant l'unification des variables, l'expression $(t \neq x) \wedge (x \neq y)$ est nécessaire afin d'empêcher l'unification de x et y avec les mêmes objets (t ne risque pas d'être unifié avec x ou y puisqu'elle utilise la propriété $prix_{groupe}$ que les objets unifiés avec x ou y ne peuvent pas avoir).

La syntaxe des primitives *perceive* et *receive* doit être augmentée pour qu'un agent puisse percevoir ou recevoir tout ou une partie de son contexte (les objets unifiés avec t, x et y dans l'exemple). En effet, avec la syntaxe de $perceive(\pi, e)$ et $receive(\pi, e)$, π pointe l'objet unifié avec *that*, alors que maintenant, nous désirons disposer de plusieurs π pointant plusieurs objets unifiés avec les variables contenues dans $varent(e)$. Nous définissons une description de contexte comme un sous-ensemble des descriptions symboliques, c'est un ensemble associant des propriétés à des variables d'entité.

Définition Descriptions de contexte

$DC \subset SDS$ est l'ensemble des description de contexte. Une description de contexte est une description symbolique qui fait correspondre des propriétés π à des variables $x_\pi \in \mathcal{X}$, i.e. $dc \equiv \{\pi \leftarrow x_\pi \mid x_\pi \in \mathcal{X}, type(x_\pi) = type(\pi) = \Omega\}_{\pi \in \mathcal{N}}$.

Nous étendons la fonction *varent* pour qu'elle puisse être appliquée à une description de contexte :

$$varent : DC \cup Exp \rightarrow 2^{\mathcal{X}}$$

$varent(dc)$ renvoie les variables d'entité contenues dans la description de contexte dc, i.e. $varent(dc) = \{x \mid (\pi \leftarrow x) \in dc\}$.

La syntaxe des primitives *perceive* et *receive* devient :

$$\mu ::= \cdots \mid perceive(dc, e) \mid receive(dc, e)$$

où dc est une description de contexte. L'expression e est évaluée avec des objets de l'environnement, et dès que celle-ci est évaluée à vrai, les objets unifiés avec les variables dans $varent(e)$ sont sauvegardées et peuvent être pointés par des propriétés dans dc.

L'instruction permettant à un voyageur de percevoir le train, et de référencer les deux autres voyageurs (les objets les représentant) est :

$perceive(\{candidat \leftarrow t, compagnon_1 \leftarrow x, compagnon_2 \leftarrow y\}, e_{context})$

La propriété *candidat* pointe l'objet (train) apparié avec t, $compagnon_1$ et $compagnon_2$ pointent les objets appariés avec les variables d'entité x et y. Ainsi, un agent définit des conditions sur les objets qui l'intéressent et désigne le sous-ensemble d'objets qu'il désire percevoir ou recevoir en une seule instruction dont l'exécution est atomique.

Avec un *perceive* ou un *receive*, l'agent décide de percevoir tous les objets qu'il désigne dans son expression ou de tous les recevoir. Or, un agent pourrait désirer recevoir des entités et percevoir quelques autres dans la même expression. Par exemple, un agent voyageur pourrait recevoir un message qui lui est adressé si deux autres agents voyageurs ont la même destination que lui. Dans ce cas, seul le message est reçu, alors que les objets représentants les deux autres agents sont simplement perçus. Afin de prendre en compte les cas où un agent combine des

réceptions et des perceptions dans la même action, nous modifiant l'ensemble des primitives ν en introduisant la primitive suivante :

$$\mu ::= \cdots \mid look(dc_p, dc_r, e)$$

où $look(dc_p, dc_r, e)$ remplace les deux primitives $perceive(dc, e)$ et $receive(dc, e)$. La description de contexte dc_p désigne les objets de l'environnement qui doivent être perçus, et dc_r ceux qui doivent être reçus, i.e. perçus et supprimés de l'environnement. $look(dc_p, dc_r, e)$ remplace $perceive(dc, e)$ et $receive(dc, e)$ puisque les deux restent exprimables avec la nouvelle primitive : $perceive(dc, e)$ devient $lokk(dc, \emptyset, e)$ et $receive(dc, e)$ devient $look(dc_p, \emptyset, e)$.

L'instruction de l'exemple précédant devient :
$look(\{candidat \leftarrow t, compagnon_1 \leftarrow x, compagnon_2 \leftarrow y\}, \emptyset, e_{context})$

Remarque 6 *Nous décidons d'empêcher la présence des mêmes variables dans dc_p et dans dc_p, i.e. $dc_p \cap dc_r = \emptyset$. En effet, cela n'a aucun sens de désirer percevoir et recevoir un objet en même temps, puisque recevoir un objet revient à le percevoir (en le gardant dans l'environnement). De même, nous empêchons l'unification du même objet avec une variable dans dc_p et d'une autre dans dc_r, i.e. étant donné $look(dc_p, dc_r, e)$, $\forall \sigma_e \in \Sigma_e, x_p \in varent(dc_p), x_r \in varent(dc_r), \sigma_e(x_r) \neq \sigma_e(x_p)$.*

Les algorithmes d'appariement (le comportement de l'environnement) relatifs à l'exécution de *look* sont développés dans le chapitre suivant, lorsque nous discutons de l'implémentation du langage.

4.6 Sécurité

Nous décidons de garder un partage total des données entre tous les agents dans l'espace des données, et non de les cloisonner dans des environnements privés (de type *tuplespaces* multiples). Nous restons par ce fait fidèles au modèle Linda initial. Cependant, faire ce choix revient à rencontrer les mêmes problèmes de sécurité que le modèle original. Plus précisément, deux situations critiques doivent être soulignées.

4.6.1 Ajouts frauduleux

Pour la première situation, considérons l'exemple suivant. L'agent v veut démarrer un protocole afin d'effectuer une demande de réservation groupée à adresser au train qu'il a perçu. Pour ce faire, v envoie un message adressé vers les deux agents qu'il a désignés par $compagnon_1$ et $compagnon_2$, et qui ont formé le contexte d'exécution de *look* décrit dans la section précédente. Il exécute donc l'action suivante :

$add(\{émetteur \leftarrow id, récepteur \leftarrow compagnon_1.id, sujet \leftarrow$ "*demande groupée*"$, train \leftarrow candidat, prix \leftarrow candidat.prix_{groupe}\})$.

Un problème survient si l'agent v est tenté d'influencer la décision de l'agent récepteur de son message. Pour ce faire, il pourrait émettre un message identique au précédent vers le même agent, mais avec une valeur *émetteur* différente, égale à $compagnon_2.id$. Son intérêt serait de faire croire au destinataire que, la même demande émanant de deux agents différents, elle risque donc fort d'aboutir. Cette action est clairement frauduleuse, puisqu'un agent est en train d'essayer de se faire passer pour quelqu'un d'autre.

Plus généralement, certains ajouts d'objets dans l'environnement peuvent être proscrits par le concepteur, selon la logique de l'application. Dans une enchère par exemple, le concepteur peut

désirer interdire à tout agent de faire une enchère inférieure à un certain seuil. Cette première situation concerne donc des règles de sécurité spécifiées par le concepteur du système, et qui doivent être vérifiées lors de l'exécution d'un *add*.

Afin d'interdire des ajouts frauduleux d'objets dans l'environnement, le concepteur du système identifie les situations critiques et spécifie chacune par un prédicat s.

Définition RÈGLES DE SÉCURITÉ

$\mathcal{S} \in Exp$ est l'ensemble des règles de sécurité du système. Une règle $s \in \mathcal{S}$ est une expression booléenne i.e. $type(s) = boolen \cup \text{nil}$, telle que $varent(s) = \{that\} \vee varent(s) = \emptyset$.

Une expression s dans $\mathcal{S}$ est une expression booléenne dans laquelle le concepteur spécifie les conditions à vérifier par l'agent exécutant *add* et éventuellement par l'objet qu'il ajoute (désigné par *that*). Les variables d'entité autres que *that* n'ont en effet aucune utilité dans une règle de sécurité.

Notation 5 *Par soucis de simplicité, nous utilisons désormais that exclusivement dans le contexte des expressions relatives à la sécurité du SMA.*

Voici l'expression interdisant à un agent d'ajouter un objet ayant pour propriété *émetteur* un identifiant différent du sien : $s = (that.emetteur = id)$, avec id désignant l'identifiant de l'agent exécutant le *add* et *that* désignant l'objet ajouté par l'agent. Lors de l'exécution par v de

$add(\{émetteur \leftarrow compagnon_1.id, récepteur \leftarrow compagnon_1.id, sujet \leftarrow \text{“}demandegroupée\text{”}, train \leftarrow candidat.id, prix \leftarrow candidat.prix_{groupe}\})$

s est évaluée à faux, car *émetteur*$= compagnon_1.id \neq id$, et l'opération annulée, i.e. considérée comme inexistante. Aucune primitive ne permet de modifier l'ensemble $\mathcal{S}$, puisque seul le concepteur y a accès.

4.6.2 Restrictions d'accès

La deuxième situation critique concerne des règles de sécurité spécifiées par les agents du système. Considérons l'exemple suivant. L'agent v désire réserver une place chez le train qu'il a perçu. Pour ce faire, il exécute l'instruction suivante : $add(émetteur \leftarrow id, récepteur \leftarrow candidat.id, sujet \leftarrow \text{“}réservation\text{”})$. Évidemment, cet objet ne doit être perçu ni reçu que par l'agent dont l'identifiant est égal à *candidat.id* et aucun autre. L'agent v pourrait également désirer rendre cette décision réservée aux seuls agents qui vont prendre le même train que lui. Un exemple plus classique est celui où un agent veut simuler une mémoire locale - à accès associatif - qui n'est accessible qu'à lui. En l'état, la syntaxe n'offre aucun moyen à v de déclarer ces restrictions : une fois l'objet ajouté dans l'environnement, il échappe au contrôle de l'agent l'ayant créé.

Nous donnons la possibilité à un agent lors de l'ajout d'un objet de le protéger, i.e. d'identifier les situations où sa perception ou sa réception est interdite. Pour ce faire, nous remplaçons la syntaxe de la primitive *add* comme suit.

$$\mu ::= \cdots \mid add(sds, e_p, e_r)$$

où e_p et e_r sont des expressions booléennes. Chaque objet ajouté dans l'environnement a deux expressions qui lui sont associées. L'expression e_p définit les conditions qu'un agent doit satisfaire

afin d'avoir le droit de percevoir l'objet décrit par *sds*, et e_r définit les conditions qu'un agent doit satisfaire afin d'avoir le droit de le recevoir.

Lorsqu'un agent exécute $look(dc_p, dc_r, e)$, un ensemble d'objet de l'environnement C est sélectionné pour appariement avec e (un objet par variable de contexte), et e doit être évaluée à vrai pour que la conséquence de l'action soit effective. Maintenant, nous disposons d'une condition supplémentaire : pour chaque objet o de C unifié avec une variable de contexte dans dc_p, l'expression e_p associée à o doit être évaluée à vrai, et pour chaque objet o unifié avec une variable de contexte dans dc_r, l'expression e_r associée à o doit être évaluée à vrai. Sinon, l'action *look* ne peut être exécutée avec cet ensemble d'objets. Lorsque l'agent ne veut pas restreindre la perception ou la réception de l'objet décrit par *sds*, il affecte vrai à e_p ou e_r respectivement.

Par exemple, soit l'agent v voulant empêcher la réception de son message à destination de son *compagnon* par d'autres que ce dernier, et sa perception par les agents autre que lui même ainsi :

$add(\{propriétaire \leftarrow id, \text{emphémetteur} \leftarrow id, \text{emphrécepteur} \leftarrow compagnon.id, sujet \leftarrow$ "*decision*"$, train \leftarrow candidat.id\}$,

$id \neq that.propriétaire, id \neq that.récepteur)$

Il peut ne permettre sa perception à personne, et empêcher sa réception par des agents n'empruntant pas le même train que lui ainsi :

$add(\{émetteur \leftarrow id, récepteur \leftarrow compagnon.id, sujet \leftarrow$ "*décision*"$\}, train \leftarrow candidat.id,$ vrai$, train \neq that.train)$

4.7 Interaction avec un système externe

Le langage de coordination défini jusqu'à présent spécifie un SMA ouvert dans le sens où les agents peuvent rejoindre le système - avec *spawn* - et découvrir les autres agents et les données échangées - avec *look* - mais il s'agit d'un SMA clos dans le sens où on ne peut exprimer son interaction avec un système externe. En effet, toutes les données nécessaires à l'exécution d'un programme doivent exister dans le code des agents. Par exemple, nous désirons décrire le comportement d'un agent voyageur. Un agent voyageur représente un usager humain dans le SMA, dont les propriétés (*destination*, *budget*, etc.) sont inconnues avant l'exécution. Ici, les valeurs des propriétés d'un agent voyageur proviennent d'un système externe (serveur Web par exemple). Nous avons introduit la notion de variable afin d'exprimer les variables de contexte dans une action *look*. Nous étendons cette notion afin d'exprimer l'action d'un système externe sur le comportement d'un agent. Voici l'instruction lançant un agent voyageur ayant des propriétés qui seront connues lors de l'exécution grâce à l'action d'un système externe :

$$spawn(P, \{budget \leftarrow b, age \leftarrow a, fumeur \leftarrow f, destination \leftarrow d\})$$

avec a, b, d et f des variables. Le système externe n'est pas modélisé, seule son action est observée, i.e. l'affectation d'une valeur à la variable.

La syntaxe d'une expression est augmentée avec les variables ainsi :

$e ::= \ldots$	
$\mid x$	avec $x \in \mathcal{X}$

Les variables d'une expression prennent une valeur prise d'une manière non déterministe dans leur domaine de valeurs (leur type).

L'introduction des variables pour l'interaction avec un système externe est intéressante dans la mesure où elle sépare clairement l'aspect coordination - ce que fait le programme - et l'aspect

	loc	*contextvar*	*openvar*	*that*
sds	✓		✓	
e_p, e_r	✓		✓	✓
$s \in \mathcal{S}$	✓			✓
dc		✓		
e_c	✓	✓	✓	

TAB. 4.2 – Synthèse de l'usage des expressions

interaction avec un système externe, i.e. la manière avec laquelle il est utilisé. Ainsi, dans le processus $P = spawn(P, \{budget \leftarrow b, age \leftarrow a, fumeur \leftarrow f, destination \leftarrow d\})$, peut importe le système instanciant les variables de P (serveur Web, interface graphique, simulateur, etc.) le code relatif au comportement des agents reste le même.

4.8 Synthèse

Dans cette section, nous récapitulons la syntaxe que nous avons introduite et rappelons les principales règles d'écriture. Les primitives définitives du langages sont :

$$\mu ::= spawn(P, sds) \mid add(sds, e_p, e_r) \mid look(dc_p, dc_r, e_c) \mid update(sds)$$

La syntaxe définitive d'une expression est la suivante :

$e ::= \text{nil}$	
$\mid v$	, avec $v \in \mathcal{T} \backslash \text{nil}$
$\mid \pi$	, avec $\pi \in \mathcal{N}$
$\mid op(e, \ldots, e)$	, avec *op* un opérateur du langage, et nil n'apparaît dans aucun e
$\mid \pi.e$	, avec $\pi \in \mathcal{N}$ et $type(\pi) = \Omega$
$\mid that.e$	, avec $\pi \in \mathcal{N}$ et $type(\pi) = \Omega$
$\mid x.e$	avec $x \in \mathcal{X} \wedge type(x) = \Omega$
$\mid x$	avec $x \in \mathcal{X}$

Le tableau 4.2 récapitule l'usage des expressions avec les primitives du langage. Soit $loc = \{\text{nil}, v \in \mathcal{T}, \pi \in \mathcal{N}, op(e, \ldots, e), \pi.e\}$ reflétant les expressions « locales », i.e. ne faisant intervenir aucune variable ; *contextvar* sont les variables de contexte, i.e. les variables dont le type est Ω et *openvar* sont les variables matérialisant l'interaction du SMA avec un système externe.

Les descriptions symboliques *sds* paramètres de *add*, *spawn* et *update* peuvent contenir des éléments de *loc* ou des variables dans *openvar* comme vu dans l'exemple ci-haut. Les expressions e_p et e_r paramètres de *add* sont des expressions de sécurité déposées par l'agent, elles peuvent contenir des éléments de *loc*, des variables de *openvar* ou le mot clé *that* désignant l'objet ajouté. Les règles de sécurité $s \in \mathcal{S}$ peuvent contenir les mêmes éléments que e_p et e_r sauf les variables dans *openvar*. Enfin, les descriptions de contexte paramètre de *look* ne peuvent contenir des

variables dans *contextvar* et les expressions e_c paramètres de *look* peuvent tout contenir sauf le mot clé *that* que nous avons réservé pour les expressions de sécurité.

4.9 Discussion

Les problèmes que nous traitons dans le modèle Acios ont été abordés par différents modèles dans la littérature. Nous discutons les trois notions clés développées dans ce chapitre : la structure de données et l'appariement, l'interaction contextuelle et la sécurité.

4.9.1 Structure de données et appariement

Par rapport à la structure des données que nous avons retenue, il est intéressant d'observer que la majorité des systèmes proposés dans la littérature implémentent leurs données d'une manière différente du modèle original. En effet, JavaSpaces [Freeman *et al.*, 1999] utilise un modèle objet et TSpaces [Wyckoff *et al.*, 1998] utilise un modèle relationnel, chaque tuple étant représenté par une ligne dans une table. Cependant, il s'agit là de solutions de mise en oeuvre, et ces propositions n'augmentent pas la richesse de leur représentation au niveau du modèle. Or, enrichir le modèle de données seulement au niveau de l'implémentation fait que le mécanisme d'appariement reste celui des *templates* à la Linda, et ne tire donc pas profit de l'enrichissement du modèle de données. Tucson [Omicini and Zambonelli, 1999] et Shared Prolog [Brogi and Ciancarini, 1991; Ambriola *et al.*, 1990] utilisent un modèle logique du premier ordre, et partagent donc quelques uns des avantages de la structure de données d'Acios, à savoir le nommage des entités et des champs (équivalent des propriétés). Néanmoins, ils gardent toujours la contrainte de l'ordre des champs lors de l'appariement (effectué par unification). Limone [Fok *et al.*, 2004] utilise des couples *attribut-valeur* dans la représentation des données échangées dans le *tuplespace*, mais aucun intérêt n'en est tiré pour construire des expressions ou des *templates* plus riches. Par ailleurs, le modèle Acios est suffisamment général pour rester transposable, lors de l'implémentation, dans chacun de ces modèles.

Hormis les travaux de notre équipe, nous n'avons pas connaissance de travaux dans le contexte des modèles de coordination orientée-données qui considèrent des agents avec un état, qu'ils peuvent utiliser pour conditionner leur interaction. Néanmoins, des différences sont à souligner entre le modèle présenté dans ce chapitre et les autres travaux de l'équipe. D'abord, dans le modèle de données de référence, appelé EASI[30] [Balbo, 2000b; Saunier *et al.*, 2006], l'état des agents est observable des autres agents, alors que dans Acios, il n'est observable que de l'environnement. Dans EASI, lorsqu'un agent change la valeur d'une de ses propriétés, toutes les expressions en attente (les filtres dans EASI) dans l'environnement doivent être testées à la recherche d'un appariement. Dans Acios en revanche, l'état d'un agent n'est observable que de l'environnement SMA, il doit explicitement ajouter un objet dans l'environnement afin qu'il devienne observable des autres. Par conséquent, un *update* dans Acios n'affecte que les expressions en attente de l'agent, et non toutes les expressions de l'environnement. Dans EASI, le modèle est considéré à un niveau d'abstraction plus élevé, les comportements des agents ne sont donc pas modélisés, seules leurs propriétés sont observables. L'ajout d'un objet, le changement d'une propriété d'un agent, l'ajout et le retrait de filtres, etc. ne sont pas spécifiés. Acios en revanche a pour objectif de définir clairement le comportement des agents et la dynamique du SMA, un langage et une implémentation lui sont associés (c.f chapitre suivant), alors que EASI n'impose pas de règles d'implémentation particulières.

[30] *Environment as Active Support for Interactions*

4.9.2 Interaction contextuelle

L'introduction de l'interaction contextuelle *stricto sensu* dans le cadre d'un modèle de coordination orientée-données est originale. Néanmoins, il existe un travail dont l'idée générale peut être assimilée à notre proposition. Dans [Jang *et al.*, 2004], les auteurs posent le problème de la pauvreté d'expressivité des *templates* à la Linda, et proposent le système ATSpaces, où chaque agent offre son algorithme d'appariement personnel à l'environnement. Cette proposition offre plus de possibilités aux agents dans la définition de leurs besoins interactionnels. En effet, avec leur système, les auteurs peuvent exprimer des besoins tels que « la lecture des deux meilleurs vendeurs (en terme de prix) qui vendent des ordinateurs et qui se trouvent à une distance de 50 miles », ou encore « trouver tous les tuples concernant des vendeurs d'ordinateurs ». Ainsi, l'exécution des algorithmes des agents ne résulte pas en une évaluation à vrai ou faux, mais identifie un ensemble de données qui sera envoyé à l'agent. Cependant, cette manière de faire rompt avec l'utilisation de l'espace de données comme étant le système réalisant l'appariement, et l'utilise comme une boucle distribuant toutes les données de l'espace de données à tous les algorithmes personnalisés des agents. De plus, comme le notent les auteurs, ATSpaces présente des problèmes de sécurité évidents, puisque les agents manipulent directement les données de l'espace de données, et pourrait en faire un usage frauduleux.

Les primitives de lecture globale et d'extraction globale, telles que *copy-collect*(t) et *collect*(t) de TSpaces [Wyckoff *et al.*, 1998] (c.f chapitre 1), qui prennent un *template* t en paramètre ne fournissent pas le moyen d'une interaction contextuelle. En effet, les données lues ou extraites satisfont toutes au même *template* t, et ils ne proposent aucun moyen de comparer les données entre elles, comme c'est le cas dans Acios.

4.9.3 Sécurité

Comme nous l'avons montré dans le chapitre 1, la sécurité est généralement renforcée avec l'utilisation de *tuplespaces* multiples ou en utilisant des « lois ». Avec les *tuplespaces* multiples, deux agents désirant échanger des données confidentielles utilisent un *tuplespace* qui n'est connu que de eux deux. Cependant, lorsque la sécurité est garantie en isolant les données dans des espaces privés, le fait d'accéder à un espace donne la possibilité d'accéder à toutes ses données, et en être exclu revient à n'avoir accès à aucune donnée. La gestion des droits d'accès à ces espaces, en la présence d'agents sans état, ne peut se faire que d'une manière nominative. Or, le modèle Linda a comme principal avantage de permettre une communication anonyme et guidée par le contenu. Que les agents doivent se connaître afin de pouvoir sécuriser leurs données va à l'encontre de l'un des principes clés de la communication générative. Un agent inconnu des autres peut se retrouver dans un espace public (accessible à tous les agents) où personne n'envoie de données, et par ce fait se retrouver isolé. Dans Acios, les agents ont un état (une description), et grâce à la syntaxe des expressions, un agent peut protéger ses données sans forcément connaître les autres agents du SMA. Ceci permet de garantir la sécurité tout permettant un partage total des données. Il suffit à un agent d'avoir les bonnes valeurs de propriétés, spécifiées par « l'auteur » de l'entité, pour pouvoir accéder aux données dont il a besoin.

Les lois dans LGI [Minsky *et al.*, 2001] permettent de sécuriser les données en spécifiant des conditions sur l'état des agents et le contenu des données. Néanmoins, deux points différencient Acios de LGI. D'une part, les lois dans LGI sont définies par le concepteur du système afin de garantir le respect de la logique de l'application. Les agents quant à eux ne peuvent pas introduire des conditions sur la visibilité des données qu'ils produisent. D'autre part, les lois sont des règles actives, ce qui pose le problème de terminaison du processus d'appariement comme

	Struct. données	Appariement	Contexte	Sécurité
Klaim	tuples	*templates*	non	TS mult.
Limone	propriétés-valeurs	*templates*	non	TS mult.
Tucson	prédicats	unification	non	TS mult.
LGI	prédicats	unification	non	lois
ATSpaces	tuples	*templates*	oui	non
Javaspaces	objets	*templates*	non	non
TSpaces	relationnel	*templates*	non	non
Acios	propriétés-valeurs	expressions	oui	restrictions

Tab. 4.3 – Comparaison de Acios avec les travaux de la littérature

nous l'avons montré dans le chapitre 1. Dans Acios, les restrictions annulent des perceptions ou des réceptions mais n'enclenchent aucune réactions, le problème de non-terminaison ne se pose donc pas. Le tableau 4.3 synthétise les différences et similitudes entre Acios et les propositions les plus représentatives dans la littérature.

4.10 Conclusion

Dans ce chapitre, nous avons proposé le modèle de coordination Acios, qui est d'un grand intérêt dans le cadre des SMA ouverts. Ses principales caractéristiques sont, d'abord une représentation des entités en couples *propriété-valeur* qui permet un mécanisme d'appariement d'une grande expressivité. Ensuite, l'extension de la syntaxe des expressions permet d'exprimer un besoin interactionnel contextuel. Grâce à la modélisation des états pour les agents, décrits suivant la même structure de données, le modèle peut spécifier des règles garantissant la sécurité des objets échangés et leur respect de la logique de l'application. Enfin, grâce à l'introduction des variables libres, le modèle exprime l'action d'un système externe sur le comportement des agents du SMA.

Néanmoins, un programmeur désirant implémenter un SMA adhérant à Acios se trouve confronté à des ambiguïtés de comportement de son système, tels que ceux décrits dans le chapitre 2. Ces ambiguïtés sont levées par le langage de coordination Lacios.

Chapitre 5

Langage de coordination Lacios

Sommaire

5.1 Introduction

Le modèle Acios permet la modélisation d'un SMA coordonné via l'environnement, permettant aux agents d'avoir une interaction contextuelle, tout en permettant la sécurisation des données. L'objectif de ce chapitre est de définir un langage de coordination appelé Lacios[31] (*Language for Agent Contextual Interaction in Open Systems*) matérialisant le modèle de coordination Acios présenté dans le chapitre précédent. Nous avons défini une partie de la syntaxe de Lacios dans le chapitre précédant et avons donné une description intuitive du comportement d'un système coordonné via Acios. Ce chapitre complète la définition de la syntaxe du langage et lui associe une sémantique opérationnelle, définissant l'évolution de l'état d'un programme écrit en Lacios.

[31] Ce travail est effectué au sein de l'équipe du projet conception de systèmes ouverts interopérables (pôle Agents Intelligents et Modèles Coopératifs) au laboratoire Lamsade - Université Paris-Dauphine

La sémantique opérationnelle du langage permet de définir rigoureusement les transitions exécutables par un programme écrit en Lacios, et à terme, de vérifier son comportement et ses propriétés. Mais elle définit également une spécification qui guide l'implémentation du langage dans un langage de programmation hôte, de sorte que le comportement du système écrit dans le langage hôte suit rigoureusement la sémantique opérationnelle du langage de coordination. Nous choisissons d'implémenter Lacios au dessus de Java.

Ce chapitre est structuré comme suit. La section 5.2 présente le comportement des agents. Nous y introduisons les opérateurs de composition de processus, permettant à un agent de définir un comportement et non des actions isolées comme présentées dans le chapitre précédant. La structure définitive d'un système coordonné y est également introduite. La section 5.3 présente la sémantique du langage. Elle se divise en trois parties. Dans la première, nous définissons l'évaluation des expressions, i.e. comment à partir d'une expression, trouver sa valeur. Dans la seconde, nous définissons la sémantique opérationnelle du langage, i.e. les règles spécifiant l'évolution de l'état d'un système étant donné un programme Lacios. Enfin, nous introduisons les variables, nécessaires pour la définition d'un système ouvert, i.e. interagissant avec un système externe. La section 5.4 décrit l'implémentation de Lacios sous forme d'un langage au dessus de Java. Grâce à ce nouveau langage, un programmeur peut écrire un programme dans la syntaxe de Lacios comme définie dans ce chapitre et le chapitre précédant et l'exécuter directement en étant sûr que le comportement de son programme est celui spécifié par la sémantique de Lacios. Des aspects relatifs à l'implémentation tels que les algorithmes d'appariement, leur complexité et la distribution de l'environnement y sont discutés.

5.2 Comportements des agents

Dans les différents exemples présentés dans le chapitre précédent, nous avons décrit le comportement des agents en désignant les actions qu'ils exécutent. Néanmoins, le comportement d'un agent ne peut se résumer à des actions isolées, et leur composition ensemble dans un programme doit être explicitée. Comme nous l'avons vu dans le chapitre 2, il existe différents opérateurs pour la composition d'un processus. Nous utilisons les mêmes pour la définition du comportement d'un agent. De plus, dans le chapitre 2, l'opérateur de composition parallèle est implicite : tous les processus s'exécutent en parallèle, mais chaque processus est séquentiel. Dans Lacios, nous introduisons l'opérateur de composition parallèle dans la définition du comportement d'un agent, puisqu'un agent ne peut être réduit à un processus séquentiel.

Concernant l'usage des variables libres, représentant l'action d'un système externe dans le SMA, nous introduisons un opérateur νX qui affecte explicitement des valeurs à l'ensemble de variables X. Les valeurs affectées aux éléments de X sont prises d'une manière non déterministe dans leur domaine de valeurs (leur type).

5.2.1 Opérateurs de composition de processus

Ci-après la définition complète d'un processus.

Définition Processus

Étant donné un ensemble d'identifiants de processus $\{K_i\}_{i \in I}$, une définition de processus est sous la forme : $\forall i \in I, K_i \stackrel{def}{=} P_i$, où chaque P_i est généré via la grammaire du tableau 5.1.

$P ::= \mathbf{0}$	(processus nul)
$\mid \mu.P$	(préfixage par une action)
$\mid b\lfloor P\rfloor + b\lfloor P\rfloor$, où b est une expression *booléenne*	(choix)
$\mid P \| P$	(composition parallèle)
$\mid \nu X(P)$	(liaison de variables)
$\mid K_j$, pour un certain $j \in I$	(invocation de processus)
$\mu ::= spawn(P, sds) \mid add(sds, e_p, e_r) \mid look(dc_p, dc_r, e) \mid update(sds)$ avec e_p et e_r des expressions, sds une description symbolique dc_p et dc_r des descriptions de contexte	

Tab. 5.1 – Syntaxe d'un Processus

Les processus, que nous dénotons $P, Q, \ldots$ représentent les programmes du système, et les comportements des agents qui y évoluent. Un programme peut être terminé $\mathbf{0}$ (habituellement omis). Il peut être une expression de choix entre programmes $b\lfloor P\rfloor + b\lfloor P\rfloor$, où chaque P est gardé par l'évaluation d'une expression *booléenne* b : quand b est évaluée à vrai, le programme P est exécuté. Avec cette syntaxe, nous pouvons exprimer aussi bien un choix déterministe qu'un choix non déterministe. En effet, $b\lfloor P\rfloor + \neg b\lfloor Q\rfloor$ est un choix déterministe entre P et Q, alors que $\text{vrai}\lfloor P\rfloor + \text{vrai}\lfloor Q\rfloor$ est un choix non déterministe. La syntaxe du choix peut se généraliser à plus de deux processus. En effet, l'expression qui suit est un choix entre trois processus P, Q et $R : b_1\lfloor P\rfloor + (b_2 \vee b_3)\lfloor b_2\lfloor Q\rfloor + b_3\lfloor Q\rfloor\rfloor$. Un programme peut aussi être une composition parallèle de programmes $P\|Q$, P et Q sont donc exécutés en parallèle. Un programme peut être une invocation d'un autre processus ayant comme identifiant la constante K_j, et se comporte comme le processus défini par K_j. $\nu X(P)$ se comporte comme P où toutes les variables libres sont liés avec des valeurs dans leur type. Un programme peut également être un processus préfixé par une action $\mu.P$. Les actions sont les primitives du langage, comme présentées dans le chapitre précédant.

La primitive $spawn(P, sds)$ crée un nouvel agent qui se comportera comme P et a comme description le résultat de l'évaluation des expressions dans sds. La primitive $add(sds, e_p, e_r)$ ajoute dans Ω_{ENV} un objet décrit par l'évaluation de sds, l'objet est protégé en perception par e_p et en réception par e_r. $look(dc_p, dc_r, e)$ bloque jusqu'à ce qu'il y'ait un ensemble d'objet dans Ω_{ENV} tels que l'expression e est évaluée à vrai sans que les règles associées aux objets ne soient violées, les objets unifiés avec des variables dans dc_p sont perçus et ceux unifiés avec des variables dans dc_r sont reçus. $updat(sds)$ met à jour un ensemble de propriétés de l'agent avec l'évaluation des expressions dans sds. Nous définissons la congruence structurelle $\equiv$, utilisée afin de considérer deux processus comme syntaxiquement équivalents :

$$P.\mathbf{0} \equiv P \text{ (élément neutre 1)}$$

$$P\|\mathbf{0} \equiv P \text{ (élément neutre 2)}$$

$$P\|Q \equiv Q\|P \text{ (commutativité de l'opérateur de parallélisme)}$$

/*Un voyageur commence par ajouter un objet le représentant dans l'environnement. Cet objet peut être perçu par tous les autres agents (e_p = vrai) et ne peut être reçu que par lui-même ($e_r = id = id$)*/

$$voyageur \stackrel{def}{=} add(\{propriétaire \leftarrow id, id \leftarrow id, catégorie \leftarrow catégorie, fumeur \leftarrow fumeur, destination \leftarrow destination, budget \leftarrow budget\}, \text{vrai}, that.propriétaire = id).$$

/*Ensuite, le voyageur perçoit un train ayant la même destination que lui*/

$$(look(train \leftarrow x, \emptyset, x.destination = destination).$$

/*Lorsqu'un tel train est trouvé (l'objet le représentant), si son prix individuel est inférieur ou égal au budget du voyageur, ce dernier commence une procédure de réservation individuelle avec le train. Sinon, il commence à chercher des voyageurs susceptibles de demander avec lui une réservation de groupe et bénéficier ainsi des prix de groupe. Un voyageur est toujours à l'écoute de messages qui lui sont adressés (le comportement *dyadic* - voir plus bas)*/

$$(train.prix_{ind} > budget)\lfloor groupe \rfloor + (train.prix_{ind} \leq budget)\ \lfloor reservation \rfloor \| dyadic)$$

/*La procédure de demande de réservation groupée est comme décrite dans le chapitre précédent. L'agent voyageur commence par percevoir un train proposant un prix de groupe couvert par son budget, perception gardée par deux autres voyageurs avec la même destination que lui, non associés avec un train, et ayant un budget suffisant*/

$$groupe \stackrel{def}{=} look(\{candidat \leftarrow t, compagnon_1 \leftarrow x, compagnon_2 \leftarrow y\}, \emptyset, t.destination = destination) \wedge (t.prix_{groupe} \leq budget) \wedge (x.destination = destination) \wedge (x.train = unknown_{train}) \wedge (x.budget \geq t.prix_{groupe}) \wedge (y.destination = destination) \wedge (y.train = unknown_{train}) \wedge (y.budget \geq t.prix_{groupe}) \wedge (x \neq y)).$$

TAB. 5.2 – Exemple de définitions de processus (1/3)

$b\lfloor P \rfloor + b'\lfloor Q \rfloor \equiv b'\lfloor Q \rfloor + b\lfloor P \rfloor$ (commutativité de l'opérateur de choix)

$(P \| Q) \| C \equiv P \| (Q \| R)$ (associativité de l'opérateur de parallélisme)

Reprenons l'exemple des agents voyageurs défini dans le chapitre précédent. Les tableaux 5.2, 5.3 et 5.4 donnent un ensemble de définitions de processus possibles pour cet exemple. Des commentaires sont insérés dans le code pour une meilleure lisibilité (entre /* et */).

Un processus définit le comportement d'un agent. Un agent ne peut définir un comportement tel que $dyadic \stackrel{def}{=} dyadic$, car cette définition ne traduit aucune action. Ci-après une définition de la validité d'un processus qui généralise ce principe.

Définition PROCESSUS VALIDE

La transition $\twoheadrightarrow$ (tenant pour « mène directement à ») est définie d'une manière inductive comme suit, $P \twoheadrightarrow K \Leftrightarrow$

- $P = K$ ou
- $P = b\lfloor Q \rfloor + b'\lfloor R \rfloor$, avec $Q \twoheadrightarrow K \vee R \twoheadrightarrow K$ ou
- $P = Q \parallel R$, avec $Q \twoheadrightarrow K \vee R \twoheadrightarrow K$

Un processus est valide ssi $\forall i \in I, K_i \not\twoheadrightarrow K_i$

/*Ensuite, deux messages adressés sont envoyés vers les deux agents perçus leur proposant une demande de réservation groupée*/

$$(add(\{\textit{émetteur} \leftarrow id,\ \textit{destinataire} \leftarrow \textit{compagnon}_1.id, \textit{sujet} \leftarrow \text{“}\textit{demande groupée}\text{”},\ \textit{train} \leftarrow \textit{candidat}.id, \textit{prix} \leftarrow \textit{candidat}.\textit{prix}_{\textit{groupe}}\}, id = \textit{that}.\textit{destinataire},\ id = \textit{that}.\textit{destinataire})$$
$$\|add(\{\textit{émetteur} \leftarrow id,\ \textit{destinataire} \leftarrow \textit{compagnon}_1.id, \textit{sujet} \leftarrow \text{“}\textit{demande groupée}\text{”},\ \text{train} \leftarrow \textit{candidat}.id, \textit{prix} \leftarrow \textit{candidat}.\textit{prix}_{\textit{groupe}}\}, id = \textit{that}.\textit{destinataire},\ id = \textit{that}.\textit{destinataire}))$$

/*La procédure de réservation auprès d'un train se fait en envoyant un message adressé au train perçu*/

$$\textit{reservation} \stackrel{def}{=} add(\{\textit{propriétaire} \leftarrow id, \textit{émetteur} \leftarrow id, \textit{destinataire} \leftarrow \textit{train}.id,\ \textit{sujet} \leftarrow \text{“}\textit{réservation}\text{”}\}, id = \textit{that}.\textit{destinataire}, id = \textit{that}.\textit{propriétaire})$$

Tab. 5.3 – Exemple de définitions de processus (2/3)

Par exemple, les définitions de processus ci-avant sont valides alors que, e.g. $\textit{voyageur} \stackrel{def}{=} \textit{look}(dc_p, dc_r, e_1)\|\textit{voyageur}$ n'est pas un processus valide, puisque $\textit{voyageur} \twoheadrightarrow \textit{voyageur}$. Avec cette définition, nous pouvons juger de la validité syntaxique d'une définition de processus. Dans la suite de ce chapitre, nous supposons que tous les processus considérés sont valides.

5.2.2 SMA coordonné

Nous sommes à présent capables de présenter la structure du SMA coordonné, illustré par la figure 5.1. Dans ce schéma, les objets, quelque soit leur emplacement actuel sont regroupés dans l'ensemble $\mathcal{O}$, et les agents sous l'ensemble $\mathcal{A}$. Chaque agent a a une mémoire locale Ω_a, dans laquelle sont stockés les objets perçus et reçus. Les objets dans Ω_{ENV} et dans Ω_a sont considérés comme des références vers des objets dans $\mathcal{O}$. Les objets ne changent pas de valeurs de propriétés dynamiquement, il est donc correct d'estimer qu'un objet référencé dans Ω_{ENV} et dans Ω_a aura toujours la même description. Un agent a accès à sa propre description (quand π est rencontrée dans le code d'un agent a, elle se réfère à $d_a(\pi)$) et est défini également par un comportement (P ou Q dans la figure). L'environnement est quant à lui constitué d'un ensemble de références vers des objets dans $\mathcal{O}$. Une définition complète est donnée ci-après.

La sémantique opérationnelle du langage montre l'évolution d'un système coordonné d'un état vers un autre, et les conditions sous lesquelles cette évolution est possible. L'état d'un SMA coordonné est défini comme suit.

Définition SMA Coordonné

$CS = \langle \Omega, d, \Omega_{ENV}, \mathcal{S} \rangle$,
- $\Omega = \mathcal{A} \uplus \mathcal{O}$ est l'ensemble d'entités, composé de $\mathcal{A}$ l'ensemble des agents et $\mathcal{O}$ l'ensemble des objets,
  - $\mathcal{A} \subseteq \Omega$ est l'ensemble d'agents.
    - Ω_a est la mémoire privée de l'agent a, $\Omega_a \subseteq \mathcal{O} \cup \{a\}$, i.e. l'agent a accès à sa propre description,

$dyadic \stackrel{def}{=} look(\emptyset, \{message \leftarrow x\}, that.destinataire = id).$
/*Les messages adressés à un voyageur peuvent être une demande de réservation groupée d'un autre agent voyageur. Dans ce cas, s'il n'a pas encore de train associé, il change sa propriété *train* vers la nouvelle valeur de *train*, correspondant à la demande de réservation collective et envoie une acceptation au voyageur demandeur*/

$(message.sujet =$ "*demande-groupée*"$)\lfloor(train = unknown_{train})$
$\lfloor add(\{propriétaire \leftarrow id, émetteur \leftarrow id, destinataire \leftarrow message.émetteur, sujet \leftarrow$
"*décision*",
decision=vrai}, id= that.destinataire, id= that.*propriétaire*).update({train
$\leftarrow message.train\}).dyadic\rfloor$

Si sa propriété *train* est différente de $unknown_{train}$, il envoie un refus à l'agent qui l'a sollicité
*

$+(train \neq unknwon)\lfloor add(\{émetteur \leftarrow id, destinataire \leftarrow train.id, sujet \leftarrow$ "*décision*",
decision=vrai}, id = that.destinataire, id = that.destinataire).dyadic $\rfloor\rfloor$

/*S'il reçoit une décision d'acceptation d'une demande de réservation groupée d'un agent voyageur auquel il aurait adressé une demande, il change sa propriété *train* vers la nouvelle valeur de *train*, et envoie une demande de réservation groupée au train.*/

$+(message.sujet =$ "*décision*"$)\ \lfloor(message.decision =$ vrai$)\lfloor add(\{émetteur \leftarrow id,$
destinataire$\leftarrow train.id, sujet \leftarrow$ "*demande-*
groupée", $voyageur_1 \leftarrow id, voyageur_2 \leftarrow compagnon.id,$
voyageur$_3 \leftarrow compagnon_1.id, voyageur_4 \leftarrow compagnon_2.id\}, id = that.destinataire,$
id= that.*propriétaire*).update({train$\leftarrow candidat\}).dyadic\rfloor$

/*un refus de proposition de réservation groupée. Dans ce cas, il ne fait rien. */

$+(message.decision =$ faux$)\lfloor dyadic\rfloor\rfloor$

/*Le comportement d'un train est simplifié : il ne fait que recevoir des demandes de réservation, soit groupées soit individuelles et change sa capacité, qui est diminuée soit de un (réservation individuelle), soit de trois (réservation groupée)*/

$train \stackrel{def}{=} look(\emptyset, \{message \leftarrow x\}, that.destinataire = id).((message.sujet =$ "*demande-groupée*"$)$
$\lfloor update(\{capacité \leftarrow capacité{-}3\})\rfloor + (message.sujet =$ "*demande-individuelle*"$)\lfloor$
update($\{capacité \leftarrow capacité{-}1\})\rfloor.train)$

Tab. 5.4 – Exemple de définitions de processus (3/3)

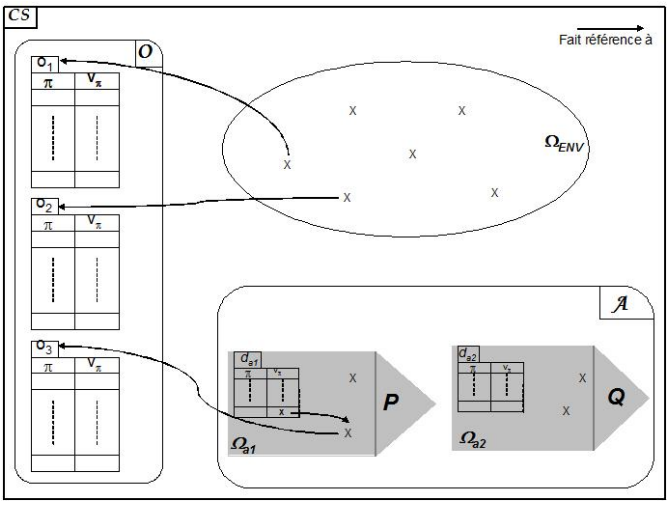

FIG. 5.1 – SMA Coordonné

  - $proc(a)$ est le processus décrivant le comportement de l'agent a.
- $\mathcal{O} \subseteq \Omega$ est l'ensemble d'objets.
  - $e_p(o)$ renvoie le prédicat spécifiant les conditions de perception de l'objet o, i.e. quels agents peuvent percevoir o.
  - $e_r(o)$ renvoie le prédicat spécifiant les conditions de réception de l'objet o,
- $d : \Omega \rightarrow (\mathcal{N} \rightarrow \mathcal{T})$ est la fonction de description du SMA, chaque $d(\omega)$ est une description d'entité comme décrite précédemment (dénotée également par d_ω),
- $\Omega_{ENV} \subseteq \mathcal{O}$ est l'ensemble d' objets qui sont dans l'environnement,
- $\mathcal{S} \subset Exp$ est l'ensemble des prédicats spécifiant les conditions devant être vérifiées, lors de l'exécution d'un *add*, par l'agent appelant et par la description ajoutée, afin d'être validé.

La mémoire locale d'un agent est composée des objets qu'il a perçus et reçus. L'environnement est Ω_{ENV}, tous les objets ajoutés par des *add* y sont placées.

5.3 Sémantique du langage

5.3.1 Évaluation des expressions

Nous avons montré dans le chapitre précédent la syntaxe d'une expression. Ci-après, nous développons la sémantique qui y est associée, i.e. comment les expressions sont évaluées. Nous commençons par la sémantique des opérateurs du langage.

$$[\![op]\!] : \prod_{i=1}^{arity(op)} type_{par(op)(i)} \rightarrow type_{ret(op)}$$

$$\forall op, \text{si } \exists i \in \{1, \ldots, arity(op)\} \mid x_i \text{ évalué à nil, alors } [\![op]\!](x_1, \ldots, x_{arity(op)}) = \text{nil}$$

La dernière définition, qui associe nil à l'évaluation d'un opérateur, lorsque l'un de ses paramètres est évalué à nil, vient en complément de l'analyse syntaxique du chapitre précédent. Nous avions interdit qu'un paramètre soit égal à nil. Si l'opérateur est valide syntaxiquement, i.e. aucun nil

n'apparaît dans ses arguments, mais que l'un de ses arguments est *évalué* à nil, nous générons une erreur sémantique, représentée par la valeur nil.

Considérons l'opérateur booléen `et`, sa sémantique est donnée par $[\![\texttt{et}]\!] : booléen^2 \rightarrow booléen$, qui est définie par la table de vérité de l'opérateur `et` (avec nil renvoyé lorsque l'un des paramètres est évalué à nil).

Étant donné une expression e bien typée, elle est évaluée dans un contexte d'exécution qui diffère selon l'action exécutée. Lors de l'exécution d'un $spawn(P, sds)$, un $add(sds, e_p, e_r)$ ou un $update(sds)$, les expressions contenues dans les sds en paramètres des primitives sont dites expressions locales, leur contexte d'évaluation est : le système coordonné CS (afin d'avoir accès à la fonction de description d), l'agent exécutant l'action a (afin d'avoir accès à sa mémoire locale), ainsi que l'entité en cours d'évaluation o. L'évaluation de l'expression e - $[\![CS, a, o \mid e]\!]$ - associe une valeur à l'expression e. Une évaluation démarre avec $o = a$, i.e. $[\![CS, a, a \mid e]\!]$. Lorsque $\pi.e'$ est rencontrée pour la première fois dans le code d'une expression, cela signifie que l'agent a désire accéder à une propriété d'un objet dans sa mémoire locale Ω_a, auquel cas $[\![CS, a, d_a(\pi) \mid e']\!]$ est appelée. Si $\pi.e'$ est rencontrée ultérieurement, cela signifie que a désire accéder à une propriété d'un objet dans sa mémoire locale pointé par un autre de ses objets (e.g le nom du voisin de son voisin). Dans ce cas, $[\![CS, a, d_o(\pi) \mid e']\!]$ est appelée. L'entité o dans le contexte de l'évaluation d'une expression désigne donc l'objet courant, pour savoir à la propriété de quel objet on se réfère quand on rencontre π.

Une expression locale est évaluée d'une manière inductive suivant la définition suivante.

Définition L'évaluation d'une expression locale

$[\![CS, a, o \mid e]\!] =$

$$\begin{cases} v & \text{si } e = v, \text{ avec } v \in type_i \\ d_o(\pi) & \text{si } e = \pi, \text{ avec } \pi \in \mathcal{N} \wedge o \in \Omega_a \\ [\![CS, a, d_o(\pi), \sigma \mid e']\!] & \text{si } e = \pi.e', \text{ avec } o \in \Omega_a \\ [\![op]\!]([\![CS, a, o, \sigma \mid e_1]\!], ..., [\![CS, a, o, \sigma \mid e_n]\!]) & \text{si } e = op(e_1, ..., e_n) \\ \text{nil} & \text{sinon} \end{cases}$$

Le cas nil correspond à une tentative d'accès à un objet ne faisant pas partie de la mémoire locale de l'agent Ω_a.

Lors de l'exécution d'un $look(dc_p, dc_r, e)$, les expressions contenues dans dc_p et dc_r (par définition, il ne peut s'agir que de variables de contexte) et e - dites expressions de contexte - sont évaluées dans le même contexte que les expressions locales, plus une unification particulière σ qui associe les variables de contexte qui figurent dans e à des objets.

Définition L'évaluation d'une expression de contexte

$[\![CS, a, o, \sigma \mid e]\!] =$

$$\begin{cases} v & \text{si } e = v, \text{ avec } v \in type_i \\ d_o(\pi) & \text{si } e = \pi, \text{ avec } \pi \in \mathcal{N} \wedge o \in \Omega_a \\ [\![CS, a, d_o(\pi), \sigma \mid e']\!] & \text{si } e = \pi.e', \text{ avec } o \in \Omega_a \\ [\![op]\!]([\![CS, a, o, \sigma \mid e_1]\!], ..., [\![CS, a, o, \sigma \mid e_n]\!]) & \text{si } e = op(e_1, ..., e_n) \\ \\ [\![CS, a, \sigma(x), \sigma \mid e']\!] & \text{si } e = x.e', \text{ avec } x \in \mathcal{X} \wedge type(x) = \Omega \\ \sigma(x) & \text{si } e = x, \text{ avec } x \in \mathcal{X} \wedge type(x) = \Omega \\ \text{nil} & \text{sinon} \end{cases}$$

Lors de l'exécution de $add(sds, e_p, e_r)$ les expressions $s \in \mathcal{S}$ sont évaluées, et lors de l'exécution d'un ou de $look(dc_p, dc_r, e)$, pour chaque objet o unifié avec une variable dans dc_p, $e_p(o)$ est évaluée afin de vérifier si l'agent a le droit de percevoir l'objet, et pour chaque objet unifié avec une variable dans dc_r, $e_r(o)$ est évaluée afin de vérifier si l'agent a le droit de le recevoir. Les expressions s, e_p et e_r sont dites expressions de sécurité, elles sont évaluées dans le même contexte que les expressions locales, plus l'objet apparié avec *that* : il s'agit de l'objet ajouté dans le cas du *add* et de l'objet à recevoir ou à percevoir dans le cas d'un *look*.

Définition L'évaluation d'une expression de sécurité

$[\![CS, a, o, o^* \mid e]\!] =$

$$\begin{cases} v & \text{si } e = v, \text{ avec } v \in type_i \\ d_o(\pi) & \text{si } e = \pi, \text{ avec } \pi \in \mathcal{N} \wedge o \in \Omega_a \\ [\![CS, a, d_o(\pi), \sigma \mid e']\!] & \text{si } e = \pi.e', \text{ avec } o \in \Omega_a \\ [\![op]\!]([\![CS, a, o, \sigma \mid e_1]\!], ..., [\![CS, a, o, \sigma \mid e_n]\!]) & \text{si } e = op(e_1, ..., e_n) \\ \\ [\![CS, a, o^*, o^* \mid e']\!] & \text{si } e = that.e' \\ \text{nil} & \text{sinon} \end{cases}$$

Remarque 7 *Remarquons l'absence des variables libres des fonctions d'évaluation. En effet, l'utilisation de l'opérateur $\nu X(P)$ fait que le cas où nous évaluons une expressions avec des variables libres, autres que les variables de contexte n'est jamais rencontré, puisque toutes les variables seront liées avec des valeurs de leur domaine avant qu'une évaluation ne soit tentée.*

Soit l'expression de contexte $e = (x.destination = compagnon.destination)$ exécutée dans le contexte d'un agent a ayant un *compagnon* (référence) avec pour destination "Rennes", et avec un objet o^* de l'environnement tel que $d_{destination}(o^*) =$ "*Rennes*". L'unification $\sigma \equiv \{x \leftarrow o^*\}$, et l'évaluation $[\![CS, a, a, \sigma \mid e]\!]$ de e se déroule comme illustré par la figure 5.2, pour être évaluée à vrai.

5.3.2 Sémantique opérationnelle

La sémantique du langage de coordination définit l'évolution de l'état du SMA sous l'effet des actions des agents qui y évoluent. La sémantique opérationnelle du langage décrit comment les processus sont dérivés et de quelle manière ils affectent l'état du SMA. Elle est construite en utilisant un système de transitions étiquetées, suivant des axiomes et des règles écrites dans le style

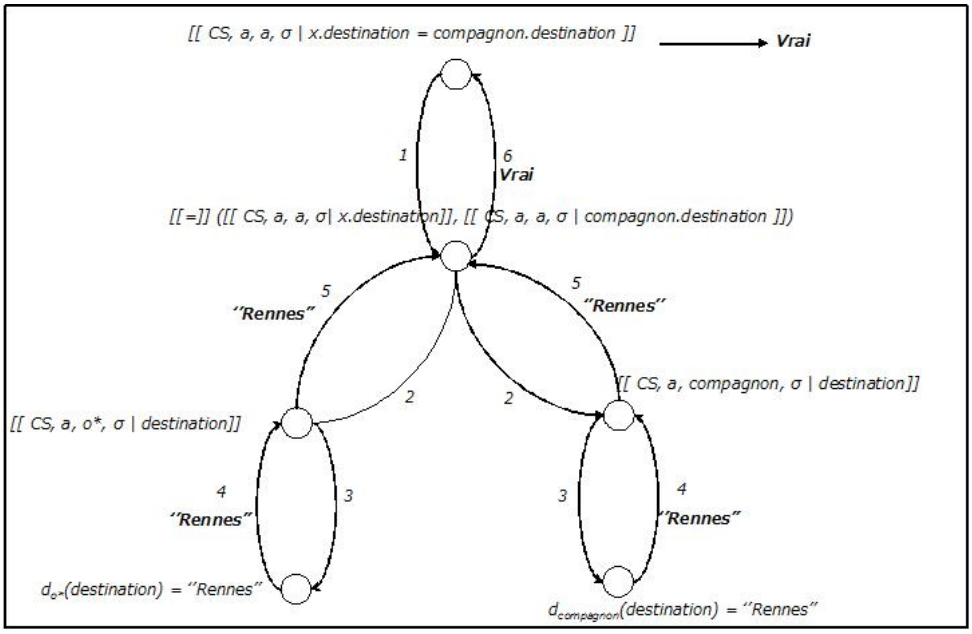

FIG. 5.2 – Evaluation d'une expression

SOS [Plotkin, 1981]. Les étiquettes sont les actions de coordination du language, éventuellement enrichies avec des informations concernant l'agent appelant ou l'objet apparié.

Nous définissons la sémantique opérationnelle de notre langage comme le système de transitions étiquetées $(\Sigma, Act, \longrightarrow)$ où Σ est l'ensemble de systèmes coordonnés possibles, $Act = \{\tau, \alpha, o : \alpha, a : o : \alpha\}$, a étant un agent, o un objet, τ l'action interne, et $\alpha \in \{spawn(P, ds), add(ds, e_p, e_r), look(ds_p, ds_r), update(ds)\}$. La relation de transition $\longrightarrow \subseteq \Sigma \times Act \times \Sigma$ est définie comme la plus petite satisfaisant les axiomes et les règles définis ci-après.

Afin d'alléger la présentation, nous ne décrivons pas tout l'état d'un CS dans les règles de transition, mais seulement les composantes qui sont concernées par des modifications. Nous formulons en langue naturelle la partie de CS concernée afin de décrire les composantes affectées. Ainsi, pour décrire l'action d'un agent a du SMA, nous utilisons "CS par a", pour décrire l'action d'un processus particulier P, nous disons "en utilisant P", pour décrire la considération d'une expression particulière e (paramètre d'un processus), nous utilisons "étant donné e", et pour identifier un ensemble d'objets particuliers C considéré avec une expression donnée, nous utilisons "sur C". Pour décrire un CS dans lequel le processus considéré a changé en P, nous utilisons "CS avec P". Nous accolons une apostrophe aux composantes d'un système ayant changé d'état suite au passage d'une transition. Par exemple, si $\mathcal{A}$ est l'ensemble des agent dans CS, $\mathcal{A}'$ est l'ensemble des agents dans CS' (et $\mathcal{A}$" est l'ensemble des agents dans CS" etc.).

Les cinq premières règles décrivent l'évolution d'un système coordonné sous l'action de l'une des primitives du langage (*spawn*, *add*, *look* et *update*). Une action *spawn* crée un nouvel agent (indépendant de l'agent appelant) pour lequel il spécifie la description et le comportement. L'effet de $spawn(P', sds)$ est de lancer un agent qui se comporte comme P' et a comme description l'évaluation de la description symbolique sds (axiome R_{spawn}). L'évaluation d'une description symbolique est la description résultant de l'évaluation de chaque expression dans sds.

$$(R_{spawn})CS \text{ par } a \text{ en utilisant } spawn(P', sds).P \xrightarrow{spawn(P',ds)} CS' \text{ avec } P$$

où $sds = \{\pi \leftarrow e_\pi\}_{\pi \in \mathcal{N}}, ds = \{\pi \leftarrow [\![CS, a, a \mid e_\pi]\!]\}_{\pi \in \mathcal{N}}$,
avec $\mathcal{A}' = \mathcal{A} \uplus \{a'\}$ où a' est un nouvel agent, avec $proc(a') = P', d_{a'} \equiv ds, \Omega_{a'} = \{a'\}$

Une action $add(sds, e_p, e_r)$, si elle ne viole aucune des règles de sécurité dans $\mathcal{S}$, ajoute un objet à l'environnement, l'objet a comme description l'évaluation de sds (règle (R_{add_1})). Les restrictions de perception et de réception de l'objet lui sont associées (e_p et e_r).

$$(R_{add_1}) \quad CS \text{ par } a \text{ en utilisant } add(sds, e_p, e_r).P \xrightarrow{add(ds, e_p, e_r)} CS' \text{ avec } P$$
$$\text{si } \forall s \in \mathcal{S}, [\![CS, a, a, o' \mid s]\!] = \text{vrai}$$

où $sds = \{\pi \leftarrow e_\pi\}_{\pi \in \mathcal{N}}, ds = \{\pi \leftarrow [\![CS, a, a, o' \mid e_\pi]\!]\}_{\pi \in \mathcal{N}}$,
avec $\Omega'_{ENV} = \Omega_{ENV} \uplus \{o'\}$où o' est un nouvel objet, avec $d_{o'} \equiv ds, e_p(o') \equiv e_p, e_r(o') \equiv e_r$

Si la description sds de l'objet o' ajouté viole une règle de sécurité, l'effet de $add(sds, e_p, e_r)$ est nul et aucun objet n'est ajouté dans l'environnement. Le processus add devient nul ($\mathbf{0}$) avec une transition interne τ (règle R_{add_2}).

$$(R_{add_2}) \quad CS \text{ par } a \text{ en utilisant } add(sds, e_r, e_p).P \xrightarrow{\tau} CS \text{ avec } P$$
$$\text{si } \exists s \in \mathcal{S}, [\![CS, a, a, o' \mid s]\!] \neq \text{vrai}$$

où $sds = \{\pi \leftarrow e_\pi\}_{\pi \in \mathcal{N}}, ds = \{\pi \leftarrow [\![CS, a, a, o' \mid e_\pi]\!]\}_{\pi \in \mathcal{N}} o'$ est un nouvel objet, avec $d_{o'} \equiv ds$

Une action $look(dc_p, dc_r, e)$ est dérivée suivant l'axiome (R_{look}). Étant donné C un ensemble d'objets sélectionnés de l'environnement tels que $card(C) = card(varent(e))$, une unification σ associant un objet de C à chaque variable de $varent(e)$.

Pour que $look(dc_p, dc_r, e)$ soit exécutée, il faut que :
- l'expression e soit évaluée à vrai avec l'unification σ,
- les conditions de perception et de réception qui sont associées aux objets dans C doivent être évalués à vrai.

Le test vérifiant que les restrictions de perception et de réception associées aux objets dans C ne sont pas violés par l'exécution de $look(dc_p, dc_r, e)$, étant donnée une unification σ, est formulé comme suit. $C_p = \{\sigma(x)\}_{x \in varent(dc_p)}$ est l'ensemble des objets sélectionnés de l'environnement et destinés à être perçus par l'agent a et $C_r = \{\sigma(x)\}_{x \in varent(dc_r)}$ est l'ensemble des objets sélectionnés de l'environnement et destinés à être reçus par l'agent a. Il faut que $\forall o \in C_p [\![CS, a, a, o \mid e_p(o)]\!] = \text{vrai}$ et $\forall o \in C_r [\![CS, a, a, o \mid e_r]\!] = \text{vrai}$. Si l'action est un succès, les objets dans C sont ajoutés dans la mémoire locale de l'agent et les objets $C_r \subseteq C$ sont supprimés de l'environnement. La description de l'agent a est augmentée des références vers les objets perçus et reçus.

La règle enrichit l'étiquette de la transition avec l'objet sélectionné et est donc étiquetée $o : look(ds_p, ds_r)$:

$$(R_{look}) \; CS \text{ par } a \text{ sur } C \text{ en utilisant } look(dc_p, dc_r, e).P \xrightarrow{o:look(ds_p, ds_r)} CS' \text{ avec } P$$
$$\text{si } \forall o \in C_p [\![CS, a, a, o \mid e_p(o)]\!] = \text{vrai} \wedge \forall o \in C_r [\![CS, a, a, o \mid e_r(o)]\!] = \text{vrai}$$

avec $C \subseteq \Omega_{ENV}, card(C) = card(varent(e)), [\![CS, a, a, \sigma \mid e]\!] = \text{vrai}, dc = \{\pi \leftarrow x_\pi\}_{\pi \in \mathcal{N}}, ds = \{\pi \leftarrow \sigma(x_\pi)\}_{\pi \in \mathcal{N}}, \sigma \equiv \{(x \leftarrow o)\}_{x \in varent(e)}$

où $\Omega'_{ENV} = \Omega_{ENV} - C_r, \Omega'_a = \Omega_a \uplus C_r$,

$$d'_a \equiv \left\{ \pi \leftarrow v \mid v = \left\{ \begin{array}{ll} v & \text{si } ds(\pi) \neq \text{nil} \\ d_a(\pi) & \text{sinon} \end{array} \right\}_{\pi \in \mathcal{N}} \right.$$

Une action $update(sds)$ change simplement la valeur des propriétés π vers l'évaluation de e_π.

$$(R_{update}) CS \text{ par } a \text{ en utilisant } update(sds).P \xrightarrow{update(ds)} CS' \text{ avec } P$$

où $sds = \{\pi \leftarrow e_\pi\}_{\pi \in \mathcal{N}}, ds = \{\pi \leftarrow [\![CS, a, a \mid e_\pi]\!]\}_{\pi \in \mathcal{N}}$,

$$d'_a \equiv \left\{ \pi \leftarrow v \mid v = \left\{ \begin{array}{ll} v & \text{si } ds(\pi) \neq \text{nil} \\ d_a(\pi) & \text{sinon} \end{array} \right\}_{\pi \in \mathcal{N}} \right.$$

La règle R_ν spécifie qu'un processus P contenant des variables libres dans X se transforme par l'action de νX en un processus P où toutes les variables sont liées à une valeur dans leur domaine de définition. $P[\{x_\nu/x\}_{x \in X}]$ dénote le processus P où toutes les occurrences libres des variables dans X sont remplacées par des valeurs dans leur domaine (notées x_ν) résultant de l'application de νX.

$$(R_\nu) CS \text{ par } a \text{ en utilisant } \nu X(P) \xrightarrow{\tau} CS \text{ avec } P[\{x_\nu/x\}_{x \in X}]$$

La règle du choix spécifie que, si l'expression gardant un processus P est évaluée à vrai alors $b\lfloor P \rfloor$ peut se transformer en P avec une action interne τ (règle (R_{choix})).

$$(R_{choix}) CS \text{ par } a \text{ en utilisant } b\lfloor P \rfloor + b'\lfloor Q \rfloor \xrightarrow{\tau} P \quad \text{si } [\![CS, a, a, \emptyset \mid b]\!] = \text{vrai}$$

La règle pour la composition parallèle $(R_{parallel})$ est la règle traditionnelle, sans synchronisation : chaque processus, composé parallèlement avec un autre, peut exécuter ses actions d'une manière indépendante.

$$(R_{parallel}) \quad \frac{CS \text{ par } a \text{ en utilisant } P1 \xrightarrow{\alpha} CS' \text{ avec } P1'}{CS \text{ par } a \text{ en utilisant } P1 \| P2 \xrightarrow{\alpha} CS' \text{ avec } P1' \| P2}$$

La règle d'invocation de processus $(R_{invocation})$ dit que, si une constante K définit un processus P, alors quelque soit l'action α, si P peut l'exécuter et se transformer en P', alors K peut l'exécuter également, et se transformer en P'.

$$(R_{invocation}) \quad \frac{CS \text{ par } a \text{ en utilisant } P \xrightarrow{\alpha} CS' \text{ avec } P'}{CS \text{ par } a \text{ en utilisant } K \xrightarrow{\alpha} CS' \text{ avec } P'} \text{si } K \stackrel{def}{=} P$$

La règle (R_{agent}) dit que le processus résultant de la transition α par l'agent a constitue son nouveau comportement.

$$(R_{agent}) \quad \frac{CS \text{ par } a \text{ en utilisant } proc(a) \xrightarrow{\alpha} CS' \text{ avec } P'}{CS \text{ par } a \xrightarrow{\alpha} CS''}$$

avec $proc(a)'' = P'$

Enfin, la règle $(R_{systeme})$ élimine l'agent considéré de la configuration prémisse et enrichit l'étiquette de la transition par l'agent considéré. Le résultat est une règle « globale » faisant cor-

respondre un SMA coordonné dans un état CS avec un SMA coordonné dans un nouvel état CS'.

$$(R_{systeme}) \quad \frac{CS \text{ par } a \xrightarrow{\alpha} CS'}{CS \xrightarrow{a:\alpha} CS'}$$

5.4 Implémentation du langage

Nous avons introduit un langage de coordination qui pourrait être implémenté dans n'importe quel langage de programmation. La procédure habituelle d'implémentation d'un langage de coordination est de fournir des librairies dans un langage de programmation hôte qui peuvent être utilisées par d'autres programmes qui désirent être coordonnés selon le modèle (e.g. Klava associé à Klaim [De Nicola *et al.*, 1998] vu au chapitre 1).

Néanmoins, afin de tirer un profit maximal de la sémantique opérationnelle associée au langage, il est plus intéressant de ne pas exiger du programmeur de veiller lui-même au respect de cette sémantique dans chaque système qu'il réalise. Ceci est effectué en lui offrant un outil lui permettant d'écrire un programme avec une syntaxe très proche de Lacios, et de générer un système prêt à être utilisé, avec la garantie qu'il adhère bien au modèle de coordination. Notamment, nous désirons utiliser les opérateurs de préfixage, de choix, de composition parallèle et de liaison des variables libres comme décrits par la syntaxe, dans la définition du comportement des agents. En nous fondant sur la syntaxe et la sémantique de Lacios présentés dans le présent chapitre et le chapitre précédent, nous pourrions réaliser des compilateurs pour Lacios vers n'importe quel langage cible. Ainsi, le même programme Lacios peut virtuellement être exécuté avec n'importe laquelle des implémentations. Il est à noter qu'il s'agit de la même approche poursuivi par Sunna dans son langage Claim [Suna, 2005].

Notre choix du langage de programmation cible comme illustration d'une implémentation de Lacios s'est porté sur Java. Ce choix est motivé par la simplicité de création et de gestion de processus concurrents (les Threads), ainsi que la facilité de création de parseurs, grâce au générateur de parseurs Java JavaCC[32].

5.4.1 Description générale

Un programme Lacios est un fichier texte où sont décrits les comportements des agents ainsi que le système coordonné. Un système coordonné est défini par l'ensemble des agents initiaux, lancés au début de l'exécution, ainsi que les règles de sécurité $\mathcal{S}$. La syntaxe exacte d'un fichier programme, ainsi que le mode d'utilisation de l'outil peuvent être trouvés en annexe A.

Nous avons implémenté un parseur qui, à partir d'un fichier script Lacios génère le programme Java correspondant. Le fonctionnement général est illustré par la figure 5.3. Les programmeurs rédigent les scripts Lacios décrivant le comportement de leurs agents. Les scripts sont parsés et compilés, générant un objet programme Java. Lors de l'exécution, une application externe peut manipuler l'objet programme et éventuellement lier les variables libres des agents le composant.

Le parseur est généré avec JavaCC, un générateur de parseurs pour Java. Un fichier JavaCC définit une grammaire permettant une analyse syntaxique du code et permet d'intégrer du code Java lors de l'analyse syntaxique associant des instructions Java à la reconnaissance des éléments du langage, permettant ainsi une analyse sémantique du code.

[32] `http://javacc.dev.java.net/`

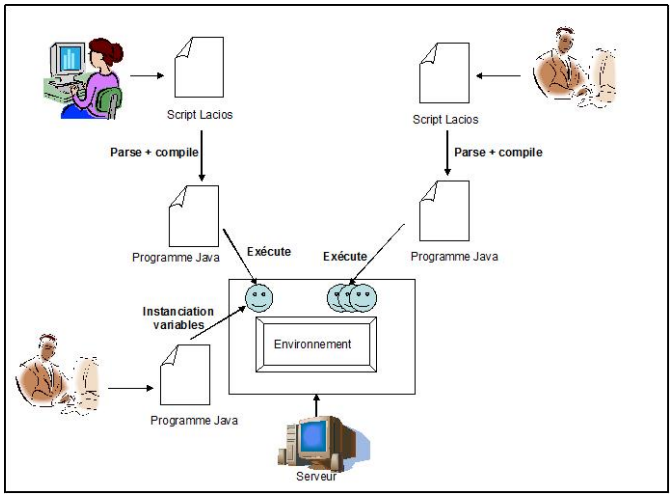

FIG. 5.3 – Fonctionnement général

5.4.2 Structure de données

L'objectif du parseur est de générer un programme composés d'objets prêts à être exécutés. Nous avons défini une structure de données vers laquelle les instructions Lacios sont traduites.

Lorsqu'une description est *sds* est rencontrée dans le code du programme Lacios, un objet de la classe Description est créé. La classe Description est implémenté sous forme d'une table de hachage (classe java.util.HashMap), ayant pour clé le nom de la propriété, et pour contenu la valeur associée (un objet Java). Une entité (classe Entity) est un objet Java qui a comme seul attribut sa description (c.f figure 5.4).

Dans la syntaxe de Lacios, nous accédons aux propriétés d'une entité via un point ($\omega.\pi$ désigne la propriété π de l'entité ω). Lorsqu'une entité suivie d'un point est rencontrée dans le code ($\omega.\pi$), nous accédons à la table de hachage description de l'entité ω et nous récupérons la valeur associée à la clé π.

Un objet est une entité ayant deux objets Expressions relatifs aux restrictions d'accès e_p et e_r. Un agent est également une entité qui a un attribut additionnel *behavior*, qui est le processus (classe Processus) définissant son comportement.

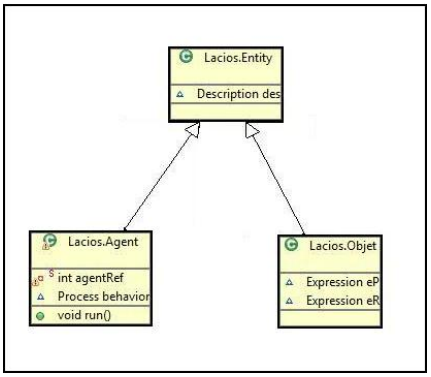

FIG. 5.4 – Classes Entité, Agent et Objet

Les comportements des agents sont des Processus (c.f figure 5.5), extension des Threads Java. Comme définie dans la syntaxe de Lacios, une propriété sans préfixe rencontrée dans le code

d'un processus désigne une propriété de l'agent qui exécute le processus. Par exemple, lorsque dans le code d'un processus, nous rencontrons *budget* + 10, il s'agit de la propriété *budget* de l'agent qui est en train d'exécuter ce processus. Pour avoir accès à cette information, chaque processus maintient un attribut *contexte*, désignant l'agent qui est en train de l'exécuter.

Différentes classes implémentent les processus Lacios. Un Processus peut être un objet de la classe EmptyProcess, i.e. un processus terminé. Il peut être également un objet Choice, auquel cas il est défini par une Expression booléenne et deux Processus correspondant aux comportements associés aux clauses alors et sinon, respectivement. Un Processus peut être un objet de la classe Parallel, ses attributs sont les deux Processus P et Q qui doivent s'exécuter en parallèle. Un Processus peut être un ProcessCall, un appel de processus. Dans ce cas, il a un attribut désignant la définition du processus appelé. Enfin, un processus peut être un Prefix, i.e. une action suivie d'un processus. Chaque Processus redéfinit la méthode *run()* qui lance le comportement qui lui est associé. Par exemple, un objet Parallel lance ses deux processus attributs en parallèle et un objet Prefix exécute l'action avant de lancer le processus.

Une Action peut être un Spawn, un Add, un Look, un Update ou un NU (relatif à $\nu X(P)$) (c.f figure 5.6). Chaque action implémente une méthode *excuter*() qui lance les instructions le définissant.

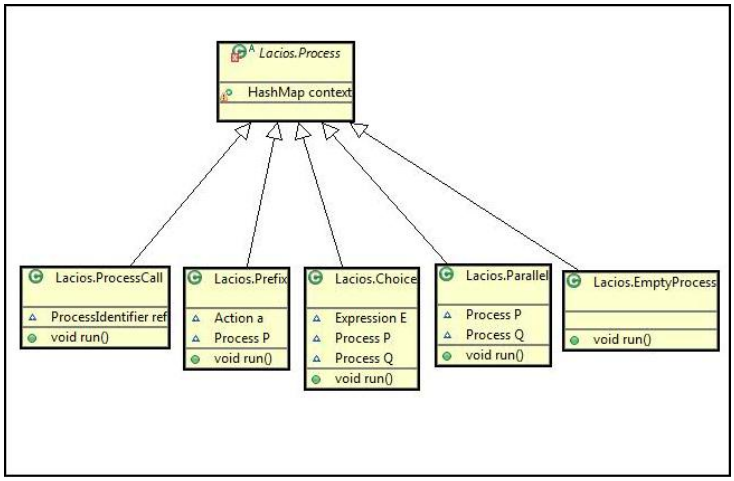

Fig. 5.5 – Les processus

5.4.3 Opérateurs

Les opérateurs de Lacios sont des instances de la classe Operator. Les opérateurs existant par défaut dans le langage reprennent l'ensemble des opérateurs de Java qui traitent des types primitifs (+, *, / etc.). Il est également donné au programmeur la possibilité de définir ses propres opérateurs. Pour ce faire, il lui faut définir et compiler une classe héritant de Operator ayant les attributs arité, nom et un tableau d'expressions paramètres de l'opérateur afin d'être reconnu par le compilateur. Dans son code Lacios, le programmeur peut faire appel à cet opérateur avec la syntaxe suivante : <nom_de_la_classe>(<paramètres>).

5.4.4 Ouverture

Lorsqu'un opérateur $\nu(X)P$ est rencontré dans le code d'un agent, un objet de la classe NU est créé. Il a comme attribut l'ensemble des variables X. Le comportement P ne peut être lancé

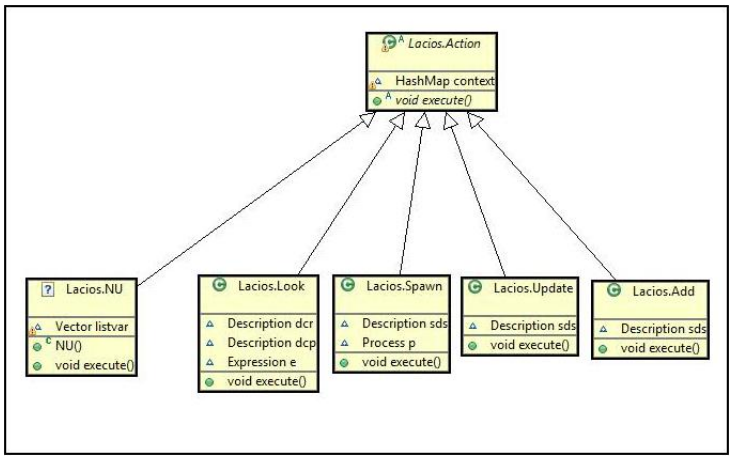

FIG. 5.6 – Les actions

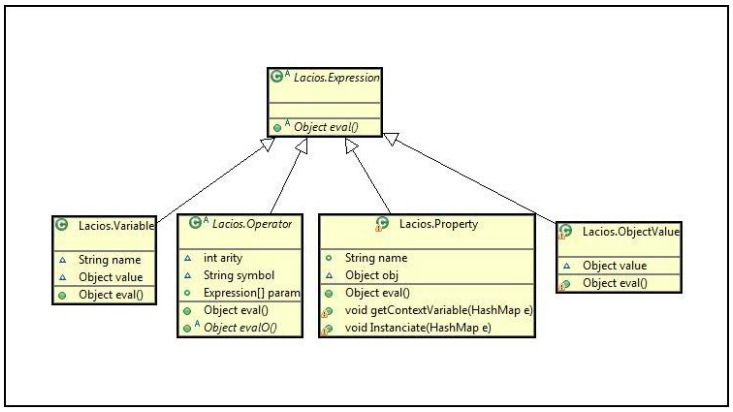

FIG. 5.7 – Les expressions

que si les variables X prennent une valeur. Ainsi, pour chaque variable, un Thread est lancé, dont le comportement est de rester bloqué (avec un $wait()$), jusqu'à ce la variable prenne une valeur.

De l'autre côté, chaque agent a une méthode $instantiate(HashMaph)$ qui permet d'associer des valeurs aux variables libres de ses processus. Dès qu'un *instantiate* est invoqué, les processus bloqués sur les différentes variables libres concernées sont réveillés, et le reste du processus est repris (c.f figure 5.8. Si un appel à *instantiate* est effectué avant qu'un NU ne soit exécuté par l'agent, l'appel est bloqué jusqu"à ce que l'agent demande une liaison de ses variables.

Ainsi, l'application interagissant avec le SMA peut à tout moment faire appel à la méthode *instantiate* d'un agent en lui fournissant une table de hachage (couples *variable-valeur*).

5.4.5 Appariement et respect de la sémantique

Respect de la sémantique

L'implémentation du langage est guidée par la sémantique que nous lui avons définie dans ce chapitre. Les principaux points qui doivent être pris en compte dans l'implémentation sont les suivants :

– Le caractère non bloquant des actions *add*, *spawn* et *update*,

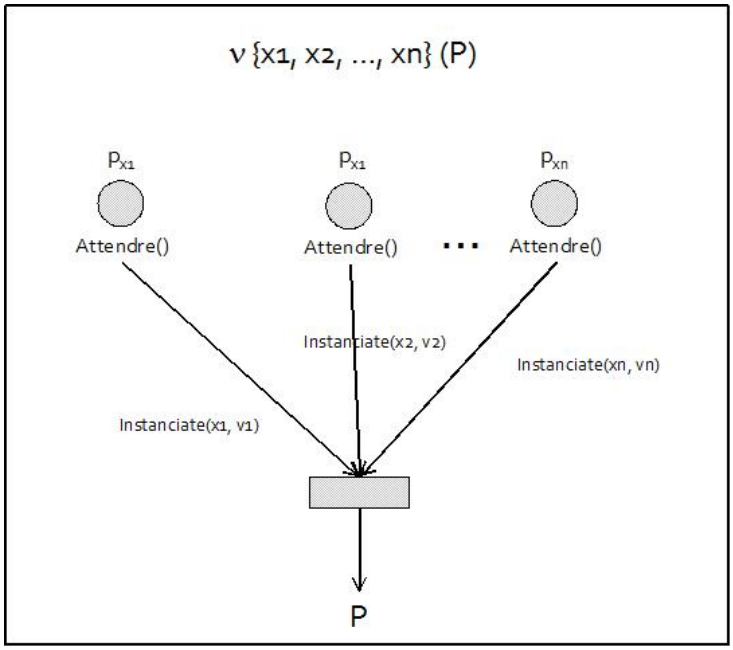

FIG. 5.8 – Traitement des variables libres

– L'obligation pour *look* d'être exécutée avec les valeurs des propriétés courantes de l'agent,
– Le caractère bloquant de *look*,
– L'impossibilité de la réception du même objet par plusieurs agents.

Les agents ont une référence vers un même objet Environnement, disposant des méthodes *add* et *look*. La méthode *exécuter()* d'un objet Add consiste en un appel de la méthode *add* de l'environnement. L'accès concurrent à l'ensemble d'objets de l'environnement avec un *look* nécessite une synchronisation des appels *add* et *look*. En revanche, un processus appelant *add* ne doit pas être mis en attente jusqu'au passage de l'environnement à un état cohérent. Nous définissons pour ce faire une zone tampon vers laquelle les agents peuvent ajouter leurs objets sans se bloquer, le vidage de la zone tampon est - lui - synchronisé avec les appels de la méthode *look*. La méthode *exécuter()* d'un objet Spawn crée un agent et le lance sans interaction avec l'environnement, aucun problème particulier n'est donc à résoudre la concernant.

Un objet Look fait appel à la méthode *look* de l'environnement. Chaque *look* dépose un verrou sur l'état de l'environnement tant qu'il est en cours de recherche d'un appariement, afin de garantir qu'il accède à un état cohérent des données, et de veiller à ce qu'un même objet ne soit pas reçu par plusieurs agents. Si aucun appariement n'est trouvé, le processus appelant la méthode (le Processus Look) est bloqué (*wait*() de Java). Les expressions des différents Look bloqués sur l'environnement sont sauvegardés comme un attribut de l'environnement. Les processus bloqués sont réveillés ($notifyAll()$ de Java) lorsqu'un *add* ajoute un objet à l'environnement. Dans ce cas, le processus réveillé ne doit pas réessayer des appariements déjà effectués, mais continue ses tentatives d'appariement à partir du point où il s'est arrêté.

Un *update* met à jour les propriétés de l'agent, son action est donc locale à l'agent, mais elle peut influencer son interaction avec l'environnement. En effet, si un appel de la méthode *look* est en cours d'exécution, les tentatives d'appariement doivent être effectuées avec les propriétés actuelles de l'agent. Lors du changement d'une propriété de l'agent, les tentatives ultérieures d'appariement se font avec les nouvelles valeurs mais il se pourrait que des appariement qui ont échoués auparavant deviennent un succès avec les nouvelles valeurs de propriétés. Afin de veiller

à ce que ces appariements soient tentés, le comportement associé à la méthode *exécuter()* de l'action Update vérifie si des expressions en cours d'exécution sont concernées par la mise à jour. Si tel est le cas, l'environnement est notifié via la méthode $notifyPropertyChanges$ qui prend un vecteur des propriétés concernées par la mise à jour. L'état d'avancement des expressions bloquées de l'agent sont remises à zéro et réveillées afin de réessayer de trouver un appariement satisfaisant avec les nouvelles propriétés.

Appariement

Un environnement est défini par l'ensemble d'objets qu'il contient, une liste d'expressions en attente (bloquées sur des *look* qui ne trouvent pas d'appariement valide), et un ensemble de règles de sécurité $\mathcal{S}$ (expressions). L'environnement a trois méthodes : *look*, *add* et $notifyPropertyChanges$. Ci-après la description des méthodes *look* et *add*.

La méthode $look(dc_p, dc_r, e)$: chaque agent désirant percevoir ou recevoir un objet appelle la méthode *look* en lui passant en paramètre l'expression à évaluer ainsi que les deux descriptions de contexte relatifs aux objets à percevoir et à recevoir. L'algorithme correspondant à *look* (c.f algorithme 4) consiste à tenter de trouver un appariement valide pour l'expression en unifiant ses variables avec les objets de l'environnement (en appelant la fonction $match(e)$). L'ensemble des unifications possibles pour e étant donné un ensemble d'objets dans l'environnement (nous utilisons Ω_{ENV} pour les désigner). Si aucune unification valide dans $e.\Sigma$ n'est trouvée, le processus appelant est mis en attente, et l'expression est ajoutée à la liste d'expressions en attente. Lorsqu'une expression est valide, l'ensemble des objets à recevoir sont supprimés de l'environnement. Les objets supprimés doivent l'être également de toutes les unifications Σ des expressions en attente.

Algorithme 4 look

Entrées : Descriptions de contexte dc_p, dc_r, Expression e
Sorties : Descriptions ds_p, ds_r

```
e.Σ ← Ω_ENV^card(varent(e))
σ ← match(e)
tant que σ ≠ null faire
    ajouter e dans la liste des expressions en attente
    attendre()
    σ ← match(e)
fin tant que
ds_p ← dc_p.lier(σ)
ds_r ← dc_r.lier(σ)
Enlever les objets dans ds_r de l'environnement, et des unifications des expressions en attente.
```

L'algorithme 5 relatif à la fonction $match(e)$ consiste à tenter des unifications à la recherche d'une qui valide l'expression. Si tel est le cas, l'unification σ qui a permis de valider l'expression est renvoyée.

$unificationSuivante()$ renvoie l'unification suivante de taille $card(varent(e))$ qui associe un objet de l'environnement à chaque variable d'entité dans e.

$add(o)$: lors de l'ajout d'un objet, l'environnement vérifie que celui-ci ne viole aucune règle de sécurité du système. Si tel est le cas, les expressions en cours d'évaluation sont notifiées de l'ajout de l'objet, pour l'intégrer dans leurs unifications ultérieures. Une fois l'objet validé par

Algorithme 5 match

Entrées : Expression e
Sorties : Unification σ
  tant que $\sigma' = unificationSuivante() \neq null$ **faire**
    $e.lier(\sigma')$
    $e.eval()$
  fin tant que
  $\sigma \leftarrow \sigma'$

les règles de sécurité, il est rajouté à l'environnement et les processus bloqués pour un *look* sont réveillés. Si l'agent viole une règle de sécurité, l'objet n'est pas ajouté dans l'environnement.

Algorithme 6 add

Entrées : objet o
  $valide \leftarrow$ vrai
  pour tout $s \in \mathcal{S}$ tant que $valide$ **faire**
    $s.lier(\{that \leftarrow o\})$
    $valide \leftarrow valide \wedge s.eval()$
  fin pour
  si $valide$ **alors**
    Ajouter o dans l'environnement
    $\forall e$ en attente, notifier l'ajout de o
    Réveiller les processus en attente
  fin si

5.5 Complexité de l'appariement

La complexité de l'appariement d'une expression e contenant une seule variable d'entité est de l'ordre de $O(n)$, avec n le nombre d'objets présents dans l'environnement, puisqu'au pire des cas, l'expression doit être testée avec l'ensemble des objets à la recherche d'un appariement. En présence de m variables d'entité dans une expression, la complexité devient de l'ordre de $O(n^m)$, puisque il s'agit d'un arrangement avec répétition, où toutes les m-combinaisons d'objets doivent être testées en vue d'un appariement avec e, et où un objet peut être unifié avec plusieurs variables. Par exemple, si l'environnement contient deux objets o et o', et qu'une expression contient deux variables x et y, les unifications possibles sont : $\{\{x \leftarrow o, y \leftarrow o\}, \{x \leftarrow o', y \leftarrow o'\}, \{x \leftarrow o, y \leftarrow o'\}, \{x \leftarrow o', y \leftarrow o\}\}$, soit 2^2 combinaisons.

Nous proposons de limiter la complexité de l'appariement suivant la structure des objets de l'environnement. En effet, il est inutile de tester un objet n'ayant pas de propriété $fumeur$ par exemple, avec l'expression $x.fumeur =$ faux, puisque selon la fonction d'évaluation définie dans la sémantique de Lacios, le résultat de l'appariement sera de toutes les manières égal à nil, et sera inévitablement un échec. L'idée afin de limiter la complexité de l'appariement c'est, lors de la soumission d'une expression à l'environnement (avec un *look*), d'extraire les propriétés sur lesquelles elle porte, et de ne tester e pour appariement qu'avec les objets de l'environnement pour lesquelles cette propriété est définie, i.e. différente de nil.

5.5.1 Première solution : création d'index

La première solution que nous proposons pour limiter le nombre d'objets sur lesquels porte le test d'une expression, c'est de créer un index : une liste de propriétés, où chaque propriété pointe sur les objets de l'environnement pour lesquels elle est définie.

L'index est utilisé comme suit. Lors de l'ajout d'un objet o, nous créons des références vers o depuis chacune des propriétés qui sont définies pour lui. Ainsi, une instruction est ajoutée dans la procédure *add* (algorithme 6 présenté ci-haut), après l'ajout de l'objet dans l'environnement :

$ajouter\_index(o)$

L'algorithme 7 relatif à la procédure $ajouter\_index(o)$ ajoute l'objet o (ayant pour propriétés différentes de nil $propriétés(o)$) à l'index. L'accès à l'index se fait par propriété et renvoie un ensemble de références d'objets (noté $index[p]$ avec p une propriété).

Algorithme 7 ajouter_index

Entrées : Objet o
  pour tout $p \in propriétés(o)$ **faire**
    si $index$ = null **alors**
      créer une nouvelle entrée dans l'index avec comme clé p et comme seul élément o
    sinon
      $index[p] \leftarrow index[p] \cup o$
    fin si
  fin pour

Lors de la soumission d'une expression e (avec *look*), les variables d'entités et les propriétés qui sont concernées pour chacune des variables sont extraites. Pour chaque variable d'entité de $varent(e)$, seul un sous-ensemble des objets de l'environnement risque de valider l'expression, celui pour lequel toutes les propriétés demandées pour la variable est défini. Ainsi, pour chaque variable d'entité v de $varent(e)$, l'ensemble des propriétés p qui sont définies pour elle sont extraites (noté $propriétés(e, v)$). Pour chacune de ces propriétés, l'ensemble des objets qu'elle pointe dans l'index est extrait. Les objets qui risquent de former une instanciation valide pour v sont ceux pour lesquels l'ensemble des propriétés p est défini, il s'agit de l'intersection des objets pointés par les propriétés p. Les combinaisons d'objets qui sont considérées avec l'expression e sont le produit cartésien des ensembles d'objets ainsi définis.

L'instruction suivante remplace la première instruction de l'algorithme 4 décrit précédemment :

$e.\Sigma \leftarrow extraire\_objets(e)$

Algorithme 8 extraire_objets

Entrées : Expression e
Sorties : Ensemble d'unifications Σ
  pour tout $v \in varent(e)$ **faire**
    $objets \leftarrow \bigcap_{p \in propriétés(e,v)} index[p]$
  fin pour
  $\Sigma \leftarrow objets^{card(varent)}$

5.5.2 Deuxième solution : définition d'un ensemble statique de propriétés

La première solution que nous venons de présenter est applicable dans le cas général, où nous ne disposons d'aucune information sur l'ensemble de propriétés que peuvent avoir les objets de l'environnement. Lorsque l'ensemble des propriétés définies pour les objets de l'environnement est relativement stable, nous pourrions améliorer le processus d'appariement, en calculant les combinaisons de propriétés en *off-line*.

Définition PROPRIÉTÉS D'OBJETS

Nous définissons $\eta \subset \mathcal{N}$ l'ensemble fini de propriétés supportées par l'environnement. L'ensemble des propriétés décrivant chaque objet ajouté dans l'environnement doit être inclus ou égal à η.

Remarque 8 *La limitation des propriétés des objets ajoutés dans l'environnement ne s'applique pas aux propriétés des agents. Ces dernières restent incluses dans l'ensemble $\mathcal{N}$ de propriétés.*

À partir de l'ensemble η de propriétés d'objets, l'ensemble des parties de η est calculé, qui constitue le nouvel index, l'accès à l'index se fait avec un ensemble de propriétés, et non une seule. L'index est structuré sous forme d'un treillis. Il existe un arc reliant l'ensemble des propriétés P_1 à l'ensemble de propriétés P_2 si $|card(P_1) - card(P_2)| = 1$ et si $card(P_1 \cap P_2) = 1$. On dit que P_1 est le père (resp. fils) de P_2 s'il existe un arc entre P_1 et P_2 et que $card(P_1) < card(P_2)$ (resp. $card(P_2) < card(P_1)$). La figure 5.9 illustre la structure d'un index avec quatre propriétés *position*, *catégorie*, *capacité* et $fumeur$. Dans cette figure, {*position*,*catégorie*} est un père de {*position*,*catégorie*, $fumeur$}.

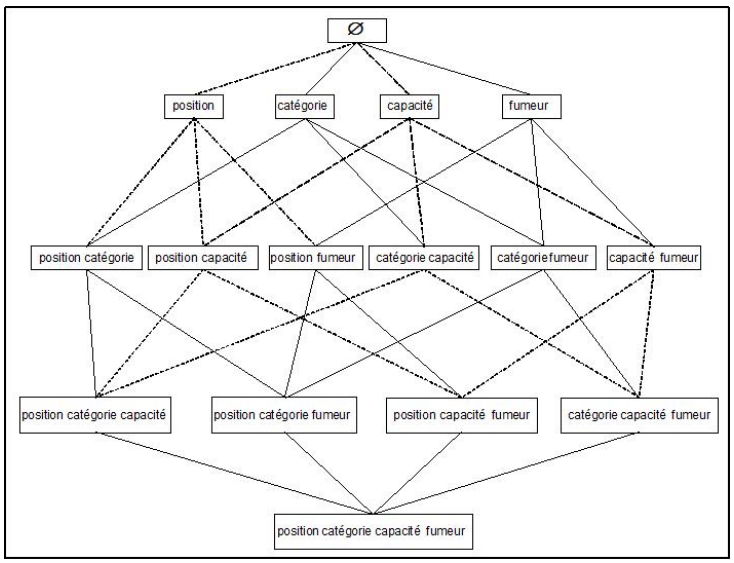

FIG. 5.9 – Treillis de propriétés

Lors de l'exécution d'un *add*, une référence vers l'objet o ajouté est créée au niveau du noeud n du treillis comprenant toutes ses propriétés. Cette référence est propagée au niveau de tous les noeuds pères de n jusqu'au suprémum du treillis (noeud correspondant à l'ensemble vide - zéro propriétés).

Algorithme 9 ajouter_treillis

Entrées : Objet o
$p \leftarrow propriétés(o)$
$treillis[p] \leftarrow treillis[p] \cup o$
propager des références vers o vers tous les noeuds du treillis pères de $treillis[p]$

Lors de l'exécution de *look*, il suffit, pour chaque variable d'entité de l'expression, d'accéder au noeud du treillis correspondant à l'ensemble des propriétés recherchées. La récupération de l'ensemble d'objets nécessaires à l'appariement n'implique plus le calcul de toutes les intersections de propriétés pour chaque variable d'entité, comme présenté plus haut, mais revient simplement à une recherche du noeud concerné dans le treillis. Lors de la recherche d'un appariement, il suffit d'accéder, pour chaque variable, au noeud du treillis contenant toutes les propriétés définies pour elle.

Afin d'appliquer cette méthode, l'instruction au début de *look* devient :
$e.\Sigma \leftarrow extraire\_treillis(e)$

Algorithme 10 extraire_treillis

Entrées : Expression e
Sorties : Ensemble d'objets Obj
pour tout $v \in varent(e)$ **faire**
$objets \leftarrow index[propriétés(e, v)]$
fin pour
$\Sigma \leftarrow objets^{card(varent(e))}$

5.5.3 Troisième solution : définition d'un ensemble statique de catégories

La troisième solution est applicable lorsque non seulement l'ensemble des propriétés définies pour les objets de l'environnement est stable, mais que nous disposons d'un certain nombre de combinaisons de propriétés qui sont concernés par des ajouts d'objets.

Définition Catégories d'objets

κ est l'ensemble fini de catégories d'objets. Une catégorie $k \in \kappa$ est définie comme un ensemble fini de propriétés $\{\pi \in \eta\}$.

Dans la suite de ce paragraphe, nous donnons quelques définitions empruntées à l'analyse formelle de concepts. Le lecteur intéressé par ce domaine est prié de se référer à e.g. [Ganter and Wille, 1999] pour plus de détails. Nous commençons par la définition d'un contexte formel.

Définition Contexte formel

Un contexte formel est un triplet $\langle \eta, \kappa, \mathcal{R} \rangle$, où η est l'ensemble des propriétés et κ est l'ensemble des catégories d'objets, le couple $\langle i, k \rangle \in \mathcal{R}$ signifie que $i \in k$.

Reprenons l'exemple des agents voyageurs. Nous définissons κ comme contenant les catégories suivantes :

$$train = \{id, catégorie, capacité, destination, position, prix_{ind}, prix_{groupe}\}$$

	id	cat	cap	$dest$	pos	$prix_{ind}$	$prix_{groupe}$	$budg$	fum	$train$
$train$	×	×	×	×	×		×	×		
$promotion$	×	×	×	×		×				
$voyageur$	×	×		×				×	×	×
$cafe$	×	×	×		×				×	
$salle$	×	×	×		×					

TAB. 5.5 – Un exemple de relation $\mathcal{R}$

$$promotion = \{id, catégorie\,capacité, destination, prix_{ind}\}$$
$$voyageur = \{id, catégorie, destination, budget, fumeur, train\}$$
$$cafe = \{id, catégorie, capacité, position, fumeur\}$$
$$salle\ d'attente = \{id, catégorie, capacité, position\}$$

La relation $\mathcal{R}$ correspondante à cette définition des catégories est présentée dans le tableau 5.5.

La propriété *catégorie* est définie pour permettre à un agent de percevoir un objet d'une certaine catégorie sans se soucier des valeurs prises par ses propriétés. Par exemple, $look(\{untrain \leftarrow x\}, \emptyset, x.catgorie = \text{“}train\text{”})$ cherche un objet ayant des valeurs de propriétés quelconques de la catégorie train.

Afin de calculer la relation entre propriétés et catégories, deux applications h et g sont définies comme suit (dans la suite, nous nous référons à h et g par $'$) :

- $g : 2^{\kappa} \rightarrow 2^{\eta}$
- $g(K) = K' = \{i \in \eta | \forall k \in K, (i,k) \in \mathcal{R}\}$
- $h : 2^{\eta} \rightarrow 2^{\kappa}$
- $h(E) = E' = \{k \in \kappa | \forall i \in E, (i,k) \in \mathcal{R}\}$

$g(K)$ associe à K les propriétés qui sont définies pour toutes les catégories $k \in K$, et $h(E)$ associe à E toutes les catégories pour lesquelles toutes les propriétés $i \in E$ sont définies. La paire d'applications $\langle h, g\rangle$ définit une connexion de Galois entre 2^{η} et 2^{κ} ($h \circ g$ et $g \circ h$ sont notés $''$ dorénavant).

Dans notre exemple, $g(\{train, cafe\}) = \{id, cat, cap, pos\}$ et $h(\{id, fum\}) = \{cafe, voy\}$

Définition CONCEPT FORMEL

Un concept formel du contexte $\langle \eta, \kappa, \mathcal{R}\rangle$ est un couple $\langle E, K\rangle$, tel que $E' = K$ et $K' = E$. K et E sont appelés, respectivement, l'intention et l'extension du concept formel $\langle X, Y\rangle$

Par exemple, $h \circ g(\{train, cafe\}) = \{train, cafe\}'' = \{train, cafe, salle\}, \{train, cafe, salle\} \times \{id, cat, cap, pos\}$ est un concept formel.

L'ensemble des concepts formels ainsi définis forme un treillis qui représente les relations mutuelles que les différentes catégories ont les unes avec les autres, et leurs propriétés.

Définition TREILLIS DE CONCEPTS FORMELS

Soit $\mathcal{C}$ l'ensemble de concepts formels construits à partir du contexte formel $\langle \eta, \kappa, \mathcal{R}\rangle$.
Le couple $\mathcal{T}_{\mathcal{C}} = \langle \mathcal{C}, \leq\rangle$ est un treillis de concepts formels. La relation d'ordre partiel $\leq$ est définie comme suit :
$\forall c_1 = \langle \eta_1, \kappa_1\rangle, c_2 = \langle \eta_2, \kappa_2\rangle \in \mathcal{C}, c_1 \leq c_2$ ssi $\kappa_2 \subseteq \kappa_1 (\Leftrightarrow \eta_1 \subseteq \eta_2)$

$train$	$\times$	$id, cat, cap, dest, pos, prix_i, prix_g$	
$promo$	$\times$	$id, cat, cap, dest, prix_i$	✓
voy	$\times$	$id, cat, dest, budg, fum$	
$cafe$	$\times$	id, cat, cap, pos, fum	
$salle$	$\times$	id, cat, cap, pos	✓

TAB. 5.6 – L^1

La relation d'ordre partiel est utilisée afin de construire le graphe du treillis, appelé diagramme de Hasse, comme suit : il existe un arc entre deux noeuds étiquetés c_1 et c_2 si $c_1 \preceq c_2$, où $\preceq$ est la réduction transitive de $\leq$, i.e. $c_1 \preceq c_2, c_1 \leq c_3 \leq c_2 \Rightarrow c_1 = c_3$ ou $c_3 = c_2$.

L'analyse formelle de concepts est un domaine très actif, et un travail considérable a été effectué sur les algorithmes de génération de treillis à partir d'une relation binaire. Parmi les algorithmes de calcul des concepts et de construction du treillis, nous pouvons citer Bordat [Bordat, 1986], Ganter [Ganter, 1984], Chein [Chein, 1969], Norris [Norris, 1978], Godin [Godin *et al.*, 1995] et Nourine [Nourine and Raynaud, 1999]. Le lecteur intéressé peut se référer à, e.g. [Fu and Mephu Nguifo, 2004], [Guénoche, 1990], [Kuznetsov and Obiedkov, 2002] ou encore [Zargayouna and Bsiri, 2002] pour un état de l'art de ces algorithmes.

La complexité de la construction d'un treillis est exponentielle par rapport au nombre de propriétés, et ce quelque soit l'algorithme utilisé. Cependant, les catégories étant fixées par le concepteur, avant le démarrage de l'application, le calcul du treillis se fait une seule fois en *off-line*, sa complexité n'est donc pas un handicap, d'autant plus que le nombre de propriétés que nous traitons est relativement petit, en comparaison aux milliers (voire millions) d'objets manipulés en analyse formelle de concepts. La Nous avons choisi l'algorithme de Chein [Chein, 1969] pour calculer l'ensemble de concepts formels (c.f 11. L'algorithme est itératif. À chaque étape k, l'ensemble des concepts L^{k+1} est construit à partir de l'ensemble L^k. Les éléments de L^{k+1} sont obtenus par combinaison de deux éléments de L^k, en calculant l'union de leur intension et l'intersection de leur extension si elle n'est pas vide. Les éléments de L^k dont l'extension est contenue dans au moins un élément de L^{k+1} ne sont pas des concepts fermés et sont non considérés ultérieurement (ils sont marqués). Si L^{k+1} contient moins de deux éléments, l'algorithme s'arrête, sinon il passe à l'itération suivante. L'ensemble des concepts formels est l'ensemble des éléments non marqués. Initialement, L^1 est l'ensemble des catégories et les propriétés correspondantes définis dans la relation $\mathcal{R}$.

Ci-après l'illustration du déroulement de l'algorithme avec notre exemple. Dans ce qui suit, $id, cat, cap, dest, budg, fum, pos$ dénotent respectivement les propriétés *identifiant, catégorie, capacité, destination, budget, fumeur* et *position*, et $promo, voy, salle$ dénotent les catégories *promotion, voyageur* et *salle*.

L'ensemble des concepts formels dérivés de la définition de nos catégories est donc :
$\{\{train\}, \{cafe\}, \{voy\}, \{train, promo\}, \{cafe, salle\}, \{train, promo, voy\}, \{train, cafe, salle\}, \{train, promo, cafe, salle\}, \kappa\}$

Dans la figure 5.10, nous donnons le diagramme (le treillis) correspondant à cet ensemble de catégories. Chaque noeud est un couple $intension \times extension$ et la base du treillis est $\perp \times \eta$, puisqu'il n'existe aucune catégorie pour laquelle toutes les propriétés sont définies. L'algorithme de construction du treillis est simple, une fois les concepts formels découverts. En effet, chaque

Algorithme 11 Découverte des concepts formels(Algorithme de Chein)

Entrées : Relation $\mathcal{R}$
Sorties : Ensemble de concepts formels $\mathcal{L}$

```
L^1 ← x_i × g(x_i)_{i=1...n}
k ← 1
tant que |L^k| > 1 faire
  L^{k+1} ← ∅
  pour tout i < j indices d'éléments de L^k non marqués faire
    Y_ij ← Y_i ∩ Y_j
    si Yij ≠ ∅ alors
      si Yij ∈ L^{k+1} alors
        X_ij ← X_ij ∪ X_j
      sinon
        L^{k+1} ← L^{k+1} + X_i ∪ X_j × Y_ij
      fin si
      si Y_ij = Y_i alors
        Marquer X_i × Y_i dans L^k
      fin si
      si Y_ij = Y_j alors
        Marquer X_j × Y_j dans L^k
      fin si
    fin si
  fin pour
fin tant que
L est l'ensemble des éléments non marqués
```

$promo, train$	$\times$	$id, cat, cap, prix_i$	
$voy, promo, train$	$\times$	$id, cat, dest$	
$cafe, train, salle$	$\times$	id, cat, cap, pos	
$cafe, promo$	$\times$	$id, cat, cap, prix_i$	✓
$cafe, voy$	$\times$	id, cat, fum	
$salle, voy$	$\times$	id, cat	✓

TAB. 5.7 – L^2

$promo, train, voy, cafe$	$\times$	id, cat	✓
$cafe, train, salle, promo$	$\times$	id, cat, cap, pos	

TAB. 5.8 – L^3

$promo, train, voy, cafe, salle$	$\times$	id, cat	

TAB. 5.9 – L^4

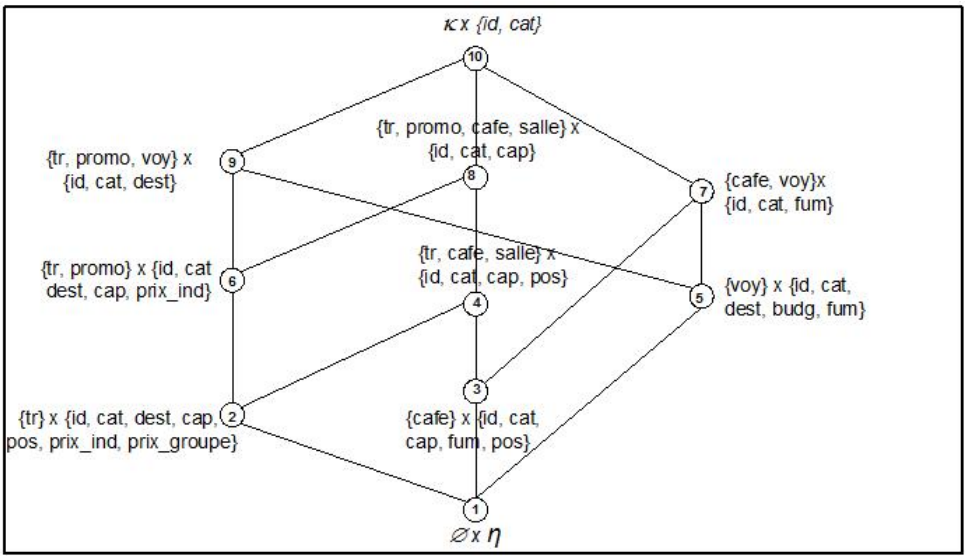

FIG. 5.10 – Le treillis de Galois dérivé depuis $\mathcal{R}$

noeud correspond à un concept formel, les pères d'un noeud sont ses ancêtres diminués des ancêtres de ses ancêtres. L'ancêtre d'un noeud comprend tous les concepts dont l'intension couvre la sienne.

Aucune modification supplémentaire n'est nécessaire dans les fonctions *look* et *match*, puisqu'il suffit d'appeler *ajouter_treillis* et *extraire_treillis* au début de *add* et de *look* respectivement, comme c'était le cas pour la deuxième solution proposée.

Perspectives d'implémentation : distribution de l'environnement

Dans la suite de nos travaux, nous proposons d'utiliser le treillis de concepts que nous venons de définir dans la structuration de plusieurs environnements partagés. Nous décidons de ne pas fractionner l'environnement en plusieurs espaces, en cohérence avec notre choix de garder un partage total des données de l'environnement. Au niveau architectural, nous disposons de trois possibilités d'implémentation d'environnements partagées. La première possibilité correspond au modèle Linda original, qui propose une architecture centrée sur le *tuplespace* comme le montre la figure 5.11. Tous les agents sont connectés sur un espace de données commun.

Les inconvénients de cette architecture sont multiples. Le premier inconvénient est relatif à la tolérance aux pannes. En effet, si l'hôte où l'espace de données est hébergé est défaillant, les agents n'ont plus aucun moyen d'interagir, et le système ne peut plus fonctionner correctement. Le deuxième inconvénient est relatif au congestionnement des traitements au niveau de l'espace de données. L'espace de données centralisé constitue un goulet d'étranglement de la communication au sein du système. Enfin, un environnement centralisé supporte mal le passage à l'échelle. L'augmentation du nombre d'agents et d'objets dans l'environnement augmente sa charge, et l'appariement devient de plus en plus coûteux, ce qui cause un ralentissement de l'exécution, qui ne devient plus supportable au delà d'un certain seuil.

La deuxième possibilité correspond à l'architecture proposée par les modèles à *tuplespaces* multiples. Ils proposent une architecture où l'espace de données est distribué sur différents hôtes comme le montre la figure 5.12. Les inconvénients de cette architecture sont relatifs à la limitation du partage des données, qui est l'un des principaux intérêts des modèles orientés-données. En

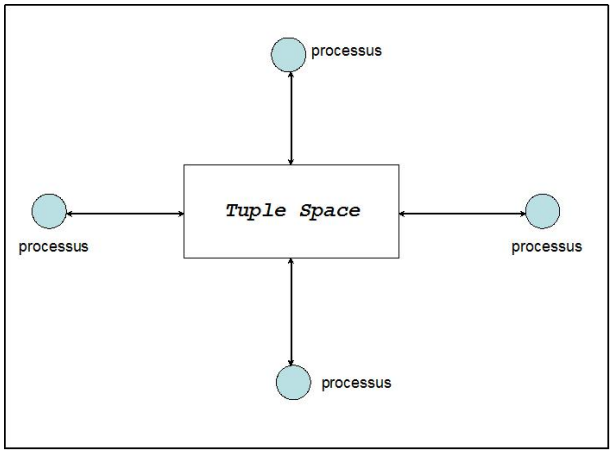

FIG. 5.11 – Architecture d'un modèle à *Tuplespaces*

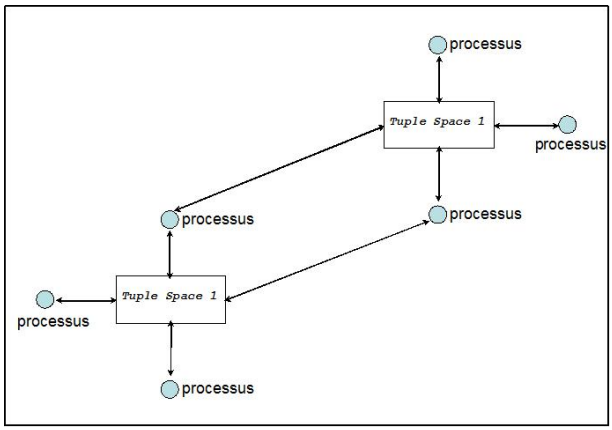

FIG. 5.12 – Architecture d'un modèle à *Tuplespaces* multiples

effet, les agents évoluant dans un *tuplespace* donné sont isolés par rapport aux autres, comme évoluant dans des applications distinctes. Les agents, afin d'interagir avec un espace de données, doivent connaître son emplacement ou son identifiant pour pouvoir envoyer et récupérer des données. Or, l'un des intérêts des modèles orientés-données, c'est qu'un agent n'a pas à connaître l'emplacement des autres agents pour pouvoir interagir avec eux. Ainsi, face à une multiplication des *tuplespaces* un agent est confronté au même problème que dans un système ouvert totalement distribué : comment découvrir les espaces de données pertinents et les autres agents évoluant dans le système, avec qui il pourrait avoir une interaction profitable.

Notre proposition de structurer les catégories d'objets de l'environnement sous forme d'un treillis de concepts offre une alternative à ces deux manières de distribuer l'environnement. L'environnement est partagé sur différents hôtes (nous disposons de plusieurs environnements) mais les agents, en interagissant avec n'importe lequel de ces environnements, ont accès à l'union de leur contenu. Chaque hôte contiendrait un sous-treillis connexe, et avec chaque noeud, l'ensemble des objets ayant les propriétés définies pour ce noeud sont stockés. Cette solution limiterait les problèmes liés à la centralisation et garantit un partage total des données entre les agents. Lorsqu'un *look* est exécuté, il l'est sur un sous-ensemble limité des noeuds du treillis - probablement sur le même hôte - et lorsqu'un *add* est exécuté, la duplication est limitée au noeud concerné

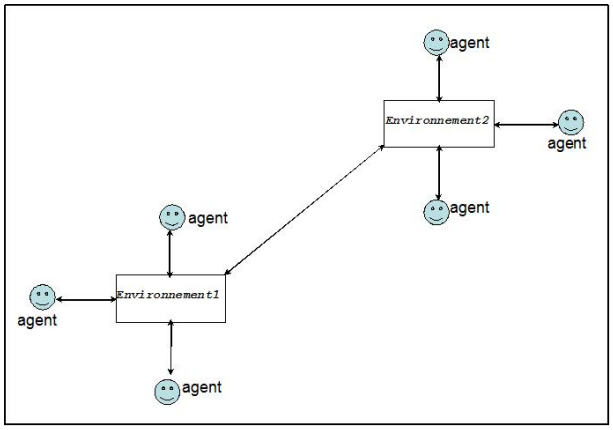

FIG. 5.13 – Perspectives architecturales du modèle Acios

ainsi qu'à ses pères, soit un sous-ensemble réduit des hôtes du réseau.

5.6 Conclusion

Dans ce chapitre, nous avons proposé un langage de coordination relatif au modèle de coordination Acios. La sémantique opérationnelle offre une spécification non ambiguë du comportement des agent d'un SMA écrit avec notre langage. Elle a servi de repère pour l'implémentation du langage au dessus de Java. Dorénavant, l'implantation d'un système adhérant au modèle Acios devient beaucoup plus aisée, implique beaucoup moins d'effort de programmation, limite grandement la taille des codes, et nécessite moins d'effort de débuggage et de maintenance.

Chapitre 6

Le système Lacios-TAD

Sommaire

6.1 Introduction

Dans ce chapitre, nous traitons du problème dynamique de tournées de véhicules avec fenêtres temporelles (D-VRPTW) et du Transport À la Demande (TAD) introduits au chapitre 3. Notre système, appelé Lacios-TAD[33], est un système adhérant au modèle Acios et écrit dans le langage Lacios pour une résolution multi-agent de ces deux problèmes.

La contribution de ce chapitre est double. D'une part, ce chapitre propose une illustration de l'usage du langage Lacios pour la résolution du problème dynamique de tournées de véhicules avec fenêtres temporelles et du problème TAD. Le système réalise une version distribuée des heuristiques d'insertion parallèle vues au chapitre 3. D'autre part, nous proposons une nouvelle

[33]Ce travail est effectué au sein de l'unité de recherche Gretia - institut de recherche Inrets

mesure d'insertion qui pourrait être appliquée avec différentes heuristiques, pour choisir un véhicule parmi les candidats pour l'insertion d'un même client.

Ce chapitre suit le plan suivant. Dans la section 6.2, nous donnons les motivations de ce chapitre. Dans la section 6.3, nous décrivons le fonctionnement général du système, nous y présentons les paramètres de l'application, les expressions permettant le respect des contraintes du problème ainsi que la mesure de calcul du prix d'une insertion que nous introduisons. La section 6.4 explicite les détails relatifs à la mise en oeuvre du système tels que la gestion du temps, les comportements des agents et le code Lacios correspondant. La description de l'outil et la validation de la mesure du calcul du prix d'insertion sont reportées dans la section 6.5.

6.2 Motivation

Nous avons vu dans le chapitre 3 que les problèmes D-VRPTW et son extension TAD sont des problèmes d'une grande complexité et qui se prêtent bien à une modélisation distribuée. Dans le contexte d'une modélisation SMA, les heuristiques d'insertion sont celles qui ont été le plus adaptées afin de fournir des solutions rapidement (e.g. les systèmes ADART [Dial, 1995], MARS [Fischer *et al.*, 1994] et In-Time [Kohout and Erol, 1999]). Lors de l'adaptation d'une heuristique d'insertion à une modélisation distribuée, le protocole naturel à mettre en place est un *Contract Net Protocol* (CNP) : les clients s'annoncent, reçoivent des prix d'insertion des véhicules et en choisissent celui qui propose le prix le plus bas. C'est également le protocole adopté pour la proposition qui fait l'objet de ce chapitre.

L'adoption d'un modèle de coordination orienté-données pour la réalisation d'un tel système est motivée principalement par le caractère dynamique et naturellement ouvert du problème. En effet, les clients rejoignent le système dynamiquement, et les véhicules sont créés au fur et à mesure de la résolution. Afin de limiter les interactions et la mise à jour des connaissances des agents les uns des autres, nous croyons qu'il est pertinent de mettre ces connaissances en commun dans l'environnement SMA sous forme de descriptions, et que les véhicules et les clients découvrent leurs interlocuteurs d'une manière associative, sans les connaître *a priori*.

Grâce à la structure de données retenue pour Acios (*propriétés-valeurs*), et à la syntaxe des expressions (utilisation des propriétés et des opérateurs), un véhicule peut restreindre sa perception aux seuls clients qu'il peut effectivement insérer dans son plan, en transcrivant les contraintes du problème sous forme d'expressions Lacios.

La majorité des systèmes distribués de la littérature utilisant des heuristiques d'insertion que nous avons présentés dans le chapitre 3 séquentialisent le processus de traitement d'un client. En effet, tous les véhicules du SMA sont mobilisés pour l'insertion d'un seul client à la fois avant de passer au suivant. Grâce à l'utilisation des expressions Lacios, les agents représentant des véhicules qui ne peuvent pas insérer un nouveau client ne le perçoivent pas, et peuvent donc se porter candidats pour l'insertion d'autres clients, en parallèle. La mise en place d'un tel système avec le langage de coordination Lacios est relativement simple et nécessite peu de code. De plus, grâce à l'utilisation des variables libres et de l'opérateur de liaison de variables ν, le code relatif au comportement des agents est indépendant du contexte applicatif. Le résultat est que le même code peut être utilisé dans le cadre d'une simulation, comme dans notre implémentation, mais également dans le cadre d'un système opérationnel, interagissant avec un serveur Web par exemple, et ce sans rien changer au code du programme Lacios.

Le caractère myope des heuristiques d'insertion a été souligné dans le chapitre 3. Pour pallier ce problème, nous proposons une modification des heuristiques d'insertion classiques concernant la manière de calculer le prix d'insertion d'un client dans le plan d'un véhicule. Nous proposons

une nouvelle mesure visant à prendre en compte le futur dans le calcul de ce prix fondée sur le champ de perception des agents.

6.3 Description du système

Dans cette section, nous décrivons notre système en mettant l'accent sur son principe de fonctionnement sans entrer dans tous les détails de sa mise en oeuvre avec le langage Lacios. Nous donnons ses paramètres, son fonctionnement général, ainsi que la manière avec laquelle le respect des contraintes du problème est assuré. Nous introduisons également le calcul du prix d'insertion d'un client dans le plan d'un véhicule.

6.3.1 Paramètres

L'environnement physique d'un système de transport est essentiellement le réseau routier dans lequel évoluent ses différents acteurs (véhicules, conducteurs, voyageurs, etc.). Pour sa représentation, nous utilisons les conventions standards de la théorie des graphes. Ainsi le réseau est réduit à un graphe avec un ensemble d'arcs et de noeuds. Étant donné que différents niveaux de détails sont envisageables, plusieurs représentations sont possibles. Néanmoins, la granularité de la représentation est non pertinente dans notre problème, du moment que ces conditions sont satisfaites [Diana and Dessouky, 2004] :

- il existe un noeud du graphe qui peut être associé à chaque point de départ (ainsi qu'éventuellement à chaque point d'arrivée) d'un client,
- le graphe est connexe.

Les paramètres du système reprennent les données du problème présentées dans le chapitre 3. Rappelons-en les principaux composants. Soit $[e_0, l_0]$ l'horizon d'ordonnancement du problème, toutes les fenêtres temporelles doivent être comprises entre ces deux bornes. Pour le problème VRPTW, chaque client i est décrit par un tuple $(n_i, [e_i, l_i], s_i, q_i)$ où : n_i est le noeud du réseau concerné par la demande, q_i est le nombre de personnes qui posent cette demande, e_i est l'instant le plus petit auquel le(s) client(s) peuvent être au niveau de n_i, l_i est le moment le plus tardif où le(s) client(s) veulent être desservis et s_i est le temps de service de la demande. Pour le problème TAD, un client i est décrit par deux tuples $(n_{i1}, [e_{i1}, l_{i1}], s_{i1}, q_{i1})$ et $(n_{i2}, [e_{i2}, l_{i2}], s_{i2}, q_{i2})$, correspondant aux points de ramassage et de livraison du client.

Le temps de service d'un client est le temps que le véhicule est obligé de passer au niveau du noeud visité, avant de pouvoir le quitter. Ce temps de service peut être une fonction du nombre de voyageurs posant la demande, il représenterait donc le temps qui leur est nécessaire afin de monter à bord. Il peut également représenter le temps nécessaire pour charger un client dans le cas des personnes à mobilité réduite. Une autre signification du temps de service est la suivante. Souvent, le graphe considéré par le problème est une abstraction simplifiée du réseau routier réel. En effet, le réseau physique peut être décomposé en régions, et le centre de chaque région représenté par un noeud du graphe. En recevant une demande qui concerne une région donnée, elle est associée au noeud correspondant, et le temps de service associé au client représente le temps de parcours nécessaire depuis l'emplacement du noeud à l'emplacement réel du client.

Chaque véhicule a une capacité maximale Q. Deux matrices de coûts sont définies, la matrice des temps de parcours et la matrice des distances entre chaque couple de noeuds du réseau. L'objectif est de trouver une ensemble de plans de véhicules permettant de desservir tous les clients. Le critère à optimiser est d'abord la minimisation de la taille de la flotte de véhicules, et ensuite la distance totale parcourue par les véhicules.

6.3.2 Fonctionnement général

Notre système est composé d'un ensemble dynamique d'agents agissant sur un environnement partagé. Tous les échanges d'informations entre agents passent par l'environnement. Nous définissons quatre catégories d'agents. Les agents Client (AC) représentant des clients, les agents Véhicule (AV) représentant des véhicules, l'agent Dépôt (AD) représentant le dépôt et l'agent Interface (AI) qui représente une borne d'accès au système (serveur Web, interface graphique, simulateur, etc.). Les AC sont créés dynamiquement par l'AI lors de la connexion d'un utilisateur au système et les AV sont créés dynamiquement par l'agent Dépôt lorsqu'un AC ne peut être inséré dans le plan d'aucun des AV présents dans le système.

Toutes les données relatives aux clients sont des propriétés d'un AC : n, e, l, s et q pour un système modélisant un problème VRPTW, et n_1, e_1, l_1, s_1, q_1, n_2, e_2, l_2, s_2 et q_2 pour un système modélisant un problème TAD. Les noeuds du réseau sont représentés par des identifiants entiers uniques, et la matrice des temps de parcours est implantée sous forme d'un opérateur

$$cout : \mathbb{N}^2 \rightarrow \mathbb{R}^+$$

$cout(n_1, n_2)$ renvoie le temps de parcours entre le noeud ayant pour identifiant n_1 et le noeud ayant pour identifiant n_2. Chaque couple de noeuds (n_1, n_2) du graphe dispose d'une valeur $cout(n_1, n_2)$ correspondante, i.e. *cout* renseigne les temps de parcours de tous les couples de noeuds du réseau.

Les AC déposent des objets les représentant dans l'environnement afin d'être perçus par les autres agents du SMA, renseignant leurs descriptions.

La figure 6.1 illustre les propriétés additionnelles définies pour les objets représentant les AC dans l'environnement. Les objets représentant des AC ont une propriété *id* renseignant l'identifiant de l'AC que l'objet représente. Chaque objet client a une propriété *veh* qui renseigne l'identifiant de l'AV au plan duquel appartient le client, et deux propriétés *succ* et *pred* qui renseignent les identifiants des clients immédiatement successeurs et immédiatement prédécesseurs de ce client dans le plan du véhicule. Un objet représentant un AC qui n'est pas encore associé au plan d'un AV a des propriétés *veh* égale à $unknown_{veh}$, et *succ* et *pred* égales à $unknown_{client}$. Ainsi, le plan d'un véhicule peut être reconstitué à partir des descriptions des objets représentant des AC dans l'environnement.

Le fonctionnement général du système suit les étapes suivantes :

1. un AC ajoute un (ou plusieurs) objets le représentant dans l'environnement,
2. le (ou les) objet(s) est (sont) perçu(s) par zéro ou plusieurs AV,
3. les AV l'ayant perçu proposent des prix d'insertion à l'AC,
4. Si l'AC a reçu des offres d'insertion des AV, alors aller à l'étape 4,
5. s'il ne reçoit aucune offre :
   (a) il demande à l'agent Dépôt de créer un nouvel AV,
   (b) il reçoit une offre d'insertion du nouvel AV,
6. l'AC choisit l'AV proposant le prix le plus bas,
7. l'AV choisi insère l'AC dans son plan, et le processus recommence à l'étape 1.

Ce fonctionnement général est valable autant pour le D-VRPTW que pour le TAD.

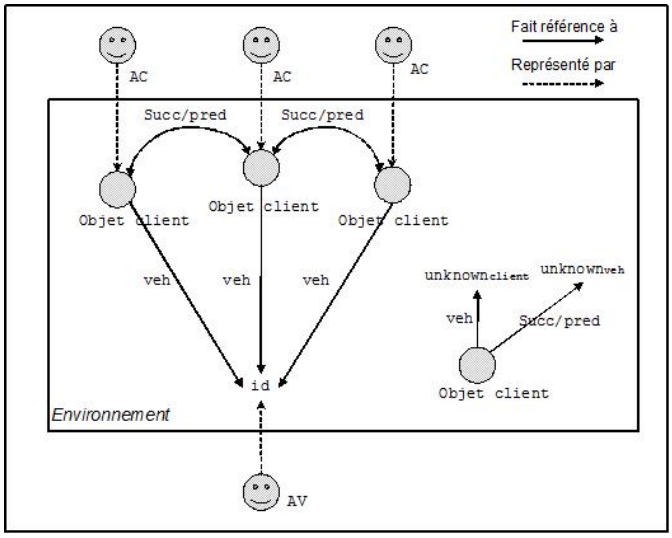

FIG. 6.1 – Sous-ensemble des objets de l'environnement

6.3.3 Les contraintes

L'utilisation du modèle Acios a l'avantage de limiter le nombre d'AV candidats pour un client à ceux pour lesquels son insertion est possible au vu des contraintes du problème et de l'état actuel du véhicule, i.e. les clients précédemment insérés dans son plan. Pour ce faire, un AV se fonde sur les descriptions des objets déposés par les AC dans l'environnement pour définir une expression *exp* associé à un *look* qui limite sa perception aux objets représentant des AC insérables dans son plan.

Ce paragraphe détaille la prise en compte des contraintes du problème VRPTW, les contraintes additionnelles du TAD sont introduites ultérieurement.

Dans le problème VRPTW, une insertion d'un client c dans le plan d'un véhicule v est valide si :

- la quantité demandée par c ne dépasse pas la capacité courante de v,
- il existe une position dans le plan de v entre deux clients adjacents c_1 et c_2 telle que v peut desservir c_1, ensuite c et enfin c_2 sans violer les fenêtres temporelles des trois clients.

Ainsi, pour qu'un client c puisse être inséré après un client c_1 dans le plan du véhicule v, il faut que la capacité du véhicule ne soit pas dépassée avec l'insertion de c, i.e. :

$$c.q \leq v.cap$$

et que :

$$c_1.e + c_1.s + cout(c_1.n, c.n) \leq c.l$$

Pour que le client c puisse être inséré avant un client c_2, il faut que la capacité du véhicule ne soit pas dépassée avec l'insertion de c et que :

$$c.e + c.s + cout(c.n, c_2.n) \leq c_2.l$$

Par conséquent, pour qu'un client c puisse être inséré après c_1 et avant c_2, il faut que :

$$c.q \leq v.cap \wedge c_1.e + c_1.s + cout(c_1.n, c.n) \leq c.l$$

$$\wedge Max(c.e, c_1.e + c_1.s + cout(c_1.n, c.n)) + c.s + cout(c.n, c_2.n) \leq c_2.l$$

$Max(c.e, c_1.e + c_1.s + cout(c_1.n, c.n)$ reflète la contrainte du problème spécifiant que, si le véhicule peut être au niveau de c avant le début de sa fenêtre temporelle $c.e$, il doit attendre jusqu'à $c.e$; sinon, il ne peut pas quitter c avant $c_1.e + c_1.s + cout(c_1.n, c.n)$. Cette règle est illustrée par la figure 6.2.

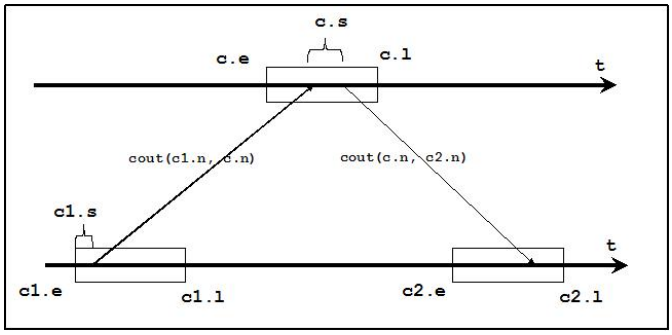

FIG. 6.2 – Insertion valide

L'objectif d'un AV est de ne percevoir que les clients ne violant pas ces deux contraintes. Pour ce faire, il exécute un $look(dc_p, \emptyset, exp)$ afin de percevoir les objets représentant des clients insérables dans son plan. L'expression exp retranscrit les contraintes que nous venons d'énoncer.

$$exp = (c.veh = unknown_{veh} \wedge c.q \leq capc_1.succ = c_2.id \wedge c_1.veh = id$$

$$\wedge c_1.e + c_1.s + cout(c_1.n, c.n) \leq c.l$$

$$\wedge Max(c.e, c_1.e + c_1.s + cout(c_1.n, c.n)) + c.s + cout(c.n, c_2.n) \leq c_2.l)$$

Le contexte de l'expression (les objets de l'environnement validant l'expression) est constitué du client à insérer (c) et des deux clients (c_1 et c_2) entre lesquels c risque d'être inséré. Le client c concerné ne doit évidemment appartenir au plan d'aucun véhicule, ce qui est traduit par la condition $c.veh = unknown_{veh}$. Les conditions $c_1.succ = c_2.id$ et $c1.veh = id$ identifient deux AC adjacents dans le plan de l'AV. Le reste des conditions traduisent la faisabilité de l'insertion décrite plus haut.

La description de contexte dc_p associée à cette expression dans $look(dc_p, \emptyset, exp)$ est la suivante :

$$dc_p = \{client \leftarrow\ c, pred \leftarrow\ c_1, succ \leftarrow\ c_2\}$$

Un AV v a deux clients « virtuels » - ne correspondant à aucun vrai client - dans son plan : deux fois son dépôt. Ainsi, deux objets client représentant le dépôt et ayant $v.id$ comme valeur associée à la propriété veh existent dans l'environnement. En l'absence de ces deux clients, l'expression définie plus haut ne peut pas aboutir, puisqu'aucun objet ne peut être unifié avec c_1 ou c_2 en présence d'un plan vide. Pour qu'un client puisse être inséré à la première (resp. dernière) position dans le plan de l'AV, c_1 (resp. c_2) doit être unifiée avec un client situé au dépôt et qui a pour propriété veh l'identifiant du véhicule déposant l'expression.

Comme on le voit, les propriétés des objets représentant des AC dans l'environnement sont utilisées par les AV afin de limiter leur perception aux clients qu'ils peuvent effectivement insérer dans leurs plans. Une fois inséré dans le plan d'un AV, les propriétés e et l représentant la

fenêtre temporelle initiale d'un AC ne reflètent plus les vraies bornes inférieures et supérieures à l'intérieur desquelles l'AV peut visiter le client. En effet, considérons un client c qui vient d'être inséré entre deux clients c_1 et c_2. Ses « vraies » fenêtres temporelles sont illustrées dans la figure 6.3 (elles sont encadrées dans la figure). Si c ne met pas à jour ses fenêtres temporelles ainsi que celles de l'objet le représentant dans l'environnement, il peut induire en erreur son AV qui percevrait des clients qui ne sont pas faisables.

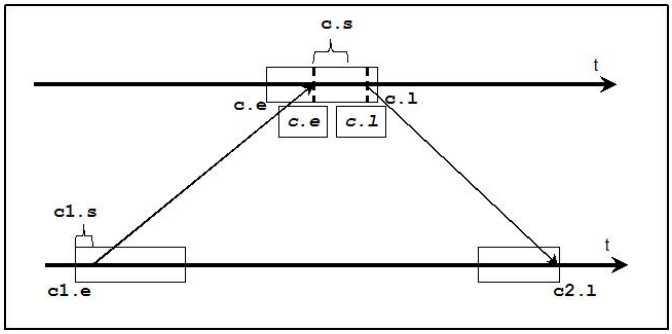

FIG. 6.3 – Nouvelle fenêtre temporelle

L'insertion du client implique également des mises à jour dans les fenêtres temporelles des autres clients précédemment insérés dans le plan de son véhicule. En effet, considérons un client c_1 avec $c_1.e$ et $c_1.l$ comme bornes inférieure et supérieure de sa fenêtre temporelle et $c_1.n$ le noeud du graphe concerné par sa demande. La borne inférieure de c_2, le client suivant c_1 dans le plan du véhicule ne peut être inférieure à $c_1.e + c_1.s + cout(n_1, n_2)$ car, si c'était le cas, cela violerait les contraintes temporelles de c_1 (voir figure 6.4).

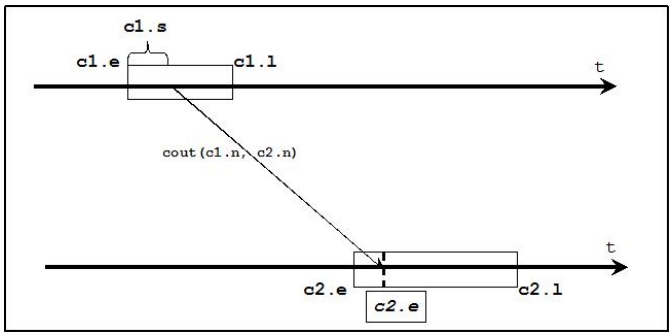

FIG. 6.4 – Mise à jour des fenêtres des successeurs

Pareillement, si c_1 est le client précédant c_2 dans le plan du véhicule, il ne peut avoir une borne supérieure de sa fenêtre temporelle qui soit supérieure à $c_2.l - cout(c_1, c_2) - s(c_1)$, car cela aurait pour conséquence de violer la contrainte temporelle relative à la borne supérieure de c_1 (voir figure 6.5).

Ainsi, un AC qui voit sa propriété e changée doit propager cette modification vers son successeur immédiat, et un AC qui voit sa propriété l changée doit propager cette modification à son prédécesseur immédiat.

Durant le processus de mise à jour des fenêtres temporelles des AC suite à une nouvelle insertion d'un client dans la tournée d'un AV, i.e. tant que les propriétés publiés des AC ne reflètent pas l'état réel du plan de l'AV, ce dernier ne doit pas se porter candidat à un autre AC.

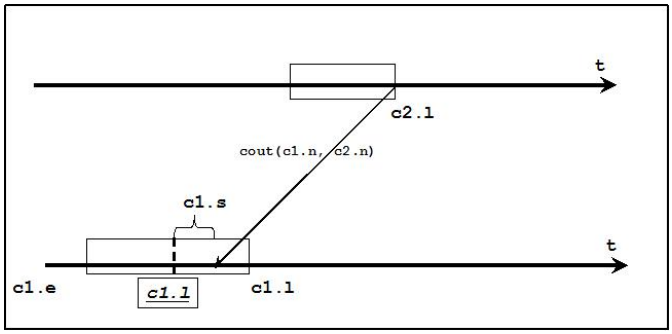

FIG. 6.5 – Mise à jour des fenêtres des prédécesseurs

L'objectif est d'empêcher la situation suivante. Soit l'AV v qui a été choisi par son AC courant pour l'insérer dans son plan. S'il commence immédiatement à percevoir de nouveaux clients, alors que le processus de mise à jour des propriétés des AC qui sont déjà dans son plan n'est pas encore terminé, l'AV peut percevoir un AC alors qu'il ne peut pas l'insérer dans son plan, car sa perception pourrait avoir été fondée sur des propriétés obsolètes, i.e. avant la mise à jour. Chaque AV maintien son plan courant *plan*, qui contient initialement les deux dépôts.

La notification de fin des mises à jour est envoyée à l'AV concerné par un AC dans son plan ayant reçu une demande de mise à jour, mais qui n'a pas eu à changer ses fenêtres temporelles. C'est le cas lorsqu'il reçoit une demande de mise à jour de la part de son prédécesseur, alors que sa propriété e est supérieure à la valeur demandée, et lorsqu'il reçoit une demande de son successeur alors que sa propriété l est inférieure à la valeur demandée. Au pire, les mises à jour sont propagées sur tous les AC du plan jusqu'aux dépôts. Dans ce cas, l'agent Dépôt reçoit la demande et envoie une notification à l'AV concerné.

6.3.4 Comportement des agents

Les figures 6.6, 6.7, 6.8 et 6.9 illustrent le comportement des quatre types d'agent que nous définissons.

Un AV perçoit un client grâce à l'instruction *look* définie plus haut et propose un prix d'insertion à l'AC perçu. S'il reçoit une décision positive de la part de l'AC, cela signifie qu'il peut l'insérer dans sa tournée. L'AV attend de recevoir une notification du fin du processus de mise à jour des fenêtres temporelles des clients dans son plan, et recommence à percevoir de nouveaux clients. L'insertion d'un client dans le plan d'un AV est effectuée avec un opérateur *insertion* défini ainsi :

$$insertion : \Omega^2 \rightarrow \Omega$$

$insertion(plan, c)$ insère le client c dans le plan *plan*. S'il reçoit une décision négative de la part de l'AC, il est immédiatement disponible pour la perception de nouveaux clients (figure 6.6).

Un AC reçoit un ensemble de prix d'insertion de la part des AV du SMA, en choisit un auquel il envoie une acceptation de son offre. S'il ne reçoit aucune offre, l'AC envoie une demande au dépôt lui demandant de mobiliser un nouvel AV pour le desservir. Il reçoit une demande du nouvel AV, et il le choisit. Ensuite, à chaque fois qu'il reçoit une demande de mise à jour, il la transfère à son successeur ou son prédécesseur - s'il a dû effectivement mettre à jour ses propriétés - ou il envoie une notification de fin de mise à jour à son AV (figure 6.7).

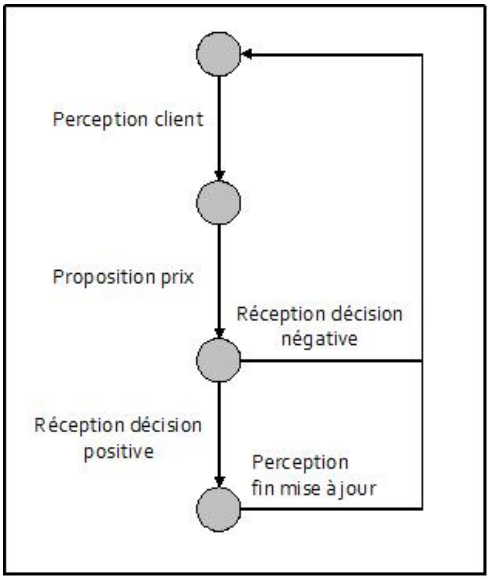

FIG. 6.6 – Comportement d'un agent Véhicule

Un agent Dépôt crée des AV lorsqu'il reçoit une demande de création de la part d'un AC, et est responsable des messages de mise à jour reçus de la part de ses successeurs ou prédécesseurs. Étant au bout de la chaîne de mises à jour, dès qu'il reçoit une demande de mise à jour, il notifie à l'AV correspondant que le processus de mise à jour est terminé. Lors de la réception d'une demande de création d'AV, il lance un nouvel AV dans le système (figure 6.8).

L'agent Interface a un comportement très simple : il attend qu'il client se connecte au système et lance un AC le représentant dans le SMA (figure 6.9).

6.3.5 Adaptation au TAD

Le système que nous venons de définir propose une solution pour le problème VRPTW dynamique. Dans le problème TAD, un client est décrit par les données suivante :

$\{id, n_1, q_1, e_1, l_1, s_1, n_2, q_2, e_2, l_2, s_2\}$, avec les variables indicées par 1 désignant le point de ramassage, et les variables indicées par 2 désignant le point de livraison, avec $q_2 = -q_1$. Les modifications qui doivent être apportées au programme pour être appliqué au problème TAD concernent deux points :

- Les deux demandes effectuées par un AC (le noeud de départ et le noeud d'arrivée) doivent être desservis par le même véhicule et le noeud de départ doit être visité avant le noeud d'arrivée.
- La capacité n'est plus relatives aux véhicules, mais aux clients.

Contraintes spatio-temporelles

Dans la nouvelle version du système, un AC c est désormais représenté par deux objets dans l'environnement : un objet représentant le point de ramassage (avec comme description initiale $d_{c_r} \equiv \{id \leftarrow id, n \leftarrow n_1, q \leftarrow q_1, e \leftarrow e_1, l \leftarrow l_1, s \leftarrow s_1, veh \leftarrow unknown_{veh}, pred \leftarrow unknown_{client}, succ \leftarrow unknown_{client}\}$) et un deuxième représentant le point de livraison (avec comme description initiale $d_{c_l} \equiv \{id \leftarrow id, n \leftarrow n_2, q \leftarrow q_2, e \leftarrow e_2, l \leftarrow l_2, s \leftarrow s_2,$

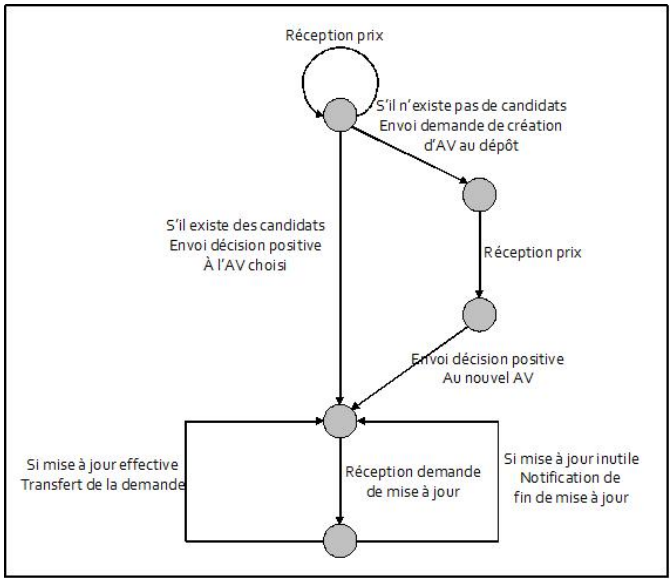

FIG. 6.7 – Comportement d'un agent Client

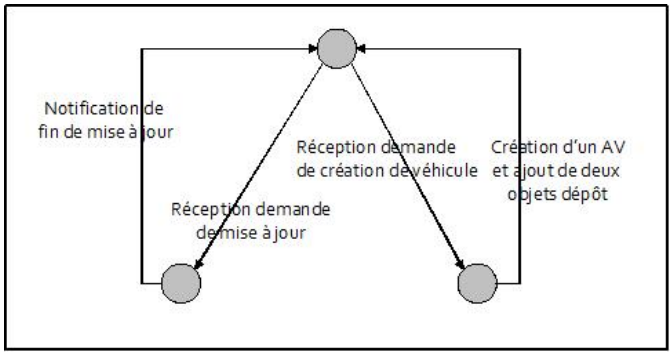

FIG. 6.8 – Comportement d'un agent Dépôt

$veh \leftarrow unknown_{veh}, pred \leftarrow unknown_{client}, succ \leftarrow unknown_{client}\}$). Le comportement des AC reste le même : ils reçoivent des offres des AV, et ils en choisissent le meilleur.

En revanche, les AV doivent modifier leur comportement. D'une part, un AV doit percevoir les deux objets représentant le client (représentant le noeud de ramassage et le noeud de livraison) afin de se porter candidat à son insertion. Mais cela n'est pas suffisant. En effet, un AV perçoit un objet représentant le client si ce dernier est insérable dans son plan. Or, quand un AV perçoit deux objets représentant le même client - même id - (un pour le point de ramassage et un pour le point de livraison), il sait qu'il peut desservir l'un ou l'autre, mais peut être pas les deux en même temps. Les figures 6.11, 6.12, 6.13, 6.14 illustrent ce cas. Soit le plan partiel initial de la figure 6.11 : un véhicule peut quitter le premier noeud du client c_1 ($c_1.n1$) et visiter son second noeud ($c_1.n2$). Soit un nouveau client c dont le premier noeud ($c.n1$) peut être inséré entre $c_1.n1$ et $c1.n2$. son insertion implique un décalage de la borne inférieure correspondant au noeud $c_1.n2$ (c.f figure 6.12). Le second noeud de c ($c.n2$) peut être inséré après $c_1.n2$ comme le montre la figure 6.13. En revanche, comme le montre la figure 6.14, l'insertion simultanée de $c.n1$ et de $c.n2$ n'est pas valide, puisque $c.n2$ ne peut plus être visité avant sa borne supérieure $c.l2$, en

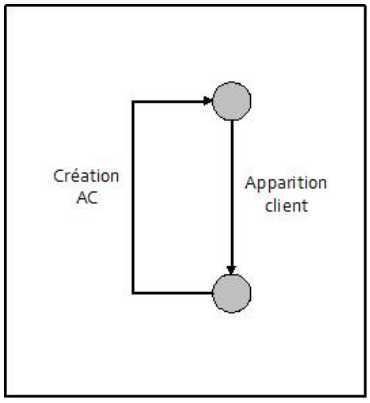

FIG. 6.9 – Comportement d'un agent Interface

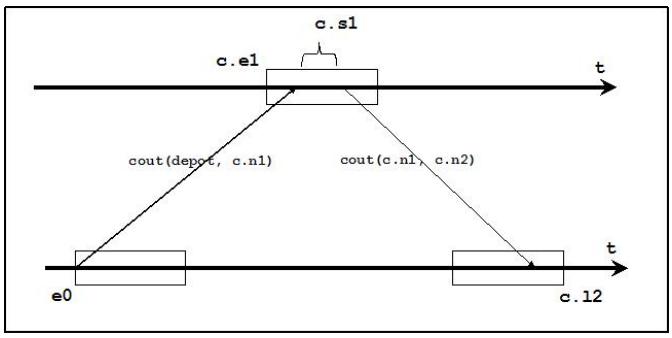

FIG. 6.10 – Client valide

raison du décalage causé par l'insertion de son point de ramassage $c.n1$.

Afin de pallier ce problème, un AV ayant perçu le noeud de départ et le noeud d'arrivée d'un client, et avant de lui proposer une offre, essaie d'insérer les deux noeuds et vérifie que l'insertion est valide i.e. ne viole aucune contrainte des clients déjà insérés dans son plan. Nous définissons l'opérateur

$$insertionValide : \Omega^3 \rightarrow \text{vrai}, \text{faux}, \text{nil}$$

$insertionValide(c_{r1}, c_{r_2}, plan)$ renvoie vrai si l'insertion des deux clients ne viole pas les contraintes temporelles des clients dans le plan de l'AV. Si tel n'est pas le cas, l'AV ne propose pas de prix d'insertion à l'AC correspondant et se met immédiatement à la recherche d'autres clients à insérer.

Contraintes de capacité

Dans un système traitant un problème VRPTW, la variation de la capacité d'un véhicule est monotone décroissante avec l'ajout de clients dans son plan, c'est la raison pour laquelle nous avons associé la propriété *cap* (capacité courante) à l'AV. En revanche, dans le problème TAD, la capacité d'un véhicule varie selon la portion de son plan qui est considérée. Plus précisément, la capacité du véhicule augmente après le passage par un point de ramassage et diminue après le passage par un point de livraison. Ainsi, la capacité du véhicule devient dépendante des clients dans son plan. Ainsi, nous définissons pour chaque objet client de l'environnement une propriété

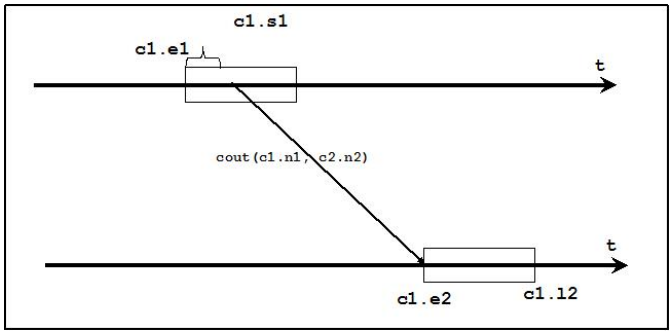

FIG. 6.11 – Plan partiel initial

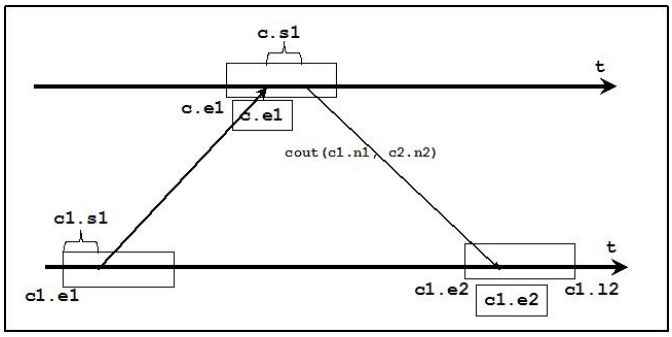

FIG. 6.12 – Conséquence de l'insertion de $c.n1$ seul (valide)

cap qui spécifie la capacité du véhicule au plan duquel il appartient, après que le véhicule l'ait visité. La condition de l'expression *exp* qui traite de la capacité devient :

$$c_1.cap \geq c.q$$

au lieu de $cap \geq c.q$

Encore une fois, il s'agit d'une condition nécessaire mais non suffisante pour la validité de l'insertion d'un client. En effet, que le nouveau client ne viole pas les contraintes de capacité d'un véhicule, et qu'il puisse donc être inséré après un noeud c_1 ne veut pas dire que son insertion est valide. La figure 6.15 illustre ce cas.

Le nouveau noeud de départ du client c ($c.n1$) est à insérer entre $c_1.n1$ et $c_2.n1$. Suivant la condition $c_1.cap \geq c.q$, cette insertion est valide. Néanmoins, lorsque cette nouvelle charge est propagée sur le reste du plan du véhicule, c_3 verra sa capacité devenir négative ($5-10$), l'insertion est donc non valide, malgré la perception du client par l'AV. Nous intégrons la vérification de la non-violation des contraintes de capacité lors de la vérification locale de la validité d'un client ($insertionValide$). Ainsi, la vérification de la validité d'une insertion est divisée en deux parties : la première est réalisée par l'environnement : la vérification des conditions nécessaires pour que l'insertion d'un client soit faisable (à travers l'expression du *look*), et une partie interne à l'AV qui vérifie le reste des conditions relatives au contraintes spatio-temporelles et de capacité.

La nouvelle expression exp_{tad} est la suivante :

$exp_{tad} = c_r.veh = unknown_{veh} \wedge c_l.veh = unknown_{veh} \wedge c_l.id = c_r.id \wedge c_r.q \leq c_{r1}.cap \wedge c_{r1}.succ = c_{r2}.id \wedge c_{r1}.veh = id \wedge c_{l1}.veh = id \wedge c_{r1}.e + c_{r1}.s + cout(c_{r1}.n, c_r.n) \leq c.rl \wedge Max(c_r.e, c_{r1}.e + c_{r1}.s + cout(c_{r1}.n, c_r.n)) + c_r.s + cout(c_r.n, c_{r2}.n) \leq c_{r2}.l \wedge c_{l1}.e + c_{l1}.s +$

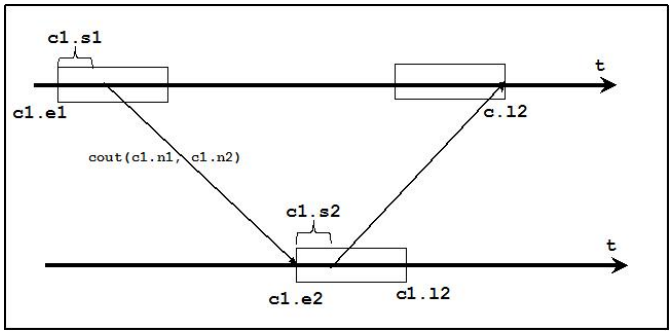

FIG. 6.13 – Conséquence de l'insertion de $c.n2$ seul (valide)

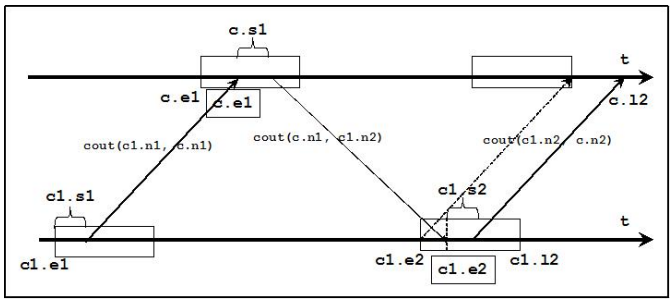

FIG. 6.14 – Conséquence de l'insertion de $c.n1$ et de $c.n2$ ensemble (non valide)

$$cout(c_{l1}.n, c_l.n) \leq c.ll \wedge Max(c_l.e, c_{l1}.e + c_{l1}.s + cout(c_{l1}.n, c_l.n)) + c_l.s + cout(c_l.n, c_{l2}.n) \leq c_{l2}.l$$

6.3.6 Suivi d'exécution

Nous considérons des temps de parcours statiques entre les noeuds du réseau, Ainsi, le seul objectif que nous avons pour le suivi d'exécution concerne la garantie de la correction de l'exécution des *look* des agents Véhicule à la recherche de clients à insérer. En effet, les positions des véhicules n'est pas importante en soi, puisqu'elle n'est utilisée par aucune expression de l'environnement. En revanche, nous devons empêcher qu'un client non insérable dans le plan d'un agent Véhicule ne soit perçu par ce dernier.

Il suffit pour cela de faire en sorte qu'un client visité par un véhicule n'ait plus d'objet le représentant dans l'environnement. L'agent Dépôt et les agents Client intègrent le suivi d'exécution dans leur comportement face au passage du temps. L'agent Dépôt qui voit arriver le moment nécessaire à un véhicule de bouger vers le premier noeud de son plan enlève l'objet le représentant de l'environnement. Aussi, chaque AC qui voit arriver le moment où le véhicule doit le quitter, enlève l'objet le décrivant de l'environnement. Cela signifie qu'une fois que le véhicule quitte un client c, il ne peut plus se porter candidat pour des clients situés entre c et son successeur immédiat. Concrètement, cela signifie qu'un véhicule engagé dans un chemin entre deux noeuds ne peut plus effectuer de détour pour chercher d'autres clients.

6.3.7 Calcul du prix d'insertion

Suivant la description présentée plus haut, un AC choisit entre plusieurs AV celui dont le prix d'insertion proposé est minimal. Les systèmes simulant des heuristiques d'insertion utilisent

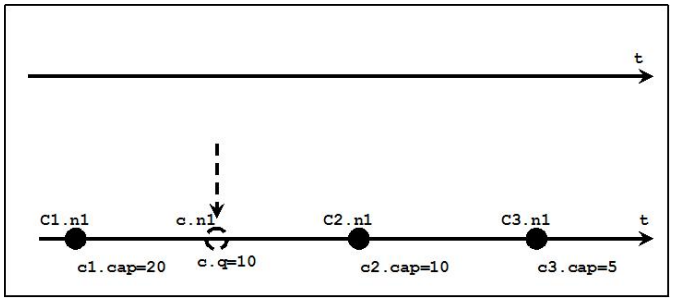

FIG. 6.15 – Insertion non valide de c (voir $c_3.cap$)

généralement la mesure utilisée par Solomon [Solomon, 1987] comme prix d'insertion. Rappelons que cette mesure consiste à prendre le client dont l'insertion entraîne une augmentation minimale du coût général du plan du véhicule. Si le prix calculé par un AV est l'accroissement de la distance parcourue par le véhicule, notre système aurait un comportement équivalent à celui des systèmes distribués de la littérature implémentant des heuristiques d'insertion. Cette mesure est simple et est la plus intuitive mais elle présente un inconvénient évident, c'est que l'insertion de ce client pourrait faire qu'un grand nombre de clients futurs deviennent irréalisables. Le problème avec cette mesure, c'est qu'elle génère des plans courants très contraints dans le temps et dans l'espace, i.e. des plans offrant peu de possibilités d'insertion entre chaque couple de clients adjacents dans le plan d'un véhicule. L'apparition de nouveaux clients risque fort de mobiliser un nouveau véhicule pour les servir. L'objet de cette section est de proposer une mesure originale de choix entre véhicules candidats pour l'insertion d'un même client.

Nous proposons une mesure dont l'objectif est de choisir l'AV dont « la diminution de probabilité de participation à des insertions futures est minimale ». Indépendamment des problèmes de tournées de véhicules, le principe général de la mesure est d'associer à chaque agent du SMA une quantité, que nous appelons « champ de perception » de l'agent. La mesure reflète les objets « perceptibles » par un agent, et ce indépendamment des objets présents effectivement dans l'environnement. Dans l'application présente, nous utilisons la variation du champ de perception de l'AV comme prix de l'insertion d'un client dans son plan.

Intuition de la mesure

Soit un agent Véhicule v qui n'a aucun client dans son plan (hormis le dépôt). Afin que cet agent perçoive un nouveau client c, il faudrait que la borne temporelle supérieure de c ($c.l$) soit assez grande pour permettre à v d'être au niveau du noeud ayant pour identifiant $c.n$ sans violer les contraintes temporelles de c. Plus précisément, il faut que le temps courant t, plus le temps de parcours entre le dépôt et $c.n$ soit inférieur ou égal à $c.l$. Partant de cette observation, nous définissons le champ de perception d'un AV comme le nombre de clients potentiels qui satisfont à cette contrainte. Pour ce faire, nous définissons « l'environnement physique » comme un ensemble de couples $\langle noeud, temps \rangle$, et le champ de perception d'un AV comme le nombre de couples qui restent valides étant donné son plan courant.

Lorsqu'un client est inséré dans le plan d'un AV, le champ de perception de ce dernier est recalculé, puisqu'un certain nombre de couples $\langle noeud, temps \rangle$ deviennent non valides suite à son insertion. La mesure associée à une offre déposée par un AV v à destination d'un AC c correspond à la diminution hypothétique du champ de perception de v suite à l'insertion de c dans son plan.

L'idée est que l'AV choisi pour l'insertion d'un client est celui qui perd le moins de chances d'être candidat pour l'insertion de clients futurs. Ainsi, le critère maximisé par l'ensemble des AV est la somme de leurs champs de perception, i.e. la capacité qu'à le SMA de réagir à l'apparition d'AC, sans mobiliser de nouveaux véhicules.

À ce jour, la version implémentée de la mesure de choix est relative à un problème euclidien, i.e. où les temps de parcours sont calculés suivant la métrique euclidienne. Les deux paragraphes suivants détaillent la mesure dans le cas euclidien et le troisième énonce les modifications à apporter pour pouvoir appliquer la mesure pour un réseau de transport quelconque.

Mesure dans le cas euclidien

Dans le cas euclidien, le réseau de transport est un plan, et le temps de parcours entre deux points i (décrit par (x_i, y_i)) et j (décrit par (x_j, y_j)) est égal à $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Par conséquent, si un véhicule se trouve au point i à l'instant t_i, il peut être au plus tôt au point j à l'instant $t_i + \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

Nous pouvons calculer à tout moment, à partir de la position d'un véhicule, l'ensemble des triplet (x, y, t) où il peut être dans le futur. En effet, en considérant un plan avec un axe des abscisses dans $[x_{min}, x_{max}]$ et un axe des ordonnées dans $[y_{min}, y_{max}]$, l'ensemble des positions spatio-temporelles est l'ensemble des points dans le cube délimité par $[x_{min}, x_{max}]$,$[y_{min}, y_{max}]$ et $[e_0, l_0]$. Soit un véhicule se trouvant au niveau du dépôt (x_0, y_0) à l'instant t_0. L'ensemble des points (x, y, t) accessibles par ce véhicule sont décrits par l'inéquation suivante :

$$\sqrt{(x - x_0)^2 + (y - y_0)^2} \leq (t - t_0)$$

Les (x, y, t) satisfaisant cette inéquation sont ceux qui sont positionnés à l'intérieur du cône $\mathcal{C}$ de sommet (x_0, y_0, t_0) et d'équation $\sqrt{(x - x_0)^2 + (y - y_0)^2} = (t - t_0)$ (c.f figure 6.16). Ce

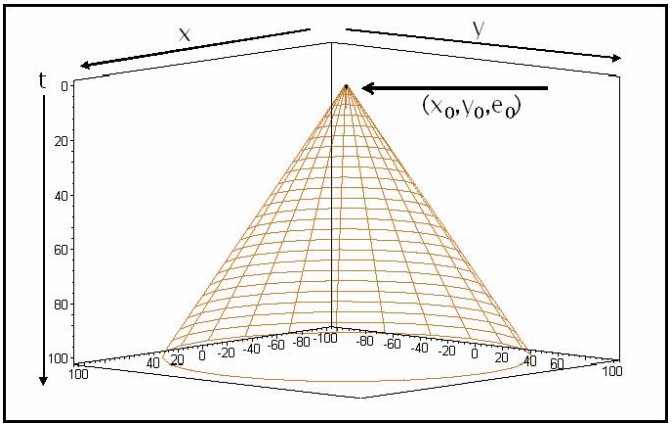

FIG. 6.16 – Champs de perception spatio-temporel initial

cône est le champ de perception d'un AV dans le cas euclidien. Il représente toutes les positions spatio-temporelles possibles que cet AV est susceptible d'avoir dans le futur.

Nous utilisons le champ de perception des AV lors du choix entre les prix d'insertion proposés par plusieurs AV à un AC. Il faut donc pouvoir comparer des champs de perception de différents AV. Pour ce faire, nous proposons de quantifier ce champ de perception, en calculant le volume du cône $\mathcal{C}$ représentant les positions futures possibles du véhicule :

$$Volume(\mathcal{C}) = \frac{1}{3} \times \pi \times (l_0 - e_0)^3$$

Il s'agit d'une quantification du champ de perception initial de tout nouvel AV créé. Lors de la perception d'un client, un AV calcule son nouveau champ de perception, le prix qu'il propose à l'AC est la différence entre son ancien champ de perception et son nouveau, que nous détaillons dans le paragraphe qui suit.

Dynamique du champ de perception

Considérons un client c_2 (de coordonnées (x_2, y_2) et avec une fenêtre temporelle $[e_2, l_2]$) qui rejoint le système, et supposons qu'il soit dans le champ de perception de v (l'expression exp est évaluée à vrai), avec v étant le seul AV du système et ayant un plan vide. L'agent v doit déduire son nouveau champ de perception i.e. les nouvelles zones espace-temps perceptibles, et qu'il peut donc atteindre sans violer les contraintes temporelles de c_2. Le nouveau champ de perception répond à la question suivante : « si v devait être au niveau de (x_2, y_2) à l_2, où est ce qu'il aurait pu être avant ? Et s'il devait y être à e_2 où est-ce-qu'il pourrait être après $e_2 + s_2$? ». Les triplets (x, y, t) où l'AV peut être avant la desserte de c_2 sont décrits par l'inéquation ($[a]$), et les triplets (x, y, t) où l'AV peut être après la desserte de c_2 sont décrits par l'inéquation ($[b]$).

$$\sqrt{(x - x_2)^2 + (y - y_2)^2} \leq (l_2 - (t + s)) \qquad [a]$$

$$\sqrt{(x - x_2)^2 + (y - y_2)^2} \leq (t - (e_2 + s_2)) \qquad [b]$$

Le nouveau champ de perception est illustré par la figure 6.17 : la nouvelle mesure consiste en l'intersection du cône initial $\mathcal{C}$ avec l'union des deux nouveaux cônes décrits par les inéquations $[a]$ et $[b]$ (dénotés respectivement par $\mathcal{C}_1$ et $\mathcal{C}_2$). La nouvelle mesure du champ de perception est égale au volume de l'intersection de $\mathcal{C}$ avec l'union de $\mathcal{C}_1$ et $\mathcal{C}_2$.

Afin de calculer le nouveau volume, nous calculons d'abord le volume de l'intersection de $\mathcal{C}$ avec $\mathcal{C}_2$, qui est égale au volume de $\mathcal{C}_2$, car il est totalement inclus dans $\mathcal{C}$. Le volume du cône $\mathcal{C}_2$ de sommet $(x_2, y_2, e_2 + s_2)$ est égal à $\frac{1}{3} \times \pi \times (l_0 - (e_2 + s_2))^3$. Ensuite, nous calculons l'intersection du cône initial avec le cône de sommet (x_2, y_2, l_2^*), qui donne deux cônes identiques de base elliptique. Le calcul du volume de l'intersection de ces deux cônes est reporté en annexe B. Il repose d'une part, sur le calcul de la surface de l'ellipse qui constitue la base des deux cônes, et d'autre part sur le calcul de la hauteur du cône, séparant son sommet de sa base. Les deux volumes sont additionnés avec le premier volume ($\mathcal{C}_2$). Cependant, l'intersection des deux petits cônes de sommets $(x_2, y_2, e_2 + s_2)$ et (x_2, y_2, l) a été comptabilisée deux fois (lors du calcul du volume de $\mathcal{C}_2$ et lors du calcul de l'intersection de $\mathcal{C}$ avec $\mathcal{C}_1$), elle doit donc être déduite.

Le prix de l'insertion d'un client dans le plan d'un véhicule est égal à la mesure associée à l'ancien champ de perception du véhicule moins la mesure du nouveau champ de perception, après l'insertion du client. La quantité ainsi mesurée représente les positions spatio-temporelles que le véhicule ne peut désormais plus avoir, s'il avait à insérer ce client dans son plan. L'AV retenu pour la desserte d'un client est celui dont l'insertion du client lui diminue le moins son champ de perception. Cela correspond à choisir le véhicule qui perd le moins de possibilités d'être candidat pour des clients futurs. Le calcul est réalisé avec l'opérateur

$$clacul_{prix} : \Omega^2 \rightarrow \mathbb{R}^+$$

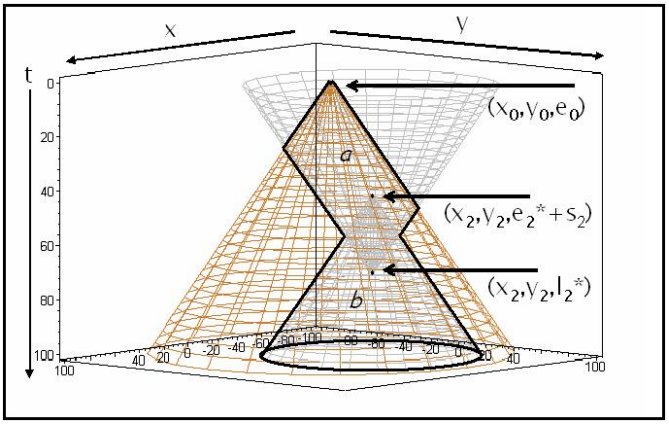

FIG. 6.17 – Champs de perception spatio-temporel après l'insertion de c_2

$calcul_{prix}(c, plan)$ renvoie le prix de l'insertion du client c dans le plan $plan$. Un client peut avoir différentes positions d'insertion possibles dans le plan d'un AV, le prix renvoyé par $calcul_{prix}$ correspond à la position ayant le prix le plus faible.

Mesure dans le cas général

L'environnement physique dans le cas non-euclidien n'est plus un cube espace-temps, mais un réseau espace-temps comme le montre la figure 6.18. Chaque AV associe à chaque noeud n du graphe un intervalle définissant les temps où il peut être à son niveau.

Ainsi, afin de généraliser la mesure du champ de perception au cas non-euclidien, lors de la perception d'un client, un AV doit parcourir les noeuds du graphe et mettre à jour les intervalles de temps qui seraient désormais associés avec chaque noeud, sans violer aucune des contraintes des clients dans son plan. L'offre d'un véhicule pour un client serait la différence entre la somme des longueurs de ces intervalles avant et après son insertion.

Présentée ainsi, cette méthode est d'une grande complexité, puisque la mise à jour des intervalles est effectuée pour chaque nouveau client, par chaque AV l'ayant perçu, et pour tous les noeuds du réseau. Nous travaillons à la définition d'un moyen plus efficace de généraliser la mesure du champ de perception à un réseau quelconque.

6.4 Mise en oeuvre avec Lacios

Dans cette section, nous entrons dans les détails de mise en oeuvre du SMA avec le langage Lacios.

6.4.1 Gestion du temps

La gestion du temps intervient lors du suivi d'exécution, puisque les AC doivent connaître le temps courant afin d'enlever les objets les représentant de l'environnement et les AV doivent connaître le moment de visiter un client, afin de l'enlever de leur plan. Elle intervient également durant le protocole d'insertion d'un client dans le plan d'un véhicule.

Lorsqu'un AC reçoit des offres d'AV, il doit en choisir celui qui propose le prix le plus bas, et déposer une décision favorable pour cet agent dans l'environnement. Pour qu'un AC sache le

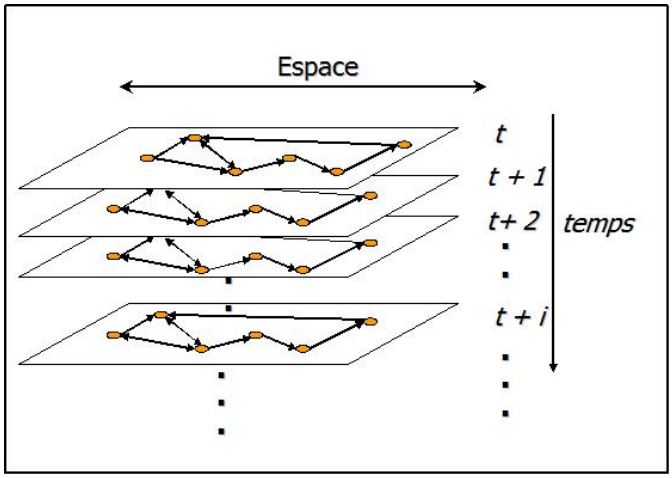

FIG. 6.18 – Réseau espace-temps pour le TAD

moment où il doit prendre sa décision, nous disposons de deux moyens. Le premier moyen est de faire savoir à l'AC combien d'AV l'ont perçu. Ainsi, l'AC connaîtrait le nombre d'offres qu'il va recevoir, et par conséquent le moment où il doit prendre sa décision. Si nous optons pour ce choix, la procédure de candidature des AV se ferait en deux temps. Dans le premier temps, les AV ayant perçu le client lui envoient des messages l'informant qu'il a été perçu. Dans un second temps, tous les AV envoient leurs offres à l'AC. Ainsi, l'AC peut savoir quand la procédure de candidature est terminée et quand il peut choisir un AV pour qu'il l'insère dans son plan. Cette procédure serait coûteuse en nombre de messages (deux fois le nombre d'offres) et n'est pas réalisable avec notre langage. En effet, comment notifier aux AV et à l'AC que la première phase est terminée ? Comment garantir qu'aucune perception n'est désormais possible pour un certain client ?

Le second moyen consiste en la définition d'un délai associé à chaque AC, à l'expiration duquel il doit prendre sa décision. Ainsi, avec un calibrage approprié de ce délai, un AC sait quand prendre sa décision sans avoir à savoir combien d'AV l'ont perçu. C'est ce second moyen que nous décidons d'utiliser.

Pour gérer le passage du temps dans Lacios, en l'absence de constructeurs temporels, nous proposons de définir un agent Temps dont l'objectif est de déposer des objets renseignant le temps courant. Cet agent dispose d'une variable libre t, et il ne met à jour l'objet renseignant le temps courant que lorsque ladite variable est instanciée (via $\nu\{t\}$). Nous externalisons ainsi la gestion du passage du temps à l'application externe, qui instancie la variable libre de l'agent Temps au bout d'intervalles de temps réguliers.

6.4.2 Système coordonné

Le système démarre avec l'agent Interface, l'agent Temps et l'agent Dépôt et aucun objet dans l'environnement.

$CS = \langle \Omega, d, \Omega_{ENV}, \mathcal{S} \rangle$,

- $\Omega = \mathcal{A} \uplus \mathcal{O}$
  - $\mathcal{A} = \{ai, at, ad\}$
    - $\Omega_{ai} = ai, \Omega_{ad} = ad, \Omega_{at} = at$, i.e. chaque agent dispose de sa propre description dans sa mémoire,

  - $proc(ai) = interface, proc(ad) = depot, proc(at) = temps$, les comportements (processus) des différents agents,
- $\mathcal{O} = \emptyset$,
- $d_{ai} \equiv \{id_{statique} \leftarrow 0, temps_{courant} \leftarrow 0\}, d_{ad} = \{ids \leftarrow 0, id \leftarrow$ "*depot*"$\}, d_{at} = \{id \leftarrow$ "*temps*"$\}$
- $\Omega_{ENV} = \emptyset$,
- $\mathcal{S} = \{id = that.id, that.émetteur = id\}$, i.e. les règles de l'environnement empêchent tout agent d'ajouter un objet le représentant avec un identifiant différent du sien, et tout agent d'envoyer un message dont il n'est pas l'*émetteur*.

Ci-après la définition du comportement des agents du SMA et la justification des propriétés définies pour chacun d'eux.

6.4.3 Comportement des agents

Agent Temps

Le comportement de l'agent Temps est défini ainsi :

$temps \stackrel{def}{=} \nu\{t\}(look(\emptyset, \{ancien_t \leftarrow x\}, x.catégorie =$ "*temps*"$).add(\{catégorie \leftarrow$ "*temps*"$, temps_{courant} \leftarrow t, propriétaire \leftarrow id\}, \text{vrai}, id = that.propriétaire)).temps$

L'ajout d'un objet renseignant la temps courant dans l'environnement est conditionné par l'affectation d'une valeur à la variable t. C'est le système externe interagissant avec le programme Lacios qui définit le pas de temps. À tout moment, il existe un objet o dans l'environnement renseignant le temps courant ($d_o \equiv \{catégorie \leftarrow$ "*temps*"$, temps_{courant} \leftarrow t, propriétaire \leftarrow$ "*temps*"$\}$), il peut être perçu par tous les agents ($e_p(o) =$ vrai) et ne peut être reçu que par l'agent l'ayant ajouté dans l'environnement, l'agent Temps. L'objet renseignant l'ancien temps désigné par $ancien_t$ n'est pas utilisé, la propriété $ancien_t$ figure dans l'expression simplement pour spécifier que l'objet unifié avec x est reçu, i.e. supprimé de l'environnement.

Agent Interface

Le code relatif au comportement de l'agent Interface est reporté dans le tableau 6.1. L'agent Interface interagit avec le monde extérieur et crée des AC représentant des clients. Comme pour le comportement *temps*, le comportement $interface$ utilise l'opérateur ν qui sert à instancier des variables avec des valeurs dans leur domaine par le système extérieur.

L'agent Interface génère des AC avec des identifiants uniques. Ceci est effectué en maintenant une propriété $id_{statique}$ (initialement à 0) incrémentée à chaque création d'un nouvel AC, dont la valeur courante sert à donner un identifiant unique à tout nouvel AC créé. Les identifiants des clients sont sous la forme "*c*" suivi d'un entier.

L'agent Interface reste également à l'écoute des messages émanant de l'agent Temps afin de mettre à jour sa propriété $temps_{courant}$ à chaque passage du temps (comportement $temps_{interface}$).

Agent Dépôt

L'agent Dépôt est responsable du lancement d'AV dans le système. Il génère un nouvel AV suite à la réception d'un message d'un AC. Il dispose des propriétés suivantes : $id_{statique}$,

/*L'agent Interface commence par instancier les paramètres du système : l'identifiant du noeud dépôt, la capacité maximale des AV et l'horizon d'ordonnancement du système */

$interface \stackrel{def}{=} \nu\{e_0, l_0, Q, n_{depot}, délai\}(creation_{client} \| temps_{interface})$

/*Dans le comportement $creation_{client}$, l'agent Interface met à jour sa propriété $id_{statique}$, attend l'arrivée d'un nouveau client, matérialisé par la liaison de ses variables libres, et lance un AC avec les propriétés correspondantes au nouveau client, qui se comporte suivant le processus *client*. La propriété e de l'AC à créer est mise à jour de telle manière qu'elle reflète la « vraie » borne inférieure du client : permettant à un véhicule partant immédiatement du dépôt de desservir le client.

$creation_{client} \stackrel{def}{=} update(\{id_{statique} \leftarrow id_{statique}+1\}).\nu\{n, q, e, l, s\}(spawn(\{id \leftarrow$ "c"$\&id_{statique}$, $n \leftarrow n, q \leftarrow q, e \leftarrow Max(temps_{courant} + cout(depot, n_1), e_1), l \leftarrow l, s \leftarrow s$, $veh \leftarrow unknown_{veh}, succ \leftarrow unknown_{client}, pred \leftarrow unknown_{client}$, $delai \leftarrow delai$, $temps_{courant} \leftarrow temps_{courant}\}, client)interface)$

/*Dans le comportement $temps_{interface}$, l'agent Interface reste continuellement à l'écoute d'un nouveau message renseignant le temps courant*/

$temps_{interface} \stackrel{def}{=} look(\emptyset, \{message \leftarrow x\}, x.temps \geq temps_{courant}).$
$update(\{temps_{courant} \leftarrow x.temps\}).temps_{interface}$

TAB. 6.1 – Comportement de l'agent Interface

délai et $temps_{courant}$ qui ont la même fonction que les propriétés correspondantes de l'agent Interface, n_{depot} qui renseigne l'identifiant du noeud où le dépôt est localisé et Q qui renseigne la capacité maximale des véhicules. Il a un comportement $temps_{depot}$ qui a la même fonction que $temps_{interface}$, i.e. la mise à jour de la propriété $temps_{courant}$ (c.f tableau 6.2).

Agent Client

Nous définissons les propriétés suivantes pour un AC :
- $delai_{statique}$ qui correspond au délai au delà duquel l'AC doit choisir un véhicule,
- *delai* qui renseigne le délai lui restant avant de prendre une décision,
- $temps_{courant}$, qui renseigne le temps courant connu pour l'AC,
- n, q, e, l, s comme défini précédemment,
- *veh* qui désigne l'identifiant du véhicule au plan duquel le client appartient,
- *succ* et *pred* désignant les identifiants des AC successeurs et prédécesseurs de l'AC dans le plan du véhicule *veh* (*veh*, *succ* et *pred* sont évalués à *unknown* si le client n'est associé à aucun plan),
- $cout_{pred}$ renseigne le temps de parcours entre le client prédécesseur et l'AC,
- $cout_{succ}$ renseigne le temps de parcours entre l'AC et le client successeur,
- s_{pred} désigne le temps de service de l'AC prédécesseur de l'AC,
- $vehicule_{courant}$ qui désigne l'identifiant de l'AV proposant la meilleure offre courante,
- $offre$ qui renseigne la meilleure offre courante lui ayant été proposée,
- $e_{courant}$ et $l_{courant}$ qui désignent les valeurs de la nouvelle fenêtre temporelle qu'on demande à l'AC d'avoir (demande émanant d'un AC ou résultat de son insertion dans le plan d'un AV).

/*L'agent Dépôt commence par attendre la liaison de ses variables libres par le système externe*/

$depot \stackrel{def}{=} \nu\{l_0, Q, n_{depot}, délai\}(creation_{vehicule} \| temps_{depot})$

/*La création des AV est le résultat de la réception d'un message de la catégorie "création" de la part d'un AC n'ayant reçu aucune offre d'insertion*/

$creation_{vehicule} \stackrel{def}{=} look(\emptyset, \{message \leftarrow x\}, x.catégorie = \text{“}création\text{”}).$
$update(\{id_{statique} \leftarrow id_{statique} + 1\}).$
$spawn(\{id \leftarrow \text{“}v\text{”}\&id_{statique}, cap \leftarrow Q, temps_{courant} \leftarrow temps_{courant}\}, vehicule).$

/*Deux objets dépôt sont également déposés dans l'environnement, représentant le dépôt. Les deux objets sont associés au nouvel AV lancé (veh = l'identifiant du nouvel AV), et sont chaînés, i.e. $succ$ du premier objet est égal à l'id du second*/

$add(\{catégorie \leftarrow \text{“}client\text{”}, propriétaire \leftarrow id, id \leftarrow \text{“}dep_2\text{”}, n \leftarrow n_{depot}, e \leftarrow temps_{courant},$
$l \leftarrow l_0, s \leftarrow 0, q \leftarrow 0, succ \leftarrow unknown_{client}, veh \leftarrow \text{“}v\&id_{statique}\text{”}\}, \text{vrai}, id = that.propriétaire)$
$.add(\{propriétaire \leftarrow id, catégorie \leftarrow \text{“}client\text{”}, id \leftarrow \text{“}dep_1\text{”}, n \leftarrow n_{depot}, e \leftarrow e_0, l \leftarrow l_0, s \leftarrow 0,$
$q \leftarrow 0, succ \leftarrow \text{“}dep_2\text{”}, veh \leftarrow \text{“}v\text{”}\&id_{statique}\}, \text{vrai}, id = that.propriétaire).creation_{vehicule}$

/*Le comportement $temps_{depot}$ met à jour le temps courant de l'agent Dépôt*/

$temps_{depot} \stackrel{def}{=} look(\emptyset, \{message \leftarrow x\}, x.temps \geq temps_{courant}).$
$update(\{temps_{courant} \leftarrow x.temps\}).\ temps_{depot}$

Tab. 6.2 – Comportement de l'agent Dépôt

Chaque AC doit connaître ses successeurs et ses prédécesseurs ainsi que le temps de parcours entre son noeud et les leur. Ceci est nécessaire lors de la propagation des mises à jour des fenêtres vers ses successeurs et ses prédécesseurs. Ainsi, l'offre d'un AV comprend : L'identifiant de l'AV proposant l'insertion (veh), le prix associé $prix$, le successeur du client s'il venait à être inséré dans le plan du véhicule ($succ$), son prédécesseur ($pred$), le temps de parcours entre le client et son successeur ($cout_{succ}$), avec son prédécesseur ($cout_{pred}$) ainsi que le temps de service de son prédécesseur (s_{pred}). Toutes ces données sont sauvegardées par l'agent, en vue d'être utilisées ultérieurement lors des mises à jour.

Le code Lacios relatif au comportement $offre$ est reporté en tableau 6.5

La définition du comportement $miseAjour$ est reportée au tableau 6.6.

Dans le cas d'un passage du temps, si l'AC n'est pas encore inséré dans le plan d'un AV associé ($veh = unknown$), il décrémente son délai. Si le nouveau délai est inférieur ou égal à zéro (l'intervalle de temps spécifié pour le client est expiré), l'agent dépose un objet renseignant une décision favorable à son meilleur véhicule courant. Il met également à jour ses propriétés e, l, $succ$, $pred$, s_{pred}, $cout_{pred}$, $cout_{succ}$ vers celles demandées par l'AV.

Un AC maintient une propriété $faisable$. Si elle est égale à faux, cela signifie que l'agent a déjà demandé à l'agent Dépôt de créer un nouvel AV. Si l'AC n'a reçu aucune proposition d'un AV ($vehicule_{courant} = unknown$), et que $faisable \neq$ faux ; l'AC dépose un message demandant au dépôt de générer un nouvel AV, et réinitialise son délai.

La définition du comportement $temps_{client}$ est reportée dans le tableau 6.7.

/*Un AC commence par sauvegarder la valeur de son délai de prise décision afin de pouvoir le réinitialiser ultérieurement, avant de s'annoncer en ajoutant un objet le représentant dans l'environnement. Un AC ne publie pas toutes ses propriétés, mais seulement celles qui sont pertinentes lors d'accès associatifs à l'environnement : sa catégorie (client), son identifiant *id* (pour les agents désirant lui envoyer des messages adressés), n, e, l s, q, *succ*, *pred* et *veh*. La description d'un AC ne peut être reçue que par lui même ($e_r = (id = that.propriétaire)$)*/

$client \stackrel{def}{=} update(delai_{statique} \leftarrow delai).\ add(\{propriétaire \leftarrow id,\ catégorie \leftarrow$ "*client*", $id \leftarrow id,\ n \leftarrow n,\ e \leftarrow e,\ l \leftarrow l,\ s \leftarrow s,\ q \leftarrow q,\ succ \leftarrow succ,\ pred \leftarrow pred,\ veh \leftarrow veh\}$, vrai, $id = that.propriétaire$).

/*Ensuite, il reste à l'écoute des messages qui lui sont adressés (comportement $dyadic_{client}$) et ceux relatifs au passage du temps (comportement $temps_{client}$). */

$(dyadic_{client} \| temps_{client})$

TAB. 6.3 – Comportement de l'agent Client

/*Un AC peut recevoir des messages adressés de mise à jour ou des offres d'insertion de la part d'AV. S'il s'agit d'une offre de la part d'un AV, l'AC sauvegarde le prix proposé ainsi que exécute le comportement $offre$. Si le message reçu par l'AC est une demande de mise à jour, il met à jour ses propriétés avant d'effectuer une tentative de mise à jour (comportement miseajour)*/

$dyadic_{client} \stackrel{def}{=} (dyadic_{offre} \| dyadic_{miseajour})$

$dyadic_{offre} \stackrel{def}{=} look(\emptyset, \{une\_offre \leftarrow x\}, x.dest = id \wedge x.categorie = \text{“}offre\text{”}).offre$

$dyadic_{miseajour} \stackrel{def}{=} look(\emptyset,\ \{une\_demande \leftarrow x\},\ x.dest = id \wedge x.categorie = \text{“}MaJ\text{”}).$ $update(\{succ_{courant} \leftarrow une\_demande.succ,\ pred_{courant} \leftarrow une\_demande.pred,$ $e_{courant} \leftarrow une\_demande.e,\ l_{courant} \leftarrow une\_demande.l\}).\ miseajour$

TAB. 6.4 – Comportement $dyadic_{client}$ de l'agent Client

Agent Véhicule

Nous définissons les propriétés suivantes pour un AV.
- La propriété *id* renseigne son identifiant,
- *cap* sa capacité actuelle,
- $client_{courant}$ renseigne l'identifiant de l'AC auquel l'AV a proposé une offre d'insertion,
- $temps_{courant}$ qui correspond au temps courant connu pour l'AV.

Un AV reste à l'écoute des objets de l'environnement représentant des AC (comportement $offre_{vehicule}$), aux messages qui lui sont adressés (comportement $dyadic_{vehicule}$), et au passage du temps (comportement $temps_{vehicule}$).

La description de contexte dc_p et l'expression *exp* concernent la faisabilité de l'insertion d'un client dans le plan d'un AV, et ont déjà été introduites dans la section précédente (section 6.3.2).

Ci-après la définition du comportement $dyadic_{vehicule}$ qui traite les message qui sont adressés à l'AV. Les messages adressés que peut recevoir un AV sont de la catégorie Décision, renseignant une décision d'un client par rapport à une offre proposé par l'AV ou de la catégorie Notification

$offre \stackrel{def}{=} (prix_{courant} = unknown_{prix} \vee prix_{courant} > une\_offre.prix) \lfloor update(\{vehicule_{courant} \leftarrow une\_offre.veh,\ prix_{courant} \leftarrow une\_offre.prix,\ succ \leftarrow une\_offre.succ,\ pred \leftarrow une\_offre.pred,\ cout_{succ} \leftarrow une\_offre.cout_{succ},\ cout_{pred} \leftarrow une\_offre.cout_{pred},\ s_{pred} \leftarrow une\_offre.s_{pred}\}).\ dyadic_{offre} \rfloor + (prix_{courant} \neq unknown_{prix} \wedge prix_{courant} <= une\_offre.prix) \lfloor add(\{emetteur \leftarrow id,\ destinataire \leftarrow une\_offre.veh,\ decision \leftarrow false\},\ that.dest = id,\ that.dest = id).\ dyadic_{offre} \rfloor$

TAB. 6.5 – Comportement $offre$ de l'agent Client

/*Les demandes de mise à jour adressées à un CA sont le résultat de l'insertion du client dans le plan d'un VA ou d'une insertion d'un client dans le plan de l' AV auquel il appartient. Si l'AC doit mettre à jour sa propriété e, il doit la propager vers son successeur immédiat (dépôt de message de la catégorie "MaJ" vers son successeur immédiat et mettre à jour sa propre description dans l'environnement, afin qu'elle reflète sa vraie fenêtre temporelle ($look + add$)*/

$miseajour \stackrel{def}{=} (e_{courant} > e \vee l_{courant} < l) \lfloor (e_{courant} > e) \lfloor update(\{e, e_{courant}\}).look(\emptyset, \{anciendesc \leftarrow x\}, x.id = id).add(\{propriétaire \leftarrow id, catégorie \leftarrow \text{“}client\text{”}, id \leftarrow id, n \leftarrow n, e \leftarrow e, l \leftarrow l, s \leftarrow s, q \leftarrow q, succ \leftarrow succ, pred \leftarrow pred, veh \leftarrow veh\}, \text{vrai}, that.propriétaire = id).add(\{émetteur \leftarrow id,\ catégorie \leftarrow \text{“}MaJ\text{”}, dest \leftarrow succ, e \leftarrow e + s + cout_{succ}, l \leftarrow \infty\}, that.dest = id, that.dest = id) \rfloor +$

/*S'il doit changer sa propriété l, il doit propager cette modification à son prédécesseur immédiat et mettre à jour sa description dans l'environnement*/

$(l_{courant} < l) \lfloor update(\{l, l_{courant}\}).\ look(\emptyset,\ \{anciendesc \leftarrow x\}, \text{vrai},\ x.id = id).add(\{propriétaire \leftarrow id,\ catégorie \leftarrow \text{“}client\text{”},\ id \leftarrow id,\ n \leftarrow n,\ e \leftarrow e,\ l \leftarrow l,\ s \leftarrow s,\ q \leftarrow q,\ succ \leftarrow succ,\ pred \leftarrow pred,\ veh \leftarrow veh\},\ \text{vrai},\ that.proprietaire = id).\ add(\{émetteur \leftarrow id,\ catégorie \leftarrow \text{“}MaJ\text{”},\ dest \leftarrow pred,\ l \leftarrow l - s_{pred} - cout_{pred},\ e \leftarrow 0\},\ that.dest = id,\ that.dest = id) \rfloor +$

/*S'il ne doit changer ni sa propriété e, ni sa propriété l, il dépose une notification de fin de mise à jour pour son AV.*/

$(e_{courant} <= e \wedge l_{courant} >= l)\ \lfloor add(\{émetteur \leftarrow id,\ destinataire \leftarrow veh,\ categorie \leftarrow \text{“}notification\text{”}\}) \rfloor$

TAB. 6.6 – Comportement $miseajour$ de l'agent Client

$temps_{client} \stackrel{def}{=} look(\emptyset, \{ancien_{temps} \leftarrow x\}, x.temps \geq temps_{courant}).\ update(\{temps_{courant}, x.temps\}).(veh = unknown) \lfloor update(\{delai \leftarrow ancien_{temps} - temps\}).\ (delai \leq 0) \lfloor (vehicule_{courant} \neq unknown) \lfloor update(\{veh \leftarrow vehicule_{courant},\ e \leftarrow e_{courant}, l \leftarrow l_{courant}\}).add(\{émetteur \leftarrow id, destinataire \leftarrow vehicule_{courant},\ catégorie \leftarrow \text{“}décision\text{”}, decision \leftarrow \text{vrai}\}, that.dest = id, that.dest = id).miseAjour \rfloor + (vehicule_{courant} = unknown_{veh}) \lfloor (faisable \neq \text{faux}) \lfloor add(\{émetteur \leftarrow id,\ destinataire \leftarrow vehicule_{courant}, catégorie \leftarrow \text{“}création\text{”},\ n \leftarrow n, l \leftarrow l\} that.dest = id, that.dest = id).update(délai \leftarrow délai_{statique},\ faisable \leftarrow \text{faux}) \rfloor + (faisable = \text{faux}) \lfloor look(\emptyset, \{ancienne_{desc} \leftarrow x\}, x.catégorie = \text{“}client\text{”} \wedge x.id = id) \rfloor \rfloor \rfloor + (veh \neq unknown) \lfloor \mathbf{0} \rfloor$

TAB. 6.7 – Comportement $temps_{client}$ de l'agent Client

$vehicule \stackrel{def}{=} ((offre_{vehicule} \| dyadic_{vehicule}) \| temps_{vehicule})$

/*Le comportement $offre_{vehicule}$ consiste à percevoir des objets représentant
des clients qui peuvent être insérés dans le plan de l'AV et à leur envoyer des offres
$offre_{vehicule} \stackrel{def}{=} look(dc_p, \emptyset, exp).add(sds, \text{faux}, id = that.destinataire).offre_{vehicule}$

TAB. 6.8 – Comportement de l'agent Vehicule

$dyadic_{vehicule} \stackrel{def}{=} dyadic_{decision} \| dyadic_{notification}$

$dyadic_{decision} \stackrel{def}{=} look(\emptyset, \{message \leftarrow x\}, x.\text{catégorie} = \text{“}décision\text{”} \wedge x.dest = id).$
$(message.dec = \text{vrai}) \lfloor update(\{cap \leftarrow cap - client.q, plan \leftarrow insertion(client_{courant}, plan)\}).$
$update(\{client_{courant} \leftarrow unknown_{client}\}).dyadic_{decision} \rfloor + (message.dec = \text{faux}) \lfloor dyadic_{decision} \rfloor$

$dyadic_{notification} \stackrel{def}{=} look(\emptyset, \{notification_1 \leftarrow x, notification_2 \leftarrow y\}, x.\text{catégorie} = \text{“}notification\text{”} \wedge x.dest = id \wedge y.\text{catégorie} = \text{“}notification\text{”} \wedge y.dest = id \wedge y \neq x).update(\{libre \leftarrow \text{vrai}\}).dyadic_{notification}$

TAB. 6.9 – Comportements *dyadic* de l'agent Véhicule

qui informe l'AV qu'il peut être de nouveau intéressé par la perception de nouveaux clients. L'agent Véhicule attend de recevoir deux notifications, une de chaque client, avant de mettre son état à libre.

$temps_{vehicule} \stackrel{def}{=} look(\emptyset, \{ancien_{temps} \leftarrow x\}, x.temps \geq temps_{courant}).update(\{temps_{courant}, x.temps\}).temps_{vehicule}$

TAB. 6.10 – Comportement $temps_{vehicule}$ de l'agent Véhicule

6.4.4 Adaptation au TAD

Nous donnons dans cette sous-section les modifications nécessaires Ci-après le nouveau comportement de l'agent Interface :

$interface \stackrel{def}{=} creation_{client} \| temps_{interface}$

$creation_{client} \stackrel{def}{=} update(\{id_{statique} \leftarrow id_{statique} + 1\}).\nu\ \{n_1, q_1, e_1, l_1, s_1, n_2, q_2, e_2, l_2, s_2\}$
$spawn(\{id \leftarrow \text{“}c\text{”}\&id_{statique}, n_1 \leftarrow n_1, q_1 \leftarrow q_1, e_1 \leftarrow Max(temps_{courant} + cout(depot, n_1), e_1),$
$l_1 \leftarrow l_1, s_1 \leftarrow s_1, n_2 \leftarrow n_2, q_2 \leftarrow q_2, e_2 \leftarrow Max(temps_{courant} + cout(depot, n_1), e_1) +$
$s_1 + cout(n_1, n_2), l_2 \leftarrow l_2, s_2 \leftarrow s_2\ veh \leftarrow unknown_{vehicule}, succ_1 \leftarrow unknown_{client},$
$pred_1 \leftarrow unknown_{client}, succ_2 \leftarrow unknown_{client}, pred_2 \leftarrow unknown_{client}, delai \leftarrow delai,$
$temps_{courant} \leftarrow temps_{courant}\}, client).creation_{client}$

L'agent Client ajoute désormais deux objets le représentant et non un seul :

$client \stackrel{def}{=} update(delai_{statique} \leftarrow delai).\ add(\{\text{propriétaire} \leftarrow id, \text{catégorie} \leftarrow \text{“}client\text{”},$
$id \leftarrow id, n \leftarrow n_1, e \leftarrow e_1, l \leftarrow l_1, s \leftarrow s_1, q \leftarrow q_1, succ \leftarrow succ_1, pred \leftarrow pred_1,$
$veh \leftarrow veh\}, \text{vrai}, id = that.\text{propriétaire}).\ add(\{\text{propriétaire} \leftarrow id, \text{catégorie} \leftarrow \text{“}client\text{”},$

$id \leftarrow id$, $n \leftarrow n$, $e \leftarrow e$, $l \leftarrow l$, $s \leftarrow s$, $q \leftarrow q$, $succ \leftarrow succ$, $pred \leftarrow pred$, $veh \leftarrow veh\}$, vrai, $id = that.propriétaire) \ldots$

Outre l'utilisation de l'expression exp_{tad} et le test de validité d'une insertion réalisé avec l'opérateur $insertionValide$, le reste des comportements des agents reste inchangé.

Suivi d'exécution

Concernant le suivi d'exécution, l'agent Dépôt se contente de recevoir itérativement les descriptions des objets devant être enlevés de l'environnement. Il s'agit des objets représentant le dépôt, qui doivent être quittés immédiatement afin d'être visités au début de leur fenêtre temporelle. Nous définissons le comportement $execution_{depot}$ pour un agent Dépôt comme suit :

$$execution_{depot} \stackrel{def}{=} look(\{letemps \leftarrow t, lesuivant \leftarrow c\}, \{obj \leftarrow c\}, d.succ = c.id \wedge t.temps + cout(c.id, d.id) \geq c.e).execution_{depot}$$

Les clients quant à eux intègrent le comportement $execution_{client}$ en parallèle avec leurs autres comportements.

$$execution_{client} \stackrel{def}{=} look(\{letemps \leftarrow t\}, \{obj \leftarrow c\}, c.id = id \wedge t.temps \geq c.e)$$

6.5 Outil et expérimentation

6.5.1 Outil

L'implantation d'un système D-VRPTW et TAD avec Lacios est effectuée en deux étapes. La première étape consiste en l'écriture du code Lacios comme présenté dans la section précédente. La seconde concerne l'alimentation du système avec les données relatives aux clients et aux paramètres de l'application. Nous utilisons un simulateur de clients pour la liaison des variables libres de l'agent Interface.

La figure 6.19 montre l'évolution du système avec l'interface graphique que nous avons réalisée pour Lacios. À gauche figurent tous les objets actuellement dans l'environnement, et à droite les instructions actuellement effectuées par les agents Interface et Dépôt. Afin d'avoir un affichage propre au problème, composé de noeuds d'un réseau, d'arcs, de véhicules etc. il suffit de réaliser un interpréteur des propriétés des objets de l'environnement. En effet, les plans des AV sont reconstituables à partir des propriétés des objets client de l'environnement.

Les graphes relatifs aux définitions de processus peuvent être visualisés avant l'exécution de l'application afin de vérifier son adéquation avec les attentes de l'utilisateur (e.g. la figure 6.20 illustre le comportement d'un agent Véhicule).

6.5.2 Expérimentation

Marius M. Solomon [Solomon, 1987] a créé un ensemble de problèmes différents pour le problème de tournées de véhicules avec fenêtres temporelles. Il est admis que ces problèmes sont assez divers et assez nombreux afin de pouvoir comparer avec une confiance suffisante les différentes approches proposées. Une preuve que ces problèmes sont assez diversifiés et assez difficiles est qu'il n'existe aucune heuristique donnant les meilleurs résultats pour tous les problèmes en même temps.

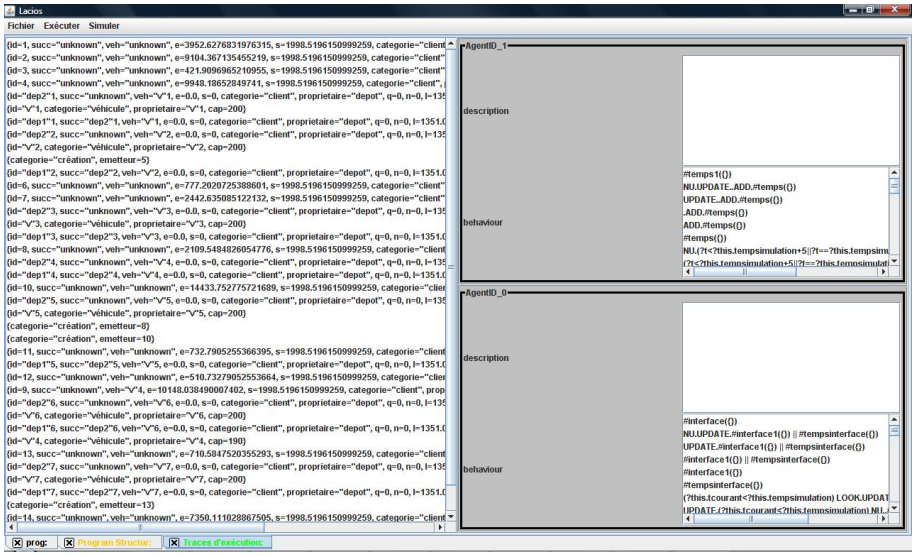

FIG. 6.19 – Visualisation des comportements des agents (ici agent Véhicule)

Dans les *benchmarks* de Solomon, six différents ensembles de problèmes sont définis : C1, C2, R1, R2, RC1 et RC2. Les clients sont uniformément distribués dans les problèmes du type R, regroupés dans les problèmes du type C, et un mélange de clients uniformément distribués et regroupés est utilisé dans les problèmes de type RC. Les problèmes du type 1 ont des fenêtres temporelles étroites, ainsi très peu de clients peuvent coexister dans le même plan, et les problèmes de type 2 ont des fenêtres temporelles larges. Enfin, un temps de service constant est associé à chaque client, il est de 10 dans les problèmes de types Ret RC, et de 90 dans les problèmes de type C.

Nous avons utilisé la classe R de problèmes, classe des clients uniformément distribués dans l'espace, puisque dans notre mesure, nous supposons implicitement une distribution uniforme des clients dans l'espace. Nous choisissons également la classe 1 de problèmes, les plus contraints, puisque c'est ces problèmes qui motivent l'utilisation de l'environnement afin de limiter les échanges d'informations inutiles entre agents.

Le premier indicateur auquel nous sommes intéressés est relatif au coûts réseau matérialisés par le nombre d'objets client perçus par les VA. Nous comparons notre approche avec une approche par diffusion, consistant en l'envoi de chaque nouvelle requête à tous les véhicules disponibles. La figure 6.21 montre que notre approche devient de plus en plus pertinente en considérant de plus en plus de clients, puisque le nombre de communications inutiles est plus grand quand nous considérons plus de clients.

Le second indicateur est relatif au processus de résolution. Quand la taille de flotte de véhicule est fixée à l'avance, le principal objectif est de minimiser le nombre de requêtes rejetées. Cependant, puisque nous créons un nouveau véhicule dynamiquement lorsqu'aucun véhicule ne peut desservir le nouveau client, notre système ne rejette aucune requête. Par conséquent, notre objectif devient la minimisation du nombre de véhicules mobilisés.

Nous avons codé un système dont le comportement est identique au notre. La seule différence est que le prix calculé par un VA est égal à l'accroissement de la distance parcourue, et non plus la diminution de son champ de perception. La table 6.11 reporte les résultats avec chaque fichier de la classe R1 où nous considérons successivement 25, 50, 100 et 200 clients. Les résultats montrent que l'utilisation de notre mesure mobilise moins de VA que la mesure d'accroissement

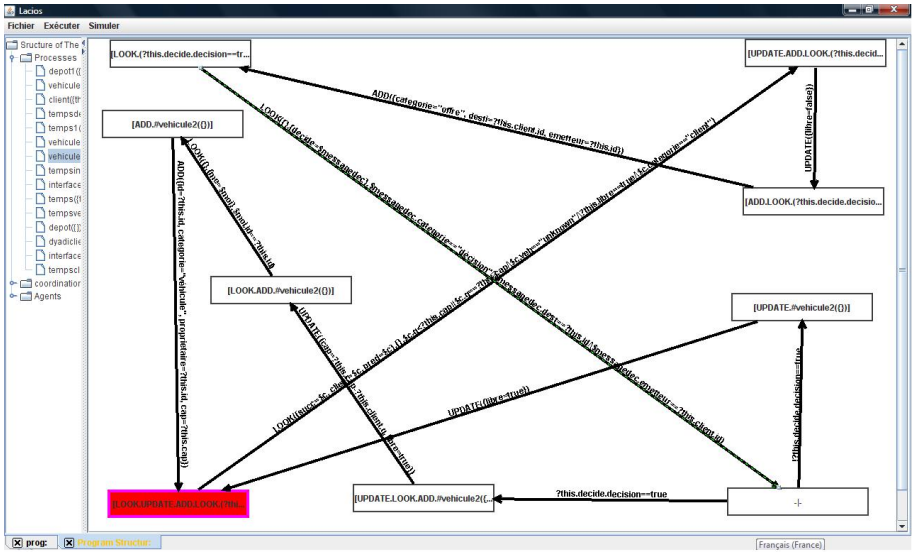

Fig. 6.20 – Evolution du système

des distances, et ce quelque soit le nombre de clients considérés. Ce résultat valide l'intuition première de la mesure qui consiste à maximiser les possibilités d'insertion futures des VA. En revanche, notre mesure se focalisant exclusivement sur la faisabilité des insertions, la distance totale parcourue par tous les VA est en moyenne supérieure à la distance générée par la mesure d'accroissement de distance. Nous pensons qu'un compromis entre les deux mesure, e.g. une somme pondérée de l'accroissement de la distance et de la diminution du champ de perception, est susceptible de donner de meilleurs résultats.

6.6 Conclusion

Comme nous l'avons précisé au début de ce chapitre, le choix d'adopter un protocole CNP pour la mise en place d'une heuristique d'insertion distribuée a été déjà suivi dans la littérature, notamment dans [Kohout and Erol, 1999; Fischer *et al.*, 1994; Dial, 1995]. La nouveauté en revanche, c'est l'utilisation d'un modèle de coordination orienté-données pour sa réalisation.

L'adoption d'un modèle de coordination orienté-données pour la réalisation de ce système est motivée principalement par le caractère dynamique et ouvert du problème. En effet, les AC rejoignent le système dynamiquement, et les AV sont créés au fur et à mesure de la résolution. Afin de limiter les interactions et la mise à jour des connaissances des agents les uns des autres, il nous semble pertinent de mettre ces connaissances en commun dans l'environnement SMA sous forme de descriptions, et que les AV découvrent leurs clients d'une manière associative, sans les connaître *a priori*.

Tous les modèles orientés-données ne sont pas adaptés à la mise en place d'un CNP pour les problèmes de tournées de véhicules. La structure de données en couples *propriétés-valeurs* du modèle Acios nous permet de le faire aisément, et surtout de limiter la perception des AV aux seuls agents qu'il peuvent effectivement insérer dans leurs plan. Comme nous l'avons vu lors du passage du problème VRPTW au problème TAD, l'intégration de nouvelles contraintes est assez aisée en utilisant notre modèle. Par ailleurs, nous pouvons également traiter des problèmes avec plusieurs dépôts en affectant simplement une valeur différente à la propriété *depot* des deux

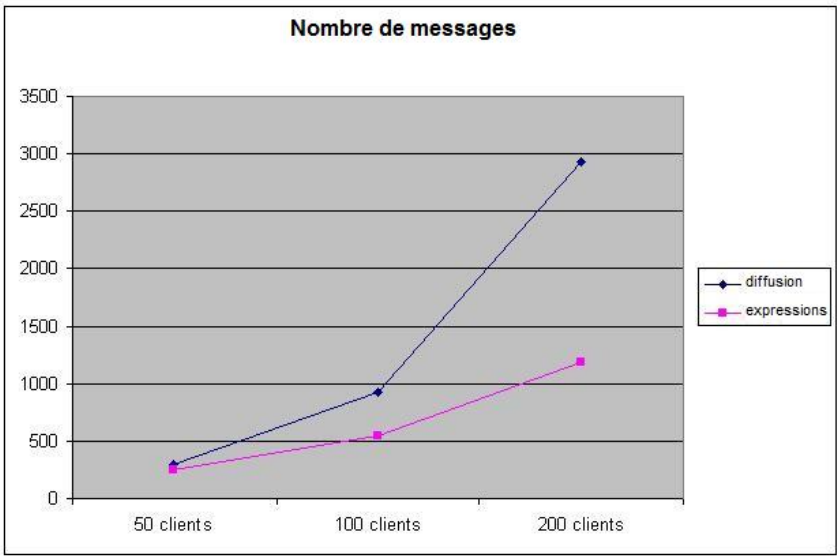

FIG. 6.21 – Nombre de messages : Expressions vs Broadcast

Δ Distance		
	Nombre de véhicules	Distance
25	3,4	316,6
50	6	671,2
100	12,1	1601,302095
200	21,6	6315,5
Δ Champ de perception		
	Nombre de véhicules	Distance
25	3,3	347,9
50	5,9	731,5
100	11,9	1774,4
200	21,4	6979,8

TAB. 6.11 – Résultats expérimentaux avec la classe R1 avec 25, 50, 100 et 200 clients

objets Client créés par l'agent Dépôt, et sans rien changer au code de l'application.

La mesure du champ de perception est originale, et pourrait être utilisée avec d'autres heuristiques de la littérature. Notre objectif dans le cadre de cette thèse a été d'en expérimenter l'usage avec une heuristique d'insertion simple. Il ressort de notre expérience que cette mesure mérite d'être plus étudiée et devrait être utilisée conjointement avec des mesures traditionnelles telles que l'accroissement de la distance parcourue, puisque son utilisation seule tend à donner des solutions avec des distances parcourues supérieures aux mesures traditionnelles.

Annexe A

Le langage Java-Lacios

Sommaire

A.1 Mode d'utilisation

Afin d'utiliser le langage Java-Lacios, il faut d'abord créer un un fichier texte avec un programme Lacios écrit dans la syntaxe définie plus bas. Ensuite, l'application externe utilisatrice du programme appelle la méthode statique *Compiler.parse(⟨ le fixhier⟩)* et récupère un objet Programme. Il suffit de lancer le programme en invoquant la méthode *exécuter()* de l'objet renvoyé par la méthode.

L'objet Programme dispose d'un vecteur d'agents en exécution, l'application utilisatrice peut appeler la méthode $instantiate(HashMapvarval)$ d'un agent pour lier ses variables libres, *varval* étant une table de hachage associant des valeurs aux variables libre de l'agent appelé. Afin de récupérer l'état du programme (les objets de l'environnement ou les agents), le programmeur peut à tout moment récupérer les valeurs des propriétés des objets ou des agents en invoquant les méthodes $getProperties()$ des agents ou des objets de l'environnement. La seconde solution consiste.

A.2 Syntaxe

Un programme Lacios est composé de deux parties : une partie définition des processus (identifiant ::= instructions) et une partie définition du système coordonné (CS ⟨ nom du programme ⟩{lois} ::= [une succession de *Spawn*]). Ci-après un sous-ensemble de la grammaire correspondante à Java-Lacios (issu du code JavaCC associé à Java-Lacios) :

```
/* LACIOS RESERVED TYPES AND WORDS*/
TOKEN ::≡ /* RESERVED WORDS AND LITERALS */
 < TRUE : ”true” > | < FALSE :”false” > | < NULL : ”null” >
| < THAT : ”that” > | < EMPTY : ”empty” >
| < LOOK : ”look” > | < ADD :”add” > | < UPDATE : ”update” > | < SPAWN : ”spawn”>
| < NU : ”NU”>  < COORSYS : ”CS”> |
```

```
/* LACIOS PROGRAM DEFINITION*/
ProgLacios ::≡ (ProccessDef)+ (CSDef)*
CSDef::≡ ”CS” <ID> ””(Expression ”;”)* ”” ” : :=” ”[”(”spawn” ”(” NewExp ”,” Process”)”)+ ”]”
ProccessDef::≡ <ID> ” : :=” Process
Process ::≡ ”empty” |action ”.” Process | ”(” Expression ”)” Process ” :” Process
| ”[” Process ”|” Process ”]” | <ID> [”(” <ID> ”/” Expression ( ”,” <ID> ”/” Expression)* ”)”]
action ::≡ ”look” ”(” NewExp ”,” NewExp ”,” Expression ”)”
|”add” ”(” NewExp [”,” ExpressionR [”,” Expression ]] ”)”
|”update” ”(” NewExp ”)” |”spawn” ”(” NewExp ”,” Process ”)”
| ”NU” ”(” VariableExp (”,” VariableExp)* ”)”
```

```
/* EXPRESSION SYNTAX */
VariableExp ::≡ ”?” <ID>
VariableExpContext ::≡ ”$” <ID>
ThatExp ::≡ ”that”
OPExp ::≡ ”” t=<ID> (”.” <ID> )* ”(” ArgumentList ”)”
NewExp ::≡ ”[” [”” <ID> (”.” t=<ID> n=n+”.”+t.image ;)* ”,”
Expression ”” (”,” ”” <ID> (”.” <ID> )* ”,” Expression”” )*] ”]”
Expression::≡ ConditionalExpression
ConditionalExpression ::≡ ConditionalOrExpression
ConditionalOrExpression ::≡ ConditionalAndExpression [ ”||” ConditionalAndExpression]
ConditionalAndExpression ::≡ EqualityExpression [ ”&&” e2=EqualityExpression ]
EqualityExpression ::≡ InstanceOfExpression(tree,treeC) [ ( ”==” | ”!=” )
InstanceOfExpression ]
InstanceOfExpression ::≡ RelationalExpression
```

```
/* EXPRESSION SYNTAX 2 */
RelationalExpression ::≡ ShiftExpression [ ( "<" | ">" | "<=" | ">=") ShiftExpression ]
ShiftExpression ::≡ AdditiveExpression [ ( "≪" | "≫" | "⋙" ) AdditiveExpression]
AdditiveExpression ::≡ MultiplicativeExpression [ ( "+" | "-" ) MultiplicativeExpression ]
MultiplicativeExpression ::≡ UnaryExpression [ ( "*" | "/" | "%" ) UnaryExpression ]
UnaryExpression ::≡ ( ( "+" | "-" ) UnaryExpression |
PreIncrementExpression | PreDecrementExpression | UnaryExpressionNotPlusMinus )
PreIncrementExpression ::≡ "++" PrimaryExpression
PreDecrementExpression ::≡ "–" PrimaryExpression
UnaryExpressionNotPlusMinus ::≡ ( "!" UnaryExpression | PostfixExpression)
PostfixExpression ::≡ PrimaryExpression [ "++" | "–" ]
PrimaryExpression ::≡ PrimaryPrefix ( PrimarySuffix)*
PrimaryPrefix ::≡ ( Literal | Name | ThatExp |"(" Expression ")" |OPExp
|VariableExp |VariableExpContext |NewExp )
PrimarySuffix ::≡ ( "[" Expression "]" | "." <ID> )
Literal ::≡ (<INTEGER_LITERAL> | <FLOATING_POINT_LITERAL>
| <CHARACTER_LITERAL> | t=<STRING_LITERAL> | BooleanLiteral | NullLiteral )
BooleanLiteral() ::≡ ( "true" | "false" )
void NullLiteral() ::≡ "null"
String StringLiteral() ::≡ <STRING_LITERAL>
Identifier() ::≡ <ID>
Name ::≡ <ID> ( "." t=<ID> )*
NameList ::≡ Name ( "," Name )*
ArgumentList ::≡ Expression ( "," Expression )*
```

Ci-après l'exemple des voyageurs écrit dans la syntaxe de Java-Lacios :

A.3 Interface graphique

Nous avons implémenté une interface graphique permettant d'expérimenter quelques exemples de code Lacios. Elle permet de créer ou d'ouvrir un fichier texte où le programme Lacios est écrit et de le compiler (figure A.1).

Lorsqu'un code de programme Lacios gagne en longueur, il peut devenir difficilement contrôlable. Lorsque le code devient difficilement lisible et donc difficilement vérifiable, il est toujours utile d'avoir une représentation des comportements définis par le programme. Cette possibilité est offerte au programmeur avec l'outil que nous avons réalisé. Une fois le programme compilé sans erreurs, et avant de lancer l'exécution, on peut visualiser le comportment des agents initialement définis, sous la forme de graphes. Les figures A.2, A.3, A.4 et A.5 illustrent les graphes des processus *train*, *voyageur*, *dyadic* et *groupe*. Les états initiaux de chaque processus sont en rouge et les états finaux en gris. En l'absence d'un état final, cela signifie que l'agent termine sur un appel de processus ou qu'il a un comportement infini. Les arcs sont étiquetées par la primitive à exécuter pour passer d'un état à un autre, et de la condition de passage de la transition dans le cas d'un branchement conditionnel.

$voyageur$ ::= $add([\{proprietaire, id\}, \{id, id\}, \{categorie, categorie\}, \{fumeur, fumeur\}, \{destination, destination\}, \{budget, budget\}], true, that.proprietaire == id).$ $[look([\{train, x\}], [], \$x.destination == destination).$ $(train.prixind > budget)$ $groupe : reservation|dyadic]$

$groupe$ ::= $look([\{candidat, \$t\}, \{compagnon1, \$x\}, \{compagnon2, \$y\}], [],$ $((\$t.destination == destination)\&\&(\$t.prixgroupe <= budget))\&\&$ $((((x.destination == destination)\&\&(\$x.train == "unknown")) \&\&((\$x.budget >= \$t.prixgroupe)\&\&$ $(\$y.destination == \$destination)))\&\&(((\$y.train == unknown) \&\&(\$y.budget >= \$t.prixgroupe))\&\&(\$x! = \$y)))).$ $[add([\{emetteur, id\}, \{destinataire, compagnon1.id\},$ $\{sujet,$"demande groupee"$\}, \{train, candidat.id\}, \{prix, candidat.prixgroupe\}], id ==$ $that.destinataire, id == that.destinataire).$ $empty|add([\{emetteur, id\},$ $\{destinataire, compagnon1.id\}, \{sujet,$"demande groupée"$\},$ $\{train, candidat.id\},$ $\{prix, candidat.prixgroupe\}], id == that.destinataire, id == that.destinataire).$ $empty]$

$reservation$ ::= $add([\{proprietaire, id\}, \{emetteur, id\}, \{destinataire, train.id\},$ $\{sujet, "reservation"\}], id == that.destinataire, id == that.proprietaire).$ $empty$

$dyadic$ ::= $look([], [\{message, \$x\}], that.destinataire == id).$ $(message.sujet ==$"demande groupée"$)(train == "unknown")$ $add([\{proprietaire, id\}, \{emetteur, id\},$ $\{destinataire, message.emetteur\}, \{sujet, "decision"\}, \{decision, true\}], id ==$ $that.destinataire, id == that.proprietaire)$ $.update([\{train, message.train\}]).$ $dyadic :$ $add([\{emetteur, id\}, \{destinataire, train.id\}, \{sujet, "decision"\}, \{decision, false\}],$ $id == that.destinataire, that.proprietaire == id).dyadic : (message.sujet == "decision")$ $(message.decision == true)$ $add([\{emetteur, id\}, \{destinataire, train.id\}, \{sujet,$"demande groupee"$\},$ $\{voyageur1, id\}, \{voyageur2, compagnon.id\}, \{voyageur3, compagnon1.id\},$ $\{voyageur4, compagnon2.id\}], id == that.destinataire, id == that.proprietaire).$ $updrate([\{train, candidat\}])$ $.dyadic : dyadic : dyadic$

$train$ ::= $look([], [\{message, \$x\}], that.destinataire == id).$ $(message.sujet ==$"demande groupee"$)update([\{capacite, capacite - 3\}]).$ $train : update([\{capacite, capacite - 1\}]).train$

$CS\ gare\{that.dest == id\}$ ::= $[spawn([\{capacite, 100\}, \{id, 0\}], train)$ $spawn([\{id, 1\},$ $\{categorie, "voyageur"\}, \{fumeur, true\}, \{destination, "Paris"\}], voyageur)$ $spawn([\{id, 2\},$ $\{categorie, "voyageur"\}$ $\{fumeur, false\}, \{destination, "Paris"\}], voyageur)$ $spawn([\{id, 3\},$ $\{categorie, "voyageur"\}, \{fumeur, true\}, \{destination, "Paris"\}], voyageur)]\|$

Tab. A.1 – Code source de l'exemple des agents voyageurs

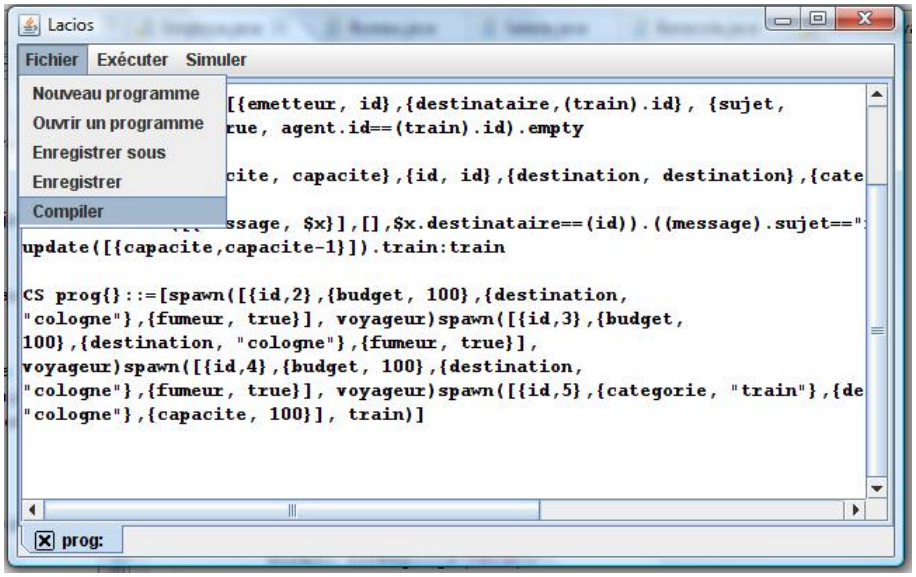

FIG. A.1 – Compiler

Deux choix sont offerts pour l'exécution d'un programme. Soit l'exécuter, soit en simuler une exécution. La différence entre l'exécution et la simulation, c'est qu'en simulation, on peut choisir un ordonnancement particulier dans l'exécution des actions, alors qu'en exécution, l'ordonnancement d'actions concurrentes est non déterministe. Les choix à effectuer surviennent précisément dans le choix d'actions parallèles, i.e. lors de l'exécution d'actions parallèles du même agent, ou lors de l'exécution d'actions de différents agents. La simulation consiste à cliquer sur les arcs des différents graphes de comportement afin de lancer les instructions associées.

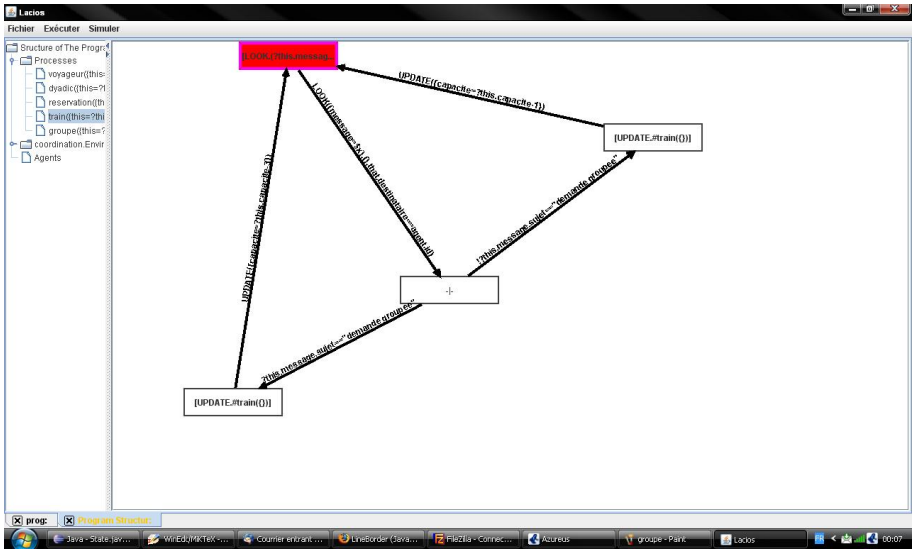

FIG. A.2 – Le graphe du processus *train*

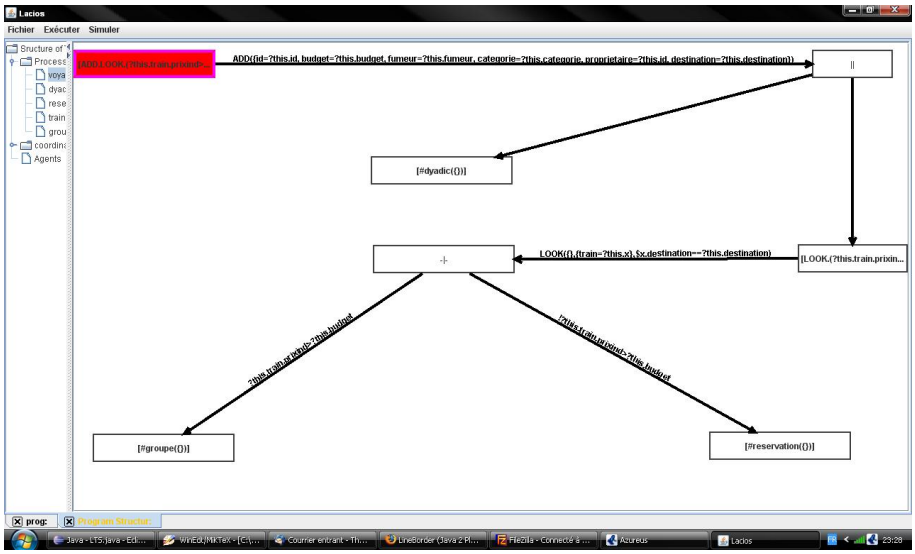

FIG. A.3 – Le graphe du processus *voyageur*

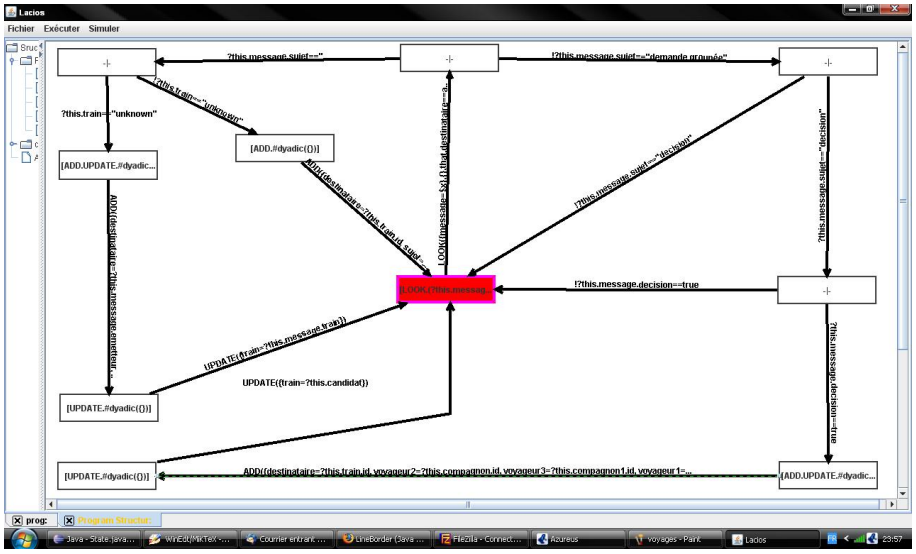

FIG. A.4 – Le graphe du processus *dyadic*

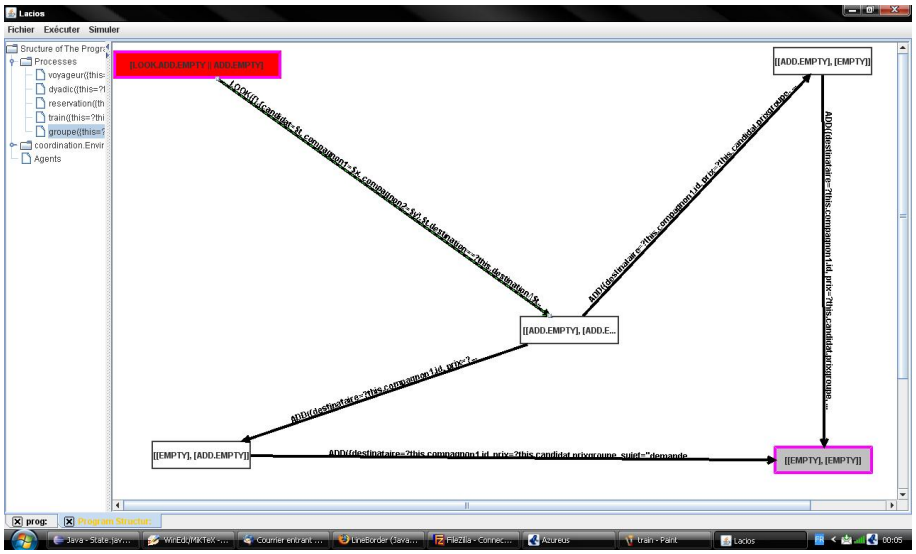

FIG. A.5 – Le graphe du processus *groupe*

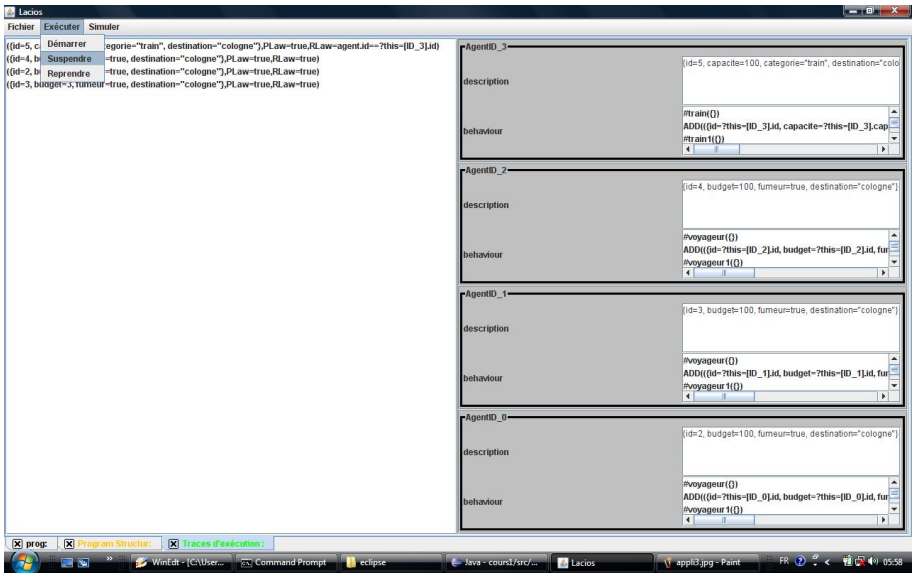

FIG. A.6 – Exécuter le programme

Annexe B

Calcul des champs de perception dans le cas euclidien

Sommaire

B.1 Illustration du problème

Le problème principal consiste à calculer le volume de l'intersection de deux cônes opposés, de sommets S1 (de coordonnées (x_1, y_1, t_1)) et S2 (de coordonnées (x_2, y_2, t_2)) (c.f. figure B.1). Les deux cônes sont coupés par un plan les divisant en deux cônes identiques de base elliptique dans le cas général.

L'équation (1) est celle du cône de sommet $S1$ et l'équation $S2$ est celle du cône de sommet $S2$.

$$\begin{cases} (x - x1)^2 + (y - y1)^2 = (t - t1)^2 & (1) \\ (x - x2)^2 + (y - y2)^2 = (t - t2)^2 & (2) \end{cases}$$

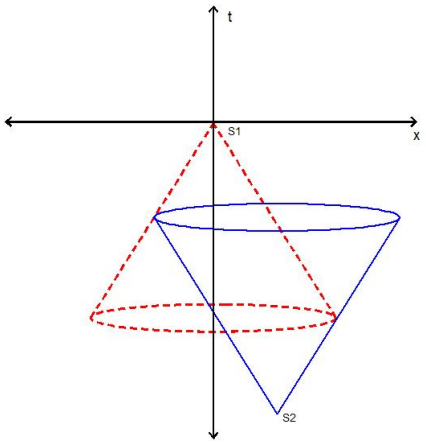

FIG. B.1 – Intersection de deux cônes opposés (1/2)

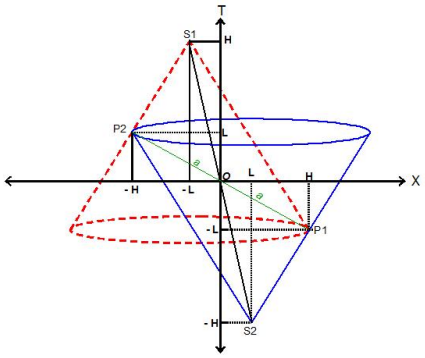

FIG. B.2 – Intersection de deux cônes opposés (2/2)

Nous désirons calculer l'intersection des cônes d'équation (1) et (2) dans le cas où :

$$(x1 - x2)^2 + (y1 - y2)^2 <= (t1 - t2)^2 \quad (3)$$

Cette restriction prend en compte le fait que, grâce aux conditions de perception du véhicule, seuls les clients à l'intérieur du cône peuvent être perçus. Si $(x1 - x2)^2 + (y1 - y2)^2 > (t1 - t2)^2$, alors le véhicule ne peut pas desservir le client. Graphiquement, cela correspond à ce que le sommet $S1$ doit être à l'intérieur du cône de sommet $S2$ et réciproquement, le sommet $S2$ doit être à l'intérieur du cône de sommet $S1$.

B.2 Calcul de l'équation du plan

Pour simplifier l'écriture, on se place dans un système d'axes (OX, OY, OT) tel que les deux cônes sont symétriques par rapport à l'origine O : l'origine (O) est prise au milieu du segment $S1S2$. La figure B.2 montre cette disposition. L'axe OY, perpendiculaire à ce plan, ne peut être représenté sans surcharger la figure.

Dans le nouveau système d'axes, les équations (1) et (2) deviennent :

$$\begin{cases} (X - H)^2 + Y^2 = (T - L)^2 \quad (1') \\ (X + H)^2 + Y^2 = (T + L)^2 \quad (2') \end{cases}$$

avec $L = \frac{1}{2}\sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2}$ et $H = \frac{1}{2}|t_2 - t_1|$

En résolvant $(1')$ et $(2')$, cela donne :

$(X + H)^2 - (X - H)^2 = (T + L)^2 - (T - L)^2$

$\Leftrightarrow XH = TL$

$\Leftrightarrow T = \frac{H}{L}X$ qui est l'équation du plan où se trouve l'intersection des deux cônes.

B.3 Calcul du volume

Afin de connaître la forme de l'intersection des deux cônes, il suffit de remplacer T par son équation dans l'une ou l'autre des deux équation ($(1')$ ou $(2')$).

Par exemple : $(X - H)^2 + Y^2 = (T - L)^2 = (\frac{H}{L}X - L)^2$

$\Leftrightarrow X^2 - 2XH + H^2 + Y^2 = (\frac{H}{L}X)^2 - 2(\frac{H}{L}X)L + L^2$

$\Leftrightarrow X^2 - 2XH + H^2 + Y^2 = (\frac{H}{L}X)^2 - 2(\frac{H}{L}X)L + L^2$

$\Leftrightarrow (1 - \frac{H}{L})X^2 + Y^2 = L^2 - H^2 \quad (3)$

(3) est l'équation de l'ellipse, avec a et b ses demi-axes :

$$\begin{cases} a = \sqrt{H^2 + L^2} \\ b = \sqrt{H^2 - L^2} \end{cases}$$

L'aire de l'ellipse vaut $A = \pi ab = \pi\sqrt{H^4 - L^4}$

On calcule la hauteur h du cône de sommet $S1$ et ayant l'ellipse pour base est égale à celle d'un triangle de base $2a$, ce qui donne :

$h = \frac{H^2-L^2}{\sqrt{H^2+L^2}}$. Le volume d'un cône est égal à $v = \frac{1}{3}ha$, avec h sa hauteur et A l'aire de sa base. Le volume commun aux deux cônes est son double, i.e. :

$$V = \frac{2}{3}\frac{H^2 - L^2}{\sqrt{H^2 + L^2}}\pi\sqrt{H^4 + L^4} = \frac{2\pi}{3}(H^2 - L^2)^{(\frac{3}{2})}$$

Conclusion générale et perspectives

Bilan

Ce travail de thèse est principalement motivé par la proposition d'un modèle de coordination orienté-données pour les Systèmes Multi-Agents ouverts. Ce travail propose un langage prenant en compte les aspects particuliers de la réalisation d'un système où les acteurs sont des agents et non des processus séquentiels, qui peuvent entrer et sortir librement du système et qui interagissent avec des systèmes externes. Les principaux points forts de notre proposition sont :

La modélisation de l'état des agents et de leur mémoire locale

À la différence des modèles orientés-données de la littérature, les agents dans Acios ont une mémoire locale et sont décrits par des propriétés qu'ils peuvent publier, mettre à jour et utiliser pour conditionner leur interaction avec les autres agents.

La richesse de la structure de données et l'expressivité de l'appariement

L'utilisation d'une structure de données utilisant des couples *propriété-valeur* typées permet une représentation de données proche de l'implémentation et augmente l'expressivité de l'appariement grâce à l'utilisation d'expressions utilisant les propriétés des agents, des variables et des opérateurs.

L'interaction contextuelle

Un agent peut percevoir plusieurs objets de l'environnement du SMA, en comparant leurs propriétés entre elles. L'interaction contextuelle couplée avec l'utilisation d'une structure de données utilisant des couples *propriété-valeur* permet la représentation de besoins interactionnels complexes tels que les contraintes d'un problème en programmation par contraintes.

Le partage total des données et leur sécurité

Dans le modèle Acios, les données sont théoriquement accessibles à tous les agents, et la sécurité est assurée d'une manière associative. Les règles de sécurité sont définies par le concepteur du système concernant tout ajout d'objet dans l'environnement, et par les agents du système pour toute données qu'ils ajoutent dans l'environnement.

L'ouverture du SMA

Le premier aspect de l'ouverture d'un SMA concerne la possibilité qu'ont les agents de rejoindre ou de quitter le système librement. L'ouverture dans ce sens est gérée naturellement par les agents, puisqu'ils n'ont pas à maintenir de connaissance sur les autres agents, mais interagissent exclusivement via l'environnement et découvrent leurs interlocuteurs dynamiquement. Le second aspect de l'ouvertur d'un SMA concerne la possibilité d'interaction avec un système externe. L'utilisation des variables libres dans le code des agents, et l'utilisation d'un opérateur de liaison des variables par le système externe permet d'isoler le comportement des agents du contexte applicatif du programme, et facilite sa réutilisation.

Le langage Lacios associé au modèle de coordination Acios définit une sémantique opérationnelle permettant de spécifier la dynamique du SMA en définissant la manière avec laquelle les expressions sont évaluées et la transformation de l'état du système en appliquent des règles bien définies. La définition d'un langage et de sa sémantique opérationnelle complète la présentation du modèle en spécifiant le comportement attendu d'un système adhérant au modèle.

L'implémentation du langage au dessus de Java est guidée par la sémantique opérationnelle que nous avons définie pour Lacios. Elle nous a permis de développer des aspects transparents au niveau de la définition du modèle et du langage tels que l'appariement, sa complexité et la distribution de l'environnement. La structure de données nous permet d'envisager différentes améliorations lors de l'implémentation du processus d'appariement, en utilisant des index sous forme de treillis de propriétés ou de treillis de Galois, susceptibles d'accélérer l'appariement lorsque le nombre de propriétés est stable.

Nous avons choisi une implantation sous forme d'un langage de script proche de la syntaxe formelle du langage, s'abstrayant ainsi des mécanismes de création de Threads, de leurs synchronisation et de la gestion des sémaphores sur l'environnement. Le résultat est une économie considérable en effort de programmation et de débuggage et un moyen de vérifier le comportement d'un système en appliquant les règles de la sémantique opérationnelle. Avoir une implémentation de référence du langage permet de s'abstraire des détails d'implémentation et de se focaliser sur la coordination au sens SMA, en raisonnant au niveau du modèle.

Nous avons proposé un système adhérant au modèle Acios pour la résolution des problèmes D-VRPTW et TAD. Le choix de ces problèmes comme illustration est motivée par le caractère dynamique et naturellement ouvert du problème. L'utilisation de l'environnement et des expressions Acios permet de limiter les interactions et la mise à jour des connaissances des agents les uns des autres. L'adaptation au TAD à partir d'un système réalisé pour le problème D-VRPTW est simple et nécessite des changements limités du code.

L'application nous a également inspiré une mesure du champ de perception qui pourrait être utilisée dans le cadre d'autres applications. Elle déplace l'objectif du SMA vers la maximisation des champs de perception des agents, ce qui permet dans le cadre du problème TAD de réduire le nombre de véhicules mobilisés pour desservir un ensemble de clients.

Travaux futurs

Distribution de l'environnement

L'un des avantages du modèle Acios est la proposition d'une structure de données utilisant des couples *propriétés-valeurs* pour la description de entités du SMA. Nous avons vu dans le chapitre 5 qu'une structure symbolique naturelle des objets de l'environnement était déductible moyennant l'introduction de catégories d'objets, sous forme d'un treillis de concepts. Cette structuration n'est pas possible pour les autres modes de représentation de données. Dans nos travaux futurs, nous utilisons cette structure comme un patron pour guider la distribution des objets sur différents environnements distribués. L'avantage principal est une duplication limitée des objets sur les différents hôtes d'un réseau et la circonscription des sites où les requêtes des agents sont soumises.

Introduction du temps

Nous avons défini le langage Lacios et nous lui avons associé une sémantique opérationnelle spécifiant le comportement de tout système adoptant sa syntaxe. Cependant, l'absence de

constructeurs temporels est un handicap pour la réalisation d'applications où les agents sont sensibles au passage du temps. Comme nous l'avons vu dans le chapitre 6, grâce à l'opérateur de liaison de variables ν, nous sommes arrivés à modéliser des comportements sensibles au passage du temps. Néanmoins, il s'agit d'une solution *ad hoc* qui ne saurait être satisfaisante dans le cas général. Dans nos travaux futurs, nous introduisons le temps sous forme de délais associés aux objets de l'environnement, au delà desquels ils disparaissent du système, et sous formes de paramètre de *look* : si au bout de t unités de temps $look_t$ n'arrive pas à trouver d'objets satisfaisants dans l'environnement, $look_t$ se transforme en un processus terminé ($\mathbf{0}$).

Extension de la mesure

La mesure du champ de perception semble prometteuse dans le contexte des heuristiques d'insertion. Comme nous l'avons dit dans le chapitre 6, nous travaillons à la définition d'un moyen efficace de généraliser la mesure du champ de perception à un réseau quelconque. Nous travaillons également à un moyen de l'utiliser conjointement avec des mesures classiques de la littérature.

La mesure du champ de perception que nous proposons permet au SMA de maximiser la somme des champs de perception spatio-temporels des agents Véhicule du système. Or, une mesure plus intéressante serait de maximiser l'union de ces champs de perception, et non de leur somme. Plus précisément, qu'un véhicule perde des zones espace-temps qu'il est le seul à couvrir devrait être plus coûteux que de perdre des zones que d'autres agents perçoivent. Cela suppose que les agents doivent publier leur champ de perception et que l'environnement procède à la pondération de leurs prix par l'importance des zones désormais non couvertes.

Bibliographie

[Adams, 2001] Julie A. Adams. Multiagent Systems : A modern approach to distributed artificial intelligence. *AI Magazine*, 22(2) :105–108, 2001.

[Aknine *et al.*, 2004a] Samir Aknine, Suzanne Pinson, and Melvin F. Shakun. An extended multi-agent negotiation protocol. *International Journal on Autonomous Agents and Multi-Agent Systems*, 8(1) :5–45, 2004.

[Aknine *et al.*, 2004b] Samir Aknine, Suzanne Pinson, and Melvin F. Shakun. A multi-agent coalition formation method based on preference models. *International Journal Group Decision and Negotiation*, 13(6) :513–538, 2004.

[Alami *et al.*, 1994] Rachid Alami, Frédéric Robert, Félix F. Ingrand, and Sho'ji Suzuki. A paradigm for plan-merging and its use for multi-robot cooperation. In *IEEE International Conference on Systems, Man, and Cybernetics 94*, 1994.

[Ambriola *et al.*, 1990] Vincenzo Ambriola, Paolo Ciancarini, and Marco Danelutto. Design and distributed implementation of the parallel logic language shared prolog. In *PPOP'90 : Proceedings of the second ACM Sigplan symposium on Principles & practice of parallel programming*, pages 40–49, New York, NY (USA), 1990. ACM Press.

[Arbab *et al.*, 1993] Farhad Arbab, Ivan Herman, and Per Spilling. An overview of Manifold and its implementation. *Concurrency : Practice and Experience*, 5(1) :23–70, 1993.

[Aronson, 1996] Leon D. Aronson. Algorithms for vehicle routing - a survey. Technical report, Faculty of Technical Mathematics and Informatics, Delft (The Netherlands), 1996.

[Attanasio *et al.*, 2004] Andrea Attanasio, Jean-François Cordeau, Gianpaolo Ghiani, and Gilbert Laporte. Parallel tabu search heuristics for the dynamic multi-vehicle dial-a-ride problem. *Parallel Computing*, 30(3) :377–387, 2004.

[Bachem *et al.*, 1996] Achin Bachem, Winfried Hochstättler, and Martin Malich. The simulated trading heuristic for solving vehicle routing problems. *Discrete Applied Mathematics*, 65(1-3) :47–72, 1996.

[Baeten, 2005] Jos C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3) :131–146, 2005.

[Balakrishnan, 1993] Nagraj Balakrishnan. Simple heuristics for the vehicle routing problem with soft time windows. *Journal of the Operational Research Society*, 44 :279–87, 1993.

[Balbo, 1999] Flavien Balbo. A model of environment, active support of the communication. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence (AAAI'99/IAAI'99) Workshop on Reasoning in context for AI Applications*, pages 1–5, Menlo Park, CA (USA), 1999. AAAI Press.

[Balbo, 2000a] Flavien Balbo. Environnement : un intermédiaire privilégié pour l'interaction, méthodologie, technologie et expérience. In *Proceedings des Journées Francophones d'Intelligence Artificielle Distribuée et Systèmes Multi-Agents (JFIADSMA'00)*, pages 403–416, 2000.

[Balbo, 2000b] Flavien Balbo. *ESAC : un modèle d'interaction multi-agent utilisant l'environnement comme support actif de communication. Application à la gestion des transports urbains.* PhD dissertation, Université Paris-Dauphine, Paris (France), 2000.

[Balbo, 2000c] Flavien Balbo. Modélisation de l'environnement comme support à l'interaction dans un système multi-agent. In *Proceedings de Rencontres des Jeunes Chercheurs en Intelligence Artificielle (RJCIA'00)*, pages 19–28, 2000.

[Balbo, 2001] Flavien Balbo. The environnement : a privileged intermediary for agent interaction. In *Proceeding of the 10th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'01), poster session*, pages 403–416, Annecy (France), 2001.

[Balbo, 2004] Flavien Balbo. A new interaction model for agent based simulation. In *Proceeding of the 18th European Simulation Multiconference (ESM'04)*, pages 372–378, Magdeburg (Germany), 2004.

[Beaufils *et al.*, 1996] Bruno Beaufils, Jean-Paul Delahaye, and Philippe Mathieu. Our meeting with gradual, a good strategy for the iterated prisoner's dilemma. In *Proceedings of Artificial Life*, volume V, pages 202–209, 1996.

[Bent and Hentenryck, 2003] Russell Bent and Pascal Van Hentenryck. Dynamic vehicle routing with stochastic requests. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1362–1363, Acapulco (Mexico), 2003. Morgan Kaufmann.

[Bergstra and Klop, 1984] Jan A Bergstra and Jan W. Klop. Process algebra for synchronous communication. *Information and Control, 60(1-3)*, pages 109–137, 1984.

[Bock and Diday, 2000] Hans-Herman Bock and Edwin Diday. *Analysis of Symbolic Data. Exploratory Methods for Extracting Statistical Information from Complex Data.* Springer-Verlag, Heidelberg, second edition, 2000. 425 pages.

[Bodin *et al.*, 1983] Lawrence Bodin, Bruce Golden, Arjang Assad, and Michael O. Ball. Routing and scheduling of vehicles and crews - the state of art. *Computers & Operations Research*, 10(2) :62–212, 1983.

[Bordat, 1986] Jean-Paul Bordat. Calcul pratique du treillis de galois dune correspondance. *Mathématique, Informatique et Sciences Humaines*, 24(96) :31–47, 1986.

[Boudali *et al.*, 2004] Imen Boudali, Wajdi Fki, and Khaled Ghedira. How to deal with the vrptw by using multi-agent coalitions. In *Proceedings of the Fourth International Conference on Hybrid Intelligent Systems (HIS'04)*, pages 416–421, Washington D.C, WA (USA), 2004. IEEE Computer Society.

[Boudali *et al.*, 2005] Imen Boudali, Wajdi Fki, and Khaled Ghedira. An interactive distributed approach for the vrp with time windows. *International Journal of Simulation*, 6(10) :48–59, 2005.

[Bowerman *et al.*, 1994] Robert L. Bowerman, Paul H. Calamai, and G. Brent Hall. The spacefilling curve with optimal partitioning heuristic for the vehicle routing problem. *European Journal of Operational Research*, 76 :128–142, 1994.

[Bräysy *et al.*, 2004] Olli Bräysy, Wout Dullaert, and Michel Gendreau. Evolutionary algorithms for the vehicle routing problem with time windows. *Journal of Heuristics*, 10(6) :587–611, 2004.

[Briot and Demazeau, 2001] Jean-Pierre Briot and Yves Demazeau. *Principes et architectures des systèmes multi-agents.* Hermès IC2 (information, commande, communication), Paris (France), 2001. 268 pages.

[Brogi and Ciancarini, 1991] Antonio Brogi and Paolo Ciancarini. The concurrent language, shared prolog. *ACM Transactions on Programming Languages and Systems*, 13(1) :99–123, 1991.

[Busi and Zavattaro, 1999] Nadia Busi and Gianluigi Zavattaro. On the expressiveness of event notification in data-driven coordination languages. Technical report, University of Bologna, Bologna (Italy), 1999.

[Busi and Zavattaro, 2003] Nadia Busi and Gianluigi Zavattaro. Expired data collection in shared dataspaces. *Theoretical Computer Science*, 298(3) :529–556, 2003.

[Busi *et al.*, 1998] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2) :167–199, 1998.

[Busi *et al.*, 2000a] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Comparing three semantics for linda-like languages. *Theoretical Computer Science*, 240(1) :49–90, 2000.

[Busi *et al.*, 2000b] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. On the expressiveness of linda coordination primitives. *Information and Computation*, 156(1–2) :90–121, 2000.

[Busi *et al.*, 2001] Nadio Busi, Paolo Ciancarini, Roberto Gorrieri, and Gianluigi Zavattaro. Models for coordinating agents : a guided tour. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination for Internet Agents : Models, Technologies, and Applications*, chapter 1, pages 6–24. Springer-Verlag, 2001.

[Bussmann and Müller, 1993] Stefan Bussmann and Jörg Müller. A negotiation framework for cooperating agents. In S. M. Deen, editor, *CKBS-SIG : Proceedings of the Special Interest Group on Cooperating Knowledge Based Systems*, pages 1–17. DAKE Centre, Keele (UK), 1993.

[Cabri *et al.*, 1999] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Reactive tuple spaces for mobile agent coordination. In *MA'98 : Proceedings of the Second International Workshop on Mobile Agents*, pages 237–248, London (UK), 1999. Springer-Verlag.

[Caillou *et al.*,] Philippe Caillou, Samir Aknine, and Suzanne Pinson. Méthode pareto-optimale de formation et de restructuration dynamique de coalitions d'agents. *Revue d'Intelligence Artificielle*, 17(4) :655–685.

[Carriero *et al.*, 1986] Nicholas Carriero, David Gelernter, and Jerrold Leichter. Distributed data structures in linda. In *POPL'86 : Proceedings of the 13th ACM Sigact-Sigplan symposium on Principles Of Programming Languages*, pages 236–242, New York, NY (USA), 1986. ACM Press.

[Carriero *et al.*, 1995] Nicholas Carriero, David Gelernter, and Lenore D. Zuck. Bauhaus linda. In *ECOOP '94 : Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, pages 66–76, London (UK), 1995. Springer-Verlag.

[Certu, 2002] Certu. Systèmes de transport à la demande : enquête sur les caractéristiques et les modes d'exploitation. Technical report, Ministère de l'équipement, des transports et du logement, Paris (France), 2002.

[Cervero, 1997] Robert Cervero. *Paratransit in America : Redefining Mass Transportation.* Praeger, Westport, CT (USA), 1997.

[Chein, 1969] Michel Chein. Algorithme de recherche des sous matrices premières dune matrice. *Bulletin Mathématique de la Sociologie Scientifique de la R.S de Roumanie*, 13(61) :21–25, 1969.

[Ciancarini *et al.*, 1995] Paolo Ciancarini, Keld K. Jensen, and Daniel Yankelevich. On the operational semantics of a coordination language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems : Proceedings of the ECOOP'94 Workshop on Modles and Languages for Coordination of Parallelism and Distribution*, pages 77–106. Springer-Verlag, Heidelberg, Berlin (Germany), 1995.

[Ciancarini *et al.*, 1998] Paolo Ciancarini, Robert Tolksdorf, Fabio Vitali, David Rossi, and Andreas Knoche. Coordinating multiagent applications on the www : A reference architecture. *IEEE Transactions on Software Engeneering*, 24(5) :362–375, 1998.

[Ciancarini, 1990] Paolo Ciancarini. Coordination languages for open systems design. In *Proceedings of the International Conference on Computer Languages (ICCL'90)*, pages 252–260, New Orleans, LA (USA), 1990. IEEE Computer Society.

[Clarke and Wright, 1964] G. Clarke and J.W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12 :569–581, 1964.

[Colorni and Righini, 2001] Alberto Colorni and Giovani Righini. Modeling and optimizing dynamic dial-a-ride problems. *International Transactions in Operational Research*, 8 :155–166, 2001.

[Cordeau and Laporte, 2003] Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem (DARP) : Variants, modeling issues and algorithms. *4OR : A Quarterly Journal of Operations Research*, 1(2) :89–101, 2003.

[Cordeau *et al.*, 2004] Jean-François Cordeau, Gilbert Laporte, Jean-Yves Potvin, and Martin W.P. Savelsberg. Transportation on demand. Technical report, Centre de Recherche sur les transports (CRT), Montréal (Canada), 2004.

[Cordeau, 2006] Jean-Franois Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54(3) :573–586, 2006.

[Corkill, 1979] Daniel D. Corkill. Hierarchical planning in a distributed environment. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI'79)*, Tokyo (Japan), 1979.

[Czech and Czarnas, 2002] Zbigniew J. Czech and Piotr Czarnas. A parallel simulated annealing for the vehicle routing problem with time windows. In *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 376–383, Canary Islands (Spain), 2002.

[Davis and Smith, 1983] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20 :63–109, 1983.

[De Nicola *et al.*, 1998] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Klaim : A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing)*, 24(5) :315–330, 1998.

[Decker and Lesser, 1992] Keith S. Decker and Victor R. Lesser. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, 1(2) :319–346, 1992.

[Desaulniers *et al.*, 2001] Guy Desaulniers, Jacques Desrosiers, Andreas Erdmann, Marius M. Solomon, and François Soumis. Vrp with pickup and delivery. *The vehicle routing problem*, pages 225–242, 2001.

[Desaulniers *et al.*, 2002] Guy Desaulniers, Jacques Desrosiers, Marius M. Solomon, François Soumis, and Jean-François Cordeau. The VRP with time windows. *The Vehicle Routing Problem, SIAM Monographs on Discrete Mathematics and Applications*, 9 :157–193, 2002.

[Desrochers *et al.*, 1988] Martin Desrochers, Jan Karel Lenstra Lenstra, Martin W.P. Savelsbergh, and François Soumis. Vehicle routing with time windows : Optimization and approximation. In B.L. Golden and A.A. Assad, editors, *Vehicle Routing : Methods and Studies*, pages 65–84, Amsterdam (The Netherlands), 1988.

[Desrosiers *et al.*, 1986] Jacques Desrosiers, Yvan Dumas, and François Soumis. A dynamic programming solution of the largescale single-vehicle dial-a-ride problem with time windows. *American Journal of Mathematical and Management Sciences*, 6 :301–325, 1986.

[Dial, 1995] Robert B. Dial. Autonomous dial-a-ride transit introductory overview. *Transportation Research Part C : Emerging Technologies*, 3 :261–275(15), October 1995.

[Diana and Dessouky, 2004] Marco Diana and Maged Dessouky. A new regret insertion heuristic for solving large-scale dial-a-ride problems with time windows. *Transportation Research Part B*, 38(6) :539–557, 2004.

[Diana, 2002] Marco Diana. *Methodologies for the tactical and strategic design of large-scale advance-request and real-time demand responsive transit services*. PhD dissertation, Politecnico di Torino, Dipartimento di Idraulica, Transporti e Infrastrutture, Torino (Italy), 2002.

[Diana, 2006] Marco Diana. The importance of information flows temporal attributes for the efficient scheduling of dynamic demand responsive transport services. *Journal of advanced Transportation*, 40(1) :23–46, 2006.

[Dumas *et al.*, 1991] Yvan Dumas, Jacques Desrosiers, and François Soumis. The pickup and delivery problem with time windows. *European Journal of Operational Research*, 54 :7–22, 1991.

[Durfee and Montgomery, 1991] Edmund H. Durfee and Thomas A. Montgomery. Coordination as distributed search in a hierarchical behavior space. *IEEE Transactions on Systems, Man, and Cybernetics, Special Issue on Distributed Artificial Intelligence*, SMC–21(6) :1363–1378, 1991.

[Durfee *et al.*, 1987] Edmund H. Durfee, Victor Lesser, and Daniel D. Corkill. Coherent cooperation among communicating problem solvers. In M. N. Huhns, editor, *Distributed Artificial Intelligence*, pages 29–58. Morgan Kaufmann Publishers, Los Alamos, CA, 1987.

[Durfee, 1988] Edmund H. Durfee. *Coordination of Distributed Problem Solvers*. Kluwer Academic Publishers, Norwell, MA (USA), 1988. 269 pages.

[El Fallah-Seghrouchni and Haddad, 1995] Amal El Fallah-Seghrouchni and Serge Haddad. A formal model for coordinating plans in multiagents systems. In *Proceedings of Intelligent Agents Workshop*, Augusta Technology Ltd, Brooks University. Oxford (UK), 1995.

[El Fallah-Seghrouchni and Haddad, 1996a] Amal El Fallah-Seghrouchni and Serge Haddad. A coordination algorithm for multi-agent planning. In Walter Van de Velde and John W. Perram, editors, *Modelling Autonomous Agents in a Multi-Agent Word*, volume 1038 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Eindhoven (The Netherlands), 1996.

[El Fallah-Seghrouchni and Haddad, 1996b] Amal El Fallah-Seghrouchni and Serge Haddad. A recursive model for distributed planning. In Gerhard Weiß, editor, *Proceedings of the International Conference on Multi-Agent Systems (ICMAS'96)*, pages 307–314, Kyoto (Japan), 1996. AAAI Press.

[El Fallah-Seghrouchni *et al.*, 2000] Amal El Fallah-Seghrouchni, Pavlos Moraitis, and Alexis Tsoukiàs. An aggregation-disaggregation approach for automated negotiation in multi-agent systems. In *Proceedings of the International Confernce on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'2000)*, 2000.

[El Fallah-Seghrouchni, 2001] Amal El Fallah-Seghrouchni. *Les modèles de coordination d'agents cognitifs*, pages 139–177. Hermès IC2 (information, commande, communication), 2001.

[Enoch *et al.*, 2004] Marcus Enoch, Stephen Potter, Graham Parkhurst, and Mark Smith. Intermode : Innovations in demand responsive transport. Technical report, Department for Transport and Greater Manchester Passenger Transport Executive, London (UK), 2004.

[Enoch *et al.*, 2006] Marcus Enoch, Stephen Potter, Graham Parkhurst, and Mark Smith. Why do demand responsive transport systems fail ? *85th annual meeting of the Transportation Research Board*, 2006.

[Ephrati *et al.*, 1995] Eithan Ephrati, Martha E. Pollack, and Jeffrey S. Rosenschein. A tractable heuristic that maximizes global utility through local plan combination. In Victor R. Lesser and Les Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems (ICMAS'95)*, pages 94–101, San Francisco, CA (USA), 1995. The MIT Press.

[Ferber *et al.*, 2000] Jacques Ferber, Olivier Gutknecht, Catholijn M. Jonker, Jean-Pierre Müller, and Jan Treur. Organization models and behavioural requirements specification for multi-agent systems. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS'00)*, pages 387–388, Boston, MA (USA), 2000. IEEE Press.

[Ferber, 1995] Jacques Ferber. *Les systèmes multiagents : Vers une intelligence collective*. Inter-Editions, 1995. 522 pages.

[Fischer *et al.*, 1994] Klaus Fischer, Norbert Kuhn, and Jörg Müller. Distributed, knowledge-based, reactive scheduling of transportation tasks. In *Proceedings of the Tenth Conference on Artificial Intelligence for Applications*, pages 47–53, San Antonio, TX (USA), 1994.

[Fischer *et al.*, 1995] Klaus Fischer, Jörg Müller, Markus Pischel, and Darius Schier. A model for cooperative transportation scheduling. In Victor R. Lesser and Les Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems (ICMAS'95)*, pages 109–116, Menlo park, CA (USA), 1995. AAAI Press / MIT Press.

[Fisher and Jaikumar, 1981] Marshall L. Fisher and Ramchandran Jaikumar. A generalized assignment heuristic for vehicle routing. *Networks*, 11 :109–124, 1981.

[Fisher, 1997] Marshall Fisher. Vehicle routing. In Michael O. Ball, Thomas L. Magnanti, Clyde Monma, and George Nemhauser, editors, *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science, ch. 1*, pages 1–79. North-Holland, 1997.

[Fok *et al.*, 2004] Chien-Liang Fok, Gruia-Catalin Roman, and Gregory Hackmann. A lightweight coordination middleware for mobile computing. In Rocco De Nicola, Gianluigi Ferrari, and Greg Meredith, editors, *Coordination'04 : Proceedings of the Sixth International Conference on Coordination Languages and Models*, volume 2949 of *Lecture Notes in Computer Science*, pages 135–151, Pisa (Italy), 2004. Springer-Verlag.

[Freeman *et al.*, 1999] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces : Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex (UK), 1999. 304 pages.

[Fu and Mephu Nguifo, 2004] Huaiguo Fu and Engelbert Mephu Nguifo. Etude et conception d'algorithmes de gÃ©nÃ©ration de concepts formels. *Revue d'Ingénierie des Systèmes d'Information (ISI)*, 9(3-4) :109–132, 2004.

[Fu and Teply, 1999] Liping Fu and Stan Teply. On-line and off-line routing and scheduling of dial-a-ride paratransit vehicles. In *Computer-Aided Civil and Infrastructure Engineering*, volume 14, pages 309–319. Blackwell Publishers, Oxford (UK), 1999.

[Gambardella *et al.*, 1999] Luca Maria Gambardella, Eric D. Taillard, and Giovanni Agazzi. MACS-VRPTW : A multiple ant colony system for vehicle routing problems with time windows. *New Ideas in Optimization*, pages 63–76, 1999.

[Ganter and Wille, 1999] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis.* Springer-Verlag, Heidelberg, Berlin (Germany), 1999.

[Ganter, 1984] Bernhard Ganter. Two basic algorithms in concept analysis. Technical report, Technische Hochschule Darmstadt, Darmstadt (Germany), 1984. Preprint 831.

[Garaix *et al.*, 2006] Thierry Garaix, Christian Artigues, Dominique Feillet, and Didier Josselin. Résolution de problèmes de transport à la demande avec chemins alternatifs. In *MOSIM 2006-Modélisation, optimisation et simulation des systèmes : défis et opportunités.* Hermès Science, 2006. ISBN 2-7430-0893-8.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Instractability : A Guide to the Theory of NP–Completeness.* W.H Freeman, San Francisco, CA (USA), 1979.

[Gelernter and Carriero, 1992] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2) :97–107, 1992.

[Gelernter, 1985] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1) :80–112, 1985.

[Gendreau and Potvin, 1998] Michel Gendreau and Jean-Yves Potvin. Dynamic vehicle routing and dispatching. In Teodor Gabriel Crainic and Gilbert Laporte, editors, *Fleet Management and Logistics*, pages 115–126. Kluwer Academic Publishers, 1998.

[Gendreau *et al.*, 1999] Michel Gendreau, Francois Guertin, Jean-Yves Potvin, and Eric D. Taillard. Parallel tabu search for real-time vehicle routing and dispatching. *Transportation Science*, 33(4) :381–390, 1999.

[Gendreau *et al.*, 2006] Michel Gendreau, Francois Guertin, Jean-Yves Potvin, and R. Séguin. Neighborhood search heuristics for a dynamic vehicle dispatching problem with pick-ups and deliveries. *Transportation Research Part C*, 14 :157–174, 2006.

[Georgeff, 1983] Michael P. Georgeff. Communication and interaction in multi-agent planning. In Michael R. Genesereth, editor, *Proceedings of the third national conference on Artificial intelligence (AAAI'83)*, pages 125–129, Washington D.C, WA (USA), 1983. AAAI Press.

[Gillett and Miller, 1974] B.E. Gillett and L.R. Miller. A heuristic algorithm for the vehicle-dispatch problem. *Operations Research*, 22 :340–349, 1974.

[Godin *et al.*, 1995] Robert Godin, Guy W. Mineau, Rokia Missaoui, and Hafedh Mili. Méthodes de classification conceptuelle basées sur les treillis de galois et applications. *Revue d'intelligence artificielle 9(2)*, 9(2) :105–137, 1995.

[Guénoche, 1990] A Guénoche. Construction du treillis de galois dune relation binaire. *Mathématique, Informatique et Sciences Humaines*, 28(109) :41–53, 1990.

[Halse, 1992] Karsten Halse. *Modeling and Solving Complex Vehicle Routing Problems.* PhD dissertation, Department for MathematicalModeling, Technical University of Denmark, 1992.

[Hayes-Roth, 1985] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3) :251–321, 1985.

[Hoare, 1985] Charles A.R. Hoare. *Communicating sequential processes.* Prentice Hall, 1985.

[Holland, 1975] John Holland. *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI : University of Michigan Press, 1975. 211 pages.

[Holzbacher, 1996] Anne Alexandra Holzbacher. A software environment for concurrent coordinated programming. In Paolo Ciancarini and Chris Hankin, editors, *Coordination'96 : Proceedings of the First International Conference on Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 249–266. Springer-Verlag, Cesena (Italy), 1996.

[Homberger and Gehring, 1999] Jörg Homberger and Hermann Gehring. Two evolutionary metaheuristics for the vehicle routing problem with time windows. *INFORMS*, 37(6) :297–318, August 1999.

[Horn, 2002] Mark E.T. Horn. Fleet scheduling and dispatching for demand-responsive passenger services. *Transportation Research C*, 10(1) :35–63, 2002.

[Housroum *et al.*, 2006] Haiyan Housroum, Tiente Hsu, Rémi Dupas, and Gilles Goncalves. A hybrid ga approach for solving the dynamic vehicle routing problem with time windows. In IEEE Computer Society, editor, *Proceedings of the IEEE Conference on Information and Communication Technologies : from Theory to Applications*, pages 3347–3352, Damascus, Syria, 2006.

[Huhns and Singh, 1994] Michael N. Huhns and Munindar P. Singh. CKBS-94 tutorial : Distributed artificial intelligence for information systems. Technical report, University of Keele, Keele (UK), June 1994.

[Ichoua *et al.*, 2000] Soumia Ichoua, Michel Gendreau, and Jean-Yves Potvin. Diversion issues in real-time vehicle dispatching. *Transportation Science*, 34(4) :426–438, 2000.

[Ichoua *et al.*, 2003] Soumia Ichoua, Michel Gendreau, and Jean-Yves Potvin. Vehicle dispatching with time-dependent travel times. *European Journal of Operational Research*, 144 :379–396, 2003.

[Jang *et al.*, 2004] Myeong-Wuk Jang, Amr Ahmed, and Gul Agha. A flexible coordination framework for application-oriented matchmaking and brokering services. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, IL (USA), 2004.

[Jennings *et al.*, 1998] Nicholas R. Jennings, Katia P. Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1) :7–38, 1998.

[Jennings, 1996] Nicholas R. Jennings. Coordination techniques for distributed artificial intelligence. In G. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, pages 187–210. John Wiley & Sons, 1996.

[Kearney *et al.*, 1994] Paul E. Kearney, Arvindra Sehmi, and Robert M. Smith. Emergent behaviour in a multi-agent economics simulation. In Cohn A G, editor, *Proceedings of the 11th European Conference on Artificial Intelligence*, London (UK), 1994. John Wiley.

[Kefi and Ghedira, 2004] Mariem Kefi and Khaled Ghedira. A multi-agent model for the vrptw. In *Proceedings of Urban Transport'04*, Germany, 2004.

[Ketchpel, 1994] Steven Ketchpel. Forming coalitions in the face of uncertain rewards. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 1) (AAAI '94)*, Menlo Park, CA (USA), 1994. American Association for Artificial Intelligence.

[Kiechle *et al.*, 2007] Guenter Kiechle, Karl Doerner, Michel Gendreau, and Richard Hartl. Patient transportation - dynamic dial-a-ride and emergency transportation problems. In *Proceedings of 6th triennial symposium on transportation analysis, TRISTAN VI*, Phuket Island (Thailand), June 2007.

[Kohout and Erol, 1999] Robert Kohout and Kutluhan Erol. In-Time agent-based vehicle routing with a stochastic improvement heuristic. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence (AAAI'99/IAAI'99)*, pages 864–869, Menlo Park, CA (USA), 1999. AAAI Press.

[Komorowski *et al.*, 1998] Jan Komorowski, Zdzislaw Pawlak, Lech Polkowski, and Andrzej Skowron. Rough sets : a tutorial. In Sankar K. Pal and Andrzej Skowron, editors, *Rough-Fuzzy Hybridization : A New Method for Decision Making*. Springer-Verlag, Singapore, 1998.

[Kuznetsov and Obiedkov, 2002] Sergei O. Kuznetsov and Sergei A. Obiedkov. Comparing the performance of algorithms for generating concept lattices. *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2–3) :189–216, 2002.

[Larsen, 1999] Jesper Larsen. *Parallelization of the Vehicle Routing Problem with Time Windows*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 1999.

[Larsen, 2000] Allan Larsen. *The Dynamic Vehicle Routing Problem*. PhD thesis, University of Denmark, 2000.

[Lee *et al.*, 1994] Insup Lee, Patrice Bremond-Gregoire, and Richard Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE, Special Issue on Real-Time Systems*, pages 158–171, January 1994.

[Lenstra and Kan, 1981] Jan Karel Lenstra Lenstra and A.H.G. Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11 :221–227, 1981.

[Leong and Liu, 2006] Hon Wai Leong and Ming Liu. A multi-agent algorithm for vehicle routing problem with time window. In *SAC '06 : Proceedings of the 2006 ACM symposium on Applied computing*, pages 106–111, New York, NY (USA), 2006. ACM Press.

[Lesser and Corkill, 1988] Victor R. Lesser and Daniel D. Corkill. Functionally accurate, cooperative distributed systems. *Distributed Artificial Intelligence*, pages 295–310, 1988.

[Lesser, 1995] Victor R. Lesser. Multiagent Systems : An Emerging Subdiscipline of AI. *ACM Computing Surveys*, 27(3) :340–342, January 1995.

[Linden *et al.*, 2006] Isabelle Linden, Jean-Marie Jacquet, Koen De Bosschere, and Antonio Brogi. On the expressiveness of timed coordination models. *Science of Computer Programming*, 61(2) :152–187, 2006.

[Liu and Shen, 1999] Fuh-Hwa Liu and Sheng-Yuan Shen. A route-neighborhood-based metaheuristic for vehicle routing problem with time windows. *European Journal of Operational Research*, 118 :485–504, 1999.

[Madsen *et al.*, 1995] Oli B.G. Madsen, Hans F. Ravn, and Jens M. Rygaard. A heuristic algorithm for a dial-a-ride problem with time windows, multiple capacities, and multiple objectives. *Operations Research*, 60 :193–208, 1995.

[Mageean and Nelson, 2003] Jenny Mageean and John D. Nelson. The evaluation of demand responsive transport services in europe. *Journal of Transport Geography*, 11(4) :255–270, 2003.

[Milner, 1989] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989. 272 pages.

[Minsky *et al.*, 2001] Naftaly H. Minsky, Yaron Minsky, and Victoria Ungureanu. Safe tuplespace-based coordination in multiagent systems. *Applied Artificial Intelligence*, 15(1) :11–33, 2001.

[Minsky, 2004] Naftaly Minsky. Law Governed Interaction (LGI) : A distributed coordination and control mechanism (An introduction, and a reference manual), department of computer science Rutgers University, New Jersey, NJ (USA) http ://www.moses.rutgers.edu/documentation/, october 2004.

[Mitrovic-Minic and Laporte, 2004] Snezana Mitrovic-Minic and Gilbert Laporte. Double-horizon based heuristics for the dynamic pickup and delivery problem with time windows. *Transportation Research Part B*, 38 :669–685, 2004.

[Moraitis and Tsoukiàs, 1996] Pavlos Moraitis and Alexis Tsoukiàs. A multicriteria approach for distributed planning and negotiationin multiagent systems. In Gerhard Weiß, editor, *Proceedings of the International Conference on Multi-Agent Systems (ICMAS'96)*, pages 212–219, Kyoto (Japan), 1996. AAAI Press.

[Moraitis and Tsoukiàs, 1999] Pavlos Moraitis and Alexis Tsoukiàs. Dynamic planning model for agents preference satisfaction : first results. In *Proceedings of the International Conference on Intelligent Agent Technology (IAT'99)*, pages 182–191, 1999.

[Nash, 1950] John F. Nash. The bargaining problem. *Econometrica*, 18(2) :155–162, 1950.

[Nixon *et al.*, 2007] Lyndon Nixon, Olena Antonechko, and Robert Tolksdorf. Towards semantic tuplespace computing : the semantic web spaces system. In *SAC '07 : Proceedings of the 2007 ACM symposium on Applied computing*, pages 360–365, New York, NY, USA, 2007. ACM Press.

[Norris, 1978] Eugene M. Norris. An algorithm for computing the maximal rectangles in a binary relation. *Revue Romaine de Mathématiques Pures et Appliquées*, XXIII(4) :243–250, 1978.

[Nourine and Raynaud, 1999] Lhouari Nourine and Olivier Raynaud. A fast algorithm for building lattices. *Information Processing Letters*, 71(5-6) :199–204, 1999.

[Nwana *et al.*, 1996] Hyacinth S. Nwana, Lyndon C. Lee, and Nicholas R. Jennings. Coordination in software agent systems. *The British Telecom Technical Journal*, 14(4) :79–88, 1996.

[Nwana, 1994] Hyacinth S. Nwana. Negotiation strategies : An overview. Technical report, British Telecom Labs, 1994.

[Omicini and Zambonelli, 1999] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3) :251–269, 1999.

[Omicini *et al.*, 2004] Andrea Omicini, Alessandro Ricci, Mirko Viroli, and Giovanni Rimassa. Integrating objective & subjective coordination in multiagent systems. In *SAC'04 :Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 449–455, New York, NY (USA), 2004. ACM Press.

[Paessens, 1988] Heinrich Paessens. The savings algorithm for the vehicle routing problem. *European Journal of Operational Research*, 34(3) :62–212, 1988.

[Papadopoulos and Arbab, 1998] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in Computers*, 46 :55, 1998.

[Paton and Diaz, 1999] Norman W. Paton and Oscar Diaz. Active database systems. *ACM Computing Surveys*, 31(1) :63–103, 1999.

[Picco and Buschini, 2002] Gian Pietro Picco and Marco L. Buschini. Exploiting transiently shared tuple spaces for location transparent code mobility. In F. Arbab and C. Talcott, editors, *Proceedings of the 5th International Conference on Coordination Models and Languages*, York (UK), 2002. Springer-Verlag.

[Picco *et al.*, 1999] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME : Linda meets mobility. In *International Conference on Software Engineering*, pages 368–377, 1999.

[Platon *et al.*, 2005] Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Overhearing and direct interactions : Point of view of an active environment, a preliminary study. In Danny Weyns, H.V.D Parunak, and Fabien Michel, editors, *Environments For Multiagents Systems I*, volume 3374 of *Lecture Notes in Artificial Intelligence*, pages 121–138. Springer-Verlag, 2005.

[Plotkin, 1981] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, Aarhus (Denmark), 1981.

[Potvin *et al.*, 2006] Jean-Yves Potvin, Ying Xu, and Ilham Benyahia. Vehicle routing and scheduling with dynamic travel times. *Computers & Operations Research*, 33(4) :1129–1137, 2006.

[Psaraftis, 1980] Harilaos N. Psaraftis. A dynamic programming solution to the single-vehicle, many-to-many, immediate request dial-a-ride problem. *Transportation Science*, 14 :130–154, 1980.

[Psaraftis, 1983] Harilaos N. Psaraftis. An exact algorithm for the single-vehicle, many-to-many dial-a-ride problem with time windows. *Transportation Science*, 17 :351–357, 1983.

[Psaraftis, 1988] Harilaos N. Psaraftis. *Vehicle Routing : Methods and Studies*, pages 223–248. Elsevier Science Publishers Ltd., B.V. (North Holland), 1988.

[Ralphs, 1995] Theodore Kenneth Ralphs. *Parallel branch and cut for vehicle routing.* PhD thesis, Cornell University, Ithaca, NY (USA), 1995.

[Ricci *et al.*, 2006] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Programming MAS with artifacts. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems III*, volume 3862 of *Lecture Notes in Computer Science*, pages 206–221. Springer-Verlag, 2006.

[Rochat and Taillard, 1995] Yves Rochat and Eric D. Taillard. Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics*, 1 :147–167, 1995.

[Roldan *et al.*, 2003] Ana M. Roldan, Ernesto Pimentel, and Antonio Brogi. Safe composition of linda-based components. *Electronic Notes in Theoretical Computer Science*, 82(6), 2003.

[Rowstron and Wood, 1998] Alan I. T. Rowstron and Anthony M. Wood. Solving the linda multiple rd problem using the copy-collect primitive. *Science of Computer Programming*, 31(2-3) :335–358, 1998.

[Saunier *et al.*, 2006] Julien Saunier, Flavien Balbo, and Fabien Badeig. Environment as active support of interaction. In *Proceeding of the third workshop on Environment for Multiagent Systems (E4MAS'06)*, Hakodate (Japan), 2006.

[Savelsbergh and Sol, 1998] Martin W.P. Savelsbergh and Marc Sol. DRIVE : dynamic routing of independent vehicles. *Operations Research*, 46 :474–490, 1998.

[Schelfthout and Holvoet, 2004] Kurt Schelfthout and Tom Holvoet. Objectplaces : An environment for situated multiagent systems. In *Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, pages 1500–1501, july 2004.

[Schumacher, 2001] Michael Schumacher. *Objective coordination in multi-agent system engineering : design and implementation.* Springer-Verlag New York, Inc., Secaucus, NJ (USA), 2001.

[Shaw *et al.*, 1995] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering*, 21(4) :314–335, 1995.

[Shehory and Kraus, 1996] Onn Shehory and Sarit Kraus. Formation of overlapping coalitions for precedence-ordered task-execution among autonomous agents. In Gerhard Weiß, editor, *Proceedings of the International Conference on Multi-Agent Systems (ICMAS'96)*, pages 330–337, Kyoto (Japan), 1996. AAAI Press.

[Sichman, 1998] Jaime Simão Sichman. DEPINT : Dependence-based coalition formation in an open multi-agent scenario. *Journal of Artificial Societies and Social Simulation*, 2(1), 1998.

[Smith, 1980] Reid G. Smith. The contract net protocol : High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12) :1104–1113, December 1980.

[Solomon, 1987] Marius M. Solomon. Algorithms for the vehicle routing and scheduling with time window constraints. *Operations Research*, 15 :254–265, 1987.

[Suna, 2005] Alexandru Suna. *CLAIM et SyMPA : Un environnement pour la programmation d'agents intelligents et mobiles*. PhD dissertation, Laboratoire d'Informatique de l'université Paris VI (Lip6), 2005.

[Sycara *et al.*, 1997] Katia P. Sycara, Keith Decker, and Mike Williamson. Middle-agents for the Internet. In *Proceedings of the 15th Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 578–583, 1997.

[Sycara, 1990] Katia P. Sycara. Multiagent compromise via negotiation. *Distributed artificial intelligence : vol. 2*, pages 119–137, 1990.

[Sycara, 1998] Katia P. Sycara. Multiagent Systems. *AI Magazine*, 19(2) :79–92, 1998.

[Taillard *et al.*, 1997] Eric D. Taillard, Philippe Badeau, Michel Gendreau, François Geurtin, and Jean-Yves Potvin. A tabu search heuristic for the vehicle routing problem with time windows. *Transportation Science*, 31 :170–186, 1997.

[Thangiah *et al.*, 2001] Sam R. Thangiah, Olena Shmygelska, and William Mennell. An agent architecture for vehicle routing problems. In *Proceedings of the 2001 ACM symposium on Applied computing (SAC '01)*, pages 517–521, New York, NY (USA), 2001. ACM Press.

[Tolksdorf and Glaubitz, 2001] Robert Tolksdorf and Dirk Glaubitz. Xmlspaces for coordination in web-based systems. In *Proceedings of the 10th IEEE International Workshops on Enabling Technologies (WETICE'01)*, pages 322–327, Washington D.C, WA (USA), 2001. IEEE Computer Society.

[Vauvert and El Fallah-Seghrouchni, 2000a] Guillaume Vauvert and Amal El Fallah-Seghrouchni. Coalition formation for egoistic agents. In *Proceedings of the International Confernce on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'2000)*, 2000.

[Vauvert and El Fallah-Seghrouchni, 2000b] Guillaume Vauvert and Amal El Fallah-Seghrouchni. Formation de coalitions pour agents rationnels. In *Proceeding of JLIPN2000*, Villetaneuse (France), 2000.

[von Martial, 1992] Frank von Martial. *Coordinating Plans of Autonomous Agents*, volume 610 of *Lecture Notes on Artificial Intelligence*. Springer Verlag, Berlin, 1992.

[Wegner, 1996] Peter Wegner. Coordination as constrained interaction. In Paolo Ciancarini and Chris Hankin, editors, *Coordination'96 : Proceedings of the First International Conference*

on Coordination Languages and Models, volume 1061 of *Lecture Notes in Computer Science*, pages 28–33. Springer-Verlag, Cesena (Italy), 1996.

[Werkman, 1990] Keith Werkman. Knowledge-based model of negotiation using shareable perspectives. In *Proceedings of the Tenth International Workshop on Distributed AI*, Bandera, TX (USA), October 1990.

[Weyns and Holvoet, 2004] Danny Weyns and Tom Holvoet. Formal model for situated multi-agent systems. *Formal Approaches for Multi-agent Systems, Special Issue of Fundamenta Informaticae*, 63(2-3), 2004.

[Weyns, 2006] Danny Weyns. *An Architecture-Centric Approach for Software Engineering with Situated Multiagent Systems*. PhD dissertation, Katholieke Universiteit Leuven, Leuven (Belgium), 2006.

[Wong and Sycara, 2000] Hao Chi Wong and Katia P. Sycara. A taxonomy of middle-agents for the Internet. In *Proceedings of the Fourth International Conference on MultiAgent Systems*, pages 465–466, July 2000.

[Wooldridge and Jennings, 1995] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents : Theory and practice. *Knowledge Engineering Review*, 10(2) :115–152, 1995.

[Wooldridge, 2002] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, Chichester (UK), February 2002. 256 pages.

[Wyckoff *et al.*, 1998] Peter Wyckoff, Stephen McLaughry, Tobin Lehman, and Daniel Ford. Tspaces. *IBM Systems Journal*, 37(3) :454–474, 1998.

[Zargayouna and Bsiri, 2002] Mahdi Zargayouna and Sandra Bsiri. Textmining : Découverte des règles non redondantes et hiérarchiques. mémoire de fin d'études, Institut Superieur de Gestion, Tunis, Tunisie, june 2002. 100 pages.

[Zargayouna *et al.*, 2006a] Mahdi Zargayouna, Flavien Balbo, and Julien Saunier. Agent information server : a middleware for traveler information. In Oguz Dikenelli, Marie-Pierre Gleizes, and Alessandro Ricci, editors, *Engineering Societies in the Agents World VI*, volume 3963 of *Lecture Notes in Artificial Intelligence*, pages 3–16. Springer-Verlag, Kusadasi, Aydin (Turkey), 2006.

[Zargayouna *et al.*, 2006b] Mahdi Zargayouna, Julien Saunier Trassy, and Flavien Balbo. Property based coordination. In Jerome Euzenat and John Domingue, editors, *Artificial Intelligence : Methodology, Systems, Applications*, volume 4183 of *Lecture Notes in Artificial Intelligence*, pages 3–12. Springer-Verlag, 2006.

[Zargayouna, 2006] Mahdi Zargayouna. Time constrained vrp : An agent environment-perception model. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *Proceedings of the European Conference on Artificial Intelligence*, pages 849–850. IOS Press, Riva del Garda (Italy), 2006.

[Zhu and Ong, 2000] Kenny Qili Zhu and Kar-Loon Ong. A reactive method for real time dynamic vehicle routing problem. In *12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00)*, pages 176–179, Vancouver (Canada), 2000. IEEE Computer Society.

[Zlotkin and Rosenschein, 1994] Gilad Zlotkin and Jeffrey S. Rosenschein. Coalition, cryptography, and stability : mechanisms for coalition formation in task oriented domains. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 1) (AAAI '94)*, pages 432–437, Menlo Park, CA (USA), 1994. AAAI Press.