Simulations orientées-interaction des systèmes complexes

THÈSE

présentée et soutenue publiquement le 6 Décembre 2010

pour l'obtention du

Doctorat de l'Université Lille 1 - Sciences et Technologies

(spécialité informatique)

par

Yoann Kubera

Composition du jury

Président :	Pr. Jean-Luc DEKEYSER	Université Lille 1
Rapporteurs :	DR. Yves DEMAZEAU	Université Grenoble I
	Pr. Jacques FERBER	Université de Montpellier II
Examinateurs :	Dr. Olivier GUTKNECHT	Fotonauts Inc
	Pr. Juan PAVÓN	Universidad Complutense Madrid
Directeur :	Pr. Philippe MATHIEU	Université Lille 1
Encadrant :	Dr. Sébastien PICAULT	Université Lille 1

UNIVERSITÉ LILLE 1 - SCIENCES ET TECHNOLOGIES
Laboratoire d'Informatique Fondamentale de Lille — UPRESA 8022
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 – Télécopie : +33 (0)3 28 77 85 37 – email : `direction@lifl.fr`

Mis en page et composé avec LaTeX et la classe thlifl.

Remerciements

Ce manuscrit est l'aboutissement de travaux qui n'auraient pas vu le jour et n'auraient pas abouti sans la participation, l'aide et le soutien d'un grand nombre de personnes.

En premier lieu, je remercie les membres de mon jury d'avoir accepté de juger mon travail et les « quelques » pages de mon manuscrit. Merci à M. Yves Demazeau et M. Jacques Ferber de m'avoir fait l'honneur de rapporter ma thèse. Merci à M. Olivier Gutknecht et M. Juan Pavón d'avoir accepté d'assister à ma soutenance et d'examiner mes travaux. Merci enfin à M. Jean-Luc Dekeyser d'avoir présidé mon jury.

Je tiens à remercier Philippe Mathieu et Sébastien Picault de m'avoir encadré pendant ces quatre dernières années. Leur implication et leur passion pour la recherche ont rendu ces années particulièrement stimulantes. Leur confiance, leurs conseils, leur soutien et les nombreuses discussions que nous avons pu avoir m'ont été d'une grande aide tout au long de ma thèse. Je me joins aux forêts d'Amazonie pour les remercier d'avoir freiné ma frénésie rédactionnelle, et d'avoir ainsi sauvé la vie à moult arbres en m'empêchant de dépasser les 300 pages de manuscrit.

Je remercie également les anciens ou actuels membres de l'équipe SMAC du LIFL que j'ai pu côtoyer. L'ambiance et la bonne humeur du Bureau 14A, des repas d'équipe et autres « évènements mondains » ont rendu mes années de thèse particulièrement plaisantes. Merci donc à Antoine, Benoît, Bruno, Cédric, Damien, David, François, Iryna, Jean-Baptiste, Jean-Christophe, Jean-Paul, Julien, Laetitia, Maxime, Patricia, Philippe, Sam, Sébastien, Tony, Yann. Je remercie en particulier mes compagnons de voyage Antoine, Benoît et Tony pour les nombreuses discussions que nous avons pu avoir, alliant sujets sérieux tels que le raisonnement d'entités virtuelles et moins sérieux comme la survie des trekkers imprudents en milieu sauvage, le Panda à qui il ne faut pas dire non ou les mœurs amoureuses du canard colvert.

Merci aussi à tous ceux avec qui j'ai partagé repas, pauses café, un verre en terrasse ou un moment suite à une dure journée de travail : Antonin, Amaury, Asli, Benjamin, Bruno, Fadila, Jean-François, Jean-Philippe, Loïc, Olivier, Patricia, Raphaël, Simon et tous ceux que je ne mentionne pas mais que je n'oublie pas. Je leur suis reconnaissant de m'avoir fourni ces moments de détente ô combien nécessaires pour dépasser les périodes de stress et de doute qui jalonnent la thèse.

Je remercie enfin chaleureusement ma famille, en particulier mes parents Marie-Thérèse et Serge et ma sœur Élise, pour le soutien et la compréhension dont ils ont fait preuve lors de ces années marquant la fin de mes longues études. Je le promets, c'est fini !

Table des matières

Introduction

1 Contexte

De tout temps, la compréhension des mécanismes intrinsèques au monde qui nous entoure a été moteur d'avancées technologiques, techniques et logistiques. Nous pouvons citer en exemple les travaux de Léonard de Vinci qui, en s'inspirant du vol des chauves-souris, a imaginé des machines volantes ou, plus récemment les travaux d'une société de San Francisco ayant conçu des écrans plats iMoD (pour *interferometric MoDulator*) consommant le dixième de l'énergie nécessaire à un écran à cristaux liquides, en s'inspirant de certains papillons aux des ailes aux couleurs éclatantes malgré l'absence totale de pigments colorés.Le prérequis à de telles avancées est la compréhension de ces mécanismes, ce qui est en particulier le sujet des sciences expérimentales, dont les principes sont résumés par le chimiste Michel-Eugène Chevreul en les termes qui suivent :

> « *Un phénomène frappe vos sens ; vous l'observez avec l'intention d'en découvrir la cause, et pour cela vous en supposez une dont vous cherchez la vérification en instituant une expérience. Si l'hypothèse n'est pas fondée, vous en faites une nouvelle que vous soumettez à une nouvelle expérience, et cela jusqu'à ce que le but soit atteint, si toutefois l'état de la science le permet. (...) [L'expérience constitue alors] le critérium de l'exactitude du raisonnement dans la recherche des causes ou de la vérité.* » [Che56].

L'expérimentation réelle a ses limites : il est par exemple impossible d'établir un protocole expérimental vérifiant que l'apparition d'une termitière est due à un comportement simple de termites, qui consiste à ramasser un bout de bois et à le déposer dès qu'un second bout de bois est rencontré. Certains éléments d'un phénomène peuvent ainsi échapper à tout contrôle des expérimentateurs.

Ce problème a donné naissance à une discipline en informatique appelée *simulation informatique*, qui consiste à reproduire de manière artificielle un phénomène, en construisant puis en implémentant un *modèle*, *i.e.* une représentation abstraite du phénomène. Les travaux présentés dans cette thèse se placent dans le cadre plus particulier des *simulations dites explicatives*, qui visent à expliquer l'apparition d'un phénomène macroscopique (par exemple la formation d'une termitière) par la description du comportement microscopique des entités y participant (par exemple le comportement individuel de termites). La construction de telles simulations n'est pas aisée, car elle nécessite d'aller de descriptions d'hypothèses exprimées en langage naturel des experts du domaine simulé (par exemple des biologistes, des sociologues, *etc.*) jusqu'à une implémentation de ces descriptions à l'aide d'un langage de programmation. Il faut alors retranscrire aussi fidèlement que possible les spécifications des experts du domaine dans l'implémentation, mais aussi compléter ces spécifications par des informations nécessaires à l'implémentation, sans dénaturer le modèle du phénomène.

Les paradigmes de modélisation et de programmation reposant sur des métaphores issues des phénomènes simulés facilitent la construction de telles simulations. L'une des approches les plus prolifiques reposant sur ces principes est la modélisation centrée sur le concept d'agent, *i.e.* d'entité logicielle décidant de manière autonome des actions qu'elle entreprend. En effet, ce concept est très proche du concept d'entité composant le phénomène étudié par la simulation, que sont les molécules, les animaux, les humains, *etc.* L'application de ce paradigme à la simulation informatique a alors donné naissance à un nouveau courant de conception de simulations, appelé *simulation multi-agents*, dans lequel se placent les travaux présentés dans ce manuscrit.

2 Motivations

La transition entre des descriptions exprimées en langage naturel et leur implémentation ne peut être réalisée qu'à l'aide d'outils appropriés. Cela est d'autant plus vrai dans les simulations « large échelle » contenant un grand nombre d'agents se comportant de manière très diversifiée et interagissant de manière variée entre eux. En effet, la masse d'informations requiert une structure appropriée dans le modèle et un procédé facilitant sa construction. Les approches les plus appropriées à la modélisation de telles simulations sont celles accompagnant les concepteurs dans la construction graduelle du modèle de la simulation à l'aide de différents outils graphiques, tout en automatisant son implémentation. Nous les qualifions alors d'approches transversales. Actuellement, les approches multi-agent reposant sur ce principe ne sont pas adaptées à la simulation, à cause de la structure du modèles qu'elles utilisent. En effet, les seules interactions entre agents y sont des échanges de messages. En simulation, « interaction » a un sens plus large, puisque cette notion représente aussi les interactions fondamentales de la physique telles que la gravitation, ou toute autre action impliquant deux entités. Une réaction d'oxydo-réduction qui implique à la fois un oxydant et un réducteur, ou le fait qu'un prédateur puisse manger une proie en sont des exemples.

Peu d'approches reposent actuellement sur une représentation explicite de telles interactions : cette notion est le plus souvent distribuée dans le comportement des différents agents. Celles s'en approchant le plus sont dédiées à la modélisation d'interactions fondamentales de la physique ou reposent sur l'application de la théorie des affordances à la simulation multi-agents. Elles ne sont toutefois pas satisfaisantes, car sont dédiées à des architectures comportementales très spécifiques ou reposent sur des modèles actuellement incomplets et ne permettent donc pas d'établir une approche transversale.

3 Thèse

Dans ce manuscrit, nous proposons une approche générique de conception de simulations, intitulée **Interaction Oriented Design of Agent simulations** (IODA), cumulant à la fois les avantages des approches transversales, qui permettent de concevoir progressivement des modèles contenant un grand nombre d'informations, mais aussi des approches centrées sur la notion d'interaction appropriées à la simulation informatique.

Pour des questions d'uniformité, il est bien plus simple de concevoir de telles simulations en considérant que :

1. **toute entité** pertinente du phénomène simulé est **concrétisée par un agent** ;
2. **tout comportement** d'un agent est **concrétisé par une interaction** ;
3. toute **interaction** est **décrite indépendamment** des spécificités des agents.

Une interaction doit alors représenter tout type d'action impliquant simultanément plusieurs agents.

Cette unification de la représentation des connaissances n'est possible que si la simulation est **régie par un seul algorithme**, permettant de **conserver la structure du modèle lors de l'implémentation** et d'**implémenter automatiquement** un modèle. Il est alors fondamental de **séparer ce qui est déclaratif de ce qui est procédural**, c'est-à-dire séparer « agents », « interactions » et « moteur de simulation » autant dans le modèle que lors de l'implémentation. Ces trois éléments doivent donc inévitablement être réifiés dans des entités logicielles séparées.

Enfin, une telle approche n'est transversale que si elle accompagne non seulement le passage du modèle à l'implémentation, mais aussi la **construction graduelle d'un modèle**. Pour cela, un modèle doit être conçu en passant de **descriptions de niveau macroscopique du système** (les organisations et les interactions) **à des description de niveau microscopique** (les actions que les agents sont capables de faire et leur processus comportemental).

Les avantages attendus d'une telle approche sont multiples. En voici les principaux.

Conception guidée : La conception du modèle est guidée, pour y introduire progressivement des connaissances de plus en plus fines du phénomène. Débuter la conception sur des aspects macroscopiques permet d'avoir une vue d'ensemble sur le modèle du phénomène conçu, qui est difficilement obtenue

si l'on se concentre directement sur le comportement des agents. Cette approche est indispensable à la conception de simulations large échelle, *i.e.* de simulations contenant un grand nombre d'agents se comportant différemment et entretenant un grand nombre d'interactions variées.

Expertise en informatique : Les éléments du modèle nécessitant une expertise en informatique apparaissent lors des descriptions fines du modèle et figurent donc à la fin du processus de conception. Cela permet d'impliquer des experts du domaine dans le processus de conception plus longtemps qu'avec une spécification directe du comportement et réduit donc les problèmes liés à l'interprétation de leurs propos pour construire le modèle.

Révisions de modèle : On passe graduellement de descriptions macroscopiques du phénomène, qui sont observables et donc relativement objectives, à des descriptions microscopiques, qui relèvent des hypothèses relatives à l'origine du phénomène simulé. **Puisque les hypothèses sont les éléments les plus susceptibles d'être modifiés lors des itérations du processus de simulation, leur introduction tardive dans le modèle réduit les efforts de révision du modèle.**

Réutilisation logicielle : La séparation entre ce que les agents sont capables de faire (*i.e.* les interaction) et le processus qu'ils utilisent pour décider de ce qu'ils font (*i.e.* la sélection d'interaction) favorise la réutilisation logicielle à deux niveaux :

- une interaction peut être réemployée chez des agents ayant des processus décisionnels différents ;
- une interaction peut être réutilisée dans d'autres simulations du même domaine. Des bibliothèques d'interactions sont ainsi conçues au fil des simulations.

De plus, les révisions du modèle sont facilitées puisqu'une modification du processus de décision des agent n'implique pas systématiquement un révision des actions qu'ils sont capables de faire.

Automatisation de l'implémentation : Il est possible d'automatiser partiellement ou entièrement la transformation du modèle en une implémentation, réduisant ainsi la possibilité d'introduire de mauvaises interprétations du modèle.

Vérification et validation : Dans le cas où le simulateur ne fournit pas les résultats escomptés, la conservation de la structure du modèle lors de l'implémentation favorise l'interprétation dans le modèle d'une erreur identifiée dans le programme.

4 Une pyramide d'outils conceptuels et logiciels

Afin de répondre à la problématique posée, nous décrivons dans ce manuscrit l'**approche transversale de conception intitulée IODA** (*Interaction Oriented Design of Agent simulations*), qui facilite la construction d'un vaste ensemble de simulations à l'aide d'une représentation des connaissances **centrée sur les interactions**. Nous explorons de plus selon la perspective de IODA certaines problématiques simulation et en dégageons des extensions facilitant la conception de certaines catégories de simulations. Nous définissons ainsi une approche reposant sur un **cœur générique pouvant être complexifié par une ou plusieurs extensions lorsque la complexité du phénomène simulé le nécessite**.

À cet effet, nous définissons d'une part le cœur de l'approche IODA (décrit dans la partie II), qui facilite la construction d'un vaste ensemble de simulations à l'aide d'une représentation des connaissances centrée sur les interactions.

D'autre part, en nous appuyant sur cette caractérisation « minimale » de IODA, nous explorons trois problématiques de simulation selon la perspective d'une approche centrée sur les interactions :

1. Faut-il nécessairement des interactions complexes afin de modéliser des comportements complexes ?
2. Comment utiliser IODA afin d'éviter d'introduire des biais dans une simulation ?
3. Comment adapter le concept d'héritage des langages objets à la simulation, afin de faciliter l'ingénierie des connaissances ?

Nous utilisons ces études afin de définir quatre extensions disjointes de IODA (décrites dans la partie III) qui fournissent des concepts et des outils méthodologiques traitant efficacement de ces problèmes.

Ce manuscrit suit cette logique, que nous schématisons sur la figure 1.

4.1 Cœur de l'approche

Dans IODA, nous cherchons à cumuler à la fois les avantages des approches transversales qui permettent de concevoir progressivement des modèles contenant un grand nombre d'informations, mais aussi des approches accordant un sens plus large à la notion d'interaction.

Les travaux réalisés dans cette thèse concrétisent les efforts initiés en 2001 par Jean-Christophe Routier, Philippe Mathieu et Sébastien Picault [MRU01, MP05, DMR05, MP06, MPR07] pour définir des modèles, des algorithmes et une méthodologie généraux pour la conception de simulations.

Le cœur de l'approche définie dans cette thèse, que nous décrivons dans la partie II, est caractérisée par trois éléments différents, résumés dans la figure 2.

1. La **base théorique de IODA** composée :
   - d'**un modèle formel** décrivant les divers concepts de l'approche IODA à l'aide du formalisme mathématique ;
   - d'**une méthodologie de conception** fournissant des outils graphiques permettant de construire progressivement un modèle pouvant contenir un grand nombre d'agents et d'interactions différents ;
   - d'**algorithmes** exploitant les informations du modèle afin de fournir une implémentation univoque de la simulation.
2. La **plateforme de simulation** appelée JEDI (Java Environment for the Design of agent Interactions) qui constitue **une implémentation fidèle des concepts** de IODA dans le langage de programmation JAVA ;
3. L'**environnement de développement intégré** nommé JEDI-BUILDER qui constitue un prototype d'implémentation de la méthodologie de conception IODA. Ce dernier permet de construire un modèle IODA dans une interface graphique et d'en générer automatiquement le code pour la plateforme de simulation JEDI.

JEDI et JEDI-BUILDER confirment donc la faisabilité de l'approche IODA.

4.2 Exploration du potentiel applicatif

L'approche IODA permet de concevoir des simulations en centrant la conception sur les interactions. Ce changement de point de vue de modélisation ouvre un grand nombre de perspectives que nous explorons selon trois thématiques différentes. Nous utilisons ces études afin de définir quatre extensions disjointes de IODA.

Premièrement, nous questionnons la nécessité d'utiliser des interactions complexes afin de modéliser des comportements complexes. Cette étude a abouti à l'identification de **patrons de conception** permettant de décomposer la plupart des interactions complexes en un ensemble d'interactions simples, ainsi qu'une extension de IODA à un certain type d'interaction complexe ne pouvant être décomposée simplement.

Deuxièmement, nous explorons comment utiliser notre représentation centrée-interaction afin d'**éviter d'introduire des biais** dans une simulation. Pour cela, nous cherchons à faire apparaître explicitement des choix de conception usuellement implicites dans la plupart des autres approches de conception. L'apparition explicite de ces choix dans le modèle permet de décrire de manière formelle et univoque certaines décisions d'implémentation. Deux extensions du modèle formel, de la méthodologie et des algorithmes de simulations de IODA, découlent de cette étude. La première porte sur le problème de la participation simultanée d'un agent à plusieurs interaction. La seconde porte sur le problème de la concrétisation de comportements stochastiques chez des agents réactifs, dans une architecture comportementale générique.

Enfin, nous étudions comment **transposer les concept de l'héritage** des langages orientés-objets à notre approche, afin de favoriser la factorisation d'éléments sémantiques corrélés du modèle et donc mieux supporter la conception de simulations large échelle. A cette occasion, nous nous reposons sur

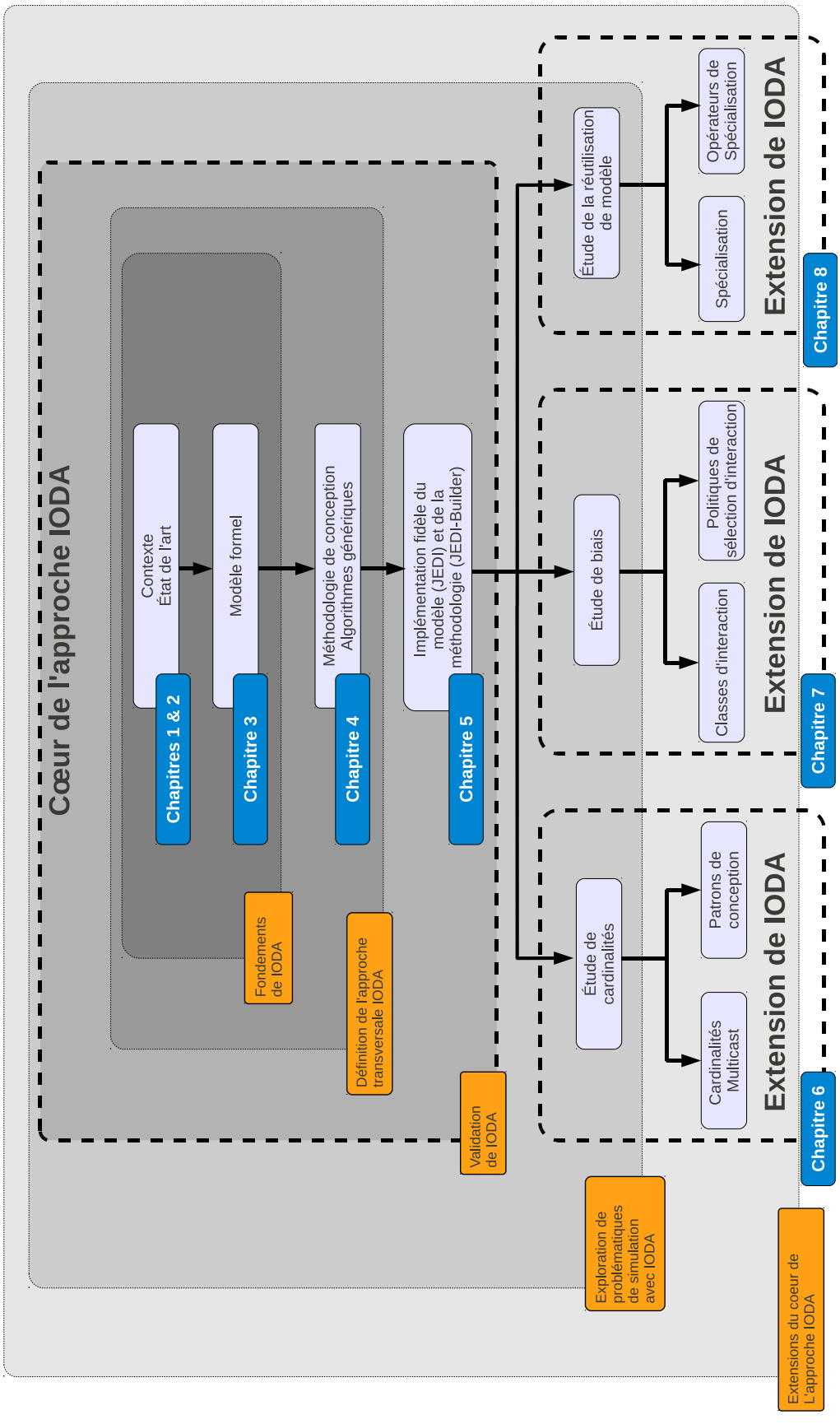


FIGURE 1 – Organisation du manuscrit de thèse et de ses contributions.

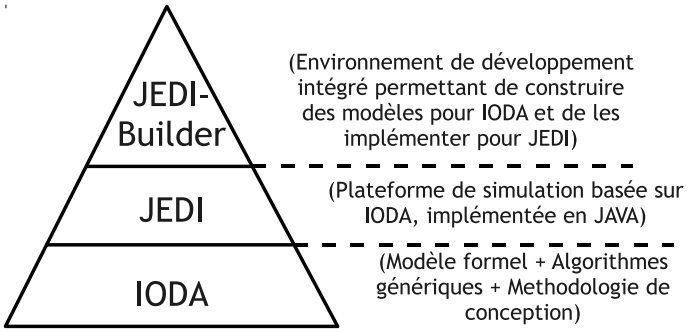


FIGURE 2 – La première contribution de cette thèse : élaboration d'un modèle formel centré sur les interactions, accompagné d'une méthodologie facilitant leur conception, d'algorithmes assurant leur implémentation univoque (IODA) et élaboration d'une implémentation fidèle du modèle (JEDI) complétée par un IDE de conception (JEDI-BUILDER).

notre approche centrée-interaction afin de mieux prendre en compte certains problèmes d'ingénierie des connaissances. Nous définissons pour cela une relation de spécialisation entre agents, qui permet d'exprimer :

- qu'un agent A est un agent B particulier sachant en plus effectuer d'autres interactions. Cette relation est similaire à la notion d'héritage trouvée dans les langages orientés-objet ;
- qu'un agent A est un agent B particulier ne sachant pas effectuer certaines interactions de B.

Cette approche est particulièrement intéressante pour concevoir des agents dont le comportement constitue une exception à un ensemble d'agents, par exemple pour spécifier qu'un individu aveugle est un individu particulier ne sachant pas lire.

5 Organisation du manuscrit

Ce manuscrit est organisé en trois parties, contenant au total huit chapitres.

Première partie Nous consacrons la première partie de ce manuscrit à l'introduction du contexte dans lequel se placent nos travaux, à la présentation de travaux connexes, et à l'identification des problématiques que nous traitons.

Le chapitre 1 justifie l'utilisation du paradigme agent et des systèmes multi-agents dans notre approche afin de faciliter la conception de simulations large échelle. Pour cela, nous identifions dans un premier temps les différents types de simulations informatiques existants, ainsi que le type de simulations informatiques que nous considérons, nommé couramment « simulations explicatives ». Dans un second temps, les problématiques inhérentes à ce type de simulations sont présentées, et nous décrivons en quoi l'usage de systèmes multi-agents contribue en partie à leur résolution.

Dans le chapitre 2, nous étudions plus précisément les différentes approches existantes pour construire une simulation multi-agents et, plus particulièrement, pour aboutir à un simulateur, *i.e.* un programme informatique permettant de réaliser des expériences. Nous dégageons de cette étude l'importance d'utiliser une approche transversale de conception, *i.e.* une approche qui :

- accompagne les concepteurs des premières étapes de la simulation à un simulateur concret ;
- permet de construire graduellement un modèle.

Nous caractérisons de plus les propriétés nécessaires à la définition de telles approches. Dans un second temps, nous étudions quels sont les apports des approches existantes vis-à-vis de des propriétés identifiées et quelles sont leurs lacunes. Nous en dégageons alors l'importance d'utiliser une approche de conception des simulations centrée sur les interactions entre les agents d'une simulation.

Deuxième partie En réponse aux divers problèmes de conception identifiés dans la partie précédente de cette thèse, nous proposons une approche transversale de conception de simulations, que nous centrons sur les interactions entre agents plutôt que sur leur comportement. Nous appelons cette approche Interaction Oriented Design of Agent simulations (IODA).

Un processus transversal est caractérisé par quatre éléments : la structure des modèles spécifiés, la structure de l'implémentation des modèles, la méthodologie permettant de spécifier un modèle et le processus permettant de passer de modèle à implémentation. Cette seconde partie de ce manuscrit, nous caractérisons ces quatre aspects de l'approche IODA, en trois chapitres (voir le résumé de la figure 3).

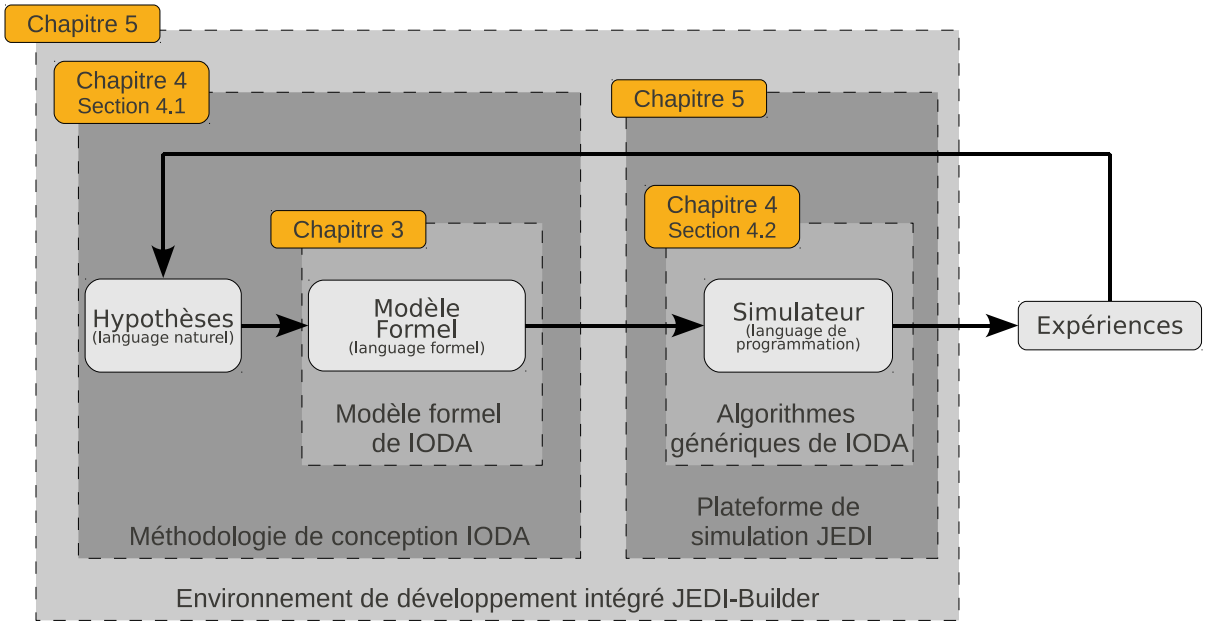


FIGURE 3 – Résumé de l'objet des chapitres 3, 4 et 5 (deuxième partie de ce manuscrit).

Le chapitre 3 détermine comment réifier la notion d'interaction dans une simulation, afin de bénéficier des avantages énoncés dans la partie précédente. Nous décrivons pour cela la structure du modèle de l'approche IODA, à l'aide du formalisme mathématique, ainsi que des méta-modèles et d'exemples.

Le chapitre 4 montre comment construire une approche transversale à l'aide du modèle IODA. Nous y définissons pour cela des outils permettant à la fois de construire graduellement un modèle (une méthodologie de conception), mais aussi de l'implémenter (des algorithmes de simulation). Nous analysons à cette occasion quatre problèmes de conception récurrents selon la perspective de IODA, et en dégageons des avantages tels que l'unification de la représentation des entités d'une simulation.

Le chapitre 5 confirme la faisabilité de l'approche IODA, en décrivant la plateforme de simulation JEDI (*Java Environment for the Design of agent Interactions*), qui implémente fidèlement les concepts de modèles IODA et en décrivant l'environnement de développement intégré JEDI-BUILDER, qui fournit une implémentation fidèle à la méthodologie IODA.

Troisième partie Le changement de perspective induit par notre approche centrée sur les interactions permet d'étudier les problèmes de simulation sous un angle nouveau. Dans cette partie, nous nous sommes appuyons sur IODA, JEDI et JEDI-BUILDER pour explorer des problématiques inhérentes à la simulation informatique, mais n'apparaissant pas explicitement dans les approches existantes, ou y étant traitées de manière non satisfaisante. Ces études ont abouti à l'identification de principes permettant d'étendre IODA et d'y intégrer des outils méthodologiques facilitant la conception de certaines catégories de simulations.

Nous avons mené l'exploration en considérant trois problématiques de simulation différentes :

1. Faut-il nécessairement des interactions complexes afin de modéliser des comportements complexes ?
2. Comment utiliser IODA afin d'éviter d'introduire des biais dans une simulation ?
3. Comment transposer le concept d'héritage des langages orientés-objets à IODA afin de faciliter l'ingénierie des connaissances ?

Puisque les actions impliquant simultanément plusieurs agents ne sont pas modélisées sous la forme d'interactions dans les approches actuelles, ces dernières éludent un problème pourtant réel, apparaissant explicitement dans notre approche centrée interaction : « Des interactions simples suffisent-elles à exprimer des comportements complexes ? » Dans le chapitre 6, nous analysons ce problème à l'aide de deux cas d'étude, ayant abouti à l'identification :

- de deux patrons de conception permettant de modéliser la plupart des interactions impliquant plus de deux agents à l'aide d'interactions en impliquant au plus deux ;
- d'un nouveau type d'interactions appelées « interactions multicast » permettant de modéliser simplement certaines interactions complexes ne pouvant être décomposées.

Afin d'éviter d'introduire des biais dans une simulation, les choix de conception doivent être faits sciemment par les concepteurs. Dans le chapitre 7, nous montrons que pour faire apparaître de tels choix, trois unités fonctionnelles sous-jacentes à toute simulation doivent être finement spécifiées. En menant des études selon cette séparation et en focalisant la conception sur les interactions, nous avons dégagé deux extensions de IODA permettant de rendre explicites les choix de conception relatifs à :

- la participation simultanée à plusieurs interactions ;
- la spécification de comportements stochastiques.

La construction de simulations large échelle ne peut être envisagée sans profiter d'outils d'ingénierie logicielle similaires à l'héritage des langages orientés-objet. En effet, ces outils permettraient d'éviter de fournir une spécification systématiquement exhaustive de la source et la cible de chaque interaction. Dans le chapitre 8, nous étudions sous quelles conditions il est possible d'intégrer des concepts proches de l'héritage dans l'approche IODA afin de faciliter l'ingénierie des connaissances. A cette occasion, nous définissons une extension de IODA fondée sur la relation de spécialisation qui facilite l'ingénierie des connaissances dans une simulation.

Ces trois chapitres identifient en tout quatre extensions possibles de l'approche IODA, qui fournissent des concepts et des outils méthodologiques permettant de traiter plus efficacement certains problèmes de conception spécifiques. L'approche IODA peut ainsi être utilisée dans sa forme « minimale » (*i.e.* sans utiliser les extensions) afin de spécifier des simulations, ou dans une de ses formes étendues, en utilisant une ou plusieurs extensions que nous décrivons. Nous définissons ainsi une approche de conception paramétrable, dont la complexité peut être adaptée à la complexité des problèmes traités.

Première partie

Contexte et État de l'art

Plan de la partie :

Nous consacrons cette partie à l'introduction du contexte dans lequel se placent nos travaux, à la présentation de travaux connexes et à la justification du besoin d'une approche de conception de simulations centrée sur les interactions.

Le chapitre 1 justifie notre proposition d'une approche multi-agents de la conception de simulations dites explicatives. Pour cela, nous caractérisons ce qu'est une « simulations explicatives » et quelles problématiques de conception sont posées par ces simulations. Nous illustrons alors comment les systèmes multi-agents facilitent leur résolution.

Dans le chapitre 2, nous identifions les limites des approches existantes, relativement aux problèmes de conception de simulations pouvant contenir un grand nombre d'agents interagissant de manière variée (simulations large échelle). Nous identifions alors sous quelles conditions un meilleur compromis peut être trouvé. Ces conditions nous servent de principes fondateurs à l'approche IODA *(Interaction Oriented Design of Agent simulations) que nous développons dans ce manuscrit.*

Chapitre 1

Contexte : Simulation multi-agents

Plan du chapitre :

Dans cette thèse, nous cherchons à faciliter la conception de simulations informatiques explicatives, et plus précisément des simulations large échelle, faisant intervenir un grand nombre d'entités entretenant des interactions variées. Pour cela, nous définissons une approche multi-agents centrée sur les interactions.

Ce chapitre justifie notre proposition d'une approche multi-agents de la conception de simulations dites explicatives. Pour cela, nous décrivons l'objet de nos travaux, à savoir la simulation informatique (section 1.1), nous en identifions les principes, ainsi que les principales problématiques (section 1.2). Nous décrivons ensuite les concepts et les propriétés principales du paradigme agent et des systèmes multi-agents (section 1.3) et montrons en quoi ils contribuent à la résolution des problématiques identifiées (section 1.4).

1.1 Contexte général : Simulation informatique de systèmes complexes

L'utilisation de simulations multi-agents (MABS[1]) est un phénomène de plus en plus répandu. Son essor est lié à sa représentation individu-centrée du phénomène, qui contribue grandement à la résolution de problématiques inhérentes à la simulation informatique. Cette première section identifie ces problématiques, en s'appuyant sur les définitions existantes de la simulation. Nous considérons en particulier la définition de Shannon [Sha98] qui suit :

> « [Simulation is] the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and /or evaluating various strategies for the operation of the system. » [Sha98]

Cette définition fait apparaître quatre grand thèmes : *les objectifs* (*i.e.* « the purpose »), *le modèle*, *les expériences* et *le processus* (voir figure 1.1). Nous traitons ces quatre thèmes séparément pour caractériser le type de simulation informatique étudié dans cette thèse.

Simulation = Objectifs + Modèle + Processus + Expériences

FIGURE 1.1 – Schématisation des principales notions trouvées dans la définition de la simulation proposée par Shannon [Sha98].

1. pour : *MultiAgent Based Simulation*. Nous n'utilisons pas l'acronyme français SMA pour qu'il ne soit pas confondu avec *Système Multi-Agents*.

1.1.1 Terminologie utilisée

La simulation est un outil au service d'objectifs variés et au service de domaines d'application très différents (voir section 1.1.2). Cette variété fait qu'aucun consensus de terminologie n'existe sur le sujet [Öre87, Fis94]. Par conséquent, ce document de thèse s'attache aux *concepts* plus qu'aux termes en eux-mêmes. Nous invitons le lecteur à se remémorer ceci, à chaque fois que l'utilisation de certains termes dans certains contextes lui semble choquante.

1.1.2 Simulation informatique : Quels objectifs ?

L'unique point commun de toute simulation est la volonté de reproduire artificiellement un phénomène (qu'il soit réel ou non) à l'aide d'un programme informatique. Par la suite, nous nous référons au phénomène imité par simulation [2] par le terme « phénomène ».

Selon [Axe97, Edm05], les usages de la simulation informatique diffèrent par les objectifs qu'ils cherchent à atteindre. Ces usages incluent [3] :

1. « la prédiction » (« prediction ») : la simulation prend en paramètres des données issues d'un phénomène et produit en sortie une estimation de l'état du phénomène dans le futur. Les applications de ce type de simulation incluent la prédiction de l'évolution des populations d'un écosystème [Vol28], la prédiction de l'évolution des conditions atmosphériques, la prédiction des performances d'un processeur, *etc.*
2. « la substitution » (« performance ») : la simulation imite le comportement d'un humain pour reproduire une tâche. Ces tâches incluent le diagnostic médical automatisé, la reconnaissance vocale, *etc.* De telles simulations sont aussi utilisées pour éprouver (*i.e.* tester, vérifier ou valider) un système informatique, en reproduisant artificiellement le comportement d'un utilisateur humain ;
3. « l'entraînement » (« training ») : la simulation est utilisée pour immerger un utilisateur dans un environnement artificiel, dynamique et interactif, à des fins d'entraînement. Les applications de ce type incluent l'apprentissage du pilotage d'un avion, l'apprentissage de la conduite, l'apprentissage de la chirurgie, *etc.*
4. « le divertissement » (« entertainment ») : la simulation est utilisée pour immerger un ou plusieurs utilisateurs dans un environnement virtuel à des fins ludiques. Les applications de telles simulations incluent les effets spéciaux dans les films ou les jeux vidéos, l'intelligence artificielle de personnages dans un jeu vidéo, *etc.*
5. « l'éducation » (« éducation ») : la simulation est utilisée comme moyen pédagogique pour mettre en pratique des notions théoriques. Par exemple Clim'way [Sci10] propose une simulation de type SimCity pour sensibiliser à l'impact écologique de l'urbanisme, *etc.*
6. « la preuve » (« proof ») : la simulation fournit la preuve de l'existence d'un cas se conformant à certaines hypothèses. Par exemple, le jeu de la vie [PW84] prouve que des comportements excessivement complexes peuvent être le résultat de règles très simples ;
7. « la découverte » (« discovery ») : la simulation a un objectif similaire à *la preuve* mais, contrairement à cette dernière, les résultats obtenus ne constituent pas une preuve. Ils permettent seulement de conforter ou d'affaiblir les hypothèses émises. Par exemple, le modèle de ségreggation de Schelling [Sch71] conforte l'hypothèse selon laquelle des communautés de personnes peuvent se former dans une ville, même si toutes ces personnes déménagent en se basant sur un critère pouvant sembler tolérant. En effet, avec un critère « déménager uniquement si moins de 30% des voisins sont de mon ethnie » [4], la simulation se termine dans une situation où chaque personne a en moyenne plus de 70% de voisins de la même ethnie qu'elle.

En tant que méthode scientifique, la simulation est principalement utilisée pour la *prédiction*, la *preuve* et la *découverte*.

2. aussi appelé « source system » [ZKP00], « system » [Axe97], « real system » [Sha98], « real or proposed system » [Rob06], « actual or theoritical physical system » [Fis94], « target system or phenomena » [Edm05], ...

3. les exemples proposés ne figurent pas tous dans [Axe97, Edm05]

4. Le critère de l'ethnie est un exemple. La simulation décrite par Schelling reste valide pour tout autre type de dichotomie : pauvre/riche, français/anglais, *etc.*

Dans cette thèse, nous nous intéressons principalement aux simulations utilisées pour la *découverte*. Nous les appelons par la suite *simulations explicatives* (d'après l'appellation « explanatory » d'Edmonds [Edm05]).

Avant de passer à la description de l'élément suivant composant les simulations (le modèle), il nous semble important de caractériser plus précisément la différence entre entre simulations utilisées pour la preuve, pour la prédiction et pour la découverte. En effet, bien qu'elles soient utilisées comme méthodes scientifiques, ces simulations sont conçues et utilisées de manière radicalement différentes [Edm05].

La prédiction

La simulation prédictive est utilisée comme alternative à la résolution de modèles mathématiques, dans le cas où il est analytiquement difficile, voire impossible de les résoudre :

> « A computer simulation is any computer-implemented method for exploring the properties of mathematical models where analytical methods are unavailable » [Hum91] (d'après [AS07])

Dans de tels cas, la simulation permet de faire évoluer les différents paramètres de la simulation progressivement au fil du temps. Une estimation des résultats du modèle est ainsi obtenue sans pour autant le résoudre analytiquement.

De telles simulations ne peuvent être effectuées que si un modèle mathématique peut être formulé. Elles sont donc principalement utilisées dans les cas où les phénomènes étudiés sont pleinement (ou majoritairement) compris, par exemple en sciences physique [AS07].

La découverte

La découverte diffère de la simulation prédictive principalement par l'absence, ou la carence en théories précises permettant de décrire le phénomène. Ses domaines d'application incluent la biologie, ou les sciences sociales. Ces simulations sont utilisées pour reproduire le comportement du phénomène si jamais il fonctionnait selon les hypothèses émises. Si les résultats de simulation sont similaires aux propriétés observées du phénomène, alors les hypothèses émises offrent une explication candidate du mécanisme à l'origine du phénomène. En effet, la simulation ne permet pas de prouver que le phénomène fonctionne réellement tel qu'énoncé. Elle permet seulement de renforcer ou, si les résultats ne sont pas concluants, d'affaiblir les hypothèses émises.

Sauf mention contraire, dans la suite de ce document le terme générique « simulation » est utilisé dans son sens « simulation explicative ».

1.1.3 Simulation informatique : Le rôle du modèle

La diversité de ses applications fait qu'il n'existe aucune définition précise et communément acceptée de la simulation informatique, ni aucune description de la façon dont la simulation est conduite. Dans le cas de son utilisation comme méthode scientifique et en particulier dans son utilisation pour la découverte, certaines définitions permettent toutefois d'en avoir une idée générale.

Les définitions de Shannon [Sha98] citée en section 1.1, de Zeigler [ZKP00] (voir figure1.2) et de Fishwick [Fis94] caractérisent la simulation comme suit : une simulation a pour but d'imiter *un phénomène* dans un certain *cadre expérimental* (*i.e.* un objectif que la simulation cherche à atteindre), en se basant sur une représentation formelle de la façon dont le phénomène est supposé fonctionner, appelée *modèle du phénomène*, qui est traduit par la suite en *simulateur* permettant d'effectuer des expériences *in sillico*.

Le modèle est central à toute simulation : il sert de pivot entre les experts du domaine (*i.e.* les biologistes, les économistes, les sociologues, *etc.*) qui disposent des connaissances permettant d'abstraire le phénomène et les experts en informatique qui disposent des connaissances permettant de construire le simulateur. Il décrit de plus toutes les hypothèses sous lesquelles sont conduites les expériences. Il est donc indispensable pour communiquer et interpréter les résultats expérimentaux obtenus.

Un modèle constitue une représentation abstraite *particulière* d'un phénomène : un phénomène peut être abstrait en modèles très différents. La construction d'un modèle nécessite donc de faire des choix, qui ont des conséquences sur les résultats expérimentaux obtenus. En effet, un modèle peut reproduire

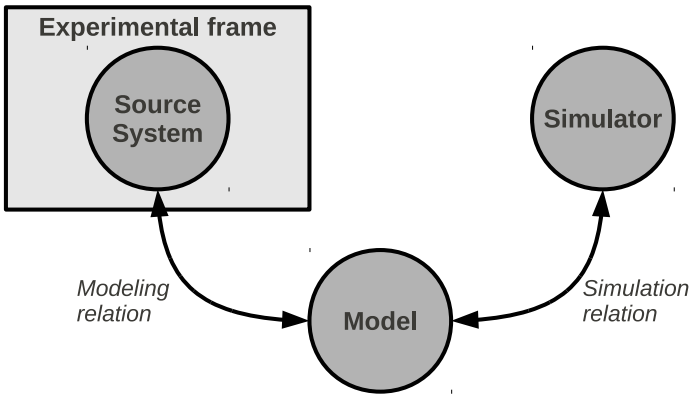


FIGURE 1.2 – Entités fondamentales constituant la simulation informatique et relations qu'elles entretiennent selon Zeigler [ZKP00]

correctement certains aspects d'un phénomène sans pour autant répondre aux hypothèses émises lors de la description du cadre expérimental. Par exemple, le modèle de Lotka-Volterra [J.L25, Vol28] permet de prédire l'évolution des populations de proies et prédateurs d'un écosystème, mais ne permet pas de connaître l'espérance de vie des prédateurs. Un modèle peut aussi répondre correctement aux hypothèses émises, en faisant toutefois défaut au principe de parcimonie.

Le principe de parcimonie

L'objectif d'une simulation n'est pas simplement de découvrir les causes du phénomène étudié, mais aussi de trouver le modèle qui l'explique le mieux. Ce principe, appelé principe de parcimonie, veut que, pour avoir des résultats concluants, il est nécessaire d'identifier les hypothèses minimales (et donc le modèle minimal) permettant de reproduire le phénomène.

La pratique du principe de parcimonie a pour conséquence une ou plusieurs itérations lors de la conception d'une simulation : un modèle est conçu, testé, puis simplifié jusqu'à obtenir un modèle minimal ou inversement, conçu le plus simple possible puis augmenté.

1.1.4 Le processus de simulation

Bien que la grande majorité des définitions de la simulation s'accordent sur le fait que leur conception fait partie d'un processus particulier, aucune définition détaillant ce processus ne fait consensus [And74, GT05, LM91, Sha98]. L'étude de différentes publications traitant du sujet permet toutefois d'en identifier les principales étapes. Nous étudions ici deux de ces définitions, qui décrivent le processus de simulation selon deux perspectives différentes : la réalisation d'expériences et la conception du programme de simulation.

Puisque le terme « simulation » peut être interprété comme « programme informatique permettant d'exécuter le modèle » (*i.e.* simulateur), nous mentionnons par la suite explicitement « processus de simulation » pour éviter toute ambiguïté

Un processus expérimental

Shannon [Sha98] décrit le processus de simulation informatique comme un processus expérimental composé des 12 étapes qui suivent (voir figure 1.3) :

1. « Problem definition » : Définir précisément l'objectif de la simulation, *i.e.* pourquoi étudie-t-on ce phénomène et à quelles questions souhaite-t-on répondre ;
2. « Project planning » : Déterminer si les moyens techniques et logistiques à disposition permettent d'entreprendre la résolution d'un tel problème ;
3. « System definition » : Déterminer les informations pertinentes à prendre en compte dans le modèle ;

4. « Conceptual model formulation » : Développer un modèle préliminaire de la simulation décrivant les composants [5], les variables ainsi que les interactions constituant le système simulé de manière graphique ou dans un pseudo-code ;
5. « Preliminary Experimental Design » : Déterminer sur quels critères la qualité de la simulation est évaluée, quels paramètres faire varier, comment les faire varier et combien d'expériences il est nécessaire d'exécuter ;
6. « Input Data preparation » : Identifier et sélectionner les valeurs pertinentes utilisées initialement pour chaque paramètre de la simulation ;
7. « Model Translation » : Implémenter le modèle dans un langage de simulation particulier ;
8. « Verification and Validation » : Confirmer que l'implémentation est conforme au modèle (vérification) et que les résultats obtenus par une simulation sont qualitativement et quantitativement conformes à ce qui est attendu ;
9. « Final experimental design » : Affiner les critères décrivant la qualité d'une simulation, ainsi que les jeux de paramètres utilisés et le nombre d'expériences à exécuter ;
10. « Experimentation » : Exécuter l'ensemble des expériences prévues et collecter les résultats obtenus ;
11. « Analysis and Interpretation » : Analyser les résultats obtenus et déterminer si la solution proposée répond au problème initialement posé ;
12. « Implementation and Documentation » : Documenter la simulation effectuée et publier les résultats obtenus.

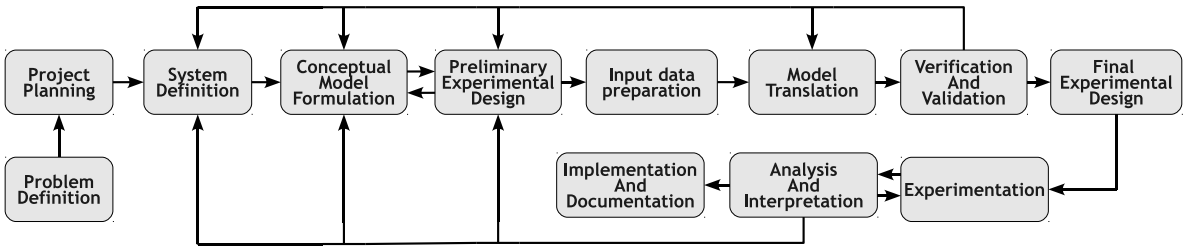


FIGURE 1.3 – Schéma du processus de conception de simulations défini par Shannon dans [Sha76] adapté à sa proposition dans [Sha98].

Cette définition est fortement inspirée des protocoles expérimentaux : seules les étapes de « Project planning », « Model Translation » et « Verification » (mais pas « Validation ») sont propres aux simulations informatiques.

> Puisque nous nous intéressons uniquement aux problématiques liées à la conception de simulations informatiques, ce travail de thèse étudie un sous-ensemble du processus de simulation, s'arrêtant à l'étape de « Verification ». Nous appelons ce sous-ensemble le « processus de conception de simulations ».

Notre volonté n'est pas ici de minimiser les difficultés liées aux autres étapes, mais de restreindre le cadre de l'étude menée dans cette thèse. Ces autres étapes posent également des problèmes non triviaux, tels que la reproduction d'une simulation (aussi appelé problème de divergence implémentatoire par Michel [Mic04]), le problème de la diffusion des résultats de simulation, *etc.*

Un processus de conception

D'autres définitions se focalisent sur une vision orientée « experts en informatique » du processus de simulation [ZKP00, MEU04, Edm05, GII+09] . Ces définitions, en particulier celle de Galan [GII+09]

5. « component » dans le texte. L'auteur utilise ce terme en son sens général. Il ne fait pas référence au paradigme de conception par composants.

(voir figure 1.4), sont particulièrement intéressantes, car le changement de perspective permet de mettre en valeur l'un des problèmes fondamentaux de la simulation explicative. Dans la définition de Galan, le processus de simulation ne repose plus sur un seul, mais sur plusieurs modèles :

1. « Modèle non-formel » : formulé à l'aide du langage naturel, il est écrit par des experts du domaine afin de définir l'objectif de la simulation (décrits dans la section 1.1.2), le but de la simulation (par exemple « reproduire les variations des populations d'un écosystème »), les principaux éléments du phénomène, leurs connexions, ainsi que les principales relations causales ;
2. « Modèle formel » (aussi appelé « modèle conceptuel » [Rob06, Sar98]) : Ce modèle correspond à la réécriture du « modèle non-formel » en utilisant un formalisme particulier. Lors de sa conception, les choix effectués par les experts du domaine dans le « modèle non-formel » sont complétés pour qu'ils puissent être exprimés dans le formalisme choisi ;
3. « Modèle exécutable » (aussi appelé « modèle computationel » [EH03] ou « modèle de la simulation » [Sar98]) : Ce modèle complète le « modèle formel », en lui ajoutant tous les éléments qui permettront son implémentation ;
4. « Programme informatique » : Comme son nom l'indique ce dernier modèle est une implémentation du « modèle exécutable » s'appuyant sur un langage de programmation ou une plateforme de simulation particulière.

Chaque modèle est décrit en fonction des informations extraites du modèle précédent et se rapproche progressivement du programme informatique. Ils permettent donc de passer graduellement d'une représentation abstraite du phénomène réel à son implémentation. Selon Galan, ces modèles peuvent être en pratique disjoints, ou décrire différentes parties d'un même modèle général. De plus, il est intéressant de noter qu'il décrit aussi différents rôles joués par le concepteur durant la construction de la simulation. Chaque rôle ne peut être joué qu'avec des compétences adéquates, variant grandement d'un rôle à un autre.

Dans la section qui suit, nous nous appuyons sur les définitions de Shannon et Galan afin d'identifier les problématiques liées à la conception de simulations.

1.2 Principales problématiques liées au processus de conception de simulations

Les deux définitions présentées dans la section précédente permettent de caractériser les problématiques fondamentales (voir tableau 1.1(a)) rencontrées en simulation informatique en deux thèmes principaux : *l'obtention de résultats erronés* et *la révision du modèle*.

Afin de clarifier nos propos, nous nommons $\mathcal{P}prior_i$ les problèmes de modélisation et d'implémentation menant à des résultats erronés, $\mathcal{P}post_i$ les problèmes liés à la détection de résultats erronés et $\mathcal{P}rev_i$ les problèmes liés aux révisions du modèle.

1.2.1 Obtention de résultats erronés

En sciences expérimentales, il est commun d'obtenir des résultats différents de ceux attendus. Ces résultats signifient que le phénomène réel fonctionne différemment des hypothèses émises par l'expert du domaine.

En simulation informatique, cela n'est pas nécessairement le cas. L'implémentation d'une simulation peut différer de ce qui était imaginé par l'expert du domaine. En effet, comme l'illustre la figure 1.4, l'implémentation d'une simulation informatique est le fruit d'un ensemble non négligeable d'étapes, pouvant faire intervenir des personnes aux compétences très différentes. Chaque étape est sujette à différents types d'erreurs [GII+09], liées à la traduction d'un modèle source en un modèle cible (par exemple transformer le modèle formel en modèle exécutable) :

- le modèle cible peut refléter quelque chose de différent de ce que pense son créateur (le formalisme du modèle cible est mal utilisé) ; ($\mathcal{P}prior_1$)
- une ambiguïté des informations fournies par le modèle source a abouti à un mauvais choix dans le modèle cible ; ($\mathcal{P}prior_2$)

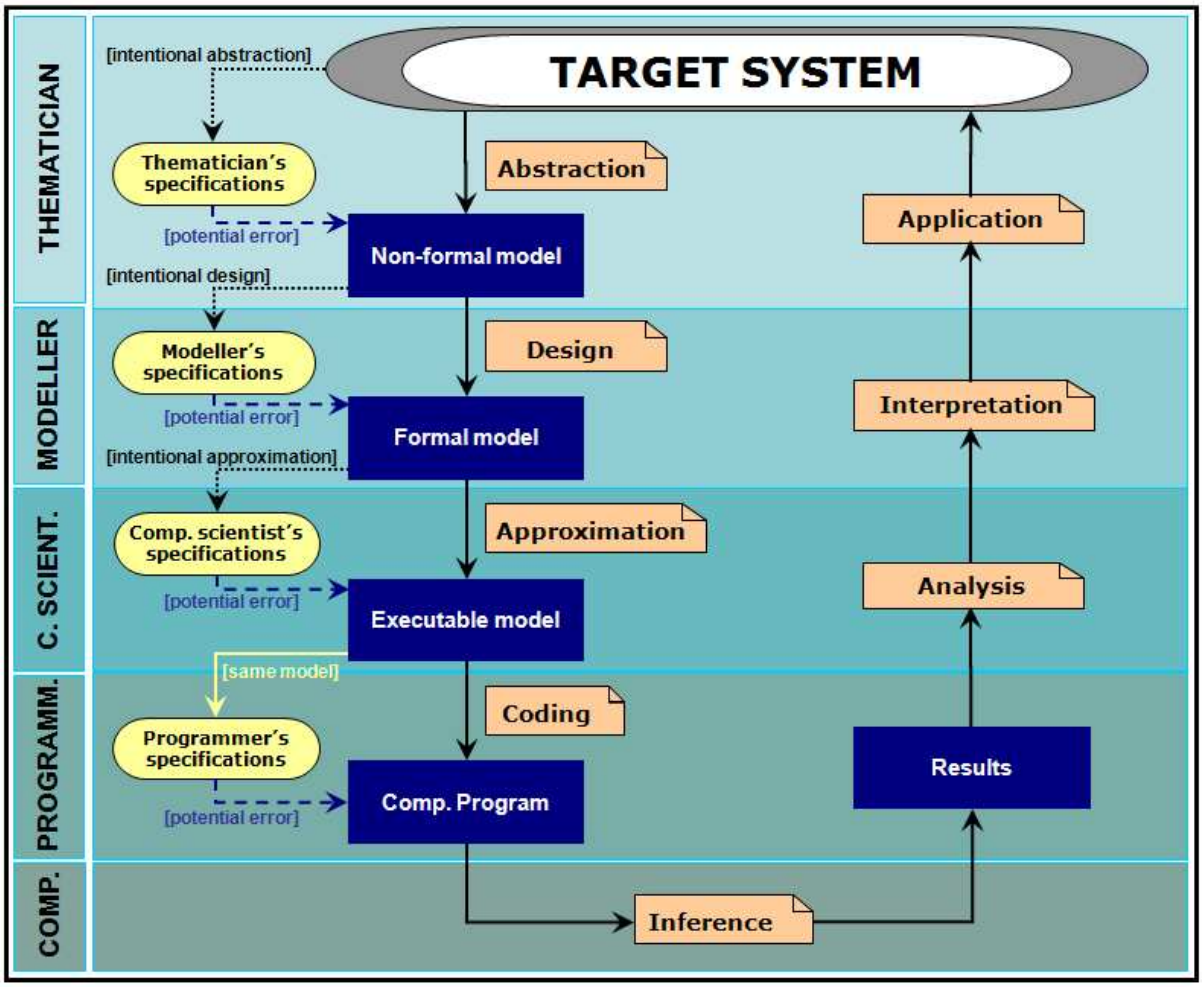


FIGURE 1.4 – Les différentes étapes du processus de conception de simulations, selon Galan [GII+09].

– l'absence d'une information dans le modèle source a abouti à un choix arbitraire dans le modèle cible ; ($\mathcal{P}prior_3$)
– indépendamment des informations fournies par le modèle source, un choix involontaire et implicite a été fait dans le modèle cible. ($\mathcal{P}prior_4$)

Si à un moment ou un autre, une personne concevant la simulation se retrouve dans un de ces cas de figure, le modèle (qu'il soit non-formel, formel, exécutable, ou le simulateur) et par extension la simulation deviennent biaisés. Les conséquences d'un modèle biaisé sont diverses et dépendent de la nature du biais (voir le chapitre 7 pour une description plus détaillée de ce problème). Dans le meilleur des cas, les biais ne concernent que des aspect mineurs de la simulation et n'en altèrent pas les résultats ($\mathcal{P}post_1$). Les biais peuvent aussi aboutir à des résultats de simulation erronés ($\mathcal{P}post_2$). Ces résultats inattendus vont occasionner une nouvelle itération du processus de conception de simulations, qui va donner l'opportunité de détecter l'erreur et de corriger le modèle. Dans le pire des cas, une situation critique survient : le modèle ne décrit pas correctement le phénomène réel et devrait donc produire des résultats erronés, mais son implémentation biaisée compense ce phénomène et donne l'illusion d'avoir des résultats concluants ($\mathcal{P}post_3$).

Il est donc primordial de s'assurer que les choix effectués lors de la constitution des modèles et de l'implémentation sont conformes à l'abstraction du phénomène que l'expert du domaine souhaite obtenir.

1.2.2 Révisions du modèle

Comme le montre sa représentation schématique sur la figure 1.3, le processus de simulation n'est pas strictement séquentiel. Des itérations surviennent lors de trois étapes : l'étape intitulée « Preliminary Experimental Design », l'étape intitulée « Verification and Validation » et l'étape intitulée « Analysis and Interpretation ».

L'étape intitulée « Preliminary Experimental Design » a pour but de raffiner le modèle afin de prendre en compte les différents paramètres utilisés dans les expériences. Puisque l'étape d'implémentation (intitulée « Model Translation ») n'est pas encore atteinte, la prise en compte de cette itération se fait uniquement au niveau modèle. Ce dernier doit être suffisamment souple pour prendre en compte les changements sans avoir à modifier profondément sa structure. Dans le cas contraire, un changement risque de causer l'un des problèmes $\mathcal{P}prior_1$, $\mathcal{P}prior_2$, $\mathcal{P}prior_3$ ou $\mathcal{P}prior_4$. Ce problème est particulièrement vrai dans des modèles équationnels, où certains changements peuvent nécessiter une mise en équation complète du système. ($\mathcal{P}rev_1$)

L'étape intitulée « Verification and Validation » a pour but d'identifier les situations $\mathcal{P}post_1$, $\mathcal{P}post_2$ et $\mathcal{P}post_3$. Si une de ces situation est détectée, une révision du modèle s'impose, *i.e.* une modification du modèle permettant de corriger les erreurs identifiées. La révision n'est possible que s'il y a identification préalable de ce qui a causé l'erreur détectée et donc détermination de quels éléments du modèle (niveau microscopique) sont à l'origine des résultats erronés de simulation (niveau macroscopique). ($\mathcal{P}rev_2$)

L'étape intitulée « Analysis and Interpretation » permet de déterminer si la simulation est considérée comme utile (ayant atteint ses buts et ses objectifs) ou pas. Dans le cas où elle ne le serait pas, il faudrait alors explorer une autre piste, et donc produire un modèle différent du phénomène. Cette étape est propre aux protocoles expérimentaux et n'introduit de problématique spécifique à la simulation, mis à part celle découlant de la révision de modèle, que nous décrivons ci-après.

Toute révision du modèle peut s'avérer coûteuse en termes d'implémentation. Bien que les modèles diffèrent d'une itération à l'autre, ils ont, dans une certaine mesure, une base commune puisqu'ils représentent le même phénomène. Il serait donc possible de réutiliser certains éléments de l'implémentation, afin de réduire les coût induits par le développement de la simulation. ($\mathcal{P}rev_3$)

1.2.3 Comment les résoudre ?

Différentes approches sont utilisées pour résoudre les problèmes mentionnés dans la section précédente. Nous n'en fournissons ici qu'un aperçu, qui nous guidera par la suite dans l'état de l'art.

L'obtention de résultats inattendus est gérée par deux types d'approches : les approches *a priori* et les approches *a posteriori*. Le principe de base des approches *a priori* est d'éviter, à l'aide de divers moyens, les situations pouvant aboutir aux problèmes $\mathcal{P}prior_1$, $\mathcal{P}prior_2$, $\mathcal{P}prior_3$ et $\mathcal{P}prior_4$. Ces solutions se

focalisent donc sur la structure du(des) modèle(s), la façon dont est représentée la connaissance, ainsi que la façon dont les modèles sont construits. Elles incluent :

- favoriser l'intervention des experts du domaine dans le processus de conception de simulations, pour éviter ces problèmes lors de la transition entre abstraction du phénomène et modèle formel ;
- favoriser l'identification des problèmes et choix relatifs à l'implémentation, lors de la transition entre modèle formel et modèle exécutable ;
- fournir des représentations visuelles favorisant la communication entre les différents acteurs du processus de conception de simulations ;
- favoriser la relation de morphisme [ZKP00] entre les différents modèles, *i.e.* une relation d'équivalence qui met en correspondance les éléments de deux spécifications différentes ;
- aider le choix d'un formalisme de modélisation et d'une plateforme d'implémentation à l'aide de critères de comparaison.

A l'opposé, les approches *a posteriori* ont pour principe de fournir des outils permettant d'identifier les situations où la simulation n'est pas valide ($\mathcal{P}post_1$, $\mathcal{P}post_2$ et $\mathcal{P}post_3$), qui sont le symptôme des problèmes $\mathcal{P}prior_1$, $\mathcal{P}prior_2$, $\mathcal{P}prior_3$ et $\mathcal{P}prior_4$. Ces solutions se focalisent sur l'analyse du modèle, de l'implémentation, et des résultats expérimentaux. Elles incluent :

- favoriser le test unitaire de l'implémentation ;
- comparer les résultats expérimentaux à des résultats extraits du phénomène, obtenus d'une simulation prédictive du même phénomène, ou par une résolution analytique d'un modèle mathématique équivalent ;
- comparer les résultats expérimentaux obtenus de modèles et de simulateurs différents ;
- analyser la sémantique du modèle.

Le problème de la révision de modèle est lui géré par des approches favorisant le génie logiciel lors de la conception de la simulation. Ces solutions se focalisent sur la structure et l'architecture de l'implémentation. Elles cherchent en particulier à :

- rendre l'architecture de la simulation modulaire ;
- construire des librairies d'éléments réutilisables ;
- favoriser le morphisme entre modèle et implémentation, afin d'identifier plus facilement la part du modèle à l'origine d'une erreur identifiée dans l'implémentation.

Dans ce document, nous nous intéressons en particulier aux solutions *a priori*, ainsi qu'aux solutions liées aux révisions de modèle. Pour traiter ces problématiques, nous nous focalisons sur la représentation des connaissances dans le modèle, ainsi que sur le processus de conception.

Le paradigme « multi-agents » fournit une représentation des connaissances qui contribue grandement à l'ensemble des problèmes mentionnés dans cette section. La section qui suit décrit ces différents apports, après avoir présenté les principaux concepts des systèmes multi-agents.

1.3 La conception orientée-agent

La simulation multi-agents s'appuie sur le paradigme des systèmes multi-agents. Nous commençons donc naturellement par la présentation des différents concepts liés à ce paradigme.

1.3.1 Les agents

L'étendue des domaines d'application des systèmes multi-agents fait qu'aucune définition de la notion « agent » n'est communément acceptée. En effet, selon les domaines, certaines caractéristiques des agents sont contestées [Woo01].

Selon Wooldridge :

> « An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives. » [Woo01]

Notre interprétation de la définition précédente, conduite en la confrontant avec les définitions trouvées dans [Fer99, WJ95], nous amène à considérer qu'un agent est un système informatique qui est :

- « capable d'agir dans son environnement » : un agent peut modifier son environnement au travers des actions qu'ils effectue. Environnement est à comprendre en son sens systémique, c'est à dire « tout ce qui est extérieur à l'agent lui-même » ;
- « situé dans un environnement » : un agent n'est pas omniscient. Il n'agit qu'en fonction de la partie de son environnement qu'il *perçoit* et ne peut agir que sur un sous-ensemble de son environnement ;
- « autonome » : le comportement d'un agent (*i.e.* les actions qu'il choisit d'entreprendre) n'est pas le fruit d'un programme tiers : l'agent choisit lui même comment il agit.

Cette interprétation se retrouve dans [Mic04], qui la schématise d'ailleurs sous une forme très proche de la systèmique (voir figure 1.5). Comme l'indique cette figure, un agent dispose d'une architecture interne

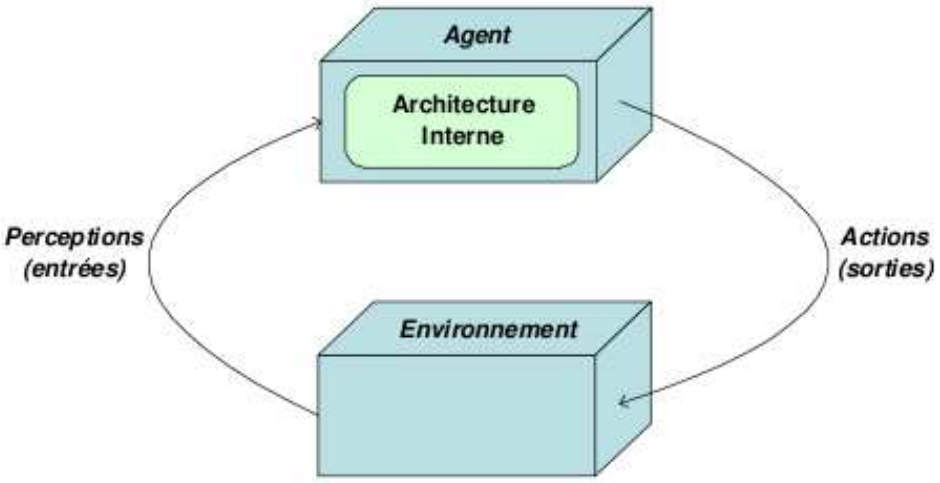


FIGURE 1.5 – Représentation classique d'un agent et de son environnement (d'après [Mic04]).

permettant de décider quelle action effectuer, en fonction des éléments qu'il perçoit. Nous revenons sur cette notion d'architecture interne des agents dans le chapitre 2.

D'autres propriétés peuvent compléter cette définition, mais sont contestées selon les domaines d'application. Parmi les plus communes figurent :

- « la possession d'un état interne » ([Woo01]) : un agent peut avoir un état interne, qui conditionnera ses actions conjointement à ce qu'il perçoit de l'environnement ;
- « la proactivité » ([WJ95]) : un agent n'agit pas uniquement en réaction à ce qu'il perçoit. Il agit aussi en fonction de buts qui lui sont propres ;
- « la communication » ([WJ95]) : un agent peut interagir avec d'autres agents à l'aide d'un langage de communication (un ACL, *i.e.* Agent Communication Language) ;
- « les croyances » ([RG91]) : un agent n'agit pas directement en fonction de ce qu'il perçoit dans l'environnement, mais en fonction de ce qu'il croit savoir de l'environnement ;

Ces différences sont le fruit de l'hétérogénéité des domaines d'application des systèmes multi-agents. En effet, leur utilisation pour concevoir des applications réparties nécessite une structure et une architecture différente de celle utilisée pour la résolution distribuée de problèmes, ou de celle utilisée pour la simulation. Nous détaillons ces problèmes dans le chapitre 2.

1.3.2 Les systèmes multi-agents

Les agents font partie d'un système multi-agents (que l'on notera par la suite MAS [6]), dont la définition est tout aussi contestée. À notre connaissance, toutes les définitions ne s'accordent que sur le fait qu'un système multi-agents ne se résume pas à un système ne contenant qu'un ensemble d'agents.

La définition proposée par Demazeau dans son approche Voyelles et sa plateforme Volcano [Dem95] est particulièrement intéressante. En effet, non seulement elle caractérise (sans faire consensus) ce qu'est

6. Cet acronyme est issu du nom anglais « MultiAgent System »

un système multi-agents, comment ils sont conçus, mais illustre aussi parfaitement le potentiel de ce paradigme du point de vue génie logiciel.

Selon Volcano, un système multi-agents est composé d'un ensemble d'*agents* potentiellement *organisés* qui *interagissent* dans un *environnement* commun. Cette approche repose sur trois caractérisations élémentaires d'un MAS, ainsi que sur une quatrième caractérisation optionnelle :

1. Agents : comment les agents décident-ils des actions qu'ils effectuent ;
2. Environnement : dans quel milieu évoluent les agents ;
3. Interaction : quels sont les moyens utilisés par les agents pour influer sur les actions effectuées par les autres agents ;
4. Organisation (optionnel) : comment structurer en ensembles les agents du système.

La conception du MAS se fait selon un processus reposant l'abstraction du système en un (ou plusieurs) modèle(s), qui sont ensuite implémentés sur une plateforme particulière. Chaque caractérisation est spécifiée dans une « brique » indépendamment des autres, autant dans le modèle qu'à l'implémentation (voir figure 1.6). Ainsi, le système multi-agents est conçu à l'aide de modèles multiples et indépendants permettant de réutiliser et modifier aisément divers éléments logiciels.

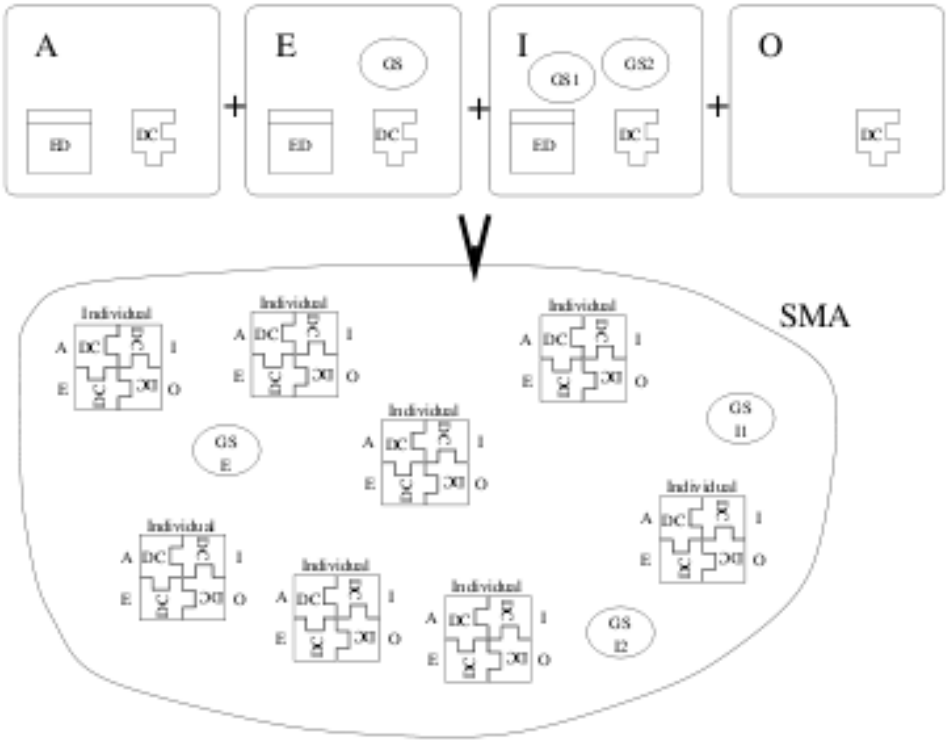


FIGURE 1.6 – La construction d'un MAS avec l'approche Volcano.

Cette définition est toutefois incomplète, car elle postule que toutes les entités contenues dans l'environnement sont des agents, *i.e.* des entités au comportement autonome. Pourtant, d'autres types d'entités peuvent exister dans l'environnement sans pour autant avoir un comportement autonome. Par exemple, une base de données dans une application répartie, un objet physique tel qu'une chaise dans un environnement virtuel, ou un caddie utilisé par des clients dans une simulation. En complément à la définition fournie par Demazeau, nous considérons donc un sous-ensemble de la définition de Ferber [Fer99] qui offre une caractérisation de ces entités. Selon lui, un système multi-agents est caractérisé entre autres par :

– Un environnement E ;
– Un ensemble d'objets [7] O. Ces objets sont situés, c'est-à-dire que tout objet a une position dans E.

7. Ce terme est utilisé en son sens général et indépendant de celui trouvé en programmation orientée-objet.

Ces objets peuvent être perçus, créés, détruits et modifiés par les agents ;
- Un ensemble A d'agents, qui sont des objets particuliers ($A \subseteq O$), lesquels représentent les entités actives du système ;
- Un ensemble de relations R qui unissent des objets (et donc des agents) entre eux. Cette notion est utilisée pour définir les accointances d'un agent, c'est à dire les autres objets de l'environnement qu'il peut percevoir.

Ce complément permet d'identifier les propriétés des entités n'ayant pas de comportement autonome dans la simulation, et en quoi elles participent au comportement des agents.

1.4 La simulation multi-agents

Plusieurs raisons font que les systèmes multi-agents sont devenus très populaires en tant qu'outil de conception de simulations.

Les agents constituent une abstraction anthropomorphique similaire à la notion d'entité (par exemple une molécule, un animal, un humain) dans un phénomène. Ainsi, la description de la dynamique du phénomène ne se fait pas en déclarant des équations portant sur des propriétés macroscopiques de la simulation (par exemple des équations exprimant l'évolution de la population d'une espèce animale dans [Vol28]), mais sur le comportement individuel de chaque entité. Une telle métaphore est proche de la description naturelle d'un phénomène et est donc simple à utiliser par les experts du domaine. Ils effectuent donc des choix moins sujets aux problèmes $\mathcal{P}prior_1$, $\mathcal{P}prior_2$, $\mathcal{P}prior_3$ et $\mathcal{P}prior_4$ lors de la constitution du modèle formel à partir du modèle non-formel. Comme les agents sont aussi des entités logicielles, cette abstraction est conservée dans le modèle exécutable et dans l'implémentation, réduisant d'autant les chances d'obtenir des résultats erronés ($\mathcal{P}post_1$, $\mathcal{P}post_2$ et $\mathcal{P}post_3$).

De plus, la conservation d'une partie de la structure du modèle à l'implémentation favorise l'identification de l'origine de résultats erronés, en particulier lors de la vérification de l'implémentation. En effet, une fois l'erreur repérée dans l'implémentation d'un agent, son interprétation dans le modèle se fait aussi dans le modèle de l'agent ($\mathcal{P}prev_2$).

Les agents sont aussi un paradigme de programmation, proposant un ensemble d'outils logiciels favorisant la réutilisation [Jen99]. Les révisions du modèle sont ainsi rendues plus simples, car une grande partie de l'implémentation peut être réutilisée ($\mathcal{P}prev_3$). De plus, la division de la simulation en différentes parties facilite les tests unitaires, et facilite donc en partie l'étape de vérification ($\mathcal{P}post_1$, $\mathcal{P}post_2$ et $\mathcal{P}post_3$).

1.5 Synthèse

Dans cette thèse, nous cherchons à faciliter la conception de simulations informatiques, modélisant des phénomènes faisant intervenir un grand nombre d'entités entretenant des interactions variées (simulations large échelle). À cette fin, nous avons caractérisé dans ce chapitre l'objet de nos études (la simulation informatique), les problématiques lui étant inhérentes ; Nous avons de plus montré la contribution des notions d'*agent* et de *systèmes multi-agents* à leur résolution.

Parmi les différents sens pouvant être donnés à la « simulation informatique », **nous nous attachons dans cette thèse aux simulations dites explicatives**. Ces simulations visent à expliquer l'apparition d'un phénomène macroscopique observé (par exemple une termitière formant une arche), en décrivant uniquement le comportement individuel des entités présentes dans la simulation (par exemple le comportement de termites).

Pour conduire de telles simulations, une explication candidate au phénomène doit être formulée à l'aide d'une représentation abstraite du phénomène, appelée **modèle**, puis implémentée en un **simulateur**, afin d'être éprouvée par des **expérimentations**. Nous couvrons dans cette thèse un sous-ensemble de ce processus, appelé **processus de conception de simulations**, allant de la construction du modèle à son implémentation.

Le procédé permettant d'exprimer et valider l'explication fournie à un phénomène est sujet à divers problèmes, que nous résumons dans la table 1.1(a) page 33. Les principes permettant de résoudre ces

TABLE 1.1 – Description des problèmes rencontrés lors de la conception de simulations (a), et des différents principes facilitant leur résolution (b). Chaque principe décrit dans la table (b) concerne un ou plusieurs problèmes mentionnés dans la colonne « problème traité ». Ce tableau n'est pas exhaustif : il fournit uniquement une synthèse des solutions évoquées dans ce chapitre.

(a)

Origine	Nom	Description du problème
Construction d'un modèle biaisé	$\mathcal{P}prior_1$	Le modèle (ou l'implémentation) reflète quelque chose de différent de ce que pense son créateur (le formalisme du modèle est mal utilisé).
	$\mathcal{P}prior_2$	Une ambiguïté des informations fournies a abouti à une mauvaise interprétation et donc à un mauvais choix lors de la construction d'un modèle ou de son implémentation.
	$\mathcal{P}prior_3$	L'omission d'une information a abouti à un choix arbitraire lors de la construction d'un modèle ou de son implémentation.
	$\mathcal{P}prior_4$	Indépendamment des informations fournies, un choix involontaire et implicite a été fait lors de la construction d'un modèle ou de son implémentation.
Identification de simulations erronées	$\mathcal{P}post_1$	La simulation fournit les résultats attendus, malgré des erreurs/biais concernant des aspects mineurs du modèle ou de l'implémentation.
	$\mathcal{P}post_2$	La simulation ne fournit pas les résultats attendus, à cause de biais/d'erreurs contenus dans le modèle ou l'implémentation.
	$\mathcal{P}post_3$	La simulation fournit les résultats souhaités uniquement à cause des biais/erreurs concernant des aspects majeurs du modèle ou de l'implémentation.
Révisions du modèle	$\mathcal{P}rev_1$	La prise en compte de nouveaux paramètres change profondément la structure du modèle et de son implémentation.
	$\mathcal{P}rev_2$	Une erreur identifiée dans le simulateur doit pouvoir être interprétée au niveau du modèle de la simulation.
	$\mathcal{P}rev_3$	Les révisions de modèle induisent une ré-implémentation complète de la simulation.

(b)

Origine	Description de la solution	Problème traité
Construction d'un modèle biaisé	Utiliser dans le modèle des notions issues des phénomènes simulés.	$\mathcal{P}prior_1$
	Favoriser l'identification des problèmes et choix relatifs à l'implémentation lors de la construction du modèle.	$\mathcal{P}prior_2$, $\mathcal{P}prior_3$, $\mathcal{P}prior_4$
	Fournir des représentations visuelles favorisant la communication entre les différents acteurs du processus de conception de simulations.	$\mathcal{P}prior_1$, $\mathcal{P}prior_3$
	Aider le choix d'un formalisme de modélisation et d'une plateforme implémentation à l'aide de critères de comparaison.	$\mathcal{P}prior_1$
Identification de simulations erronées	Favoriser les tests unitaires d'implémentation.	$\mathcal{P}post_1$, $\mathcal{P}post_2$, $\mathcal{P}post_3$
	Comparer les résultats expérimentaux à des résultats extraits du phénomène, obtenus d'une simulation prédictive du même phénomène, ou d'une résolution analytique par modèle mathématique équivalent.	$\mathcal{P}post_1$, $\mathcal{P}post_2$, $\mathcal{P}post_3$
	Comparer les résultats expérimentaux obtenus de modèles et de simulateurs différents.	$\mathcal{P}post_1$, $\mathcal{P}post_2$, $\mathcal{P}post_3$
	Analyser la sémantique du modèle.	$\mathcal{P}post_1$, $\mathcal{P}post_2$, $\mathcal{P}post_3$
Révisions du modèle	Rendre l'architecture de la simulation modulaire.	$\mathcal{P}rev_1$, $\mathcal{P}rev_3$
	Construire des librairies d'éléments réutilisables	$\mathcal{P}rev_1$, $\mathcal{P}rev_3$
	Favoriser le morphisme entre modèle et implémentation, afin d'identifier plus facilement la part du modèle à l'origine d'une erreur identifiée dans l'implémentation.	$\mathcal{P}rev_2$

problèmes sont nombreux et sont plus ou moins bien exprimés dans chaque approche de conception de simulations. Nous en résumons les principaux dans la table 1.1(b). Afin d'éviter au maximum les problèmes de simulation, le modèle doit fournir un compromis exprimant au mieux chaque principe. Il doit pour cela **définir une représentation des connaissances appropriée**.

De ce point de vue, le paradigme *« agent »* et les ***systèmes multi-agents*** **sont particulièrement adéquats à la modélisation de simulations**. En effet, la représentation des connaissances y repose sur la définition du comportement d'entités autonomes (les agents) pouvant interagir. Elle a pour particularités :

- de reposer sur un concept familier aux experts du domaine. Elle favorise donc leur intervention dans le processus de conception de simulations ;
- d'implémenter les entités sous la forme d'agents. Elle favorise donc le morphisme entre modèle et implémentation ;
- de fournir, dans certains cas, des outils logiciels favorisant modularité et constitution de librairies logicielles.

Toutefois, les approches reposant actuellement sur ce paradigme **ne sont pas forcément toutes adaptées à la conception de simulations explicatives**. Dans le chapitre qui suit, nous étudions différents modèles, méthodologies et plateformes permettant de concevoir des systèmes multi-agents et en analysons les avantages et limites. Nous identifions par ce procédé les propriétés qui, à notre sens, doivent être exprimées par une approche de conception de simulations large échelle.

Chapitre 2

État de l'art : Conception de Simulations Multi-Agents

Plan du chapitre :

Du fait de l'hétérogénéité des domaines d'application de la simulation, différentes approches aux spécificités très différentes sont utilisées pour implémenter le modèle d'une simulation multi-agents. Nous discutons dans ce chapitre des avantages et inconvénients de ces différentes approches, en les regroupant par philosophies de conception. Cette étude est scindée en trois sections.

Dans la section 2.1, l'analyse porte sur les moyens employés afin d'implémenter la simulation. Nous dégageons de cette étude l'importance d'utiliser une approche transversale de conception, accompagnant les concepteurs dès les premières étapes de la simulation jusqu'à l'obtention d'un simulateur concret. La possibilité de construire une telle approche dépend fortement de la structure du modèle utilisé.

Dans les deux sections qui suivent, les apports et les lacunes de différents modèles sont étudiés relativement au processus transversal de conception de simulations. La section 2.2 étudie cette question selon la perspective de l'architecture interne des agents et la façon dont leur comportement est construit. Enfin, la section 2.3 étudie cette question selon la perspective des interactions ayant lieu entre les agents et la façon dont le lien entre interactions et comportement des agents est établi.

Avant de nous attaquer l'état de l'art, nous commençons ce chapitre par une brève discussion concernant la distinction entre modèle conceptuel, modèle exécutable et simulateur.

Distinction pratique entre modèle conceptuel/modèle exécutable La distinction entre modèle conceptuel, modèle exécutable et simulateur marque la transition progressive entre descriptions de haut niveau, abstraites et peu précises (modèle non-formel) à des descriptions de bas niveau concrètes et pouvant être implémentées (modèle exécutable). La distinction modèle/implémentation que l'on trouve par exemple avec Aalaadin/Madkit [FG98] ou DEVS [ZKP00]/JAMES [SU01] est syntaxique. Elle est donc simple à identifier. La distinction entre modèle conceptuel et modèle exécutable n'est pas aussi simple.

En théorie l'unique différence entre modèle conceptuel et modèle exécutable tient à la présence (ou non) d'informations propres à l'implémentation. Il n'existe à ce jour aucune ontologie permettant de décrire précisément un phénomène et symétriquement aucune ontologie ne permet de définir clairement ce qui est propre à l'implémentation. Par conséquent, nous ne faisons pas de différence entre modèle conceptuel et modèle exécutable dans ce chapitre. Nous les désignons tous deux indistinctement par le terme « modèle ».

2.1 Pratique du processus de conception de simulations

Le processus de conception de simulations peut être pratiqué selon des approches très différentes nécessitant des degrés d'expertise en informatique variables. Nous en distinguons trois grandes familles :

- les approches ouvertes qui reposent sur des langages permettant de spécifier librement presque n'importe quel type de simulations. Cette famille d'approche comprend les langages de programmation, les plateformes multi-agents ouvertes, les plateformes multi-agents dédiées aux experts du domaine et les plateformes de simulation multi-agents ouvertes ;
- les approches dirigées par la structure du modèle qui aident la spécification des simulations en fournissant une architecture spécifique et précise au modèle et à l'implémentation ;
- les approches transversales qui décrivent non seulement un modèle formel et une architecture d'implémentation, mais fournissent aussi une méthodologie de conception. Cette dernière décrit explicitement comment construire le modèle et comment aboutir à son implémentation.

2.1.1 Langages de programmation

Le choix de la plateforme à utiliser pour implémenter une simulation est loin d'être évidente pour plusieurs raisons. La première raison est qu'il existe un très grand nombre de plateformes multi-agents : Nikolai [NM09] en compte plus de 50 et sa liste n'est pas exhaustive. Certains travaux tels que Masev [GGB08] établissent des critères de comparaison inter-plateformes pour aider ce choix. Il reste toutefois nécessaire de comparer les plateformes deux à deux pour trouver la plus appropriée à la simulation réalisée. Ces critères sont donc peu utilisés. La seconde raison est liée à l'opacité des plateformes et leur manque de documentation. En effet, en l'absence de connaissances précises sur le fonctionnement interne de la plateforme et des choix d'implémentation lui tant sous-jacents, des résultats erronés peuvent être obtenus avec un modèle pourtant correct.

Afin d'éviter ces problèmes, une première approche de l'implémentation d'un simulation consiste à implémenter intégralement le modèle dans un langage de programmation tel que C, C++, Java ou Fortran [Axe97, Sha98]. Les choix d'implémentation du simulateur sont pleinement contrôlés, évitant ainsi les erreurs liées à l'opacité ou le manque de documentation des plateformes existantes. Toutefois, ce gain est obtenu au détriment de deux problèmes majeurs du processus de simulation. D'une part, la réduction des efforts d'implémentation lors des révisions du modèle dépend uniquement des compétences du programmeur. D'autre part, la validité des choix d'implémentation repose uniquement sur l'expérience du programmeur concernant la simulation et le langage utilisés. De nos jours, ce type d'approche n'est donc que très peu utilisé [Sha98].

2.1.2 Plateformes multi-agents ouvertes

Comme nous l'avons mentionné dans le chapitre 1, le paradigme multi-agents est particulièrement adapté pour implémenter les simulations. La spécification d'une simulation pourrait donc s'appuyer sur des plateformes ouvertes dédiées à la conception de systèmes multi-agents telles que Cougaar [HTW04], JADE [BPR99], Madkit [FG98] ou MAGIQUE [BM97]. En effet, ces plateformes implémentent de manière concrète le concept d'agent et d'ordonnanceur de l'activité des agents. De plus elles s'appuient pour la plupart sur un modèle formel qui peut être utilisé pour représenter le modèle de la simulation. Par exemple, JADE repose sur les spécifications de la FIPA [FIP10b] et Madkit [FG98] sur le modèle AGR.

Exemple : la plateforme Madkit

Madkit est une plateforme écrite en Java permettant d'implémenter des systèmes multi-agents reposant sur le modèle Agent/Groupe/Role [FG98]. Cette approche vise principalement à concevoir des applications hétérogènes et réparties. Elle se focalise donc sur des problématiques d'interopérabilité entre agents hétérogènes. La spécification et l'implémentation est centrée sur l'organisation du système multi-agents et se base sur trois concepts fondamentaux : les agents, les groupes et les rôles.

Un *agent* est une entité autonome et communicante pouvant jouer des *rôles* dans des *groupes*. Un groupe est un regroupement d'agents représentant un sous-système du système multi-agents. Un agent

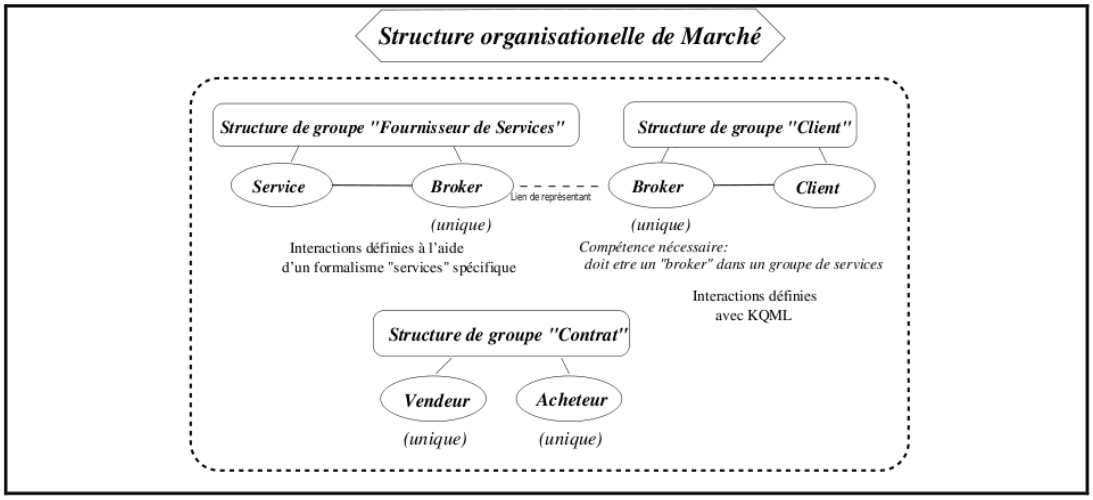


FIGURE 2.1 – Illustration des concepts de rôle, groupe, structure de groupe et structure organisationnelle du modèle AGR décrivant un marché. Dans cet exemple, un agent du rôle `Broker` sert de pivot entre agents du rôle `Client` et agents du rôle `Service` afin d'appareiller de manière dynamique un `Client` et un `Service` en tant que `Vendeur` et `Acheteur`.

peut appartenir à plusieurs groupes. Un rôle est une représentation abstraite d'une fonction, d'un service ou sert à identifier un agent au sein de son groupe. Chaque groupe spécifie l'ensemble des rôles qu'il peut contenir et chaque agent présent dans un groupe spécifie le ou les rôles qu'il peut y jouer. Pour compléter ces notions et permettre l'implémentation d'un système multi-agents, s'ajoutent au modèle les notions de *structure de groupe* et de *structure organisationnelle*. Une structure de groupe décrit les interactions [8] pouvant avoir lieu entre agents dans un groupe en fonction des rôle qu'ils peuvent y jouer. Elle est représentée sous la forme d'un graphe dirigé où les nœuds sont les identifiants des différents rôles du groupe et où les arcs connectent les rôles pouvant interagir. La structure organisationnelle décrit de manière macroscopique le système multi-agents. Elle est constituée de l'ensemble des structures de groupe composant le MAS et identifie les agents dits représentants qui servent d'interface entre des groupes différents.

L'architecture comportementale des agents ne faisant pas partie des problématiques fondamentales de cette approche, sa spécification est délaissée et doit être intégralement réalisée lors de l'implémentation. Il en va de même pour la notion d'environnement, limitée dans ce cas à un environnement social (le groupe). Malgré ces difficultés, il est toutefois possible d'implémenter des simulations sur ces plateformes. Par exemple, la plateforme Madkit a été utilisée pour implémenter la librairie Turtlekit [Mic00]. Cette dernière permet d'implémenter des simulations ayant lieu dans un environnement en deux dimensions.

Une extension du modèle AGR appelée AGRE [JFB05] réduit partiellement les problèmes liés à l'implémentation de simulations en intégrant à son modèle le concept d'environnement. Une généralisation récente du modèle AGR nommée MASQ [SFT09] fait de même en ajoutant de plus des concepts sociaux tels que les institutions ou les normes.

Discussion

Les plateformes ouvertes ont pour motivation première la spécification de systèmes multi-agents hétérogènes. Elles font donc sciemment le choix de ne pas imposer d'architecture interne spécifique aux agents, afin de garantir la plus grande hétérogénéité possible. Bien qu'il soit possible d'implémenter directement une simulation multi-agents sur ces plateformes, elles sont en général limitées au développement de librairies de simulation.

8. « interaction » peut désigner des notions différentes (voir section 2.3). Dans le cas présent, ils s'agit de protocoles d'interaction, *i.e.* de protocoles décrivant des échanges de messages réalisés par des agents pour atteindre un objectif.

Dans ces approches l'implémentation d'un élément fondamental de la simulation, le comportement des agents, nécessite la maîtrise de langages de programmation tels que JAVA ou C. Pour que le comportement des agents soit le plus simple à spécifier, d'autres plateformes permettent d'implémenter des simulations à l'aide de langages simples et accessibles à des non-informaticiens.

Ce problème a donné naissance à toute une gamme de plateformes permettant d'implémenter des simulations à l'aide de langages simples à manipuler.

2.1.3 Plateformes multi-agents dédiées aux experts du domaine

Plutôt que d'utiliser des langages complexes tels que JAVA ou C, certaines plateformes reposent sur des langages de programmation plus intuitifs à utiliser. La conception du comportement des agents reste libre et non guidée, mais peut être faite par des experts du domaine. Ces plateformes reposent sur des langages de programmation expressifs tels que Netlogo [WC99] ou sur des langages de programmation graphique que l'on retrouve dans Repast Simphony [NTCO07] ou dans SeSam [KHF06].

Exemple : la plateforme Netlogo

Netlogo [WC99] est une plateforme multi-agents basée sur le langage de programmation Logo. Elle permet de spécifier des simulations dans lesquelles des agents évoluent dans un espace en deux dimensions. Une simulation y consiste à contrôler le comportement d'un ensemble de tortues [9] similaires à des agents. Le comportement de chaque tortue y est décrit à l'aide de commandes simples et intuitives telles que `forward 5` pour faire avancer la tortue de 5 unités ou `right 90` pour faire tourner la tortue de 90 degrés sur sa droite. Il est aussi possible d'utiliser des commandes, des procédures et des fonctions plus évoluées permettant de spécifier des comportements complexes. Ces commandes incluent la manipulation d'ensembles d'agents, la perception dans l'environnement, l'ordonnancement de l'activité des agents ou la définition de races [10] de tortues (*i.e.* des « types » de tortues). Cette plateforme fournit de plus des outils graphiques très simples permettant de paramétrer, exécuter et analyser la simulation sans passer par une phase de compilation.

Discussion

Les plateformes s'appuyant sur des langages de spécification simples favorisent l'implication de personnes n'ayant pas des compétences poussées en informatique lors de l'implémentation. Elles réduisent donc les erreurs d'interprétation du modèle lors de l'implémentation. Les simulations ou les agents sont toutefois définis en un seul bloc de code peu réutilisable ne facilitant pas les révisions du modèle. Ces approches sont donc adéquates pour construire des prototypes de simulation ou construire des simulations contenant un nombre restreint d'agents au comportement peu varié. Elles ne sont par contre pas adaptées pour construire des simulations à plus grande échelle [Axe97], *i.e.* contenant une grande variété d'agents et de comportements.

2.1.4 Plateformes de simulation multi-agents ouvertes

Pour faciliter la conception du comportement des agents, une approche basée sur les librairies de simulation peut être envisagée. Cette approche se retrouve au sein de plateformes de simulation multi-agents ouvertes telles qu'Ascape [Par01] et Swarm [MBLA96] ou Opensteer [Rey02].

Exemple : Swarm

Swarm est une librairie de programmation écrite dans le langage orienté-objet Objective-C, permettant d'implémenter des simulations multi-agents. Elle est générique et ouverte et repose sur les concepts fondamentaux d'*agent*, de *swarm* et d'*ordonnanceur*.

Un swarm est objet (au sens de la programmation orientée-objet) composé d'une collection d'agents et d'un ordonnanceur régulant l'activité de ces agents. Il constitue donc un système multi-agents. Chaque

9. « turtles » dans le texte
10. « breeds » dans le texte

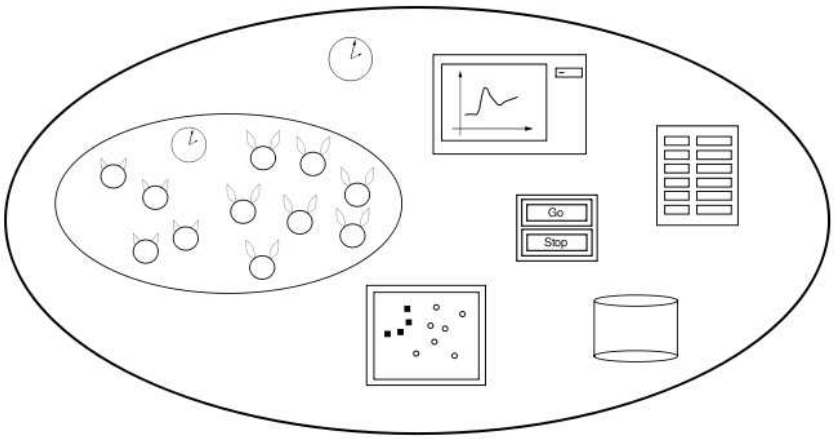


FIGURE 2.2 – Schéma représentant une simulation complète implémentée avec la librairie Swarm. Les ellipses représentent des swarms, l'horloge représente l'ordonnanceur associé au swarm. Les autres entités sont des agents.

agent est un objet (au sens de la programmation orientée-objet) qui peut aussi être un swarm. Cette structuration permet ainsi de construire une hiérarchie d'agents. Cette hiérarchie est particulièrement utile pour spécifier des phénomènes ayant lieu à des échelles différentes. Par exemple, modéliser le comportement de cellules dans un organisme, lui-même le fruit du comportement de protéines dans les cellules. La spécification du comportement des agents est libre et laissée à la discrétion du programmeur.

Cette librairie unifie la spécification du simulateur utilisé : les interfaces graphiques et de contrôle de la simulation sont aussi représentés par des agents dans un swarm (voir figure 2.2).

Outre son modèle hiérarchique, l'intérêt de swarm réside dans sa communauté. Non seulement Swarm est une plateforme ouverte et générique, mais sa communauté est très active, comme en témoigne les différentes éditions de la conférence Swarmfest [11] . Au fil de son utilisation par la communauté, des ensembles de librairies spécifiques aux domaines d'application ont été développées, telles que les swarms représentant un environnement en deux dimensions, les agents basés sur des réseaux de neurones, *etc.* Ces librairies permettent de réutiliser différentes architectures d'agents et d'environnement, réduisant ainsi les efforts d'implémentation d'une simulation à une autre.

Discussion

Les plateformes de simulation multi-agents ouvertes fournissent un modèle de base permettant de structurer la simulation à l'aide des notion d'agent ou d'ordonnanceur. Elles laissent de plus une liberté totale d'implémentation moins contraignante que dans les plateformes dédiées aux systèmes multi-agents. En effet, les plateformes ouvertes comme Swarm sont utilisées pour développer des librairies réutilisables, réduisant ainsi les coûts liés à l'implémentation d'une simulation. Elles favorisent de plus la validité de l'implémentation en réutilisant des librairies déjà validées.

De telles plateformes permettent donc d'implémenter un large spectre de simulations, tout en fournissant une aide à la conception au travers de librairies. Toutefois, cette aide se limite à la réutilisation de différentes architectures pour la simulation. La conception du modèle et de son implémentation ne sont pas guidées, si bien que la validité d'un simulateur dépend toujours compétences en programmation de la personne effectuant l'implémentation. Il en va de même pour la réutilisation de l'implémentation lors des révisions du modèle.

11. voir l'url `http://www.swarm.org/wiki/Swarm:_SwarmFest`

2.1.5 Approches dirigées par l'architecture du modèle

Les approches mentionnées jusqu'à présent sont ouvertes et permettent d'implémenter librement la simulation. Elles sont donc applicables à un grand nombre de simulations. Néanmoins, cette liberté est acquise au prix d'une absence presque totale d'aide la conception du comportement des agents. Les programmes conçus sont de plus peu robustes aux révisions du modèle.

Une autre approche consiste à fournir une architecture et un modèle précis aux agents et à leur comportement. Cette précision est acquise au prix de la liberté d'écriture du comportement et peut donc sembler restrictive. Elle est néanmoins nécessaire pour spécifier des simulations contenant un nombre important d'agents ou contenant des agents au comportement complexe. Ce problème n'est d'ailleurs pas limité aux simulations, comme le mentionne Ferguson à propos de la conception d'applications :

> « In most professions, competent work requires the disciplined use of established practices. It is not a matter of creativity versus discipline, but one of bringing discipline to the work so creativity can happen. » [FHK+97], d'après [Rob06]

La description d'une architecture précise du modèle permet de grandement guider la phase d'implémentation et d'éviter des choix pouvant aboutir à des simulateurs non valides. La transition entre modèle et implémentation est ainsi facilitée. En dépit de ces avantages, ces approches ne facilitent pas totalement la conception de simulations. En effet, les modèles ont une structure complexe et ne peuvent donc être conçus sans une aide appropriée. Ce problème est d'autant plus fondamental dans le cas de simulations contenant un grand nombre d'agents différents interagissant de manière variées.

Afin de faciliter la conception de simulations, le modèle doit pouvoir être conçu graduellement. Cette aide à la conception est obtenue dans les approches que nous qualifions de « transversales ».

2.1.6 Approches transversales

La conception du modèle nécessite plus que sa simple architecture. En effet, un modèle précis repose sur un grand nombre d'informations qu'il n'est possible de spécifier que graduellement. Il faut en particulier savoir par où commencer la spécification d'un modèle, quel cheminement suivre pour parvenir à un modèle complet et comment implémenter le modèle obtenu en un simulateur. Nous appelons *approche transversale de conception de simulations* les approches fournissant les outils répondant à ce problème.

Définition 1. ***Approche transversale de conception de simulations***
Nous appelons **approche transversale de conception de simulations** les approches supportant la conception d'une simulation du début du processus de conception de simulation (la description du modèle) à son implémentation sur une plateforme de simulation donnée.
Ces approches sont caractérisées par quatre éléments :
- un modèle formel ;
- une plateforme de simulation ;
- une méthodologie permettant de construire graduellement un modèle en passant de descriptions abstraites de haut niveau à des description fines et précises ;
- des moyens automatisant l'implémentation d'un modèle.

Bien que fondamental, ce problème est traité dans peu d'approches. Ses solutions se retrouvent principalement sous deux formes qui visent à simplifier la spécification du modèle tout en y introduisant un maximum d'informations nécessaires à l'implémentation.

Conception dirigée par des vues sur un unique modèle

Le premier type d'approches transversales repose un unique modèle qu'il est possible de remplir graduellement en usant de différentes vues. Chaque vue permet des spécifications de plus en plus précises pouvant être manipulées par des experts du domaine. L'implémentation est faite à l'aide d'un générateur de code qui se charge de la traduction des modèles graphiques en une implémentation exécutable ou en un ensemble de squelettes prêts à être remplis. Ce type de solution est appliqué par exemple dans le projet Manta [Dro93] pour des simulations en éthologie.

Exemple : Le projet Manta

Le projet Manta [DF92] a pour objectif de simuler le comportement d'insectes sociaux à l'aide du modèle formel d'Ethomodélisation [Dro93]. Cette approche postule qu'un agent dispose de primitives de comportements qui décrivent ses actes moteurs comme le déplacement, le suivi de phéromones, le dépôt d'un objet transporté, *etc.* Le comportement d'un agent est décrit sous la forme de séquences de primitives de comportement, appelées tâches. Ces tâches se déclenchent de manière exclusive en fonction de la force de stimuli internes ou externes selon un modèle d'activation de Lorenz [Lor84] : une tâche ne se déclenche qu'en présence du stimuli qui lui est associé et seulement si la force du stimuli pondérée par le poids de la tâche dépasse un seuil inhibiteur. La modification du poids et du seuil inhibiteur des tâches d'un agent lui permettent de s'adapter en fonction de ses expériences antérieures.

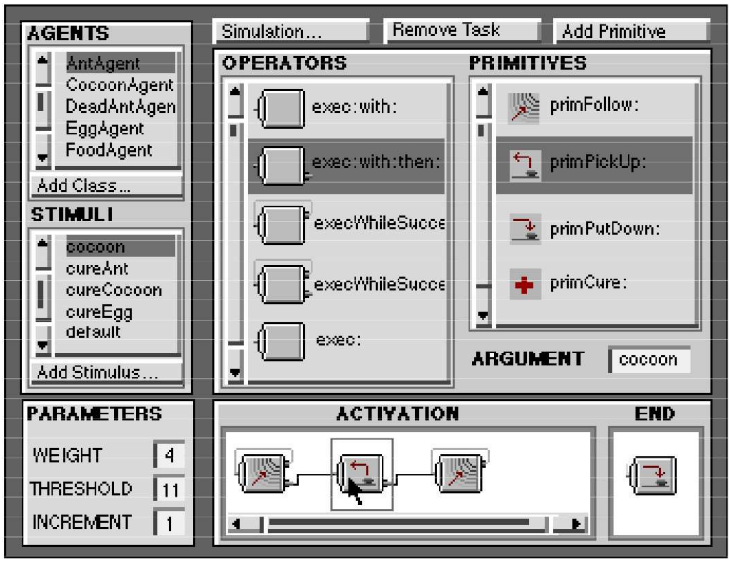


FIGURE 2.3 – Le Taskmanager du projet Manta qui permet de concevoir graphiquement des modèles d'Ethomodélisation [Dro93] et d'en générer le code. Cette capture d'écran illustre comment créer et éditer les tâches de chaque agent.

Dans Manta la conception du modèle et son implémentation se font par un outil appelé *TaskBrowser*. Cet outil permet de spécifier de manière graphique et intuitive l'ensemble des tâches de chaque agent en se reposant sur des bibliothèque de primitives de comportements pré-établies (voir figure 2.3). Il permet alors de générer le code correspondant prêt à être compilé et utilisé dans le simulateur.

Ingénierie dirigée par les modèles

Le second type d'approche transversale consiste à utiliser des méthodologies pouvant aller jusqu'à l'ingénierie dirigée par les modèles (MDE) telles que ADELFE [BCGP05], Gaia [ZJW03], INGENIAS [PGS03], O-MAsE [GODR$^+$08], Prometheus [WP04] ou Tropos [BPG$^+$04]. Dans ces approches, la conception repose sur un ensemble de modèles représentant de manière de plus en plus précise les fonctionnalités de l'application. Chaque modèle possède une représentation graphique dédiée qu'il est possible d'éditer à l'aide d'environnements de développement intégré (IDE) tels que la plateforme TAOM4E [SE 09] pour Tropos, la plateforme PDT [PTW05] pour Prometheus, la plateforme agentTool III [GODR09] pour O-MAsE ou l'Ingenias Development Kit (IDK) [Gru10] pour INGENIAS. Certaines d'entre elles permettent de plus d'automatiser la transition d'un modèle à un autre à l'aide de transformations de modèles, *i.e.* un ensemble de règles exprimant chaque élément d'un modèle source en éléments d'un modèle cible. Ces transformations sont automatisées dans les IDE par des outils dédiés comme ATL [ATL09] ou Kermeta [Tri09].

Exemple : La méthodologie INGENIAS

INGENIAS [PGS03] est une méthodologie de conception de systèmes multi-agents reposant sur un principe similaire au modèle AGR présenté en section 2.1.2. Cette approche fournit toutefois une architecture interne aux agents composée :

- d'un état mental des agents contenant des faits et des buts ;
- d'une architecture comportementale des agents fondée sur des buts réalisés à l'aide de tâches produisant et consommant des faits ;
- d'une description explicite du lien entre les tâches effectuées par les agents et la communication d'informations entre agents.

Cette méthodologie focalise la conception d'un système multi-agents selon cinq modèles/vues différents.

Le modèle portant sur l'organisation structure le système multi-agents et de définit les comportements de manière macroscopique. Ce modèle décompose le MAS en groupes et définit les rôles existant dans chaque groupe, les types d'agents présents dans chaque groupe, les dépendances sociales entre agents de l'organisation (par exemple la subordination) ainsi que l'organisation du comportement des agents. Par exemple, le comportement de chauve-souris [SPGS06] est divisé en un comportement de jour et un comportement de nuit.

Le modèle portant sur les interactions[12] décrit comment se déroulent les échanges d'informations ou les requêtes entre agents jouant certains rôles et quel est leur lien avec les tâches et l'état mental des agents.

Le modèle portant sur les agents établit le lien entre un type d'agent d'une part et les rôles qu'il joue, les buts qu'il poursuit, les états mentaux qu'il calcule ou gère d'autre part.

Le modèle portant sur l'environnement définit ce que les agents peuvent percevoir.

Le modèle portant sur les buts/tâches établit le lien existant entre les différents buts et les tâches effectuées par les agents. Il décrit de plus les faits consommés ou produits par les tâches.

Chaque modèle dispose d'une représentation graphique qui lui est propre basée sur un formalisme similaire à UML. L'IDE associé à INGENIAS appelé IDK [Gru10] permet d'éditer librement chacun de ces modèles et de passer automatiquement à une implémentation. Bien qu'initialement conçue pour la construction de systèmes multi-agents, INGENIAS peut aussi être utilisée pour décrire des simulations, en particulier dans le domaine des sciences sociales [SPGS06]. Toutefois, la conception de modèles avec INGENIAS reste fortement ouverte : aucune directive n'est donnée ni sur l'ordre dans lequel construire les modèles, ni sur la façon de procéder pour les remplir. De plus, elle laisse une part de l'architecture du comportement des agents non contrainte pour favoriser la cohabitation d'agents hétérogènes : la gestion et le calcul des états mentaux des agents n'est que mentionnée et doit être implémentée indépendamment.

Une extension [GMGSFF09] récente de la théorie liée à INGENIAS permet de remédier partiellement à ces problèmes. Elle identifie un ordre empirique dans lequel les modèles sont en pratique spécifiés (voir figure 2.4) et s'appuie dessus pour améliorer l'aide à la conception. Chaque modèle de ce cheminement fournit des informations qui sont réutilisées afin de pré-remplir automatiquement les modèles suivants à l'aide de transformations de modèles. Cette extension permet donc de mieux guider la conception des différents modèles.

Discussion

Qu'elles soient basées sur des vues multiples ou sur des modèles multiples, les approches transversales fournissent des outils facilitant la conception de simulations. En effet, elles permettent de spécifier graduellement et de manière modulaire le modèle d'une simulation à l'aide de représentations de plus en plus

12. au sens protocoles d'interaction

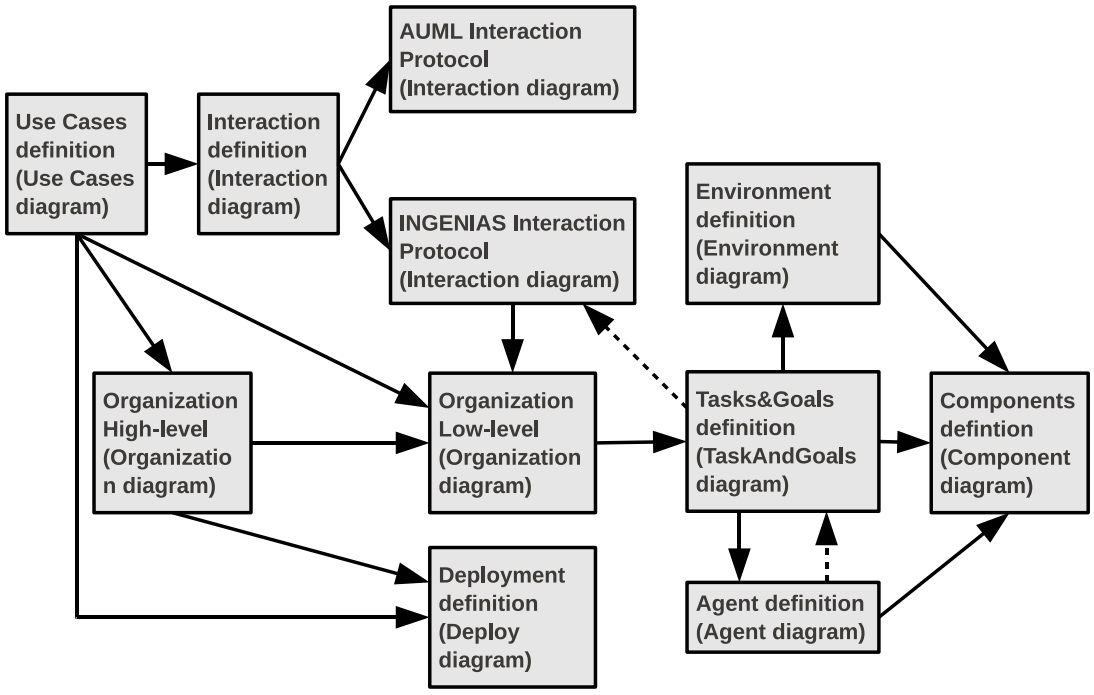


FIGURE 2.4 – Ordre de spécification des modèles guidant la conception dans l'extension d'INGENIAS présentée dans [GMGSFF09]. Les flèches pleines représentent l'ordre de conception des modèles et les flèches pointillées l'influence d'un modèle sur un autre.

précises ainsi que de passer à une implémentation par des procédés automatisés. Elles concilient donc moyens plus simples et intuitifs que la programmation pour spécifier une simulation et moyens permettant de parvenir à une implémentation. Elles fournissent donc le meilleur compromis aux différents problèmes inhérents à la simulation.

Toutefois, un problème subsiste dans ces approches : le processus de conception se focalise sur les buts que l'application doit atteindre et les fonctionnalités qu'elle doit exprimer. Elles utilisent pour cela des représentations telles que des diagrammes de cas d'utilisation (dans INGENIAS) ou de décomposition des buts devant être atteints par le logiciel (dans Tropos). Ce principe s'applique mal aux simulations puisque le but n'y est pas de fournir des fonctionnalités au sens logiciel, mais de reproduire les caractéristiques du phénomène. Pour les reproduire, la description du modèle doit d'abord se focaliser sur les éléments observables du phénomène pour seulement finir par l'introduction d'hypothèses de fonctionnement. Les éléments observables du phénomène ne s'expriment ni en termes de cas d'utilisation, ni en termes de buts : il s'agit d'actions entreprises par des entités et des interactions observées entre entités. Il est donc nécessaire d'utiliser des approches dédiées à la simulation se focalisant en priorité sur ces concepts.

Les approches transversales conçoivent une application multi-agents en suivant le même patron général résumé sur la figure 2.5. Elles commencent par décrire le système d'un point de vue macroscopique en identifiant les organisations (lorsqu'il y en a) et les relations observables entre agents (par exemple les protocoles d'interaction). Elles s'intéressent ensuite à ce que les agents sont capables de faire pour enfin finir par la description du comportement des agents. Ce cheminement permet de passer progressivement de spécifications macroscopiques observables dans le phénomène simulé à des spécifications microscopiques qui relèvent des hypothèses sur l'origine du phénomène. Il est particulièrement adapté à la conception de simulations, puisque la part des modèles les plus sujettes aux modifications sont spécifiées en aval du processus de conception.

Un tel cheminement n'est possible qu'avec des modèles ayant une structure adéquate qui va par ailleurs grandement conditionner les contributions aux problématiques inhérentes à la simulation. Dans les sections qui suivent, nous étudions différentes architectures et modèles de systèmes multi-agents et

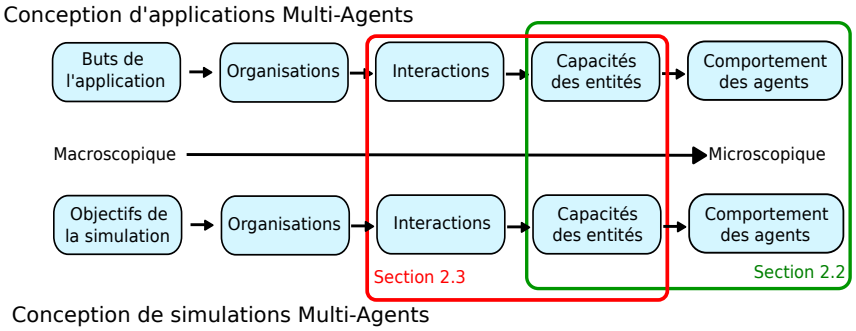


FIGURE 2.5 – Succession des principales étapes d'un processus transversal de spécification d'une simulation.

voyons en quoi elles favorisent ou non la description d'un tel cheminement. La prochaine section les étudie selon la perspective des « capacités des entités » et de leur lien avec le « comportement des agents ». Celle qui la suit se concentre davantage sur le lien entre les « Interactions » et les « Capacités des entités ». Dans cet état de l'art, nous ne prenons pas en compte la dimension « Organisations ».

2.2 Architectures multi-agents

On distingue usuellement les architectures internes par le degré de cognition exprimé par les agents. Ce dernier est caractérisé par une catégorie parmi les suivantes : *réactif*, *cognitif*, ou *hybride*. De telles architectures expriment comment un agent choisit les actions qu'il entreprend lorsque la parole lui est donnée par l'ordonnanceur de la simulation.

2.2.1 Architectures réactives

Dans les architectures réactives, le comportement des agents consiste à effectuer des actions en réaction à des stimuli perçu. Ces stimuli correspondent à une modification de leur environnement, à une sollicitation directe d'un autre agent ou à une modification de leur propre état interne.

Dans son expression la plus simple, le comportement d'un agent consiste à initier des actions en réaction à chaque stimulus qu'il perçoit. Le comportement d'un agent est alors décrit uniquement par un ensemble d'associations *stimulus/réaction*. En pratique, les modèles les plus utilisés raffinent ce principe : un agent peut y choisir les stimuli auxquels il répond en fonction de son contexte. L'agent réagit donc toujours à des stimuli, mais de manière plus intelligente permettant d'exprimer simplement des comportement parfois complexes. Ces architectures se retrouvent dans des approches telles que la subsomption [Bro86], la plateforme Maleva [BM07], la plateforme SeSam [KHF06] ou encore l'Éthomodélisation [Dro93].

Exemple : l'approche Maleva [BM07]

Maleva est une approche visant à faciliter la conception incrémentale d'agents réactifs en décrivant le comportement d'un agent comme un assemblage récursif de composants logiciels. Le comportement d'un agent y est un composant dont les bornes d'entrée correspondent à des perceptions (*e.g.* la quantité de phéromones là où l'agent se situe) et les bornes de sortie à des paramètres d'actions élémentaires (*e.g.* une distance à parcourir). Le composant peut être :

- *primitif* et calculer les valeurs des bornes de sortie à l'aide d'algorithmes manipulant les informations des bornes d'entrée ;
- ou *composite* et déterminer les valeurs produites en sortie à partir d'un assemblage interne de composants.

Le comportement est ainsi conçu de manière récursive et de plus en plus fine. La réutilisation du comportement lors des révisions du modèle est donc facilitée.

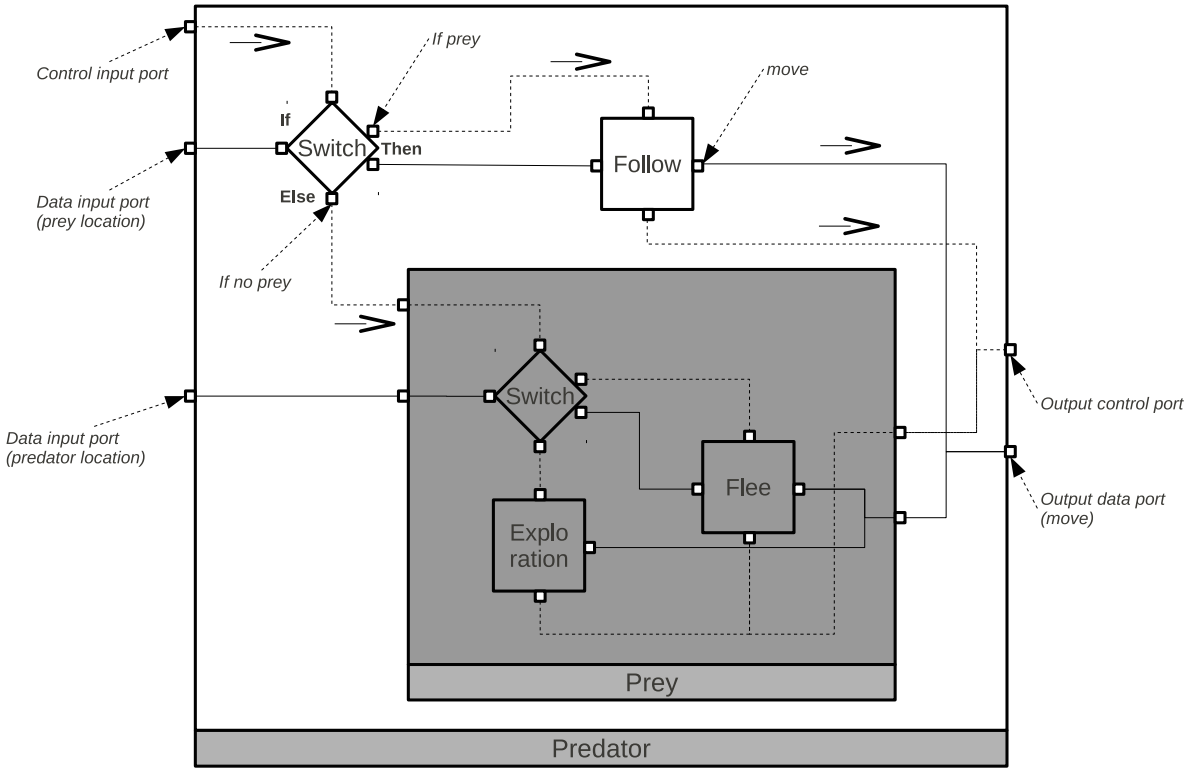


FIGURE 2.6 – Représentation du comportement d'un agent selon l'approche Maleva [BM07]. Cette représentation est ici illustrée dans le contexte de la simulation d'un écosystème où un prédateur est considéré comme une proie particulière pouvant poursuivre d'autres agents. Le comportement de proie est constitué de sous-composant décrivant dans quel cas la proie explore son environnement et dans quel cas ce comportement est subsumé par la fuite d'un prédateur. Le comportement d'un prédateur consiste à poursuivre une proie si c'est possible, sinon à se comporter comme une proie.

Chaque comportement dispose de plus de bornes de contrôle qui permettent d'inhiber le comportement d'autres composants d'une manière similaire à l'architecture de subsomption. L'inhibition se fait à l'aide de composants de contrôle qui reproduisent des structures de contrôle telles que les structures conditionnelles. La figure 2.6 illustre ces concepts pour un agent étant à la fois un prédateur et une proie.

Exemple : la plateforme SeSam

SeSam est une plateforme permettant de concevoir le comportement des agents sans connaître la syntaxe des langages de programmation « traditionnels ». Le comportement des agents y est spécifié à l'aide de graphes (voir figure 2.7). Dans ces graphes, les nœuds (appelés « activités ») sont des actions effectuées par l'agent et les arcs lient des activités effectuées en séquence. À chaque arc est associée une condition qui doit être évaluée à vrai pour que la transition vers la nouvelle activité puisse être effectuée. À chaque fois qu'un agent atteint un nouveau nœud dans son graphe, il effectue l'action qui y est associée puis cherche à effectuer une transition vers une nouvelle activité, si c'est possible.

Les actions effectuées dans une activité et les conditions de transition entre activités sont décrits sous une forme arborescente (voir figure 2.7). Chaque nœud de cet arbre correspond à une primitive d'un langage spécifique à SeSam, une opération arithmétique, une structure de contrôle ou un accesseur à l'état d'un agent ou de l'environnement.

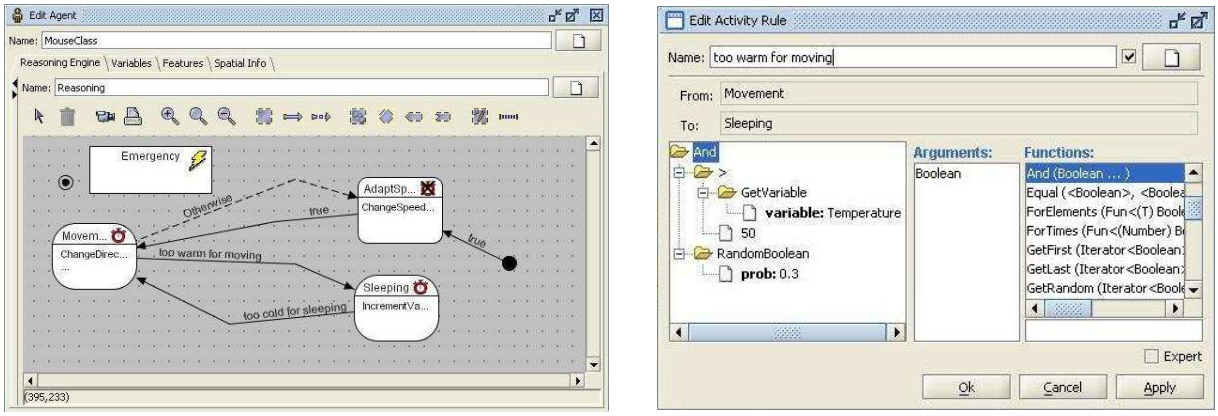


FIGURE 2.7 – Description avec SeSam du comportement d'un agent dont la température augmente lorsqu'il se déplace dans l'environnement et diminue lorsqu'il se repose. La capture d'écran gauche illustre l'interface graphique utilisée pour spécifier l'activité d'un agent sous la forme d'un graphe. Celle à droite illustre l'interface graphique utilisée pour spécifier une activité de l'agent.

La notion de temps dans le comportement est gérée en marquant les activités comme *instantanées* ou *non instantanées*. Une activité non instantanée nécessitera un pas de temps de simulation avant de pouvoir passer à une autre activité.

Discussion

Maleva et SeSam fournissent des représentations graphiques permettant de concevoir plus simplement le comportement des agents. De plus, la structure de Maleva permet de réutiliser facilement des composants logiciels. Cette approche favorise ainsi la conception de simulations contenant des agents dont une partie du comportement est un motif se retrouvant dans le comportement d'autres agents.

Toutefois, l'absence de séparation entre action et sélection d'action dans ces approches complexifie la spécification transversale de la simulation. En effet, la sélection d'action fait partie de la « boite noire » des entités du phénomène : contrairement aux actions que les entités entreprennent, ils ne sont pas directement observables dans la réalité. Ils font donc intégralement partie des hypothèses émises sur le fonctionnement du phénomène. En séparant action et sélection d'action, la modification des hypothèses est rendue plus simple et peut être interprétée plus facilement.

Cette séparation se retrouve dans des approches réactives telles que le projet Manta [DF92] présenté en section 2.1.6. Elle constitue de plus l'une des caractéristiques fondamentales des architectures cognitives et hybrides.

2.2.2 Architectures cognitives

Les architectures cognitives permettent de représenter des comportements dits « intelligents » étant mus par des buts que les entités cherchent à atteindre. Différentes architectures permettent de modéliser ce genre de comportements. Elles incluent les architectures calquées sur le fonctionnement du cerveau humain et les architectures basées sur la planification délibérative ou réactive.

Architectures issues de la théorie de la cognition humaine

Les architectures issues de la théorie de la cognition humaine se retrouvent dans des plateformes comme Soar [LNR87] ou Act-R [ABB+04]. Elles structurent la mémoire de l'agent comme la mémoire d'un humain contenant : une mémoire procédurale, une mémoire épisodique, une mémoire sémantique, une mémoire de travail, *etc.*

Ce type d'architecture sépare les actions de la sélection d'action à l'aide d'une mémoire procédurale, contenant des règles [13] de la forme *condition/action*. La condition représente les pré-requis logiques portant sur la mémoire de travail pour que l'action de la règle puisse être déclenchée. L'action décrit les effets de l'exécution de la règle sur mémoire de travail ou sur l'environnement de l'agent.

Le comportement des agents est construit par un apprentissage par renforcement : l'agent choisit la *règle* la plus adéquate à utiliser pour son contexte actuel (son but et contenu actuel de ses différentes mémoires). Il évalue ensuite la qualité du résultat et réévalue sur cette base l'intérêt de la règle pour ce contexte. Une évaluation positive favorise son utilisation future dans le même contexte alors qu'une évaluation négative l'entrave.

Les relations de l'agent avec les autres entités de la simulation n'est pas mentionnée explicitement dans le modèle. Au mieux, la mention des entités partenaires d'une action sont mêlées à la déclaration des attributs lus et modifiés dans la mémoire de travail. Leur gestion est donc laissée à la discrétion du concepteur.

Ce type d'architecture est utilisé comme support à la recherche sur la cognition humaine. Elle est peu adaptée pour décrire des phénomènes tels que la coopération entre équipes d'agents ou la construction explicite de plans pour atteindre des buts. Ils nécessitent en effet une représentation des connaissances plus abstraite.

Architectures à planification délibérative

D'autres architectures cognitives manipulent des connaissances abstraites permettant de se détacher de la théorie de la cognition propre à la psychologie humaine. Plus précisément, elles définissent explicitement dans le modèle la notion de plan. Les plans décrivent un enchaînement de règles permettant à l'agent d'aller de son état courant à un état dans lequel le but du plan est atteint. On distingue deux façons de construire les plans.

La première se trouve dans les architectures à planification délibérative telles que Prodigy [VCP+95] et les plateformes basées sur le planificateur SHOP [NCLMA99] ou GraphPlan [BF97]. Dans ces approches, les agents disposent d'une mémoire procédurale similaire à celle trouvée dans les architectures dédiées à la théorie de la cognition humaine. Pour atteindre son objectif, un agent résonne sur les règles de sa mémoire procédurale qu'il peut initier. Plus précisément, il planifie une succession de règles dont il doit effectuer successivement les actions afin de parvenir à un état dans lequel un de ses buts est réalisé. Les plans sont construits dynamiquement par un chaînage de règles selon un processus récursif. Nous présentons ici un type de chaînage particulier appelé chaînage arrière afin d'illustrer l'utilisation de la mémoire procédurale dans ce cas.

Le chaînage arrière construit un plan en déterminant dans un premier temps l'ensemble des règles dont les actions amènent à un état où le but recherché est atteint. Chacune de ces règles constitue alors

13. appelées « opérateurs » dans Soar, ou « règle de production » dans Act-R

le premier maillon d'un plan. Ces règles ne peuvent être exécutées que si le contexte de l'agent vérifie leurs conditions. Si les conditions ne sont pas toutes vérifiées, alors un plan est construit pour parvenir à un état intermédiaire où elles le sont. Le plan permettant d'atteindre le but recherché est alors la concaténation du plan permettant d'atteindre l'état intermédiaire et du maillon permettant de passer de l'état intermédiaire au but recherché. Cette construction récursive s'arrête au moment où les conditions des règles sont vérifiées par le contexte actuel de l'agent.

Le comportement d'un agent consiste à sélectionner un des buts qu'il cherche à atteindre, puis à sélectionner l'un des plans permettant de l'atteindre et enfin à exécuter l'action du premier maillon du plan.

Les architectures cognitives à planification délibérative nécessitent de fortes ressources en termes de mémoire et de calcul. En effet, le nombre de chaînes de règles qu'il est possible de construire croît exponentiellement avec le nombre de règles à la disposition de l'agent. Ces calculs peuvent être optimisés par des heuristiques estimant quels plans risquent de ne pas atteindre le but recherché. Malgré ces optimisations, une simulation ne peut contenir qu'un ensemble restreint de ce type d'agents pour être efficace en termes de temps de calculs. De plus, le comportement n'est décrit qu'en fonction de buts. Ces approches ne sont donc pas adéquates pour la conception de comportements réactifs.

Architectures à planification réactive

Pour éviter les problèmes liés à l'explosion combinatoire du nombre de plans qu'il est possible de construire une autre approche de la planification appelée planification réactive peut être utilisée. Elle consiste à construire les plans de manière réactive en se basant sur la décomposition de buts en sous-buts. Contrairement à la planification délibérative, où un plan est construit dynamiquement en construisant une chaîne de règles, les plans sont représentés de manière statique dans les règles. Chaque règle exprime dans ses conditions le but qu'elle permet de traiter et décrit dans ses actions les sous-buts qu'elle introduit pour le décomposer. Le plan est alors l'ensemble des règles qui ont permis de traiter son but et les sous-buts ayant été introduits. Cette approche est utilisée dans des plateformes comme APEX [Fre98], Jack [BHRH00], Jadex [BPL05], JAM [Hub99], Jason [BH06] ou PRS [GL87].

Exemple : la plateforme Jack

La plateforme Jack [BHRH00] repose sur une architecture à planification réactive fondée sur le modèle général Belief Desire Intention (BDI) [RG91]. La spécification du comportement y est centrée sur les *plans* qui constituent les éléments de base de la mémoire procédurale ainsi que sur les *évènements*. Un agent peut recevoir ou émettre des évènements représentant soit un stimuli interne (un but qu'il cherche à résoudre), soit un stimuli externe (un message envoyé par d'autres agents ou un évènement envoyé par l'environnement et perçu par l'agent).

Les plans sont représentés sous la forme de règles composées de 4 éléments :

- un *évènement* (« event ») auquel il répond ;
- un critère de *pertinence* (« relevance ») de l'évènement pour le plan. Ce critère permet de vérifier que des éventuels paramètres de l'évènement n'ont pas de valeur interdite pour ce plan ;
- un *contexte* (« context ») décrivant les conditions sous lesquelles le plan peut être effectué ;
- un *corps* (« body ») décrivant les actions initiées par l'agent lorsqu'il exécute ce plan.

Le corps du plan permet l'émission d'évènements internes pour ajouter des sous-buts à l'agent et ainsi effectuer la planification réactive. Il permet aussi l'émission d'évènements externes pour communiquer avec d'autres agents.

Le comportement d'un agent se déclenche lors de la réception d'un évènement et consiste à sélectionner un plan lui répondant, sous réserve que les critères de pertinence et de contexte soient vérifiés. Le corps du plan sélectionné est alors exécuté. Dans le cas où plusieurs plans peuvent répondre à un évènement, le plan utilisé est soit sélectionné aléatoirement, soit sélectionné en fonction de l'ordre dans lequel ils ont été ajoutés dans la mémoire procédurale, soit sélectionné par un méta-plan.

Jack fournit donc une architecture où il y a séparation entre mémoire procédurale et comportement. Elle permet de plus de faire de la planification réactive en décomposant des buts en sous-buts lors de l'exécution de plans. Elle nécessite donc moins de ressources computationnelles qu'une plateforme à

planification délibérative pour exécuter le comportement d'un agent. De plus, la notion d'évènement peut représenter un stimulus, un but ou un message entre deux agents. Elle unifie donc la représentation de comportements réactifs, cognitifs (planification réactive) voire coopératifs.

La modélisation des interactions s'y limite toutefois à l'utilisation de diagrammes d'interaction issus d'AUML [OPB00] et donc aux échanges de messages entre agents.

Discussion

Suite à cette étude des architectures cognitives, nous pouvons faire deux constats. Premièrement, les relations observables entre agents ne sont pas représentées ou limitées à des échanges de messages et des protocoles d'interaction. Leur intégration aux règles et au comportement des agents n'est pas détaillée et est laissée à la discrétion du concepteur, qui doit bien souvent distribuer le protocole d'interaction entre les différents agents y participant.

Deuxièmement, la spécification des buts et/ou situations auxquelles répondent une règle permet aux architectures à planification réactive de concevoir des agents étant à la fois réactifs et cognitifs avec un mécanisme unifié. Toutefois, elles nécessitent la décomposition de tous les buts qu'un agent peut avoir en sous-buts. Elles alourdissent donc la phase de conception de la simulation. De plus, ces décompositions ne sont pas exhaustives. Les agents ne bénéficient donc pas de l'ensemble des alternatives s'offrant aux agents dans les architectures délibératives.

Afin de concilier ces deux aspect et fournir un compromis entre efficacité en termes de temps de calculs et généricité, des architectures dites hybrides peuvent être utilisées.

2.2.3 Architectures hybrides

Les architectures hybrides consistent à faire coexister plusieurs architectures comportementales (appelées « couches ») et donc plusieurs représentations des connaissances au sein d'un même agent. Le comportement d'un agent y est le fruit de l'activation contextuelle d'une de ces architectures. L'activation peut se faire selon deux approches : celles dites « verticales » et celles dites « horizontales ».

Exemple d'architecture horizontale : TouringMachines [Fer92]

Les architectures hybrides horizontales consistent à activer en parallèle toutes les couches et à utiliser une unité de contrôle pour choisir la couche qui régira le comportement de l'agent. Dans l'architecture TouringMachines [Fer92], cette unité de contrôle est similaire à l'architecture de subsomption : une action proposée par la couche réactive (« Reactive Layer ») peut être ignorée si la couche de planification (« Planning Layer ») trouve une action plus intéressante. La couche de planification peut elle-même être ignorée au profit d'une couche de modélisation (« Modeling Layer ») qui permet d'anticiper les modifications futures de l'environnement et donc de choisir des plans ayant le plus de chances d'aboutir à une résolution du but de l'agent. Les critères déterminant comment la subsomption a lieu sont considérés comme dépendants du domaine d'application et sont donc décrits de manière ad-hoc.

Exemple d'architecture verticale : InteRRaP

Dans les architectures hybrides verticales, le comportement d'un agent est décidé en trois phases (les deux dernières phrases sont illustrées sur la figure 2.8 pour l'architecture hybride InteRRaP). La première phase consiste à étudier ce que l'agent perçoit, à déterminer les situations nécessitant une réaction immédiate ainsi qu'à identifier les buts que l'agent cherche à résoudre. La seconde phase détermine le degré de cognition que l'agent doit mettre en œuvre pour déterminer l'action qu'il doit effectuer. Pour cela, elle inspecte séquentiellement chaque couche jusqu'à atteindre une couche disposant des compétences suffisantes pour identifier les actions devant être exécutées. La dernière phase détermine les actions à effectuer en descendant de la couche activée la plus cognitive à la couche la plus réactive.

L'architecture InteRRaP [FMP95] repose sur trois couches :

– une couche de comportement réactive (« Behavior Based Layer » ou « BBL ») qui choisit de manière réactive une règle à exécuter (appelée « Pattern of Behavior ») afin de répondre à une situation nécessitant une réaction immédiate ;

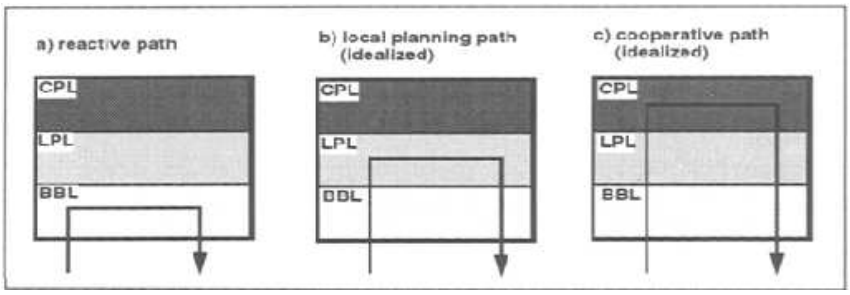


FIGURE 2.8 – Cheminement au travers des couches de l'architecture InteRRaP décrivant un comportement purement réactif (voir a), cognitif (voir b) ou coopératif (voir c).

- une couche de planification locale (« Local Planning Layer » ou « LPL ») qui est activée lorsque la couche de comportement réactive n'est pas suffisamment compétente pour répondre à la situation de l'agent. Elle consiste à construire un plan partiel permettant d'atteindre un des buts de l'agent. Les nœuds de ce plan partiel sont soit des plans, soit une situation permettant de déclencher une règle de la couche inférieure ;
- une couche de planification coopérative (« Cooperative Planning Layer » ou « CPL ») qui est activée lorsque la couche de planification locale est incapable de trouver des plans permettant d'atteindre les buts de l'agent. Elle consiste à décomposer le but en sous-buts et à résoudre chaque but. Ces buts sont résolus soit par la couche de planification locale de l'agent, soit par la couche de planification locale d'un agent acceptant de coopérer.

Discussion

Les architectures hybrides fournissent un compromis permettant de concilier aspects réactifs, cognitifs, voire coopératifs au sein d'une architecture pour agents. Ces architectures permettent de cumuler les avantages des différentes approches mentionnées dans cette section. En effet, la première couche réactive gère efficacement les situations ne nécessitant pas de planification. Ensuite, une couche de planification réactive gère efficacement les buts le plus souvent poursuivis en décrivant une bibliothèque de plans réactifs. Une couche de planification délibérative gère ensuite les cas non-supportés par la librairie de plans en construisant des plans de manière dynamique. Enfin, une dernière couche de planification coopérative gère les but ne pouvant pas être atteint sans le concours d'autres agents. Pour cela, elle permet de négocier avec d'autre agents le sous-traitement de certains de ses buts. Ces couches permettent de répondre efficacement aux différentes situations de l'agent : une couche cognitive peu efficace en termes de calculs ne sera appelée que si ses couches inférieures, qui sont plus efficaces, sont incapables de trouver une action à effectuer.

Il faut toutefois noter que la conception de tels agents se focalise sur ce que les agents sont capables de faire et sur leur comportement. Les interactions entre agents ne sont pas modélisées explicitement et sont considérées comme un effet de bord de leur comportement. La seule mention de ces relations se situe au sein de la couche de planification coopérative, où une négociation reposant sur des protocoles d'interactions est utilisée pour déléguer des buts à d'autres agents. De plus, l'utilisation de telles approches pose un problème de modélisation. En effet, elle suppose d'être capable d'établir une distinction claire entre tous les niveaux et de répondre à des questions non triviales. Par exemple : « Comment déterminer si un plan doit être spécifié dans la couche de planification réactive ou s'il doit être construit dynamiquement par la couche de planification délibérative ? ».

2.2.4 Synthèse relative aux architectures comportementales

Les architectures comportementales pour agents sont nombreuses et leurs propriétés varient du tout au tout.

Les architectures réactives permettent de décrire des comportements limités ne faisant que réagir à

des situations, mais permettent de déterminer rapidement les actions à entreprendre. La séparation entre actions et sélection d'action y est peu usuelle, impliquant ainsi des problèmes lors des révisions de modèle ainsi que lors de la conception graduelle d'une simulation. Ces architectures sont toutefois intéressantes dans le cas de simulation multi-agents contenant beaucoup d'entités de par leur efficacité en termes de calculs.

Les architectures cognitives permettent de décrire des agents plus élaborés pouvant raisonner au prix de décisions plus lentes. La séparation entre actions et sélection d'action y est usuelle et permet donc de mieux supporter les problématiques inhérentes aux révisions de modèle. Ces architectures sont donc intéressantes dans le cas de simulations contenant un faible nombre d'entités qui agissent en planifiant leurs actions.

Les architectures hybrides fournissent un compromis entre les architectures réactives et cognitives. Elles permet des décisions rapides lorsque cela est possible à l'aide de décisions réactives ou des décisions plus lentes lorsqu'il est nécessaire de raisonner pour déterminer les actions à entreprendre. La séparation entre action et sélection d'action y est usuelle autant pour les aspects réactifs que cognitifs. Ces architectures sont donc adéquates aux simulations contenant beaucoup d'agents et aux simulations contenant des agents pouvant planifier.

Bien que les architectures hybrides fournissent le meilleur compromis entre architectures réactives et cognitives, elles n'ont toutefois pas à être utilisées de manière systématique pour concevoir une simulation. En effet, selon les domaines d'application, les agents peuvent ne pas disposer de facultés réactives ou cognitives. Par exemple, en éthologie le comportement des animaux peut être décrit de manière satisfaisante avec des règles réactives, alors qu'en sociologie certains comportement ne sont mus que par des buts. Dans de tels cas, utiliser des architectures hybrides peut s'avérer moins efficace en termes de calculs puisque certaines couches ne sont jamais utilisées. Par conséquent, l'idéal serait de décrire autant que faire se peut les connaissances et le contenu de la mémoire procédurale indépendamment du comportement des agents et de décider par la suite du comportement à utiliser. Les architectures existantes permettent de le faire partiellement, grâce à la séparation entre actions et sélection d'action. Néanmoins, la structure des règles change en fonction de l'architecture utilisée pour exprimer le comportement. Il faudrait donc pouvoir trouver une représentation valide pour toutes les architectures.

La conception transversale d'une simulation nécessite de prendre en compte les interactions entre les entités présentes dans la simulation dès les premières étapes du processus de conception de simulations. Les architectures présentées dans cette section ne permettent pas ou très peu de telles spécifications. Elles sont en effet au mieux cantonnées à la description de protocoles d'interaction, *i.e.* d'échanges de messages entre agents. Pourtant, les relations entre entités ne sont pas restreintes à ce cas. Dans la section qui suit, nous étudions les différentes relations pouvant exister entre les entités d'une simulation.

2.3 Description de la dynamique macroscopique du phénomène

Les simulations explicatives visent à expliquer l'apparition d'une dynamique macroscopique observable dans l'environnement (*i.e.* un phénomène émergent) à l'aide des actions effectuées individuellement par les entités le composant. Toutefois, les comportements individuels seuls n'expliquent pas l'apparition du phénomène. Ce sont les interactions qu'ont les agents les uns sur les autres qui en sont à l'origine. Ces interactions ont différentes natures, que nous proposons d'étudier dans cette section.

La terminologie désignant ces influences ne fait pas consensus, principalement à cause de la forte connotation de certains des termes utilisés dans le domaine des systèmes multi-agents. Ainsi, avant de nous attaquer à l'étude de l'état de l'art, nous tenons à rappeler le sens premier du terme central à cette section : les « interactions ».

2.3.1 La notion d'interaction

Selon la neuvième édition du dictionnaire de l'académie française, le terme interaction a deux sens :

> « **PHYS.** Action réciproque de deux ou plusieurs corps. *La gravitation est un phénomène d'interaction entre deux corps. Interaction électromagnétique. Interaction forte, action attractive entre particules, qui assure la cohésion des noyaux atomiques. Interaction faible, qui se*

manifeste par des forces d'attraction ou de répulsion entre particules, responsables en particulier de la radioactivité bêta.
Par ext. Influence qu'exercent les uns sur les autres des phénomènes, des faits, des objets, des personnes. *L'interaction des faits économiques et politiques.* » [fra10]

En son sens premier, le terme interaction désigne les quatre interactions élémentaires entre deux corps à l'origine de tout phénomène de la physique. Elles incluent les interactions nucléaire fortes et faibles, l'interaction électromagnétique et la gravitation. En simulation informatique, le terme interaction est donc principalement compris en son sens par extension désignant tout type d'influence qu'un phénomène peut avoir sur un ou plusieurs autres phénomènes. Ce second sens est tout à fait adéquat pour désigner ce que nous étudions dans cette section. Cette définition tranche toutefois avec la définition usuelle que l'on rencontre dans les systèmes multi-agents que nous présentons dans la sous-section qui suit.

2.3.2 Protocoles d'interaction

Le sens le plus couramment rencontré de l'interaction dans le domaine des systèmes multi-agents est bien plus précis et connoté que la définition générale présentée dans la section précédente. Il est exprimé dans le contexte d'applications réparties devant communiquer entre elles pour atteindre un but qu'elles se sont fixé. Il n'est donc pas spécifique à la simulation, ce qui explique pourquoi nous parlons ici de processus de conception de systèmes multi-agents plutôt que de processus de conception de simulations.

Dans ce contexte, une **interaction** est un terme utilisé par abus langage pour désigner un **protocole d'interaction**, *i.e.* un ensemble structuré d'échanges de messages entre plusieurs entités afin d'atteindre un but particulier. Par exemple, le protocole d'interaction Contract Net de la FIPA [FIP10a] décrit les échanges de messages permettant à un agent d'effectuer un appel à propositions, puis d'accepter ou refuser les propositions provenant d'autres agents.

La description d'un protocole d'interaction peut se faire sous diverses formes. Les plus répandues sont les diagrammes issus d'UML similaires aux diagrammes de séquence, par exemple les diagrammes d'interaction de AUML [OPB00]. Dans ces diagrammes, une interaction a lieu entre plusieurs agents jouant chacun un rôle. Le diagramme d'interaction décrit les différents scénarios d'échanges de messages pouvant avoir lieu entre les rôles du protocole. La figure 2.9a illustre un tel protocole dans le cas d'un client effectuant une commande. À ces diagrammes peuvent s'ajouter des diagrammes d'activité décrivant le comportement d'un agent à la réception d'un des messages du protocole. La figure 2.9b illustre un tel diagramme dans le cas où un agent jouant le rôle *Order Processor* reçoit une requete *placeOrder* lors d'une commande.

AUML fournit ainsi des outils permettant de décrire des protocoles d'interaction. Toutefois, l'intégration de ces interactions au comportement des agents pose problème. En effet, les protocoles sont en pratique distribués dans le comportement des différents agents y participant sous la forme d'une réaction à la réception d'un message. Un protocole d'interaction n'a donc pas de représentation explicite dans les connaissances des agents. En conséquence, à chaque révision du modèle, une modification du protocole d'interaction implique la ré-implémentation partielle de tous les agents pouvant jouer un rôle dans cette interaction.

De plus, nous avons montré précédemment que le modèle seul n'est pas suffisant pour spécifier une simulation. Il faut pouvoir concevoir graduellement le modèle à l'aide d'une méthodologie. Il se pose alors un problème antérieur à la description du protocole d'interaction : dans quel cas un protocole d'interaction se révèle-t-il nécessaire pour modéliser le système multi-agents ?

Des approches permettent de remédier à certains de ces problèmes. Le langage IOM/T [DTH05] permet par exemple d'éviter la fragmentation du protocole lors de l'implémentation. De même, l'ingénierie des besoins orientée par les buts [14] que l'on retrouve dans Tropos [BPG+04] ou PASSI [Cos05] intègre la conception des protocoles d'interaction dans un processus de conception incrémental.

Exemple : Réification des protocoles à l'implémentation avec IOM/T [DTH05]

Le langage IOM/T [DTH05] donne les moyens d'éviter la fragmentation des protocoles d'interaction en les spécifiant chacun dans une unité logicielle unique. Comme pour la plupart des protocoles d'in-

14. traduction de : « Goal-Oriented Requirements Engineering »

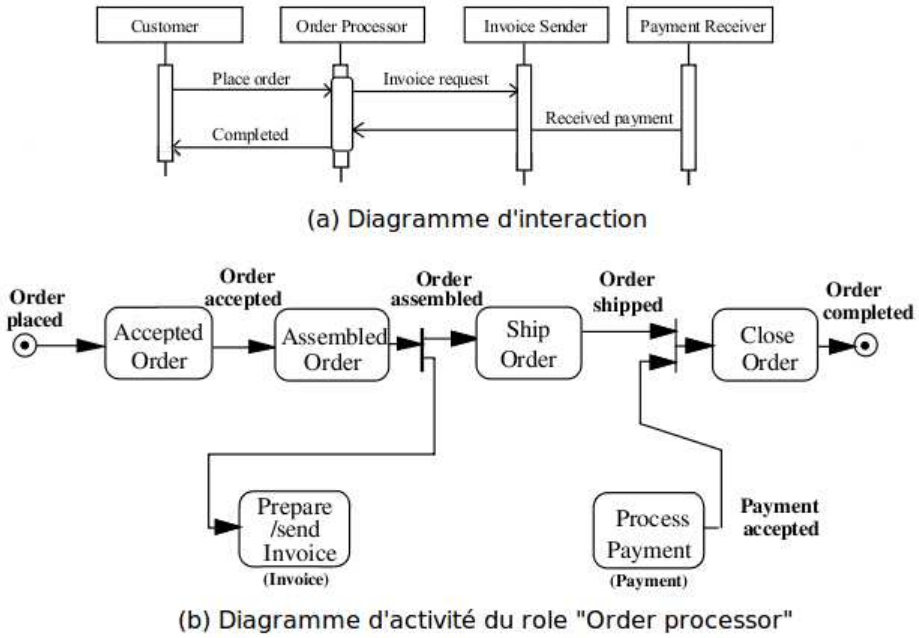


FIGURE 2.9 – Exemple d'un protocole d'interaction décrit avec AUML. Le diagramme d'interaction (a) décrit les échanges de messages permettant à un client d'effectuer une commande. Le diagramme d'activité (b) décrit le comportement d'un agent nommé « Order Processor » en réaction à la réception d'une requête de commande provenant d'un client.

teraction, la structure générale des échanges de messages y est décrite avec un diagramme d'interaction provenant d'AUML. Toutefois, IOM/T ne se restreint pas à cette simple description : chaque diagramme d'interaction est automatiquement transformé en code dans un langage proche de JAVA (voir figure 2.10). Le protocole d'interaction est implémenté sous la forme d'un unique fichier décrivant précisément :

- les rôles impliqués dans l'interaction ;
- le protocole d'échanges de messages entre les agents. Ce protocole décrit non seulement les messages échangés entre les rôles, mais aussi les instructions exécutées par les agents entre les envois de messages.

Le protocole est implémenté sous la forme d'une séquence représentant les interventions successives des différents rôles dans l'interaction. Chaque intervention d'un rôle est représentée dans un bloc, dans lequel les différentes actions entreprises par l'entité jouant ce rôle sont décrites. Ces actions incluent l'envoi d'un message, l'attente d'une réponse, des structures conditionnelles portant sur les messages reçus, *etc.* Ainsi, la sémantique du protocole n'est plus éparpillée au sein des différents agents participant à l'interaction, mais centralisée dans une seule représentation. Le protocole peut donc être réutilisé par des agents très différents, tout en minimisant les éventuelles modification possibles des agents faisant suite à une modification du protocole d'interaction.

Cette représentation du protocole d'interaction pose un problème fondamental : chaque agent dispose de spécificités ne pouvant être exprimées directement dans le protocole d'interaction. Par exemple, dans un protocole de ping itératif, le nombre d'itérations de la requête dépend de critères propres à l'agent initiant ce protocole. Pour remédier à ce dilemme, dans IOM/T les actions entreprises par les agents dans le protocole peuvent être exprimés sous la forme de fonctionnalités abstraites (« functionnalities ») relatives à chaque rôle. L'implémentation de la fonctionnalité se fait au sein des agents pouvant jouer ce rôle. Un rôle est alors similaire à une interface au sens des langages orientés-objet, où les méthodes désignent les fonctionnalités devant être implémentées par un agent jouant ce rôle. Ainsi, un protocole d'interaction est réifié en une unique entité logicielle tout garantissant la diversification du comportement des agents participant à l'interaction.

```
interaction PingProtocol {
 role Sender {
  AID getTarget();
  boolean isContinue();
  void knowAsDead();
 }
 role Receiver {
  boolean doesReply();
  ACLMessage res;
 }
 protocol {
  while (Sender.isContinue()) {
   play Sender {
    ACLMessage ping = new ACLMessage();
    ping.setReceiver(getTarget());
    ping.setContent("(ping)");
    ping.setPerformative("QUERY_REF");
    sendAsync(ping); //# m1
   }
   play Receiver {
    ACLMessage ping = recvBlock(); //# m1
    res = ping.createResponse();
   }
   if (Receiver.doesReply()) {
    play Receiver {
     res.setContent("(alive)");
     res.setPerformative("INFORM");
     sendAsync(res); //# m2
    }
    play Sender() {
     ACLMessage msg = recvBlock(); //# m2
    }
   } else {
    play Receiver {
     res.setContent(ping.getContent());
     res.setPerformative("NOT_UNDERSTOOD");
     sendAsync(res); //m3
     play Sender {
      ACLMessage msg = recvBlock(); //# m3
      knowAsDead();
     }
    }
   }
   play Sender {
    ACLMessage msg = new ACLMessage();
    msg.setReceiver(getTarget());
    msg.setContent("(end-of-loop)");
    msg.setPerformative("INFORM");
}}}}
```

FIGURE 2.10 – Description d'un protocole de ping itératif dans le langage IOM/T.

Ainsi, des librairies de protocoles d'interactions précis et réutilisables sont construits. L'absence de dispersion du protocole au sein des agents facilite de plus les itérations du processus de conception. Toutefois, cette représentation ne décrit pas comment un agent choisit d'initier un tel protocole. L'intégration du protocole d'interaction dans le comportement des agents n'est donc pas complète.

Exemple : Spécification incrémentale des protocoles d'interaction avec Tropos

Tropos [BPG+04] est une méthodologie permettant de concevoir de manière transversale un système multi-agents. Nous n'en donnons qu'une description schématique dans cette section : seul le processus donnant lieu à l'identification des interactions entre entités nous intéresse.

Cette méthodologie repose sur la décomposition des buts auxquels l'application doit répondre et les dépendances existant entre ces buts. Ces buts peuvent être de deux types :

- les « buts principaux » (« hard goal » dans le texte) représentent une fonctionnalité que le système doit fournir. Par exemple « recenser des informations relatives aux musées d'une ville » ;
- les « buts secondaires » (« soft goal » dans le texte) représentent la qualité du service fourni par le système multi-agent. Par exemple « maximiser la satisfaction des clients ».

Chaque but est représenté sous la forme d'un nœud dans un graphe dirigé. Chaque arc représente une relation de décomposition entre « buts principaux » ou une relation d'influence positive ou négative de la résolution d'un « but principal » sur un « but secondaire ».

La méthodologie Tropos conçoit une simulation en quatre phases distinctes. La première et la seconde phase, intitulées « Early Requirements Phase » et « Late Requirements Phase » ont pour objectif d'identifier tous les buts du système développé et leur décomposition par la construction puis le raffinement d'un graphe. Lors de la troisième phase, intitulée « Architectural design Phase », les « buts principaux » sont regroupés par les fonctionnalités qu'ils expriment puis sont attribués à des agents dont le rôle est de satisfaire lesdits buts. La dernière phase, intitulée « Detailed Design Phase », consiste à décrire d'un point de vue algorithmique tous les éléments identifiés par les étapes précédentes. La troisième phase permet d'identifier quels seront les protocoles d'interaction à mettre en place dans l'application. En effet, chaque

arc connectant deux buts distribués à des agents différents implique la mise en place d'une communication entre eux et donc d'un protocole d'interaction.

Ainsi, l'utilisation d'un graphe permet d'identifier de manière intuitive les protocoles d'interaction devant être mis en place. Cette représentation favorise la communication entre experts en informatique et non-informaticiens, voire la spécification directe d'une partie du système par des non-informaticiens.

Discussion

Les échanges de messages structurés sous la forme de protocoles sont une première forme d'interaction (*i.e.* d'influence entre le comportement des différentes entités d'une simulation. Ce type d'interactions est particulièrement approprié dans le cas des simulations en sciences sociales, où les différentes entités peuvent communiquer entre elles par actes de langage. Il est aussi particulièrement adapté aux simulations nécessitant une forme de coopération entre agents afin d'atteindre un but commun. Leur intégration dans le processus de conception et dans l'implémentation du système multi-agents n'est pas un problème trivial et est encore un sujet actif de recherche. Des propositions telles que IOM/T, Madkit ou Tropos y contribuent en donnant les moyens de réifier le protocole d'interaction ou d'identifier les interactions ayant lieu entre agents sous la forme d'un graphe au sein d'un procédé transversal. Ces approches souffrent toutefois de problèmes liés à l'implémentation du comportement des agents qui limitent leur intérêt pour la spécification de simulations.

En effet, une première façon d'implémenter le protocole d'interaction consiste à le réifier sous la forme d'une entité logicielle séparée des agents. Dans ce cas, la décision d'initier un protocole d'interaction est laissée à la discrétion du concepteur. Elle est donc réalisée de manière ad hoc. C'est par exemple le cas d'IOM/T ou de Madkit. Dans le cas où le protocole est fragmenté et distribué aux agents, il est possible de le traduire en un ensemble de règles déclenchées par la réception d'un message. Dans ce cas, il y a effectivement séparation entre action et sélection d'action au prix d'une dispersion du protocole d'interaction. C'est par exemple le cas dans Ingenias [SPGS06].

Ces solutions reposent sur des principes qui s'opposent et rendent donc difficile la constitution d'un compromis pourtant nécessaire à la conception de simulations.

2.3.3 Interactions et Actions

Les échanges de messages ne sont pas l'unique forme d'interaction entre entités. Odell *et al.* [OPB00] en donnent une intuition en représentant des actions telles que le clonage d'un agent par un autre agent sous la forme de diagrammes d'interaction. Par conséquent, ce qui est usuellement considéré comme une action n'est-il pas en fait une forme d'interaction ?

Cette question nous amène à considérer le sens commun du terme « action », ainsi que son sens pratique dans les systèmes multi-agents. D'après la neuvième édition du dictionnaire de l'académie française, le terme action a 5 sens dont deux propres au théâtre et au droit que nous ne mentionnons pas :

> « n.f. XIIe siècle. Emprunté du latin *actio*, dérivé de *actum*, supin de *agere*, « agir ».
> Opération d'un agent quelconque ; résultat de cette opération. *Action de* entre dans la définition des substantifs dont le sens correspond à celui d'un verbe ; le même substantif exprimant en général l'opération, le processus et le résultat de l'action considérée, on précise : « *Action de ; résultat de cette action* » [. . .].
> **1.** Exercice effectif de la faculté d'agir, par opposition à *Rêverie, Inertie, Intention, Spéculation, Contemplation, Hésitation.*[. . .]
> **2.** Production d'un effet ou manière d'agir sur quelque chose ou quelqu'un.[. . .]*La gravitation est une action à distance.*[. . .]
> **3.** Manifestation extérieure de la faculté d'agir ; ce qu'on fait. *Accomplir une bonne, une mauvaise action.*[. . .] »[fra10]

Le sens 2 est particulièrement intéressant, car il correspond exactement au problème mentionné au début de cette section. Une action peut être interprétée comme le moyen qu'un agent a d'influer sur quelque chose d'autre et donc comme une interaction entre plusieurs agents. La présence de plusieurs entités de la simulation dans une action justifie l'intérêt des diagrammes d'interaction et de séquence pour les décrire. Pourtant, dans les architectures existantes, les règles constituent une représentation des actions mais ne

font pas mention explicitement des participants autres que l'agent à leur origine. Les autres sujets de l'action sont relégués au rang de paramètres et ne sont pas mentionnés lors du processus de conception.

Certaines approches spécifiques à des domaines d'application font toutefois exception à cette règle. C'est en particulier le cas de la chimie organique, par exemple dans l'approche proposée par Desmeulles dans sa thèse [Des06], ou par Cannata *et al* [CCM08]. Ce type de représentation est aussi utilisée dans d'autres approches telles que le modèle Mascaret développé par Querrec [Que02] sous le nom d'interactions réactives.

Exemple : Réification des interactions par Desmeulles

Desmeulles propose un modèle générique permettant de réifier la notion d'interaction en son sens large, c'est à dire tout type d'action impliquant simultanément plusieurs entités de la simulation. Il fournit une application de ce modèle générique au cas des simulations en biochimie, où les seules interactions sont soit les déplacements, soit les réactions chimiques. Nous cantonnons notre présentation à ce cas particulier, afin d'avoir des propos les plus explicites possibles quant à la modélisation d'un phénomène avec cette approche.

Le modèle proposé repose sur les concepts de réaction (*i.e.* réaction chimique), d'espèce, de compartiment, de réacteur et d'organisation. Un réacteur correspond à un milieu dans lequel des réactions ont lieu. Un réacteur est caractérisé par un compartiment décrivant sa forme et son volume ainsi que par des espèces décrivant chacune le nombre de moles d'une espèce chimique qu'il contient. Enfin, une organisation est composée d'un ensemble de réacteurs dont les espèces peuvent éventuellement interagir afin de décrire les flux de molécules d'un réacteur à un autre. La figure 2.11 illustre ces concepts pour une application de transcription de l'ADN d'une cellule.

De tels modèles sont conçus à l'aide d'outils graphiques tels que le graphe présenté dans la figure 2.11. Cette représentation graphique est ensuite transformée en un fichier XML représentant explicitement les réactions, espèces, compartiments, réacteurs et organisations. Ce fichier alors exploité pour générer un simulateur dans lequel les interactions sont implémentées sous la forme d'agents. Leur comportement consiste à calculer régulièrement le nouveau nombre de moles de chaque espèce chimique impliquée cette réaction.

Cette approche se conforme à l'ensemble des pré-requis permettant de faciliter la conception de simulations. En effet, il propose un processus de conception comprenant la construction d'un modèle et son implémentation. Un morphisme entre modèle et implémentation est garanti. De plus, le modèle fait apparaître de manière explicite à la fois entités et influence des différentes entités entre elles sous la forme d'interactions. Il favorise donc la conception transversale d'une simulation.

Toutefois, dans ce modèle une interaction ne peut représenter qu'une loi de l'environnement. Cela implique que l'exécution d'une interaction ne fait pas partie d'un processus décisionnel : elles ont toujours lieu dès que leur contexte le permet. En ce sens, cette approche est similaire aux modèles de calcul à base de règles comme les P-systèmes [PĎ00] ou la chemical abstract machine [BB90]. Cette représentation n'est pas conçue pour être utilisée dans des architectures telles que la subsomption ou tout autre architecture où un agent a la possibilité de choisir l'action qu'il effectue. Cette approche n'est donc valable que pour des simulations contenant des agents purement réactifs.

Discussion

Toute action entreprise par un agent peut être interprétée comme une interaction entre plusieurs agent. La description de la dynamique du phénomène étudié passe donc par l'identification des actions pouvant lier les entités de la simulation. Actuellement, la notion d'entité subissant une action initiée par un autre agent n'est explicite que dans deux cas :

- les protocoles d'interaction, dont l'intégration dans la connaissance des agents et dans le processus de délibération des agents n'est toutefois pas complète et doit être réalisée manuellement par le concepteur ;
- les interactions purement réactives qui sont déclenchées de manière systématique sans passer par un procédé de délibération.

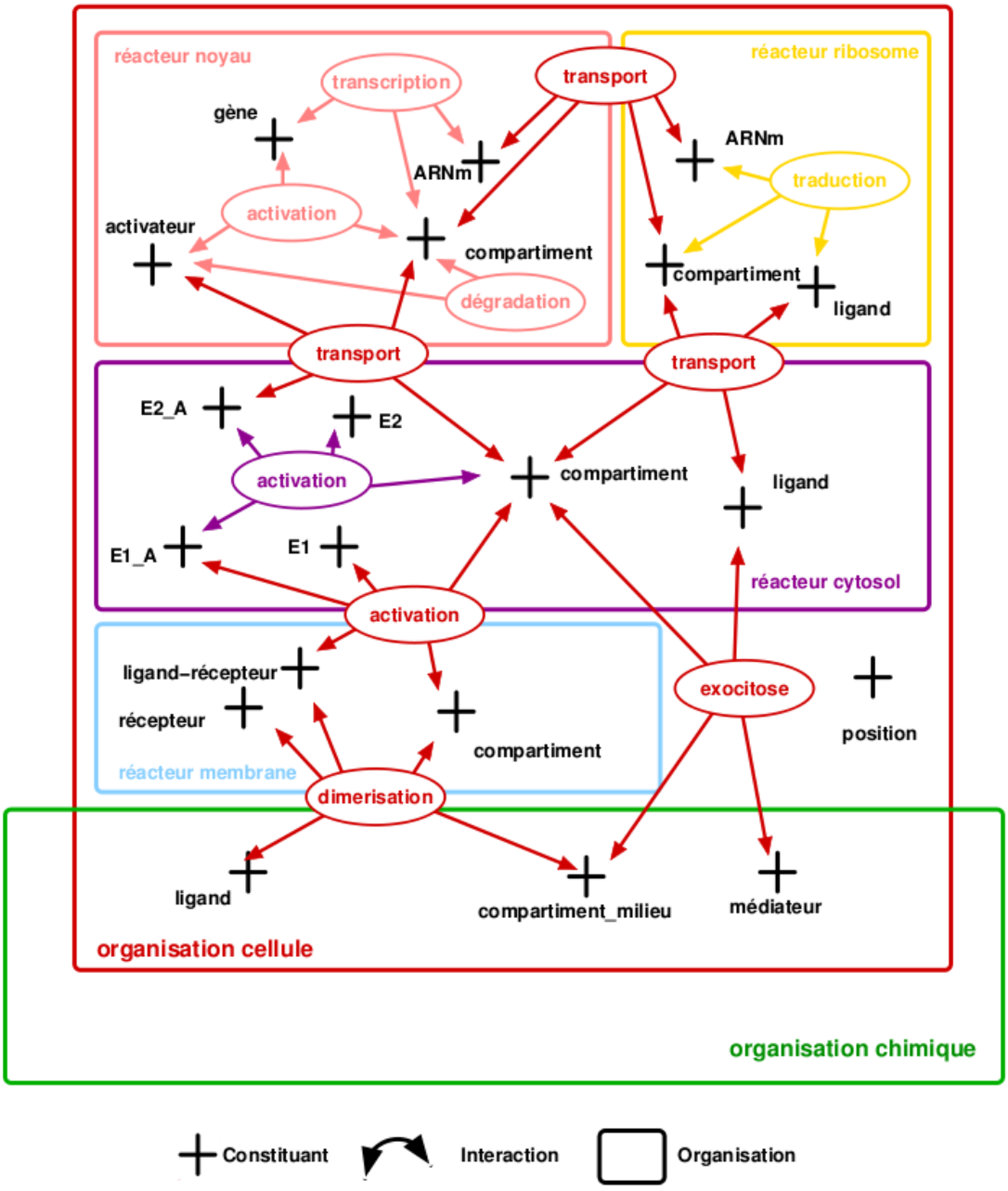


FIGURE 2.11 – Représentation graphique du modèle d'une simulation en biochimie avec l'approche de Desmeulles [Des06], dans le cas de la transcription de l'ADN dans une cellule.

La plupart des simulations explicatives font régulièrement apparaître des actions impliquant plusieurs entités sans pour autant être dans les deux cas précédemment mentionnés. Il s'avère donc nécessaire de leur trouver une représentation adéquate. Cette idée rejoint une théorie développée en psychologie appelée théorie des affordances, qui cherche à décrire les actions qu'il est possible de faire sur les éléments contenus dans un environnement.

2.3.4 Théorie des affordances

Les affordances font partie d'une théorie à l'origine de la théorie écologique de la perception (en anglais « ecological psychology ») initialement développée par le psychologue James J. Gibson en 1977 [Gib77]. Ces notions ont été ensuite reprises et étendues par Donald Norman en 1988 [Nor88] sous le concept d'affordances perçues, afin de concevoir des interfaces homme-machine.

Selon la théorie initiale émise par Gibson, tout élément contenu dans un environnement dispose de propriétés physiques (*e.g.* une surface, une masse) qui caractérisent les actions qu'une entité peut faire avec cet élément. Ces actions, appelées affordances de cet élément pour l'entité, sont relatives à l'entité cherchant à agir. Par exemple, si un élément de l'environnement est étendu, rigide et à hauteur de genoux relativement à une entité, alors il permet à l'entité d'effectuer l'action « s'asseoir ». C'est le cas d'une chaise, d'un lit ou d'une souche d'arbre pour un être humain. Ces éléments ne le permettent par contre pas pour un éléphant puisque les propriétés de rigidité et de hauteur ne sont pas vérifiées pour lui. La caractérisation de ces actions est objective et indépendante du fait qu'une entité est consciente ou pas qu'il lui est possible de les effectuer. En complément à cette théorie, Norman décrit les affordances perçues comme l'ensemble des actions qu'une entité a conscience de pouvoir faire.

Ces courants de pensée considèrent que chaque élément de l'environnement peut être le sujet d'actions lui étant intrinsèques. Ils peuvent être transposés au cas des simulations informatiques, afin de représenter les éléments de la mémoire procédurale des agents. Ce constat a en particulier inspiré les travaux de Cornwell *et al* [COST03] ainsi que de Papasimeon [Pap09]. Nous décrivons ces deux approches selon trois points de vue :

- la philosophie générale de l'approche ;
- la représentation concrète des affordances dans le modèle et en particulier leur exploitation pour définir le comportement des agents ;
- l'intégration des affordances au processus de conception de simulations.

Dans ces deux approches, il y a distinction entre d'une part les objets qui représentent toute entité présente dans l'environnement, et d'autre part les agents qui sont un sous-ensemble des objets pouvant percevoir et agir sur les autres objets de la simulation.

Les affordances en simulation selon Cornwell *et al* [COST03]

Dans l'approche de Cornwell *et al*, la définition des affordances perçues repose sur quatre principes :

- Chaque objet définit un ensemble de « types perçus » [15] lui étant rattachés.
- Chaque type perçu contient un ensemble d'actions.
- Chaque agent perçoit les objets au travers d'un ou plusieurs types perçus.
- Chaque type perçu dispose de règles de perception qui s'il est activé ou non. S'il est désactivé, un agent ne peut pas le percevoir.

Selon ces principes, les affordances d'un agent pour un objet correspondent à l'ensemble des actions contenues dans les types perçus de l'objet perçus par l'agent. Les affordances perçues sont alors les affordances contenues dans un type perçu activé.

La représentation d'une affordance dans ce modèle est propre au modèle comportemental utilisé dans l'application Performance Moderator Function Server (PMFserv). Dans ce modèle, un agent sélectionne l'affordance dont il effectue l'action en fonction de préférences calculées à l'aide d'un modèle émotionnel fondé sur l'utilisation de buts. Pour être utilisée dans cette architecture comportementale, chaque affordance décrit les effet de son action sur l'objet, mais aussi ses éventuels effets sur les buts de l'agent, sur ses émotions ainsi que sur ses préférences.

Dans cette approche, la conception d'un modèle est incrémentale et suit le procédé suivant :

15. « perceptual type » dans l'article original

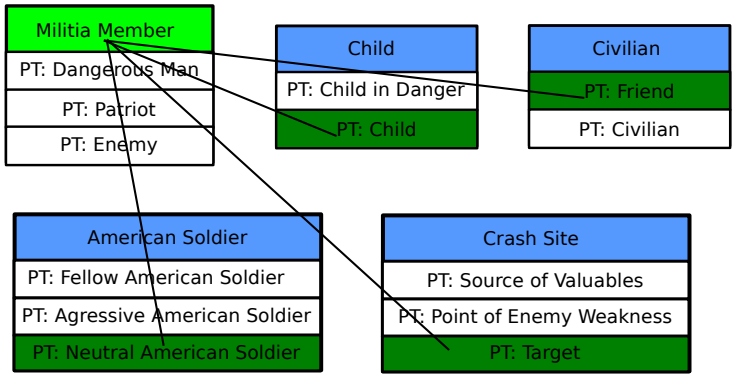


FIGURE 2.12 – Aperçu de l'outil « Object/Perception Editor » développé par Cornwell *et al* [COST03] pour éditer les affordances dans une simulation. Cette image illustre le graphe faisant apparaître les interactions entre entités de la simulation. Le trait entre « Militia Member » et « PT :Neutral American Soldier » se lit « Un agent Militia Member interagit avec un agent American Soldier via le type perçu PT :Neutral American Soldier ».

1. identifier les agents et les objets de la simulation ;
2. identifier les types perçus associés à chaque objet. Ils peuvent éventuellement provenir d'une bibliothèque de types perçus ;
3. relier agents et types perçus dont ils ont connaissance à l'aide de graphes (voir figure 2.12) ;
4. identifiées les règles de perception de chaque type perçu et les actions contenues dans chaque type perçu ;
5. décrire les actions.

La conception des affordances est donc progressive et répond en partie au besoin d'une approche transversale de conception d'une simulation. De plus, des bibliothèques réutilisables d'objets et de types perçus sont construits et contribuent donc grandement à la simplification des révisions du modèle ainsi qu'à la conception de nouvelles simulations.

Les affordances en simulation selon Papasimeon [Pap09]

Dans l'approche de Papasimeon, la définition des affordances est faite dans le cadre d'une architecture Belief-Desire-Intention (BDI) [RG91]. Deux problèmes liés à l'exploitation de la théorie des affordances en simulation y sont considérés :

- Comment utiliser la théorie des affordances au sein d'architectures BDI existantes sans pour autant en modifier profondément la structure ?
- Comment sont calculées les affordances d'un agent à un moment donné de la simulation ?

Pour répondre au premier problème, la conception des connaissances des agents est centrée sur trois éléments distincts :

- les « actions possibles » qui décrivent les actions atomiques qu'un agent peut entreprendre ;
- les « intentions possibles » qui décrivent une séquence sémantique plus ou moins complexe d'actions possibles qu'un agent peut exécuter ;
- les « affordances » qui associent à un objet de l'environnement une intention possible que l'agent peut exécuter si certaines conditions sont rencontrées.

Une affordance entre un agent et un objet est donc représentée sous la forme d'une règle condition/action. Les conditions y décrivent quelles propriétés l'agent et l'objet doivent vérifier pour que l'affordance soit perçue. Les actions consistent à ajouter une intention possible aux intentions de l'agent. Cette représentation n'émet aucune hypothèse quant à l'architecture comportementale des agents qui peut aussi bien être réactive que cognitive.

En son sens le plus strict donné par Gibson, les affordances doivent être définies dans l'environnement et non plus dans la mémoire procédurale des agents. Pourtant, un agent

doit pouvoir manipuler les affordances pour produire son comportement. Pour répondre à ce problème, Papasimeon propose deux approches différentes pour modéliser les affordances :

- s'appuyer sur le sens strict des affordances selon Gibson et stocker toutes les affordances dans l'environnement ;
- s'appuyer sur une définition plus souple des affordances et considérer que les affordances perçus d'un agent sont contenues dans sa mémoire procédurale.

Le comportement d'un agent à un instant t est défini dans cette architecture comme une séquence de 5 étapes :

1. Le recensement des entités perçues ;
2. Le recensement des affordances de l'agent avec les entités perçues. Ce recensement est plus ou moins efficace en termes de calculs selon la façon dont sont modélisées les affordances ;
3. La sélection d'une affordance en fonction de l'état mental de l'agent ;
4. L'identification de l'intention possible associée à l'affordance sélectionnée ;
5. L'exécution des actions décrites dans l'intention possible récupérée.

La différence de cet algorithme avec ceux de l'architecture BDI usuelle se trouve principalement dans la deuxième et la troisième étape.

Le modèle formel fourni par cette approche ne décrit pas concrètement la délibération de l'agent, et donne donc pas lieu à un processus de conception transversal. D'ailleurs, aucun outil d'aide à la conception n'est mentionné dans cette thèse.

Discussion

La transposition de la théorie des affordances à la simulation fait apparaître explicitement les interactions existant entre agents et entités. Cette approche est donc indiquée afin de construire un processus transversal de conception de simulations. En pratique, les approches étudiées ici rencontrent deux types de limites.

L'approche proposée par Cornwell *et al* et le modèle sous-jacent basé sur les types perçus ont été conçus pour favoriser le génie logiciel dans leur architecture appelée PMFServ. Dans cette architecture, la connaissance est représentée de manière très spécifique et repose sur certains concepts ne permettant pas son utilisation dans des agents réactifs ou dans les architectures cognitives « classiques ».

L'approche proposée par Papasimeon fournit un modèle formel précis pour décrire les affordances et leur lien au comportement des agents. Toutefois, ce modèle est voulu générique et applicable à des agents pouvant utiliser des processus de délibération très différents. En conséquence, aucune description précise du contenu des règles représentant les affordances n'est fournie. Cette absence de description précise augmente le risque de mélanger actions et sélection d'action. Pour illustrer ce point, nous reprenons un exemple développé par Papasimeon de son manuscrit de thèse [Pap09].

Cet exemple modélise un jeu de capture du drapeau impliquant des Tanks représentés par des agents. Les conditions sous lesquelles un tank α tente d'INTERCEPTER un autre tank ϵ sont :

$$IsType(\alpha, Tank) \wedge HasStatus(\alpha, Alive) \wedge IsType(\epsilon, Tank) \wedge HasStatus(\epsilon, Alive) \wedge OnOpposingSides(\alpha, \epsilon) \wedge HasCapability(\alpha, InterceptCapability) \wedge IsVisible(\alpha, \epsilon) \wedge IsClosestEnemyTank(\alpha, \epsilon) \wedge HasIntention(\alpha, DefendBase)$$

Cette description mêle :

- conditions physiques requises portant sur l'agent et l'objet. Par exemple « hasStatus(a, Alive) » précise qu'un Tank doit être en vie pour intercepter un autre tank ;
- conditions sous lesquelles une affordance est perçue. Par exemple « HasCapability(a,e) » précise que cette affordance est perçue uniquement si α est capable d'Intercepter ;
- conditions propres à la délibération. Par exemple « IsClosestEnemyTank(a,e) » qui précise que l'interception se fait sur le tank ennemi le plus proche.

L'ajout de conditions propres à la délibération réduit donc la possibilité de réutiliser ces règles. Afin de favoriser le génie logiciel, il est nécessaire de guider davantage la conception de ces conditions.

La théorie des affordances est donc prometteuse et permet de représenter les interactions entre entités provenant de tout type d'action. Toutefois, les solutions actuelles n'offrent pas de compromis satisfaisant entre généricité de l'approche utilisée, la possibilité de pratiquer le génie logiciel et conception graduelle du modèle.

2.3.5 Synthèse relative à la dynamique macroscopique du phénomène

Les éléments à l'origine d'un phénomène émergent sont le comportement des entités constituant le phénomène, mais aussi les interaction entre les entités. Le processus de conception de simulations doit donc non seulement se focaliser sur la description du comportement des agents, mais aussi sur ces interactions.

Le sens le plus commun du terme interaction dans les systèmes multi-agents désigne des protocoles d'échanges de messages entre agents. Ces protocoles sont initiés par un agent afin de répondre à un de ses propres besoins/buts. Leur conception et leur intégration dans le modèle est complexe, puisqu'elle nécessite :

- de **centraliser la description du protocole complet en une seule entité logicielle** afin de faciliter ses futures modifications ;
- de **prendre en compte les spécificités de chaque agent** participant au protocole ;
- de **concrétiser le lien entre l'aspect descriptif du protocole et son intégration dans le comportement des agents**.

Dans les solutions existant actuellement, un compromis n'est trouvé au mieux qu'entre deux de ces trois éléments.

Les protocoles d'interaction permettent de décrire des échanges de messages et supposent que les agents aient des buts. Ils constituent le moyen privilégié de modéliser une grande part de phénomènes en sciences sociales. Toutefois, ce ne sont pas les seules formes d'interactions rencontrées en simulation. En effet, dans des domaines d'applications tels que la biochimie, l'enjeu principal d'une simulation est de décrire les différentes réactions chimiques à l'origine du phénomène observé. Dans ce contexte, une interaction n'est pas un échange de message répondant à un but, mais une réaction chimique ayant lieu systématiquement entre plusieurs espèces chimiques s'y prêtant. Ce type d'interaction s'apparente beaucoup à la notion de lois de l'environnement auxquelles sont sujettes les entités d'une simulation. La notion de but leur étant extérieure, de telles interactions sont modélisées différemment des protocoles d'interaction.

Bien que ces modèles de l'interaction diffèrent, ces deux approches mettent en exergue l'intérêt d'identifier les interactions pouvant avoir lieu lors de la simulation sous la forme d'un graphe. Ils restent toutefois insuffisants pour concevoir des agents dont la délibération repose sur une architecture réactive exempte de buts. Une caractérisation des interactions dans le cas général est donc nécessaire pour poursuivre cet effort.

En appliquant aux simulations les principes de la théorie des affordances développée par Gibson et Norman, la notion d'interaction peut être généralisée à toute action qu'un agent peut faire sur une autre entité. La mention des entités sujettes aux actions est rendue explicite permettant ainsi :

- d'utiliser des représentation sous la forme de graphes pour modéliser toute action ayant lieu entre entités ;
- d'intégrer la notion d'interaction au comportement des agents.

En pratique, l'intégration au comportement est actuellement effectuée de deux manières :

- intégrée finement au modèle mais restreinte à une architecture décisionnelle particulière. Par exemple des agents émotionnels ayant des buts dans la proposition de Cornwell *et al* ;
- intégrée à gros grain au modèle en décrivant uniquement les principes à mettre en œuvre. Dans ce cas, le modèle est incomplet et aucun guide méthodologique de conception n'est défini. C'est par exemple le cas de l'approche de Papasimeon [Pap09].

Une représentation des connaissances s'inspirant de la théorie des affordances semble être la meilleure alternative pour à la fois :

- commencer le processus de conception par une vue d'ensemble sur la dynamique du phénomène ;
- intégrer les interactions au comportement des agents.

Toutefois, les approches actuelles ne sont pas suffisantes en l'état pour modéliser des simulations, en particulier lorsqu'il s'agit de modéliser des agents dont la délibération est réactive.

2.4 Synthèse du chapitre

Dans cette thèse, nous cherchons à **faciliter la conception de simulations large échelle**. Précédemment, nous avons déterminé que les agents et systèmes multi-agents reposent sur des principes généraux qui, en théorie, facilitent la conception de telles simulations. En pratique, des approches très différentes permettent de modéliser et implémenter un système multi-agents. Ce chapitre a exploré et analysé ces approches selon trois perspectives différentes, afin d'en dégager des principes facilitant la conception de simulations large échelle. Nous synthétisons ici la conclusion de ces trois études.

Comment parvenir à une implémentation ?

Pour parvenir à l'implémentation d'une simulation, il convient d'utiliser **une approche transversale**, caractérisée par :

- un **modèle précis, décrivant le phénomène à différents niveaux d'abstractions** : les organisations (lorsqu'il y en a), les interactions, ce que les agents sont capables de faire et le comportement des agents ;
- une plateforme permettant d'implémenter le modèle ;
- une **méthodologie de conception, permettant de construire progressivement le modèle** ;
- un procédé permettant d'**implémenter automatiquement** un modèle.

Une telle approche a de nombreux avantages du point de vue de la simulation :

- La conception d'une simulation commence par des descriptions abstraites de haut niveau ne nécessitant que peu de connaissances en informatique. Les experts du domaine sont donc plus facilement impliqués dans la conception du modèle ;
- L'automatisation de l'implémentation évite les erreurs provenant d'une implémentation manuelle ;
- Les différents niveaux d'abstraction permettent d'intégrer progressivement des informations dans le modèle. Il est ainsi plus aisé de concevoir des modèles contenant une grande quantité d'informations, ce qui facilite la construction simulations large échelle.

La spécification du modèle à différents niveaux d'abstraction n'est possible qu'avec une représentation des connaissances et une architecture adéquates. Pour les caractériser, nous avons étudié les modèles et architectures existants selon deux perspectives : l'architecture interne des agents ainsi que le lien entre caractérisation des interactions et spécification du comportement des agents.

Quelle architecture interne utiliser ?

Une approche transversale de conception modélise le niveau microscopique de la simulation (les agents) en deux niveaux d'abstraction : ce qu'un agent est capable de faire (ses capacités) et la façon dont il choisit les actions qu'il entreprend (sa sélection d'action). Pour modéliser ces deux niveaux, il faut :

- **séparer le déclaratif** (les éléments décrits dans le modèle) **du procédural** (les algorithmes permettant d'implémenter le modèle) ;
- représenter les **actions** des agents de manière **générique** et **indépendante de leur comportement** ;
- représenter les actions des agents sous la forme de **règles condition/effet**.

Sous ses conditions, il devient possible :

- d'**unifier la représentation des connaissances** des agents et ainsi modéliser aussi bien des agents réactifs que cognitifs avec une représentation unique ;
- de **repousser l'introduction d'hypothèses de fonctionnement à la fin du processus de conception**. En effet, contrairement aux interactions et aux actions entreprises par les agents, qui sont observables dans le phénomène, le comportement des agents est uniquement le fruit de suppositions. Sa spécification tardive permet donc de focaliser sur des descriptions plus objectives du phénomène ;
- de concevoir des **bibliothèques d'actions réutilisables**. Les révisions du modèle et la construction de nouveaux modèles dans le même domaine d'application sont ainsi facilitées.

Comment concrétiser le lien entre interactions et comportement des agents ?

Afin de passer de la description d'un niveau d'abstraction élevé (les interactions entre agents) à un niveau plus concret (le comportement des agents), il faut :

- que **toute action impliquant simultanément plusieurs agents** soit représentée par une **interaction** ;
- que les interactions entre agents soient modélisées à l'aide d'un **graphe** ou d'une forme équivalente où les agents sont des nœuds et les interaction sont des arcs ;
- intégrer les interactions au comportement des agents en les modélisant sous la forme de **règles conditions/effets**.

Sous ces conditions, il devient possible :

- de modéliser un phénomène à un haut niveau d'abstraction, sous une forme facile à manipuler ;
- d'établir un lien direct entre interactions et comportement des agents, facilitant ainsi le passage de descriptions de haut niveau à des descriptions proches de l'implémentation ;
- de faire le lien entre modèle formel et concepts spécifiques au domaine simulé, puisque la notion d'interaction peut autant modéliser une réaction chimique que la discussion entre deux agents, ou encore l'ingestion d'un aliment.

Sur quelles approches existantes faut-il se reposer ?

D'un point de vue méthodologique, les solutions s'approchant le plus des propriétés recherchées sont issues de l'ingénierie de systèmes multi-agents dirigée par les modèles. En effet, elles expriment toutes les propriétés caractérisant une approche transversale. Toutefois, ces approches sont prévues pour la conception d'applications réparties. Elles se focalisent sur l'inter-opérabilité entre agents logiciels hétérogènes et restreignent donc la notion d'interaction à des actes de langage. Le lien entre interactions et comportement des agents n'y est donc pas satisfaisant.

Du point de vue de la structure du modèle et de son implémentation, les solutions s'approchant le plus des propriétés recherchées sont issues de la théorie des affordances. En effet, elles caractérisent de manière satisfaisante le lien entre interactions et comportement des agents. Toutefois, ces approches se focalisent sur des architectures comportementales cognitives. De plus, la méthodologie permettant de construire graduellement un modèle n'y est pas définie. Elles ne constituent donc pas une approche transversale.

Notre objectif dans cette thèse est de construire une approche bénéficiant des avantages méthodologiques, logiciels et de représentation des connaissances énoncés ici. Afin de concrétiser une telle approche, nous décrivons dans la partie suivante comment construire un modèle centré sur les interactions et comment intégrer ce modèle à une approche de conception transversale bénéficiant des avantages identifiés dans ce chapitre.

Deuxième partie

Principes généraux de l'approche centrée sur les interactions IODA

Plan de la partie :

En réponse aux divers problèmes de conception identifiés dans la partie précédente de cette thèse, nous proposons une approche transversale de conception de simulations, que nous centrons sur les interactions entre agents plutôt que sur leur comportement. Cette partie du manuscrit présente cette approche générique, appelée IODA *(Interaction Oriented Design of Agent simulations), qui est composée :*

- *d'un modèle formel ;*
- *d'algorithmes génériques facilitant l'implémentation ;*
- *d'une méthodologie permettant de spécifier un modèle graduellement.*

Nous confirmons la faisabilité de cette approche par :

- *une implémentation du modèle formel en une plateforme de simulation générique et paramétrable appelée* JEDI *(Java Environment for the Design of agent Interactions) ;*
- *une implémentation de la méthodologie de conception sous la forme d'un environnement de développement intégré appelé* JEDI-BUILDER.

Le chapitre 3 détermine comment réifier la notion d'interaction dans une simulation, afin de bénéficier des avantages énoncés dans la partie précédente. Nous décrivons pour cela la structure du modèle de l'approche IODA*, à l'aide du formalisme mathématique, ainsi que des méta-modèles et d'exemples.*

Le chapitre 4 montre comment construire une approche transversale à l'aide du modèle IODA*. Nous y définissons pour cela des outils permettant à la fois de construire graduellement un modèle (une méthodologie de conception), mais aussi de l'implémenter (des algorithmes de simulation génériques). Nous analysons à cette occasion selon le point de vue de* IODA *quatre problèmes de conception rencontrés de manière récurrente en simulation. Nous dégageons de cette analyse des avantages de* IODA *tels que l'unification de la représentation des entités d'une simulation, ou la facilitation de la conception graduelle d'un modèle.*

Le chapitre 5 confirme la faisabilité de l'approche IODA *en décrivant la plateforme de simulation* JEDI*, qui implémente fidèlement les concepts de modèles* IODA *et en décrivant l'environnement de développement intégré* JEDI-BUILDER*, qui fournit une implémentation fidèle à la méthodologie* IODA.

Chapitre 3

IODA comme modèle formel

Plan du chapitre :

Nous avons établi dans la partie précédente que pour répondre aux besoins inhérents aux simulations large échelle le modèle doit exprimer un certain nombre de propriétés. Dans ce chapitre, nous définissons formellement le modèle de notre approche ***Interaction Oriented Design of Agent simulations*** *(*IODA*) et montrons que les concepts y étant développés sont conformes à ces besoins. En conséquence,* IODA *facilite la conception de simulations large échelle.*

Pour cela, la section 3.1 commence par fixer le sens donné dans IODA *aux termes « agent », « environnement » et « temps ». Nous présentons ensuite les six concepts se trouvant au cœur de l'approche* IODA *et justifions en quoi ils facilitent la conception de simulations large échelle. Ces concepts sont :*

- *les interactions (section 3.2) ;*
- *la matrice d'interaction (section 3.3) ;*
- *les entités (section 3.4) ;*
- *la matrice de mise à jour (section 3.4.1) ;*
- *l'environnement (section 3.5) ;*
- *le modèle de sélection d'interaction (section 3.6)*

Les simulations large échelle peuvent difficilement être exécutées si le comportement des agents nécessite un grand nombre de calculs, puisque dans de tels cas, seul un nombre restreint d'agents peuvent être simulés. Le compromis le plus couramment utilisé consiste à utiliser un modèle de sélection d'action réactif, dans lequel les règles condition/effet peuvent être plus ou moins cognitives. Nous intégrons à IODA *un modèle de sélection d'interaction réactif fondé sur ce principe, à l'aide d'une matrice d'interaction raffinée.*

Le modèle formel utilisé dans l'approche IODA [KMP08a, KMP08c] est structuré en six parties déclarées indépendamment : les interactions, la matrice d'interaction, l'environnement, les entités, la matrice de mise à jour et le modèle de sélection d'interaction (*c.f.* figure 3.1). Nous présentons ce modèle selon ces six axes en regroupant toutefois la présentation du modèle des entités et de la matrice de mise à jour dans une même section.

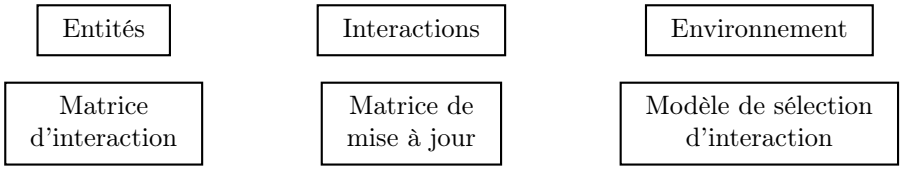


FIGURE 3.1 – Les six principales parties d'un modèle IODA.

3.1 Cadre général

Le modèle IODA est basé sur le paradigme agent et fait donc naturellement apparaître au moins trois notions : les agents, l'environnement et la représentation du temps. Ces notions peuvent prendre des sens très différents et avoir des représentation plus ou moins complexes selon les choix de modélisation effectués [KMP07]. Cette première section a pour but de dégager le sens de chacune de ces notions dans l'approche IODA, afin de définir le cadre général de ce chapitre.

3.1.1 Agent

La notion d'agent est fortement connotée et son sens est très dépendant du contexte de son utilisation. Ainsi, l'utilisation de ce terme dans ce chapitre pourrait entraver la compréhension des principes que nous énonçons. Par conséquent, bien que cela puisse paraître surprenant pour une approche de conception multi-agents, nous n'utilisons pas le terme « agent » dans ce chapitre : nous lui préférons le terme neutre « entité ». Ce choix est plus amplement justifié dans la section 4.2.1 page 109 du chapitre 4.

De plus, nous distinguons d'une part le terme *entité* qui se réfère à une entité apparaissant physiquement lors d'une simulation et d'autre part les termes *famille d'entités* qui se réfèrent à un ensemble abstrait d'entités. Cette différence est similaire à la différence entre instance et classe dans les langages orientés-objet.

Définition 2. ***Distinction Instance/Famille d'entités***
Une **famille d'entités** est une spécification abstraite d'entités pouvant initier et subir les mêmes interactions et ayant le même comportement. Une **entité** (ou **instance d'entité**) est une instance particulière d'une famille d'entités.
Une **instance d'entité** a une seule famille d'entités et une famille d'entités peut être réduite à une seule instance.

3.1.2 Environnement

La notion d'environnement est partagée entre deux sens :

1. « tout ce qui est extérieur à une entité » ;
2. « l'espace dans lequel évoluent les entités d'une simulation ».

La première définition est issue de la systémique. Elle est principalement utilisée pour concevoir le comportement de robots, où chaque entité doit construire elle-même son modèle du monde (et donc des autres entités) à partir de ce que ses senseurs perçoivent. Au contraire, en simulation explicative le modèle du monde (et donc l'environnement) fait partie des hypothèses émises sur l'origine du phénomène étudié. Par conséquent, nous considérons uniquement la seconde définition dans cette thèse.

Pour des raisons que nous exposons dans la section 3.3 page 77, l'environnement doit disposer d'une métrique permettant de calculer la distance séparant deux entités. Cette métrique n'est pas limitée à la distance euclidienne. Elle peut exprimer tout type de mesure de proximité dans un espace topologique : une distance de von Neumann, un degré de différence entre deux entités (par exemple de richesse, d'ethnie, de taille, *etc.*), un nombre d'arcs séparant deux entités dans un graphe, *etc.* Cette généralisation de la notion de distance repose sur une métaphore et des concepts similaires à ceux développés par Giavitto *et al.*[GM02] pour construire leur langage de programmation MGS. Dans leur approche, un calcul est modélisé comme un chemin parcouru dans un espace de données, qui peut être décomposé en étapes au cours desquelles un calcul élémentaire est effectué. Ils y définissent en particulier la notion de voisinage dans un espace de données pouvant directement être utilisée dans IODA pour définir la notion de distance.

Notation 1.
On pose $\mathbb{F}$ l'ensemble de toutes les familles d'entités d'une simulation, $\mathbb{E}$ l'ensemble de toutes les instances d'entité présentes dans l'environnement d'une simulation ainsi que $e \prec F$ le fait qu'une instance e ait pour famille d'entité F :

$$\forall e \in \mathbb{E}, \forall \mathcal{F} \in \mathbb{F}, (e \text{ a pour famille d'entité } F \Leftrightarrow e \prec F)$$

De plus, on définit la fonction permettant de connaître la famille d'une entité :

$$famille \ : \begin{array}{ccc} \mathbb{E} & \rightarrow & \mathbb{F} \\ e & \rightarrow & \mathcal{F} \text{ tel que } e \prec F \end{array}$$

3.1.3 Temps

Dans le cadre de la simulation, un modèle n'a de sens que s'il peut être utilisé pour construire un simulateur. À cette fin, il faut pouvoir faire évoluer le modèle dans le temps, afin de simuler le comportement du phénomène. Par conséquent, que cela soit fait implicitement ou explicitement, le modèle d'un phénomène spécifie toujours un modèle du temps.

Pour des raisons pratiques, le modèle IODA décrit dans cette thèse repose sur une représentation simple du temps dans laquelle chaque entité n'initie qu'une seule interaction en réponse au déclenchement de son processus comportemental. Il faut toutefois remarquer qu'il ne s'agit pas d'une limite de l'approche IODA, mais d'un choix effectué pour éviter d'ajouter des éléments superflus à la compréhension de notre approche.

3.2 L'interaction enfin concrétisée

Dans IODA, la spécification des simulations est centrée sur la notion d'*interaction* qui sert d'élément de base pour construire le comportement des entités. Une interaction est une séquence sémantique d'actions qui implique simultanément un nombre fixé d'entités et dont l'exécution est soumise à des conditions.

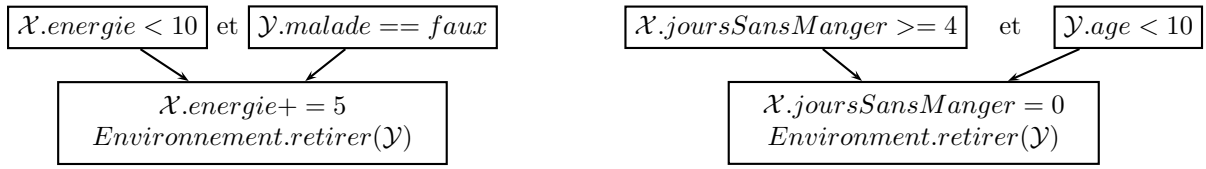
3.2.1 Polymorphisme des interactions

La présence de règles de type *conditions/actions* n'est pas rare dans les systèmes multi-agents. Toutefois, les approches décrites dans l'état de l'art reposant sur de telles règles ont un défaut commun : chaque interaction est en pratique exprimée avec des règles spécifiques à chaque entité interagissant.

Exemple.
La figure 3.2 présente la description de deux interactions sous la forme de règles appelées respectivement INTERACTION1 *et* INTERACTION2. *Ces règles semblent reposer sur des principes totalement différents puisque dans un cas l'entité $\mathcal{X}$ initie l'interaction* INTERACTION1 *en fonction d'un seuil d'énergie alors que dans l'autre cette entité initie l'interaction* INTERACTION2 *en fonction du nombre de jours de diète.*

Cet exemple montre que deux règles pourtant différentes peuvent désigner une même interaction (en l'occurrence, le fait qu'une entité mange une autre entité). Ce sens commun aux deux règles donne l'intuition de l'existence d'une description générique de l'interaction MANGER à laquelle ces deux règles se conforment. De plus, cette décomposition montre que les conditions d'une interaction peuvent être divisées en deux parties : une partie décrivant les pré-requis logiques ou physiques nécessaires à l'exécution de l'interaction et une partie décrivant les buts explicites ou implicites auxquels l'interaction répond.

Exemple.
L'interaction MANGER *mentionnée ici peut être exprimée de manière générique. En effet, $\mathcal{X}.energie < 10$ et $\mathcal{X}.joursSansManger >= 4$ sont deux façons différentes d'exprimer que $\mathcal{X}$ ressent la sensation de*



(a) Description de INTERACTION1 (b) Description de INTERACTION2

FIGURE 3.2 – Description de deux règles nommées INTERACTION1 et INTERACTION2 qui représentent une même interaction MANGER pour des entités aux spécificités différentes. Dans ce schéma, la partie supérieure de la figure représente les conditions de la règle et la partie inférieure les actions de la règle. **Ces descriptions sont utilisées pour illustrer le problème du polymorphisme des interactions. Elles ne sont pas exprimées selon l'approche défendue dans IODA.**

faim. De même, $\mathcal{Y}.age < 10$ et $\mathcal{Y}.malade == faux$ expriment que l'entité mangée doit être saine. Cette description générique est illustrée par la suite dans la figure 3.3b.

Le modèle IODA se fonde sur ce constat et considère que toute interaction peut être décrite de manière générique et indépendante des entités y participant. Pour cela, les conditions et les actions d'une interaction ne manipulent pas directement l'état des entités, mais des primitives abstraites appelées *primitives de perception* ou *d'action*.

Définition 3. ***Primitive d'action***
Une **primitive d'action** est une primitive abstraite manipulée dans les actions d'une interaction qui a un effet de bord sur au moins une des entités participant à l'interaction.
Une primitive d'action est un triplet $p = \langle id, V, (a_i)_{i\in\mathbb{N}}\rangle$ où :
– id est l'identifiant de la primitive, représenté sous la forme d'une chaîne de caractères ;
– V est un élément optionnel décrivant le type de la valeur retournée par la primitive ;
– $(a_i)_{i\in\mathbb{N}}$ représente l'ensemble ordonné (éventuellement vide) des paramètres de la primitive.

Les conditions d'une interaction ne doivent pas avoir d'effet de bord sur leurs arguments, *i.e.* les entités pour lesquelles les conditions sont testées. Par conséquent, les primitives manipulées dans ces méthodes de l'interaction ne peuvent pas manipuler des primitives d'actions. À la place, elles manipulent des *primitives de perception*.

Définition 4. ***Primitive de perception***
Une **primitive de perception** est une primitive abstraite manipulée dans les conditions ou les actions d'une interaction qui n'a aucun effet de bord sur les entités participant à l'interaction.
Une primitive de perception est un triplet $p = \langle id, V, (a_i)_{i\in\mathbb{N}}\rangle$ où :
– id est l'identifiant de la primitive, représenté sous la forme d'une chaîne de caractères ;
– V est un élément optionnel décrivant le type de la valeur retournée par la primitive ;
– $(a_i)_{i\in\mathbb{N}}$ représente l'ensemble ordonné (éventuellement vide) des paramètres de la primitive.

Exemple.
La figure 3.3b spécifie de manière générique l'interaction MANGER. *Elles repose sur deux primitives de perception appelées $aFaim()$ et $estSain()$ qui déterminent respectivement si l'entité $\mathcal{X}$ a pour but implicite de réduire sa sensation de faim et si l'entité $\mathcal{Y}$ est considérée comme saine. Elle repose de plus sur une primitive d'action appelée $reduireFaim()$ qui a pour action de réduire la sensation de faim de $\mathcal{X}$ en fonction des propriétés de $\mathcal{Y}$.*

Pour participer à une interaction, une entité doit implémenter les primitives de perception et d'action selon ses propres spécificités. Le polymorphisme des interactions est ainsi garanti tout en favorisant la réutilisation des interactions existantes.

Exemple.
La primitive abstraite $\mathcal{X}.aFaim()$ peut avoir pour sens :

- *« Avoir son énergie sous un certain seuil ». Dans ce cas, l'entité implémente cette primitive sous une forme similaire à : $\mathcal{X}.energy < 20$. On obtient alors par polymorphisme une interaction similaire à* INTERACTION1 *de la figure 3.2 ;*
- *« Ne pas avoir mangé pendant un certain temps ». Dans ce cas, l'entité implémente cette primitive sous une forme similaire à : $\mathcal{X}.joursSansManger > 10$. On obtient alors par polymorphisme une interaction similaire à* INTERACTION2 *de la figure 3.2 ;*
- *« Prédire le décès de $\mathcal{X}$ si jamais il ne mange pas dans les 10 prochains jours ». Dans ce cas, l'entité implémente cette primitive avec un algorithme d'anticipation plus ou moins complexe, etc.*

Il est à noter que l'environnement est toujours implicitement présent dans une interaction. Les conditions et les actions d'une interaction peuvent donc aussi manipuler des informations relatives à la topologie de l'environnement. La distance entre deux entités en est un exemple. Nous désignons ces primitives particulières par les termes *primitives de l'environnement*. Leur définition est similaire à celle d'une primitive abstraite d'une entité (voir section 3.5).

La **description générique des conditions et des actions rend la spécification des interactions indépendante des entités pouvant y participer**. Par conséquent, il y a séparation explicite entre l'étape de spécification des interactions et l'étape de spécification des capacités des entités. La réutilisation de cette description pour des entités différentes en devient plus aisée. De plus des **librairies d'interactions réutilisables sont construites** au fil des simulations.

3.2.2 Structure d'une interaction

Les interactions sont structurées de manière à pouvoir être utilisées aussi bien dans des architectures comportementales réactives que dans des architectures à planification délibérative ou à planification réactive. Pour ce faire, elles ont toutes une représentation homogène sous la forme de règles, composées de trois parties :

- Les **préconditions** d'une interaction décrivent des pré-requis logiques ou physiques nécessaires à l'exécution de l'interaction ;
- Le **déclencheur** d'une interaction décrit les pré-requis téléonomiques de l'interaction, *i.e.* les buts explicites ou implicites auxquels l'interaction cherche à répondre ;
- Les **actions** d'une interaction décrivent l'effet de l'interaction sur les entités y participant ainsi que sur l'environnement.

Cette structure est résumée et illustrée pour une interaction nommée MANGER sur la figure 3.3. Dans cette figure, les préconditions de l'interaction MANGER spécifient qu'une entité en mange une autre seulement si l'entité mangée est saine (par exemple si elle n'est pas toxique). Le déclencheur y spécifie que l'entité doit avoir faim pour déclencher cette interaction. Il sous-entend ainsi que le but implicite de cette interaction est de réduire la sensation de faim de l'entité. Enfin, les actions décrivent les effets de cette interaction, en l'occurrence la réduction de la sensation de faim et le retrait de l'entité mangée de l'environnement.

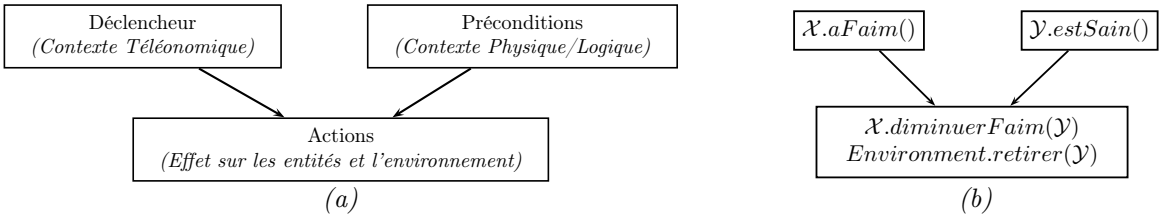


FIGURE 3.3 – Structure générale d'une interaction dans IODA (a) illustrée pour l'interaction MANGER (b) qui décrit comment et sous quelles conditions une entité $\mathcal{X}$ mange une entité $\mathcal{Y}$.

3.2.3 Typologie des interactions

Les entités impliquées dans une interaction n'y jouent généralement pas la même fonction. Pour des raisons exposées ultérieurement, nous différencions en particulier les entités qui initient une interaction des entités qui subissent l'interaction par les termes *source* et *cible*.

Définition 5. ***Entité Source/Cible d'une interaction***
On appelle entités **source** d'une interaction l'ensemble des entités qui initient l'interaction, *i.e.* les entités dont la participation dans l'interaction est le fruit de leurs propres décisions. On appelle entités **cible** d'une interaction l'ensemble des entités qui subissent cette interaction, *i.e.* les entités dont la participation dans l'interaction est le fruit de la décision d'une autre entité.

Exemple.
Dans l'interaction MANGER *décrite dans la figure 3.3b, l'entité $\mathcal{X}$ est l'unique source de l'interaction* MANGER *et l'entité $\mathcal{Y}$ en est l'unique cible.*

Une interaction est caractérisée par le nombre de sources et de cibles qu'elle implique. Nous les appelons *cardinalité de l'interaction*.

Définition 6. ***Cardinalité d'une interaction***
La **cardinalité d'une interaction** $\mathcal{I}$ est un couple $\big(card_S(\mathcal{I}), card_T(\mathcal{I})\big)$ où $card_S(\mathcal{I})$ (resp. $card_T(\mathcal{I})$) représente le nombre d'entités source (resp. cible) impliquées dans $\mathcal{I}$.
On notera par la suite $card(\mathcal{I})$ la cardinalité d'une interaction $\mathcal{I}$.

Exemple.
L'interaction MANGER *décrite dans la figure 3.3b a pour cardinalité* $(1,1)$.

Nous étudions dans cette section une division de l'espace des cardinalités d'une interaction en cinq ensembles.

Les interactions telles que $card_S(\mathcal{I}) = 0$. Ces interactions n'ont aucune entité source, ce qui sous-entend qu'aucune entité ne peut décider d'initier l'interaction. Compte tenu du fait que seule une entité peut décider d'initier une interaction de tels cas ne peuvent exister.

Les interactions telles que $card(\mathcal{I}) = (1,1)$. Les interaction de cardinalité $(1,1)$ impliquent exactement une source et une cible. Il s'agit du cas trivial d'interaction rencontré en simulation.

Exemple.
L'interaction SUIVRE *permet à une entité source de définir un de ses déplacements en fonction de la position d'une entité cible.*

Puisque ce cas se retrouve fréquemment dans les simulations, elles sont par la suite désignées par les termes *interactions individuelles*.

Définition 7. ***Interaction individuelle***
Une **interaction individuelle** est une interaction dont la cardinalité est $(1,1)$.

Les interactions telles que $card(\mathcal{I}) = (1,0)$. Le second type d'interaction le plus usuellement rencontré a pour cardinalité $(1,0)$. Elles représentent les interactions effectuées par une entité source afin d'influer sur elle-même et son propre état. Elles sont considérées comme des interactions car elles prennent implicitement leur source pour cible. Les exemples de telles interactions sont nombreux et existent dans un vaste ensemble de domaines d'application.

Exemple.
Parmi les interactions de cardinalité $(1,0)$*, on peut trouver en éthologie :*
- *l'interaction* DORMIR *qui fait passer la source dans un état de sommeil. Elle induit un changement de rythme biologique tant que l'interaction* SE RÉVEILLER *n'est pas initiée ;*
- *l'interaction* MOURIR *qui retire l'entité source de l'environnement ;*

– *l'interaction* Errer *qui déplace l'entité dans l'environnement.*

Parmi les interactions de cardinalité $(1, 0)$*, on peut trouver en biochimie :*

– *en biochimie l'interaction* Muter *qui modifie la structure génétique de l'entité source en fonction d'une probabilité ;*
– *en biochimie l'interaction* Mitose *qui divise l'entité en deux entités dans l'environnement.*

Puisque ces interactions sont retrouvée fréquemment en simulation, elles sont par la suite désignées par le terme *interaction dégénérée.*

Définition 8. ***Interaction dégénérée***
Une **interaction dégénérée** est une interaction dont la cardinalité est $(1, 0)$.

Les interactions telles que $card_T(\mathcal{I}) = n$ **ou** $card_S(\mathcal{I}) = m, \forall n > 1, \forall m > 1$**.** La complexité de certains problèmes à simuler nécessite la prise en compte d'interactions mettant en scène plus de deux entités et en particulier plus d'une seule cible.

Exemple.

Exemples d'interactions de cardinalité $(1, n), \forall n > 1$ *:*

– *l'interaction* Catalyser *en biochimie, qui a pour cardinalité* $(1, 2)$*. Dans cette interaction, la source (le plus souvent une enzyme) favorise une réaction chimique ayant lieu entre deux autres entités moléculaires ou plus (les cibles) ;*
– *l'interaction* Diffuser *en éthologie, qui a pour cardinalité* $(1, 8)$*. Cette interaction suppose que l'environnement est pavé par une grille d'entités, qui représentent chacune la concentration d'une phéromone dans une parcelle de l'environnement. Elle est exécutée par une entité source afin de diffuser une partie des phéromones qu'elle contient dans les parcelles de l'environnement à proximité.*

Dans d'autres cas, il se révèle même nécessaire d'impliquer plus d'une seule source. De telles interactions dénotent le plus souvent un problème de coordination entre entités.

Exemple.

Exemples d'interactions de cardinalité $(m, n), \forall m > 1, \forall n \geq 0$ *:*

– *l'interaction* Se Reproduire *en éthologie, qui a pour cardinalité* $(2, 0)$*. Dans cette interaction, deux sources ajoutent à l'environnement leur descendance. Dans cette interaction, il n'y a pas de cible puisque la participation de chaque entité est censée provenir du processus décisionnel de chacune des deux sources ;*
– *l'interaction* Transporter à 4 *en logistique, qui a pour cardinalité* $(4, 1)$*. Dans cette interaction, quatre sources doivent coopérer afin de transporter une entité trop lourde pour être transportée par 3 entités ou moins.*

En pratique, les interactions de cardinalité (m, n) ne sont pas utilisées. En effet, seule l'une des m sources décide d'initier l'interaction. Les autres sources se contentent d'accepter ou de refuser d'y participer. En pratique ce type d'interaction s'exprime donc avec une cardinalité $(1, m + n - 1)$.

L'utilisation d'interactions de cardinalité $(1, n)$ et (m, n) est principalement liée à des problèmes de granularité de la représentation des connaissance des entités. Nous faisons le choix de restreindre la cardinalité des interactions pouvant être spécifiées aux interactions dégénérées et individuelles. Ce choix ne restreint pas de manière significative les simulations pouvant être spécifiées avec notre approche. En effet, nous soutenons dans cette thèse que ces interactions peuvent dans la majorité des cas être décomposées en un ensemble d'interactions dégénérées et individuelles. Cette affirmation est justifiée dans la section 6.1 page 161 du chapitre 6.

3.2.4 Définition formelle d'une interaction

Comme nous l'avons mentionné précédemment, chaque entité ne joue pas la même fonction dans une interaction. Par conséquent, elles ne doivent pas implémenter les mêmes primitives. Nous caractérisons les primitives devant être implémentées par les entités participant à l'interaction par la notion de *signature d'une entité dans l'interaction.*

Signatures

La *signature d'une entité dans l'interaction* permet de caractériser les primitives d'action et de perception manipulées dans les préconditions, le déclencheur et les actions de l'interaction. Elle permet de plus d'identifier l'ensemble des primitives abstraites qu'une entité doit spécifier à l'aide d'algorithmes pour participer à une interaction. La définition qui suit est illustrée dans la figure 3.4.

Définition 9. ***Signature d'une entité dans une interaction***
La **signature** S **d'une entité dans une interaction** est définie par l'ensemble $S = \langle id, (p_i)_{i\in\mathbb{N}}\rangle$ où p_i est une primitive de perception ou d'action et id est un identifiant décrivant explicitement la fonction de l'entité dans l'interaction. On note par la suite $primitives(S)$ l'ensemble des primitives de la signature S et $id(S)$ l'identifiant de S.
On dit qu'une entité **se conforme** à une signature si elle fournit une spécification à chacune des primitives abstraites qu'elle contient.

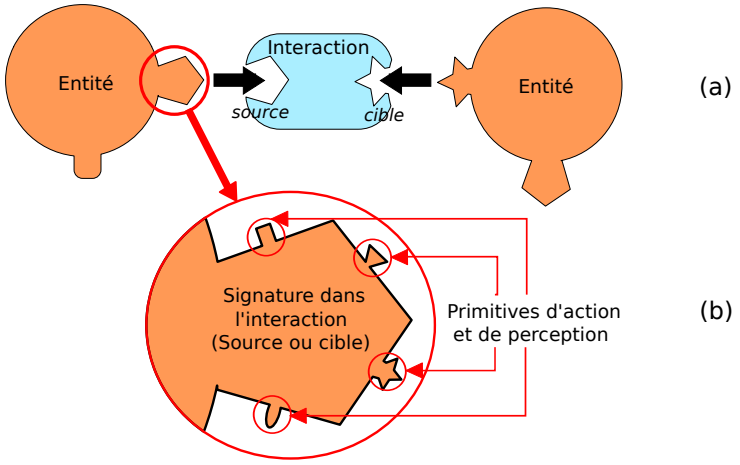


FIGURE 3.4 – Schéma illustrant le lien entre les primitives abstraites, les interactions et la signature d'une entité dans une interaction.
(a) Chaque entité (disque orange) se conforme à un ensemble de signatures (excroissances oranges). Symétriquement, chaque interaction (rectangle bleu) définit une signature d'entité à laquelle doit se conformer sa source (cavité en forme de pentagone) et une signature d'entité à laquelle doit se conformer sa cible (cavité en forme d'étoile). Afin de pouvoir être la source (resp. la cible) d'une interaction et donc pouvoir initier (resp. subir) l'interaction, une entité doit se conformer à la signature de la source (resp. de la cible) dans l'interaction. Par conséquent, une interaction entre une entité source et une entité cible n'est possible que si la source peut initier l'interaction et la cible peut la subir.
(b) Dans le détail, chaque signature décrit en fait un ensemble de primitives manipulées dans l'interaction. Elles doivent être spécifiées par toutes les entités se conformant à la signature.

La notion de signature permet alors de caractériser si une entité peut participer à une interaction. En effet, chaque interaction individuelle définit une signature d'entité à laquelle doit se conformer sa source ainsi qu'une signature d'entité à laquelle doit se conformer sa cible. On considère qu'une entité **peut initier** une interaction si elle se conforme à la **signature de la source** dans l'interaction. De même, une entité **peut subir** une interaction si elle se conforme à la **signature de la cible** dans l'interaction. Par conséquent, une **interaction peut survenir** entre deux entités si l'une d'elles est **capable d'initier l'interaction** et si l'autre entité est **capable de la subir**.

Le nombre de signatures à définir pour une interaction $\mathcal{I}$ est directement lié à la cardinalité de cette dernière. Il est égal à $card_S(\mathcal{I}) + card_T(\mathcal{I})$. Toutefois, plusieurs entités impliquées simultanément dans une interaction peuvent y jouer un rôle symétrique et doivent donc implémenter les mêmes primitives. On considère alors qu'elles ont la même signature (ce point est illustré dans l'exemple ci-après). La signature d'une entité dans une interaction peut aussi ne contenir aucune primitive. Dans ce cas, l'entité ne doit

implémenter aucune primitive abstraite particulière pour pouvoir participer à l'interaction.

Exemple.
Dans la figure 3.3b, l'interaction MANGER *est une interaction de cardinalité* $(1,1)$ *définissant par conséquent deux signatures :*

- *Une signature liée à l'entité* $\mathcal{X}$ *qui effectue l'interaction. Elle contient une primitive de perception* $\langle aFaim, booleen, \emptyset\rangle$ *et une primitive d'action* $\langle diminuerFaim, \emptyset, \{Entite\}\rangle$*. Par exemple :*

$$\big\langle sourceDeManger, \{\langle aFaim, booleen, \emptyset\rangle, \langle diminuerFaim, \emptyset, \{Entite\}\rangle\}\big\rangle$$

- *Une signature liée à l'entité* $\mathcal{Y}$ *qui subit l'interaction. Elle contient une primitive de perception* $\langle estSain, booleen, \emptyset\rangle$*. Par exemple :*

$$\big\langle cibleDeManger, \{\langle estSain, booleen, \emptyset\rangle\}\big\rangle$$

L'interaction TRANSPORTER À 4 *a pour cardinalité* $(4,1)$*, mais n'a pourtant à définir que deux signatures :*

- *Une signature liée à l'entité transportée (i.e. la cible de l'interaction) ;*
- *Une signature liée aux entités effectuant le transport (i.e. les sources de l'interaction). A priori, ces 4 entités jouent un rôle symétrique dans l'interaction et implémentent donc les mêmes primitives abstraites. Elles ont donc la même signature dans l'interaction.*

Le déclencheur, les préconditions et les actions d'une interaction peuvent aussi être amenés à manipuler des primitives de l'environnement. Par conséquent, une interaction est non seulement caractérisée par la signature des entités, mais aussi par la signature de l'environnement.

Définition 10. ***Signature de l'environnement dans une interaction***
La **signature de l'environnement dans une interaction** est définie par l'ensemble $\langle (p_i)_{i\in\mathbb{N}}\rangle$ où p_i est une primitive de l'environnement.

La signature de l'environnement a un rôle primordial au sein d'une bibliothèque d'interactions. En effet, elle caractérise la topologie de l'environnement dans lequel chaque interaction peut avoir lieu. **Une interaction ne peut avoir lieu dans un environnement que si ce dernier se conforme à la signature de l'environnement dans l'interaction.**

Définition d'une interaction

Pour des raisons pratiques liées à la spécification des préconditions, du déclencheur et des actions d'une interaction, nous associons un nom à chaque entité participant à l'interaction. Nous associons de plus à chaque nom la signature de l'entité lui correspondant dans l'interaction ainsi qu'une valeur dans $\{Source, Cible\}$ qui détermine si cette entité est une source ou une cible de l'interaction.

Nous définissons une interaction dans IODA de la manière suivante, dont un métamodèle est fourni dans la figure 3.5 :

Définition 11. ***Interaction***
Une **interaction** est un bloc sémantique décrivant comment et sous quelles conditions des entités peuvent interagir ensemble.
Il s'agit d'un 8-uplet $\mathcal{I} = \langle id, noms, cardinalite, signature, sign_{env}, preconditions, declencheur, actions \rangle$ tel que :
- id est l'identifiant de l'interaction représenté sous la forme d'une chaîne de caractères. Il exprime de manière explicite ce à quoi correspond l'interaction. Cet identifiant est supposé être unique ;
- $noms$ est un ensemble de chaînes de caractères représentant le nom attribué à chaque entité participant à l'interaction ;
- $cardinalite$ est une fonction prenant en paramètre un nom et retournant une valeur dans $\{Source, Cible\}$. Cette fonction détermine si l'entité caractérisée par un nom est une source ou une cible de l'interaction. Par conséquent, $card_S(\mathcal{I}) = |cardinalite(nom) = Source|_{\forall nom}$ et $card_T(\mathcal{I}) = |cardinalite(nom) = Cible|_{\forall nom}$
- $signature$ est une fonction associant à chaque élément de $noms$ la signature d'entité qui lui correspond ;
- $sign_{env}$ est la signature de l'environnement dans l'interaction. Elle définit les primitives devant être fournies par l'environnement pour que l'interaction puisse y avoir lieu ;
- $preconditions$ décrit les préconditions de l'interaction, *i.e.* les pré-requis physiques ou logiques nécessaires à l'exécution de l'interaction ;
- $declencheur$ décrit le déclencheur de l'interaction, *i.e.* les buts implicites ou explicites auxquels l'interaction répond ;
- $actions$ décrit les actions, *i.e.* les effets de l'interaction.

Notation 2.
On pose $\mathbb{I}_{(1,1)}$ l'ensemble des interactions individuelles d'une simulation et $\mathbb{I}_{(1,0)}$ l'ensemble de ses interactions dégénérées.

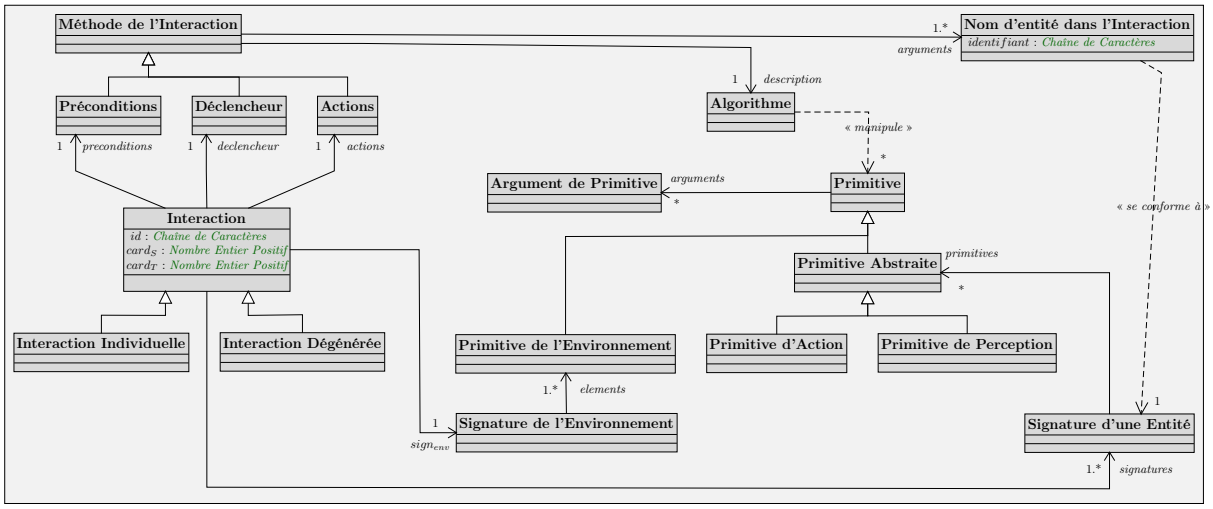


FIGURE 3.5 – Métamodèle général d'une interaction dans le modèle IODA

Dans ce chapitre, nous nous plaçons uniquement dans le cas d'interactions dont la cardinalité est soit $(1,1)$, soit $(1,0)$. Par conséquent, nous désignons les entités par les noms *source* et *cible* dans l'interaction, dont les signatures respectives sont $sign_{source}(\mathcal{I})$ et $sign_{cible}(\mathcal{I})$.

Préconditions, actions et déclencheur

Les préconditions, actions et déclencheur d'une interactions sont exprimées par des algorithmes manipulant les entités de l'interaction au travers de primitives. L'appel à une primitive particulière d'une

entité dans les préconditions, le déclencheur ou les actions de l'interaction se fait au travers du nom de l'entité dans l'interaction et de l'identifiant de la primitive appelée. La description de ces appels se fait avec une syntaxe particulière, que nous introduisons dans la notation qui suit.

Notation 3.
Dans les algorithmes décrivant les préconditions, le déclencheur ou les actions d'une interaction, l'appel à une primitive dont l'identifiant est "identifiant_primitive" d'une entité dont le nom est "nom_entité" dans l'interaction se fait avec la syntaxe :

nom_entité**.**identifiant_primitive(...)

Dans cette expression "..." est à remplacer par les paramètres utilisés pour cette primitive.

L'appel à une primitive de l'environnement se fait de la même manière en utilisant le nom "Environnement".

***Exemple. Syntaxe de la description de l'interaction* MANGER**
Considérons l'interaction MANGER *telle que décrite dans la figure 3.3 page 71. Cette interaction a pour cardinalité* $(1,1)$ *et implique donc deux entités nommées respectivement « Source » et « Cible ». La description des préconditions, déclencheur et actions de cette interaction est alors :*

Déclencheur	*Source.aFaim()*
Préconditions	*Cible.estSain()*
Actions	*Source.diminuerFaim(Cible)* *Environnement.retirer(Cible)*

3.3 Un approche centrée sur les interactions

Afin de déterminer les interactions que les entités sont capables d'initier ou de subir et donc définir le graphe des interactions entre entités, l'approche IODA repose sur une *matrice d'interaction brute* (voir figure 3.6). Chaque ligne de cette matrice représente les interactions qu'une entité peut initier (*i.e.* les interactions pour lesquelles l'entité est la source) et chaque colonne les interactions qu'une entité peut subir (*i.e.* les interactions pour lesquelles l'entité est la cible). Par conséquent, l'intersection d'une ligne et d'une colonne définit les interactions que deux entités sont capables d'effectuer ensemble.

Source \ Cible	∅	Herbe	Herbivore	Carnivore
Herbe	Pousser			
Herbivore	MOURIR SE DÉPLACER	(MANGER, d=0cm)	(SE REPRODUIRE, d=30cm)	
Carnivore	MOURIR SE DÉPLACER		(MANGER, d=30cm)	(SE REPRODUIRE, d=30cm)

FIGURE 3.6 – Structure d'une matrice d'interaction brute, illustrée sur l'exemple de la simulation d'un écosystème contenant proies et prédateurs. L'élément *(*MANGER*, d=0cm)* situé à l'intersection de la ligne *Herbivore* et de la colonne *Herbe* se lit « Une entité Herbivore a la capacité d'initier l'interaction MANGER avec une entité Herbe pour cible si cette dernière se situe au plus à une distance de 0cm de la source ».

La définition formelle de la matrice d'interaction repose sur trois notions différentes : la *garde de distance*, les *éléments d'assignation* ainsi que les *assignations*.

3.3.1 Garde de distance

Une interaction individuelle ne peut avoir lieu entre deux entités que si elles sont suffisamment *proches*. La notion de proximité n'est pas propre aux simulations se déroulant dans un espace euclidien en deux ou trois dimensions : elle peut être exprimée dans tout type d'espace topologique pouvant exprimer une

distance telle qu'une différence de richesse entre deux entités ou la distance entre deux nœuds dans un graphe.

Pour les mêmes raisons ayant conduit à l'utilisation de primitives abstraites dans les préconditions, les actions et le déclencheur d'une interactions, la garde de distance n'est pas propre à une interaction individuelle.

Exemple.
Un herbivore initie l'interaction MANGER *au contact de végétaux (i.e. à une distance de 0), alors qu'un caméléon peut initier cette interaction à une distance plus grande (de 10 à 70 cm).*

En conséquence, une *garde de distance* est associée à chaque interaction individuelle présente dans la matrice d'interaction.

Définition 12. ***Garde de distance***
La **garde de distance** d'une interaction individuelle dans la matrice d'interaction brute est un entier correspondant à la distance maximale en deçà de laquelle l'interaction peut être initiée.

La garde de distance est spécifiée indépendamment de la description de l'interaction, afin de garantir un plus grand polymorphisme. Elle est définie lors de l'assignation de l'interaction à des entités, lorsque la *matrice d'interaction brute* est construite.

3.3.2 Matrice d'interaction brute

La *matrice d'interaction brute* exprime l'ensemble des interactions pouvant avoir lieu entre les entités d'une simulation en fonction de la famille à laquelle elles appartiennent. Elle constitue ainsi une représentation synthétique du graphe liant les entités interagissant ensemble. Nous appelons *assignation* chaque cellule de cette matrice. Une assignation décrit l'ensemble des interactions pouvant être initiées par une entité source avec une entité cible particulières. Ces assignations contiennent chacune des *éléments d'assignation*. Un élément d'assignation représente une interaction qu'une entité source a la capacité d'initier avec une entité cible.

Définition 13. ***Élément d'assignation***
Un **élément d'assignation** est la représentation d'une interaction qu'une entité source est capable d'initier. Il peut représenter :
- une interaction individuelle $\mathcal{I}$ qu'une entité source de la famille $\mathcal{F}_S$ est capable d'initier avec une entité cible de la famille $\mathcal{F}_T$. Cette interaction ne peut être initiée que si la distance séparant ces entités est inférieure ou égale à la **garde de distance**. On parle alors d'**élément d'assignation individuel**. Il est décrit sous la forme du n-uplet $(\mathcal{I}, dist, \mathcal{F}_S, \mathcal{F}_T) \in \mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}$ où $dist$ est une garde de distance ;
- une interaction dégénérée $\mathcal{I}$ qu'une instance de la famille d'entités $\mathcal{F}_S$, est capable d'initier. On parle alors d'**élément d'assignation dégénéré**, décrit sous la forme du n-uplet $(\mathcal{I}, \mathcal{F}_S) \in \mathbb{I}_{(1,0)} \times \mathbb{F}$.

Notation 4.
On pose $\mathcal{I}(a)$ la fonction permettant de connaître l'interaction associée à un élément d'assignation, $dist(a)$ la fonction permettant d'en connaître la garde de distance, $source(a)$ la fonction permettant d'en connaître la famille d'entités source et *cible* la fonction permettant d'en connaître la famille d'entités cible.

Une *assignation* représente l'ensemble des interactions qu'une instance d'une famille d'entités peut initier avec une instance d'une famille d'entités cible. Par conséquent, une assignation constitue une cellule de la matrice d'interaction brute.

Définition 14. ***Assignation individuelle***
Une **assignation individuelle** $a_{\mathcal{S}/\mathcal{T}}$ est un ensemble d'éléments d'assignation individuels. Elle représente l'ensemble des interactions individuelles qu'une instance de la famille d'entités $\mathcal{S}$ peut initier en tant que source conjointement avec une instance de la famille d'entités $\mathcal{T}$ en tant que cible. Plus formellement, on a :

$$\begin{cases} \forall \mathcal{S} \in \mathbb{F}, \forall \mathcal{T} \in \mathbb{F}, a_{\mathcal{S}/\mathcal{T}} \subset \left(\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}\right) \\ \forall \mathcal{S} \in \mathbb{F}, \forall \mathcal{T} \in \mathbb{F}, a \in a_{\mathcal{S}/\mathcal{T}} \Leftrightarrow source(a) = \mathcal{S} \wedge cible(a) = \mathcal{T} \end{cases}$$

D'après la définition précédente, on sait qu'une entité a la capacité d'initier une interaction avec une autre entité comme cible s'il existe une cellule de la matrice d'interaction brute (*i.e.* une assignation) dans laquelle un élément d'assignation contient cette interaction. On a donc la propriété qui suit.

Propriété 1.
Soient $e \in \mathbb{E}, e' \in \mathbb{E}$ et $\mathcal{I} \in \mathbb{I}_{(1,1)}$

$$\begin{array}{c} e \text{ a la capacité d'initier } \mathcal{I} \text{ avec } e' \text{ pour cible} \Leftrightarrow \\ \exists \mathcal{S} \in \mathbb{F}, \exists \mathcal{T} \in \mathbb{F}, \exists a \in a_{\mathcal{S}/\mathcal{T}} | e \prec \mathcal{S} \wedge e' \prec \mathcal{T} \wedge \mathcal{I}(a) = \mathcal{I} \end{array}$$

Ces définitions peuvent aussi être appliquées au cas des interactions dégénérées.

Définition 15. ***Assignation dégénérée***
Une **assignation dégénérée** $a_{\mathcal{S}/\emptyset}$ est un ensemble d'éléments d'assignation dégénérés. Elle représente l'ensemble des interactions dégénérées qu'une instance de la famille d'entités $\mathcal{S}$ peut initier. Plus formellement, on a :

$$\begin{cases} \forall \mathcal{S} \in \mathbb{F}, a_{\mathcal{S}/\emptyset} \subset \mathbb{I}_{(1,0)} \times \mathbb{F} \\ \forall \mathcal{S} \in \mathbb{F}, a \in a_{\mathcal{S}/\emptyset} \Leftrightarrow source(a) = S \end{cases}$$

Propriété 2.
Soient $e \in \mathbb{E}$ et $\mathcal{I} \in \mathbb{I}_{(1,0)}$

$$e \text{ a la capacité d'initier } \mathcal{I} \Leftrightarrow \exists \mathcal{S} \in \mathbb{F}, \exists a \in a_{\mathcal{S}/\emptyset} | e \prec \mathcal{S} \wedge \mathcal{I}(a) = \mathcal{I}$$

On appelle alors *matrice d'interaction brute* l'ensemble de toutes les assignations pour une simulation. Le métamodèle issu de la définition qui suit est présenté sur la figure 3.7.

Définition 16. ***Matrice d'interaction brute***
Une **matrice d'interaction brute** $\mathcal{M}$ représente toutes les interactions pouvant survenir entre les différentes entités que contient l'environnement. Elle constitue l'agrégation de toutes les assignations pour la simulation. Plus formellement, on a :

$$\mathcal{M} : \begin{array}{lcl} \mathbb{F} \times \left(\mathbb{F} \cup \{\emptyset\}\right) & \rightarrow & \mathcal{P}\Big(\left(\mathbb{I}_{(1,0)} \times \mathbb{F}\right) \cup \left(\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}\right)\Big) \\ (\mathcal{S}, \mathcal{T}) & \rightarrow & a_{\mathcal{S}/\mathcal{T}} \end{array}$$

Une illustration d'une telle matrice est présentée sur la figure 3.6 pour une simulation d'un écosystème contenant de l'herbe, des herbivores et des carnivores. Cet exemple constitue un modèle simple de type proie/prédateur où de l'herbe pousse, des herbivores se déplacent dans l'environnement, se reproduisent avec d'autres herbivores, mangent de l'herbe et meurent s'ils n'ont pas assez mangé. De plus des carnivores peuvent s'y déplacer, s'y reproduire avec d'autres carnivores, y manger des herbivores et y mourir s'ils n'ont pas assez mangé.

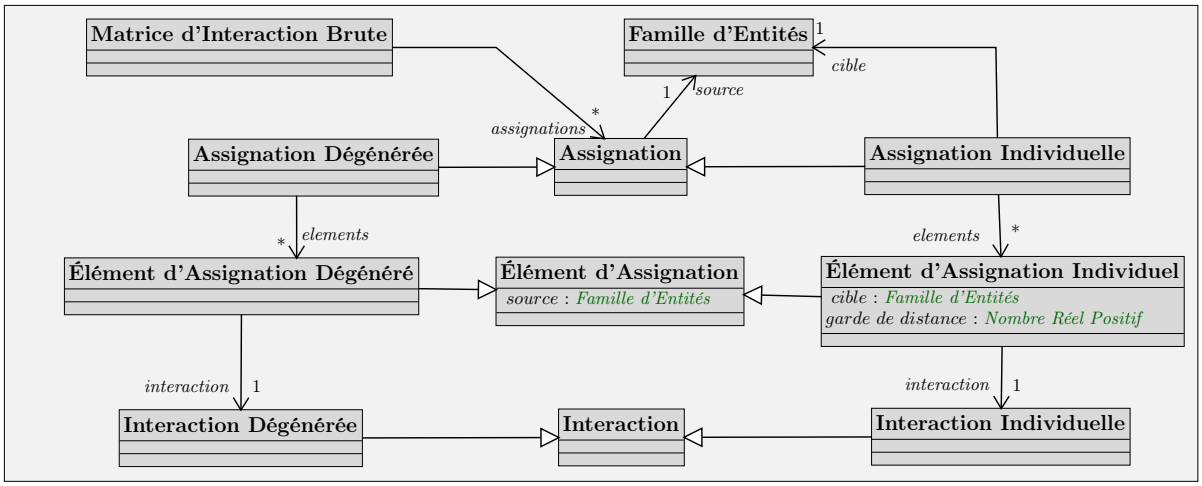


FIGURE 3.7 – Métamodèle d'une matrice d'interaction brute dans le modèle IODA

3.4 Les entités

Dans IODA, les entités d'une simulation sont représentées de manière homogène et leur comportement est régi par un processus générique schématisé dans la figure 3.8. Cette section s'articule selon les trois éléments figurant dans cette dernière : la mise à jour de l'état de l'entité, la perception des autres entités et la sélection de l'interaction initiée. Une version détaillée des algorithmes correspondant à cette figure sont décrits dans la section 4.3 page 119 du chapitre 4.

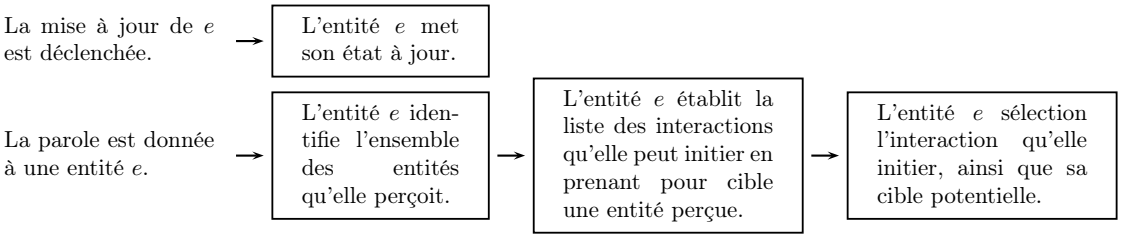


FIGURE 3.8 – Principes régissant l'évolution et le comportement d'une entité dans l'approche IODA. Ces principes se décomposent en deux parties. La première (partie supérieure) consiste à mettre à jour l'état de l'entité indépendamment des interactions qu'elle initie ou qu'elle subit. La seconde (partie inférieure) décrit le comportement de l'entité. Il consiste à recenser les entités lui étant voisines, identifier les interactions pouvant être initiées avec une entité voisine et enfin sélectionner puis initier une interaction parmi ces dernières.

3.4.1 Mise à jour

Une entité n'a pas besoin d'initier ou subir une interaction pour que son état évolue. Certaines caractéristiques peuvent évoluer en dehors de tout comportement. C'est par exemple le cas de l'âge, puisqu'une entité vieillit indépendamment de toute action qu'elle entreprend. Il en va de même pour la sensation de satiété dans une simulation de type proie/prédateur qui diminue progressivement au fil de la simulation.

Puisque la matrice d'interaction brute a pour rôle de décrire ce que les entités sont capables de faire, la mise à jour des entités ne peut y être exprimée. Nous choisissons toutefois d'exprimer la mise à jour par un moyen similaire que nous appelons *matrice de mise à jour*, afin d'unifier la représentation des connaissances des entités. Ce choix permet de plus de profiter des avantages de notre formalisme autant dans la description du comportement des entités que dans la description de leur mise à jour. En effet, les

éléments figurant dans la matrice de mise à jour sont représentés de la même manière que les éléments d'assignation dégénérés et donc à l'aide d'interactions dégénérées.

Définition 17. ***Matrice de mise à jour***
La **Matrice de mise à jour** $\mathcal{U}$ exprime quelles actions sont exécutées afin de mettre à jour l'état des entités d'une simulation. Elle est constituée d'assignations dégénérées. Plus formellement, si $\mathcal{P}$ est l'ensemble des partitions d'un ensemble, on a :

$$\mathcal{U} \ : \ \begin{array}{ccc} \mathbb{F} & \rightarrow & \mathcal{P}(\mathbb{I}_{(1,0)} \times \mathbb{F}) \\ \mathcal{S} & \rightarrow & (u^i_{\mathcal{S}}, \mathcal{S})_{i \in \mathbb{N}} \end{array}$$

La mise à jour d'une entité consiste à initier l'ensemble des interactions dégénérées qui lui correspondent dans la matrice de mise à jour. Par abus de langage, nous désignons par « interactions de mise à jour » les interactions dégénérées présentes dans la matrice d'interaction. L'avantage de cette matrice est de rendre la déclaration de la mise à jour des agents à la fois modulaire et facile à modifier.

L'ordre dans lequel les interactions de mise à jour sont initiées est important. En effet, ces interactions peuvent avoir des effets différents selon l'ordre dans lequel elles sont initiées.

Exemple.
Considérons une simulation d'un écosystème de type proie/prédateur où le comportement de prédateurs est basé sur leur propre santé, représentée par une quantité d'énergie (un nombre entier). Lorsque cette dernière passe sous un certain seuil, l'entité ressent la sensation de faim, ce qui l'amène à déclencher l'interaction Manger *développée dans les sections précédentes. Dans cet exemple, nous considérons que l'effet de l'interaction* Manger *n'est pas immédiat : l'énergie de l'entité augmente progressivement de manière naturelle au fil de sa digestion. De plus, dans tous les cas, la santé d'une entité décroit de manière inversement proportionnelle à la quantité d'énergie de l'entité. Dans ce cadre, deux interactions de mise à jour sont définies :*

- Diminuer Satiété*, qui consiste a diminuer la sensation de satiété de l'entité. Dans cet exemple, cette interaction consiste à retrancher à l'énergie de l'entité un montant inversement proportionnel à sa quantité d'énergie actuelle ;*
- Digérer*, qui consiste à poursuivre l'assimilation des éléments ingérés par l'entité. Dans cet exemple, cette interaction consiste à ajouter à l'énergie de l'entité une valeur dépendant des entités ayant été ingérées.*

Il apparaît alors évident que la valeur de l'énergie de l'entité sera différente selon l'ordre d'exécution des interactions de mise à jour Diminuer Santé *et* Digérer *: si x est la quantité d'énergie de l'entité, m la quantité d'énergie ajoutée par l'interaction* Digérer *et k la proportion d'énergie perdue lors de l'interaction* Diminuer Santé*, alors la quantité d'énergie de l'entité à la fin de la mise à jour sera dans un cas $x + m - \frac{k}{x}$, et dans l'autre $x + m - \frac{k}{x+m}$.*

Puisque l'ordre d'exécution des interactions de mise à jour a une influence sur les résultats obtenus, l'aspect déclaratif de la matrice de mise à jour doit être complété afin d'éviter tout biais lors de l'implémentation. Pour remédier à ce problème, nous choisissons d'établir une relation d'ordre entre les interactions présentes dans la matrice de mise à jour. Cet ordre est obtenu en attribuant une priorité à chaque élément présent dans la matrice de mise à jour, aboutissant à la description d'une *matrice de mise à jour ordonnée.*

Définition 18. ***Matrice de mise à jour ordonnée***
La **Matrice de mise à jour ordonnée** $\mathcal{U}_{ord}$ exprime comment a lieu la mise à jour des entités d'une simulation. Elle consiste à attribuer à chaque élément de la matrice de mise à jour une priorité représentée par un nombre entier. Plus ce nombre est élevé, plus l'interaction de mise à jour est prioritaire et est exécutée tôt lors de la mise à jour. Plus formellement, si $\mathcal{P}$ est l'ensemble des partitions d'un ensemble, on a :

$$\begin{array}{rccl} & \mathbb{I}_{(1,0)} \times \mathbb{F} & \rightarrow & \mathbb{Z} \\ \mathcal{U}_{ord} : & (u) & \rightarrow & \begin{cases} \mathcal{U}_{ord}(u) & \text{si } u \in \mathcal{U}(\mathcal{S}) \\ \text{non défini} & \text{sinon} \end{cases} \end{array}$$

On note $\mathcal{U}_{ord}^{-1}$ l'application permettant de connaître l'ensemble des interactions de mise à jour ayant une priorité donnée pour une famille d'entités $\mathcal{F}$:

$$\begin{array}{rccl} \mathcal{U}_{ord}^{-1} : & \mathbb{F} \times \mathbb{Z} & \rightarrow & \mathcal{P}(\mathbb{I}_{(1,0)} \times \mathbb{F}) \\ & (\mathcal{F}, p) & \rightarrow & \{u \in \mathcal{U}(\mathcal{F}) | \mathcal{U}_{ord}(u) = p\} \end{array}$$

Un métamodèle de la matrice de mise à jour ordonnée est proposé dans la figure 3.9.

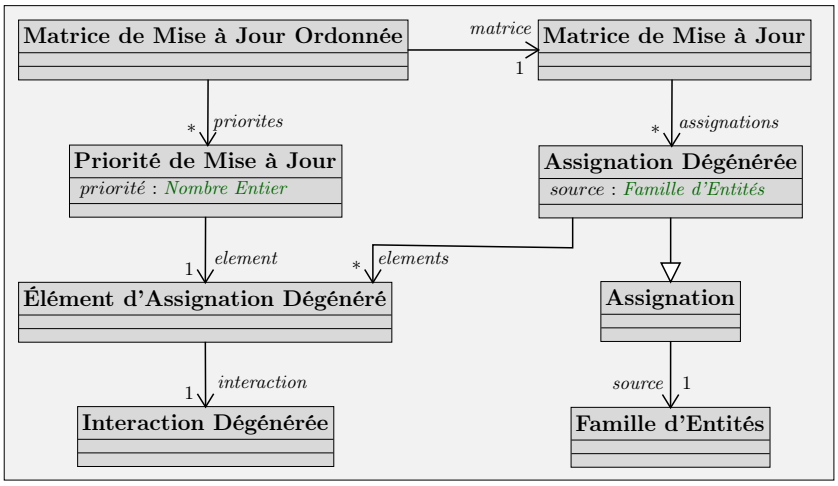


FIGURE 3.9 – Métamodèle d'une matrice de mise à jour ordonnée dans le modèle IODA

3.4.2 Perception

L'un des principes de base des systèmes multi-agents est que les entités ne sont pas omniscientes et n'interagissent qu'avec les entités qu'elles perçoivent. Il en va de même dans IODA, où la perception d'une entité est fondée sur le concept de *halo de perception*.

Le halo de perception d'une famille d'entités $\mathcal{F} \in \mathbb{F}$ est une fonction qui détermine si une entité $e' \in \mathbb{E}$ est perçue par une entité $e \prec \mathcal{F}$. Tout comme le déclencheur, les préconditions et les actions d'une entité, le halo est décrit sous la forme d'un algorithme manipulant à la fois des primitives de l'environnement et des primitives abstraites des entités. Par conséquent une famille d'entités a une signature pour son halo. Il en va de même pour l'environnement. Ces informations aboutissent à la définition du halo qui suit.

Définition 19. ***Halo de perception d'une famille entité***
Le **Halo de perception** $\mathcal{H}(\mathcal{F})$ d'une famille d'entités $\mathcal{F} \in \mathbb{F}$ permet de déterminer si une entité e' est perçue par une entité $e \prec \mathcal{F}$. Il est représenté par un triplet $\langle halo, sign_{entite}, sign_{env} \rangle$ où :

- $halo : \begin{array}{ccc} \mathbb{E} \times \mathbb{E} & \rightarrow & \{vrai, faux\} \\ (e, e') & \rightarrow & \begin{cases} vrai & \text{si } e' \text{ est perçue par } e \\ faux & \text{si } e' \text{ n'est pas perçue par } e \end{cases} \end{array}$
- $sign_{entite}$ est la signature de la famille d'entités pour son halo. Elle détermine l'ensemble des primitives abstraites que la famille d'entités doit spécifier pour utiliser son propre halo ;
- $sign_{env}$ est la signature de l'environnement pour le halo de la famille d'entités $\mathcal{F}$. Elle détermine l'ensemble des primitives de l'environnement devant être spécifiées pour que le halo $\mathcal{H}(\mathcal{F})$ puisse être utilisé.

Exemple. Halo manipulant des primitives de perception
Soit un halo désignant une entité e_2 comme perçue par e si elle figure dans la mémoire de l'entité e ou si elle si situe à une distance inférieure ou égale à l'acuité visuelle e. Il s'exprime sous la forme suivante :

halo$\Big(\mathcal{H}\big($**famille(e)**$\big)\Big)(e,e_2)$ *:*
| **retourner** $e.aDansSaMemoire(e_2)$ **ou** $environnement.distance(e, e_2) \leq e.acuiteVisuelle()$

Il implique l'ajout de deux primitives de perception dans la famille de l'agent e : $\langle aDansSaMemoire, booleen, \{Entite\} \rangle$ et $\langle acuiteVisuelle, Nombre\ a\ Virgule\ Flottante, \emptyset \rangle$.

Exemple. Halo ajoutant une primitive de l'environnement
Soit un halo désignant une entité e_2 comme perçue par e si elle ne se situe pas derrière une autre entité de la simulation. Il s'exprime sous la forme suivante :

halo$\Big(\mathcal{H}\big($**famille(e)**$\big)\Big)(e,e_2)$ *:*
| **retourner** $environnement.aucuneEntiteOpaqueEntre(e, e_2)$

Il implique l'ajout d'une primitive de l'environnement qui vérifie qu'aucune entité opaque ne se situe entre e et e_2 $\langle aucuneEntiteOpaqueEntre, booleen, \{Entite, Entite\} \rangle$. Si cette primitive retourne faux, on peut considérer que e_2 se situe derrière une autre entité.

Le voisinage d'une entité $e \in \mathbb{E}$ constitue l'ensemble des entités perçues par e. Il représente l'ensemble des entités avec lesquelles e est susceptible d'interagir et est construit à l'aide du halo de la famille de e.

Définition 20. ***Voisinage d'une entité***
Le **Voisinage** $\mathcal{V}(e)$ d'une entité e correspond à l'ensemble des entités perçues par e. Plus formellement, on a :

$$\mathcal{V} : \begin{array}{ccc} \mathbb{E} & \rightarrow & \mathcal{P}(\mathbb{E}) \\ e & \rightarrow & \{e' \in \mathbb{E} | \mathcal{H}\big(famille(e)\big)(e') == vrai \wedge e \neq e'\} \end{array}$$

Il est important de noter que le halo ne se limite pas à une proximité spatiale : une entité mémorisée peut aussi faire partie du voisinage d'une entité.

3.4.3 Sélection de l'interaction initiée

La sélection de l'interaction initiée par une entité est un processus consistant à choisir selon des modalités variables un couple « interaction individuelle/entité du voisinage » ou une interaction dégénérée afin d'en initier les actions. Les éléments déclaratifs permettant de décrire un tel processus dépendent grandement de la nature du processus décisionnel de l'entité. Le modèle général de Ioda n'émet aucune hypothèse sur ce processus, qui peut autant être à planification délibérative, à planification réactive, purement réactive, *etc.*

Pour des raisons que nous évoquons ultérieurement dans la section 3.6, l'approche Ioda se concentre sur la spécification d'entités dont le **processus de sélection d'interaction est réactif**. Le modèle permettant de décrire de tels comportements est présenté dans la section 3.6.2. **Ce processus ne présume**

pas du degré de cognition utilisé pour exprimer les primitives abstraites des entités, qui peuvent autant être réactives que cognitives.

Exemple. Expression de différents degrés de cognition dans une primitive
Dans une supermarché virtuel, un `Client` *ne peut initier une interaction* RAMASSER *prenant pour cible une entité de famille* `Article` *que si la source a envie de ramasser la cible. Cela se traduit dans l'interaction par l'usage d'une primitive abstraite* $\langle estInteresse, booleen, \{Entite\}\rangle$ *dans le déclencheur de l'interaction.*

Différents degrés de cognition peuvent alors être exprimés :

- *Une entité purement réactive posséderait par exemple une liste de courses. La primitive* $S.estInteresse(T)$ *retournerait alors la valeur «* $T \in S.listeCourses$ *».*
- *Une entité plus cognitive se baserait par exemple sur de l'inférence, des croyances ou des connaissances. Dans ce cas, la primitive* $S.estInteresse(T)$ *pourrait retourner la valeur «* $S.sait("T \in S.listeCourses")$ *ou* $S.croit("UTILE(T)")$ *ou* $(S.sait("AIME(T)")$ *et non* $S.croit("GRAS(T)"))$ *», avec :*
  - $\langle sait, booleen, \{Chaîne\ de\ Caracteres\}\rangle$ *une primitive définissant si le prédicat donné en paramètres fait partie des connaissances de l'entité ;*
  - $\langle croit, booleen, \{Chaîne\ de\ Caracteres\}\rangle$ *une primitive définissant si le prédicat donné en paramètres fait partie des croyances de l'entité.*

Le **polymorphisme permet de spécifier des entités aux capacités de cognition variables à l'aide d'un seul et même formalisme**. Les interactions peuvent ainsi être **spécifiées indépendamment du degré de cognition des entités**. Cette propriété contribue à l'aspect transversal de l'approche IODA, puisque les interactions peuvent être spécifiées sans connaître la nature réactive ou cognitive des agents.

3.4.4 Définition formelle d'une famille d'entités

Lorsqu'une entité a la capacité de participer à une interaction, elle doit se conformer à une des signatures d'entité de cette interaction et par conséquent spécifier les primitives abstraites y figurant. Cette spécification se fait à l'aide d'algorithmes dont nous ne contraignons pas l'expression dans ce modèle formel. À l'instar des préconditions, du déclencheur et des actions d'une interaction, ce choix est fait afin de ne pas contraindre l'ingénierie logicielle dans leur expression. On peut toutefois constater qu'en règle générale ces primitives manipulent l'état des entités, que nous caractérisons dans IODA par un ensemble d'*attributs*.

Le métamodèle d'une famille et d'une instance d'entité est fourni dans la figure 3.10.

Définition 21. ***Famille d'entités***
Une **Famille d'entités** $\mathcal{F}$ est une spécification abstraite d'un ensemble d'instances d'entités qui partagent toutes leurs primitives abstraites, capacité à interagir ainsi que leur comportement. Elle est caractérisée par un 8-uplet $\langle id, attributs, primitives, \mathcal{H}, \mathcal{U}_{ord}, ligne, colonne, selection\rangle$ où :
- id est l'identifiant de la famille d'entités représenté sous la forme d'une chaîne de caractères. Il exprime de manière explicite ce à quoi correspondent les entités instances de cette famille ;
- $attributs = \left(a_i\right)_{i\in\mathbb{N}}$ est l'ensemble des identifiants de chaque **attribut** de cette famille d'entités. Chaque élément a_i est représenté sous la forme d'une chaîne de caractères exprimant de manière explicite le rôle de cet attribut dans le fonctionnement interne de l'entité ;
- $primitives$ est l'ensemble des primitives abstraites spécifiées par l'entité. Elles sont spécifiées sous la forme d'algorithmes pouvant manipuler les attributs cette famille ;
- $\mathcal{H}$ est le halo des entités de cette famille. Il leur permet de percevoir les autres entités de l'environnement ;
- $\mathcal{U}_{ord}$ est l'ensemble ordonné des interactions de mise à jour des entités de cette famille. Il permet de mettre à jour l'état de l'entité sans que cela soit le fruit de leur comportement ;
- $ligne$ est la fonction permettant de connaître la ligne de la matrice d'interaction brute associée à la famille d'entités $\mathcal{F}$. Elle décrit les interactions que toute entité de cette famille est capable d'initier. On a :
$$ligne(\mathcal{F}) : \begin{array}{ccc} \mathbb{F}\cup\{\emptyset\} & \rightarrow & \mathcal{P}\Big(\left(\mathbb{I}_{(1,0)}\times\mathbb{F}\right)\cup\left(\mathbb{I}_{(1,1)}\times\mathbb{R}^+\times\mathbb{F}\times\mathbb{F}\right)\Big) \\ (\mathcal{T}) & \rightarrow & \mathcal{M}(\mathcal{F},\mathcal{T}) \end{array}$$
- $colonne$ est la fonction permettant de connaître la colonne de la matrice d'interaction brute associée à la famille d'entités $\mathcal{F}$. Elle décrit les interactions que toute entité de cette famille est capable de subir. On a :
$$colonne(\mathcal{F}) : \begin{array}{ccc} \mathbb{F} & \rightarrow & \mathcal{P}\Big(\mathbb{I}_{(1,1)}\times\mathbb{R}^+\times\mathbb{F}\times\mathbb{F}\Big) \\ (\mathcal{T}) & \rightarrow & \mathcal{M}(\mathcal{T},\mathcal{F}) \end{array}$$
- $selection$ est le modèle du processus utilisé afin de sélectionner les interactions initiées selon des modalités dépendant de la nature réactive/cognitive de l'entité.

La différence fondamentale existant entre famille d'entités et instance d'entités est qu'une famille d'entités n'associe aucune valeur à ses attributs et se contente de les identifier. Ainsi, une instance d'entité est caractérisée par tous les éléments définis dans une famille d'entités, auxquels s'ajoute une fonction permettant de connaître la valuation des attributs :

Définition 22. ***Instance d'entité***
Une **instance d'entité** $e \in \mathbb{E}$ correspond à une entité située dans l'environnement de la simulation. Elle est représentée par un couple $\langle famille(e), valeur_{attr}(e)\rangle$ tel que :
- $famille(e) \in \mathbb{F}$ est la famille de l'entité e ;
- $valeur_{attr}(e)$ est une fonction associant une valeur à chaque attribut défini dans la famille de e.

La valuation des attributs d'une entité doit être initialisée lors de la création de cette dernière. Une primitive particulière appelée *primitive d'initialisation* est définie dans ce but.

Définition 23. ***Primitive d'initialisation***
La **primitive d'initialisation** d'une famille d'entités $\mathcal{F} \in \mathbb{E}$ est une primitive décrivant comment initialiser la valuation des attributs d'une instance de cette famille d'entités. Elle a pour forme $\langle initialisation, \emptyset, \emptyset\rangle$.

Propriété 3.
Toute famille d'entités dispose d'une primitive d'initialisation :
$$\forall\mathcal{F}\in\mathcal{F}, \exists p_{init} = \langle initialisation, \emptyset, \emptyset\rangle | p_{init} \in primitives(\mathcal{F})$$

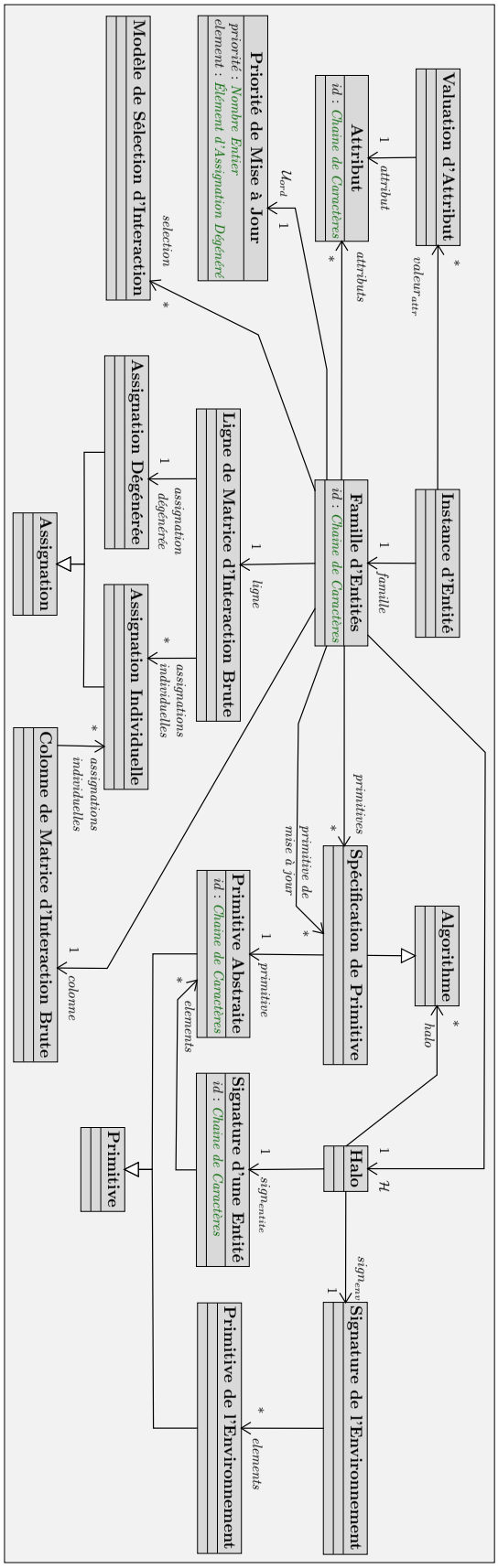


FIGURE 3.10 – Métamodèle d'une famille d'entité et d'une instance d'entité dans IODA.

3.5 L'environnement

D'après les définitions fournies dans la section 3.2.1, les interactions peuvent manipuler des primitives de l'environnement. Par conséquent, un environnement est défini par un ensemble de primitives.

Notation 5.
On note $primitives_{env}$ l'ensemble des primitives de l'environnement.

Le formalisme décrit dans ce chapitre est indépendant de la topologie de l'environnement Nous imposons seulement que l'environnement doive fournir une primitive permettant le calcul de la distance entre deux entités, des primitives permettant d'ajouter ou de retirer des entités ainsi qu'une primitive permettant de connaître l'ensemble des entités de la simulation. Cela se traduit par la propriété qui suit.

Propriété 4.
L'ensemble des primitives de l'environnement contient une primitive permettant de calculer la distance entre deux entités :

$$\langle distance, Nombre\ a\ Virgule\ Flottante, \{Entite, Entite\}\rangle \in primitives_{env}$$

Il contient de plus des primitives permettant d'ajouter ou de retirer des entités ainsi que de connaître l'ensemble des entités de la simulation. Ces primitives ne sont pas décrites ici, car leur forme est fortement dépendante de la topologie de l'environnement (voir exemples qui suivent).

Exemple. Ajout dans un environnement euclidien continu en deux dimensions
Dans le cadre d'une simulation en éthologie, l'environnement est usuellement représenté sous la forme d'un espace continu en deux dimensions dans lequel les entités sont assimilées à des points. Dans un tel environnement, l'ajout d'une entité se fait en précisant la position à laquelle elle est ajoutée. Par conséquent, la primitive de l'environnement ajoutant une entité prend la forme : $\langle ajouter, \emptyset, \{Entite, Nombre\ a\ Virgule\ Flottante, Nombre\ a\ Virgule\ Flottante\}\rangle$.

Exemple. Ajout dans un environnement discret
Dans le cadre de certaines simulations en sociologie, par exemple le modèle de ségrégation construit par Schelling [Sch71], l'environnement est représenté sous la forme d'un graphe. La position d'une entité dans l'environnement est alors un nœud du graphe. Par conséquent, la primitive de l'environnement ajoutant une entité prend la forme : $\langle ajouter, \emptyset, \{Entite, N\text{œ}ud\}\rangle$.

Ces primitives ne sont toutefois pas les seules que l'environnement ait à définir. En effet, chaque interaction a la capacité de percevoir des informations de l'environnement ou d'en modifier le contenu via des primitives de l'environnement. Par conséquent, une interaction ne peut être utilisée pour modéliser un phénomène que si elle est compatible avec l'environnement utilisé pour cette simulation, *i.e.* si l'environnement est conforme à la signature de l'environnement dans cette interaction. Ce critère permet de déterminer si une librairie d'interactions peut être utilisée pour modéliser un phénomène dans un environnement particulier. Cela se traduit par la propriété qui suit.

Propriété 5.
$\mathcal{I} \in \mathbb{I}_{(1,0)} \cup \mathbb{I}_{(1,1)}$ *peut etre utilisée pour modéliser un phénomène* $\Leftrightarrow sign_{env}(\mathcal{I}) \subseteq primitives_{env}$

Chaque primitive de l'environnement est spécifiée à l'aide d'algorithmes dont la structure est identique à celle de ceux spécifiant les préconditions, le déclencheur et les actions d'une interaction : une primitive de l'environnement peut manipuler d'autres primitives de l'environnement aussi bien que manipuler des primitives abstraites provenant des entités de la simulation. Ainsi, une primitive de l'environnement est caractérisée par une signature de l'environnement et une signature des entités de la simulation.

Définition 24. ***Primitive de l'environnement***
Une primitive de l'environnement est un triplet $p = \langle id, V, (a_i)_{i\in\mathbb{N}}, sign_{env}, sign_{entites}\rangle$ où :
- id est l'identifiant de la primitive représenté sous la forme d'une chaîne de caractères ;
- V est un élément optionnel décrivant le type de la valeur retournée par la primitive ;
- $(a_i)_{i\in\mathbb{N}}$ représente l'ensemble ordonné (éventuellement vide) des paramètres de la primitive ;
- $sign_{env}$ est la signature de l'environnement dans cette primitive, *i.e.* l'ensemble des primitives de l'environnement manipulées dans l'algorithme de cette primitive ;
- $sign_{entites}$ est la signature des entités dans cette primitive, *i.e.* l'ensemble des primitives abstraites des entités manipulées dans l'algorithme de cette primitive.

Exemple. Algorithme d'une distance en environnement euclidien
Dans le cadre de simulations en biochimie, l'environnement est souvent représenté sous la forme d'un espace continu en deux dimensions dans lequel évoluent des cellules. Dans de telles simulations, les cellules ont une épaisseur devant être prise en compte pour calculer la distance entre deux entités. La primitive correspondante peut alors avoir pour spécification :
distance(e_1,e_2) *:*

$scalaire \Leftarrow \Big(environnement.abscisse(e_1) - environnement.abscisse(e_2)\Big)^2$
$scalaire \Leftarrow scalaire + \Big(environnement.ordonnee(e_1) - environnement.ordonnee(e_2)\Big)^2$
retourner $\sqrt{scalaire} - environnement.epaisseur(e_1) - environnement.epaisseur(e_2)$

Dans cette primitive, on a :
- $sign_{env} = \left\{ \begin{array}{l} \langle abscisse, Nombre\ a\ Virgule\ Flottante, \{Entite\}\rangle, \\ \langle ordonnee, Nombre\ a\ Virgule\ Flottante, \{Entite\}\rangle, \\ \langle epaisseur, Nombre\ a\ Virgule\ Flottante, \{Entite\}\rangle \end{array} \right\}$
- $sign_{entites} = \emptyset$

Un environnement est aussi caractérisé par un ensemble d'attributs pouvant être manipulés au sein de ses primitives : la largeur d'un espace euclidien, le graphe des accointances entre entités, *etc.* Les attributs de l'environnement sont structurellement identiques aux attributs des entités : ils sont représentés par un identifiant et se voient associer une valeur qui évoluera au cours de la simulation.

En résumé, dans IODA, un environnement est caractérisé par la définition qui suit :

Définition 25. ***Environnement***
Un **environnement** est un espace métrique dans lequel figurent les agents. Il est caractérisé par les primitives de l'environnement qu'il spécifie et par les primitives abstraites que les entités doivent implémenter pour pouvoir y figurer. Il définit donc une **signature des entités** dans l'environnement. Plus formellement, un environnement est un tuple $\langle id, primitives_{env}, sign_{entites}, attributs, valeur_{attr}\rangle$ où :
- id est l'identifiant de l'environnement représenté sous la forme d'une chaîne de caractères. Cette chaîne exprime de manière explicite ce à quoi correspond l'environnement. Par exemple « environnement euclidien en trois dimensions » ;
- $primitives_{env}$ est l'ensemble des primitives de l'environnement spécifiées par cet environnement ;
- $sign_{entites}$ est l'ensemble des primitives abstraites devant être implémentées par toutes les entités de la simulation afin de pouvoir figurer dans cet environnement. On a donc $sign_{entites} = \bigcup_{p\in primitives_{env}} sign_{entites}(p)$;
- $attributs = \big(a_i\big)_{i\in\mathbb{N}}$ est l'ensemble des identifiants de chaque **attribut** de l'environnement. Chaque élément a_i est représenté sous la forme d'une chaîne de caractères, exprimant de manière explicite le rôle de cet attribut dans le fonctionnement interne de l'environnement ;
- $valeur_{attr}$ est une fonction associant une valeur à chaque identifiant d'attribut défini dans $attributs$.

Une entité ne peut être utilisée pour modéliser un phénomène que si elle est compatible avec l'environnement utilisé pour cette simulation, *i.e.* si elle implémente toutes les primitives abstraites manipulées

dans les primitives de l'environnement.

3.6 Modèle de comportements réactifs dans IODA

La simulation explicative consiste en particulier à appliquer le principe de parcimonie et donc à fournir le modèle le plus simple possible permettant d'expliquer l'apparition d'un phénomène. La description du de la sélection d'interaction n'y échappe pas. Dans un tel cadre, l'utilisation d'architectures à planification délibérative n'est pas la plus indiquée. En effet, un comportement basé sur des architectures réactives ou à planification réactive constitue dans un grand nombre de cas une approximation acceptable du comportement réel d'entités existant dans un phénomène, même lorsque ces dernières raisonnent pour produire leur comportement.

Dans cette thèse, nous nous focalisons donc sur la description d'un processus de sélection d'interaction réactif. L'approche IODA n'y est toutefois pas restreinte, puisque la structure des interaction permet leur utilisation dans tout type d'architecture fondée sur l'utilisation de règles. L'approche centrée sur les interaction développée par l'équipe SMAC pour le domaine d'application des jeux-vidéo [DMR05] en témoigne en définissant une architecture à planification délibérative reposant sur les mêmes principes fondamentaux que IODA.

3.6.1 Principes de la sélection réactive d'interaction

Dans sa forme la plus simple, un comportement réactif consiste à réagir systématiquement à chaque stimulus reçu. Autrement dit, une entité initie une interaction dès que ses conditions sont vérifiées. Cette approche ne permet de modéliser que des phénomènes physiques comme les réactions chimiques ou le déplacement de nuages de gaz. La spécification de comportement plus complexes, par exemple celui d'animaux en éthologie, nécessite un modèle comportemental plus raffiné. Ce raffinement consiste à donner aux entités la capacité de ne pas réagir aveuglément à chaque stimulus reçu. Pour cela, elles disposent de la capacité de choisir les interactions initiées parmi les interactions dont les conditions sont vérifiées.

Dans IODA un tel choix est exprimé à l'aide d'un ordre attribué aux interactions qu'une entité a la capacité d'initier. Cet ordre est exprimé de sorte qu'une interaction dont les conditions sont vérifiées subsumera l'initiation de toute interaction d'ordre moins élevé, considérées comme moins prioritaires.

Exemple. Ordre total entre deux interactions
Dans le cadre d'une simulation en éthologie, un animal peut donner à l'élément d'assignation individuel « MANGER une proie » une priorité supérieure à l'élément d'assignation dégénéré « SE DÉPLACER ». En effet, si une entité a le choix entre se déplacer et manger, elle choisira systématiquement de manger afin de réduire sa sensation de faim et donc éviter de mourir.

L'ordre permet ainsi d'obtenir une structure similaire à une instruction de type *if then... else ...* utilisée dans les plateformes de simulation ouvertes. Il permet de plus d'exprimer des préférences parmi les cibles d'une même interaction.

Exemple. Préférences entre deux cibles
Dans le cadre d'une simulation en éthologie, un prédateur peut donner à l'élément d'assignation individuel « MANGER une proie » une priorité supérieure à « MANGER une carcasse ». Ainsi, lorsqu'un prédateur a faim, il ne mangera une carcasse pour assurer sa survie que s'il ne peut pas manger une proie vivante à ce moment-là.

Une entité peut aussi ne pas exprimer de préférence entre la famille des cibles d'une même interaction ou même entre deux interactions différentes.

Exemple. Ordre partiel entre deux interactions
Dans le cadre d'une simulation en éthologie, un lion peut ne pas avoir de préférence :
- *entre les éléments d'assignation individuels « MANGER une antilope » et « MANGER une gazelle ». Cela signifie qu'un lion mange indistinctement des antilopes et des gazelles ;*

– *entre les éléments d'assignation individuels «* SE DÉPLACER VERS *une proie » et «* SE DÉPLACER VERS *un lac ». Cela signifie qu'un lion considère que se nourrir est aussi important que se désaltérer.*

Enfin, deux interactions peuvent avoir des conditions exclusives, auquel cas il n'est pas forcément utile d'établir d'ordre entre elles (voir figure 3.11).

Il est à noter que le dernier exemple ne peut être exprimé si l'ordre entre les éléments d'assignations est total. Afin de conserver la plus grande variété comportementale possible, nous considérons donc que cet ordre est partiel. Dans le cas où deux interactions sont les plus prioritaires, l'interaction initiée par une entité est choisie aléatoirement parmi elles.

3.6.2 Modèle réactif de sélection d'interaction

Dans IODA, l'ordre imposé entre les éléments d'assignation d'une matrice d'interaction brute est exprimé à l'aide de nombres entiers que nous appelons *priorité* d'un élément d'assignation. La priorité de chaque élément d'assignation pour une entité $e \in \mathbb{E}$ est connue à l'aide d'une fonction appelée *fonction d'attribution des priorités*.

Définition 26. ***Priorité d'un élément d'assignation***
La **priorité** d'un élément d'assignation a pour une entité e est nombre entier relatif utilisé pour déterminer la position de a dans l'ensemble partiellement ordonné des interactions que l'entité e a la capacité d'initier. Cet ordre est utilisé pour construire le comportement réactif de cette entité.

Définition 27. ***Fonction d'attribution des priorités***
Soit $\mathcal{S} \in \mathbb{F}$.
On appelle **fonction d'attribution des priorités** d'une entité $e \prec \mathcal{S}$, notée $priorite(e)$, la fonction associant une priorité à chaque élément d'assignation contenu dans la ligne de la matrice d'interaction brute associée à $\mathcal{S}$. Plus formellement, pour toute entité $e \in \mathbb{E}$ on a :

$$priorite(e) : \begin{array}{ccc} \left(\mathbb{I}_{(1,0)} \times \mathbb{F}\right) \cup \left(\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}\right) & \rightarrow & \mathbb{Z} \\ (a) & \rightarrow & \begin{cases} priorite(e)(a) & \text{si } a \in \mathcal{M}(famille(e), \mathcal{T}) \\ non\ défini & \text{sinon} \end{cases} \end{array}$$

On considère qu'un élément d'assignation e a un ordre plus élevé que l'élément d'assignation e' si la priorité de e est plus grande que celle de e'.

La seule connaissance de la fonction d'attribution des priorités permet de construire le comportement d'une entité sans avoir recours à une quelconque notion liée à la programmation. Les algorithmes que nous décrivons dans la section 4.3 page 119 du chapitre 4 en sont la preuve.

L'allure de la fonction d'attribution des priorités est fortement dépendante du phénomène étudié et des hypothèses de simulation émises. Dans le cas le plus simple, une priorité est associé une fois pour toute à chaque élément d'assignation.

Exemple. Priorités n'étant jamais modifiées
Dans un modèle de tri collectif tel que celui proposé par Resnick [Res97], une simulation implique deux familles d'entités que nous appelons ici `Trieur` *et* `Trié`*. Le comportement d'une entité e de la famille* `Trieur` *consiste à* SE DÉPLACER *dans l'environnement, à* RAMASSER *une entité de la famille* `Trié` *si e ne transporte rien et à* DÉPOSER *l'entité transportée à coté d'une entité de la famille* `Trié`*. Un exemple de matrice d'interaction brute de ce modèle est fournie dans la figure 3.11a. Dans cette simulation, le comportement des instances de la famille* `Trieur` *reste identique tout au long de la simulation. La fonction d'attribution des priorités est donc telle que décrite dans la figure 3.11b.*

Dans le cas le plus évolué, un changement dans l'état d'une entité peut l'amener à modifier son comportement (et donc les priorités associées à ses éléments d'assignation) sans pour autant modifier le comportement des autres instances de sa famille.

Exemple. Priorités pouvant évoluer
Soit une simulation en éthologie étudiant le comportement social d'animaux. Supposons qu'une telle simulation implique une seule famille d'entités que nous appelons `Animal`*. Le comportement d'une instance*

Source \ Cible	∅	`Trieur`	`Trié`
`Trieur`	(Se Déplacer)		(Ramasser, d=0) (Déposer, d=1)
`Trié`			

(a)

Source	Cible	Élément d'assignation	Priorité
`Trieur`	∅	(Se Déplacer)	0
`Trieur`	`Trié`	(Déposer, d=1)	1
`Trieur`	`Trié`	(Ramasser, d=0)	1

(b)

Figure 3.11 – Matrice d'interaction brute d'une simulation de tri collectif (a) et fonction d'attribution des priorités lui étant associée (b). Les éléments d'assignation (Déposer, d=1) et (Ramasser, d=0) ont la même priorité car leurs conditions sont mutuellement exclusives. Il n'y a donc en pratique jamais de choix aléatoire entre elles.

de la famille ***`Animal`*** *consiste par défaut à* Se Déplacer. *Si jamais cet animal rencontre un autre animal du genre opposé, il* Se Reproduit *avec ce dernier. Cela a pour conséquence d'ajouter un nouvel animal dans l'environnement et de provoquer une courte période de stérilité. Si jamais la reproduction n'est pas possible et si un animal du même genre est rencontré, alors ce dernier est* Attaqué. *Lorsqu'un animal est attaqué, il devient agressif et préfère alors* Attaquer *d'autres* ***`Animaux`*** *plutôt que de* Se Reproduir. *Un exemple de matrice d'interaction brute de ce modèle est fournie dans la figure 3.12a. Dans cette simulation, le comportement des instances de la famille* ***`Animal`*** *change selon leur état. La fonction d'attribution des priorités est donc telle que décrite dans la figure 3.12b.*

Source \ Cible	∅	`Animal`
`Animal`	(Se Déplacer)	(Se Repoduire, d=1) (Attaquer, d=1)

(a)

Source	Cible	Élément d'assignation	Priorité	
			¬ agressif	agressif
`Animal`	∅	(Se Déplacer)	0	0
`Animal`	`Animal`	(Se Reproduire, d=1)	2	1
`Animal`	`Animal`	(Attaquer, d=0)	1	2

(b)

Figure 3.12 – (a) Matrice d'interaction brute d'une simulation étudiant les comportements sociaux d'animaux et (b)fonction d'attribution des priorités lui étant associée. Dans cette simulation, un `Animal` non agressif préfère Se Reproduire à Attaquer. Au contraire, un `Animal` agressif préfère Attaquer à Se Reproduire.

Un grand nombre de simulations peuvent être réalisées sans avoir recours à des priorités pouvant changer au cours de la simulation. Pour nous conformer au principe de parcimonie, nous faisons le choix de ne pas considérer les cas où la priorité d'un élément d'assignation peut dépendre de l'état d'une entité. Par conséquent, il devient possible d'attribuer une unique priorité à chaque élément d'assignation directement dans la matrice d'interaction. Nous introduisons pour cela la notion de *matrice d'interaction raffinée* dont nous proposons un méta-modèle dans la figure 3.14.

Définition 28. ***Matrice d'interaction raffinée***
La **matrice d'interaction raffinée** $\mathcal{M}_{raff}$ d'une simulation est une matrice associant une priorité à chaque élément d'assignation a d'une matrice d'interaction brute $\mathcal{M}$. Plus formellement, elle est notée :

$$\mathcal{M}_{raff} : \begin{array}{ccc} \left(\mathbb{I}_{(1,0)} \times \mathbb{F}\right) \cup \left(\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}\right) & \rightarrow & \mathbb{Z} \\ (a) & \rightarrow & \begin{cases} priorite(e)(a), \forall e \prec source(a) & \text{si } \exists \mathcal{S} \in \mathbb{F}, \exists \mathcal{T} \in \mathbb{F} \cup \{\emptyset\} | a \in \mathcal{M}(\mathcal{S}, \mathcal{T}) \\ \text{non défini} & \text{sinon} \end{cases} \end{array}$$

La figure 3.13 présente un exemple d'une telle matrice pour une simulation de tri collectif dont la fonction d'attribution des priorités est décrite dans la figure 3.11.

3.7 Synthèse du chapitre

Nous avons établi dans la partie précédente que pour répondre aux besoins inhérents aux simulations large échelle le modèle doit exprimer un certain nombre de propriétés. Dans ce chapitre, nous définissons formellement le modèle de notre approche **Interaction Oriented Design of Agent simulations**

Source \ Cible	$\emptyset$	`Trieur`	`Trié`
`Trieur`	(SE DÉPLACER, $p = 0$)		(RAMASSER, d=0, $p = 1$) (DÉPOSER, d=1, $p = 1$)
`Trié`			

FIGURE 3.13 – Matrice d'interaction raffinée d'une simulation de tri collectif, dont le modèle est résumé dans la section 3.6.2. Dans cette matrice, l'élément « (DÉPOSER, d=1, $p = 1$) » est lu « l'élément d'assignation (DÉPOSER, d=1) a pour priorité 1 ».

(IODA). Nous montrons de plus que les concepts y étant développés sont conformes à ces besoins et facilitent ainsi la conception de simulations large échelle.

Le modèle formel de IODA est fondé sur six notions fondamentales et repose en particulier sur le terme neutre « entité » pour que sa lecture ne soit pas biaisée par la connotation du terme « agent ». Dans le reste de ce manuscrit, nous utilisons les termes « entité » et « agent » comme synonymes. La notion la plus fondamentale du modèle est « l'interaction ».

Interaction. Une **interaction** représente une séquence sémantique d'actions qui n'est initiée que si ses **conditions** d'exécution sont vérifiées. Elle implique des entités **source** (qui initient l'interaction) conjointement à des entités **cible** (qui subissent l'interaction). Nous caractérisons le nombre d'entités source et cible d'une interaction par un couple (n, m) appelé **cardinalité**. Il caractérise le nombre n de sources et m de cibles impliquées dans une interaction.

Dans le cœur de l'approche IODA, nous nous focalisons sur deux types d'interactions, qui permettent d'exprimer une majorité de simulations :

- les **interactions individuelles** de cardinalité est $(1, 1)$;
- les **interactions dégénérées** de cardinalité est $(1, 0)$, qui prennent implicitement leur source pour cible.

Afin de décrire les interaction indépendamment des spécificités des entités ou de l'environnement, les actions et les conditions des interactions sont exprimées à l'aide de **primitives abstraites**. Ces primitives doivent être spécifiées par l'environnement pour que l'interaction puisse y avoir lieu ou par les entités pour qu'elles puissent participer à l'interaction. Une interaction définit pour cela des **signatures** contenant l'ensemble des primitives :

- devant être spécifiées par la source de l'interaction (**signature de la source**) ;
- devant être spécifiées par la cible de l'interaction (**signature de la cible**) ;
- devant être spécifiées par l'environnement (**signature de l'environnement**).

Une entité **peut** alors **initier** (resp. **subir**) une interaction si elle spécifie les primitives de la signature de la source (resp. cible) dans l'interaction. Par conséquent, une interaction est possible entre une entité source et une entité cible si la **source peut initier** l'interaction et si la **cible peut la subir**.

Les cinq autres notions développées dans IODA permettent d'intégrer de manière effective les interactions dans le comportement des entités.

Entité. La notion d'**entité** unifie la représentation de toute entité pertinente du phénomène simulé, qu'elle soit autonome et proactive ou n'initiant aucune interaction et qu'elle soit réactive ou cognitive. Nous effectuons dans IODA une distinction similaire à la différence classe/instance des langages objets en distinguant **instance d'entité** et **famille d'entités**. Une famille d'entités est en particulier caractérisée par :

- un **halo** qui permet d'identifier les entités perçues ;
- un ensemble de primitives qu'elle spécifie ;
- un ensemble d'interactions qu'elle peut initier ;
- un **modèle de sélection d'interaction** décrivant son comportement.

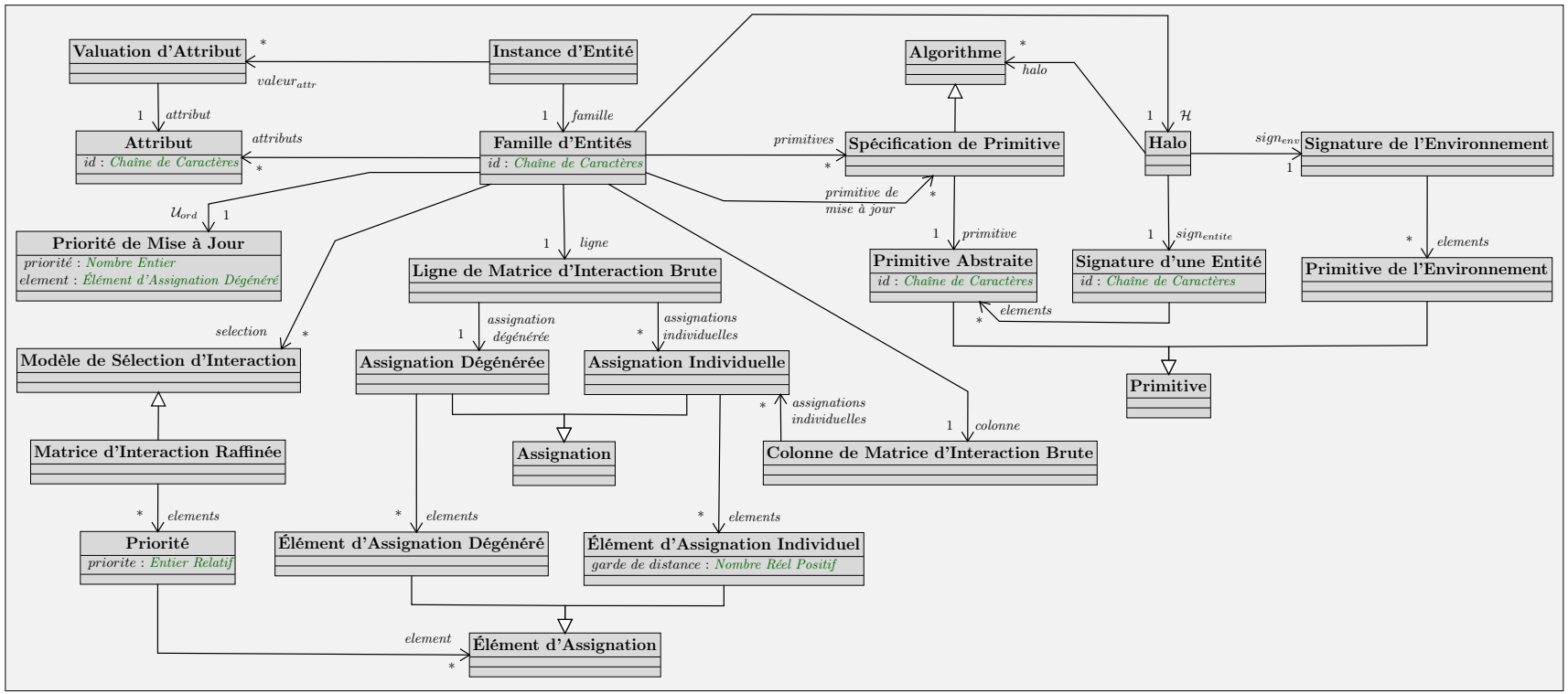


FIGURE 3.14 – Métamodèle d'une famille d'entité et d'une instance d'entité dans IODA, intégrant la notion de matrice d'interaction raffinée.

Environnement. L'**environnement** représente l'espace où se situent les entités. Il est caractérisé par un ensemble de primitives permettant de connaître son état et de le modifier. **Tout type d'environnement peut être utilisé** dans IODA du moment qu'une métrique y soit définie sous la forme d'une primitive de calcul de distance entre deux entités.

Matrice d'interaction brute. La **matrice d'interaction brute** définit l'ensemble des interactions pouvant survenir entre les entités d'une simulation. Une ligne y représente les interactions pouvant être initiées par une famille d'entités et une colonne les interactions pouvant être subies par une famille d'entités. Ainsi, une interaction figurant à l'**intersection** d'une **ligne** et d'une **colonne** caractérise une interaction pouvant avoir lieu entre la famille d'entités source et la famille d'entités cible concernées.

Matrice de mise à jour. La **matrice de mise à jour** représente les interactions initiées par une entité indépendamment de son comportement afin de mettre son état à jour. Elle permet de modifier simplement la façon dont l'état d'une entité évolue à l'aide d'interactions dégénérées.

Modèle de sélection d'interaction et matrice d'interaction raffinée. Le **modèle de sélection d'interaction** décrit comment une entité choisit l'interaction qu'elle initie à partir de sa ligne de la matrice d'interaction brute et de l'ensemble des entités qu'elle perçoit.

Les simulations large échelle peuvent difficilement être exécutées si le comportement des agents nécessite un grand nombre de calculs puisque dans de tels cas seul un nombre restreint d'agents peuvent être simulés. Le compromis le plus couramment utilisé consiste à utiliser un modèle de sélection d'action réactif dans lequel les règles condition/effet peuvent être plus ou moins cognitives. Nous intégrons à IODA un modèle de sélection d'interaction réactif fondé sur ce principe. Ce modèle repose sur une **matrice d'interaction raffinée** dans laquelle des priorités sont attribuées aux interactions figurant dans la matrice d'interaction brute. Le comportement d'une entité consiste alors à initier une interaction de priorité maximale dont les conditions sont vérifiées.

Modèle du temps. Afin de ne pas entraver la compréhension des concepts de notre approche centrée sur les interactions, nous avons choisi dans cette thèse de reposer sur une représentation du temps couramment rencontrée dans les simulations multi-agents :

- le modèle de sélection d'interaction ne peut sélectionner qu'une seule interaction ;
- le temps est discrétisé et décomposé en pas de temps de durée égale. Lors de chaque pas de temps, la parole est donnée aux agents séquentiellement.

Apports du modèle. Afin de faciliter la conception de simulations large échelle, nous avons établi dans la partie précédente que le modèle d'une simulation doit exprimer un certain nombre de propriétés. Le tableau 3.1 montre que le modèle formel IODA exprime de telles propriétés et facilite ainsi la conception de simulations large échelle.

Vers une approche transversale de conception. Le modèle IODA n'a d'intérêt que s'il peut reposer sur une approche de conception transversale afin d'implémenter une simulation. Dans le chapitre qui suit, nous prouvons ce point en décrivant la méthodologie IODA qui permet de passer d'une description d'un phénomène dans le langage naturel en un modèle IODA prêt à être implémenté en conservant sa structure à l'aide d'algorithmes.

TABLE 3.1 – Preuve que le modèle IODA facilite la conception de simulations large échelle. Dans cette figure nous résumons les propriétés facilitant la conception de simulation identifiées dans le chapitre 2 (colonne gauche) et exprimons à droite comment le modèle IODA s'y conforme (colonne droite).

Propriété souhaitée	Concept dans IODA
Le modèle est précis et décrit le phénomène à différents niveaux d'abstractions.	Les matrices décrivent la simulation à un haut niveau d'abstraction.
	Les conditions et actions des interactions décrivent la simulation à un niveau intermédiaire d'abstraction.
	Les primitives des agents décrivent la simulation à un niveau fin d'abstraction.
Déclaratif et procédural sont séparés.	La matrice d'interaction brute est une représentation abstraite des capacités des agents.
	La matrice d'interaction raffinée est une représentation abstraite du comportement des agents.
Les actions des agents sont représentées de manière générique et indépendante de leur comportement.	Il y a séparation entre matrice d'interaction brute et modèle de sélection d'interaction.
Les actions des agents sont représentées sous la forme de règles conditions/effets.	Les interactions sont construites selon ce principe.
Une interaction représente toute action impliquant simultanément plusieurs agents.	Les interactions sont construites selon ce principe.
Les interactions entre agents sont modélisées à l'aide d'un graphe ou d'une forme équivalente.	La matrice d'interaction brute est une forme équivalente au graphe.
Les interactions sont intégrées au comportement des agents.	Le modèle de sélection d'interaction se fonde sur une ligne de la matrice d'interaction brute.

Chapitre 4

IODA comme approche transversale de conception

Plan du chapitre :

Dans ce chapitre, nous caractérisons comment construire une approche transversale de conception de simulations à partir du modèle formel décrit dans le chapitre précédent. Nous décrivons pour cela des principes permettant à la fois de construire un modèle de manière incrémentale (une méthodologie de conception), mais aussi de l'implémenter à l'aide d'algorithmes de simulation se basant sur les données du modèle. Nous profitons de plus de ce chapitre pour montrer comment notre approche se positionne vis à vis de quatre questions se posant systématiquement dans toute approche de conception transversale :

- *Toute entité est-elle un agent ?*
- *Une méthodologie de conception doit-elle être dogmatique ?*
- *Faut-il prendre en compte les performances lors de la modélisation ?*
- *Faut-il tout décrire au sein d'une règle (dans notre cas d'une interaction) ?*

La section 4.1 décrit comment remplir progressivement le modèle formel décrit dans le chapitre 3, à l'aide de représentations graphiques facilement manipulables. La section 4.2 étudie les quatre questions mentionnées précédemment et caractérise comment IODA *se positionne par rapport à ces problèmes. Enfin, la section 4.3 décrit les algorithmes tirant parti des informations présentes dans un modèle formel* IODA *afin d'exécuter une simulation. Nous en dégageons des avantages de* IODA *facilitant la conception de simulations.*

Une approche transversale de conception est caractérisée par deux phases : la phase de construction d'un modèle à partir de spécifications formulées par le langage naturel et la phase de traduction de ce modèle en un ensemble d'algorithmes pouvant être implémentés.

4.1 La méthodologie de conception IODA

Le rôle du formalisme du modèle est de fournir une structure précise à la description d'une simulation, afin de donner un cadre formel permettant sa spécification. En théorie, la connaissance d'un modèle formel permet de décrire de manière précise tout type phénomène dans la limite de l'expressivité du formalisme utilisé. En pratique, de tels modèles deviennent très rapidement impossibles à spécifier uniquement à l'aide de ce formalisme. En effet, le formalisme du modèle est un tout dont certains éléments très précis ne peuvent être spécifiés dès le début du processus de conception. Par exemple, il est inutile de s'intéresser au processus de sélection d'interaction d'une entité sans connaître ce que l'entité est capable de faire. De même, l'identification d'attributs d'une famille d'entités n'a pas de sens en dehors de la spécification précise du comportement des entités, *etc.*

Dans notre approche, nous associons donc au formalisme du modèle IODA une méthodologie de conception que nous appelons méthodologie IODA. Cette méthodologie fournit un procédé permettant

de concevoir un modèle graduellement en commençant par ses aspects macroscopiques (les interactions entretenues par les entités), pour finir sur ses aspects microscopiques (le détail des actions que les entités entreprennent). Cette section a pour rôle de de décrire ce procédé, présenter les éléments graphiques permettant cette spécification ainsi qu'identifier comment les éléments graphiques permettent de remplir le modèle formel de la simulation spécifiée.

4.1.1 Processus général

La méthodologie IODA repose sur la construction de deux matrices afin de décrire la logique générale d'une simulation : la matrice d'interaction brute et la matrice d'interaction raffinée. Ces matrices permettent une conception graduelle du modèle de la simulation à deux niveaux.

Conception graduelle à deux niveaux.

D'une part, elles permettent de décrire graduellement la logique générale de la simulation, qui passe de l'identification du graphe des interactions liant les entités (la matrice d'interaction brute) à son interprétation en termes d'actions que les entités sont capables d'initier (les éléments contenus dans une ligne de la matrice), pour finir le comportement général des entités en attribuant des priorités aux différentes capacités d'une entité (matrice d'interaction raffinée). Ainsi, le schéma général d'une méthodologie de conception transversale, tel qu'il est décrit dans la figure 2.5 page 44, est respecté, assurant une spécification de plus en plus précise du fonctionnement du phénomène.

D'autre part, pour des raisons précédemment annoncées comme nécessaires au polymorphisme des interactions, les interactions ont une description indépendante des entités pouvant y participer. La seconde justification d'une telle structure est qu'elle permet de plus de décrire les connaissances des entités de manière graduelle. En effet, la description de ce que les entités sont capables de faire est divisé en deux phases. Elle commence par une description abstraite valide pour toutes les entités de la simulation (les préconditions, déclencheurs et actions d'une interaction), pour ensuite être raffinée et précisée, au moment où les primitives abstraites de chaque entité sont décrites.

Caractérisation du processus de conception.

La méthodologie de conception permettant de concevoir des modèles IODA suit un processus décomposé en treize étapes résumées sur la figure 4.1. Chaque étape facilite la spécification d'un élément du modèle formel en :

- fournissant une représentation graphique permettant une description visuelle du modèle ;
- décrivant comment interpréter automatiquement cette représentation afin de compléter le modèle formel ;
- caractérisant les conséquences de cette étape sur le modèle spécifié dans les autres étapes de la méthodologie ;
- identifiant les étapes devant être préalablement spécifiées pour passer à cette étape.

La construction d'un modèle selon la méthodologie IODA se fait en trois phases s'attachant chacune à des informations de granularités différentes : la phase de spécification *des interactions entre les entités*, la phase de spécification *du comportement général des entités* et enfin la phase de spécification *du comportement précis des entités*.

Les lettres dans la figure sont utilisées uniquement pour nommer chaque nœud. Elles ne présument en aucun cas d'un quelconque ordre dans lequel suivre la méthodologie, qui peut être arbitraire du moment qu'une étape est spécifiée lorsque les étapes dont elle dépend sont déjà spécifiées. Une discussion concernant ce point est effectuée plus tard dans ce chapitre, dans la section 4.2.2 page 114.

4.1.2 Spécification des interactions entre les entités

La première partie de la méthodologie consiste à identifier trois éléments fondamentaux à toute simulation centrée sur les interaction :

- la topologie de l'environnement utilisé (« A » dans la figure 4.1) afin de donner un sens à la notion de distance entre deux entités ;

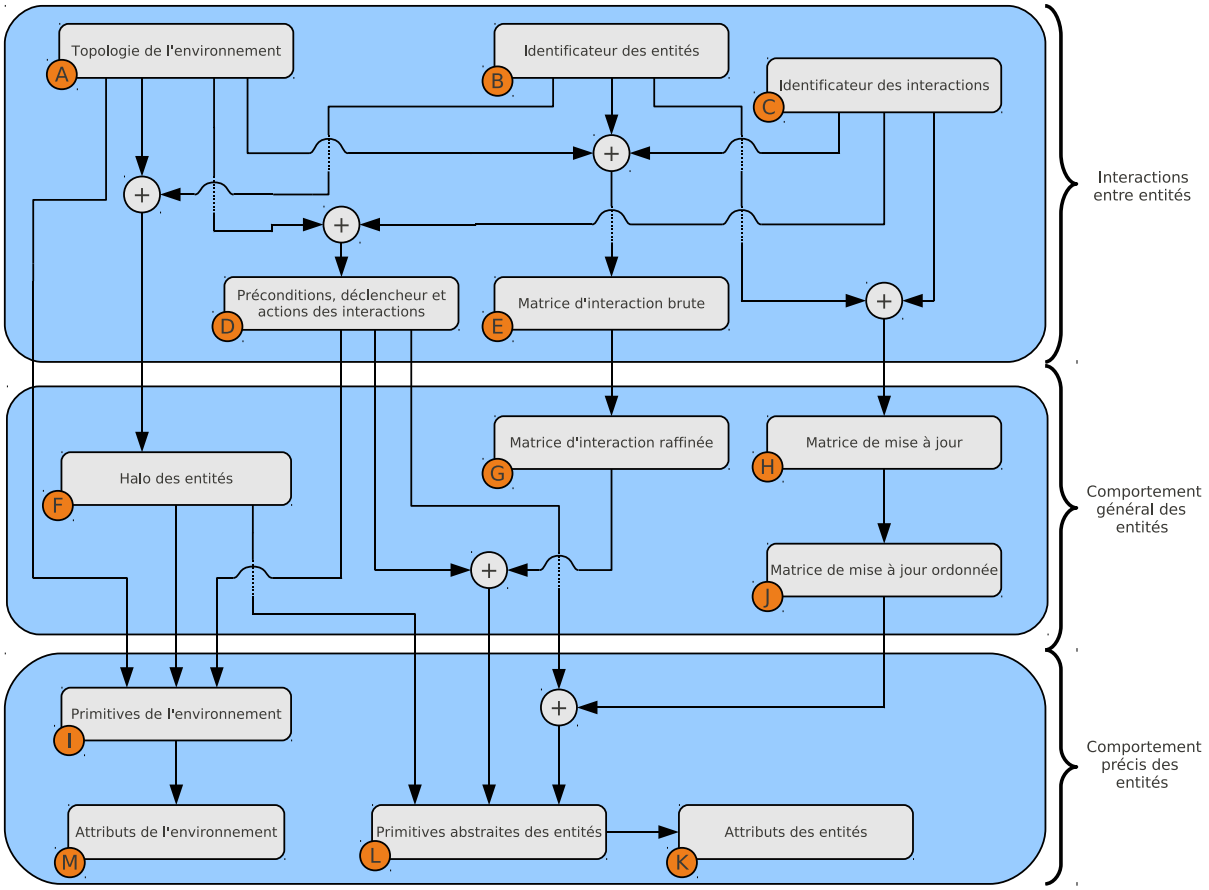


FIGURE 4.1 – Cheminement suivi dans la méthodologie IODA pour concevoir un modèle. Une flèche partant d'un nœud X vers un nœud Y signifie que pour commencer les spécifications Y d'un élément du modèle, son pendant dans X doit avoir été spécifié. Par exemple, la priorité d'un élément d'assignation dans la matrice d'interaction raffinée (G) n'est faite qu'après avoir spécifié l'élément d'assignation dans la matrice d'interaction brute (E). Deux transitions allant d'un nœud X_1 (resp. X_2) à un nœud Y signifient que le nœud Y peut être spécifié partiellement une fois que le nœud X_1 (resp. X_2) est spécifié. Une transition passant par un nœud « + » et reliant plusieurs nœuds $X_1 \ldots X_n$ à un nœud Y signifie que les nœuds $X_1 \ldots X_n$ doivent avoir été spécifiées pour commencer les spécifications du nœud Y. Par exemple, la description d'une primitive abstraite dans une entité (L) ne sera faite que si les déclencheurs, préconditions et actions de l'interaction la manipulant sont décrites (D) et si l'entité est capable d'initier ou subir cette interaction (G ou J).

- l'identifiant des différentes familles d'entités participant au phénomène simulé (« B » dans la figure 4.1) ;
- l'identifiant des différentes interactions pouvant survenir dans le phénomène (« C » dans la figure 4.1) ;
- la matrice d'interaction brute du phénomène simulé (« E » dans la figure 4.1) ;
- les préconditions, déclencheur et actions de chaque interaction du phénomène simulé (« D » dans la figure 4.1) afin de fixer la sémantique de l'interaction.

Topologie de l'environnement (« A »). La spécification de la topologie de l'environnement consiste à donner un sens à la notion de distance dans la simulation, *i.e.* à lui donner une unité, afin de pouvoir attribuer des gardes de distance dans la matrice d'interaction brute. Elle constitue un pré-requis à l'expression du halo (étape « F »), à la description des interactions (étape « D ») et à la description de la matrice d'interaction brute (étape « E »). À ce stade de la méthodologie, aucune description formelle n'est fournie à la notion de distance.

Identifiant des familles d'entités (« B »). Aucune représentation particulière n'est préconisée pour la spécification des identifiants des familles d'entités, puisque cette étape consiste simplement à dresser une liste d'identifiants, *i.e.* de chaînes de caractères. Cette étape permet de débuter la construction de l'ensemble $\mathbb{F}$: pour chaque identifiant id de la liste, une famille d'entités $\langle id, \emptyset, \emptyset, ?, \emptyset, \emptyset, \emptyset, ? \rangle$ est ajoutée à $\mathbb{F}$.

Identifiant des interactions (« C »). Tout comme lors de l'étape « B », aucune représentation particulière n'est préconisée pour la spécification des identifiants des interactions, puisqu'elle consiste aussi à dresser une liste d'identifiants. Cette étape permet de débuter la construction de l'ensemble $\mathbb{I}_{(1,0)} \cup \mathbb{I}_{(1,1)}$: pour chaque identifiant id de la liste, une interaction $\langle id, \emptyset, \emptyset, \emptyset, \emptyset, ?, ?, ? \rangle$ est ajouté à $\mathbb{I}_{(1,0)} \cup \mathbb{I}_{(1,1)}$.

À ce stade de la méthodologie, la cardinalité des interactions n'est pas connue. Elle sera déterminée soit lors de l'étape de description des interactions « D », soit lors la construction de la matrice d'interaction brute « E ».

Matrice d'interaction brute (« E »). Une fois les étapes « A », « B » et « C » terminées, il devient possible de construire le graphe des interactions entre les entités de la simulation (« E » dans la figure 4.1). Cette étape consiste dans un premier temps a construire une matrice d'interaction brute vide, en ajoutant une ligne et une colonne pour chaque identifiant de famille d'entités spécifié lors de l'étape « B ». Cette étape revient à créer une matrice telle que $\forall \mathcal{S} \in \mathbb{F}, \forall \mathcal{T} \in \mathbb{F} \cup \{\emptyset\}, \mathcal{M}(\mathcal{S}, \mathcal{T}) = \emptyset$.

Dans un second temps, cette matrice est remplie en plaçant des identifiants d'interaction, issus de l'étape « C », à l'intersection d'une ligne et d'une colonne de la matrice, afin de définir les interactions ayant lieu entre les entités de la simulation, ou les interactions dégénérées que les entités ont la capacité d'initier. Si un identifiant est ajouté dans une assignation individuelle (*i.e.* l'intersection d'une ligne et d'une colonne associées à des identifiants de famille d'entité), une garde de distance lui est associée. Cette étape permet ainsi d'obtenir une matrice dont l'apparence est illustrée sur la figure 3.12a page 91.

A partir de cette représentation graphique, il est possible de déduire l'ensemble des éléments d'assignation du modèle. En effet, si $\mathcal{S} \in \mathbb{F}$, $\mathcal{T} \in \mathbb{F}$, $\mathcal{I} \in \mathbb{I}_{(1,0)} \cup \mathbb{I}_{(1,1)}$ et $r \in \mathbb{R}^+$, on a :

- si l'identifiant $id(\mathcal{I})$ apparaît sous la forme $\Big(id(\mathcal{I}), d = r\Big)$ à l'intersection de la ligne associée à $id(\mathcal{S})$ et de la colonne associée à $id(\mathcal{T})$, alors un élément d'assignation individuel $(\mathcal{I}, r, \mathcal{S}, \mathcal{T})$ est ajouté à $\mathcal{M}(\mathcal{S}, \mathcal{T})$. Cet élément est aussi ajouté à $ligne(\mathcal{S})(\mathcal{T})$ et à $colonne(\mathcal{T})(\mathcal{S})$;
- si l'identifiant $id(\mathcal{I})$ apparaît sous la forme $\Big(id(\mathcal{I})\Big)$ à l'intersection de la ligne associée à $id(\mathcal{S})$ et de la colonne associée à $\emptyset$, alors un élément d'assignation dégénéré $(\mathcal{I}, \mathcal{S})$ est ajouté à $\mathcal{M}(\mathcal{S}, \emptyset)$. Cet élément est aussi ajouté à $ligne(\mathcal{S})(\emptyset)$.

À ce stade de la spécification, la cardinalité des interactions est déduite de la nature des cellules de la matrice où elles sont disposées : une interaction figurant dans la colonne "$\emptyset$" (*i.e.* dégénérée) devient une interaction dégénérée. Sinon, elle est considérée par défaut comme une interaction individuelle :

$$\forall \mathcal{I} \in \mathbb{I}_{(1,0)} \cup \mathbb{I}_{(1,1)} \begin{cases} \Big(\exists \mathcal{S} \in \mathbb{F}, \exists \mathcal{T} \in \mathbb{F}, \exists r \in \mathbb{R}^+ | (\mathcal{I}, r, \mathcal{S}, \mathcal{T}) \in \mathcal{M}(\mathcal{S}, \mathcal{T})\Big) \Rightarrow \Big(card(\mathcal{I}) = (1,1) \wedge \mathcal{I} \in \mathbb{I}_{(1,1)} \wedge \mathcal{I} \notin \mathbb{I}_{(1,0)}\Big) \\ \Big(\exists \mathcal{S} \in \mathbb{F} | (\mathcal{I}) \in \mathcal{M}(\mathcal{S}, \emptyset)\Big) \Rightarrow \Big(card(\mathcal{I}) = (1,0) \wedge \mathcal{I} \in \mathbb{I}_{(1,0)} \wedge \mathcal{I} \notin \mathbb{I}_{(1,1)}\Big) \end{cases}$$

Préconditions, déclencheur et actions des interactions (« D »). La définition des identifiants des différentes interactions survenant dans la simulation (étape « C »), conjointement à la spécification de la topologie de l'environnement (étape « A »), débouche également sur une étape où un sens est donné aux interaction (étape « D »). La sémantique générale des interactions est attribuée indépendamment des spécificités des entités. Elle consiste à en décrire le déclencheur, les préconditions et les actions. Cette description peut s'acquérir de deux manières. La première consiste à utiliser une interaction déjà définie dans une librairie existante, dans quel cas la spécification se limite à l'identification de l'interaction utilisée dans la librairie. Il faut toutefois s'assurer que la signature de l'environnement dans cette interaction n'entre pas en conflit avec la topologie identifiée lors de l'étape « A ». La deuxième manière de spécifier une interaction passe par la description précise de son déclencheur, de ses préconditions et de ses actions.

La description d'une interaction se fait selon les cinq étapes qui suivent, dont l'objectif est d'aboutir à une représentation graphique de la forme présentée dans la figure 4.2 pour décrire le sens de l'interaction, ainsi que les représentations graphiques de la forme présentée dans la figure 4.3 pour décrire la signature de chaque entité impliquée dans l'interaction, ainsi que la signature de l'environnement dans cette interaction :

1. L'identification de la cardinalité de l'interaction ;
2. L'attribution d'un nom aux différentes entités pouvant participer à l'interaction ;
3. La description des préconditions, déclencheur et actions à l'aide d'algorithmes ;
4. L'identification de la signature de chaque entité participant à l'interaction ;
5. L'identification de la signature de l'environnement dans l'interaction

La construction de cette représentation graphique commence par l'identification de la cardinalité de l'interaction. Elle consiste soit à interpréter le sens souhaité de l'interaction et d'en déduire la cardinalité, soit à récupérer la cardinalité déduite de la place de l'interaction dans la matrice d'interaction brute définie lors de l'étape « B », ou dans la matrice de mise à jour lors de l'étape « H ». Une fois la cardinalité de l'interaction identifiée, la représentation graphique du modèle de l'interaction est fixée à celle de la figure 4.2(a) si l'interaction est individuelle, et à celle de la figure 4.2(b) si l'interaction est dégénérée.

La seconde étape consiste à attribuer un nom aux entités participant à l'interaction. Puisque nous nous plaçons uniquement dans le cadre des interactions individuelles et dégénérées, nous avons fixé les noms des entités à {Source} si l'interaction est dégénérée et à {Source, Cible} si l'interaction est individuelle. Cette étape consiste donc à modifier l'entête de la représentation graphique pour « IDENTIFIANT DE L'INTERACTION(Source, Cible) » si l'interaction est individuelle, ou pour IDENTIFIANT DE L'INTERACTION(Source) si l'interaction est dégénérée.

Ensuite, la description des préconditions, actions et déclencheur de l'interaction consiste à écrire un algorithme manipulant à la fois les primitives de l'environnement, des primitives d'action et des primitives de perception. Bien que nous ne fournissions pas de description formelle du langage utilisé pour décrire ces algorithmes, nous imposons toutefois d'identifier explicitement les primitives abstraites des entités, en utilisant une syntaxe de la forme :

« *Nom d'une entité* »'.' « *Identifiant de primitive abstraite* »'('« *Arguments de la primtive abstraite* »')'

Nous préconisons de plus d'identifier les primitives de l'environnement sous la forme :

'Environnement.'« *Identifiant de primitive de l'environnement* » '('« *Arguments de la primitive de l'environnement* »')'

Une illustration d'une telle description est fournie dans la figure 4.4 pour l'interaction MANGER.

Puisque nous nous plaçons dans le cadre d'interactions de cardinalité $(1,1)$ ou $(1,0)$, une interaction ne définit au plus que deux signatures d'entités, que nous avons appelées $sign_{source}$ et $sign_{cible}$. Les primitives contenues dans ces signatures sont déduites de l'algorithme associé aux préconditions, aux actions et au

« interaction individuelle »
IDENTIFIANT DE L'INTERACTION(Source, Cible)

Signatures	Nom	Identifiant de la signature
	Source	*Identifiant de signature d'entité*
	Cible	*Identifiant de signature d'entité*
	Environnement	*Identifiant de signature de l'environnement*
Déclencheur	*% Déclencheur de l'interaction, décrit sous la forme d'un algorithme. Il manipule des primitives de l'entité nommée "Source", de l'entité nommée "Cible", ainsi que des primitives de l'environnement.*	
Préconditions	*% Préconditions de l'interaction, décrites sous la forme d'un algorithme. Il manipule des primitives de l'entité nommée "Source", de l'entité nommée "Cible", ainsi que des primitives de l'environnement*	
Actions	*% Actions de l'interaction, décrites sous la forme d'un algorithme. Il manipule des primitives de l'entité nommée "Source", de l'entité nommée "Cible", ainsi que des primitives de l'environnement*	

(a) Représentation graphique d'une interaction individuelle

« interaction dégénérée »
IDENTIFIANT DE L'INTERACTION(Source)

Signatures	Nom	Identifiant de la signature
	Source	*Identifiant de signature d'entité*
	Environnement	*Identifiant de signature de l'environnement*
Déclencheur	*% Déclencheur de l'interaction, décrit sous la forme d'un algorithme. Il manipule des primitives de l'entité nommée "Source", ainsi que des primitives de l'environnement.*	
Préconditions	*% Préconditions de l'interaction, décrites sous la forme d'un algorithme. Il manipule des primitives de l'entité nommée "Source", ainsi que des primitives de l'environnement.*	
Actions	*% Actions de l'interaction, décrites sous la forme d'un algorithme. Il manipule des primitives de l'entité nommée "Source", ainsi que des primitives de l'environnement.*	

(b) Représentation graphique d'une interaction dégénérée

FIGURE 4.2 – Forme générale de la représentation graphique permettant la spécification d'une interaction.

« signature d'une entité » Identifiant de la signature				
Identifiant	Valeur retournée	Attribut		Description
		Identifiant	Type	
...	...	...	...	...

(a) Représentation graphique d'une signature d'entité dans une interaction

« signature de l'environnement » Identifiant de la signature				
Identifiant	Valeur retournée	Attribut		Description
		Identifiant	Type	
...	...	...	...	...

(b) Représentation graphique d'une signature de l'environnement dans une interaction

FIGURE 4.3 – Forme générale de la représentation graphique permettant la spécification d'une signature d'entité dans une interaction.

« interaction individuelle » MANGER(Source, Cible)		
Signatures	Nom	Identifiant de la signature
	Source	SourceDeManger
	Cible	CibleDeManger
	Environnement	EnvironnementDansManger
Déclencheur	Source.aFaim()	
Préconditions	Cible.estSain()	
Actions	Source.diminuerFaim(Cible.valeurEnergetique()) Environnement.retirer(Cible)	

FIGURE 4.4 – Spécification d'une interaction dont l'identifiant est MANGER et dont la cardinalité est $(1, 1)$.

« signature d'entité » SourceDeManger				
Identifiant	Valeur retournée	Attribut		Description
		Identifiant	Type	
aFaim	Booléen	—	—	Retourne vrai si l'entité ressent la sensation de faim.
diminuerFaim	—	valeurEnergetique	Nombre Entier	Diminue la sensation de faim de l'entité d'un montant égal à la valeur énergétique fournie en paramètre.

« signature d'entité » CibleDeManger				
Identifiant	Valeur retournée	Attribut		Description
		Identifiant	Type	
estSain	Booléen	—	—	Retourne vrai si l'entité représente un aliment sain.
valeurEnergetique	Nombre Entier	—	—	Détermine la valeur énergétique de cette entité.

FIGURE 4.5 – Spécification de la signature des entités dans l'interaction MANGER précédemment décrite dans la figure 4.4.

« signature de l'environnement » EnvironnementDansManger				
Identifiant	Valeur retournée	Attribut		Description
		Identifiant	Type	
retirer	—	e	Entité	Retire une entité de l'environnement.

FIGURE 4.6 – Spécification de la signature de l'environnement dans l'interaction MANGER, précédemment décrite dans la figure 4.4.

déclencheur : $sign_{source}$ (respectivement $sign_{cible}$) contient toutes les primitives des algorithmes des préconditions, du déclencheur et des actions dont l'appel est préfixé par "Source" (resp. "Cible"). Nous prenons pour convention de donner l'identifiant « SourceDeId » à $sign_{source}$ (avec Id l'identifiant de l'interaction), et l'identifiant « CibleDeId » à $sign_{cible}$. Une illustration d'une telle description est fournie dans la figure 4.5 pour l'interaction MANGER, qui définit deux signatures d'une entité dans l'interaction. La première, dont l'identifiant est « SourceDeManger », représente la signature de l'entité dont le nom est « Source » dans l'interaction, qui est la source de l'interaction. La seconde, dont l'identifiant est « CibleDeManger », représente la signature de l'entité dont le nom est « Cible » dans l'interaction, qui est la cible de l'interaction.

Enfin, la signature de l'environnement dans l'interaction est construite de la même manière que la signature d'une entité, en analysant les algorithmes et décelant les primitives dont l'appel est préfixé par « Environnement ». Dans le cas de l'interaction MANGER, cette spécification est résumée sur la figure 4.6.

La construction de la signature des entités et de l'environnement dans l'interaction peut être automatisé. Par conséquent, la spécification d'une interaction consiste en pratique à fournir uniquement la description présente dans la figure 4.2.

Lien entre matrice d'interaction brute et spécification des interactions. La possibilité d'utiliser des librairies d'interactions prédéfinies fait que l'étape de spécification de la matrice d'interaction brute « E » ne survient pas systématiquement avant l'étape de spécification des interactions « D ». Il s'en découle un problème auquel une attention particulière doit être apportée : puisque le fait de placer une interaction dans une cellule de la matrice d'interaction brute permet d'en déduire la cardinalité, il se révèle primordial de s'assurer que la position de chaque interaction dans la matrice n'entre pas en conflit avec le nombre d'arguments du déclencheur, des préconditions et des actions de l'interaction (et réciproquement). En particulier, une interaction figurant dans une assignation individuelle, *i.e.* figurant dans une colonne associée à un identifiant d'entité, ne doit pas avoir deux arguments dans son déclencheur, ses préconditions et ses actions (et réciproquement). De même, une interaction figurant dans une assignation dégénérée, *i.e.* figurant dans la colonne nommée $\emptyset$, ne doit pas avoir un seul argument dans son déclencheur, ses préconditions et ses actions (et réciproquement).

4.1.3 Spécification du comportement général des entités.

Cette deuxième partie de la méthodologie se concentre sur la description générale du comportement des entités. Il y est initialement possible de spécifier trois de ses étapes en parallèle :
- le halo de chaque famille d'entités (étape « F ») ;
- la matrice d'interaction raffinée (étape « G ») ;
- la matrice de mise à jour (étape « H »).

Halo des familles d'entités (« F »). Cette étape consiste à identifier comment, à partir de la seule notion de distance et de la connaissance de la topologie de l'environnement, il est possible aux entités de définir leur voisinage. La halo peut être considéré comme une primitive particulière de l'entité et, à ce titre, être décrite à l'aide d'algorithmes, manipulant des primitives de l'environnement et des primitives de perception ou d'action de l'entité.

Exemple.
Voici un exemple de halo d'une famille d'entités $\mathcal{F}$, désignant une entité comme perçue si elle figure dans la mémoire de l'entité, où si elle se situe à une distance inférieure ou égale à son acuité visuelle.

$halo\Big(\mathcal{H}(\mathcal{F})\Big)(e,e_2)$:
| **retourner** $e.aDansSaMemoire(e_2)$ **ou** $environnement.distance(e,e_2) \leq e.acuiteVisuelle()$

Ce halo implique l'ajout de deux primitives de perception dans la famille de l'entité e : $\langle aDansSaMemoire, booleen, \{Entite\}\rangle$ et $\langle acuiteVisuelle, Nombre\ a\ Virgule\ Flottante, \emptyset\rangle$.

La seule description de cet algorithme permet de définir complètement le halo d'une famille d'entités $\mathcal{F} \in \mathbb{F}$. En effet, chaque occurrence d'une primitive de l'environnement dans l'algorithme mène à l'ajout d'une primitive dans la signature $sign_{env}\Big(\mathcal{H}(\mathcal{F})\Big)$ de l'environnement dans le halo. De même, chaque occurrence d'une primitive d'action ou de perception dans l'algorithme mène à l'ajout d'une primitive dans la signature $sign_{entite}\Big(\mathcal{H}(\mathcal{F})\Big)$ de la famille d'entités $\mathcal{F}$ dans son halo. Le halo de chaque famille d'entités $\mathcal{F} \in \mathbb{F}$ est alors défini par le triplet $\langle halo, sign_{env}\Big(\mathcal{H}(\mathcal{F})\Big), sign_{entite}\Big(\mathcal{H}(\mathcal{F})\Big)\rangle$. Leur représentation graphique est similaire à celle présentée lors de la description de la spécification d'une interaction.

Matrice d'interaction raffinée (« G »). Le deuxième élément pouvant être spécifié au début de cette seconde phase de la méthodologie IODA est la matrice d'interaction raffinée, qui permet de décrire le processus comportemental d'une entité réactive. Cette spécification consiste à générer, à partir de la matrice d'interaction brute spécifiée lors de l'étape « E », une nouvelle matrice, dans laquelle un nombre entier est associé à chaque interaction y apparaissant. Plus ce nombre est élevé, plus l'entité est encline à exécuter l'interaction y étant associée. Cette étape aboutit à une matrice telle que celle de la figure 3.13 page 92.

De cette représentation graphique, on déduit le modèle de la matrice d'interaction raffinée. En effet, si $\mathcal{S} \in \mathbb{F}$, $\mathcal{T} \in \mathbb{F}$, $r \in \mathbb{R}^+$, $q \in \mathbb{Z}$ et $\mathcal{I} \in \mathbb{I}_{(1,0)} \cup \mathbb{I}_{(1,1)}$, alors :

- si l'élément $(\mathcal{I}, d = r, p = q)$ apparaît à l'intersection de la ligne associée à $id(\mathcal{S})$ et de la colonne associée à $id(\mathcal{T})$, alors on sait qu'il existe $a \in \mathcal{M}(\mathcal{S}, \mathcal{T})$ tel que $a = (\mathcal{I}, r, \mathcal{S}, \mathcal{T})$, et que $\mathcal{M}_{raff}(a) = q$;
- si l'élément $(\mathcal{I}, p = q)$ apparaît à l'intersection de la ligne associée à $id(\mathcal{S})$ et de la colonne associée à $\emptyset$, alors on sait qu'il existe $a \in \mathcal{M}(\mathcal{S}, \emptyset)$ tel que $a = (\mathcal{I}, \mathcal{S})$, et que $\mathcal{M}_{raff}(a) = q$.

Matrice de mise à jour (« H »). Le dernier élément pouvant être spécifié en début de cette seconde partie de la méthodologie consiste à établir, au travers d'une matrice de mise à jour, les interactions dégénérées étant exécutées par une entité afin de mettre à jour son état. La première étape de cette spécification consiste à construire une matrice de mise à jour vide, à l'aide des identifiants de familles d'entités identifiées lors de l'étape « B » : pour chaque famille d'entités présente dans $\mathbb{F}$, une colonne dont le titre est l'identifiant de la famille est ajoutée dans la matrice. Cela revient à créer une matrice de mise à jour vide : $\forall \mathcal{S} \in \mathbb{F}, \mathcal{U}(\mathcal{S}) = \emptyset$.

Dans un second temps, la matrice de mise à jour est remplie de la même manière que la colonne dédiée aux assignations dégénérées dans la matrice d'interaction brute, en plaçant l'identifiant d'interactions dans les cases de la matrice (identifiées lors de l'étape « C »). Cette étape aboutit à une matrice dont l'apparence est illustrée sur la figure 4.7.

	Végétal	Herbivore	Carnivore
Interactions de mise à jour		VIEILLIR AUGMENTERSENSATIONFAIM	VIEILLIR AUGMENTERSENSATIONFAIM

FIGURE 4.7 – Matrice de mise à jour d'une simulation de type proie/prédateurs, où des `Herbivores` côtoient des `Carnivores` ainsi que des `Végétaux`. Cette matrice spécifie que la SENSATION DE FAIM AUGMENTE progressivement au fil du temps, même lorsque les entités ne participent pas à une interaction. Elle spécifie de plus que les `Herbivores` et les `Carnivores` peuvent VIEILLIR.

Il est possible de déduire de cette représentation graphique le modèle de la matrice de mise à jour. En effet, avec $\mathcal{S} \in \mathbb{F}$ et $\mathcal{I} \in \mathbb{I}_{(1,0)}$, si l'élément $(id(\mathcal{I}))$ apparaît dans la cellule associée à $id(\mathcal{S})$, alors on peut ajouter l'élément $(\mathcal{I}, \mathcal{S})$ à $\mathcal{U}(\mathcal{S})$.

Tout comme pour la matrice d'interaction brute, l'ajout d'interactions dans cette matrice n'est possible que si leur cardinalité est $(0, 1)$. Réciproquement, une interaction présente dans cette matrice ne peut avoir qu'un seul argument lors de sa spécification dans l'étape « D ».

Matrice de mise à jour ordonnée (« J »). Puisque l'ordre d'exécution de deux interactions de mise à jour peut changer de manière drastique les résultats de simulation obtenus, la matrice de mise à jour spécifiée lors de l'étape « H » doit être complétée, afin de déterminer dans quel ordre ces dernières sont exécutées. Cet ordre est défini par un nombre entier, *i.e.* une priorité, qui est associée à chaque interaction figurant dans la matrice de mise à jour. Cette étape aboutit à une matrice de mise à jour ordonnée, dont l'apparence est illustrée sur la figure 4.8.

	Végétal	Herbivore	Carnivore
Interactions de mise à jour		(Vieillir, p=2) (AugmenterSensationFaim, p=1)	(Vieillir, p=2) (AugmenterSensationFaim, p=1)

FIGURE 4.8 – Matrice de mise à jour ordonnée de la simulation de type proie/prédateurs présentée dans la figure 4.7. Cette matrice spécifie que l'interaction de mise à jour faisant vieillir les entités est exécutée avant l'interaction de mise à jour faisant augmenter leur sensation de faim.

Il est possible de déduire de cette représentation graphique le modèle de la matrice de mise à jour ordonnée. En effet, en posant $\mathcal{S} \in \mathbb{F}$, $\mathcal{I} \in \mathbb{I}_{(1,0)}$ et $q \in \mathbb{Z}$, on sait que si l'élément $(id(\mathcal{I}), p = q)$ apparaît dans la cellule associée à $id(\mathcal{S})$, alors il existe un élément $u \in \mathcal{U}(\mathcal{S})$ tel que $u = (\mathcal{I}, \mathcal{S})$ et que $\mathcal{U}_{ord}(u) = q$.

4.1.4 Spécification du comportement précis des entités et de l'environnement.

Cette dernière partie de la méthodologie IODA spécifie les aspects les plus microscopiques de la simulation. Elle consiste à rendre explicite la description des différentes primitives identifiées lors des étapes précédentes.

Primitives de l'environnement (« I »). Différentes étapes de la méthodologie ont abouti à l'identification de primitives de l'environnement. Parmi ces primitives figurent celle permettant le calcul de la distance séparant deux entités, identifiée lors de l'étape de spécification de la topologie de l'environnement « A », ainsi que des primitives identifiées lors de la description des préconditions, des actions et du déclencheur des interactions « D », et enfin des primitives identifiées lors de la description du halo des entités « F ». À ce stade, on a donc :

$$primitives_{env} = \left\{ p \;\middle|\; \begin{array}{l} \exists \mathcal{I} \in \mathbb{I}_{(1,0)} | p \in sign_{env}(\mathcal{I}) \vee \\ \exists \mathcal{I} \in \mathbb{I}_{(1,1)} | p \in sign_{env}(\mathcal{I}) \vee \\ \exists \mathcal{F} \in \mathbb{F} | p \in sign_{env}(\mathcal{H}(\mathcal{F})) \vee \\ p = \langle distance, Nombre\ a\ Virgule\ Flottante, \{Entite, Entite\} \end{array} \right\}$$

A ces primitives s'ajoutent :
- les primitives permettant d'ajouter ou de retirer une entité à l'environnement ;
- la primitive permettant de connaître les entités situées dans l'environnement (*i.e.* permettant de connaître $\mathbb{E}$).

Puisque la forme de ces primitives dépend de la topologie de l'environnement, elles n'apparaissent pas dans la définition formelle précédente.

L'étape « I » de la méthodologie consiste à fournir une description à ces primitives, en associant un algorithme à chaque primitive de $primitives_{env}$. Le modèle de l'environnement est alors représenté graphiquement sous la forme de la figure 4.9, à laquelle est associée une signature d'entité, dont la représentation graphique a déjà été présentée.

<table>
<tr><td colspan="5">« environnement »
Identifiant de l'environnement</td></tr>
<tr><td>Signature des entités dans l'environnement</td><td colspan="4">identifiant de la signature des entités pour cet environnement</td></tr>
<tr><td rowspan="2">Identifiant</td><td rowspan="2">Valeur retournée</td><td colspan="2">Attribut</td><td rowspan="2">Description</td></tr>
<tr><td>Identifiant</td><td>Type</td></tr>
<tr><td>...</td><td>...</td><td>...</td><td>...</td><td>...</td></tr>
</table>

FIGURE 4.9 – Forme générale de la représentation graphique permettant la spécification de l'environnement.

Une primitive de l'environnement est décrite par un algorithme, pouvant manipuler ses arguments, ainsi que des primitives de l'environnement, des attributs de l'environnement, et éventuellement des primitives abstraites des entités présentes dans ses arguments. La syntaxe de cet algorithme est identique à celle des algorithmes permettant de décrire déclencheur, préconditions et actions d'une interaction. Il en va de même des implications de la présence d'une primitive de l'environnement, d'une primitive d'action ou de perception : pour chaque primitive de l'environnement apparaissant dans cet algorithme, une nouvelle primitive de l'environnement à spécifier est ajoutée à l'ensemble $primitives_{env}$, si elle n'y figure pas déjà. De même, si une primitive d'action y apparaît, alors toutes les familles d'entités de la simulation doivent l'implémenter.

Exemple. Calcul de distance
Le calcul de la distance distance séparant deux entités dans un environnement euclidien à deux dimensions

et non torique peut prendre la forme de la primitive de l'environnement qui suit :

distance(e_1,e_2) *:*
$produitScalaire \Leftarrow \Big(environnement.abscisse(e_1) - environnement.abscisse(e_2)\Big)^2$
$produitScalaire \Leftarrow produitScalaire + \Big(environnement.ordonnee(e_1) - environnement.ordonnee(e_2)\Big)^2$
retourner $\sqrt{produitScalaire}$

Cette primitive fait mention de deux primitives de l'environnement, qu'il faut par la suite définir :
- *la primitive* $\langle abscisse, Nombre\ a\ Virgule\ Flottante, \{Entite\}\rangle$*, qui permet de connaître l'abscisse d'une entité dans l'environnement ;*
- *la primitive* $\langle ordonnee, Nombre\ a\ Virgule\ Flottante, \{Entite\}\rangle$*, qui permet de connaître l'ordonnée d'une entité dans l'environnement.*

Primitives d'action et de perception (« L »). Afin de garantir un certain polymorphisme au sein des interactions, mais aussi de la description du halo des entités, ou au sein des primitives de l'environnement, leurs algorithmes de description manipulent des primitives abstraites. Afin de terminer la spécification du modèle, chaque famille d'entités doit associer un algorithme à l'ensemble de leurs primitives abstraites, déterminé par :
- les interactions présentes dans la matrice d'interaction raffinée, conjointement à la signature de cette famille d'entités dans ces dernières. Si l'interaction $\mathcal{I}$ est présente dans la ligne (respectivement la colonne) associée à une famille d'entités $\mathcal{F} \in \mathbb{F}$, alors cette famille d'entités doit fournir la description de toutes les primitives abstraites présentes dans la signature $sign_{source}(\mathcal{I})$ (respectivement $sign_{cible}(\mathcal{I})$) de cette interaction ;
- les interactions présentes dans la matrice de mise à jour ordonnée, conjointement à la signature de cette famille d'entités dans ces dernières. Si l'interaction $\mathcal{I}$ est présente dans la cellule associée à une famille d'entités $\mathcal{F} \in \mathbb{F}$, alors cette famille d'entités doit fournir la description de toutes les primitives abstraites présentes dans la signature $sign_{source}(\mathcal{I})$ de cette interaction ;
- le halo de la famille d'entités. Une famille d'entités $\mathcal{F} \in \mathbb{F}$ doit implémenter toutes les primitives abstraites présentes dans $sign_{entite}(\mathcal{H}(\mathcal{F}))$;
- les primitives de l'environnement : une famille d'entités doit implémenter toutes les primitives abstraites apparaissant dans l'algorithme décrivant chaque primitive de l'environnement.

À ce stade, on a donc :

$$\forall \mathcal{F} \in \mathbb{F}, primitives(\mathcal{F}) = \left\{ p \;\middle|\; \begin{array}{l} \exists a \in \mathcal{M}(\mathcal{F}, \emptyset) | p \in primitives\Big(sign_{source}\big(\mathcal{I}(a)\big)\Big) \vee \\ \exists \mathcal{T} \in \mathbb{F}, \exists a \in \mathcal{M}(\mathcal{F}, \mathcal{T}) | p \in primitives\Big(sign_{source}\big(\mathcal{I}(a)\big)\Big) \vee \\ \exists \mathcal{S} \in \mathbb{F}, \exists a \in \mathcal{M}(\mathcal{S}, \mathcal{F}) | p \in primitives\Big(sign_{cible}\big(\mathcal{I}(a)\big)\Big) \vee \\ \exists u \in \mathcal{U}(\mathcal{F}) | p \in primitives\Big(sign_{source}\big(\mathcal{I}(u)\big)\Big) \vee \\ p \in sign_{entite}\big(\mathcal{H}(\mathcal{F})\big) \vee \\ p \in sign_{entites}(environnement) \end{array} \right\}$$

La représentation graphique donnée à une famille d'entités correspond alors à la représentation générale décrite dans la figure 4.10.

La seule convention imposée dans la description de telles primitives est de faire apparaître les attributs de la famille d'entités de manière explicite, en préfixant l'identifiant de l'attribut par l'expression clé « this. ».

Exemple. Description d'une primitive « acuiteVisuelle »
L'algorithme qui suit décrit la primitive de perception $\langle acuiteVisuelle, Nombre\ a\ Virgule\ Flottante, \emptyset\rangle$ *pour une famille d'entité percevant les entités situées à une distance inférieure ou égale à leur acuité visuelle.*

acuiteVisuelle() *:*
retourner this.*acuité*

Cette primitive manipule un attribut dont l'identifiant est « acuité ».

« famille d'entités » Identifiant de la famille d'entités				
Identifiant	Valeur retournée	Attribut		Description
		Identifiant	Type	
halo	Booléen	e_1 e_2	Entité Entité	*algorithme décrit lors de l'étape « F »*
...	...	...	...	...

FIGURE 4.10 – Forme générale de la représentation graphique permettant la spécification de l'environnement.

Exemple. Description d'une primitive « acuiteVisuelle »
L'algorithme proposé ici fournit une autre description de la primitive de perception $\langle acuiteVisuelle, Nombre\ a\ Virgule\ Flottante, \emptyset \rangle$*, pour une famille d'entité percevant les entités situées à une distance inférieure ou égale à leur acuité visuelle divisée. De plus, les entités de cette famille considérées comme myopes voient leur acuité visuelle divisée par deux.*

acuiteVisuelle() *:*
| **Si this.***myope* **alors :**
| | **retourner this.***acuité*/2
| **Sinon :**
| | **retourner this.***acuité*
| **FinSi**

Cette primitive manipule deux attributs dont les identifiants sont « acuité » et « myope ».

Attributs des familles d'entités (« K ») et de l'environnement (« M »). Les deux dernières étapes de la méthodologie IODA consistent à terminer la spécification des familles d'entités et de l'environnement, en identifiant leurs attributs. L'identification de ces attributs se basant sur l'analyse des spécifications des diverses primitives de perception, d'action, ou des primitives de l'environnement décrites dans les étapes précédentes.

4.2 Questions issues de la pratique d'une méthodologie de conception

Notre pratique de la méthodologie IODA nous a amenés à retrouver quatre questions récurrentes survenant dans tout type de méthodologie de conception :
- Toute entité est-elle un agent ?
- Une méthodologie de conception doit-elle être dogmatique ?
- Faut-il prendre en compte les performances lors de la modélisation ?
- Faut-il tout décrire au sein d'une règle (dans notre cas, une interaction) ?

Cette section n'a pas pour ambition de résoudre ces problèmes, mais de décrire succinctement le positionnement de notre approche vis à vis de ces problèmes.

4.2.1 Toute entité est-elle un agent ?

La modélisation des agents, et des entités échappant parfois à la définition de « agent », est un point récurrent en simulation informatique, et constitue un problème non trivial à résoudre.

Dans le cas des approches telles que Madkit [GFM01], Netlogo [WC99], RePast [NTCO07] ou Swarm [Ter98], chaque entité est modélisée par un élément logiciel appelé « agent », dont le compor-

tement est spécifié de manière libre en utilisant un langage de programmation. La conception des agent et de leur comportement n'est que très peu aidée. Les concepteurs doivent y définir leur propre représentation des connaissances et/ou implémenter directement le comportement. Elles ne facilitent donc pas la conception de simulations contenant une grande variété d'entités et d'interactions.

D'autres approches telles que Soar [LNR87], Act-R [ABB+04], Jack [BHRH00], Jason [BH06], Maleva [BM07], InteRRaP [FMP95] ou Touring Machines [Fer92] fournissent une représentation explicite aux connaissances des agents. Toutefois, elles se focalisent uniquement sur les agents. Par conséquent, les entités non-proactives sont modélisées comme des agents et gérées comme tels dans l'algorithme de simulation, impliquant de très probables pertes de performances.

Les approches telles que le méta-modèle Agents & Artifacts (A& A) [ORV08], ou la plateforme SeSam [KHF06] fournissent à la fois une représentation explicite aux connaissances des agents et identifient clairement comment spécifier les entités n'étant pas des agents. Pour ce faire, elles différencient agent et autres entités à l'aide de types. Par exemple, dans SeSam, il y a distinction entre le type « agent » et le type « ressource ». De manière similaire, il y a distinction entre « Agent » et « Artefact » dans le méta-modèle Agents & Artifacts. Chaque type définit une méthodologie particulière spécifique pour définir une entité, une façon spécifique de représenter ses connaissances et des algorithmes dédiés permettant de les implémenter.

Bien que l'attribution de types aide la conception de tout type d'entité dans la simulation, et permette d'optimiser les simulateurs obtenus, elle ajoute aussi des problèmes qui, contrairement à leur vocation initiale, rend la conception de simulation plus complexe. Pour s'en convaincre, nous illustrons dans cette section trois de ces problèmes. Puisque d'une approche à l'autre, les types utilisés peuvent avoir des sens très différents, nous introduisons dans un premier temps une terminologie qui permettra de les caractériser et les comparer.

Terminologie caractérisant l'activité des entités

Nous considérons que l'activité d'une entité dans une simulation multi-agents peut être caractérisée par la présence d'aucune, une, deux ou toutes les capacités suivantes [KMP10] :

- la capacité d'*agir sur les autres entités ou sur l'environnement*, qui caractérise ce que nous appelons **entités actives**. Par exemple un piéton qui se déplace sur une route, un globule blanc qui combat les infections, un acheteur émettant des ordres sur un marché, *etc.*
- la capacité de *subir les actions d'une autre entité, ou d'être utilisée comme un outil par une autre entité*, qui caractérise ce que nous appelons **entités passives** . Par exemple un interrupteur qui est allumé ou éteint par un utilisateur, un fournisseur d'informations dans un aéroport (aussi bien un réceptionniste qu'une borde d'information), ou encore une porte permettant à une personne de transiter d'une pièce à une autre, *etc.*
- la capacité de *changer leur propre état sans agir ou subir d'action*, qui caractérise ce que nous appelons **entités labiles**. Par exemple de la nourriture, dont les propriétés nutritives diminuent avec le temps, une entité dont la sensation de faim augmente avec le temps, *etc.*

Une entité peu exprimer une ou plusieurs de ces capacités. Par exemple, en éthologie, des animaux pouvant se reproduire sont au moins actives (puisqu'elles peuvent initier la reproduction) et passives (puisqu'elles peuvent subir la reproduction). Une entité peut aussi n'exprimer aucune de ces capacités. Par exemple, un rideau dont le seul rôle dans la simulation est de contraindre la perception des entités. Puisque ce cas est récurrent en simulation, nous complétons notre terminologie par une valuation particulière des capacités mentionnées ici, que nous appelons **entités permanentes**, qui représente les entités n'étant **ni actives, ni passives, ni labiles**. Cette terminologie est résumée dans le tableau 4.1.

La plupart des types utilisés en simulation multi-agents correspondent alors à une **valuation statique** de ces capacités. Par exemple, une entité dont le type est « artefact » dans [ORV08] correspond selon notre terminologie à une entité passive. Une entité dont le type est « ressource » dans [KHF06] correspond selon notre terminologie soit à une entité passive, soit à une entité permanente. En fin, notre terminologie permet d'identifier que le type « agent » n'a pas le même sens dans [ORV08] et [FM96] : d'après notre terminologie, un agent est une entité active et passive dans le premier cas et une entité active, passive et labile dans le second.

	Définition	**Capacité correspondante**	**Exemple**
Active	Qui manifeste de l'activité, de l'énergie, qui est capable d'agir	Agir sur les autres entités ou sur l'environnement	Un globule blanc combattant les infections
Passive	Qui subit l'action, par opposition à Actif	Subir les actions d'une autre entité, ou d'être utilisée comme un outil par une autre entité	Un interrupteur qui est allumé ou éteint par un utilisateur
Labile	Qui est peu stable ; qui est sujet à se transformer	Changer leur propre état sans agir ou subir d'action	Une entité dont la sensation de faim augmente avec le temps
Permanente	Établi de manière durable et continue, sans interruption ni modification	Ni active, ni passive, ni labile	Un rideau dont le seul rôle dans la simulation est de contraindre la perception des entités

TABLE 4.1 – Résumé des différentes capacités qu'une entité d'une simulation peut exprimer. Les définition fournies dans ce tableau proviennent du dictionnaire en ligne de l'académie française [fra10].

D'autres types sont utilisés pour différencier d'autres propriétés des entités. Par exemple, la seule différence existant entre les types « turtle » et « patch » dans Netlogo [WC99] est leur représentation dans l'environnement : dans le premier cas, il s'agit d'un point mobile et dans le second un carré de taille fixe et immobile. Dans de tels cas, l'usage de types peut être évité si la représentation des entités dans l'environnement est utilisé comme un attribut de l'environnement ou des entités.

Problèmes liés à l'utilisation de types

Dans les approches telles que le méta-modèle A& A, l'un des points cruciaux de la conception d'une simulation consiste à déterminer quel type utiliser pour modéliser une entité. En effet, en fonction de ce type, une structure de données spécifique est utilisée pour représenter l'entité. Elle est de plus spécifiée à l'aide d'une méthodologie dédiée.

Dans cette sous-sous-section, nous illustrons de trois points de vue différents pourquoi l'usage de types rend la conception de simulations moins intuitive et plus difficile. Puisque le sens de types tels que « agents » ou « artefact » est fortement dépendant de l'approche d'où ils ont été extraits, les différents types mentionnés jusqu'à présent sont identifiés par la suite à l'aide de la terminologie introduite dans la sous-sous-section précédente, afin d'éviter toute ambiguïté dans nos propos.

Conception incrémentale de simulations Supposons qu'une simulation mette en jeu des personnes qui se déplacent, ainsi que des murs et portes qui cachent les personnes cachées derrière elles. On peut supposer sans risques que les personnes sont modélisées par des entités au moins actives, puisqu'elles se déplacent. Les murs et les portes semblent n'influer que la perception de personnes. Par conséquent, ce sont des entités permanentes. Si l'on considère que les portes sont un moyen utilisé par une personne pour passer d'une pièce à une autre, elles ne sont plus de simples entités permanentes : elles deviennent un outil utilisé par les personnes et donc des entités passives.

Cette simulation peut être modélisée en deux phases :

1. déplacement des personnes et leur perception. Les personnes peuvent ainsi se mettre dans une situation les poussant à utiliser les portes ;
2. utilisation des portes comme outil de téléportation.

La porte passe ainsi d'entité permanente (utilisée uniquement pour la perception) lors de la première itération, à passive (un outil de téléportation) lors de la seconde.

D'une itération de conception à l'autre, le type d'une entité peut donc évoluer. Puisque ce type conditionne comment l'entité est modélisée et quelle structure de données utiliser, il sera nécessaire de la re-spécifier et la ré-implémenter presque complètement.

Cette situation survient également dans le cas où un modèle est révisé. Elle survient aussi dans des simulations large échelle, *i.e.* de simulations contenant une grande variété d'entités pouvant effectuer et subir des actions diversifiées. En effet, pour illustrer de manière intuitive le problème de la conception incrémentale, un problème de taille réduite a été utilisé, ne nécessitant la modélisation que de peu d'entités et d'interactions. Cela a permis de trouver le type des différentes entités dès le départ. Dans des problèmes réels de simulation, le nombre d'entités et d'actions peut être bien plus grand, il n'est pas possible de

connaître *a priori* toute les fonctionnalités de la simulation. Par conséquent, de telles simulations sont construites par étapes, par une succession de modèles de complexité croissante. Chaque étape introduit de nouvelles informations dans le modèle, et rend la révision du type de certaines entités inévitable.

Types et concepts naturels L'argument le plus couramment utilisé pour justifier l'utilisation du paradigme agent pour modéliser des simulations repose sur l'ontologie qu'il propose, considérée comme proche de celle des phénomènes simulés [Edm05]. Dans ce contexte, les entités actives telles que les « agents » sont souvent interprétées comme « entité animée », « entité mobile » ou « entité vivante » puisque les entités actives sont supposées percevoir et décider par elles-mêmes des actions qu'elles initient. La notion de type entre en contradiction avec un tel argument.

Considérons par exemple une extension de la simulation présentée plus haut, dans laquelle un mur ait la possibilité de s'écrouler sur une personne. Le mur est capable d'effectuer une action et devient par conséquent une entité active. Bien qu'un mur soit interprété naturellement comme une « entité inanimée », il peut avoir la capacité d'initier des actions et peut donc être une entité active. Il en va de même pour les portes automatiques, les contrôleurs de vitesses sur une route, *etc.*

La distinction entre entités actives, passives, labiles et permanentes est opérationnelle. De tels concepts ne s'associent pas intuitivement avec des concepts naturels tels que « entité animée », « entité mobile » ou « entité vivante ». Ce problème peut sembler insignifiant du point de vue d'un expert en informatique, qui approchent la conception d'une simulation d'un point de vue opérationnel. Toutefois, la simulation est un outil au service d'experts du domaine simulé (*i.e.* biologistes, sociologues, *etc.*) qui ne partagent pas systématiquement ce point de vue. Ainsi, le choix de la notion à utiliser pour spécifier une entité est, dans certains cas, contre-intuitif et fait défaut à la conservation d'une ontologie proche des phénomènes simulés, qui est pourtant l'un des principaux intérêts de l'utilisation de systèmes multi-agents dans le cas de simulations.

Entités hybrides L'utilisation de types impose une division statique de l'espace des représentations possibles pour les entités. Dans ce contexte, la conception d'une entité hybride, *i.e.* devant exprimer les capacités exprimées dans deux types différents, peut poser problème.

Considérons par exemple une simulation faisant intervenir des employés de magasin pouvant endosser deux rôles différents :

- fournir des information aux clients lorsqu'ils sont interpellés (l'entité est alors un *informateur*) ;
- ordonner des articles sur des étagères (l'entité est alors un *trieur*) ;

Un informateur ne fait que répondre aux requêtes des clients et est donc une entité passive. Un trieur est capable d'agir dans l'environnement et est par conséquent une entité active. Un employé peut aussi être à la fois un trieur et un informateur. Dans ce cas, il est à la fois une entité passive et une entité active.

Se pose alors une question : si un trieur est représenté à l'aide d'un type T_1 et un informateur à l'aide d'un type T_2, qu'en est il d'un employé qui est à la fois trieur et informateur ? Différentes solutions existent à ce problème, en fonction de la combinaison des capacités de notre terminologie exprimées par un type (voir figure 4.11).

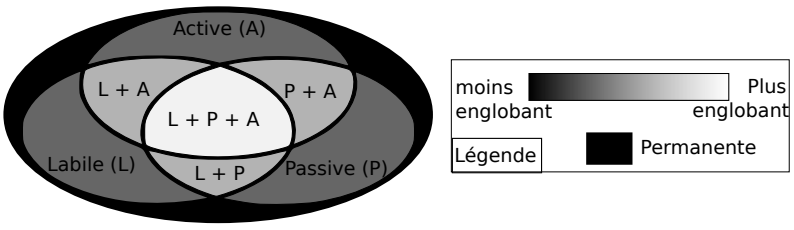


FIGURE 4.11 – Caractérisation des différentes combinaisons de compétences décrivant l'activité d'une entité d'une simulation, selon la terminologie décrite dans la section 4.2.1.

Dans le cas idéal reposant sur les types, chaque combinaison possible de capacités est modélisé par un type. Il est alors possible de représenter toute entité de la simulation. À notre connaissance, aucune approche ne permet ceci.

Un cas intermédiaire survient lorsqu'une entité est caractérisée par une combinaison de capacités qu'aucun type n'exprime directement, mais qu'il existe un type représentant un sur-ensemble de la combinaison recherchée. Dans ce cas, le type exprimant la plus petite combinaison de capacité englobantes est utilisé pour modéliser l'entité, réduisant ainsi les performances du simulateur. Par exemple, une entité uniquement labile peut être modélisée à l'aide d'un type représentant des entités labiles et actives, où l'activité de l'entité est nulle. Ce cas se retrouve par exemple dans la plateforme SeSam (voir figure 4.12a), où les entités permanentes ou passives sont représentées à l'aide du type « ressource » et où les entités exprimant toute autre combinaison de capacités sont représentées avec le type « agent ».

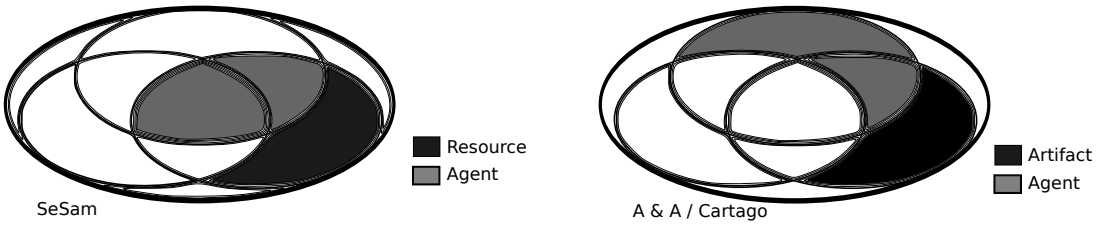


FIGURE 4.12 – Caractérisation des différentes combinaisons de compétences d'une entité exprimées par les types de la plateforme SeSam et du méta-modèle A&A. Elle se base sur la représentation utilisée sur la figure 4.11.

Dans le pire des cas, certaines combinaisons de capacités ne peuvent pas être exprimées. C'est par exemple le cas du méta-modèle A&A, où les entités labiles ne peuvent être spécifiées (voir figure 4.12b).

Ainsi, l'utilisation de types peut, dans certains cas, restreindre l'ensemble des simulations qu'il est possible de modéliser.

Modélisation des entités et agents dans IODA

L'approche IODA fournit une représentation unifiée des entités, basée sur une identification dynamique et transparente des différentes capacités de la terminologie résumée sur le tableau 4.1. En effet, dans notre méthodologie, **ces capacités sont déduites de la structure de la matrice d'interaction brute** et de la matrice de mise à jour :

- Une entité e est dite *active* si la ligne associée à sa famille d'entités dans la matrice d'interaction brute contient au moins un élément d'assignation, *i.e.* $\exists \mathcal{T} \in \mathbb{F} \cup \{\emptyset\} | \mathcal{M}\Big(famille(e), \mathcal{T}\Big) \neq \emptyset$;
- Une entité e est dite *passive* si la colonne associée à sa famille d'entités dans la matrice d'interaction brute contient au moins un élément d'assignation, *i.e.* $\exists \mathcal{S} \in \mathbb{F} | \mathcal{M}\Big(\mathcal{S}, famille(e)\Big) \neq \emptyset$;
- Une entité e est dite *labile* si la cellule associée à sa famille d'entités dans la matrice de mise à jour contient au moins un élément d'assignation, *i.e.* $\mathcal{U}\Big(famille(e)\Big) \neq \emptyset$.

Par conséquent, la conception incrémentale d'une simulation est mieux supportée qu'avec des types. En effet, une entité peut passer de non-active à active, de non-passive à passive, ou de non-labile à labile de manière complètement transparente, par la simple modification de ce que l'entité est capable de faire, sans que cela implique une re-spécification complète ou partielle de cette entité. De plus, tout type d'entité hybride peut être modélisé, puisque toute combinaison caractérisant l'activité d'une entité peut être exprimée dans notre modèle (voir figure 4.13). Enfin, la notion de type n'existe pas dans notre approche. Le fait qu'une entité est labile, active, passive ou permanente est déduit directement du contenu du modèle et ne doit donc pas être spécifié explicitement par le concepteur. Par conséquent, il n'y a plus de confusion possible entre l'ontologie induite par les types et l'ontologie provenant du phénomène modélisé : chaque entité apparaissant physiquement dans le phénomène est représentée par une entité dans IODA.

L'approche IODA fournit donc une représentation unifiée des entités de la simulation. Puisque cette approche se place dans le cadre de la simulation multi-agents, nous appelons « agent » cette représentation.

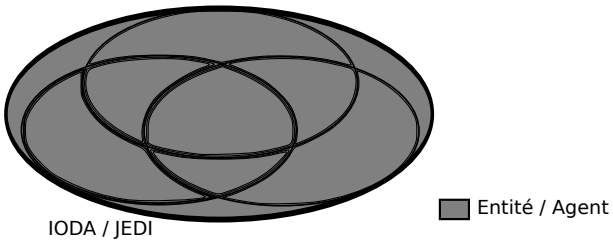
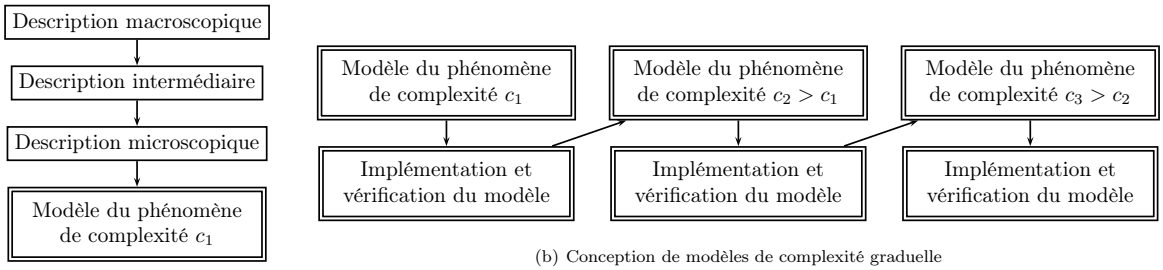


FIGURE 4.13 – Caractérisation des différentes combinaisons de compétences d'une entité pouvant être exprimées avec l'approche IODA. Elle se base sur la représentation utilisée sur la figure 4.11.

4.2.2 Une méthodologie de conception doit-elle être dogmatique ?

L'objectif d'une méthodologie de conception est de fournir un cadre formel facilitant la description un modèle. Cette simplification se fait au travers de deux traits de la méthodologie : la possibilité de construire un modèle en y introduisant de manière progressive des connaissance (en passant de descriptions macroscopiques à des descriptions microscopiques), et la possibilité d'atteindre le modèle souhaité en construisant des modèles de complexité croissante (par exemple un modèle dans lequel les entités peuvent se déplacer, puis un modèle dans lequel les entités peuvent interagir de manière plus diversifiée). Ces deux traits sont schématisés sur la figure 4.14.



(a) Conception graduelle d'un modèle

(b) Conception de modèles de complexité graduelle

FIGURE 4.14 – Cheminements pouvant être employés par une méthodologie de conception de simulation afin de concevoir le modèle d'un phénomène.

Ces schémas permettent d'illustrer les principes généraux simplifiant la conception de modèles complexes, *i.e.* de modèles contenant un grand nombre de familles d'entités interagissant de manière variée, mais ne constituent en aucun cas un dogme qu'il est nécessaire de respecter tel quel. En effet, la construction d'un modèle est un procédé itératif, lors duquel certaines descriptions microscopiques peuvent amener à réviser les descriptions macroscopiques, qui se révélaient incomplètes. Illustrons ce propos à l'aide de la méthodologie IODA.

Reprenons par exemple le cas d'une simulation d'un écosystème, dans lequel des proies sont mangées par des prédateurs. Une première étude d'une telle simulation aboutit à une matrice d'interaction raffinée, ainsi qu'à une matrice de mise à jour ordonnée décrites dans la figure 4.15. Ces matrices spécifient que le comportement des entités consiste en priorité à mourir si jamais elle le doivent, sinon à se reproduire ou à manger d'autres entités, et à se déplacer par défaut si aucune autre interaction ne pouvait être initiée. La description du déclencheur de l'interaction MANGER (voir figure 4.4) mène à l'identification d'une primitive dont l'identifiant est « a faim », ainsi qu'une autre primitive dont l'identifiant est « diminuer sensation faim ». Ces primitives impliquent implicitement que la sensation de faim d'une entité augmente avec le temps indépendamment du comportement des entités. En conséquence, la matrice de mise à jour est révisée, pour devenir celle décrite dans la figure 4.16. Lors la description de la primitive de perception « a faim » de la famille d'entités `Prédateur` (voir figure 4.17), il est spécifié que la sensation de faim est corrélée à l'âge d'une entité : plus une entité est vieille, moins elle ressent la sensation de faim. Puisque l'âge intervient dans le calcul de la sensation de faim, il se révèle nécessaire de le mettre à jour via une

Source \ Cible	∅	Proie	Prédateur
Proie	(Se Déplacer, p=0) (Mourir, p=6)	(Se Reproduire, d=0cm, p=3)	
Prédateur	(Se Déplacer, p=0) (Mourir, p=6)	(Manger, d=15cm, p=3)	(Se Reproduire, d=0cm, p=3)

(a) Matrice d'interaction raffinée

	Proie	Prédateur
Interactions de mise à jour		

(b) Matrice de mise à jour ordonnée

FIGURE 4.15 – Exemple d'une matrice d'interaction raffinée, ainsi que d'une matrice de mise à jour d'une simulation de type proie/prédateur, lors d'une première phase de la méthodologie IODA.

Source \ Cible	∅	Proie	Prédateur
Proie	(Se Déplacer, p=0) (Mourir, p=6)	(Se Reproduire, d=0cm, p=3)	
Prédateur	(Se Déplacer, p=0) (Mourir, p=6)	(Manger, d=15cm, p=3)	(Se Reproduire, d=0cm, p=3)

(a) Matrice d'interaction raffinée

	Proie	Prédateur
Interactions de mise à jour		(Augmenter sensation faim, p=1)

(b) Matrice de mise à jour ordonnée

FIGURE 4.16 – Exemple d'une matrice d'interaction raffinée, ainsi que d'une matrice de mise à jour d'une simulation de type proie/prédateur, lors d'une seconde phase de la méthodologie IODA.

« famille d'entités » `Predateur`				
Identifiant	Valeur retournée	Attribut		Description
		Identifiant	Type	
halo	Booléen	e_1 e_2	Entité Entité	...
aFaim	Booléen	—	—	Retourner **this.**$energie\times$ **this.**$age <$ **this.**$seuilFaim$.
...	...	...	...	...

FIGURE 4.17 – Exemple de spécification de l'interaction MANGER dans le cas d'une simulation de type proie/prédateur.

interaction de mise à jour. Il faut donc de réviser la matrice de mise à jour ordonnée telle que présentée jusqu'à présent (voir figure 4.18).

Source \ Cible	∅	Proie	Prédateur
Proie	(Se Déplacer, p=0) (Mourir, p=6)	(Se Reproduire, d=0cm, p=3)	
Prédateur	(Se Déplacer, p=0) (Mourir, p=6)	(Manger, d=15cm, p=3)	(Se Reproduire, d=0cm, p=3)

(a) Matrice d'interaction raffinée

	Proie	Prédateur
Interactions de mise à jour		(Augmenter sensation faim, p=1) (Vieillir, p=2)

(b) Matrice de mise à jour ordonnée

FIGURE 4.18 – Exemple d'une matrice d'interaction raffinée, ainsi que d'une matrice de mise à jour d'une simulation de type proie/prédateur, lors d'une troisième phase de la méthodologie IODA.

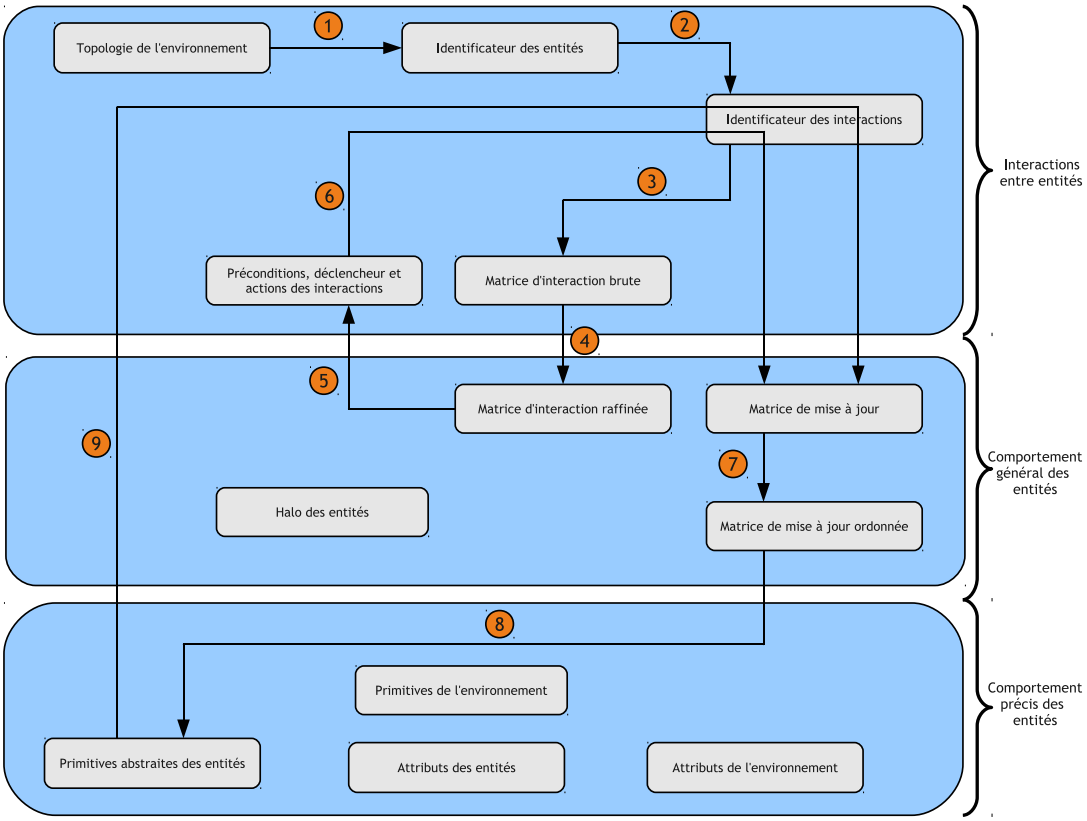
Le cheminement décrit ici transite entre aspects microscopiques et microscopiques de la spécification selon le schéma de la figure 4.19. On y constate que le schéma suivi n'est pas aussi strict que celui présenté dans la figure 4.19, mais il en respecte toutefois les principes fondamentaux : une spécification au niveau intermédiaire fait suite à une spécification au niveau macroscopique, et une spécification au niveau microscopique fait suite à une description au niveau intermédiaire. Dès lors, il est difficile d'établir un ordre strict entre les différentes étapes du processus de conception d'un modèle. Nous avons donc choisi de décrire le processus de conception d'un modèle uniquement par les dépendances existant entre les différentes étapes de la méthodologie IODA (résumées par la figure 4.1). La forme de la méthodologie IODA ayant été décrite dans la première section de ce chapitre est l'aboutissement de différents travaux cherchant à appréhender au mieux le problème décrit dans cette section. Elle fait suite à différentes descriptions imposant un ordre aux différentes étapes de la simulation [KMP08a, KMP08c], qui se sont révélées ne pas appréhender de manière satisfaisante le présent problème.

4.2.3 Faut-il prendre en compte les problèmes de performances lors de la conception du modèle d'un phénomène ?

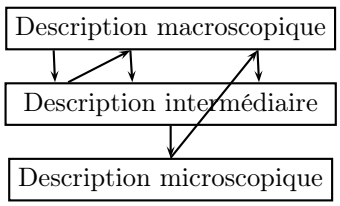
La viabilité des hypothèses émises sur le phénomène simulé ne se mesure pas par les résultats d'une seule expérimentation, mais par les résultats moyens obtenus par plusieurs expériences. Plus le nombre de ces expériences est élevé, plus résultats sont viables. Par conséquent, bien qu'annexe, l'optimisation du simulateur utilisé pour réaliser des expérience reste cruciale lors de la conception de simulations. La façon de structurer un modèle influe grandement sur les performances du simulateur produit. S'en suit un paradoxe : le modèle doit pouvoir être spécifié au maximum par des non-informaticiens, ce qui implique la quasi-absence d'optimisations dans la structure du modèle, mais doit aussi pouvoir rendre efficace le simulateur sous-jacent, sans quoi la simulation risque de ne pas pouvoir être exécutée, faute d'obtention de résultats suffisamment rapide. Ce problème a amené [MEU04] à ajouter un modèle transitoire, appelé modèle opérationnel, afin de prendre en compte ces optimisations sans pour autant altérer la nature du modèle spécifié par les experts du domaine.

Dans notre approche, cette distinction n'est pas faite. Cela ne rend pas pour autant impossible la spécification du modèle par des experts du domaine. Nous illustrons ceci sur un exemple : la spécification de la notion de distance entre deux entités.

Considérons la spécification suivante, décrivant comment la distance entre deux entités est calculée dans un environnement euclidien à deux dimensions non torique, où les entités sont assimilées à des points :



(a) Cheminement précis dans la méthodologie Ioda



(b) Cheminement équivalent dans le schéma issu de la figure 4.14

FIGURE 4.19 – Cheminement utilisé pour décrire une partie du modèle de type proie/prédateur illustré dans la section 4.2.2.

distance(e_1,e_2) :
$produitScalaire \Leftarrow \Big(environnement.abscisse(e_1) - environnement.abscisse(e_2)\Big)^2$
$produitScalaire \Leftarrow produitScalaire + \Big(environnement.ordonnee(e_1) - environnement.ordonnee(e_2)\Big)^2$
retourner $\sqrt{produitScalaire}$

Il est intéressant de remarquer que, dans l'exemple précédent, les coordonnées d'une entité sont connues à l'aide des primitives de l'environnement $\langle abscisse, Nombre\ a\ Virgule\ Flottante, \{Entite\}\rangle$ et $\langle ordonnee, Nombre\ a\ Virgule\ Flottante, \{Entite\}\rangle$, pouvant être implémentées de plusieurs manières. Une première façon serait de considérer que l'environnement contienne une structure de données associant à chaque entité ses coordonnées :

abscisse(e) :
retourner **this.**$abscisses[e]$

Une autre façon serait de considérer que ces primitives fassent appel à une primitive abstraite des entités, permettant de connaître leurs coordonnées :

abscisse(e) :
retourner e**.**$abscisse()$

Dans le premier cas, chaque lecture ou modification d'une coordonnée implique le parcours d'une structure de données plus ou moins efficace, qui nécessite une quantité de calculs corrélée au nombre d'entités dans la simulation. Plus ce nombre est élevé, moins ce calcul est efficace. Dans le second cas, l'utilisation d'une structure de données n'est plus nécessaire, optimisant d'autant le calcul de la distance entre deux entités.

Dans la méthodologie IODA, les optimisations ont lieu en pratique lors de la spécification des primitives abstraites des entités ou de l'environnement. Elle a donc lieu à la fin du processus de conception d'un modèle. Toutes les étapes préalables peuvent être faites sans prêter attention aux optimisations et peuvent donc être spécifiées par des experts du domaine.

4.2.4 Faut-il tout décrire au sein d'une règle/interaction ?

Dans IODA, le recensement des identificateurs des interactions, la description de leurs préconditions, actions et déclencheur, ou la description des primitives de l'environnement et des primitives abstraites amène à se questionner sur le contenu de chacun de ces éléments. En effet, un même comportement peut être décrit de manière très différentes au sein d'un modèle. Il se pose en la question de quand utiliser des primitives abstraites, et quand écrire directement une spécification précise, ne permettant pas de polymorphisme.

Par exemple, la description du déclencheur d'une interaction manger peut aller de très abstrait et générique (voir figure 4.20a), et donc réutilisable dans un maximum de cas en lui faisant perdre de son expressivité, à très spécifique et précis (voir figure 4.20b), et donc très expressive mais peu réutilisable, en passant par des descriptions intermédiaires (voir figure 4.20c), qui fournit un compromis entre les deux cas précédemment mentionnés.

declencheur(e_1,e_2) :
retourner e_1**.**$veutManger(e_2)$

(a) Spécification très abstraite

declencheur(e_1,e_2) :
retourner e_1**.**$get\ energie()$< e_1**.**$seuil\ de\ faim()\wedge$
e_2**.**$valeur\ energetique()$> e_1**.**$seuil\ d\ interet()$

(b) Spécification très spécifique

declencheur(e_1,e_2) :
retourner e_1**.**$a\ faim()\wedge e_1$**.**$faim\ significativement\ reduite\ par(e_2)$

(c) Spécification intermédiaire

FIGURE 4.20 – Différentes façons de spécifier le déclencheur d'une interaction MANGER, dans le contexte d'une simulation en éthologie, cherchant à modéliser un écosystème.

Ce problème n'est toutefois pas restreint à la méthodologie IODA. Il se pose aussi dans des approches telles que :

- l'architecture de subsomption [Bro86] lorsqu'il faut déterminer si une partie du comportement d'un robot doit être décrit au sein d'un module, ou donner lieu à la création d'un second module ;
- les architectures à planification réactive telles que Jack [BHRH00] lorsqu'il faut déterminer s'il faut décrire une partie du comportement d'un agent au sein d'un seul plan ou me décomposer en des sous-plans.

Le choix de la façon de spécifier une interaction, une primitive de l'environnement ou une primitive abstraite est un problème difficile, qui dépend principalement du compromis recherché par le concepteur. En effet, la délégation d'une partie de la spécification d'une interaction à des primitives abstraites rend l'interprétation directe des interactions moins aisée. Elle permet toutefois d'obtenir une grande diversité comportementale en un nombre réduit de modifications du modèle, tout en favorisant son utilisation dans des contextes différents. L'approche IODA offre la possibilité de personnaliser ces éléments finement, à l'aide des différents niveaux d'abstraction qu'elle propose.

4.3 Simulation d'un modèle centré sur les interactions

La construction d'un modèle n'est qu'une des deux étapes de la construction d'une simulation à l'aide d'une approche transversale. La seconde consiste à définir par quel procédé il est possible de passer d'un modèle à une implémentation.

Dans l'approche IODA, **nous soutenons que l'implémentation doit conserver la même structure que le modèle, ainsi que sa terminologie**. Sous réserve que cette structure soit conservée, il est possible de décrire des algorithmes tirant parti des informations contenues dans le modèle pour produire automatiquement une simulation, en conservant un morphisme presque total entre éléments du modèle et éléments de l'implémentation. Bien entendu, une implémentation ne préservant pas ce morphisme est possible, mais réduit significativement l'intérêt de notre approche.

Cette section décrit les algorithmes permettant d'implémenter un modèle décrit avec le formalisme présenté dans le chapitre 3 et utilisant la terminologie introduite dans la section 4.2.1 du chapitre courant résumée dans le tableau 4.1 page 111. Nous choisissons de décomposer la présentation de ces algorithmes en deux parties. Dans une première partie, nous décrivons comment sont utilisées les notions de halo, de matrice d'interaction brute et matrice d'interaction raffinée afin de construire le comportement d'une entité. Dans une seconde partie, nous décrivons comment implémenter une simulation dans le cas particulier du temps discret.

4.3.1 Processus comportemental d'une entité

Le processus utilisé par une entité pour déterminer l'interaction qu'elle souhaite initier est indépendant de la représentation du temps utilisée. Il suit un processus générique en quatre étapes résumé dans la figure 4.21 : la perception du voisinage de l'entité, le recensement des interactions que l'entité est capable d'initier dans son contexte actuel, la sélection de l'interaction initiée en fonction de son modèle comportemental et enfin l'exécution l'interaction. Dans cette section, nous proposons de décrire les concepts et algorithmes permettant d'implémenter ces différentes étapes.

Perception du voisinage

Avant de pouvoir déterminer les interactions qu'elle a l'opportunité d'effectuer, une entité doit déterminer quelles sont les entités présentes dans son voisinage. Ce calcul, décrit dans l'algorithme 1, s'appuie sur le halo de la famille de cette entité, ainsi que sur l'ensemble des entités pouvant être perçues dans l'environnement, qui sont caractérisées par la capacité de pouvoir subir au moins une interaction. Cet ensemble correspond donc à l'ensemble $\mathbb{E}_{passive}$ des entités passives de la simulation.

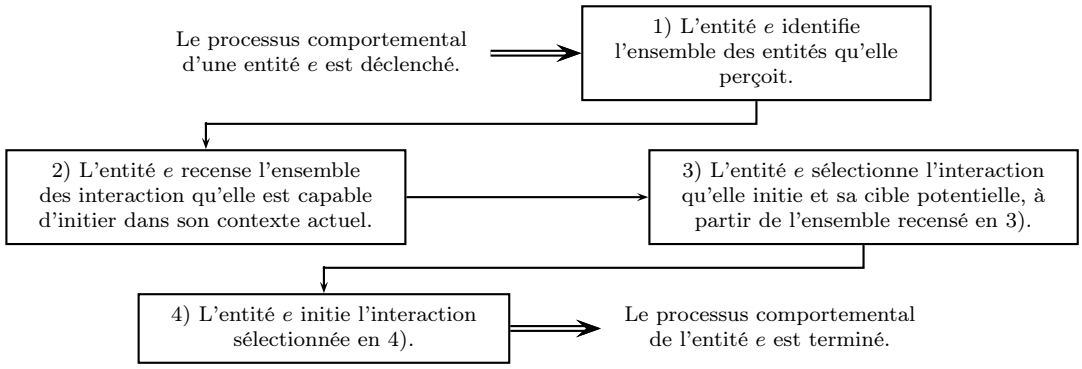


FIGURE 4.21 – Principes régissant le comportement d'une entité e dans l'approche IODA.

Algorithme 1 : Algorithme décrivant comment est calculé du voisinage d'une entité e. L'ensemble $\mathbb{E}_{passive}$ représente l'ensemble des entités passives présentes dans l'environnement.

e**.recenser voisinage() :**
début
  $\mathcal{V} \Leftarrow \emptyset$;
  pour $e' \in \mathbb{E}_{passive}$ **faire**
    si $e' \neq e$ ***et*** $halo\Big(\mathcal{H}\big(famille(e)\big)\Big)\Big(e,\, e'\Big) == vrai$ **alors**
      $\mathcal{V} \Leftarrow \mathcal{V} \cup \{e'\}$;
  retourner $\mathcal{V}$;
fin

Recensement des interactions qu'une entité a l'opportunité d'initier

Afin de construire l'ensemble des interactions qu'une entité a l'opportunité d'initier dans sont contexte actuel, nous introduisons quatre notions : les *tuples*, le critère d'éligibilité d'un *tuple*, le critère de *réalisabilité* d'un tuple et le *Potentiel d'interaction* d'une entité.

Les interactions initiées effectivement par une entité source sont soit des interactions dégénérées, soit des interactions individuelles auxquelles est associée une entité du voisinage de la source en tant que cible. Puisqu'une interaction qu'une entité a la capacité d'initier est définie par un élément d'assignation, nous représentons une interaction qu'une entité peut potentiellement effectuer par un tuple, associant à un élément d'assignation une entité source et, si l'élément d'assignation n'est pas dégénéré, une entité cible.

Définition 29. ***Tuple éligible***
Soient $s \in \mathbb{E}$ une entité dont la famille est $\mathcal{F}_S$, $t \in \mathbb{E}$ une entité dont la famille est $\mathcal{F}_T$, $a_{indiv} = (\mathcal{I}, d, \mathcal{F}_S, \mathcal{F}_T) \in \mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}$ un élément d'assignation individuel et $a_{deg} = (\mathcal{I}', \mathcal{F}_S) \in \mathbb{I}_{(1,0)} \times \mathbb{F}$ un élément d'assignation dégénéré.
Un tuple $\mathcal{T} = (a_{indiv}, s, t)$ est dit **éligible** si :
- $a_{indiv} \in \mathcal{M}(\mathcal{F}_S, \mathcal{F}_T)$, *i.e.* si s a la capacité d'initier l'interaction individuelle associée à l'élément d'assignation a_{indiv} avec t pour cible ;
- $environnement.distance(s,t) \leq dist(a_{indiv})$, *i.e.* si la distance entre s et t est inférieure ou égale à la garde de distance associée à l'élément d'assignation a_{indiv} ;
- $t \in \mathcal{V}(e)$, *i.e.* si l'entité t appartient au voisinage de l'entité s.

De même, un tuple $\mathcal{T} = (a_{deg}, s)$ est dit **éligible** si s a la capacité d'initier l'interaction dégénérée associée à l'élément d'assignation a_{deg}, *i.e.* $a_{deg} \in \mathcal{M}(\mathcal{F}_S, \emptyset)$

On dit qu'un tuple est éligible si l'entité source a la capacité d'initier l'interaction du tuple. Ce critère est syntaxique et se fonde sur l'interprétation de la matrice d'interaction brute et l'exploitation du voisinage d'une entité. Il ne porte pas sur la sémantique de l'interaction lui étant associée. Ainsi,

les préconditions ou le déclencheur d'une interaction associée à un tuple éligible peuvent très bien être fausses pour la source et la cible éventuelle contenues dans le tuple. La validité sémantique d'un tuple est vérifiée à l'aide du critère de *réalisabilité*.

Définition 30. ***Tuple réalisable***
Soient $s \in \mathbb{E}$ une entité dont la famille est $\mathcal{F}_S$, $t \in \mathbb{E}$ une entité dont la famille est $\mathcal{F}_T$, $a_{indiv} = (\mathcal{I}, d, \mathcal{F}_S, \mathcal{F}_T) \in \mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}$ un élément d'assignation individuel et $a_{deg} = (\mathcal{I}', \mathcal{F}_S) \in \mathbb{I}_{(1,0)} \times \mathbb{F}$ un élément d'assignation dégénéré.
Le tuple $\mathcal{T} = (a_{indiv}, s, t)$ est dit **réalisable** si :
– $\mathcal{T}$ est éligible, et ;
– $\mathcal{I}.preconditions(s,t) = vrai$, *i.e.* les préconditions de l'interaction $\mathcal{I}$ sont vérifiées pour les entités s et t, et ;
– $\mathcal{I}.declencheur(s,t) = vrai$, *i.e.* le déclencheur de l'interaction $\mathcal{I}$ est vérifié pour les entités s et t ;
De même, le tuple $\mathcal{T} = (a_{deg}, s)$ est dit **réalisable** si :
– $\mathcal{T}$ est éligible, et ;
– $\mathcal{I}.preconditions(s) = vrai$, *i.e.* les préconditions de l'interaction $\mathcal{I}$ sont vérifiées pour l'entité s, et ;
– $\mathcal{I}.declencheur(s) = vrai$, *i.e.* le déclencheur de l'interaction $\mathcal{I}$ est vérifié pour l'entité s ;

Les interactions qu'une entité $s \in \mathbb{E}$ peut potentiellement initier dans son contexte actuel est alors décrit par l'ensemble de tous les tuples réalisables où l'entité s est la source. Nous appelons cet ensemble particulier le *potentiel d'interaction* de l'entité s.

Définition 31. ***Potentiel d'interaction d'une entité***
On appele **potentiel d'interaction** d'une entité $e \in \mathbb{E}$ l'ensemble des tuples réalisables où cette entité apparaît comme source. Cet ensemble décrit toutes les interactions que l'entité a l'opportunité d'initier dans son contexte actuel.

L'algorithme 2 décrit comment calculer le potentiel d'interaction d'une entité, en se reposant directement sur les éléments déclaratifs du modèle IODA décrits dans le chapitre 3.

Algorithme 2 : Algorithme permettant de calculer le potentiel d'interaction d'une entité x.

potentiel d'interaction(x)
début
  $\mathcal{R} \Leftarrow \emptyset$;
  % Recensement des tuples contenant des interactions individuelles
  pour tous les $y \in \mathcal{V}(x)$ **faire**
    pour tous les $a \in \mathcal{M}\Big(famille(x), famille(y)\Big)$ **faire**
      si (a, x, y) *est réalisable* **alors**
        $\mathcal{R} \Leftarrow \mathcal{R} \cup \{(a, x, y)\}$;
  % Recensement des tuples contenant des interactions dégénérées
  pour tous les $a \in \mathcal{M}\Big(famille(x), \emptyset\Big)$ **faire**
    si (a, x) *est réalisable* **alors**
      $\mathcal{R} \Leftarrow \mathcal{R} \cup \{a, x\}$;
  retourner $\mathcal{R}$;
fin

Sélection réactive de l'interaction effectuée

Une fois que l'entité source a calculé sont potentiel d'interaction, elle doit déterminer quelle interaction elle va initier. Ce procédé dépend de la nature réactive, cognitive ou hybride du processus de sélection d'interaction de l'entité.

Dans le chapitre précédent, nous avons défini un modèle de sélection réactive d'interaction, fondé sur la notion de matrice d'interaction raffinée. Selon ce modèle, une priorité est associée à chaque élément d'assignation (et donc à chaque tuple). Il est toujours fait de sorte que le tuple sélectionné a la plus grande priorité parmi les tuples contenus dans le potentiel d'interaction de l'entité. Si plusieurs de ces tuples ont cette priorité, le tuple sélectionné est choisi parmi eux aléatoirement. L'algorithme 3 décrit ce procédé.

Algorithme 3 : Algorithme décrivant comment une entité utilisant un modèle de sélection réactive d'interaction détermine le tuple dont il initie l'interaction. Dans cet algorithme, on suppose disposer d'une fonction nommée *élément* permettant de connaître l'élément d'assignation d'un tuple.

sélection(x)
début
  $\mathcal{R} \Leftarrow$ potentiel d'interaction(x);
  % $\mathcal{O}$ contiendra l'ensemble des priorités des éléments d'assignation où e est une source
  $\mathcal{O} \Leftarrow \emptyset$;
  pour tous les $a \in ligne\Big(famille(x)\Big)$ **faire**
    si $\mathcal{M}_{raff}(a) \notin \mathcal{O}$ **alors**
      $\mathcal{O} \Leftarrow \mathcal{O} \cup \{\mathcal{M}_{raff}(a)\}$;
  élémentChoisi $\Leftarrow$ null;
  tant que *élémentChoisi* $==$ *null* ***et*** $\mathcal{O} \neq \emptyset$ **faire**
    % Récupération de la plus grande priorité contenue dans $\mathcal{O}$
    p $\Leftarrow$ max($\mathcal{O}$);
    $\mathcal{O} \Leftarrow \mathcal{O} \backslash \{p\}$;
    % Récupération de tous les tuples de priorité p
    $\mathcal{R}_p \Leftarrow \emptyset$;
    pour tous les $\mathcal{T} \in \mathcal{R}$ **faire**
      si $\mathcal{M}_{raff}\Big(\textit{élément}(\mathcal{T})\Big) == p$ **alors**
        $\mathcal{R}_p \Leftarrow \mathcal{R}_p \cup \{\mathcal{T}\}$;
    si $\mathcal{R}_p \neq \emptyset$ **alors**
      élémentChoisi $\Leftarrow$ élément au hasard de $\mathcal{R}_p$;
  retourner élémentChoisi;
fin

Dans des simulations où le voisinage des entités contient un nombre élevé d'entités, et où les entités peuvent initier une grande variété d'interactions, l'algorithme présenté précédemment, bien que correct, n'est pas satisfaisant. En effet, en supposant que le voisinage d'une entité contienne n_v voisins, qu'une entité ait la capacité d'initier n_d interactions dégénérées et n_i interactions individuelles, le calcul du potentiel d'interaction nécessite $n_d \times n_i + n_d$ tests d'éligibilité et au pire $n_d \times n_i + n_d$. Ce nombre important de calculs n'est pas inévitable dans le cas d'une sélection réactive. En effet, on sait que si un tuple de priorité p est sélectionné, alors il n'est même pas la peine d'effectuer le test d'éligibilité et de réalisabilité pour les tuples de priorité inférieure. On introduit donc la notion de potentiel d'interaction de priorité $p \in \mathbb{Z}$ pour remédier à ce problème.

Définition 32. ***Potentiel d'interaction de priorité*** p
Le **potentiel d'interaction de priorité** $p \in \mathbb{Z}$ d'une entité contient tous les tuples réalisables du potentiel d'interaction de cette entité, dont l'élément d'assignation a pour priorité p.

Cette optimisation aboutit à l'algorithme 4.

Initiation d'une interaction

Une fois un tuple réalisable sélectionné, une entité initie l'interaction lui correspondant. Cette étape consiste à exécuter les actions de l'interaction, en remplaçant chaque occurrence d'une primitive abstraite

Algorithme 4 : Algorithme optimisé décrivant comment une entité utilisant un modèle de sélection réactive d'interaction détermine le tuple dont il initie l'interaction. Dans cet algorithme, on suppose disposer d'une fonction nommée *priorité* permettant de connaître la priorité de l'élément d'assignation d'un tuple, à l'aide de la matrice d'interaction raffinée.

```
sélection(x)
début
    % O contiendra l'ensemble des priorités des éléments d'assignation où e est une source
    O ⇐ ∅;
    pour tous les a ∈ ligne(famille(x)) faire
        si M_raff(a) ∉ O alors
            O ⇐ O ∪ {M_raff(a)};
    élémentChoisi ⇐ null;
    tant que élémentChoisi == null et O ≠ ∅ faire
        % Récupération de la plus grande priorité contenue dans O
        p ⇐ max(O);
        O ⇐ O\{p};
        % Récupération de tous les tuples de priorité p
        R_p ⇐ potentiel d'interaction de priorité(x, p);
        si R_p ≠ ∅ alors
            élémentChoisi ⇐ élément au hasard de R_p;
    retourner élémentChoisi;
fin
```

par la spécification lui correspondant dans la famille d'entités associée, et chaque occurence d'une primitive de l'environnement par sa spécification.

4.3.2 Simulation en temps discret

L'algorithme général permettant d'effectuer une simulation dépend grandement du modèle du temps utilisé. Dans cette thèse, nous décrivons comment faire pour implémenter une simulation usant d'un modèle de temps discret. Les algorithmes présentés jusqu'à présent restent toutefois valides pour toute autre représentation du temps.

Le modèle de temps discret que nous utilisons modélise le temps comme un ensemble discret et totalement ordonné d'unités atomiques de temps, que nous appelons pas de temps. La simulation avance dans le temps en parcourant de manière séquentielle cet ensemble. Afin de garantir la cohérence de la simulation, les entités doivent être à jour avant de percevoir et pouvoir initier une quelconque interaction lors d'un pas de temps. Par conséquent, **un pas de temps commence par mettre à jour chaque entité de la simulation**, avant de donner la possibilité aux entités d'exécuter leur processus comportemental.

Le nombre de pas de temps que dure une simulation dépend d'un critère fixé lors de la préparation des expérimentations. La description de ce critère sort du cadre de cette thèse.

Mise à jour des entités

La mise à jour d'une entité consiste à initier toutes les interactions de mise à jour présentes dans la cellule de la matrice de mise à jour ordonnée associée à la famille de cette entité dont les préconditions et le déclencheur sont vérifiés, dans l'ordre défini par leur priorités. L'algorithme 5 décrit ainsi comment est mis à jour l'état d'une entité e.

Déclenchement du processus comportemental des entités

Puisqu'un pas de temps est une unité de temps indivisible, en théorie, toutes les entités de l'environnement doivent sélectionner l'interaction qu'elles souhaitent initier en parallèle. Cette simultanéité

Algorithme 5 : Algorithme décrivant comment a lieu la mise à jour de l'état d'une entité e.

```
miseAJour(e)
début
    % O contiendra l'ensemble des priorités des interactions de mise à jour de e
    O ⇐ ∅;
    pour tous les u ∈ U(famille(e)) faire
        si U_ord(u) ∉ O alors
            O ⇐ O ∪ {U_ord(u)};
    tant que O ≠ ∅ faire
        p ⇐ max(O);
        O ⇐ O\{p};
        % Exécution de toutes les interactions de mise à jour de e de priorité p
        pour tous les u ∈ U_ord^-1(famille(e), p) faire
            si I(u).preconditions(e) == Vrai et I(u).declencheur(e) == Vrai alors
                I(u).actions(e);
fin
```

introduit deux problèmes non triviaux à résoudre : la détection de conflits pouvant survenir lors de l'exécution de deux interactions modifiant de manière non compatible l'environnement, ou l'état d'une entité. Par exemple deux entités ne peuvent pas ramasser la même entité cible. Des théories telles que le modèle Influence/Réaction [FM96] et ses extensions [WH03, Mic07] offrent un cadre formel permettant de spécifier de tels problèmes.

Une telle représentation du temps nécessite d'une part un grand nombre de calculs et d'autre part de déterminer dans le modèle comment détecter et résoudre les conflits. Dans cette thèse, nous utilisons une représentation du temps presque équivalente, que l'on retrouve de manière récurrente dans les plateformes de simulations. Cette approche se fonde sur le constat que, dans la majorité des cas, lorsque plusieurs actions sont en conflit, l'action initiée est choisie aléatoirement. Une façon alternative permettant d'éviter de tels conflits consiste a faire agir chaque entité en une séquence prédéfinie. La résolution de conflits aléatoire est alors assurée si la séquence utilisée pour faire agir les entité est mélangée au début de chaque pas de temps.

L'algorithme 6 résume comment s'exécute une simulation en temps discret.

4.4 Synthèse du chapitre

Dans ce chapitre, nous montrons comment utiliser le modèle formel IODA afin d'établir une approche transversale de conception où un modèle est construit graduellement puis implémenté selon des principes pouvant être automatisés.

Conception graduelle d'un modèle La conception graduelle d'un modèle dans IODA est assurée par une méthodologie de conception reposant sur 13 étapes, résumées dans le tableau 4.2, où :

- un modèle est conçu à l'aide de **représentations graphiques** qui facilitent sa description ;
- la construction de modèles conséquents est facilitée par la **complétion automatique de certains éléments du modèle**. Par exemple, la description des conditions et des actions d'une interaction, et l'ajout de cette interaction dans la matrice d'interaction implique la génération automatique de primitives abstraites vides que les familles d'agents pouvant initier ou subir l'interaction doivent spécifier ;
- **chaque étape** de la méthodologie se **focalise sur un niveau d'abstraction** particulier du modèle ;
- la **conception progressive** d'un modèle est **guidée** en suivant les dépendances entre étapes de la méthodologie ;

Algorithme 6 : Algorithme général d'une simulation en temps discret. Dans cet algorithme, on note $\mathbb{E}_{labile}$ l'ensemble des entités labiles de la simulation.

```
effectuerSimulation()
début
    t ⇐ 0;
    tant que ¬ la simulation est terminée faire
        % Début d'un nouveau pas de temps
        t ⇐ t + 1;
        % Mise à jour de l'état des entités
        pour tous les e ∈ E_labile faire
            e.miseAJour();
        % Exécution du processus comportemental des entités
        Mélanger aléatoirement le contenu de E_active;
        pour tous les e ∈ E_active faire
            % Sélection du tuple réalisable dont l'interaction est effectuée
            T ⇐ sélection(e);
            si T ≠ null alors
                Exécuter l'interaction représentée par le tuple T;
fin
```

– les hypothèses de fonctionnement du phénomène sont introduites à la fin du processus de conception, lors de la spécification des primitives abstraites des agents. La conception est donc focalisée sur des descriptions plus objectives du phénomène.

Algorithmes de simulation Nous montrons qu'il est possible d'écrire des algorithmes de simulation reposant sur le modèle IODA afin de construire un simulateur. Ces algorithmes permettent de **conserver la structure du modèle lors de l'implémentation**, et ainsi constituer une **architecture modulaire facilement modifiable** favorisant les tests unitaires. Ils permettent de plus de construire des **bibliothèques d'interactions réutilisables**, facilitant ainsi les révisions du modèle.

Nous positionnons ensuite IODA vis à vis de quatre questions se posant systématiquement dans toute approche de conception transversale. Nous mettons ainsi en lumière certains autres avantages de IODA facilitant la conception de simulation.

Toute entité est-elle un agent ? De nombreux concepteurs en programmation multi-agents utilisent plusieurs notions pour définir les entités utilisées dans leurs simulations. Ces dernières sont concrétisées sous divers noms, allant de « agent » à « objet », en passant par « artefact », « patch », *etc.* Nous montrons que ces différentes **entités** peuvent être **avantageusement unifiées** avec IODA afin d'être traitées par **un seul et même algorithme** facilitant à la fois les processus de conception et d'implémentation. Dans la phase de conception, la question de l'appartenance familiale des entités ne se pose plus, simplifiant d'autant le processus de réflexion. Dans la phase d'implémentation, il n'est plus nécessaire d'avoir différentes structures de données et surtout différents algorithmes pour traiter chacune des familles d'entités du logiciel. Le logiciel devient alors **plus léger et plus maintenable**.

Une méthodologie de conception doit-elle être dogmatique ? Une méthodologie dogmatique n'est pas appropriée à la construction itérative de modèles, qui est pourtant le fondement de la simulation. En effet, la description du modèle à un niveau d'abstraction peut parfois mettre en lumière des lacunes, amenant à compléter les spécifications d'un niveau d'abstraction plus élevé. Par exemple, la description des conditions d'une interaction peut amener à identifier d'autres interactions. Un processus dogmatique ne permet pas un tel « retour en arrière » dans ses étapes.

L'approche IODA ne souffre pas de ce problème, puisqu'elle repose sur un unique modèle, et sur une **méthodologie dont les étapes peuvent être entremêlées aisément**. En effet, les seules contraintes

TABLE 4.2 – Méthodologie utilisée pour concevoir des modèles dans IODA, décomposée en 13 étapes. Chaque étape est caractérisée dans ce tableau synthétique par l'élément du modèle qu'elle édite, son objectif et les moyens graphiques utilisés pour éditer le modèle. Le modèle spécifié lors de chaque étape a une influence ou un effet sur le modèle édité lors d'autres étapes qui peuvent être automatisés. Par exemple, l'ajout d'une interaction lors de l'étape C implique l'ajout d'une interaction devant être spécifiée à l'étape D. La colonne « pré-requis » exprime cette relation, en spécifiant quelles étapes ont un effet ou une influence sur l'étape de la ligne. Elle contient soit le nom de ces étapes, des opérateurs $\vee$ signifiant que deux étapes peuvent séparément avoir un effet sur cette étape, et des opérateurs $\wedge$ signifiant que deux étapes n'ont un effet que conjointement sur cette étape. Par exemple, le pré requis B $\wedge$ C de l'étape H signifie que pour ajouter une interaction de mise à jour dans la matrice de mise jour, l'interaction doit avoir été ajoutée lors de l'étape B, et l'entité à laquelle l'interaction est attribuée doit avoir été ajoutée lors de l'étape C.

Étape	Modèle édité	Objectif de l'étape	Moyens graphiques	Pré-requis
A	Topologie l'environnement	Indiquer la signification de la distance, en lui donnant une unité	aucun	aucun
B	Identificateur des entités	Identifier le « nom » des entités impliquées dans la simulation	Construction d'une liste de noms	aucun
C	Identificateur des interactions	Identifier le « nom » des interactions apparaissant dans la simulation	Construction d'une liste de noms	aucun
D	Conditions et actions des interactions	Décrire les conditions et les actions des interactions, à l'aide de primitives abstraites identifiées à cette occasion	Description algorithmique	C
E	Matrice d'interaction brute	Construire la matrice d'interaction brute en plaçant des noms d'interaction dans les cellules de la matrice	Faire glisser des éléments de la liste des interactions dans les cellules d'une matrice	A $\wedge$ B $\wedge$ C
F	Halo des entités	Décrire comment le halo des entités détermine si une entité est perçue ou non	Description algorithmique	A $\wedge$ B
G	Matrice d'interaction raffinée	Construire la matrice d'interaction raffinée, en attribuant une priorité à chaque élément de la matrice d'interaction brute	Associer un entier à chaque élément présent dans la matrice d'interaction brute	E
H	Matrice de mise à jour	Construire la matrice de mise à jour en plaçant des noms d'interaction dans les cellules de la matrice	Faire glisser des éléments de la liste des interactions dans les cellules d'une matrice	B $\wedge$ C
I	Primitives de l'environnement	Spécifier l'effet de chaque primitive fournie par l'envionnement	Description algorithmique	A $\vee$ (A $\wedge$ B) $\vee$ D
J	Matrice de mise à jour ordonnée	Construire la matrice de mise à jour ordonnée, en attribuant une priorité à chaque interaction de la matrice de mise à jour	Associer un entier à chaque élément présent dans la matrice de mise à jour	H
K	Attribut des entités	Identifier les attributs d'une entité	aucune	L
L	Primitives abstraites des entités	Spécifier l'effet de chaque fournie primitive par la famille d'agents	description algorithmique	F $\vee$ (D $\wedge$ G) $\vee$ (D $\wedge$ J)
M	Attributs de l'environnement	Identifier les attributs de l'environnement	aucune	M

sur notre méthodologie sont d'ordre logique. Par exemple, une primitive abstraite n'est spécifiée par une entité que si cette entité est la source ou la cible d'une interaction manipulant la primitive dans ses conditions ou actions.

Faut-il prendre en compte les performances lors de la modélisation ? La conception d'une simulation performante est un problème lié à l'implémentation. Il est paradoxal de le prendre en compte lors de la modélisation, puisque les experts du domaine doivent être impliqués au maximum dans la construction du modèle.

L'utilisation d'un modèle décrivant un phénomène à différents niveaux d'abstraction fournit un compromis à ce problème. En effet, les descriptions effectuées à un niveau fin d'abstraction sont par définition les plus proches de l'implémentation. Il est donc moins gênant que la question des performances y soit considérée.

Faut-il tout décrire au sein d'une règle ? Déterminer le **montant d'informations** à décrire **dans les interactions**, et la part à **déléguer dans** la spécification des **primitives** des entités est un problème **complexe**, auquel seul le concepteur de la simulation peut répondre.

Ce point n'est toutefois **pas une limitation intrinsèque à l'approche IODA**, puisqu'une question similaire se pose dans toute approche réifiant les actions entreprises par des entités : faut-il décrire un comportement complexe par une seule action complexe, ou par un ensemble d'actions simples ?

Preuve des avantages attendus de IODA Afin de confirmer les avantages théoriques de IODA, nous décrivons dans le chapitre suivant une implémentation fidèle de notre modèle formel, de nos algorithmes, et de notre méthodologie en une plateforme de simulation appelée JEDI, ainsi qu'un environnement de développement intégré nommé JEDI-BUILDER.

Chapitre 5

JEDI : une plateforme générique et paramétrable nécessaire

Plan du chapitre :

Le modèle formel et les algorithmes présentés dans le chapitre précédent sont avancés comme suffisants pour construire des simulations en conservant le morphisme entre modèle et implémentation. Afin d'en confirmer la faisabilité et ainsi montrer que les avantages avancés de l'approche IODA *ne sont pas purement théoriques, nous décrivons dans ce chapitre :*

- *Une implémentation fidèle du modèle formel et des algorithmes en une plateforme de simulation paramétrable appelée* JEDI *(Java Environment for the Design of agent Interactions) ;*
- *Une implémentation de la méthodologie de conception* IODA *appelée* JEDI-BUILDER *permettant de construire graduellement un modèle, et de l'implémenter sur la plateforme* JEDI.

En plus de confirmer la faisabilité de l'approche IODA*, ces implémentations ont aussi servi de support à l'exploration des problématiques de simulation selon la perspective d'une approche centrée sur les interactions, que nous menons dans la partie III.*

5.1 Réification de IODA dans la plateforme JEDI

Le morphisme entre modèle et implémentation est garanti dans JEDI par l'implémentation quasi-systématique d'un élément du modèle formel de IODA par une classe, une classe abstraite, ou une interface lui étant dédiée. Nous ne décrivons ici que la part émergée de l'implémentation, *i.e.* la part de l'implémentation manipulée par les concepteurs afin d'implémenter les spécifications issues de la méthodologie IODA.

Choix effectués dans JEDI L'approche IODA est générique, et peut en particulier être utilisée pour implémenter des environnements, ou des modèles de sélection d'interaction très différents. Dans JEDI, nous avons fait le choix de nous restreindre aux environnements, modèles de sélection d'interaction que l'on retrouve le plus souvent en simulation : les environnements euclidiens en deux dimensions, et les modèles de sélection d'interaction réactifs. Nous avons de plus fait le choix d'effectuer l'implémentation en JAVA pour trois raisons :

- C'est un langage orienté-objet, ce qui facilite la construction d'une plateforme modulaire où presque chaque concept de IODA est représenté par une classe.
- C'est l'un des langages de programmation les plus utilisés actuellement. JEDI est ainsi ouvert à un public aussi large que possible.
- C'est un langage reposant sur un typage fort. Il permet ainsi de réifier la notion de signature sous la forme d'interface et donc de vérifier plus aisément si un agent se conforme à la signature de la

source ou la cible dans interaction.
Les langages permettant d'implémenter l'approche IODA ne se limitent toutefois pas au JAVA, ou plus généralement aux langages orientés-objet. En effet, des prototypes ont pu être implémentés dans le langage Netlogo [WC99] et dans le langage Prolog [Col90].

L'environnement dans JEDI Dans JEDI, nous avons fait le choix de restreindre les simulations aux environnements les plus couramment rencontrés en simulation informatique : les environnements euclidiens en deux dimensions, pouvant être non-toriques ou toriques selon l'axe des abscisses, des ordonnées, ou les deux. Ce choix a plusieurs implications sur l'implémentation que nous fournissons :

- Nous supposons l'ensemble des primitives de l'environnement fixe et ne pouvant pas être modifié.
- La signature de l'environnement dans les interactions ou le halo des entités ne peut contenir que des sous-ensembles de ces primitives. Dans JEDI, ces signatures n'ont pas à être définies : elles sont définies implicitement, en leur ajoutant l'intégralité des primitives définies pour les environnement euclidiens.
- Les entités ont une position dans l'environnement, et possèdent donc deux attributs *abscisse* et *ordonnée* ainsi que des primitives permettant d'y accéder.
- Les entités sont soit ponctuelles, soit occupent une surface dans la simulation. Cette surface est approximée dans JEDI par une « bounding box », *i.e.* un rectangle dont les côtés sont parallèles à l'axe des abscisses et des ordonnées.
- Le halo des entités est exprimé à l'aide d'une surface de perception centrée sur la position de l'entité, appelée *surface de perception référente*. Le voisinage d'une entité sont les entités dont la surface intersecte la surface de perception.

Le détail des principes de calcul de la distance, du calcul du voisinage, et plus généralement les primitives fournies par l'environnement, et leur implémentation sont décrites dans l'annexe A page 259. Il est à noter que l'extension de JEDI à d'autres formes d'environnement ne nécessite pas de modification profonde des diagrammes UML que nous décrivons ici.

Le modèle de sélection d'interaction dans JEDI Nous restreignons dans JEDI les modèles de sélection d'interaction au modèle réactif basé sur une matrice d'interaction raffinée, que nous avons présenté dans le chapitre précédent. Ce modèle fournit, à notre sens, un bon compromis entre la complexité du modèle utilisé, et la complexité des comportements qu'il est possible de modéliser simplement.

En effet, bien le modèle utilisé soit réactif, il est possible de modéliser des agents cognitifs. Par exemple, les primitives de perception et d'actions peuvent se baser sur un modèle fondé sur les croyances et les connaissances de l'entité, et utiliser ainsi des moteurs d'inférences de connaissance que l'on retrouve chez certains agents cognitifs, ou encore les interactions peuvent produire et consommer des buts poursuivis par une entité, et permettre ainsi de modéliser de la planification réactive.

Vue d'ensemble sur le modèle IODA dans JEDI Les éléments logiciels que nous présentons sont résumés sous la forme de diagrammes UML dans les figures 5.1 et 5.2. La plateforme JEDI est actuellement composée de 95 classes et interfaces, totalisant 14820 lignes de code physiques (commentaires et mise en page inclus), équivalentes à 5541 lignes de code logique [16].

Remarque. JEDI est une plateforme écrite en anglais. Tous les éléments du modèle décrits jusqu'à présent y sont donc traduits.

5.1.1 Interactions

Le modèle d'une interaction est un 8-uplet $\mathcal{I}$ tel que :

$$\mathcal{I} = \langle id, noms, cardinalite, signature, sign_{env}, preconditions, declencheur, actions \rangle$$

L'implémentation de ce modèle dépend de la façon d'implémenter la signature des entités dans une interaction et la signature de l'environnement dans une interaction.

16. comptabilisées selon la métrique SLOC de la commande unix `sloccount`

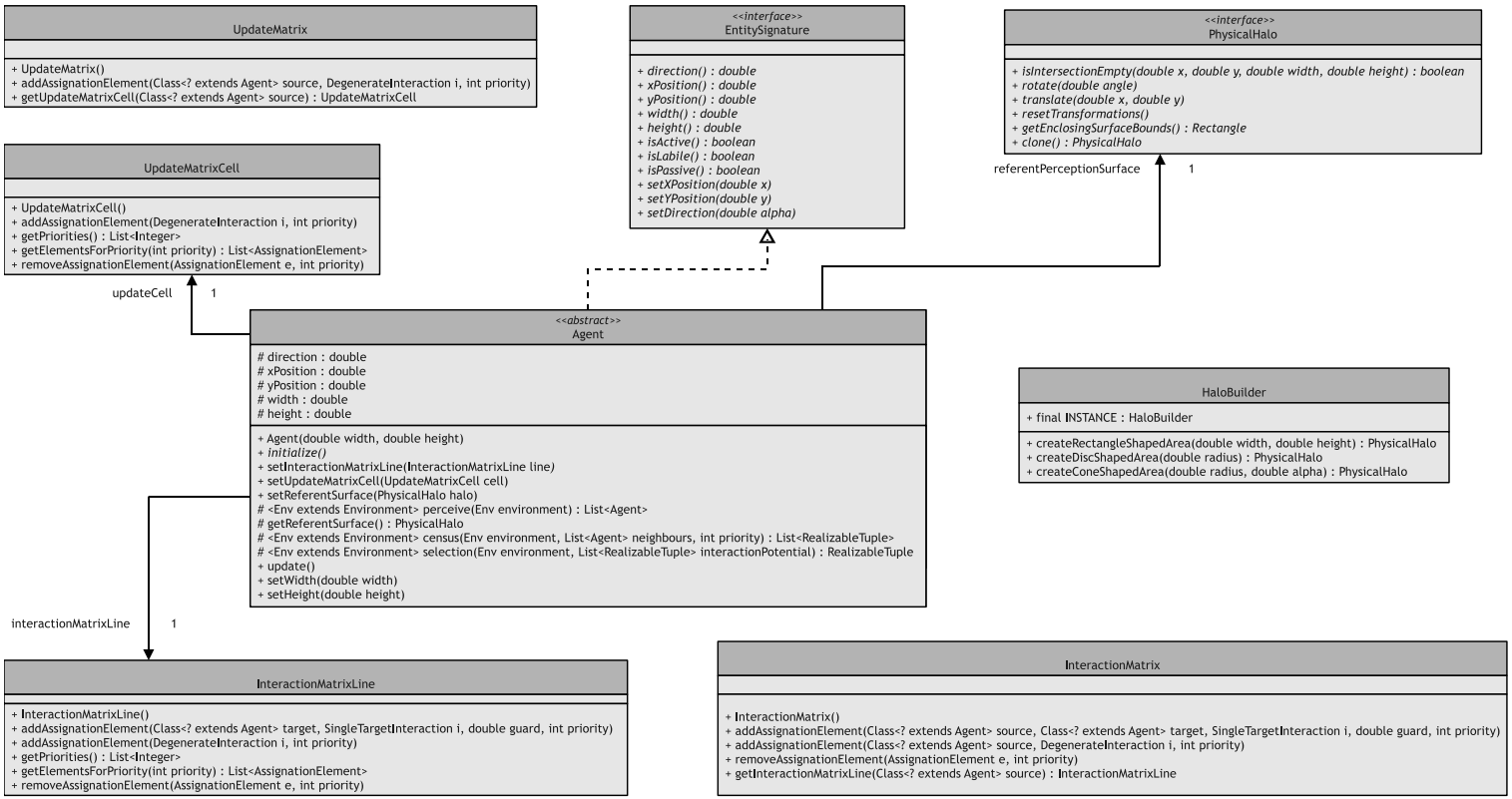


FIGURE 5.1 – Première partie du diagramme UML décrivant les classes et interfaces de la plateforme JEDI directement manipulées pour implémenter un modèle décrit avec la méthodologie IODA.

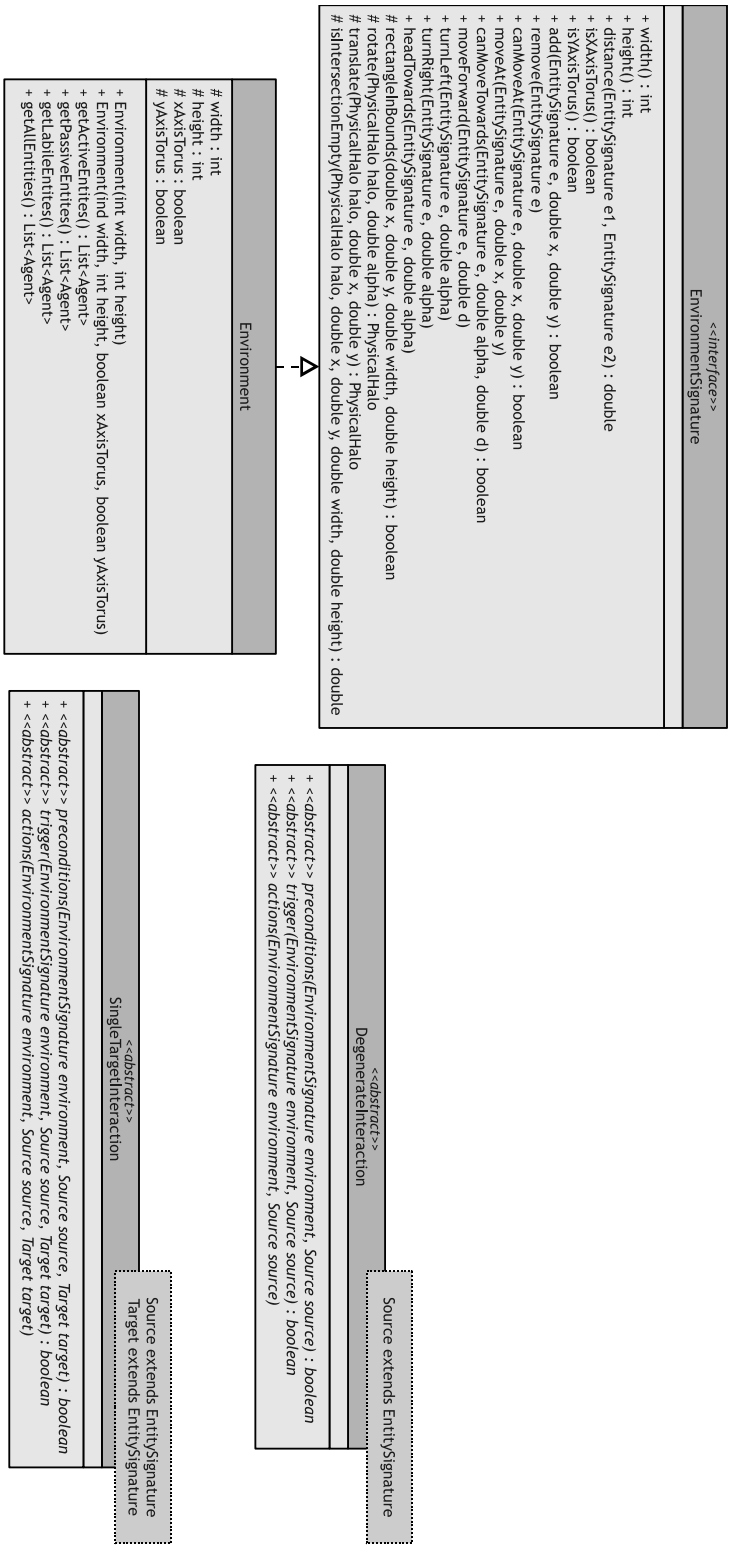


FIGURE 5.2 – Seconde partie du diagramme UML décrivant les classes et interfaces de la plateforme JEDI directement manipulées pour implémenter un modèle décrit avec la méthodologie IODA.

Dans le modèle IODA, la signature d'une entité dans une interaction est représentée par un couple S tel que :

$$S = \langle id(S), primitives(S) \rangle$$

Dans JEDI, ce couple est implémenté sous la forme d'une interface étendant l'interface nommée *EntitySignature*. La valeur $id(S)$ correspond au nom de l'interface créée, et l'ensemble $primitives(S)$ des primitives abstraites de cette signature sont représentées par des méthodes dans l'interface. Ces interfaces correspondent aux éléments contenus dans l'ensemble $signatures(\mathcal{I})$ du 7-uplet définissant une interaction. Le diagramme UML de cette spécification apparaît sur la figure 5.1.

Remarque. Nous nous plaçons dans le cadre des environnements euclidiens en deux dimensions décrits précédemment. Par conséquent, toutes les entités d'une simulation doivent implémenter les primitives abstraites provenant de la signature des entités dans l'environnement, dont un résumé est fourni dans le tableau A.5 page 270 de l'annexe A. Pour des raisons exposées par la suite, ces primitives sont intégrées à l'interface *EntitySignature*.

Le modèle de la signature de l'environnement dans une interaction est un élément S tel que :

$$S = \langle (p_i)_{i \in \mathbb{N}} \rangle$$

Sa structure est très similaire au modèle de la signature d'une entité dans une interaction. Par conséquent, son implémentation dans JEDI use de moyens similaires, sous la forme d'une interface nommée *EnvironmentSignature*, qui correspond à l'élément $sign_{env}$ du 8-uplet définissant une interaction. Les primitives de l'environnement de cette signature sont représentées par des méthodes dans l'interface.

Le diagramme UML de cette spécification apparaît sur la figure 5.2.

Remarque. Nous nous plaçons dans le cadre des environnements euclidiens en deux dimensions décrits précédemment. Par conséquent, toutes les entités d'une simulation doivent implémenter les primitives abstraites provenant de la signature des entités dans l'environnement, dont un résumé est fourni dans le tableau A.5 page 270 de l'annexe A. Pour des raisons exposées par la suite, ces primitives sont intégrées à l'interface *EntitySignature*.

Nous nous plaçons dans le cadre des environnements euclidiens en deux dimensions décrits précédemment. Par conséquent, l'environnement fournit un ensemble important de primitives de l'environnement, dont un résumé est fourni dans le tableau A.5 page 270 de l'annexe A. Pour des raisons exposées par la suite, ces primitives sont intégrées à l'interface *EnvironmentSignature*.

L'approche IODA est présentée dans ce chapitre uniquement dans le cadre d'interactions de cardinalité $(1,1)$ (les interactions individuelles) et $(1,0)$ (les interactions dégénérées). Dans JEDI, nous identifions la cardinalité $card(\mathcal{I})$ de l'interaction par la classe utilisée pour implémenter son modèle.

Les interactions individuelles sont représentées par une classe étendant la classe parente à toutes les interactions individuelles, nommée *SingleTargetInteraction*. L'identifiant *id* de cette interaction est alors utilisé pour nommer la classe ainsi créée. Cette classe est paramétrée par deux types génériques appelés *Source* et *Target*, qui correspondent respectivement à la signature de l'entité source, et à la signature de l'entité cible dans l'interaction. Puisque l'on sait que l'élément *signature* du modèle de l'interaction contient les deux signatures $sign_{source}(\mathcal{I})$ et $sign_{cible}(\mathcal{I})$, on a :

- $Source = id\Big(sign_{source}(\mathcal{I})\Big)$, où $id\Big(sign_{source}(\mathcal{I})\Big)$ est le nom de l'interface représentant la signature de l'entité source dans l'interaction ;
- $Cible = id\Big(sign_{cible}(\mathcal{I})\Big)$, où $id\Big(sign_{cible}(\mathcal{I})\Big)$ est le nom de l'interface représentant la signature de l'entité cible dans l'interaction.

Les interactions dégénérées sont représentées de manière similaire, par une classe étendant la classe parente à toutes les interactions dégénérées, nommée *DegenerateInteraction*. L'identifiant $id(\mathcal{I})$ de cette interaction est alors utilisé pour nommer la classe ainsi créée. Cette classe est paramétrée par un type générique appelé *Source*, dont la définition a déjà été fournie dans le paragraphe précédent.

Les interactions ne manipulent pas directement les entités et l'environnement, mais uniquement leurs primitives. Par conséquent, le type des paramètres des préconditions, du déclencheur et des actions d'une

interaction sont la signature des entités impliquées dans l'interaction, ainsi que la signature de l'environnement dans l'interaction, *i.e.* les types génériques *Source* et *Target*, et l'interface *EnvironmentSignature*. De par la forme préconisée pour spécifier les préconditions, le déclencheur, et les actions dans le modèle d'une interaction, l'implémentation des méthodes éponymes dans la classe *SingleTargetInteraction* est directe et non ambiguë. Le diagramme UML de la figure 5.2 résume la structure des interactions dans JEDI.

Remarque. Comme les préconditions, le déclencheur et les actions de l'interaction sont susceptibles de manipuler les primitives abstraites et les primitives de issues de la description de l'environnement euclidien, nous avons fait le choix d'intégrer toutes les primitives liées à ce type d'environnement dans les interfaces *EntitySignature* et *EnvironmentSignature*.

5.1.2 Matrice d'interaction

Dans IODA, le modèle d'une matrice d'interaction brute est une fonction $\mathcal{M}(S,T)$ telle que :

$$\begin{array}{rccl} \mathcal{M} \ : & \mathbb{F} \times \big(\mathbb{F} \cup \{\emptyset\}\big) & \rightarrow & \mathcal{P}\Big(\big(\mathbb{I}_{(1,0)} \times \mathbb{F}\big) \cup \big(\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}\big)\Big) \\ & (\mathcal{S},\mathcal{T}) & \rightarrow & a_{\mathcal{S}/\mathcal{T}} \end{array}$$

Dans cette notation, $\mathbb{F}$ est l'ensemble des familles d'entités de la simulation, $\mathbb{I}_{(1,0)}$ est l'ensemble des interactions dégénérées de la simulation, et $\mathbb{I}_{(1,1)}$ est l'ensemble des interactions individuelles de la simulation. Dans le cadre de cette thèse, le modèle de sélection d'interaction d'une entité consiste à construire une matrice d'interaction raffinée, qui prend la forme d'une fonction $\mathcal{M}_{raff}(a)$ telle que :

$$\begin{array}{rccl} \mathcal{M}_{raff} \ : & \Big(\mathbb{I}_{(1,0)} \times \mathbb{F}\Big) \cup \Big(\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}\Big) & \rightarrow & \mathbb{Z} \\ & (a) & \rightarrow & \mathcal{M}_{raff}(a) \end{array}$$

Dans cette notation, a est un élément d'assignation, contenu dans l'une des assignations $\mathcal{M}(S,T)$ de la matrice d'interaction brute.

Dans JEDI, une famille d'entités est représentée par une classe étendant la classe abstraite parente de toute famille d'entités, nommée *Agent*. Par conséquent, les paramètres de la fonction $\mathcal{M}$ sont des instances de la classe *Class< ? extends Agent>*.

Dans JEDI, nous faisons le choix d'implémenter la matrice d'interaction brute et la matrice d'interaction raffinée dans une seule et même classe, nommée *InteractionMatrix*. Cette classe utilise des sous classes telles que *Assignation*, *AssignationElement* ou *InteractionMatrixLine* comme structure de données permettant de mémoriser les assignations $\mathcal{M}(S,T)$, ainsi que leur priorité définie par $\mathcal{M}(a)$. La description de ces éléments sort du cadre de ce chapitre, qui offre une présentation générale de JEDI. Deux méthodes sont définies pour ajouter les différents éléments d'assignation présents dans la matrice d'interaction. La première s'intitule *addAssignationElement(Class< ? extends Agent>, Class< ? extends Agent>, SingleTargetInteraction,double,int)*, et permet d'ajouter un élément d'assignation individuel à la matrice d'interaction brute, tout en lui associant une priorité dans la matrice d'interaction raffinée. La seconde méthode s'intitule *addAssignationElement(Class< ? extends Agent>, DegenerateInteraction,int)*, et permet d'ajouter un élément d'assignation dégénéré à la matrice d'interaction brute, tout en lui associant une priorité dans la matrice d'interaction raffinée. Une dernière méthode intitulée *getInteractionMatrixLine(Class< ? extends Agent>)* permet d'avoir accès à la ligne de la matrice d'interaction pour une famille d'entités source particulière. Le diagramme UML de cette classe est fourni sur la figure 5.1.

Une ligne de la matrice d'interaction est implémentée à l'aide de la classe *InteractionMatrixLine*. Cette classe est instanciée et initialisée automatiquement par un appel à la méthode *getInteractionMatrixLine(Class< ? extends Agent>)* de la classe représentant la matrice d'interaction. Cette ligne de la matrice d'interaction fournit en particulier deux méthodes nécessaires à l'implémentation du moteur de simulation. La première méthode, intitulée *getPriorities()*, est utilisée lors de la sélection d'interaction, et permet de connaître toutes les priorités présentes dans cette ligne de la matrice. La seconde méthode est intitulée *getElementsForPriotity(int)*, et permet de récupérer l'ensemble des éléments d'assignation ayant une priorité particulière. Cette méthode est utilisée pour calculer le potentiel d'interaction d'une priorité particulière. Le diagramme UML de cette classe est fourni sur la figure 5.1.

5.1.3 Matrice de mise à jour

Dans IODA, le modèle d'une matrice de mise à jour est une fonction $\mathcal{U}(S)$ telle que :

$$\mathcal{U} : \begin{array}{ccc} \mathbb{F} & \rightarrow & \mathcal{P}(\mathbb{I}_{(1,0)} \times \mathbb{F}) \\ \mathcal{S} & \rightarrow & (u^i_{\mathcal{S}}, \mathcal{S})_{i \in \mathbb{N}} \end{array}$$

Dans cette notation, $\mathbb{F}$ est l'ensemble des familles d'entités de la simulation, et $\mathbb{I}_{(1,0)}$ est l'ensemble des interactions dégénérées de la simulation. Afin d'implémenter la mise à jour des entités, le modèle d'une matrice de mise à jour ordonnée $\mathcal{U}_{ord}(u)$ a aussi été introduit :

$$\mathcal{U}_{ord} : \begin{array}{ccc} \mathbb{I}_{(1,0)} \times \mathbb{F} & \rightarrow & \mathbb{Z} \\ (u) & \rightarrow & \left\{ \begin{array}{ll} \mathcal{U}_{ord}(u) & \text{si } u \in \mathcal{U}(\mathcal{S}) \\ \text{non défini} & \text{sinon} \end{array} \right. \end{array}$$

Dans cette notation, u est un élément d'assignation dégénéré, contenu dans l'une des assignations dégénérée $\mathcal{U}(S)$ de la matrice de mise à jour.

Dans JEDI, nous faisons le choix d'implémenter la matrice de mise à jour et la matrice de mise à jour ordonnée dans une seule et même classe, nommée *UpdateMatrix*. Cette classe utilise des sous classes telles que *DegenerateAssignation*, *DegenerateAssignationElement* ou *UpdateMatrixCell* comme structure de données permettant de mémoriser les assignations dégénérées $\mathcal{U}(S)$, ainsi que leur priorité définie par $\mathcal{U}_{ord}(u)$. La description de ces éléments sort du cadre de ce chapitre. Une méthode intitulée *addAssignationElement(Class< ? extends Agent>, DegenerateInteraction,int)* est définie afin d'ajouter les différents éléments d'assignation dégénérés présents dans la matrice de mise à jour. Elle permet d'ajouter un élément d'assignation dégénéré à la matrice de mise à jour, tout en lui associant une priorité dans la matrice de mise à jour ordonnée. Une autre méthode intitulée *getUpdateMatrixCell(Class< ? extends Agent>)* permet d'avoir accès à la cellule de la matrice de mise à jour pour une famille d'entités source particulière. Cette méthode est utilisée lors de la spécification d'une famille d'entités. Le diagramme UML de cette classe est fourni sur la figure 5.1.

Une cellule de la matrice de mise à jour est implémentée à l'aide de la classe *UpdateMatrixCell*. Cette classe est instanciée et initialisée automatiquement par un appel à la méthode *getUpdateMatrixCell(Class< ? extends Agent>)* de la classe représentant la matrice de mise à jour. Cette cellule de la matrice de mise à jour fournit en particulier deux méthodes nécessaires à l'implémentation du moteur de simulation. La première méthode est intitulée *getPriorities()*, et permet de connaître toutes les priorités présentes dans cette cellule de la matrice. Cette méthode est utilisée lors de la mise à jour de l'état d'une entité. La seconde méthode est intitulée *getElementsForPriotity(int)*, et permet de récupérer l'ensemble des éléments d'assignation ayant une priorité particulière. Cette méthode joue le rôle de $\mathcal{U}^{-1}_{ord}$ dans l'algorithme de mise à jour d'une entité. Le diagramme UML de cette classe est fourni sur la figure 5.1.

5.1.4 Environnement

Dans IODA, le modèle de l'environnement est un quadruplet ε tel que :

$$\varepsilon = \langle id, primitives_{env}, sign_{entites}, attributs, valeur_{attr} \rangle$$

Dans JEDI, ce modèle est implémenté sous la forme de la classe *Environment*, qui étend l'interface *EnvironmentSignature*. Les primitives de l'environnement $primitives_{env}$ y sont implémentées par des méthodes, et les attributs *attributs* du modèle IODA sont implémentés par des attributs dans la classe ainsi créée. Puisque l'environnement est une structure figée dans JEDI (*i.e.* on ne peut y ajouter de nouvelles primitives de l'environnement), la signature de l'environnement et la signature des entités dans les diverses primitives de l'environnement sont égales respectivement à *EnvironmentSignature* et *EntitySignature*.

La classe *Environment* fournit une implémentation de toutes les primitives de l'environnement décrites dans l'annexe A. Elle offre de plus une implémentation à diverses méthodes nécessaires au fonctionnement du moteur de simulation. Parmi ces méthodes figurent *getActiveEntities()*, *getPassiveEntities()*, *getLabileEntities()* ou *getAllEntities()*, qui permettent d'avoir accès en lecture et en écriture à l'ensemble

des entités actives de la simulation (*i.e.* $\mathbb{E}_{active}$), à l'ensemble des entités passives de la simulation (*i.e.* $\mathbb{E}_{passive}$), à l'ensemble des entités labiles de la simulation (*i.e.* $\mathbb{E}_{labile}$), et enfin à l'ensemble des entités de la simulation (*i.e.* $\mathbb{E}$). Le diagramme UML de cette classe apparaît sur la figure 5.2.

5.1.5 Entités

Dans IODA, le modèle d'une famille d'entités est un 8-uplet $\mathcal{F}$ tel que :

$$\mathcal{F} = \langle id(\mathcal{F}), attributs(\mathcal{F}), primitives(\mathcal{F}), \mathcal{H}(\mathcal{F}), \mathcal{U}_{ord}(\mathcal{F}), ligne(\mathcal{F}), colonne(\mathcal{F}), selection(\mathcal{F})\rangle$$

Dans JEDI, ce modèle est implémenté sous la forme d'une classe étendant à la fois la classe abstraite *Agent*, et les interfaces représentant la signature de cette famille d'entités dans les diverses interactions auxquelles elle peut participer. La valeur $id(\mathcal{F})$ correspond au nom de la classe ainsi créée. L'ensemble $primitives(\mathcal{F})$ des primitives figurant dans les signatures sont implémentées sous la forme de méthodes, et l'ensemble des attributs $attributs(\mathcal{F})$ du modèle de la famille d'entités est implémenté sous la forme d'attributs dans cette classe. La ligne $ligne(\mathcal{F})$ dans la matrice d'interaction associée à cette famille d'entités est implémentée sous la forme d'un attribut nommé *interactionMatrixLine*, dont la classe est *InteractionMatrixLine*. La cellule $\mathcal{U}_{ord}(\mathcal{F})$ de la matrice de mise à jour ordonnée associée à cette entité est implémentée sous la forme d'un attribut nommé *updateMatrixCell*, dont la classe est *UpdateCellMatrix*. Le modèle de sélection d'interaction d'une entité est défini dans son attribut *interactionMatrixLine*.

Pour des raisons d'optimisation de calculs, la forme générale du halo $\mathcal{H}(\mathcal{F})$ d'une famille d'entités, où une fonction booléenne permet de vérifier l'appartenance d'une entité au voisinage, n'apparaît pas dans JEDI. À la place,une méthode intitulée *perceive(Environment)* est utilisée comme alternative. Elle est l'équivalent de la fonction $\mathcal{V}$ du modèle, qui calcule le voisinage d'une entité. De plus, nous fournissons une implémentation par défaut de cette méthode, où une entité perçoit les autres entités présentes dans une portion de l'environnement, identifiée à l'aide d'une surface. Ce type de halo implique la spécification d'une unique primitive abstraite appelée *referentPerceptionSurface()*, qui permet de connaître la surface de perception référente de l'entité (*i.e.* la surface de l'environnement dans laquelle les autres agents sont considérés comme perçus). Dans JEDI, nous considérons que cette méthode est un accesseur vers un attribut intitulé *referentPerceptionSurface*, dont la classe, intitulée *PhysicalHalo*, représente une surface de l'environnement. La valeur de cet attribut est initialisée par la méthode *setReferentPerceptionSurface(PhysicalHalo)*, dont le paramètre est obtenu à l'aide d'une fabrique abstraite, définie par la classe *HaloBuilder*. Cette classe définit trois types de surface de perception, construites à l'aide des méthodes :

- la méthode *createRectangleShapedArea(double, double)* permet de créer une surface de perception rectangulaire ;
- la méthode *createDiscShapedArea(double)* permet de créer une surface de perception prenant la forme d'un disque. Le voisinage est alors composé de toutes les entités située à une distance inférieure ou égale au rayon de ce disque ;
- la méthode *createConeShapedArea(double, double)* permet de créer une surface de perception en forme de cône.

Les surfaces de perception référentes n'y sont toutefois pas restreintes, et peuvent être complétées par des classes étendant l'interface *EntitySignature*.

La classe *Agent* définit aussi d'autres méthodes telles que census, selection ou upate, qui implémentent les différents algorithmes décrits dans le chapitre précédent. Nous ne décrivons pas ces méthodes dans ce chapitre, puisqu'elles ne constituent qu'une implémentation directe en JAVA de ces algorithmes. Le diagramme UML de cette classe apparaît sur la figure 5.1.

5.2 Un moteur générique et paramétrable nécessaire

Puisqu'une plateforme de simulation est un outil permettant de faciliter la conception de simulations, il apparaît normal d'y demander la spécification *a minima* d'un simulateur, et donc de réutiliser un maximum de concepts déjà implémentés. Toutefois, le simulateur ne doit pas devenir une boîte noire pour autant. En effet, dans une simulation explicative, le procédé suivi pour obtenir les résultats est tout aussi important, voire plus important que les résultats mêmes, contrairement aux autres types de

simulation qui ont été mentionnés dans la section 1.1.2 du chapitre 1, où seul le résultat obtenu compte, et pas le procédé (et les calculs) pour y parvenir. Dans ce cadre, les algorithmes utilisés pour spécifier une simulation particulière ne sont pas forcément adaptés à d'autres simulations. Nous illustrons ce point pour le modèle de tri collectif, et le modèle SugarScape.

Exemple. Tri « collectif » et SugarScape

Dans un modèle de « tri collectif »[17]*, par exemple dans le modèle décrit par Resnick [Res97], le nombre d'entités trieuses n'a pas d'influence sur l'apparition de tas [GRHT07]. Par conséquent, l'ordre dans lequel le processus comportemental des entités est déclenché n'a aucune influence sur les résultats obtenus.*

Au contraire, dans la simulation SugarScape *développée par Epstein et Axtell [EA96], l'ordre dans lequel le processus comportemental des entités est déclenché influe fortement sur la nature des résultats expérimentaux. En effet, l'objectif de cette simulation est de mesurer l'évolution d'une population d'entités se nourrissant d'une ressource disposée dans l'environnement sous la forme d'un tas. Si l'ordre de déclenchement du comportement des entités est identique pour chaque pas de temps, les entités les plus vieilles auront la priorité sur la consommation de ressources, ce qui a pour conséquence d'augmenter le taux de mortalité des entités plus jeunes. La pyramide des ages, qui constitue l'un des résultats étudiés de la simulation, s'en trouve modifiée.*

Par conséquent, le moteur de simulation utilisé pour le tri collectif n'est pas adapté pour modéliser SugarScape *[Mic04].*

La solution la plus simple à ce problème consiste à utiliser le moteur de simulation le plus complexe. Par exemple, dans les deux modèles mentionnés ci-dessus, un simulateur qui déclenche le comportement des entités dans un ordre aléatoire permet d'éviter les biais dans le modèle *SugarScape*, et permet aussi d'exécuter correctement une simulation de tri collectif. Toutefois, cette solution est obtenue au prix d'une complexification superflue du simulateur utilisé pour le tri collectif. Elle a pour conséquence la réduction du nombre d'expériences pouvant être effectuées en un temps donné, et donc une réduction de la viabilité des résultats expérimentaux obtenus. Elle fait de plus défaut au principe de parcimonie.

Ainsi, une plateforme supportant la conception de simulateurs doit prendre en compte une problématique à notre sens fondamentale : le contrôle fin du moteur de simulation, afin d'adapter la complexité du simulateur au problème simulé. Dans Jedi, nous offrons la possibilité de paramétrer finement trois aspects du moteur de simulation, via la classe singleton appelée *SimulationProperties* : le générateur de nombres aléatoires utilisé, l'ordonnanceur utilisé pour gérer l'activité des entités lors d'un pas de temps, ainsi que la façon de déterminer le sous-ensemble des entités à mettre à jour au début d'un pas de temps. Le diagramme UML résumant la structure logicielle mise en place pour gérer ces paramètres est résumée sur la figure 5.3.

5.2.1 Reproductibilité des résultats

La possibilité de reproduire exactement une simulation est l'un des fondements de la vérification d'une implémentation. En effet, la résolution d'erreurs dans un simulateur passe par la découverte d'une expérience particulière lors de laquelle des erreurs surviennent, puis par l'analyse de cette expérience afin d'identifier ce qui a participé à leur apparition. Cette analyse n'est possible que si l'expérience d'intérêt peut être exécutée à nouveau pas à pas, et donc être reproduite dans ses moindres détails. Ce déterminisme ne peut être atteint que par le contrôle des éléments régissant le non-déterminisme de la simulation dans l'implémentation : les générateurs de nombres aléatoires.

La génération d'un nombre aléatoire se fait dans Jedi via la méthode statique intitulée *random()* de la classe *SimulationProperties*. Cette méthode sans paramètres retourne un nombre à virgule flottante aléatoire (plus précisément, un double), situé dans l'intervalle $[0; 1[$. Puisque les algorithmes de simulation présentés dans le chapitre précédent utilisent par moment un générateur de nombre aléatoires, la reproductibilité d'une simulation ne peut être garantie dans Jedi que si ce générateur de nombre aléatoires est le seul à être utilisé.

17. Ce type de modèle est usuellement appelé « tri collectif », ce qui explique pourquoi nous utilisons cette terminologie ici. Il a toutefois été montré formellement par Gaubert *et al.*[GRHT07] que ce tri émerge aussi en la présence d'une unique entité trieuse. Le tri n'est donc pas « collectif ».

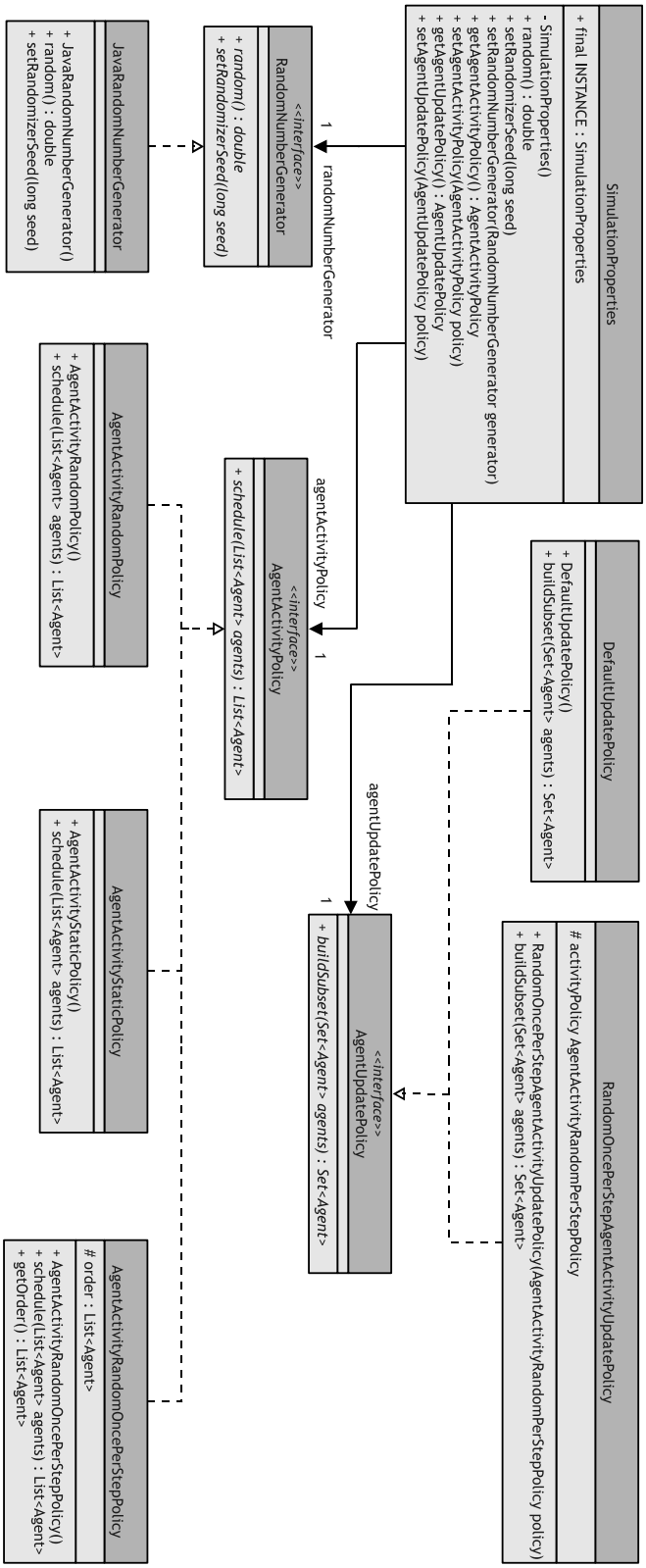


FIGURE 5.3 – Diagramme UML résumant la structure logicielle utilisée pour spécifier les paramètres de la plateforme JEDI.

Le premier paramètre de JEDI permet de caractériser finement comment les nombres aléatoires sont générés par la méthode *random()*. Ce paramètre est représenté sous la forme d'un attribut présent dans la classe *SimulationProperties* s'intitulant *randomNumberGenerator*, et qui implémente l'interface *RandomNumberGenerator*. Cette interface définit une méthode intitulée *setRandomizerSeed(long)* qui définit la graine utilisée pour initialiser le générateur de nombres aléatoires, ainsi qu'une méthode intitulée *random()*, qui retourne un nombre à virgule flottante aléatoire (plus précisément, un double), situé dans l'intervalle $[0; 1[$. La valeur de cet attribut peut être redéfinie par la méthode statique intitulée *setRandomNumberGenerator(RandomNumberGenerator)* de la classe *SimulationProperties*.

Pour simplifier l'initialisation et l'utilisation des générateurs de nombres aléatoires dans JEDI, nous avons de plus défini deux méthodes statiques intitulées *random()* et *setRandomizerSeed(long)* dans la classe *SimulationProperties*, dont le rôle est de déléguer l'appel aux méthodes éponymes de son attribut *randomNumberGenerator*. Nous proposons de plus une implémentation par défaut de l'interface *RandomNumberGenerator* que nous avons intitulée *JavaRandomNumberGenerator*, qui consiste a utiliser une instance de la classe *java.util.Random* fournie pour générer des nombres aléatoires.

5.2.2 Ordonnancement de l'activité des entités

Comme nous l'avons énoncé au début de cette sous-section, l'ordonnancement de l'activité des entités peut avoir une grande influence dans les résultats obtenus. Dans le cas où le temps est modélisé par un ensemble discret de pas de temps, au cours desquels chaque entité agit selon une séquence particulière, l'ordonnancement consiste à définir comment est construite cette séquence lors de chaque pas de temps.

Le second paramètre de JEDI permet de caractériser finement la construction de la séquence utilisée pour ordonner l'activité des entités, en ayant pour seule donnée l'ensemble des entités actives présentes dans l'environnement au début du pas de temps. Ce paramètre est représenté sous la forme d'un attribut présent dans la classe *SimulationProperties* qui s'intitule *agentActivityPolicy*, et qui implémente l'interface *AgentActivityPolicy*. Cette interface définit une unique méthode intitulée *schedule(List<Agent>)* qui est appelée dans le moteur de simulation au début de chaque pas de temps, en ayant pour paramètre l'ensemble des entités actives présentes dans l'environnement au début du pas de temps. Cette méthode construit puis retourne une instance de l'interface *List<Agent>*, qui représente la séquence dans laquelle les entités vont agir. La valeur de l'attribut *agentActivityPolicy* peut être redéfinie par la méthode statique intitulée *setAgentActivityPolicy(AgentActivityPolicy)* de la classe *SimulationProperties*.

La plateforme JEDI propose trois implémentations réutilisables de l'interface *AgentActivityPolicy*, qu'il est possible de compléter par des implémentations spécifiques à un problème de simulation particulier. Bien que le comportement le plus courant de la méthode *schedule(List<Agent>)* consiste à retourner une version réordonnée de liste des entités actives, l'ordonnancement dans JEDI n'y est pas restreint. Il peut aussi consister à extraire un sous-ensemble ordonné de cette liste, comme l'illustre la troisième implémentation que nous proposons.

La première façon d'ordonnancer, utilisée par défaut, est celle définie dans les algorithmes du chapitre 4. Elle consiste à réordonner aléatoirement la liste des entités actives fournie en paramètres, en utilisant le générateur de nombres aléatoires défini dans la classe *SimulationProperties*. Son implémentation est faite dans une classe intitulée *AgentActivityRandomPolicy*, dont l'implémentation de la méthode *schedule* est fournie dans l'algorithme 5.4). Cet ordonnancement est particulièrement utile pour des simulations telles que SugarScape, où l'ordre dans lequel les entités agissent a une forte influence sur les résultats obtenus.

FIGURE 5.4 – Ordonnancement aléatoire de l'activité des entités au sein d'un pas de temps, tel qu'implémenté dans la classe *AgentActivityRandomPolicy*

```
public List<Agent> schedule(List<Agent> list){
    Collections.shuffle(list, SimulationProperties.getInstance().getRandomizer());
    return list;
}
```

La seconde façon d'ordonnancer est définie dans la classe *AgentActivityStaticPolicy*. Elle consiste à ne pas modifier l'ordre dans lequel les entités agissent, et à retourner directement l'ensemble des entités

actives de la simulation. Cet ordonnancement est intéressant pour des simulations telles que le tri du couvain, où l'ordre dans lequel les entités agissent n'a pas d'influence sur les résultats obtenus.

La dernière façon d'ordonnancer que nous proposons est définie dans la classe *AgentActivityRandomOncePerStepPolicy*, et consiste à conserver un ordre rigoureusement identique à celui obtenu à l'aide de l'ordonnanceur *AgentActivityRandomPolicy*, en garantissant toutefois que seule une entité initiera une interaction au cours d'un pas de temps (voir algorithme 5.5). Il permet donc de réduire la granularité du temps de la simulation, tout en conservant la même façon d'ordonnancer l'activité des entités (voir figure 5.6). Cet ordonnanceur est défini afin de pallier à l'absence d'observateurs au niveau des individus : en s'assurant que seule une entité initie une interaction par pas de temps, l'observation au niveau supra-individuel (*i.e.* à la fin d'un pas de temps) est équivalent à l'observation au niveau des individus (sous réserve d'utiliser une mise à jour appropriée des entités, décrite dans la section qui suit).

FIGURE 5.5 – Ordonnancement aléatoire de l'activité des entités au sein d'un pas de temps, tel qu'implémenté dans la classe *AgentActivityRandomOncePerStepPolicy*. Le diagramme UML de la figure 5.3 résume la structure de la classe où cette méthode est déclarée.

```
public List<Agent> schedule(List<Agent> list){
    List<Agent> result = new LinkedList<Agent>();
    if(order.isEmpty()){
        // Début d'un nouveau pas de temps.
        // L'attribut order mémorise alors l'ensemble des sous-pas de temps qu'il faut exécuter
        list = AgentActivityRandomPolicy.INSTANCE.schedule(list);
        order.addAll(list);
    }
    if(! order.isEmpty()){
        // Ordonnancement du prochain sous-pas de temps
        result.add(order.remove(0));
    }
    return result;
}
```

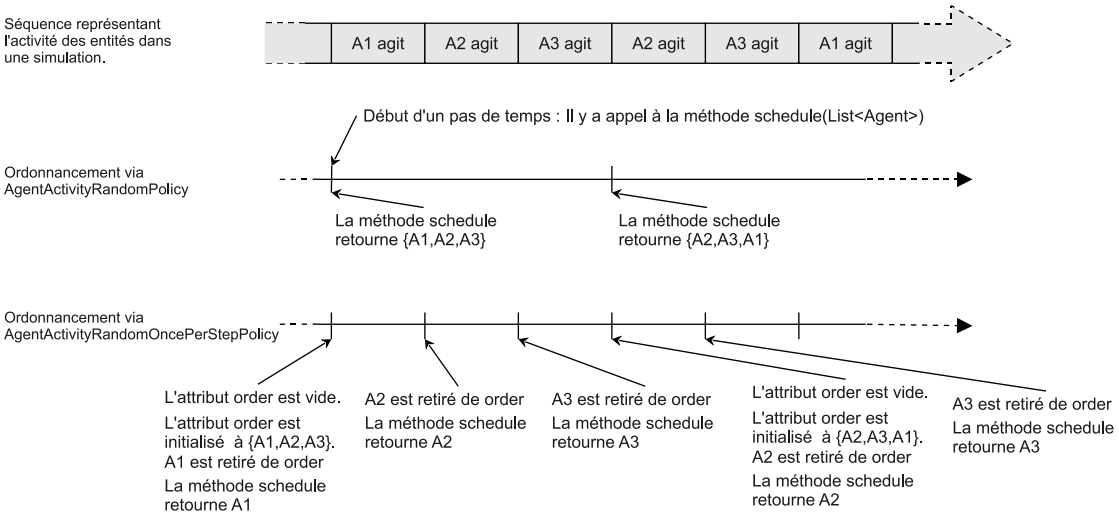


FIGURE 5.6 – Illustration de l'équivalence entre les ordonnanceurs de l'activité des entités, décrits par les classes *AgentActivityRandomPolicy* et *AgentActivityRandomOncePerStepPolicy*.

5.2.3 Gestion de la mise à jour des entités

Le changement de la granularité de la représentation du temps ne se fait pas uniquement au niveau de l'ordonnancement de l'activité des entités, mais aussi au niveau de la gestion de la mise à jour de l'état des entités. La gestion de la mise à jour consiste à déterminer quelles entités labiles doivent mettre à jour leur état au début d'un pas de temps.

Le dernier paramètre de JEDI permet de caractériser finement la construction du sous-ensemble caractérisant les entités étant mises à jour au début d'un pas de temps, en ayant pour seule donnée l'ensemble des entités labiles présentes dans l'environnement au début du pas de temps. Ce paramètre est représenté sous la forme d'un attribut intitulé *agentUpdatePolicy*, qui est présent dans la classe *SimulationProperties*, et qui implémente l'interface *AgentUpdatePolicy*. Cette interface définit une unique méthode intitulée *buildSubSet(Set<Agent>)*, qui est appelée dans le moteur de simulation au début de chaque pas de temps, en ayant pour paramètre l'ensemble des entités labiles présentes dans l'environnement. Cette méthode construit puis retourne une instance de l'interface *Set<Agent>*, qui représente le sous-ensemble des entités labiles qui est mis à jour au début du pas de temps courant. La valeur de l'attribut *agentUpdatePolicy* peut être redéfinie par la méthode statique intitulée *setAgentUpdatePolicy(AgentUpdatePolicy)* de la classe *SimulationProperties*.

La plateforme JEDI propose deux implémentations réutilisables de l'interface *AgentUpdatePolicy*, qu'il est possible de compléter par des implémentations spécifiques à un problème de simulation particulier.

La première façon d'extraire l'ensemble des entités mises à jour, utilisée par défaut, est définie dans la classe *DefaultUpdatePolicy*. Elle consiste à mettre à jour toutes les entités labiles au début de chaque pas de temps. Sa méthode *buildSubSet(Set<Agent>)* consiste donc à retourner l'ensemble des entités labiles de la simulation. Il s'agit de la politique de mise à jour décrite dans le modèle IODA.

La seconde façon d'extraire l'ensemble des entités mises à jour est définie dans la classe *AgentActivityRandomOncePerStepUpdatePolicy*. Elle est utilisée afin de garantir l'équivalence entre les ordonnanceurs de l'activité des entités des classes *AgentActivityRandomPolicy* et *AgentActivityRandomPerStepPolicy*. Pour cela, elle assure que les agents ne sont mis à jour que lors des pas de temps où l'attribut *order* de la politique d'ordonnancement *AgentActivityRandomOncePerStepUpdatePolicy* est réinitialisé. Une instance de la classe *AgentActivityRandomPerStepUpdatePolicy* mémorise donc dans un attribut nommé *activityPolicy* l'instance de la classe *AgentActivityRandomPerStepPolicy* à laquelle elle est associée. Sa méthode *buildSubSet(Set<Agent>)* consiste à retourner l'ensemble vide si la liste *order* de son attribut *activityPolicy* est non vide, et à retourner l'ensemble des entités labiles dans le cas contraire (voir l'algorithme 5.7).

FIGURE 5.7 – Méthode de la classe *AgentActivityRandomOncePerStepUpdatePolicy* assurant la mise à jour de toutes les entités labiles seulement lors de certains pas de temps. Cette politique de mise à jour n'est utilisée que conjointement à la politique d'ordonnancement de l'activité des entités de la classe *AgentActivityRandomPerStepPolicy*.

```
public Set<Agent> buildSubSet(Set<Agent> agents){
    if(this.activityPolicy.getOrder().isEmpty()){
        return agents;
    } else {
        return new HashSet<Agent>();
    }
}
```

5.3 Implémenter une simulation avec JEDI

L'implémentation d'expériences se fait avec JEDI en deux phases. La première consiste à construire un simulateur à l'aide des informations contenues dans le modèle. Lors de cette phase, chaque élément du modèle est transformé en un élément de l'implémentation, en utilisant des moyens tels que l'implémentation directe, la génération de code, ou la transformation de modèles. La seconde phase consiste à

spécifier l'expérience effectuée sur le simulateur ainsi créé. Elle comprend la spécification de l'initialisation de l'environnement de la simulation, l'identification du critère de terminaison de la simulation, ou encore l'identification des observateurs de la simulation.

Dans cette section, nous illustrons dans un premier temps comment utiliser les classes définies dans la section 5.1 afin de fournir une implémentation à un modèle. Cette implémentation n'est toutefois pas suffisante pour réaliser des expériences. En effet, en plus des interactions entre entités, du comportement des entités, et de l'environnement dans lequel les entités interagissent, une expérience est caractérisée par un état initial de la simulation, où des entités sont créées et ajoutées à l'environnement, mais aussi par la nature des résultats expérimentaux obtenus et la façon de les récupérer au fil de la simulation, ou encore par un critère d'arrêt déterminant quand stopper la simulation. Nous caractérisons ces deux points séparément dans un second temps dans cette section.

5.3.1 Implémentation d'un modèle avec JEDI

Les descriptions fournies dans la sous-sous section précédente ont pour but de décrire la structure utilisée pour implémenter un modèle construit avec l'approche IODA. Dans cette sous-sous section, nous spécifions comment implémenter un modèle en utilisant cette structure.

Puisque la plateforme JEDI fournit déjà l'implémentation de l'environnement et ses primitives, l'implémentation d'une simulation consiste à spécifier les interactions, spécifier la matrice d'interaction, spécifier la matrice de mise à jour, et enfin spécifier les primitives abstraites et le halo des entités. Nous profitons de cette section pour illustrer comment écrire une simulation simple de type « HelloWorld ».

Interactions L'implémentation d'une interaction individuelle $\mathcal{I}$ revient dans un premier temps à créer deux interfaces étendant l'interface *EntitySignature*, dont le nom est défini par la concaténation de la chaîne "SourceDe" et l'identifiant $id(\mathcal{I})$ de l'interaction, ou par la concaténation de la chaîne "CibleDe" et l'identifiant $id(\mathcal{I})$ de l'interaction. Ces interfaces représentent la signature des entités dans l'interaction créée. La seconde étape de cette implémentation consiste à créer une classe dont le nom correspond à l'identifiant $id(\mathcal{I})$ de l'interaction. Cette classe doit étendre la classe abstraite *SingleTargetInteraction*, ainsi qu'attribuer la concaténation de "SourceDe" et l'identifiant $id(\mathcal{I})$ comme valeur au type générique *Source*, et attribuer la concaténation de "CibleDe" et l'identifiant $id(\mathcal{I})$ comme valeur au type générique *Target*. La troisième étape de l'implémentation d'une interaction consiste à implémenter la spécification des préconditions, du déclencheur, et des actions de l'interactions, d'y identifier les primitives abstraites introduites, puis de reporter ces primitives dans les deux interfaces créées précédemment.

L'implémentation d'une interaction dégénérée est similaire, à cela près qu'une seule une interface, dont le nom est la concaténation de "SourceDe" et l'identifiant $id(\mathcal{I})$, est nécessaire à son implémentation.

Nous illustrons l'implémentation de ces deux types d'interactions, au travers de deux exemples.

***Exemple. Interaction dégénérée* VIEILLIR**
La figure 5.8 décrit l'implémentation des classes et interfaces nécessaires à la définition d'une interaction dégénérée dont l'identifiant est VIEILLIR*. Dans cette interaction, qui n'est soumise à aucunes conditions, une entité source voit son age augmenter.*

***Exemple. Interaction individuelle* SALUER**
La figure 5.9 décrit l'implémentation des classes et interfaces nécessaires à la définition d'une interaction individuelle dont l'identifiant est SALUER*. Dans cette interaction, qui n'est soumise à aucunes conditions, une entité source envoie un message de salutation à l'entité cible.*

Remarque. Si une signature n'introduit que des primitives abstraites figurant déjà dans l'interface *EntitySignature*, alors il n'est pas nécessaire d'introduire de nouvelle interface pour cette signature. Dans ce cas, le type générique lui étant associé (*i.e. Source* ou *Target*) vaut *EntitySignature*.

Matrice d'interaction et Matrice de mise à jour L'implémentation d'une matrice d'interaction ou d'une matrice de mise à jour ne nécessite pas la définition de nouvelles classes ou interfaces. Elle ne requiert que l'instanciation des classes *InteractionMatrix* et *UpdateMatrix*.

FIGURE 5.8 – Implémentation d'une interaction dégénérée VIEILLIR avec la plateforme JEDI.

```
public interface SourceDeVieillir extends EntitySignature {
   // Augmente l'age d'une entité d'une valeur t
   public void augmenterAge(int t);
}
```

```
public class Vieillir extends DegenerateInteraction<SourceDeVieillir>{
   public boolean declencheur(EnvironmentSignature environment,
                              SourceDeVieillir source){
      return true;
   }
   public boolean preconditions(EnvironmentSignature environment,
                                SourceDeVieillir source){
      return true;
   }
   public void actions(EnvironmentSignature environment,
                       SourceDeVieillir source){
      source.augmenterAge(1);
   }
}
```

FIGURE 5.9 – Implémentation d'une interaction individuelle SALUER avec la plateforme JEDI.

```
public interface SourceDeSaluer extends EntitySignature {
   // Retourne un message de salutation
   public String messageDeSalutation();
}
```

```
public interface CibleDeSaluer extends EntitySignature {
   // Permet de recevoir un message envoyé par une autre entité
   public void recevoirMessage(String message);
}
```

```
public class Saluer extends SingleTargetInteraction<SourceDeSaluer,CibleDeSaluer>{
   public boolean declencheur(EnvironmentSignature environment, SourceDeSaluer source,
                              CibleDeSaluer target){
      return true;
   }
   public boolean preconditions(EnvironmentSignature environment, SourceDeSaluer source,
                                CibleDeSaluer target){
      return true;
   }
   public void actions(EnvironmentSignature environment, SourceDeSaluer source,
                       CibleDeSaluer target){
      String message = source.messageDeSalutation();
      target.recevoirMessage(message);
   }
}
```

Famille d'entités L'implémentation d'une famille d'entités $\mathcal{F}$ dans JEDI revient dans un premier temps à créer une classe qui étend la classe abstraite *Agent*, de lui donner pour nom l'identifiant $id(\mathcal{F})$ du modèle de la famille d'entités, et de lui associer un constructeur vide. Dans un second temps, cette classe est complétée en lui faisant étendre toutes les interfaces représentant la signature de cette famille d'entités dans les interactions qu'elle est capable d'initier ou de subir. Dans un dernier temps, cette classe est complétée par l'implémentation des diverses primitives abstraites présentés dans les interfaces qu'elle étend, et par l'ajout des attributs manipulés dans ces primitives.

Exemple. Famille d'entités `ExempleDeFamille`
En supposant qu'une famille d'entités dont l'identifiant est `ExempleDeFamille` *est capable d'initier les interactions* VIEILLIR *et* SALUER*, et qu'elle est capable de subir l'interaction* SALUER*, alors son implémentation dans* JEDI *correspond au code présent dans la figure 5.10.*

FIGURE 5.10 – Implémentation d'une famille d'entités `ExempleDeFamille` avec la plateforme JEDI. Cette famille peut initier et subir l'interaction SALUER, et initier l'interaction VIEILLIR.

```
public class ExempleDeFamille extends Agent implements SourceDeSaluer,CibleDeSaluer,
                                                        SourceDeVieillir {
    // Attributs de la famille d'entités
    protected int age;
    // Constructeur vide de la famille d'entités
    public ExempleDeFamille(){
        super();
        age = 0;
    }
    // Primitives abstraites provenant de la signature SourceDeSaluer
    public String messageDeSalutation(){
        return "Bonjour, j'ai " + this.age + " ans. Et vous ?";
    }
    // Primitives abstraites provenant de la signature CibleDeSaluer
    public void recevoirMessage(String message){
        System.out.println("J'ai reçu le message : \"" + message + "\"");
    }
    // Primitives abstraites provenant de la signature SourceDeVieillir
    public void augmenterAge(int t){
        this.age = this.age + t;
    }
}
```

Instanciation d'entités Les classes décrites dans la section 5.3.1 suffisent à implémenter un modèle. Toutefois, l'instanciation d'une famille d'entités requiert la définition de la ligne de la matrice d'interaction et de la cellule de la matrice d'interaction lui étant associées. Afin de faciliter cette instanciation, une classe particulière est mise en place pour être utilisée comme fabrique abstraite d'entités. Cette fabrique permet de centraliser la déclaration des matrices d'interaction et de mise à jour, ainsi que la déclaration des halo, tout en permettant de créer des instances de familles d'entités simplement, à l'aide d'une méthode appelée *createEntity(Class<T>, double,double)*, ou d'une méthode appelée *createEntity(Class<T>)*.

La fabrique mentionnée ici s'implémente sous la forme d'une classe étendant la classe abstraite *DefaultEntityFactory*, dont il faut spécifier une unique méthode abstraite intitulée *initializeFactory()*. Cette méthode est utilisée pour spécifier la matrice d'interaction, la matrice de mise à jour, et déterminer quelle est la surface de perception référente associée à chaque famille d'entités, à l'aide des méthodes dont le nom est *addAssignationElementToInteractionMatrix* (pour initialiser la matrice d'interaction), *addAssignationElementToUpdateMatrix* (pour initialiser la matrice de mise à jour) ou *setReferentSurfaceFor* (pour définir la surface de perception référente d'une entité).

Exemple. Fabrique de la simulation HelloWorld
Si nous poursuivons notre exemple reposant sur les interactions VIEILLIR *et* SALUER*, et reposant sur*

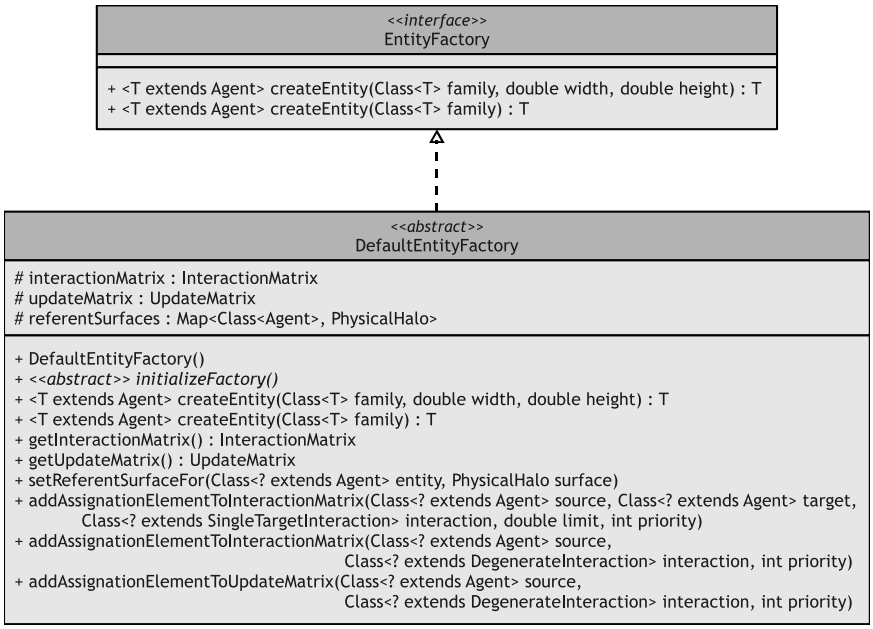


FIGURE 5.11 – Troisième partie du diagramme UML décrivant les classes et interfaces de la plateforme JEDI directement manipulées pour implémenter un modèle décrit avec la méthodologie IODA.

la famille d'entités `ExempleDeFamille`*, alors la fabrique permettant de créer des instances de la famille d'entités* `ExempleDeFamille` *est implémentée selon le code décrit dans la figure 5.12. Si* `fabric` *est une instance de cette classe nouvellement créée, alors une entité de la famille* `ExempleDeFamille` *est instanciée par exemple par l'appel de méthode :* `fabric.createEntity(ExempleDeFamille.class, 1, 1)`*.*

Remarque. Cette fabrique ne fait que créer une instance d'une famille d'entités dont la ligne de la matrice d'interaction, la cellule de la matrice de mise à jour, et la surface de perception référente sont initialisées correctement. L'initialisation à proprement parler d'une entité, dans le cadre d'une expérience particulière, n'est pas encore effectuée ici. Leur initialisation doit être faite lors de l'exécution d'une interaction, à l'aide d'une primitive abstraite, dans le constructeur de l'entité, ou lors de l'initialisation de la simulation.

5.3.2 L'observation des résultats d'une simulation

Un simulateur est un programme informatique dont l'objectif est de valider ou invalider un modèle, à l'aide d'expériences. Il constitue une implémentation plus ou moins fidèle du modèle et peut donc, dans certains cas, aboutir à des expérimentations fournissant des résultats erronés. Il se révèle alors primordial d'identifier l'origine des erreurs, et de les corriger. Cette vérification se fait au moyen de l'étude de la trace de simulations, *i.e.* l'observation pas à pas de l'exécution d'expériences conduites avec ce simulateur.

Un simulateur est aussi un moyen utilisé pour valider un modèle, *i.e.* pour s'assurer que si le modèle reproduit de manière satisfaisante le phénomène étudié. Il permet ainsi de déterminer si les hypothèses sur lesquelles se fonde le modèle fournissent une explication candidate à l'apparition du phénomène, ou si au contraire elles ne permettent pas d'expliquer l'apparition du phénomène étudié sous leur forme actuelle. La validation est faite au moyen de la comparaison de données extraites du phénomène, et de données extraites de différentes expériences menées sur un simulateur, et se fonde donc sur l'observation de l'exécution d'une simulation.

Dans les deux cas mentionnés ci-dessus, il est fondamental de pouvoir observer l'exécution de la simulation à plusieurs niveaux, et d'en retirer un maximum d'informations, afin de les interpréter. L'ob-

FIGURE 5.12 – Implémentation de la fabrique permettant de créer des instances de la famille d'entités `ExempleDeFamille` avec la plateforme JEDI. Cette famille peut initier et subir l'interaction SALUER, et initier l'interaction VIEILLIR.

```
public class ExempleDeFabrique extends DefaultEntityFactory {
    // Constructeur de la fabrique (optionnel)
    public ExempleDeFabrique(){
        super();
    }

    public void initializeFactory(){
        // Initialisation de la matrice d'interaction
        // Une entité peut en saluer une autre à une distance de 3.5, avec une priorité de 1
        this.addAssignationElementToInteractionMatrix(ExempleDeFamille.class,
                                        ExempleDeFamille.class,Saluer.class, 3.5, 1);
        // Initialisation de la matrice de mise à jour
        this.addAssignationElementToInteractionMatrix(ExempleDeFamille.class,Vieillir.class,1);
        // Définition de la surface de perception référente de la famille ExempleDeFamille
        // La perception se fait dans un cone de 90° de longueur 3.
        this.setReferentPerceptionSurface(ExempleDeFamille.class,
                                        HaloBuilder.createConeShapedArea(3, Math.PI / 2));
    }
}
```

servation de ces informations peut être effectué à deux niveaux selon [PMR+05] : l'« observation à l'échelle de l'individu », qui consiste à « [observer et éventuellement éditer] les informations spécifiques à un agent (valeurs des attributs, messages,courbes d'évolution des attributs) », et l'« observation à l'échelle supra-individuelle », qui consiste à « observer les interactions entre agents communiquant et négociant par échange de messages ». Dans le cas de la méthodologie IODA, l'observation prend un sens plus précis, permettant ainsi de faciliter l'interprétation des données observées : ces informations peuvent être la valeur des attributs d'une entité, le voisinage de l'entité lors de ce pas de temps, les interactions auxquelles l'entité a participé (et non plus seulement les messages échangés), le nombre d'entités dans l'environnement, ou toute valeur calculée à partir de ces informations (par exemple la densité en entités d'une zone de l'environnement, le nombre moyen de naissances et décès, l'age moyen des entités, *etc.*). Nous décrivons ci-après les moyens mis en place dans la plateforme JEDI afin de prendre en compte ce problème.

L'observation : un problème complexe

La construction d'observateurs de la simulation est une tache s'avérant complexe, pour deux raisons. D'une part un observateur doit être non-invasif dans le simulateur, *i.e.* l'ajout d'un observateur ne doit pas influer sur l'implémentation du modèle. La résolution de ce problème passe par la mise en place d'une logicielle évoluée au sein du simulateur, utilisant par exemple des patrons de conception tels que le Modèle-Vue-Contrôleur ou Observateur/Observé, ou encore une structure logicielle utilisant la programmation par aspects, *etc.* Des travaux tels que ceux de Ralambondrainy *et al.* [RCP06, PMR+05] vont en ce sens, et proposent une ontologie pouvant être utilisée pour fournir une structure logicielle générique et réutilisable d'observateurs, pouvant autant observer au niveau de l'individu qu'au niveau supra-individuel. D'autre part, un observateur doit observer une simulation en un temps raisonnable, sans quoi le nombre d'expériences qu'il est possible de conduire avec le simulateur peut être réduit significativement, en particulier lorsque les observations portent sur le niveau de l'individu. En effet, l'observation de toutes les entités est extrêmement coûteux dès que le nombre d'entités dans une simulation augmente. Des travaux tels que ceux de Morvan *et al.* [MVDJ09] s'intéressent à ce problème, et proposent de réduire le nombre de calculs nécessaires à l'observation au niveau des individus. La réduction des calculs s'y fait en mesurant de manière approchée les résultats d'une observation, en se basant sur un filtrage statistique des entités observées.

Ces deux problèmes sortent du cadre de l'étude menée dans cette thèse, et nous n'avons pas ici

l'ambition de les résoudre. L'observation d'une simulation reste toutefois un problème fondamental à la constitution d'un simulateur, qu'il est nécessaire de prendre en compte. Nous présentons donc de manière minimale comment l'observation est prise en compte dans la plateforme JEDI, en limitant nos descriptions à la structure logicielle utilisée, ainsi que les différents observateurs spécifiés par défaut dans cette plateforme.

Structure des observateurs dans JEDI

JEDI offre la possibilité d'inspecter le contenu de l'environnement d'une simulation au début d'une expérimentation, à la fin d'une expérimentation, ainsi qu'à la fin de chaque pas de temps. Les données observées incluent la valeur des attributs des entités, les interactions auxquelles ils ont participé (via les tuple réalisable dont l'interaction a été initiée au cours du pas de temps), ou plus généralement l'environnement d'une simulation.

L'observation repose sur le patron de conception *Modèle-Vue-Contrôleur*, où un observateur est une vue sur la simulation exécutée. Une simulation est exécutée dans un processus instance de la classe *SimulationThread*. Ce processus se fige dans un état stable, propice à l'observation, au début de la simulation, à la fin de la simulation, ainsi qu'à la fin de chaque pas de temps. Un contrôleur instancié par la classe *SimulationCore* est notifié de ces situation, et déclenche alors en réponse la mise à jour de chaque observateur. La simulation ne se poursuit qu'une fois la mise à jour de tous les observateurs terminée (voir figure 5.13).

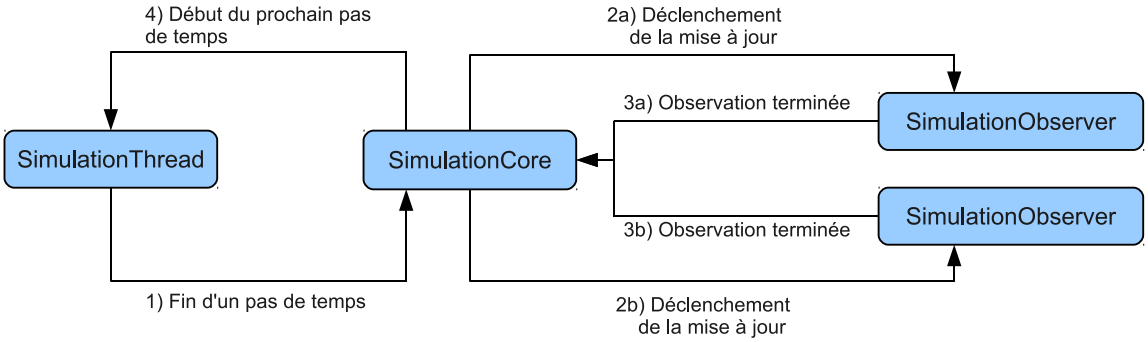


FIGURE 5.13 – Patron de conception Modèle-Vue-Contrôleur mis en place dans la plateforme JEDI afin d'observer des simulations. La classe « SimulationThread » correspond au processus où sont exécutés les pas de temps d'une simulation, la classe « SimulationCore » sert de pivot entre le modèle de la simulation et les observateurs, et contrôle la mise à jour des observateurs. La classe « SimulationObserver » permet de définir des observateurs dans JEDI.

La mise à jour d'un observateur consiste à extraire des informations de l'environnement ou des entités qui y résident, et à les mettre à la disposition des expérimentateurs, que cela soit sous la forme d'une interface graphique, ou sous la forme de fichier de données. Nous en fournissons deux exemples ci-après.

Exemple. Grille en deux dimensions
L'observation d'un environnement euclidien en deux dimensions peut être faite par l'affichage d'une grille en deux dimensions, où les entités sont représentées par des formes géométriques placées dans la grille en fonction de leur position. Dans ce cas, la mise à jour de l'observateur consiste à construire une image représentant une telle grille pour l'état courant de l'environnement, puis à l'afficher.

Exemple. Fichier de données
L'observation d'une simulation en éthologie peut être faite en mesurant l'évolution des populations de chaque espèce animale contenue dans l'environnement. Si une espèce animale est représentée par une famille d'entités, alors la mise à jour de cet observateur consiste à recenser le nombre d'entités de chaque famille d'entités de la simulation, et à écrire ces valeurs dans un fichier de données.

Observateurs fournis par défaut

La plateforme JEDI permet de greffer un nombre arbitraire d'observateurs à un simulateur, et offre une implémentation par défaut de quatre observateurs valides pour tout type de simulation ayant lieu dans un environnement euclidien continu en deux dimensions :

- Une interface graphique affichant l'environnement sous la forme d'une grille en deux dimensions, où une entité est représentée avec une forme géométrique colorée, ou avec un image ;
- Une interface graphique affichant la population actuelle de chaque famille d'entités ;
- Une interface graphique affichant l'historique des interactions auxquelles chaque entité a participé au cours de chaque pas de temps, ainsi que les pas de temps lors desquels une entité a été ajoutée ou retirée de l'environnement ;
- Un observateur mémorisant sous la forme d'un fichier des informations recensées à la fin de chaque pas de temps.

L'instanciation de ces quatre observateurs est décrite ultérieurement, dans la section 5.3.3.

Les valeurs observées lors d'une simulation, et la façon de les observer peut grandement varier d'une simulation à l'autre. Par conséquent, JEDI offre la possibilité de construire des observateurs spécifiques à une simulation, en créant une classe étendant la classe abstraite *SimulationObserver*. La spécification de l'observateur se fait par l'implémentation des trois méthodes *timeStepEnded(Environment,int)*, *simulationInitializationEnded(Environment)*, et *simulationEnded(Environment)*, qui décrivent la mise à jour de l'observateur respectivement lorsqu'un pas de temps se termine, lorsque l'initialisation de la simulation est terminée, et lorsque la simulation est terminée.

Dans JEDI, nous ne définissons pas de structure permettant de construire des observateurs au niveau des individus, *i.e.* au moment où chaque entité initie une interaction, qui se révèlent extrêmement coûteux en termes de performances à mettre en place. Leur définition reste toutefois possible en imposant qu'une seule entité puisse agir par pas de temps. Avec une gestion adaptée de l'ordonnanceur de l'activité et de la mise à jour des entités au cours des pas de temps, il est même possible d'obtenir des résultats strictement équivalent de ceux obtenus avec l'ordonnanceur « classique » décrit dans l'algorithme 6 page 125, sous réserve qu'aucune interaction, et qu'aucune primitive abstraite d'une entité ne manipule la valeur courante du pas de temps (voir section 5.2.2).

La figure 5.14 représente le diagramme UML des classes présentées ici.

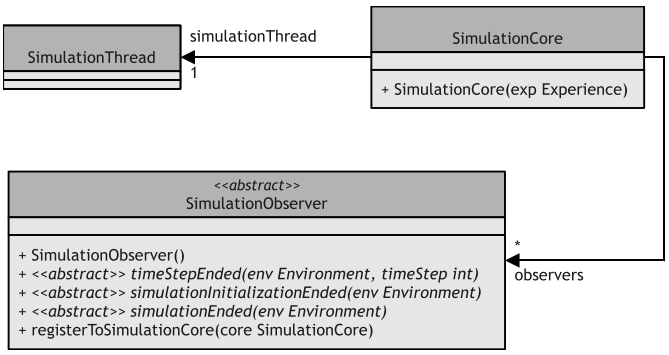


FIGURE 5.14 – Diagramme UML des classes permettant de définir des observateurs dans JEDI.

5.3.3 Construction d'une expérience

Une fois le modèle implémenté, il est possible de construire une expérience reposant sur le simulateur ainsi défini. Cette part n'est pas propre à la méthodologie IODA, et n'a donc pas été décrite dans notre approche formelle. Pour des raisons d'implémentation, nous proposons des classes permettant de créer une expériences. Il s'agit là d'un choix ayant permis de restreindre l'étude menée dans cette thèse.

A terme, la phase de spécification d'une expérience devrait disposer de son propre modèle, pouvant être spécifié et implémenté avec une approche transversale. En effet, comme le mentionnent Shannon *et*

al. ainsi que Grimm *et al.*[Sha98, GBB+06], un modèle validé ne prend son sens que si les résultats des simulations sont communiqués à la communauté, et vérifiables par cette communauté. Cela implique la description précise des expériences ayant permis d'obtenir les résultats communiqués (d'où la nécessité d'un modèle), mais aussi la possibilité de reproduire ces expériences (d'où la nécessité d'un processus garantissant une transition valide entre modèle et implémentation, et donc une approche transversale). Certains travaux, tels que [GBB+06, DDG09] vont en ce sens, et étudient comment modéliser l'initialisation de l'environnement d'une simulation. Ces spécifications sont toutefois en l'état insuffisantes, car peu formelles (par exemple le protocole ODD [GBB+06] ne fournit qu'une description verbale de ce qui doit être contenu dans un modèle), ou très proche d'un langage informatique (par exemple XELOC [DDG09], qui est une extension de XML nécessitant de définir ses propres balises afin de décrire une simulation). De plus, ces modèles ne sont actuellement pas intégrés à une approche de conception transversale. Il s'agit là d'une perspective que nous ne décrirons pas plus dans ce chapitre.

Dans JEDI, pour qu'une expérience puisse être exécutée, on doit avoir la connaissance :

- des dimensions de l'environnement, ainsi que de sa nature torique ou non selon l'axe des abscisses et l'axe des ordonnées ;
- de comment initialiser l'environnement de la simulation, ce qui implique de savoir quelles entités initialement instancier, comment initialiser leurs divers attributs, et en particulier leur surface dans l'environnement, et enfin savoir à quelle position les placer dans l'environnement ;
- des conditions sous lesquelles la simulation doit se terminer ;
- du moyen utilisé pour initialiser et démarrer une simulation ;
- de comment les résultats de la simulation sont observés, et donc de définir les observateurs de la simulation (voir section 5.3.2) ;
- de comment initialiser les divers paramètres du moteur de simulation (voir section 5.2).

Dans JEDI, une unique classe abstraite intitulée *Experiment* est étendue pour réaliser toutes ces tâches. Le diagramme UML spécifiant cette classe est représenté sur la figure 5.15.

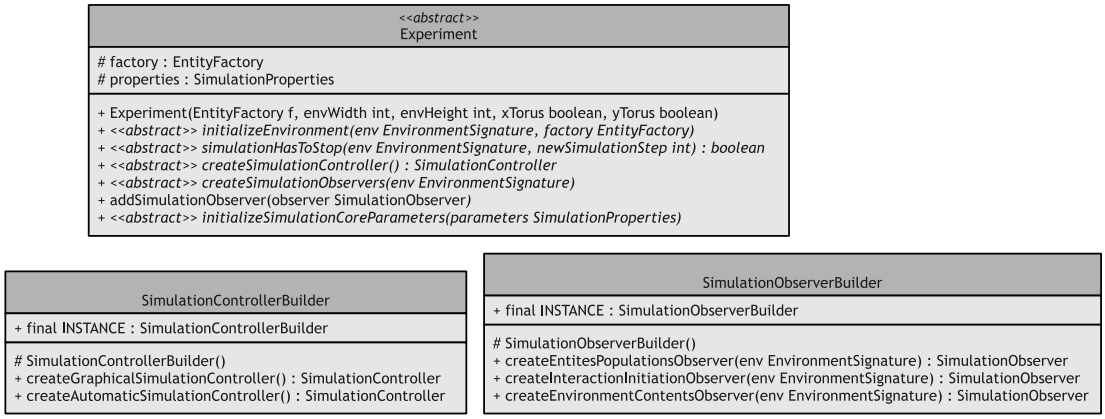


FIGURE 5.15 – Diagramme UML décrivant les classes et interfaces de la plateforme JEDI permettant de créer des expériences reposant sur l'implémentation d'un modèle IODA.

Création d'une expérience La création d'une expérience passe par la création d'une classe qui étend la classe abstraite *Experiment*. Cette classe doit disposer d'un constructeur dont l'unique paramètre est une instance de la classe *EntityFactory*, dont l'implémentation consiste à appeler le constructeur *Experiment(EntityFactory,double,double,boolean,boolean)* de la classe *Experiment*, en lui fournissant en paramètres la description de l'environnement utilisé pour cette expérience. Une telle description comprend la spécification de la largeur et de la hauteur de l'environnement, ainsi que la nature torique ou non de l'environnement selon l'axe des abscisses et l'axe des ordonnées.

Exemple. Création d'une expérience pour le modèle « HelloWorld »

Les lignes 1 à 5 de la figure 5.16 illustrent l'implémentation de la création d'une expérience, et de la spécification des dimensions de l'environnement avec JEDI.

FIGURE 5.16 – Implémentation avec la plateforme JEDI d'une expérience utilisant le modèle « Hello-World » décrit au long de la section 5.3.1.

```
01:  public class ExperienceAvecHelloWorld extends Experiment {
02:    public ExperienceAvecHelloWorld(){
03:      // L'expérience a lieu dans un environnement 10 x 10 non torique.
04:      super(10,10,false,false);
05:    }
06:    public void initializeEnvironment(EnvironmentSignature env, EntityFactory factory){
07:      // Création d'une entité dont la surface dans l'environnement est de 1 x 1
08:      ExempleDeFamille entite = factory.createEntity(ExempleDeFamille.class, 1, 1);
09:      // Placement de l'entité dans l'environnement, à la position (1,1)
10:      env.add(entite,1,1);
11:      // Création d'une autre entité dont la surface dans l'environnement est de 1 x 1
12:      entite = factory.createEntity(ExempleDeFamille.class, 1, 1);
13:      // Placement de l'autre entité dans l'environnement, à la position (1,2)
14:      env.add(entite,1.5,2);
15:    }
16:    public boolean simulationHasToStop(EnvironmentSignature env, int newSimulationStep){
17:      // La simulation se termine à la fin du premier pas de temps.
18:      return newSimulationStep == 2;
19:    }
20:    public SimulationController createSimulationController(){
21:      // L'initialisation de la simulation, et son démarrage sont gérés par des boutons.
22:      return SimulationControllerBuilder.INSTANCE.createGraphicalSimulationController();
23:    }
24:    public void createSimulationObservers(){
25:      // Dans cette simulation, nous observons les interactions étant déclenchées.
26:      SimulationObserver obs = SimulationObserverBuilder.INSTANCE
           .createInteractionInitiationObserver();
27:      this.addSimulationObserver(obs);
28:    }
29:    public void initializeSimulationCoreParameters(SimulationProperties parameters){
30:      // Dans cette simulation, nous utilisons "2000" comme graine dans le générateur
31:      // de nombres aléatoires.
32:      parameters.setRandomizerSeed(2000);
33:      // Nous utilisons la politique d'ordonnancement de l'activité des entités par
34:      défaut.
35:      // Nous utilisons la politique de mise à jour des entités par défaut.
36:    }
37:  }
```

Initialisation de l'environnement Le contenu de l'environnement est initialisé par la méthode abstraite *initializeEnvironment(EnvironmentSignature,EntitesFactory)* de la classe *Experiment*, qui prend en paramètres l'environnement initialisé, ainsi que la fabrique abstraite permettant de créer des instances de familles d'entités. L'implémentation de cette méthode consiste à créer des instances de familles d'entités à l'aide de la fabrique abstraite d'entités, d'initialiser les entités ainsi créées, puis de les ajouter à l'environnement.

Exemple. Initialisation de l'environnement pour le modèle « HelloWorld »
Les lignes 6 à 15 de la figure 5.16 illustrent l'implémentation de l'initialisation de l'environnement avec JEDI.

Critère d'arrêt d'une expérience Le critère déterminant quand stopper l'algorithme de simulation est défini dans la classe *Experiment* par une méthode abstraite appelée *simulationHasToStop(EnvironmentSignature,int)*, qui prend en paramètres deux valeurs, et retourne un booléen. Le premier paramètre est l'environnement dans lequel a lieu la simulation. Le second paramètre est le numéro du pas de temps sur le point de commencer. Si jamais cette méthode retourne « faux », le pas de temps sur le point de commencer est passé, et la simulation s'arrête. Les implémentation de ce critère sont très diversifiées, et peuvent par exemple consister à vérifier qu'un pas de temps maximal est atteint, ou encore que le nombre d'entités dans la simulation n'est pas nul, *etc.*

Exemple. Critère d'arrêt d'une expérience pour le modèle « HelloWorld »
Les lignes 16 à 19 de la figure 5.16 illustrent l'implémentation du critère d'arrêt d'une simulation avec JEDI.

Mode de contrôle de la simulation Selon les simulations effectuées, le contrôle du démarrage ou de l'initialisation d'une expérience peut se faire de manières différentes. Les deux plus courantes consistent soit à fournir une interface graphique, où des boutons permettent d'initialiser, de démarrer, ou encore de mettre en pause une simulation, soit à initialiser automatiquement une simulation, puis démarrer automatiquement la simulation, afin d'effectuer du traitement intensif de données.

Dans JEDI, la façon de contrôler une simulation est définie par une instance de la classe abstraite *SimulationController*, dont ne fournissons pas le détail dans ce chapitre. Deux implémentations par défaut de cette classe sont fournies, et donnent accès aux deux façons de contrôler une simulation présentées ci-avant. Une fabrique abstraite appelée *SimulationControllerBuilder* a été créée dans le but d'instancier simplement ces classes, à l'aide des deux méthodes *createGraphicalSimulationController()* et *createAutomaticSimulationController()*.

Le contrôleur utilisé dans une expérience est défini par la méthode abstraite *createSimulationController()*. Son implémentation consiste à appeler l'une des deux méthodes de la fabrique abstraite *SimulationControllerBuilder*, ou d'instancier directement un autre type de contrôleur.

Exemple. Mode de contrôle de la simulation pour le modèle « HelloWorld »
Les lignes 20 à 23 de la figure 5.16 illustrent l'implémentation du mode de contrôle de la simulation avec JEDI.

Observation de la simulation L'observation d'une simulation se fait dans JEDI à l'aide d'instances de la classe abstraite *SimulationObserver*, dont le principe de fonctionnement est exposé dans la section 5.3.2. La spécification des observateurs associés à une expérience se fait dans la méthode abstraite *createSimulationObservers(EnvironmentSignature)* de la classe *Experiment*. Son implémentation consiste à instancier les différents observateurs de l'expérience, et de les mémoriser en appelant la méthode *addSimulationObserver(SimulationObserver)* de la classe *Experiment*.

L'instanciation des observateurs définis par défaut dans JEDI se fait via une fabrique abstraite intitulée *SimulationObserverBuilder*, qui définit trois méthodes. La première méthode, appelée *createEntitiesPopulationsObserver(EnvironmentSignature)*, permet d'instancier un observateur affichant dans une interface graphique le nombre courant d'instances de chaque famille d'entités présentes dans l'environnement. La deuxième, appelée *createInteractionInitiationObserver(EnvironmentSignature)*, permet d'instancier un observateur affichant dans une interface graphique l'historique des interactions initiées et subies par chaque entité de l'environnement. La dernière méthode, appelée *createEnvironmentContentsObserver(EnvironmentSignature)*, permet d'instancier un observateur affichant dans une interface graphique le contenu de l'environnement, sous la forme d'une grille où les entités sont représentées par des formes géométriques colorées, ou par des images. Nous ne décrivons pas comment créer le dernier type d'observateur mentionné dans la section 5.3.2, qui consiste à mémoriser dans un fichier des données lues à chaque pas de temps.

Exemple. Observation de la simulation pour le modèle « HelloWorld »
Les lignes 24 à 28 de la figure 5.16 illustrent l'implémentation de la création des observateurs de la simulation avec JEDI.

Initialisation des paramètres du moteur de simulation La façon de paramétrer le moteur de simulation est déjà décrite dans la section 5.2. Cette étape de la construction d'une expérience consiste à mettre en pratique cette spécification dans la méthode abstraite *initializeSimulationCoreParameters(SimulationProperties)* de la classe *Experiment*.

Exemple. Paramètres du moteur de simulation pour le modèle « HelloWorld »
Les lignes 29 à 36 de la figure 5.16 illustrent l'implémentation de l'initialisation des paramètres du moteur de simulation avec JEDI.

Expérimentation Une fois la classe spécifiant une expérience définie, il est possible de créer la classe qui exécutera la simulation. Cette classe dispose d'une unique méthode statique intitulée *main(String[])*, qui consiste à exécuter l'expérience. Son implémentation consiste à instancier une fabrique d'entités telle que spécifiée dans la section 5.3.1, puis à instancier une expérience, dont la spécification est décrite dans la section 5.3.3, et enfin à créer une instance de la classe *SimulationCore* à l'aide de son constructeur *SimulationCore(Experiment)*, et à appeler la méthode *start* de cette instance.

Exemple. Classe principale de l'expérience utilisant le modèle « HelloWorld »
La figure 5.17 illustre l'implémentation de la classe principale utilisant le modèle « HelloWorld » décrit dans cette section, pour une expérience intitulée ExperienceAvecHelloWorld.

FIGURE 5.17 – Classe principale permettant d'exécuter l'expérience décrite dans la section 5.3.3, qui repose sur le modèle décrit dans la section 5.3.1

```
public class HelloWorldMain {
   public static void main(String[] args){
     EntityFactory fabrique = new ExempleDeFabrique();
     Experiment experience = new ExperienceAvecHelloWorld(fabrique);
     SimulationCore moteurDeSimulation = new SimulationCore(experience);
     moteurDeSimulation.start();
   }
}
```

5.4 L'environnement de développement intégré JEDI-Builder

Comme nous venons de le voir, écrire une simulation avec JEDI nécessite la création d'un grand nombre de classes pour implémenter fidèlement chaque concept de IODA. L'implémentation de tels modèle est facilitée s'il est possible de transformer automatiquement un modèle IODA en une implémentation.

Les principes énoncés textuellement dans la section 4.1 page 97 du chapitre 4 peuvent être automatisés afin que la construction d'un modèle IODA ne se fasse que de manière graphique, dans un environnement de développement intégré (IDE). Il devient alors possible de tirer parti de la description informatisée du modèle, et des algorithmes de simulation afin de générer automatiquement un simulateur dans un langage de programmation cible.

Afin d'éprouver ces principes nous avons développé un IDE appelé JEDI-BUILDER, qui permet de construire un modèle IODA, et d'en générer automatiquement l'implémentation pour la plateforme de simulation JEDI. La seconde vocation de ce prototype a été d'expérimenter comment adapter le concept d'héritage au modèle de IODA et à JEDI. Nous en décrivons le résultat dans le chapitre 8 page 221. Cet IDE a été implémenté en JAVA, et est composé de 239 classes et interfaces, cumulant en tout 21440 lignes de code (commentaires compris), équivalentes à 12770 lignes de code logique [18].

18. comptabilisées selon la métrique SLOC de la commande unix `sloccount`

Étapes de la méthodologie couverte par JEDI-BUILDER JEDI-BUILDER constitue la première étape d'une preuve de la possibilité d'implémenter la méthodologie IODA et de générer à partir des description du modèle une implémentation sur une plateforme reposant sur les principes de IODA. Nous avons pour cela émis deux hypothèses simplificatrices, que nous résumons sur la figure 5.18.

La première a consisté à restreindre à JEDI les plateformes pour lesquelles JEDI-BUILDER permettait de générer du code. Puisque dans JEDI, l'environnement est fixé et ne peut être modifié, les étapes de la méthodologie IODA permettant de définir la sémantique de l'environnement ne sont pas intégrées à JEDI-BUILDER (voir rectangle vert dans la figure 5.18). À l'avenir, cet IDE devra être étendu à la génération de code pour d'autres plateformes, afin de confirmer que les algorithmes décrits dans le chapitre précédent permettent non seulement de conserver la structure du modèle lors de l'implémentation, mais de plus que ces algorithmes sont valides pour tout type de langage de programmation.

La seconde a consisté à restreindre l'ensemble des étapes de la méthodologie couvertes par l'édition graphique dans JEDI-BUILDER aux étapes ne nécessitant pas de descriptions algorithmiques. Cette hypothèse simplificatrice est émise afin de maximiser le rapport entre effort d'implémentation de JEDI-BUILDER et gain en termes de facilité de conception d'un modèle IODA. En effet, les étapes reposant sur des algorithmes sont les plus spécifiques de la méthodologie IODA, et nécessitent un certain degré d'expertise en informatique pour être menées. Il est donc moins gênant que ces étapes soient réalisées directement par une implémentation selon le langage de programmation JAVA, plutôt que par des outils graphiques.

JEDI-BUILDER se focalise actuellement sur un sous-ensemble de sept étapes de la méthodologie IODA, résumées par le rectangle rouge dans la figure 5.18.

Identification des entités. L'étape d'identification des entités, notée B sur la figure 5.18, est effectuée dans JEDI-BUILDER par l'édition d'une liste de chaînes de caractères, où chaque chaîne représente l'identifiant d'une famille d'entités. Un identifiant de famille d'entités peut être ajouté, supprimé ou modifié à l'aide d'un menu contextuel apparaissant par un clic droit de la souris.

Identification des interactions. L'étape d'identification des interactions, notée C sur la figure 5.18, est effectuée dans JEDI-BUILDER de la même manière que l'étape d'identification des entités, par l'édition d'une liste de chaînes de caractères.

Spécification d'une matrice d'interaction brute. L'étape de spécification d'une matrice d'interaction brute, notée E sur la figure 5.18, consiste à faire glisser des identifiants situés dans la liste des interactions vers des cellules de la matrice d'interaction. Ces opérations mènent à l'ouverture d'un menu, dans lequel la garde de distance des éléments d'assignations ainsi créés peut être éditée.

Spécification d'une matrice d'interaction raffinée. L'étape de spécification d'une matrice d'interaction raffinée, notée G sur la figure 5.18, consiste à définir la priorité de chaque élément d'assignation, à partir du menu ayant permis d'éditer la garde de distance.

Spécification d'une matrice de mise à jour. L'étape de spécification de la matrice de mise à jour, notée H sur la figure 5.18, repose sur des principes identiques à l'édition de la matrice d'interaction brute. Elle a toutefois lieu dans une fenêtre graphique différente de celle permettant l'édition de la matrice d'interaction brute.

Spécification d'une matrice de mise à jour ordonnée. L'étape de spécification d'une matrice de mise à jour ordonnée, notée J sur la figure 5.18, repose sur des principes identiques à l'édition de la matrice d'interaction raffinée, et permet de modifier via un menu la priorité de chaque élément d'assignation figurant dans la matrice de mise à jour ordonnée.

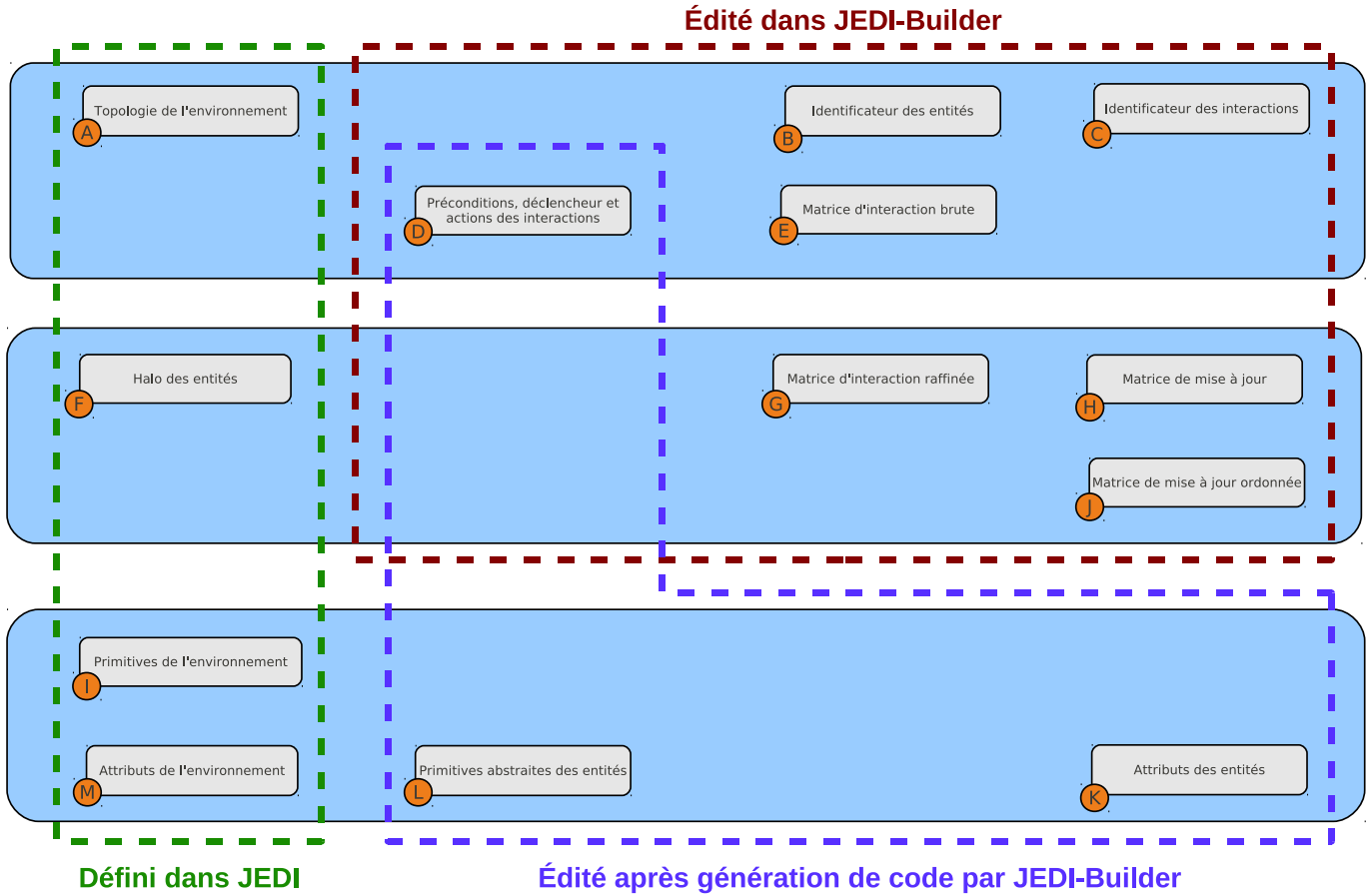


FIGURE 5.18 – Résumé des étapes de la méthodologie IODA couvertes par JEDI-BUILDER. Le rectangle vert représente la part de la méthodologie IODA établissant la sémantique de l'environnement, fixée par la palteforme JEDI, et ne devant pas conséquent pas être éditée dans JEDI-BUILDER. Le rectangle rouge représente la part de la méthodologie IODA pouvant être éditée graphiquement dans l'IDE JEDI-BUILDER. Le rectangle violet représente la part de la méthodologie IODA ne pouvant être édité qu'après la génération de code par JEDI-BUILDER.

Spécification des préconditions, du déclencheur et des actions des interaction. Dans la méthodologie de l'approche IODA, l'étape de spécification des préconditions, du déclencheur et des actions des interactions, notée D sur la figure 5.18, consiste à décrire les préconditions, le déclencheur et les actions d'une interaction, et d'en déduire les primitives de perception et d'action figurant dans la signature des source et dans la signature des cibles de cette interaction. Puisque nous avons émis l'hypothèse simplificatrice de spécifier les algorithmes après la génération de code, cette étape de la méthodologie consiste à établir la liste des primitives abstraites des sources et des cibles de chaque interaction. Ces primitives peuvent alors être éditées afin d'en définir l'identifiant, le type de retour, la description de sa fonction en langage naturel, ou les paramètres. La figure 5.19 illustre comment cette édition est effectuée par une capture d'écran de l'interface graphique de JEDI-BUILDER.

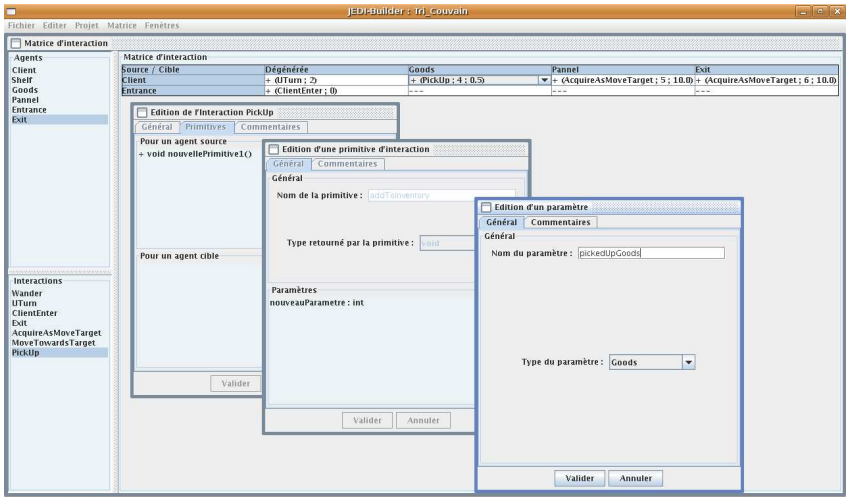


FIGURE 5.19 – Interface permettant d'éditer les primitives d'une entité dans JEDI-BUILDER.

Génération de code. A partir des informations de finies graphiquement, JEDI-BUILDER peut générer un ensemble de classes implémentant fidèlement le modèle décrit pour la plateforme JEDI. Les classes générées incluent l'implémentation :

- du squelette de chaque interaction, sous la forme d'une classe. Dans ce squelette, seules restent à définir l'implémentation des préconditions, du déclencheur et des actions ;
- de chaque signature d'entité dans une de ces interactions, sous la forme d'une interface ;
- du squelette de chaque famille d'entités, sous la forme d'une classe. Dans ce squelette, seules restent à définir l'implémentation des primitives abstraites, ainsi que la surface utilisée comme halo, et la primitive d'initialisation de l'entité ;
- d'une fabrique, sous la forme d'une classe, permettant de créer des instances de chaque famille d'agents ;
- de la matrice d'interaction brute et la matrice d'interaction raffinée, sous la forme d'un attribut de la fabrique ;
- du squelette d'une expérience, sous la forme d'une classe. Dans ce squelette, restent à définir le critère d'arrêt de la simulation, et l'initialisation de l'environnement.

Le code généré fait automatiquement le lien entre familles d'entités et signature d'entités, et génère à cette fin une méthode vide pour chaque primitive devant être spécifiée par une famille d'entités. Ainsi, le prototype de JEDI-BUILDER permet actuellement de générer automatiquement un squelette d'implémentation pour JEDI, dans lequel restent à implémenter les préconditions, le déclencheur et les actions des interactions, les primitives des entités, et enfin l'initialisation de la simulation et des entités.

Nous montrons ainsi qu'il est possible d'automatiser la première partie de la méthodologie IODA, et de passer automatiquement à son implémentation sur la plateforme JEDI.

Relation dans IODA	Implémentation dans JEDI
L'entité e a pour famille $\mathcal{F}$, *i.e.* $e \prec \mathcal{F}$	L'entité e est une instance (au sens de java) de la classe $\mathcal{F}$
Les instances de la famille d'entités $\mathcal{F}$ ont pour signature S dans une interaction $\mathcal{I}$	$\mathcal{F}$ implémente l'interface S
Les entités d'une simulation ont pour signature S dans l'environnement	Toutes les familles d'entités implémentent l'interface S
L'environnement a pour signature S dans une interaction	L'environnement implémente l'interface S

TABLE 5.1 – Tableau récapitulant comment les relations entre les concepts du modèle formel IODA sont implémentées dans la plateforme JEDI

5.5 Synthèse du chapitre

Dans ce chapitre, nous avons caractérisé la plateforme de simulation JEDI qui implémente IODA pour les environnements les plus couramment rencontrés en simulation : les environnements euclidiens à deux dimensions. Les tableaux 5.1 et 5.2 montrent qu'il y a une correspondance directe entre chaque concept de IODA et son implémentation dans JEDI sous la forme d'une classe, d'une interface, d'une méthode ou d'un attribut. JEDI est donc une implémentation fidèle de IODA qui conserve la structure du modèle. Elle permet ainsi :

- d'automatiser l'implémentation d'un modèle ;
- de réviser les modèles aisément grâce à la structure modulaire de l'implémentation ;
- de constituer des bibliothèques d'agents et des bibliothèques d'interaction qui facilitent la construction de modèles.

Nous prouvons ainsi que les avantages de IODA ne se limitent pas à la phase d'analyse d'un modèle.

La plateforme JEDI offre un ensemble de paramètres assurant la reproductibilité des simulations, et fournissant un contrôle total sur la prise de parole des entités et sur le déclenchement de leur mise à jour. Ce réglage fin est une nécessité, puisque les algorithmes utilisés pour spécifier une simulation particulière ne sont pas forcément adaptés à d'autres simulations. Les paramètres de IODA permettent de régler finement les algorithmes de simulation, et de les adapter aux besoins des simulations.

Une expérience nécessite plus que l'implémentation du modèle du phénomène. En réponse à cette nécessité, nous avons donc décrit différentes classes nécessaires à l'implémentation d'une expérience. Elles permettent entre autres de :

- définir le critère d'arrêt d'une simulation ;
- définir l'initialisation de l'environnement au début de l'expérience ;
- décrire comment observer la simulation et surtout comment obtenir les données correspondant aux résultats des expériences réalisées.

Enfin, nous avons conçu un prototype d'environnement de développement intégré (IDE) appelé JEDI-BUILDER, que nous utilisons pour confirmer la transversalité de l'approche IODA. En effet, cet IDE permet de spécifier un modèle IODA dans une interface graphique, et d'en générer automatiquement un squelette d'implémentation prêt à remplir pour la plateforme JEDI.

Le changement de perspective induit par notre approche centrée sur les interactions permet d'étudier les problèmes de simulation sous un angle nouveau. Dans la partie suivante, nous nous sommes appuyés sur JEDI et JEDI-BUILDER pour explorer des problématiques inhérentes à la simulation informatique, mais n'apparaissant pas explicitement dans les approches existantes, ou y étant traitées de manière non satisfaisante.

Notion dans IODA	Nature dans JEDI	Détails
Attribut d'une famille d'entités	Attribut	Un attribut d'une famille d'entités est représentée par un attribut au sens java dans la classe représentant la famille d'entités.
Cellule de la matrice de mise à jour	Classe	Une cellule de la matrice de mise à jour est une instance de la classe *fr.lifl.jedi.model.interactionDeclaration. UpdateMatrixCell*
Cellule de la matrice de mise à jour ordonnée	Classe	Une cellule de la matrice de mise à jour ordonnée est une instance de la classe *fr.lifl.jedi.model.interactionDeclaration. UpdateMatrixCell*
Entité	Instance	Une entité est une instance d'une famille d'entités, *i.e.* une instance d'une classe étendant la classe abstraite *fr.lifl.jedi.model.Agent*
Environnement	Classe	Un environnement est une instance de la classe *fr.lifl.jedi.model.Environment*
Famille d'entités	Classe	Une famille d'entités est une classe étendant la classe abstraite *fr.lifl.jedi.model.Agent*
Halo	Classe	Le halo est une instance de la classe *fr.lifl.jedi. model.halo.PhysicalHalo*, générée à l'aide de la classe *fr.lifl.jedi.model.halo.HaloBuilder* (une fabrique abstraite)
Interaction Dégénérée	Classe	Une interaction dégénérée est une classe (en général singleton) étendant la classe abstraite *fr.lifl.jedi.model.interactionDeclaration. DegenerateInteraction*
Interaction Individuelle	Classe	Une interaction individuelle est une classe (en général singleton) étendant la classe abstraite *fr.lifl.jedi.model.interactionDeclaration. SingleTargetInteraction*
Ligne de la matrice d'interaction brute	Classe	Une ligne de la matrice d'interaction brute est une instance de la classe *fr.lifl.jedi.model.interactionDeclaration. InteractionMatrixLine*
Modèle de sélection d'interaction (*i.e.* ligne de la matrice d'interaction raffinée)	Classe	Le modèle de sélection d'interaction réactif est une instance de la classe *fr.lifl.jedi.model.interactionDeclaration. InteractionMatrixLine*, où une priorité est associée à chaque élément d'assignation
Matrice d'interaction brute	Classe	La matrice d'interaction brute est une instance de la classe *fr.lifl.jedi.model.InteractionMatrix*
Matrice d'interaction raffinée	Classe	La matrice d'interaction raffinée est une instance de la classe *fr.lifl.jedi.model.InteractionMatrix*
Matrice de mise à jour	Classe	La matrice de mise à jour est une instance de la classe *fr.lifl.jedi.model.UpdateMatrix*
Matrice de mise à jour ordonnée	Classe	La matrice de mise à jour ordonnée est une instance de la classe *fr.lifl.jedi.model.UpdateMatrix*
Primitive abstraite	Méthode	Une primitive abstraite est une méthode
Signature d'une entité dans une interaction	Interface	La signature d'une entité dans une interaction est une interface étendant l'interface *fr.lifl.jedi.model.interactionDeclaration. EntitySignature*
Signature de l'environnement dans une interaction	Interface	La signature de l'environnement dans une interaction est une interface étendant l'interface *fr.lifl.jedi.model.interactionDeclaration. EnvironmentSignature*
Signature d'une entité dans l'environnement	Interface	La signature d'une entité dans l'environnement est une interface étendant l'interface *fr.lifl.jedi.model.interactionDeclaration. EntitySignature*

TABLE 5.2 – Tableau récapitulant comment chaque concept du modèle formel IODA est implémenté dans JEDI. Certains éléments n'y apparaissent pas car JEDI est une plateforme spécifiée pour un cas particulier des modèles IODA : les environnements euclidiens continus en deux dimensions, où les interactions sont de cardinalité $(1, 0)$ ou $(1, 1)$, où le halo est défini par la spécification d'une surface de perception, ou les entités ont un modèle de sélection d'interaction réactif axé sur les priorités.

Troisième partie

Exploration de problématiques de simulation avec IODA

Plan de la partie :

Le changement de perspective induit par notre approche centrée sur les interactions permet d'étudier les problèmes de simulation sous un angle nouveau. Dans cette partie, nous nous appuyons sur IODA, JEDI *et* JEDI-BUILDER *pour explorer des problématiques inhérentes à la simulation informatique, mais n'apparaissant pas explicitement dans les approches existantes, ou y étant traitées de manière non satisfaisante. Ces études ont abouti à l'identification de principes permettant d'étendre* IODA *et d'y intégrer des outils méthodologiques facilitant la conception de certaines catégories de simulations. Nous définissons ainsi une approche basée sur le principe d'extensions méthodologiques, reposant sur un cœur générique décrit dans la partie précédente, pouvant être complexifié par une ou plusieurs extensions lorsque la complexité du phénomène simulé le nécessite.*

Nous avons mené l'exploration en considérant trois problématiques de simulation différentes :

1. *Des interactions complexes sont-elles nécessaires pour modéliser des comportements complexes ?*
2. *Comment utiliser* IODA *afin d'éviter d'introduire es biais dans une simulation ?*
3. *Comment adapter le concept d'héritage afin de faciliter l'ingénierie des connaissances ?*

Chapitre 6

Étude sur les cardinalités des interactions

Plan du chapitre :

Afin d'expliquer au mieux l'apparition d'un phénomène, le modèle d'une simulation doit toujours en fournir une description la plus simple possible. Pour aider cet effort, le modèle ne doit pas reposer sur des interactions et des comportements d'agents plus complexes que le nécessitent ces explications. Puisque les actions impliquant simultanément plusieurs agents ne sont pas modélisées sous la forme d'interactions dans les approches actuelles, ces dernières éludent un problème pourtant réel, apparaissant explicitement dans notre approche centrée sur les interaction : « Des interactions simples suffisent-elles à exprimer des comportements complexes ? »

Nous étudions et analysons ce problème à l'aide de deux cas d'études se focalisant sur deux problèmes de simulation différents :

- *la coordination de plusieurs agents dans une interaction ;*
- *les interactions ayant un effet simultané sur plusieurs agents.*

Nous identifions à cette occasion :

- *deux patrons de conception, permettant de modéliser la plupart des interactions complexes impliquant plus de deux agents à l'aide d'interactions individuelles et dégénérées (section 6.1) ;*
- *un nouveau type d'interactions appelé « interactions multicast » permettant de modéliser simplement certaines interactions complexes ne pouvant être décomposées simplement en interactions individuelles et dégénérées (section 6.2).*

6.1 Des interactions simples suffisent-elles à modéliser des comportements complexes ?

Un phénomène réel peut être modélisé de manière arbitrairement complexe, en particulier si l'on cherche à en représenter les moindres détails. Bien que de tels modèles soient réalistes, ils n'offrent pas pour autant une « bonne » explication au phénomène puisqu'il y est difficile d'identifier les éléments ayant une réelle influence sur l'apparition du phénomène. Ainsi, un modèle formel offrant des concepts arbitrairement complexes ne fournit pas systématiquement de meilleurs explications à un phénomène que des modèles simples. La représentation des connaissances doit donc maximiser le rapport entre simplicité du modèle et ensemble des simulations pouvant être spécifiées.

Puisque les actions impliquant simultanément plusieurs agents ne sont pas modélisées sous la forme d'interactions dans les approches actuelles, ces dernières éludent un problème pourtant réel, apparaissant explicitement dans notre approche centrée sur les interaction : « Des interactions simples suffisent-elles à exprimer des comportements complexes ? ». Dans la description de IODA fournie dans la partie précédente, la cardinalité des interactions est restreinte à un unique agent source, et à aucun ou un seul agent

cible. Bien que ces interactions soient particulièrement adéquates pour représenter des interactions impliquant au plus deux agents, qu'en est-il des interactions en impliquant plus ? On peut en effet aisément imaginer des cas où plus de deux agents participent à une interaction :

- une réaction chimique catalysée par une enzyme, dans laquelle plus de deux substrats sont impliqués ;
- le transport d'un objet volumineux (par exemple une armoire), qui ne peut être transporté que par plusieurs personnes ;
- une bombe qui explose et blesse tous êtres vivants situés dans un certain périmètre ;
- un malade qui propage une maladie au contact d'autres personnes ;
- une parcelle de l'environnement qui transfère une part des phéromones qu'elle contient aux parcelles lui étant contiguës.

Dans cette section, nous discutons ce point, et montrons en quoi il est possible de représenter la plupart des interactions de cardinalité $(n, m) \in \mathbb{N}^2$ uniquement à l'aide d'interactions de cardinalité (1,1) et (1,0). Nous nous appuyons pour cela sur les exemples mentionnés ci-avant, et nous en déduisons deux modèles génériques permettant de décomposer une interaction en un ensemble d'interactions plus simples.

6.1.1 Problème de coordination

Dans cette première section, nous étudions comment modéliser deux phénomènes impliquant des interactions de cardinalité $(n, m) \in \mathbb{N}^2$, dans deux domaines d'application différents. Dans un premier temps, nous nous intéressons à la modélisation du transport d'un objet volumineux par plusieurs personnes, et donc à la modélisation d'interactions de cardinalité $(n, 1)$, avec $n \in \mathbb{N}$. Dans un second temps, nous étudions comment modéliser des réactions enzymatiques en biochimie, ce qui nous amène à considérer comment décomposer des interactions de cardinalité $(1, n)$, avec $n \in \mathbb{N}$. Ces deux études nous amènent à identifier un premier modèle générique permettant de décomposer toute interaction en un ensemble d'interactions de cardinalité (1,1) et (1,0), que nous présentons dans la section qui suit.

Modélisation du transport d'un objet volumineux

Nous illustrons la modélisation d'interactions impliquant plusieurs sources et une cible dans le contexte de simulations où un objet volumineux, par exemple une armoire, ne peut être déplacé que par les efforts conjoints de $n \in \mathbb{N}$ personnes. Ce phénomène peut être représenté dans IODA en modélisant les personnes par une famille d'agents `Personne`, en modélisant l'armoire par une famille d'agents `Armoire`, et en modélisant le déplacement de l'armoire par une interaction Transport, de cardinalité (n,1).

La restriction des cardinalités effectuées dans la description du modèle IODA ne permet pas de modéliser une interaction de cardinalité (n,1). Il est toutefois possible de modéliser ce phénomène à l'aide d'interactions de cardinalité (1,1) et (1,0), et s'affranchir ainsi de la limitation des cardinalités. Pour s'en convaincre, nous nous intéressons plus en détails au phénomène lié à cette interaction.

Le déplacement d'une armoire par n personnes est un problème de coordination. Une fois les personnes nécessaires rassemblées, le déplacement de l'armoire peut être effectué.

Modéliser ce processus avec IODA consiste par exemple à modéliser une personne par une famille d'agents `Personne`, une armoire par une famille d'agents `Armoire`, et un rassemblement de personnes par une famille d'agents `Rassemblement`. Le transport d'une armoire est décomposé en quatre phases, résumées sur la matrice d'interaction brute de la figure 6.1.

Source \ Cible	$\emptyset$	Armoire	Rassemblement
Personne		(Créer Rassemblement, d = 1)	(Rejoindre, d = 1)
Rassemblement	(Dissoudre)	(Déplacer, d = 1)	

Figure 6.1 – Matrice d'interaction brute exprimant comment décomposer le transport d'une armoire à plusieurs personnes en interactions de cardinalités (1,1) et (1,0).

Lors de la première phase, une personne créée un rassemblement de personnes ne contenant qu'elle

même, dont l'objectif est de transporter une armoire située à proximité (interaction CRÉER RASSEMBLEMENT de la figure 6.1). Lors de la seconde phase, chaque personne souhaitant aider au transport de l'armoire rejoint le rassemblement de personnes, en initiant l'interaction REJOINDRE. Une fois que le rassemblement est suffisamment conséquent pour transporter l'armoire, le rassemblement de personnes déplace l'armoire à l'endroit souhaité en initiant l'interaction DÉPLACER. La dernière phase consiste à dissoudre le rassemblement de personnes en initiant l'interaction DISSOUDRE.

Ce modèle montre qu'il est possible de décomposer une interaction de cardinalité $(n, 1)$ en trois interactions de cardinalité (1,1), et une interaction de cardinalité (1,0). La restriction des cardinalités énoncée dans le chapitre 3 n'empêche donc pas la modélisation de phénomènes impliquant ce type d'interactions.

Modélisation de réactions chimiques

Nous illustrons la modélisation d'interactions entre une source et plusieurs cibles dans le contexte de la biochimie. Plus précisément, nous nous intéressons à la modélisation de réactions chimiques impliquant une enzyme.

Le rôle d'une enzyme dans une réaction chimique est d'accélérer la formation d'un ensemble de molécules, appelé produit, à partir d'un ensemble de molécules appelé substrat. Ces réactions chimiques prennent donc la forme générale :

$$Enzyme + Substrat \rightarrow Enzyme + Produit$$

L'interaction entre l'enzyme et le substrat a pour effet de transformer le substrat en produit. Ce phénomène peut donc être représenté dans IODA en modélisant l'enzyme par un agent, chaque molécule du substrat par un agent, et en représentant la réaction chimique sous la forme d'une interaction dont la source est l'enzyme, dont les cibles sont les molécules du substrat, et dont les actions consistent à retirer de l'environnement les molécules du substrat, et d'y ajouter les molécules du produit.

Si le substrat est formé de deux molécules, que l'on note A et B, et que le produit est un complexe, que l'on note C, alors la réaction chimique prend la forme :

$$Enzyme + A + B \rightarrow Enzyme + C$$

Les agents y sont les espèces chimiques Enzyme, A, B et C, et la réaction chimique y est une interaction ASSOCIER, dont la cardinalité est (1,2), qui consiste à retirer de l'environnement ses deux cibles, et à créer une troisième entité correspondant à l'association de ses deux cibles.

La restriction des cardinalités effectuées dans la description du modèle IODA ne permettent pas de modéliser une interaction de cardinalité (1,2). Il est toutefois possible de modéliser ce phénomène avec des interactions de cardinalité (1,1) et (1,0), et donc de s'affranchir de la limitation des cardinalités. Pour s'en convaincre, nous proposons d'étudier plus en détails le phénomène lié à cette réaction chimique. En nous basant sur cette décomposition, nous proposons un modèle permettant de décrire l'association de respectivement deux et trois molécules de substrat, en n'utilisant que des interaction permises par la restriction des cardinalités émise dans le chapitre 3.

Décomposition d'une réaction chimique contenant deux molécules de substrat Une réaction chimique a lieu entre les molécules d'un substrat lorsque ces dernières restent suffisamment longtemps dans une configuration spatiale particulière. Une enzyme est une protéine qui catalyse les réactions chimiques en favorisant le contact entre les molécules du substrat. Elle dispose pour cela d'un site actif, sur lequel les molécules du substrat peuvent venir se fixer (voir figure 6.2).

D'après ces informations, cette réaction chimique peut être décomposée en cinq réactions chimiques plus simples. En posant Enzyme-A (resp. Enzyme-B et Enzyme-A-B) une protéine contenant une enzyme sur le site actif de laquelle est fixée une molécule A (resp. B et C), ces réactions sont :

$$Enzyme + A \rightarrow Enzyme\text{-}A \tag{6.1}$$
$$Enzyme + B \rightarrow Enzyme\text{-}B \tag{6.2}$$
$$Enzyme\text{-}A + B \rightarrow Enzyme\text{-}A\text{-}B \tag{6.3}$$
$$Enzyme\text{-}B + A \rightarrow Enzyme\text{-}A\text{-}B \tag{6.4}$$
$$Enzyme\text{-}A\text{-}B \rightarrow Enzyme + C \tag{6.5}$$

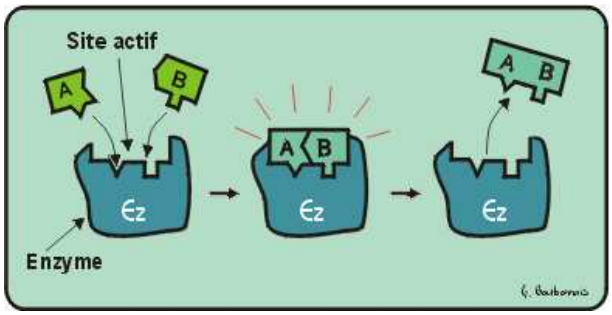


FIGURE 6.2 – Description détaillée d'une réaction chimique catalysée par une enzyme, dans laquelle le substrat est composé de deux molécules A et B, et où le produit est un complexe C formé d'une molécule A et d'une molécule B.

Les réactions 6.1 et 6.2 décrivent le fait qu'une molécule du substrat peut venir se fixer sur le site actif d'une enzyme, ce qui aura pour résultat de créer une protéine complexe contenant à la fois l'enzyme et une molécule de substrat. Les réactions 6.3 et 6.4 décrivent le fait que la seconde molécule du substrat peut venir se fixer sur l'espace restant du site actif d'une enzyme, ce qui aura pour résultat de créer une protéine complexe contenant l'enzyme et le substrat. Enfin, la réaction 6.5 décrit la décomposition de la protéine complexe en une enzyme et le produit de la réaction chimique.

Ces nouvelles réactions font apparaître six espèces chimiques, et cinq réactions. Un modèle IODA possible contient alors six familles d'agents (qui représentent chacune une espèce chimique), et de cinq interactions (qui représentent chacune une réaction). Les réactions 6.1, 6.2, 6.3 et 6.4 ne font réagir que deux espèces chimiques. Par conséquent, elles sont représentées dans IODA à l'aide d'interactions de cardinalité (1,1). De plus, la réaction 6.5 ne fait réagir qu'une seule espèce chimique. Elle est donc représentée à l'aide d'une interaction de cardinalité (1,0). La matrice d'interaction brute de la figure 6.3 résume ce que nous énonçons ici, sous l'hypothèse qu'une molécule A (resp. B) peut se fixer sur le site actif de l'enzyme que si la distance entre cette molécule et l'enzyme est inférieure ou égale à δ_A (resp. δ_B).

Source \ Cible	$\emptyset$	Enzyme	Enzyme-A	Enzyme-B
A		(RÉACTION6.1, $d = \delta_A$)		(RÉACTION6.4, $d = \delta_A$)
B		(RÉACTION6.2, $d = \delta_B$)	(RÉACTION6.3, $d = \delta_B$)	
Enzyme-A-B	(RÉACTION6.5)			

FIGURE 6.3 – Matrice d'interaction brute décrivant la décomposition de la réaction chimique $Enzyme + A + B \rightarrow Enzyme + C$ en cinq réactions chimiques plus simples.

Il est donc possible de décomposer l'interaction ASSOCIER de cardinalité (1,2) en un ensemble d'interactions de cardinalité (1,1) et (1,0). Cette décomposition peut de plus se généraliser à des réactions chimiques impliquant plus de deux molécules de substrat. Nous en donnons l'exemple avec une réaction chimique dont le substrat contient trois molécules.

Décomposition d'une réaction chimique contenant trois molécules de substrat Considérons la réaction chimique suivante, ayant pour substrat trois molécules A, B et C, et pour produit un complexe D :

$$Enzyme + A + B + C \rightarrow Enzyme + D$$

Cette réaction chimique peut être décomposée en un ensemble de réactions ne faisant réagir à chaque fois au plus deux espèces chimiques :

$$Enzyme + A \rightarrow Enzyme\text{-}A \tag{6.6}$$
$$Enzyme + B \rightarrow Enzyme\text{-}B \tag{6.7}$$
$$Enzyme + C \rightarrow Enzyme\text{-}C \tag{6.8}$$
$$Enzyme\text{-}A + B \rightarrow Enzyme\text{-}A\text{-}B \tag{6.9}$$
$$Enzyme\text{-}A + C \rightarrow Enzyme\text{-}A\text{-}C \tag{6.10}$$
$$Enzyme\text{-}B + A \rightarrow Enzyme\text{-}A\text{-}B \tag{6.11}$$
$$Enzyme\text{-}B + C \rightarrow Enzyme\text{-}B\text{-}C \tag{6.12}$$
$$Enzyme\text{-}C + A \rightarrow Enzyme\text{-}A\text{-}C \tag{6.13}$$
$$Enzyme\text{-}C + B \rightarrow Enzyme\text{-}B\text{-}C \tag{6.14}$$
$$Enzyme\text{-}A\text{-}C + B \rightarrow Enzyme\text{-}A\text{-}B\text{-}C \tag{6.15}$$
$$Enzyme\text{-}A\text{-}B + C \rightarrow Enzyme\text{-}A\text{-}B\text{-}C \tag{6.16}$$
$$Enzyme\text{-}B\text{-}C + A \rightarrow Enzyme\text{-}A\text{-}B\text{-}C \tag{6.17}$$
$$Enzyme\text{-}A\text{-}B\text{-}C \rightarrow Enzyme + D \tag{6.18}$$

En construisant le modèle IODA de cette réaction chimique selon les principes énoncés précédemment, l'interaction de cardinalité (1,3) impliquant les trois espèces chimiques A, B et C est décomposée en 11 familles d'agents, 12 interactions de cardinalité (1,1), et une interaction de cardinalité (1,0). Ce modèle est résumé sur la matrice d'interaction brute de la figure 6.4.

Source \ Cible	$\emptyset$	Enzyme	Enzyme-A	Enzyme-B
A		(Réaction6.6, $d = \delta_A$)		(Réaction6.11, $d = \delta_A$)
B		(Réaction6.7, $d = \delta_B$)	(Réaction6.9, $d = \delta_B$)	
C		(Réaction6.8, $d = \delta_C$)	(Réaction6.10, $d = \delta_C$)	(Réaction6.12, $d = \delta_C$)
Enzyme-A-B-C	(Réaction6.18)			

Source \ Cible	Enzyme-C	Enzyme-A-B	Enzyme-A-C	Enzyme-B-C
A	(Réaction6.13, $d = \delta_A$)			(Réaction6.17, $d = \delta_A$)
B	(Réaction6.14, $d = \delta_B$)		(Réaction6.16, $d = \delta_B$)	
C		(Réaction6.15, $d = \delta_C$)		
Enzyme-A-B-C				

FIGURE 6.4 – Matrice d'interaction brute décrivant la décomposition de la réaction chimique $Enzyme + A + B + C \rightarrow Enzyme + D$ en 13 réactions chimiques plus simples. Pour des raisons d'affichage, cette matrice est coupée en deux parties.

Cette décomposition est forcément très lourde. Il ne s'agit pourtant pas d'une complexification artificielle du phénomène. En effet, cette description correspond exactement à la décomposition d'une équation-bilan selon la cinétique chimique. Chaque interaction simple correspond alors à une réaction élémentaire et chaque nouvelle famille d'agents à un intermédiaire réactionnel.

Praticabilité de la décomposition d'une interaction Jusqu'à maintenant, nous avons montré qu'il était possible de décomposer une interaction de cardinalité (1,n), avec $n \geq 2$, en un ensemble d'interactions de cardinalité (1,1) ou (1,0), dans le cadre de la description de réactions chimiques. Toutefois, on peut remarquer que le nombre de familles d'agent et le nombre d'interactions obtenus par la décomposition augmente exponentiellement par rapport à n. Considérons la décomposition d'une réaction chimique catalysée par une enzyme dont le substrat contient $n \in \mathbb{N}$ molécules différentes, et dont le produit contient $m \in \mathbb{N}$ molécules. La modélisation de cette décomposition telle que présentée jusqu'à présent implique l'utilisation de $2^n + n + m - 1$ familles d'agents, de $\sum_{i \in [0,n]} (n-i) C_i^n$ interactions de cardinalité (1,1) et d'une interaction de cardinalité (1,0). Cette décomposition est donc très coûteuse en termes de complexité

du modèle, puisqu'elle y introduit un nombre non négligeable de familles d'agents et d'interactions. Le modèle en devient moins intelligible, et moins facile à modifier. Cette décomposition ne peut donc être pratiquée telle quelle.

La description générique des interactions, et son indépendance aux spécificités des agents permet de s'affranchir de ce problème. En effet, avec une modélisation appropriée, le nombre de familles d'agents peut être ramené à $m + n + 1$ (soit le même nombre de familles que la modélisation de ce phénomène avec une interaction de cardinalité $(1, n)$), et le nombre d'interactions à 2 (et donc une interaction de plus que la modélisation de ce phénomène avec une interaction de cardinalité $(1, n)$). Nous illustrons cette affirmation en appliquant la méthodologie IODA pour décrire la décomposition de la réaction chimique $Enzyme + A + B + C \rightarrow Enzyme + D$.

Soit un phénomène mettant en jeu les réactions chimiques 6.6 à 6.6. Dans ce modèle, les espèces chimiques Enzyme-A, Enzyme-B, Enzyme-C, Enzyme-A-B, Enzyme-A-C, Enzyme-B-C et Enzyme-A-B-C représentent une enzyme, sur laquelle des substrats se sont fixés. Nous pouvons les représenter par une unique famille d'agents `Enzyme`, qui contiendra un nombre variable de substrats. La première phase de la méthodologie IODA nous amène donc à identifier quatre familles d'agents, dont les identificateurs sont `Enzyme`, `A`, `B` et `C`.

Source \ Cible	∅	A	B	C	Enzyme
A					(Se Fixer Sur, $d = 0$)
B					(Se Fixer Sur, $d = 0$)
C					(Se Fixer Sur, $d = 0$)
Enzyme	(Produire D)				

Figure 6.5 – Matrice d'interaction brute modélisant une décomposition sous une forme synthétique de la réaction chimique d'équation bilan $Enzyme + A + B + C \rightarrow Enzyme + D$.

Les réactions chimiques 6.6 à 6.18 expriment le fait qu'une espèce chimique A, B ou C a la capacité de se fixer sur une enzyme. Cela mène à l'identification d'une interaction Se Fixer Sur dans la matrice d'interaction brute (voir figure 6.5). Cette interaction, dont la spécification est fournie sur la figure 6.6, consiste à ajouter la source sur le site actif de la cible (primitive *ajouterAuSiteActif*), et à retirer la source de l'environnement (puisqu'elle est maintenant associée à l'cible), seulement si le site actif de la cible peut contenir la source (primitive *peutAjouterAuSiteActif*).

« interaction individuelle » Se Fixer Sur(*Source*, *Cible*)		
Signatures	nom	identifiant de la signature
	Source	SourceDeSeFixerSur
	Cible	CibleDeSeFixerSur
	Environnement	EnvironnementDansSeFixerSur
Préconditions	Cible.peutAjouterAuSiteActif(Source)	
Déclencheur	vrai	
Actions	Cible.ajouterAuSiteActif(Source) Environnement.retirer(Source)	

Figure 6.6 – Spécification de l'interaction Se Fixer Sur, permettant aux molécules du substrat de se fixer sur le site actif d'une enzyme

La dernière réaction chimique 6.18 représente le fait qu'une enzyme sur laquelle sont fixés les trois substrats A, B et C peut produire une molécule D. Cela amène à l'identification d'une interaction Produire D dans la matrice d'interaction brute (voir figure 6.5). Cette interaction, dont la spécification est fournie sur la figure 6.7, consiste à retirer une molécule de l'espèce chimique A, une molécule de l'espèce chimique B et une molécule de l'espèce chimique C du site actif de la source (primitive *retirerDuSiteActif*), à les combiner pour produire une molécule de l'espèce chimique D (primitive *créerMoléculeD*), et à ajouter

cette nouvelle molécule dans l'environnement, à la position de l'enzyme. Cette interaction ne peut être effectuée que si le site actif de la source contient au moins une molécule des espèces chimiques A, B et C (primitive *siteActifContient*).

« interaction dégénérée » PRODUIRE D(*Source*)		
Signatures	nom	identifiant de la signature
	Source	SourceDeProduire D
	Environnement	EnvironnementDansProduire D
Préconditions	Source.siteActifContient(1, "A") **et** Source.siteActifContient(1, "B") **et** Source.siteActifContient(1, "C")	
Déclencheur	vrai	
Actions	Agent substratA = Source.retirerDuSiteActif(1, "A") Agent substratB = Source.retirerDuSiteActif(1, "B") Agent substratC = Source.retirerDuSiteActif(1, "C") Agent d = Source.créerMoléculeD(substratA, substratB, substratC) Environnement.ajouter(d, Source.abscisse(), Source.ordonnee())	

FIGURE 6.7 – Spécification de l'interaction PRODUIRE D, permettant à l'enzyme de retirer de son site actif une molécule A, une molécule B, et une molécule C, de les combiner afin de créer une molécule D, et enfin de placer cette molécule dans l'environnement.

Cette façon de modéliser permet de décomposer une interaction de cardinalité (1,3) en un ensemble réduit d'interactions de cardinalités (1,1) et (1,0). Les principes mis en place peuvent aisément être généralisés aux réactions chimiques faisant intervenir un nombre quelconque de molécules dans son substrat. Ainsi, les interactions de cardinalité (1,n), $n \geq 2$, peuvent être décrites dans IODA à l'aide d'interactions individuelles et dégénérées, n'accroissant que légèrement la complexité du modèle. Il est donc possible de modéliser des réactions chimiques avec les restrictions de cardinalités imposées dans IODA.

6.1.2 Première décomposition générique : modélisation par agrégation d'agents

Les modèles décrits dans la section précédente montrent qu'il est possible de représenter des interactions de cardinalité (1,n) et (n,1), avec $n \in \mathbb{N}$, en utilisant uniquement des interactions de cardinalité (1,1) et (1,0). Elles reposent toutes deux sur le même principe général, qui décompose l'exécution de telles interactions en deux phases. Lors d'une première phase, les $n+1$ participants à l'interaction sont agrégés en un unique agent, dont la fonction est d'initier l'interaction souhaitée. La seconde phase consiste à initier l'interaction avec l'agent agrégat en tant que source. Ce principe de modélisation peut être généralisé à tout domaine d'application, et à toute interaction de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$. Dans cette section, nous proposons une description possible d'un tel modèle. Nous fournissons uniquement une description schématique des primitives impliquées dans ces interactions, car ces dernières dépendent grandement de la sémantique de l'interaction modélisée.

Principes généraux du modèle Le principe général permettant la décomposition d'une interaction $\mathcal{I}$ de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$ en un modèle ne contenant que des interactions de cardinalités (1,1) et (1,0) consiste dans un premier temps à créer un agent A, auquel on agrègera les sources (s_i) et les cibles (c_j) de l'interaction $\mathcal{I}$. Pour cela, l'une des sources s_i initie une interaction dégénérée DÉBUT, dont la spécification est schématisée sur la figure 6.8. Cette interaction consiste à créer l'agent A, et à agréger s_i à A. Elle n'est effectuée que si s_i a l'intention d'initier l'interaction $\mathcal{I}$, et si s_i vérifie des pré-requis logiques associés à $\mathcal{I}$.

Les autres sources s'agrègent A en initiant une interaction individuelle ENTRER ayant pour cible l'agent A, dont la spécification est schématisée sur la figure 6.9. Elle n'est initiée que si sa source souhaite

« interaction dégénérée » DÉBUT(*Source*)		
Signatures	nom	identifiant de la signature
	Source	SourceDeDébut
	Environnement	EnvironnementDansDébut
Déclencheur	*% Source souhaite initier l'interaction $\mathcal{I}$*	
Préconditions	*% Source vérifie des pré-requis logiques associés à $\mathcal{I}$*	
Actions	*% Un agent agrégat A est créé* *% A est ajouté à l'environnement* *% Source est retiré de l'environnement* *% Source est agrégé à A*	

FIGURE 6.8 – Spécification schématique de l'interaction DÉBUT, permettant à un agent source d'une interaction $\mathcal{I}$ de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$ de débuter l'agrégation des agents participant à $\mathcal{I}$.

participer en tant que source à l'interaction $\mathcal{I}$, si cette source vérifie des pré-requis logiques associés à $\mathcal{I}$, et si le nombre de sources contenues dans A est strictement inférieur à n. Cette interaction assure que la participation d'un agent source s_i à l'interaction $\mathcal{I}$ est le fruit de son processus décisionnel.

« interaction individuelle » ENTRER(*Source*, *Cible*)		
Signatures	nom	identifiant de la signature
	Source	SourceDeEntrer
	Cible	CibleDeEntrer
	Environnement	EnvironnementDansEntrer
Déclencheur	*% Source souhaite initier l'interaction $\mathcal{I}$*	
Préconditions	*% Le nombre de sources contenues dans Cible est strictement inférieur à n* ***et*** *% Source vérifie des pré-requis logiques associés à $\mathcal{I}$*	
Actions	*% Source est retiré de l'environnement* *% Source est agrégé à Cible*	

FIGURE 6.9 – Spécification schématique de l'interaction ENTRER, permettant à un agent source d'une interaction $\mathcal{I}$ de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$ de s'ajouter à l'agent agrégeant toutes les sources et les cibles de l'interaction $\mathcal{I}$.

Chaque cible c_j est ajoutée à A via l'interaction individuelle CAPTER, qui est initiée par l'agent A et prend pour cible c_j, dont la spécification est schématisée sur la figure 6.10. Elle n'est initiée que si sa source A souhaite que sa cible soit l'une des cibles de l'interaction $\mathcal{I}$, si cette cible vérifie des pré-requis logiques associés à $\mathcal{I}$, et si le nombre de cibles contenues dans A est strictement inférieur à m. Cette interaction assure que la participation d'un agent cible c_j à l'interaction $\mathcal{I}$ est le fruit du processus décisionnel de l'ensemble des sources (s_i).

La seconde phase de ce processus commence une fois l'agrégation terminée. Elle consiste dans un premier temps à effectuer les actions de l'interaction $\mathcal{I}$, en initiant l'interaction dégénérée INITIER avec A pour source dont la spécification est schématisée sur la figure 6.11. Cette interaction n'est initiée que si les agents sources et cibles agrégés à A sont en nombre suffisant, et vérifient le déclencheur, et les préconditions de $\mathcal{I}$.

Une fois ces actions effectuées, l'agent A désagrège tous les agents qu'il contient, et les replace dans l'environnement en initiant l'interaction dégénérée FIN. Cette interaction a aussi pour effet de retirer A de l'environnement, et n'est effectuée que si l'interaction $\mathcal{I}$ a bien eu lieu. La spécification de cette interaction est schématisée sur la figure 6.12.

Si $\mathcal{F}$ est la famille de l'agent A, S_i est la famille de l'agent source s_i, et C_j est la famille de l'agent source c_j, alors ce principe se traduit par la matrice d'interaction de la figure 6.13.

« interaction individuelle » CAPTER($Source, Cible$)		
Signatures	nom	identifiant de la signature
	Source	SourceDeCapter
	Cible	CibleDeCapter
	Environnement	EnvironnementDansCapter
Déclencheur	*% Source souhaite que Cible soit l'une des cibles de l'interaction $\mathcal{I}$*	
Préconditions	*% Le nombre de cibles contenues dans Source est strictement inférieur à m* ***et*** *% Cible vérifie des pré-requis logiques associés à $\mathcal{I}$*	
Actions	*% Cible est retiré de l'environnement* *% Cible est agrégé à Source*	

FIGURE 6.10 – Spécification schématique de l'interaction CAPTER, permettant aux agents sources d'une interaction $\mathcal{I}$ de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$ d'ajouter un agent cible à l'agent agrégeant toutes les sources et les cibles de l'interaction $\mathcal{I}$.

« interaction dégénérée » INITIER($Source$)		
Signatures	nom	identifiant de la signature
	Source	SourceDeInitier
	Environnement	EnvironnementDansInitier
Déclencheur	*% Le déclencheur de $\mathcal{I}$ est vrai pour les agents sources et cibles contenus dans Source*	
Préconditions	*% Source agrège n agents sources et m agents cibles* ***et*** *% Les préconditions de $\mathcal{I}$ sont vraies pour les agents sources et cibles contenus dans Source*	
Actions	*% Les actions de $\mathcal{I}$ sont effectuées pour les agents sources et cibles contenus dans Source* *% On spécifie dans Source que l'interaction $\mathcal{I}$ a été effectuée*	

FIGURE 6.11 – Spécification schématique de l'interaction INITIER, permettant aux agents sources et cibles d'une interaction $\mathcal{I}$ de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$ d'initier l'interaction $\mathcal{I}$.

« interaction dégénérée » FIN($Source$)		
Signatures	nom	identifiant de la signature
	Source	SourceDeFin
	Environnement	EnvironnementDansFin
Déclencheur	vrai	
Préconditions	*% Il est spécifié dans Source que l'interaction $\mathcal{I}$ a été effectuée*	
Actions	*% Ajouter dans l'environnement tous les agents agrégés à Source* *% Retirer Source de l'environnement*	

FIGURE 6.12 – Spécification schématique de l'interaction FIN, permettant de replacer dans l'environnement les agents sources et cibles ayant participé à une interaction $\mathcal{I}$ de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$.

Source \ Cible	$\emptyset$	$\mathcal{F}$	C_1	...	C_n
$\mathcal{F}$	(INITIER, p=1) (FIN, p=0)		(CAPTER, $d = d_2$, p=1)	...	(CAPTER, $d = d_2$, p=1)
S_1	(DEBUT, p=0)	(ENTRER, $d = d_1$, p=1)		...	
⋮	⋮				
S_n	(DEBUT, p=0)	(ENTRER, $d = d_1$, p=1)		...	

FIGURE 6.13 – Matrice d'interaction raffinée décrivant comment modéliser une interaction de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$ en utilisant uniquement des interactions de cardinalité (1,1) et (1,0).

Simplifications ponctuelles du modèle générique Le modèle générique présenté ici est valable pour toute interaction $\mathcal{I}$ de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$, et permet de modéliser ces interactions à l'aide d'au maximum trois interactions de cardinalité $(1, 1)$, de deux interactions de cardinalité (1,0), et d'une famille d'agents supplémentaire. Comme tout modèle générique, il peut être simplifié dans certains cas particuliers.

Premièrement, l'une des sources de l'interaction $\mathcal{I}$ peut être utilisée en tant qu'agent agrégat. C'est par exemple le cas de la famille d'agents `Enzyme` qui agrège le substrat d'une réaction chimique. Cela évite de créer une nouvelle famille d'agents pour agréger les sources et les cibles de $\mathcal{I}$.

Secondement, la distinction entre les interactions Initier et Fin est faite pour marquer la différence entre la sémantique de l'interaction $\mathcal{I}$, qui est décrite dans l'interaction Initier, et la désagrégation des agents ayant participé à l'interaction $\mathcal{I}$, qui est décrite dans l'interaction Fin. Dans certaines interactions, la désagrégation fait partie de la sémantique de l'interaction. C'est par exemple le cas de l'interaction Produire A-B-C de la décomposition d'une réaction chimique, puisque les agents libérés dans l'environnement à la fin de l'interaction diffèrent des agents présents au début de l'interaction. Dans de tels cas, la distinction entre Initier et Fin n'est pas souhaitable. Une seule interaction est donc utilisée pour modéliser ces dernières.

Enfin, si jamais l'interaction $\mathcal{I}$ n'implique qu'une seule cible, alors il est possible de se passer de l'interaction Capter en considérant que l'interaction Initier est de cardinalité (1,1) et prend pour cible la cible de $\mathcal{I}$. C'est par exemple le cas de l'interaction Déplacer un objet lourd à plusieurs.

6.1.3 Problème d'actions simultanées

Dans cette troisième section, nous étudions sur deux exemples une autre façon de modéliser des interactions de cardinalité $(1, n)$, où $n \in \mathbb{N}$, avec les restrictions de cardinalité de IODA. Dans un premier temps, nous nous intéressons à la modélisation de la propagation d'une maladie contagieuse. Dans un second temps, nous étudions comment modéliser une explosion ayant un effet sur tous les agents situés dans le périmètre de l'explosion. Ces deux études nous amènent à identifier un second modèle générique permettant de décomposer certaines interactions en un ensemble d'interactions de cardinalité (1,1) et (1,0), sans utiliser d'agent agrégateur.

Modélisation de la propagation d'une maladie contagieuse

Considérons une simulation visant à reproduire la propagation d'une épidémie dans une population. Dans cette simulation, une personne peut être saine ou malade. Une personne malade contamine par contact toute personne saine affaiblie (par exemple parce qu'elle souffre d'un syndrome immuno-déficience, parce qu'elle vient d'être opérée ou encore parce qu'elle a des plaies ouvertes).

La modélisation d'un tel phénomène se fait dans IODA à l'aide d'une famille d'agents `Personne`. Une `Personne` malade contamine toutes les personnes affaiblies situées à proximité. Par conséquent, l'interaction Propager contagion prend pour cibles un nombre variable d'agents, qui dépend de la position dans l'environnement du malade et des personnes saines. Dans le modèle IODA présenté dans le chapitre 3, la notion de cardinalité permet de caractériser uniquement un nombre fixe de cibles. Elle n'est donc pas appropriée pour modéliser l'interaction Contaminer telle quelle. Il est toutefois possible de représenter un tel phénomène à l'aide d'interactions de cardinalité (1,1).

Dans le phénomène décrit ici, contaminer n cibles revient à effectuer successivement n fois la contamination d'une cible. On peut donc représenter la contamination par une interaction de cardinalité (1,1) que nous appelons Contaminer, qui sera répétée autant de fois qu'il y a de cibles possibles (voir figure 6.14).

Modélisation d'une explosion

Considérons une simulation visant à reproduire le comportement de soldats devant traverser un champ de mines. Dans cette simulation, une mine est un explosif se déclenchant lorsque qu'un soldat marche dessus, c'est à dire lorsque la distance entre le soldat et la mine est de 0. Lorsque l'explosion est déclenchée, la déflagration qu'elle engendre blesse tous les soldats situés dans un certain périmètre (par exemple un disque de rayon 3 mètres), et a pour conséquence la destruction de la mine.

Source \ Cible	∅	Personne
Personne		(CONTAMINER, d = 0)

« interaction individuelle » CONTAMINER(Source,Cible)		
Signatures	nom	identifiant de la signature
	Source	SourceDeContaminer
	Cible	CibleDeContaminer
	Environnement	EnvironnementDansContaminer
déclencheur	vrai	
préconditions	Source.estMalade() **et** Cible.estAffaiblie()	
actions	Cible.devenirMalade()	

FIGURE 6.14 – Matrice d'interaction brute (en haut) permettant de modéliser la propagation d'une infection à l'aide de l'interaction individuelle CONTAMINER (en bas), qui est effectuée par un agent source malade afin de contaminer un agent cible affaibli.

La modélisation d'un tel phénomène se fait dans IODA à l'aide de deux familles d'agents `Mine` et `Soldat`. Un `Soldat` se déplace dans l'environnement, et peut déclencher l'explosion d'une `Mine` en marchant dessus. Une interaction DÉCLENCHER de cardinalité (1,1) est donc ajoutée au modèle, afin de modéliser l'évènement déclencheur d'une explosion. La déflagration d'une explosion affecte tous les soldats situés à proximité de la mine. **Par conséquent, l'interaction EXPLOSER prend pour cible un nombre variable d'agents**, qui dépend de la position de la `Mine`, ainsi que de la position des `soldats` dans l'environnement. Dans le modèle IODA présenté dans le chapitre 3, la notion de cardinalité permet uniquement de caractériser un nombre fixe de cibles. Elle n'est donc pas appropriée pour modéliser l'interaction EXPLOSER telle quelle. Il est toutefois possible de représenter un tel phénomène à l'aide d'interactions de cardinalité (1,1) et (1,0). **Néanmoins, contrairement à l'exemple précédent, une explosion affectant n cibles n'est pas équivalent à n explosions successives affectant à chaque fois une cible**. En effet, une explosion a un effet de bord sur la `Mine`, qui consiste à retirer la mine de l'environnement.

Dans le phénomène décrit ici, l'explosion d'une mine se décompose en trois phases : le déclenchement de l'explosion par un soldat marchant sur la mine, la déflagration de la mine sur les soldats situés à moins de trois mètres, et la disparition de la mine dont la déflagration sur les soldats a eu lieu. Il est donc possible de modéliser ce phénomène à l'aide de trois interactions résumées sous la forme de la matrice d'interaction raffinée de la figure 6.15.

Source \ Cible	∅	Mine	Soldat
Mine	(DISPARAÎTRE, p=0)		(EXPLOSER SUR, d = 3, p=1)
Soldat	(SE DÉPLACER, p=0)	(DÉCLENCHER, d = 0, p=1)	

FIGURE 6.15 – Matrice d'interaction brute décrivant l'explosion d'une mine et son effet sur les soldats situés à proximité, à l'aide d'interactions de cardinalité (1,1) et (1,0).

La première interaction, que nous nommons DÉCLENCHER, a pour cardinalité (1,1), et pour fonction de notifier à la mine que son explosion doit avoir lieu. Cette interaction a pour source un `Soldat`, a pour cible une `Mine`, et se déclenche seulement si la distance entre le `Soldat` et la `Mine` est de 0. Ses actions consistent à spécifier que la `Mine` a été activée, et qu'elle doit donc exploser (voir figure 6.16).

La seconde interaction, nommée EXPLOSER SUR, a pour cardinalité (1,1), et pour fonction de blesser un à un chaque `Soldat` situé à moins de trois mètres de la mine. Il s'agit d'une interaction ayant pour source une `Mine`, pour cible un `Soldat`, et qui se déclenche seulement si la mine a été activée, et si la déflagration de cette mine n'a pas déjà touché le soldat. Ses actions consistent à diminuer la santé du

« interaction individuelle » DÉCLENCHER(Source, Cible)		
Signatures	nom	identifiant de la signature
	Source	SourceDeDéclencher
	Cible	CibleDeDéclencher
	Environnement	EnvironnementDansDéclencher
déclencheur	vrai	
préconditions	vrai	
actions	Cible.activer()	

FIGURE 6.16 – Spécification de l'interaction DÉCLENCHER.

`Soldat` (voir figure 6.17).

« interaction individuelle » EXPLOSER SUR(Source, Cible)		
Signatures	nom	identifiant de la signature
	Source	SourceDeExploserSur
	Cible	CibleDeExploserSur
	Environnement	EnvironnementDansExploserSur
déclencheur	not Source.aDejaBlessé(Cible)	
préconditions	Source.estActivée()	
actions	Cible.diminuerSanté(Source.puissance())	

FIGURE 6.17 – Spécification de l'interaction EXPLOSER SUR.

Enfin, la dernière interaction, nommée DISPARAÎTRE, a pour cardinalité (1,0), et pour fonction de détruire la `Mine` ayant explosé. Il s'agit d'une interaction ayant pour source une `Mine`, et qui se déclenche seulement si la `Mine` a été activée, et si tous les `Soldats` qui devaient être les victimes de la `Mine` ont subi l'interaction BLESSER. Ses actions consistent à retirer la `Mine` de l'environnement (voir figure 6.18). La condition relative au fait que tous les soldats doivent avoir été blessés est acquise à l'aide d'une priorité plus forte attribuée à l'interaction EXPLOSER SUR.

« interaction dégénérée » DISPARAÎTRE(Source)		
Signatures	nom	identifiant de la signature
	Source	SourceDeDisparaître
	Environnement	EnvironnementDansDisparaître
déclencheur	vrai	
préconditions	Source.estActivée()	
actions	Environnement.retirer(Source)	

FIGURE 6.18 – Spécification de l'interaction DISPARAÎTRE.

Ainsi, bien que le formalisme de IODA décrit dans le chapitre 3 ne permette pas de modéliser une explosion par une interaction ayant plusieurs cibles, il est possible de modéliser ce phénomène en initiant une interaction de cardinalité (1,1) autant de fois qu'il n'y a de cibles. Il faut toutefois noter que, contrairement à la propagation d'une contagion, l'explosion a un effet de bord sur la source de l'interaction (la disparition de l'agent source de l'environnement). Pour modéliser cet effet de bord, il se révèle nécessaire d'ajouter une interaction supplémentaire au modèle. Cette décomposition reste toutefois une solution naïve, sujette à des problèmes temporels exposés dans la section 6.1.5.

6.1.4 Seconde décomposition générique : modélisation par répétition d'interactions

Les modèles décrits dans la section précédente montrent qu'il est possible de représenter certaines interactions impliquant une source et $n \in \mathbb{N}$ cibles en utilisant une unique interaction de cardinalité (1,1), qui est répétée n fois. Toutefois, ce principe de modélisation ne peut être appliqué que si l'effet de l'interaction sur chaque cible est identique, si cette interaction n'a pas d'effet de bord sur la source de l'interaction, et si ses conditions ou ses actions ne dépendent pas du nombre de ses cibles. Si l'effet de l'interaction diffère selon ses cibles, ou si ses conditions ou ses actions dépendent du nombre de ses cibles, il est alors impossible d'appliquer le principe de modélisation que nous exposons dans cette section. Il reste toutefois possible de représenter cette interaction avec le principe décrit dans la section 6.1.2.

Principes généraux du modèle Le principe général permettant la décomposition de certaines interaction $\mathcal{I}$ de cardinalité $(1, n) \in \mathbb{N}$ en interactions de cardinalités (1,1) et (1,0) consiste à diviser l'interaction $\mathcal{I}$ en une interaction Affecter Cible de cardinalité (1,1), et une interaction Affecter Source, de cardinalité (1,0). L'interaction Affecter Cible est une interaction dont les actions appliquent l'effet de $\mathcal{I}$ sur une de ses n cibles. Cette interaction n'est effectuée que si sa cible est l'une des n cibles de l'interaction $\mathcal{I}$. La spécification de cette première interaction est telle que schématisée sur la figure 6.19.

« interaction individuelle » Affecter Cible(Source, Cible)		
Signatures	nom	identifiant de la signature
	Source	SourceDeAffecterCible
	Cible	CibleDeAffecterCible
	Environnement	EnvironnementDansAffecterCible
déclencheur	*% Vérifier que Cible est l'une des n cibles de $\mathcal{I}$*	
préconditions	*% Vérifier que l'effet de $\mathcal{I}$ n'a pas déjà été appliqué sur Cible*	
actions	*% Appliquer les effet de $\mathcal{I}$ uniquement sur Cible*	

Figure 6.19 – Description schématique de l'interaction Affecter Cible, qui permet d'appliquer les effet d'une interaction $\mathcal{I}$ de cardinalité (1,n), avec $n \in \mathbb{N}$, uniquement sur l'une de ses cibles.

L'interaction Affecter Source joue une fonction similaire à Affecter Cible, en appliquant toutefois les effets de l'interaction $\mathcal{I}$ à sa source, si ces effets ne lui ont pas déjà été appliqués. La spécification de cette seconde interaction est telle que schématisée sur la figure 6.20.

« interaction dégénérée » Affecter Source(Source)		
Signatures	nom	identifiant de la signature
	Source	SourceDeAffecterSource
	Environnement	EnvironnementDansAffecterSource
déclencheur	vrai	
préconditions	*% Vérifier que l'effet de $\mathcal{I}$ n'a pas déjà été appliqué sur Source*	
actions	*% Appliquer les effet de $\mathcal{I}$ uniquement sur Source*	

Figure 6.20 – Description schématique de l'interaction Affecter Source, qui permet d'appliquer les effet de $\mathcal{I}$ uniquement sur sa source, et avant d'avoir appliqué un quelconque effet sur ses cibles.

Afin de ne pas biaiser la modification des n cibles de l'interaction $\mathcal{I}$, la modification de sa source ne peut avoir lieu qu'avant ou après l'exécution des interactions Affecter Cible. Cela aboutit donc à deux modèles possibles, dont la description diffère dans l'ordre dans lequel exécuter ces interactions. Dans un cas, l'interaction Affecter Source n'est initiée que si l'interaction Affecter Cible ne peut plus être initiée. Dans l'autre, les interactions Affecter Cible ne sont initiées que si l'interaction Affecter Source a été préalablement initiée.

6.1.5 Limites des interactions de cardinalités (1,1) et (1,0)

Dans cette section, nous avons décrit deux modèles génériques permettant de décomposer des interactions de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$ en un ensemble d'interactions de cardinalité (1,1) et (1,0). Le premier repose sur l'agrégation des $n + m$ agents participant à l'interaction en un seul agent, et permet de décomposer toute interaction. Le second repose sur le fait que certaines interactions de cardinalité (1,n) peuvent être dans certains cas (qui dépendent de la sémantique de l'interaction) n répétitions d'une interaction de cardinalité (1,1). Puisque le premier modèle permet de décomposer toute interaction, la restriction aux interactions individuelles et dégénérées effectuée dans le chapitre 3 ne limite pas l'ensemble des simulations pouvant être spécifiées avec IODA. Cette décomposition n'implique l'ajout que d'un nombre restreint de familles d'agents et d'interactions, et ne complexifie donc que légèrement la spécification des interactions, des familles d'agents et des matrices d'interaction.

Dans certains cas, cette décomposition pose toutefois des problèmes du point de vue de la représentation du temps, et de l'implémentation de l'algorithme d'ordonnancement de l'activité des agents.

Nature du problème rencontré

Là où une interaction de cardinalité (n,m) nécessite l'initiation d'une seule interaction, la décomposition de cette interaction nécessite l'initiation de plusieurs interactions. Si l'on se place dans le cadre des simulations en temps discret dont le modèle est décrit dans le chapitre 4, cela signifie que l'interaction n'est plus effectuée en un seul pas de temps, mais en plusieurs pas temps, dont le nombre dépend de la cardinalité de l'interaction, et de la façon utilisée pour décomposer l'interaction.

La première décomposition effectue une interaction de cardinalité (n, m) en au mieux $3 + m$ pas de temps : un pas de temps lors duquel toutes les interactions Début et Entrer sont initiées, m pas de temps lors desquels l'agent agrégat initie l'interaction Capter avec chacune des m cibles, un pas de temps lors duquel l'agent agrégat initie l'interaction Initier, et enfin un pas de temps lors duquel l'agent agrégat initie l'interaction Fin. La seconde décomposition effectue une interaction de cardinalité $(1, n)$ en $n + 1$ pas de temps : n pas de temps lors desquels l'agent source effectue l'interaction Affecter Cible avec chacune de ses n cibles, et un pas de temps lors duquel l'agent source effectue l'interaction Affecter Source.

Cet étalement dans le temps peut poser problème si les agents sont capables d'initier ou subir d'autres interactions pendant cette période. En effet, dans certains cas, des agents qui auraient dû être la source ou la cible de l'interaction peuvent ne pas l'être, et inversement, des agents qui n'auraient pas du être la source ou la cible de l'interaction peuvent le devenir. Ce fait peut aboutir à des résultats de simulation faussés.

Par exemple, dans le cas du modèle décrit dans la section 6.1.3, des soldats peuvent se déplacer et déclencher une mine, et une mine qui a été déclenchée peut exploser et tuer les soldats victimes de sa déflagration. Dans cet exemple, pendant que la mine initie l'interaction Exploser sur sur un `Soldat`, les autres `Soldat`s peuvent se déplacer, et échapper à la déflagration (ceci est illustré sur la figure 6.21). S'ensuit alors un biais : un `Soldat` ayant déclenché la `Mine` peut s'enfuir pendant que la `Mine` est occupée à Exploser sur les autres `Soldat`s à proximité.

Toutefois, l'utilisation des décompositions proposées ici n'aboutissent pas systématiquement à des biais. Par exemple, dans le cas de la biochimie, cette représentation du temps est tout à fait appropriée pour modéliser la décomposition d'une réaction chimique, car les différentes réaction faisant partie de cette décomposition ne sont pas supposées survenir en parallèle, contrairement aux interactions Exploser sur. Le problème mentionné ici concerne le cas restreint des interactions ayant nécessairement un effet simultané sur les agents qu'elle implique.

Solutions possibles

Plusieurs approches permettent de résoudre ce problème, mais nécessitent pour la plupart de prendre en compte une représentation plus complexe du temps. Une première approche consiste à permettre aux agent d'initier plusieurs interactions par pas de temps. Bien qu'elle permette de résoudre le problème mentionné ici, cette solution nécessite de fournir un modèle spécifiant de manière générique quelles interactions peuvent être initiées simultanément par un agent, intégrer ces spécifications au modèle de sélection

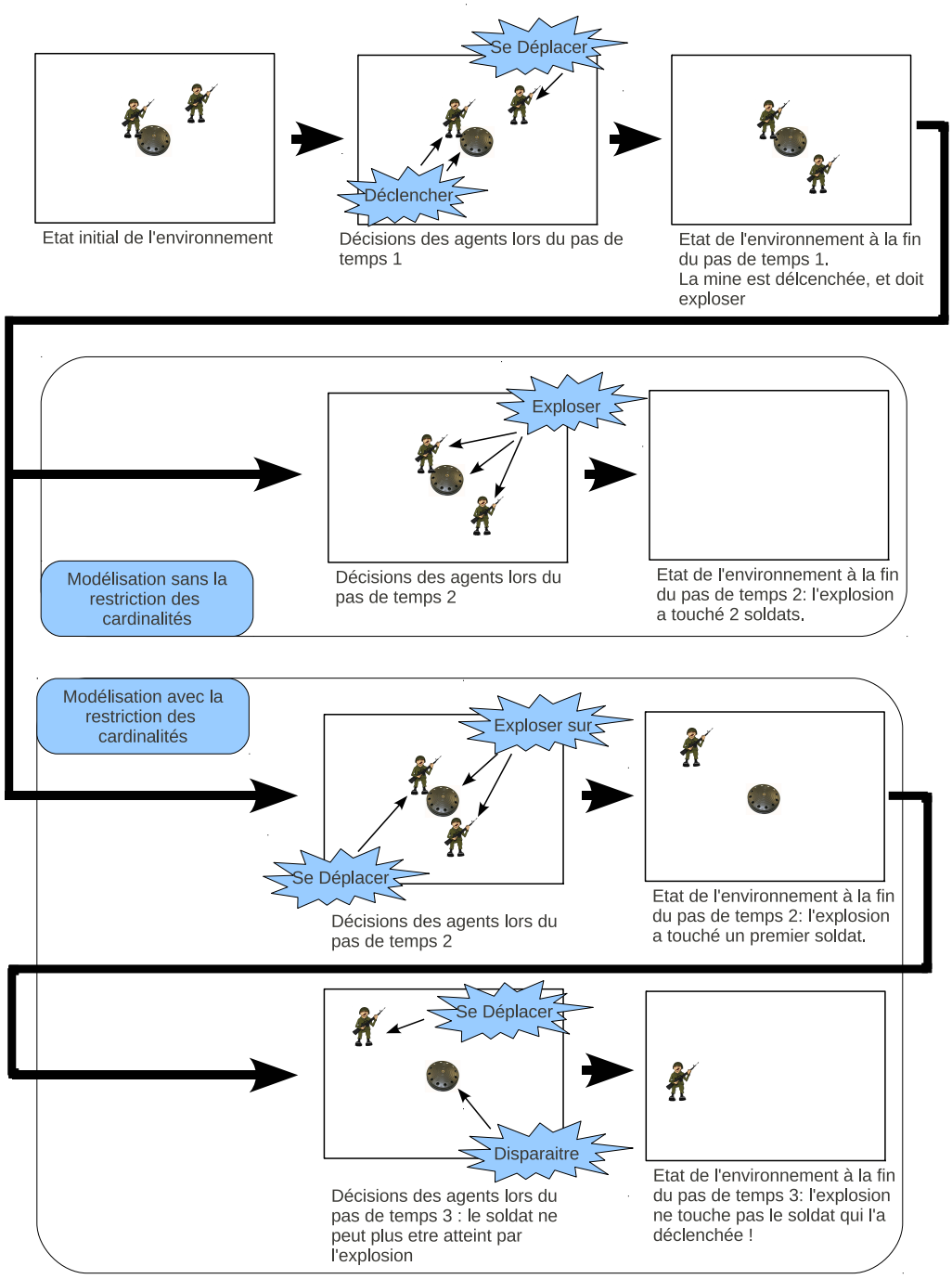


FIGURE 6.21 – Illustration du problème de représentation du temps lié à la décomposition d'une interaction en un ensemble d'interactions de cardinalité (1,1) et (1,0). Cette illustration s'appuie sur le modèle décrit dans la section 6.1.3, dans lequel des soldats peuvent déclencher l'explosion d'une mine. Pour simplifier la compréhension de ce schéma, nous supposons que les actions des interactions EXPLOSER et EXPLOSER SUR ont pour effet de retirer les soldats blessés de l'environnement.

d'interaction, décrire les algorithmes lui étant associés, et permettre de telles spécifications dans la méthodologie de conception. Ainsi, pour résoudre un problème propre à un nombre restreint d'interactions (*i.e.* les interaction qui doivent être exécutées en parallèle), l'ensemble du modèle et de la méthodologie doit être complexifié. Cette solution n'est donc pas souhaitable.

Une seconde solution consiste à utiliser une représentation du temps très fine, où une épaisseur temporelle (*i.e.* une durée) serait attribuée à chaque interaction. Attribuer une épaisseur temporelle presque nulle à une interaction permettrait alors d'obtenir une sorte de parallélisme. Toutefois, tout comme la solution mentionnée précédemment, cette solution nécessite de complexifier la structure du modèle existant, et est donc à éviter.

Une troisième approche consiste à construire le simulateur de sorte qu'il puisse détecter l'initiation d'une interaction située en amont de la décomposition (par exemple l'interaction Début) lors d'un pas de temps. Lorsque cette situation est détectée, le simulateur passe dans un état transitoire avant l'exécution du prochain pas de temps. Dans cette situation transitoire, les agents ne peuvent initier que les interactions faisant partie de la décomposition (dans l'exemple donné ici, il s'agit des interactions Capter, Entrer, Initier et Fin). Cette situation transitoire prend alors fin lorsque la dernière interaction de la décomposition est effectuée (ici, l'interaction Fin). Cette approche requiert l'annotation des interaction, afin de déterminer quelles sont celles faisant partie d'une décomposition, et fait perdre leur généricité aux algorithmes utilisés. Elle est donc aussi à éviter.

Une autre solution pourrait être de s'assurer que le modèle ne spécifie que des interactions survenant à la même échelle temporelle, comme c'est par exemple le cas dans la décomposition d'une réaction chimique. Cette solution a l'avantage de ne pas nécessiter d'intervention au niveau de la structure du modèle, et porte seulement sur son sens. Toutefois, cette solution ne permet pas de modéliser les interactions devant survenir en parallèle (par exemple Exploser sur).

Enfin, une dernière solution serait de prendre en compte les cardinalités plus complexes dans IODA.

Synthèse et discussion

En résumé, les patrons de conception proposés dans cette section peuvent décomposer la plupart des interactions de cardinalité $(n, m) \in \mathbb{N}^* \times \mathbb{N}$ en n'utilisant que des interactions individuelles et dégénérées. Toutefois, certaines interactions de cardinalité (n, m) ont nécessairement un effet simultané sur toutes leurs cibles et ne peuvent donc pas être décomposées. Différentes solutions permettent de remédier à ce problème, mais nécessitent pour la plupart la complexification de l'ensemble du modèle. S'ensuit alors un paradoxe : l'ensemble du modèle est complexifié pour modéliser une minorité d'interactions.

Afin de concilier la possibilité de modéliser de telles interactions sans pour autant complexifier la spécification des autres interactions, nous définissons une extension de Ioda où :

- le modèle formel permet de modéliser un troisième type d'interactions appelées interactions multicast ;
- le modèle des interactions individuelles et des interactions dégénérées ne sont pas modifiés ;
- la seule modification de Ioda consiste à identifier faire le lien entre tuple réalisable et interaction multicast.

L'ajout des interactions multicast n'induit que peu de modifications dans Ioda. De plus, comme il s'agit d'une extension méthodologique, les modèles ne contenant que des interactions pouvant être décomposées peuvent toujours être modélisés simplement à l'aide du c ?ur de l'approche Ioda.

6.2 Extension de IODA aux interactions de type multicast

Certaines interactions initiées par une seule source peuvent avoir un nombre de cible dynamique dépendant du voisinage de l'agent source. Par exemple :

- l'interaction Exploser qui prend pour source un engin explosif et pour cible tous les agents se situant à distance de déflagration de cette source ;
- l'interaction Infecter qui prend pour source une personne malade et pour cible tous les agents sains se trouvant à son contact ;

– l'interaction ALERTER qui prend pour source un animal ayant perçu un danger ou une alarme anti-incendie et pour cible d'agents alertés par la source.

Ces interactions ont un effet nécessairement simultané sur toutes les cibles. Ces interactions ne peuvent être décomposées en utilisant nos patrons de conception si la représentation du temps ne permet d'initier qu'une seule interaction par pas de temps. En effet, pendant que cette interaction individuelle est initiée sur une cible particulière, les autres cibles de l'interaction ont l'opportunité d'initier ou subir d'autres interactions pouvant modifier leur état. Cela aboutit à des situations où certaines cibles échappent à l'interaction. La figure 6.21 illustre ce point dans le cas d'un `Soldat` pouvant échapper à la déflagration d'une `mine` qu'il a pourtant DÉCLENCHÉE.

La spécification de telles interactions induit alors un choix :

– soit complexifier la représentation du temps et permettre à un agent d'initier plus d'une interaction à la fois ;
– soit complexifier les cardinalités supportées par IODA, et ajouter un type d'interaction pour couvrir ce cas problématique en plus des interactions individuelles et dégénérées.

Il n'est pas envisageable de complexifier la représentation du temps, puisque cela reviendrait à complexifier l'ensemble du modèle pour prendre en compte des situations très minoritaires. Nous proposons donc d'étendre le modèle ainsi que la méthodologie et les algorithmes présentés dans le chapitre 4 aux **interactions ayant un nombre dynamique de cibles, que nous appelons *interactions de multicast*.**

6.2.1 Extension du modèle des interactions

La différence principale entre les interactions individuelles et dégénérées d'une part, et les interactions de multicast d'autre part, tient dans la façon d'exprimer le nombre de cibles de l'interaction. Alors que dans le premier cas, le nombre de cibles est fixe, et égal à 0 ou 1, le nombre de cibles d'une interaction de multicast dépend agents présents dans le voisinage de sa source. Par conséquent, la définition de telles interactions est complétée par un critère permettant de déterminer quelles sont les cibles de l'interaction. Nous appelons *Critère d'acceptation d'une cible* un tel critère. Il prend pour paramètres l'agent source et un agent du voisinage de l'agent source, et retourne une valeur booléenne, vraie si l'agent est une cible de l'interaction et fausse sinon.

Définition 33. ***Interaction de multicast***
Une **interaction de multicast** est une interaction ayant une seule source et un nombre dynamique de cibles, qui dépend du voisinage de l'agent source. Ses cibles sont identifiées parmi le voisinage de l'agent source à l'aide d'un **critère d'acceptation d'une cible**, qui retourne vrai si l'agent est une cible de l'interaction.
Le critère d'acceptation de ces interactions prend la forme :

$$\begin{array}{l} accepterCible : \\ \quad \mathbb{A} \times \mathbb{A} \quad \rightarrow \quad \{vrai, faux\} \\ (source, agent) \rightarrow \left\{ \begin{array}{ll} faux & \text{si } source \text{ n'accepte pas } agent \text{ comme cible de l'interaction} \\ vrai & \text{sinon} \end{array} \right. \end{array}$$

On note $\mathbb{I}_{(1,*)}$ l'ensemble des interactions de multicast définies pour une simulation.

Dans la suite de cette section, nous décrivons comment compléter le modèle de IODA pour prendre en compte de telles interactions.

Caractérisation du modèle formel

Les principes généraux énoncés jusqu'à ne suffisent pas à la spécification précise d'une interaction de multicast. En effet, dans le modèle formel de IODA, nous avons caractérisé une interaction I comme un 9-uplet contenant un identifiant (id), un ensemble de noms attribués aux agents impliqués dans l'interaction ($noms$), une fonction déterminant si l'agent ayant un nom particulier est une source ou une cible de l'interaction ($nature$), une cardinalité $card = (card_S(I), card_T(I)) \in \mathbb{N}^* \times \mathbb{N}$, une fonction associant une signature à chaque nom d'agent dans l'interaction ($signatures$), une signature de l'environnement

dans l'interaction ($sign_{env}$), et enfin des préconditions, déclencheur et actions. Nous proposons ici de caractériser la valeur de ces divers éléments pour les interactions de multicast. Cette phase est nécessaire à la construction d'une représentation graphique permettant de spécifier ces interactions.

Caractérisation de "noms" : Pour spécifier de telles interaction, le premier problème qui se pose est de pouvoir trouver le nom de chaque agent participant à l'interaction de multicast, afin d'en décrire les conditions et les effets. Pour cela, nous utilisons deux conventions de notation. Premièrement, la source d'une telle interaction doit avoir pour nom "*source*". Ensuite, puisque le nombre de cibles d'une interaction de multicast peut varier de 0 à une valeur n quelconque, le nom des cibles est la concaténation de la chaîne "*cibles[i]*" où i est un nombre allant de 1 à n, $n \in \mathbb{N}$ étant le nombre de cibles participant à l'interaction.

Caractérisation de "nature" : La spécification de la fonction *nature* est triviale, et se base sur l'analyse du nom des entités. La source est l'agent dont le nom dans l'interaction est "*source*". Les agents ayant pour nom "*cibles[i]*" sont les cibles de l'interaction.

Caractérisation de "actions" : Comme nous l'avons mentionné dans le second modèle générique que nous avons présenté, les actions d'une telle interaction se décomposent :

- d'une part en un effet appliqué successivement à chaque cible ;
- d'autre part en un effet appliqué à la source soit avant, soit après la modification des cibles.

Afin de garder un maximum de liberté dans l'expression de ces interactions, nous permettons les deux cas. Pour cela, la description des actions est libre, et nous introduisons un mot clé particulier nommé *nombreCibles* propre à ces interactions, permettant de connaître le nombre de cibles de l'interaction. Ces informations sont suffisantes pour décrire les actions de telles interactions.

***Exemple. Description des actions de l'interaction* CONTAMINER**

Dans le contexte d'une simulation visant à modéliser la propagation d'une infection, l'interaction de multicast CONTAMINER *peut avoir la spécification qui suit :*

« interaction de multicast » CONTAMINER*(source,cibles,nombreCibles)*	
...	
Actions	*Pour tout i allant de 1 à nombreCibles, faire :* *Environnement.retirer(cibles[i])* *Fin pour*

***Exemple. Description des actions de l'interaction* EXPLOSER**

Dans le contexte d'une simulation visant à modéliser le déplacement de soldats dans un champ de mines, l'interaction de multicast EXPLOSER *peut avoir la spécification qui suit :*

« interaction de multicast » EXPLOSER*(source,cibles,nombreCibles)*	
...	
Actions	*Pour tout i allant de 1 à nombreCibles, faire :* *Environnement.blesser(cibles[i])* *Fin pour* *Environnement.retirer(source)*

Caractérisation de "signatures" et de "identifiant" : Dans les interactions de multicast, les cibles ont toutes un rôle réflexif, puisqu'elles subissent toutes le même effet en parallèle. Par conséquent, seules deux signatures d'entités sont définies : une pour leur source, et l'autre pour leurs cibles. Nous utilisons les mêmes notations que les interactions individuelles pour les désigner : la signature de la source de l'interaction est notée $sign_{source}$, et celle de ces cibles est notée $sign_{cibles}$. La signature de l'environnement dans de telles interactions, ainsi que l'identifiant de ces interactions, s'exprime exactement comme dans les interactions individuelles et dégérénées.

Caractérisation de "card" : Le nombre de cibles de l'interaction n'est pas fixe, et ne peut donc être une valeur dans $\mathbb{N}$. Nous prenons pour convention de noter la cardinalité de telles interactions $(1, *)$, où $*$ représente un nombre dynamique d'agents. Afin de déterminer quels sont les agents participant à l'interaction lors de l'exécution de la simulation, nous ajoutons à la spécification de telles interactions un critère d'acceptation *accepterCible*. Ce critère prend en paramètre un agent situé dans le voisinage de la source, et retourne une valeur booléenne. Il permet de déterminer quels agents parmi le voisinage de la source sont des cibles de cette interaction de multicast. Ce critère manipule des primitives abstraites de la source, ainsi que des cibles.

Caractérisation de "préconditions" et de "declencheur" : Le critère d'acceptation de cibles permet de définir si un agent situé dans le voisinage de la source doit être une cible de l'interaction. Ce critère ne définit que les cibles pouvant participer à l'interaction. Pour que cette interaction puisse être initiée par un agent source, les préconditions et le déclencheur doivent être vérifiés. Ils définissent des conditions portant sur la source ainsi que l'ensemble des cibles de l'interaction.

Intégration à la matrice d'interaction

Le modèle que nous venons de décrire permet de caractériser la sémantique d'une interaction de multicast. Pour les utiliser dans une simulation, ces dernières doivent pouvoir figurer dans la matrice d'interaction brute. Puisque les cibles de cette interaction ont un rôle réflexif, nous considérons qu'elles sont placées dans la matrice d'interaction de la même manière que les interactions individuelles, de cardinalité (1,1). Une interaction de multicast I figurant à l'intersection de la ligne associée à une famille d'agents F_1 et de la colonne associée à une famille d'agents F_2 se lit alors « I est une interaction initiée par une instance de la famille F_1 sur un nombre variable déterminé dynamiquement d'instances de la famille F_2 ». Ce nombre d'instances est déterminé à l'aide du critère d'acceptation de l'interaction, ainsi que la garde de distance de l'interaction dans la matrice. Un agent ne peut être la cible de cette interaction que si elle vérifie à la fois le critère d'acceptation, et si la distance la séparant de la source est inférieure ou égale à la garde de distance. L'intégration des interactions de multicast ne modifie donc que légèrement le modèle d'une matrice d'interaction brute.

Dans le cas de la simulation où des soldats se déplacent dans un environnement contenant des mines, où ces derniers peuvent déclencher une mine en marchant dessus, et où une mine déclenchée explose et blesse tous les soldats avoisinants, la matrice d'interaction brute prend l'apparence décrite sur la figure 6.22.

Source \ Cible	∅	Mine	Soldat
Mine			(Exploser, d = 3)
Soldat	(Se Déplacer)	(Déclencher, d = 0)	

Figure 6.22 – Exemple de matrice d'interaction brute intégrant la notion d'interaction de multicast. Cet exemple décrit un modèle dans lequel des soldats peuvent se déplacer dans l'environnement ou déclencher une mine, et où une mine peut exploser et blesser les soldats se trouvant à une distance inférieure ou égale à 3.

Dans le modèle formel, cela se traduit dans un premier temps par l'ajout de la notion d'*élément d'assignation de multicast*, qui représente l'association d'une interaction de multicast, d'une famille d'agents source, d'une famille d'agents cible et d'une garde de distance. Un élément d'assignation de multicast représente donc une interaction de multicast qu'un agent source peut initier avec un nombre déterminé dynamiquement d'agents cibles, tous instances de la même famille d'agents cible.

Définition 34. ***Élément d'assignation de multicast***
Un **élément d'assignation de multicast** est la représentation d'une interaction de multicast $\mathcal{I} \in \mathbb{I}_{(1,*)}$ qu'un agent instance d'une famille d'agents source $\mathcal{F}_S \in \mathbb{F}$ est capable d'initier avec un ensemble déterminé dynamiquement d'agents instances d'une famille d'agents cible $\mathcal{F}_T \in \mathbb{F}$, qui se situent au maximum à une distance $dist \in \mathbb{R}^+$ de l'agent source. Il est représenté sous la forme du n-uplet $(\mathcal{I}, dist, \mathcal{F}_S, \mathcal{F}_T) \in \mathbb{I}_{(1,*)} \times \mathbb{R}^+ \times \mathbb{F}^2$.

L'ensemble des interactions de multicast qu'un agent a la capacité d'initier avec un ensemble d'agents instances d'une famille d'agents cible est alors représenté sous la forme d'une *assignation de multicast*.

Définition 35. ***Assignation de multicast***
Une **assignation de multicast** $a^*_{S/T}$ est un ensemble d'éléments d'assignation de multicast. Elle représente l'ensemble des interactions de multicast qu'une instancec de la famille d'agents S peut initier en tant que source conjointement avec un ensemble d'instances de la famille d'agents T en tant que cible.
Plus formellement, on a :

$$\forall S \in \mathbb{F}, T \in \mathbb{F}, a^*_{S/T} \subset (\mathbb{I}_{(1,*)} \times \mathbb{R}^+ \times \mathbb{F}^2)$$

La matrice d'interaction brute représente l'ensemble des interactions que des instances d'une famille d'agents source est capable d'initier avec l'environnement ou une instance d'une famille d'agents cible. Elle correspond à l'ensemble des assignations pour une simulation, et se représente sous la forme d'une fonction associant une assignation dégénérée à chaque famille d'agent source, et une assignation individuelle à chaque couple de familles d'agents source et cible. L'ajout de la notion d'assignation de multicast enrichit cette fonction, puisqu'elle associe aussi une assignation de multicast à un couple de familles d'agents source et cible. Plus formellement, la matrice d'interaction brute se note alors :

$$\begin{array}{rlcl} & \mathbb{F} \times \big(\mathbb{F} \cup \{\emptyset\}\big) & \rightarrow & \mathcal{P}(\mathbb{I}_{(1,0)} \times \mathbb{F}) \cup \mathcal{P}\big(\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F}^2\big) \cup \mathcal{P}\big(\mathbb{I}_{(1,*)} \times \mathbb{R}^+ \times \mathbb{F}^2\big) \\ \mathcal{M} : & (\mathcal{S}, \mathcal{T}) & \rightarrow & \begin{cases} a_{\mathcal{S}/\emptyset} & \text{si } \mathcal{T} = \emptyset \\ a_{\mathcal{S}/\mathcal{T}} \cup a^*_{\mathcal{S}/\mathcal{T}} & \text{sinon} \end{cases} \end{array}$$

6.2.2 Intégration à la méthodologie

Pour intégrer les interactions de multicast à la méthodologie, et donc permettre leur spécification, nous devons reconsidérer les étapes ayant une influence sur la spécification du modèle d'une interaction, ainsi que les étapes où les spécifications dépendent du contenu du modèle d'une interaction. Ces étapes sont au nombre de trois : l'étape de description de la sémantique d'une interaction, l'étape de description de la matrice d'interaction brute, et l'étape de spécification des primitives abstraites des agents. Nos reconsidérons ces trois étapes afin de répondre à trois problèmes se posant suite à la définition du modèle d'une interaction de multicast :

- Quelle représentation graphique utiliser pour spécifier la sémantique d'une interaction de multicast ?
- Comment déterminer si une interaction est individuelle ou de multicast d'après la matrice d'interaction ?
- Quelles sont les primitives abstraites devant être implémentées par les agents pouvant être la source ou la cible d'une interaction de multicast ?

Spécification graphique d'une interaction de multicast

Une interaction de multicast reste avant tout une interaction. Sa spécification graphique est donc très similaire aux interactions individuelles et dégénérées. Toutefois, puisque ces interactions définissent un critère d'acceptation de cibles, cette représentation graphique doit être étendue pour décrire ce critère. La représentation graphique d'une interaction de multicast prend la forme générale décrite dans la figure 6.23.

Par exemple, dans le cadre de la modélisation de la propagation d'une contagion, la représentation graphique de l'interaction Propager contagion est décrite sur la figure 6.24. Puisque cette interaction ne cible que les agents sains, le critère d'acceptation y est « une cible doit être saine », exprimé sous la forme d'une primitive "estSain" que doivent implémenter toutes les cibles. Puisque l'infection n'a de sens que si la source est malade, et s'il y a au moins une cible à infecter, les préconditions de cette interaction portent sur une primitive "porteUneMaladie" que la source doit implémenter, et sur le nombre de cibles devant être différent de 0. Les actions de cette interaction consistent à rendre malade chaque cible, via leur primitive "devenirMalade".

« interaction de multicast » IDENTIFIANT DE L'INTERACTION(source,cibles,nombreCibles)		
Signatures	nom	identifiant de la signature
	Source	*identifiant de la signature d'agent pour "Source"*
	Cible	*identifiant de la signature d'agent pour "Cible"*
	Environnement	*identifiant de la signature de l'environnement*
Déclencheur	*% Déclencheur de l'interaction, décrit sous la forme d'un algorithme. Il manipule des primitives de l'agent nommé "source" dans cette interaction, ainsi que des agents cibles nommés "cible[1]", "cible[2]" ... "cible[nombreCibles]". "nombreCibles" est le nombre d'agents vérifiant le critère d'acceptation de l'interaction, ainsi que la garde de distance dans la matrice d'interaction.*	
Préconditions	*% Préconditions de l'interaction, décrites sous la forme d'un algorithme. Elles manipulent des primitives de l'agent nommé "source" dans cette interaction, ainsi que des agents cibles nommés "cible[1]", "cible[2]" ... "cible[nombreCibles]". "nombreCibles" est le nombre d'agents vérifiant le critère d'acceptation de l'interaction, ainsi que la garde de distance dans la matrice d'interaction.*	
Actions	*% Actions de l'interaction, décrites sous la forme d'un algorithme. Elles manipulent des primitives de l'agent nommé "source" dans cette interaction, ainsi que des agents cibles nommés "cible[1]", "cible[2]" ... "cible[nombreCibles]". "nombreCibles" est le nombre d'agents vérifiant le critère d'acceptation de l'interaction, ainsi que la garde de distance dans la matrice d'interaction.*	
Critère d'acceptation (source,cible)	*% Critère d'acceptation d'un agent cible par un agent source. Il manipule des primitives de l'agent "source" et de l'agent "cible"*	

FIGURE 6.23 – Forme générale de la représentation graphique permettant la spécification d'une interaction de multicast.

« interaction de multicast » PROPAGER CONTAGION(source,cibles,nombreCibles)		
Signatures	nom	identifiant de la signature
	Source	SourceDePropagerContagion
	Cible	CibleDePropagerContagion
	Environnement	EnvironnementDansPropagerContagion
Déclencheur	vrai	
Préconditions	source.porteUneMaladie() **et** nombreCibles $\neq 0$	
Actions	Pour tout i allant de 1 à nombreCibles, faire : cibles[i].devenirMalade() Fin pour	
Critère d'acceptation (source,cible)	cible.estSain()	

FIGURE 6.24 – Spécification d'une interaction de multicast PROPAGER CONTAGION, qui propage la maladie portée par une source sur un ensemble de cibles saines.

Interaction individuelle ou de multicast ?

Les interactions de multicast sont représentées dans la matrice d'interaction de la même manière que les interactions individuelles. Par conséquent, cette représentation graphique seule ne permet pas de déterminer si une interaction est de multicast ou individuelle. Seule la description sémantique de l'interaction permet de faire la différence. Ainsi, lors de l'étape de construction de la matrice d'interaction brute, placer l'identifiant d'une interaction encore non spécifiée dans une cellule de la matrice d'interaction située à l'intersection d'une ligne et d'une colonne associées à des familles d'agents ne permet de déduire que partiellement sa cardinalité : il s'agit soit d'une interaction individuelle, soit d'une interaction de multicast. Les interactions placées dans la colonne $\emptyset$ échappent à cette incertitude : ce sont des interactions dégénérées.

Inversement, si une interaction est définie comme de multicast lors de la description de sa sémantique, elle ne peut être placée que dans les cellules dont la colonne est associée à une famille d'agents : elle ne peut être placée dans la colonne $\emptyset$.

Quelles primitives spécifier dans les agents ?

Les agents source et cibles d'une interaction sont caractérisés par une signature dans l'interaction, qui décrit l'ensemble des primitives qu'ils doivent spécifier. Dans le cas des interactions individuelles et de dégénérées, ces signatures fournissent un résumé des diverses primitives abstraites manipulées dans les préconditions, le déclencheur et les actions de l'interaction. Les interactions de multicast ne se limitent pas à ces spécifications, et définissent de plus un critère d'acceptation d'une cible. Ce critère manipule lui aussi des primitives devant être implémentées par les sources ou les cibles de l'interaction, qui doivent par conséquent figurer dans les signatures des entités participant à l'interaction.

6.2.3 Intégration dans les algorithmes de simulation

Intégrer les interaction de multicast dans les algorithmes de simulation revient à déterminer comment recenser les interactions de multicast qu'un agent a l'opportunité d'initier, et à intégrer de telles interactions dans le modèle de sélection d'interaction.

Dans le cadre restreint de IODA, une interaction qu'un agent a potentiellement l'opportunité d'initier est représentée sous la forme d'un tuple composé d'un agent source, d'un élément d'assignation, et d'aucun ou un agent cible. L'ensemble des interactions qu'un agent a l'opportunité d'initier, appelé potentiel d'interaction dans la section 4.3 du chapitre 4, est uniquement composé de tels tuples. Sa construction repose sur les concepts de tuples éligibles et tuples réalisables, que nous étendons ici aux interactions de multicast.

Un tuple est dit éligible si l'agent source du tuple a la capacité d'initier l'interaction du tuple avec les cibles du tuple, ce qui se mesure sur trois critères. Premièrement, la matrice d'interaction doit attester que l'interaction peut avoir lieu entre la famille d'agent source et des instances de la famille d'agents cible. Cela se traduit par le fait que l'élément d'assignation du tuple est présent dans la matrice d'interaction. Secondement, il faut que l'ensemble des cibles du tuple fassent partie des agents perçus par l'agent source (son voisinage). Enfin, la distance séparant la source de chacune de ses cibles doit être inférieure ou égale à la garde de distance de l'élément d'assignation.

Définition 36. ***Éligibilité pour une interaction de multicast***
Soient $s \in \mathbb{E}$ une entité dont la famille est $\mathcal{F}_S \in \mathbb{F}$, $(t_i)_{i\in\mathbb{N}} \in \mathcal{P}(\mathbb{E})$ un ensemble d'entités dont la famille est $\mathcal{F}_T \in \mathbb{F}$, et $a_{multi} = (\mathcal{I}, dist, \mathcal{F}_S, \mathcal{F}_T) \in \mathbb{I}_{(1,*)} \times \mathbb{R}^+ \times \mathbb{F}^2$ un élément d'assignation de multicast.
Un tuple $T = (a_{multi}, s, (t_i)_{i\in\mathbb{N}})$ est dit **éligible** si toutes les conditions qui suivent sont vérifiées :
- s a la capacité d'initier l'interaction de multicast associée à l'élément d'assignation a_{multi} avec les agents $(t_i)_{i\in\mathbb{N}}$ pour cible, *i.e.* $a_{multi} \in \mathcal{M}(\mathcal{F}_S, \mathcal{F}_T)$;
- la distance entre la source s et chaque cible t_i est inférieure ou égale à la garde de distance associée à l'élément d'assignation a_{multi}, *i.e.* si $\forall i, environnement.distance(s, t_i) \leq dist$;
- toutes les cibles t_i appartiennent au voisinage de la source s, *i.e.* si $\forall i, t_i \in \mathcal{V}(e)$.

Ce critère est syntaxique, et se fonde sur la matrice d'interaction brute de la simulation, ainsi que sur le voisinage de l'agent source. La validité sémantique d'un tuple est vérifiée à l'aide du critère de réalisabilité, qui vérifie si les préconditions et le déclencheur sont vérifiés pour la source et l'ensemble de cibles du tuple, si chaque cible vérifie le critère d'acceptation de l'interaction de multicast, et s'il n'existe pas d'autre agent dans le voisinage de la source qui aurait pu être ajouté aux cibles de l'interaction de multicast.

Définition 37. ***Réalisabilité pour une interaction de multicast***
Soient $s \in \mathbb{E}$ une entité dont la famille est $\mathcal{F}_S \in \mathbb{F}$, $(t_i)_{i\in[1,n]} \in \mathbb{E}^n$ un ensemble de n entités dont la famille est $\mathcal{F}_T \in \mathbb{F}$, et $a_{multi} = (\mathcal{I}, dist, \mathcal{F}_S, \mathcal{F}_T) \in \mathbb{I}_{(1,*)} \times \mathbb{R}^+ \times \mathbb{F}^2$ un élément d'assignation de multicast.
Un tuple $T = (a_{multi}, s, (t_i)_{i\in\mathbb{N}})$ est dit **réalisable** si toutes les conditions qui suivent sont vérifiées :
- T doit être éligible ;
- les préconditions de l'interaction $\mathcal{I}$ sont vérifiées pour l'agent source s et les agents cibles $(t_i)_{i\in\mathbb{N}}$, *i.e.* $preconditions(\mathcal{I})(s, (t_i)_{i\in[1,n]}) = vrai$;
- le déclencheur de l'interaction $\mathcal{I}$ est vérifié pour l'agent source s et les agents cibles $(t_i)_{i\in\mathbb{N}}$, *i.e.* $declencheur(\mathcal{I})(s, (t_i)_{i\in[1,n]}) = vrai$;
- le critère d'acceptation de cible est vérifié par chaque cible du tuple, *i.e.* $\forall i \in [1, n], accepterCible(\mathcal{I})(s, (t_i)_{i\in[1,n]})$;
- s'il n'existe pas d'autre agent dans le voisinage de la source qui aurait pu être ajouté aux cibles de l'interaction de multicast, *i.e.* si $\forall c \in \mathcal{V}(e), (a_{multi}, s, (t_i)_{i\in[1,n]} \cup \{c\})$ est réalisable $\Rightarrow c \in (t_i)_{i\in[1,n]}$.

La figure 7 décrit comment calculer le potentiel d'interaction d'un agent, en se reposant directement sur les éléments déclaratifs du modèle IODA.

Le modèle de selection réactive d'interaction n'est pas modifié par l'ajout des interactions de multicast. Il consiste toujours à attribuer une priorité aux éléments d'assignation de la matrice d'interaction brute, et à sélectionner un tuple réalisable de priorité maximale.

6.3 Synthèse du chapitre

Dans ce chapitre, nous montrons que les interactions individuelles et les interactions dégénérées suffisent à modéliser la majorité des simulations. Pour cela, nous établissons des patrons de conception permettant de décomposer la plupart des interactions de cardinalité (n, m) en un ensemble d'interactions individuelles et dégénérées. Nous pouvons ainsi modéliser des interactions complexes à l'aide d'interactions simples.

Patrons de conception. Le premier patron est appelé modélisation par agrégation d'agents. Il base la décomposition d'une interaction complexe sur un agent auquel toutes les sources et les cibles sont agrégées. Les sources y sont ajoutées en initiant une interaction individuelle prenant pour cible l'agent agrégat. Les cibles y sont ajoutées en subissant une interaction individuelle prenant pour source l'agent agrégat. L'interaction complexe est alors initiée par l'agent agrégat, à l'aide d'une interaction dégénérée. Les principes de ce patron permettent en théorie de modéliser tout type d'interaction complexe.

Le second patron de conception est appelé modélisation par répétition d'une interaction. Il permet de modéliser les interactions de cardinalité $(1, n)$ ayant un effet identique sur toutes les cibles (par exemple une explosion prenant pour cible des soldats). La décomposition y est basée sur :
- une interaction individuelle initiée par l'agent source successivement sur chaque agent cible, afin d'appliquer l'effet de l'interaction complexe sur chacune de ses cibles ;
- sur une interaction dégénérée initiée par la source avant ou après les interactions individuelles, afin d'appliquer une seule fois l'effet de l'interaction complexe sur sa source.

Ces patrons de conception ne permettent pas en pratique de modéliser tout type d'interaction si les agents ne peuvent initier qu'une interaction par pas de temps. En effet, seule une cible peut être modifiée ou agrégée par pas de temps, laissant l'opportunité aux autres cibles d'échapper à l'interaction.

Algorithme 7 : Algorithme permettant de calculer le potentiel d'interaction d'une entité x. Il s'agit d'une extension de l'algorithme 2 de la page 121 aux interactions de multicast.

```
potentiel d'interaction(x)
début
    R ⇐ ∅;
    % Recensement des tuples contenant des interactions individuelles ou de multicast
    pour tous les F ∈ 𝔽 faire
        pour tous les a ∈ M(famille(x), F) faire
            % Si l'interaction de l'élément d'assignation est une interaction individuelle
            si card(I(a)) = (1, 1) alors
                pour tous les y ∈ V(x) faire
                    si (a, x, y) est réalisable alors
                        R ⇐ R ∪ {(a, x, y)};
            sinon
                % Si l'interaction de l'élément d'assignation est une interaction de multicast
                si card(I(a)) = (1, ∗) alors
                    T ⇐ ∅;
                    pour tous les y ∈ V(x) faire
                        si accepterCible(I(a))(x, y) et environnement.distance(x, y) ≤ dist(a)
                        alors
                            T ⇐ T ∪ {y};
                    si (a, x, T) est réalisable alors
                        R ⇐ R ∪ {(a, x, T)};
    % Recensement des tuples contenant des interactions dégénérées
    pour tous les a ∈ M(famille(x), ∅) faire
        si (a, x) est réalisable alors
            R ⇐ R ∪ {a, x};
    retourner R;
fin
```

Extension aux interactions multicast. Deux principes différents permettent de modéliser des interactions ayant un effet simultané sur leurs cibles :

- utiliser une représentation du temps complexe, et conserver la représentation actuelle des interactions ;
- utiliser une représentation des interactions complexe, et conserver la représentation du temps.

L'utilisation d'une représentation du temps complexe n'est pas envisageable, car cela revient à complexifier la spécification de l'ensemble des simulations pour traiter des cas minoritaires. Nous montrons que la représentation des interactions complexes ne nécessite dans IODA qu'un nombre restreint de modifications et est donc praticable.

Afin de modéliser ces interactions, notre extension de IODA repose sur un type d'interaction plus complexe appelé interactions multicast. Les interactions multicast impliquent simultanément un nombre de cibles déterminé dynamiquement par l'agent source lors de sa prise de parole. Le nombre de cibles de telles interactions dépend du voisinage de l'agent source et d'un critère défini dans l'interaction multicast déterminant si un voisin est pris pour cible par l'interaction. Cet ajout dans notre taxinomie ne modifie pas de manière significative le modèle formel de IODA, et ne nécessite pour sa grande partie que des ajustements dans les algorithmes déja existants.

Cette extension permet donc de couvrir un plus grand nombre de simulations avec IODA pour un coût restreint en terme de complexification de la structure du modèle et de la méthodologie de conception.

Chapitre 7

Éviter les biais de simulation

Plan du chapitre :

Des choix de conception apparaissent tout au long du processus de conception d'une simulation. Lorsque ces choix ne sont pas faits volontairement par le concepteur, des biais sont introduits dans le modèle ou son implémentation. Ces biais altèrent les résultats de simulation et mènent donc à des conclusions erronées quant à la validité d'une explication d'un phénomène. Afin d'éviter d'introduire des biais dans une simulation, les choix de conception doivent être faits sciemment par les concepteurs.

Dans ce chapitre, nous montrons que pour faire apparaître de tels choix, trois unités fonctionnelles sous-jacentes à toute simulation doivent être finement spécifiées.

En menant des études selon cette séparation et en focalisant la conception sur les interactions, nous avons dégagé deux extensions de IODA *permettant de rendre explicites les choix de conception relatifs deux problèmes de simulation :*

- *la participation simultanée à plusieurs interactions ;*
- *la spécification de comportements stochastiques.*

Une simulation multi-agents est un outil utilisé afin de renforcer ou affaiblir des hypothèses émises sur le comportement et les interactions des agents qui seraient à l'origine d'un phénomène émergeant. Le modèle d'une simulation est la concrétisation de ces hypothèses. Il doit à ce titre fournir toutes les informations permettant de décrire le phénomène simulé, mais aussi fournir les informations permettant son implémentation.

La frontière entre ce qui est propre au modèle et ce qui est propre à l'implémentation est ténue et ambiguë. Par conséquent, aucun consensus n'existe concernant la nature des informations devant être fournies dans un modèle. Des choix de conception apparaissent donc tout au long du processus de conception d'une simulation. Les problèmes liés à ces choix sont multiples et incluent en particulier les problèmes qui suivent :

- un expert du domaine peut difficilement effectuer seul des choix de conception propres à des points techniques de l'implémentation ;
- un concepteur peut effectuer des choix de modélisation à son insu, de manière implicite ;
- un concepteur peut omettre de faire certains choix, aboutissant alors à des choix arbitraires d'implémentation.

Les choix effectués ont une influence variable sur les résultats de simulation. Elle peut être mineure, auquel cas les résultats de simulation sont valides malgré un choix arbitraire ou un choix mal inspiré, ou majeure et donc aboutir à des résultats erronés et biaisés. Concevoir une simulation non-biaisée consiste donc à identifier où ces choix surviennent lors du processus de conception de simulation, déterminer quelle influence ces choix ont sur les résultats de simulation, et fournir un moyen d'exprimer de manière explicite ces choix dans le modèle.

L'identification des choix de conception n'est pas aisée, car elle dépend grandement de l'expressivité du modèle utilisé pour concevoir une simulation. Dans ce chapitre, nous proposons dans une première section de caractériser les problèmes liés aux choix de modélisation ainsi que la solution que nous envisageons

afin de sensibiliser les concepteurs à ces problèmes. Ensuite, en nous appuyant sur les concepts avancés par notre approche centrée sur les interaction de la conception de simulations, nous identifions une décomposition possible du fonctionnement de tout simulateur en trois unités fonctionnelles disjointes, permettant d'étudier les choix de modélisation selon trois perspectives différentes. Dans les deux autres sections de ce chapitre, nous nous appuyons sur cette décomposition afin de caractériser deux problèmes liés aux choix de conception et nous leur proposons une solution en nous aidant de l'approche IODA. Dans la seconde section, nous nous focalisons sur des problèmes liés à la participation simultanée d'un agent à plusieurs interactions, et leur proposons une solution basée sur l'attribution d'une classe à chaque interaction. Dans la dernière section, nous nous intéressons à des problèmes liés à la description de comportements réactifs stochastiques et leur proposons une solution basée sur la définition de politiques de sélection d'interaction.

7.1 Caractérisation des problèmes liés aux choix de modélisation

Le pré-requis fondamental à la résolution des problèmes issus des choix de modélisation consiste à comprendre comment ces problèmes apparaissent lors de la conception d'une simulation. Ce n'est qu'à cette condition qu'il est possible de fournir un procédé de conception permettant de résoudre ces problèmes. Le point de vue de Galán *et al.* [GII+09] à ce propos est particulièrement intéressant. La terminologie qu'ils proposent permet en effet de caractériser la nature des différents problèmes liés aux choix de modélisation. Nous nous appuyons sur cette terminologie afin d'identifier différents moyens permettant d'éviter ces problèmes, et déterminer ainsi dans quel cadre se situe l'approche que nous proposons en tant que solution.

7.1.1 Sémantique des biais selon Galán *et al.*

Selon Galán *et al.*, deux types de problèmes sont à l'origine de résultats de simulation biaisés. L'un est lié à ce qu'ils appellent des *erreurs*, l'autre à ce qu'ils appellent des *artefacts*.

Erreurs

Selon Galán *et al.*, **une *erreur* est le résultat d'une différence entre ce que le concepteur pense décrire dans le modèle (ou l'implémentation), et ce qu'il décrit réellement**. Ces dernières peuvent survenir à trois niveaux dans le processus de conception de simulations.

Une erreur peut survenir dans un premier temps lors de la description du modèle, dans le cas où ce que le concepteur pense décrire dans le modèle diffère de ce qui y est réellement décrit. Nous interprétons ce type d'erreur comme provenant de choix de conception faits implicitement dans le modèle, aboutissant à une mauvaise utilisation de son formalisme.

Exemple.
Dans le cas de IODA*, cela consisterait par exemple à définir dans une matrice d'interaction qu'un* `Proie` *est capable de* FUIR *un* `Predateur` *se situant au plus à une distance de* 10*, et à définir qu'une* `Proie` *ne perçoit les autres agents que dans un disque de rayon* 0,5*. Dans ce cas, on souhaitait faire* FUIR *les* `Proies` *à une distance de* 10*, mais la mauvaise utilisation du halo des* `Proies` *ne leur permet de* FUIR *qu'à une distance de* 0,5*.*

La solution à ces erreurs consiste à favoriser la bonne utilisation du formalisme, en fournissant par exemple des outils faciles à utiliser permettant de construire le modèle en faisant explicitement état des choix de conception s'offrant au concepteur, et en documentant de manière précise les implications possibles d'un choix de modélisation sur les résultats de la simulation.

Le second type d'erreur survient lors de la construction du simulateur, dans le cas où ce que le concepteur pense implémenter diffère de ce qu'il implémente réellement. Nous interprétons ce type d'erreur comme provenant d'une mauvaise utilisation du langage de programmation, ou comme provenant de l'utilisation de bibliothèques ne fonctionnant pas comme elles le devraient.

Exemple.
Ce type d'erreur a par exemple été rencontré par Axtell et al. [AAEC96] lors de l'implémentation de leur modèle SugarScape qu'ils ont dû corriger afin d'obtenir des résultats de simulation viables. Dans ce modèle, ils utilisent un modèle de temps discret décomposé en pas de temps. Dans leur implémentation initiale, ils souhaitaient garantir l'équité dans la prise de parole des agents en donnant lors d'un pas de temps la parole à un seul agent choisi aléatoirement. Cette façon de procéder ne garantissait pas que tous les agents puissent prendre la parole régulièrement. Ils ont donc dû envisager une autre approche, dans laquelle la liste des agents subit des permutations aléatoires au début d'un pas de temps et où la parole est donnée séquentiellement à chaque agent de cette liste.

Les solutions à ces erreurs sont multiples. Elles peuvent consister à mieux documenter les bibliothèques utilisées et à s'assurer que leur comportement est correct, à mettre en garde les concepteurs contre les erreurs d'implémentation courantes pouvant survenir ou encore à automatiser au maximum la transformation d'un modèle en un simulateur. Ce dernier point nécessite de faire apparaître explicitement les choix fondamentaux à l'implémentation dans le modèle de la simulation.

Le dernier type d'erreur survient lors du passage du modèle à l'implémentation, dans le cas où ce que le concepteur pense décrire dans son modèle diffère de l'implémentation qui en est faite. Nous interprétons ce dernier type d'erreur de deux façons. Le concepteur peut avoir fourni une spécification incomplète à son modèle laissant certains choix de conception non spécifiés, ce qui aboutit alors à des choix implicites ou arbitraires dans l'implémentation. Le modèle formulé peut aussi fournir une spécification ambiguë à certains choix de conception laissant place à plusieurs interprétations, ce qui mène alors à des choix arbitraires ou implicites lors de l'implémentation. Dans les deux cas, la solution consiste à identifier de manière exhaustive tous les choix de conception existants, à identifier toutes les alternatives existant pour chaque choix de conception, à fournir un modèle formel à toutes les alternatives, et à pousser les concepteurs à choisir explicitement dans la méthodologie l'alternative qu'ils utilisent pour chaque choix de conception s'offrant à eux.

Artefacts

Au contraire des *erreurs*, dont l'origine est la mauvaise utilisation du formalisme du modèle ou des outils d'implémentation, ou encore des choix de conception effectués implicitement ou arbitrairement, **les *artefacts* apparaissent lorsqu'un modèle et une implémentation correspondent exactement à ce que le concepteur souhaite. Leur présence dans un modèle constitue le symptôme d'hypothèses erronées**.

Galán *et al.* caractérisent un artefact en se reposant sur quatre concepts : « Core assumptions » (que nous traduisons par « hypothèses fondamentales »), « Accessory assumptions » (que nous traduisons par « hypothèses annexes »), « Significant assumptions » (que nous traduisons par « hypothèses significatives ») et « Non-significant assumptions » (que nous traduisons par « hypothèses non-significatives »). Une *hypothèse fondamentale* correspond à une hypothèse que le concepteur considère comme importante pour le fonctionnement du modèle. Idéalement, seules ces hypothèses doivent figurer dans le modèle. Toutefois, **en pratique, ces hypothèses doivent être complétées par des *hypothèses annexes*, qui ne sont pas considérées comme cruciales dans la description du modèle, mais sont nécessaires afin de construire un simulateur fonctionnel**. Le choix de la nature de l'ordonnanceur utilisé dans une simulation de tri collectif en est un exemple, puisque ce dernier n'a pas d'influence sur l'apparition de tas.

En complément, une hypothèse est qualifiée de *significative* si elle est la cause de changements significatifs dans les résultats obtenus. Dans le cas contraire, l'hypothèse est qualifiée de *non-significative*.

D'après cette terminologie, un *artefact* dans une simulation correspond à une hypothèse annexe se révélant être une hypothèse significative. Un artefact est donc le fruit de la sous-estimation de l'impact d'une hypothèse sur les résultats de la simulation. Remédier à ce problème revient alors à réviser les hypothèses émises concernant le phénomène.

7.1.2 Approche retenue dans ce chapitre

La présence d'erreurs ou d'artefacts dans une simulation est intimement liée aux choix de conception effectués explicitement ou implicitement lors de la construction du modèle. Lorsqu'ils sont implicites, une partie de la sémantique du modèle est déterminée arbitrairement. Nous soutenons que le concepteur doit être conscient de chaque choix de modélisation ou d'implémentation qu'il effectue. Pour cela, la méthodologie de conception de simulation doit pousser le concepteur à les exprimer de manière exhaustive et explicite.

La construction d'une telle méthodologie nécessite la résolution de trois problèmes différents :

- Identifier les choix de conception apparaissant de manière explicite ou implicite lors de la construction du modèle et de l'implémentation ;
- Fournir un modèle permettant d'exprimer explicitement de tels choix ;
- Fournir une méthodologie de conception aidant les utilisateurs à exprimer le plus simplement possible ces choix, afin de ne pas complexifier excessivement la construction de modèles.

Dans ce chapitre, nous décrivons dans un premier temps la démarche que nous avons suivie afin d'identifier certains choix de conception, puis nous détaillons dans un second temps deux études nous ayant permis d'établir un complément au modèle, à la méthodologie, et aux algorithmes de l'approche IODA présentée dans la partie présente, permettant de rendre explicites des choix de conception concernant deux aspects différents de la simulation :

- le problème de la simultanéité de deux interactions ;
- le problème de la spécification de comportements stochastiques.

Décomposition fonctionnelle d'une simulation

Le spectre des choix possibles est excessivement large et concerne des aspects très différents de la simulation. Afin de faciliter leur identification, nous considérons une décomposition particulière (résumée sur la figure 7.1) d'une simulation en trois unités fonctionnelles disjointes : l'*Unité d'activation*, l'*Unité de sélection* et l'*Unité de définition*. Dans la décomposition fonctionnelle que nous proposons, chaque unité est en charge d'une fonction particulière de la simulation.

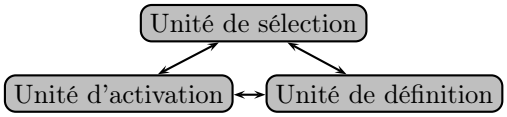


FIGURE 7.1 – Décomposition d'une simulation en trois unités fonctionnelles.

L'*unité d'activation* est utilisée afin de décrire tous les éléments liés à la représentation du temps dans la simulation. Elle consiste donc en particulier à définir **quand donner la parole aux agents**, combien durent les actions/interactions qu'ils effectuent, que faire si un agent tente d'interagir avec un agent déjà en interaction avec un autre agent, comment effectuer la mise à jour des agents, *etc.*

L'*unité de définition* est utilisée afin de décrire **quelles interactions peuvent avoir lieu** entre les agents présents dans la simulation, dans quelles conditions elles peuvent survenir et quels sont leurs effets. Elle décrit aussi les interactions qu'un agent peut entretenir avec l'environnement, ou avec son propre état, et décrit aussi comment son état évolue en absence de toute action.

Enfin, l'*unité de sélection* décrit le procédé réactif ou cognitif utilisé par un agent pour **sélectionner les interactions** qu'il initie lorsque l'unité d'activation lui donne la parole.

Les trois unités fonctionnelles sont sous-jacentes à toute simulation multi-agents, bien qu'elles ne soient pas systématiquement séparées logiciellement (nous illustrons ce point dans la section suivante). Puisque la conception d'une simulation implique des choix cruciaux concernant ces trois unités, nous soutenons qu'il est fondamental de faire apparaître explicitement cette séparation, même dans le cas de simulations contenant des agents réactifs.

Études menées dans ce chapitre

Dans ce chapitre, nous nous basons sur notre approche centrée sur les interaction afin d'étudier dans deux sections différentes des problèmes liés aux choix de modélisation respectivement dans l'unité d'activation, et dans l'unité de sélection.

La deuxième section de ce chapitre porte sur l'unité d'activation. Nous y étudions des problèmes liés à la participation simultanée d'un agent à plusieurs interactions. Nous tirons parti de cas d'étude ainsi que de la représentation centrée sur les interaction fournie par IODA afin de caractériser un modèle permettant de rendre explicite les choix de conception concernant ce problème.

La dernière section de ce chapitre porte sur l'unité de sélection. Nous y étudions des problèmes liés à la description de comportements réactifs régis par des probabilités. Nous tirons parti de cas d'étude ainsi que de la représentation centrée sur les interactions fournie par IODA afin d'étendre le modèle réactif de sélection d'interaction présenté dans le chapitre 3. Ce modèle permet alors de rendre explicites les différentes façons d'implémenter des comportements réactifs stochastiques. Il constitue ainsi un support de choix permettant d'éviter certaines erreurs survenant couramment lors de la spécification de tels comportements.

7.2 Participation simultanée à plusieurs interactions

Dans bien des approches permettant de construire des simulateurs, le processus de conception est centré sur les agents et les actions qu'ils peuvent initier, plutôt que sur les interactions qui peuvent survenir entre eux. En conséquence, l'identification des interactions qu'un agent peut initier lorsque la parole lui est donnée prend rarement en compte les interactions auxquelles il est en train de participer en tant que cible, ou les interactions dans lesquelles ses cibles potentielles sont déjà en train de participer. Lorsque ces informations ne sont pas prises en compte, un choix implicite de conception est fait : il est supposé qu'un agent peut initier une interaction avec une cible dans tous les cas, même lorsque cette cible est déjà en train de participer à une autre interaction, ou que l'agent source est la cible d'une autre interaction. Ce choix a un effet critique dans certaines simulations.

En effet, Michel *et al.* [MGF03] ont par exemple montré que dans une simulation modélisant l'évolution d'un écosystème, le nombre de naissances par pas de temps et l'espérance mathématique du nombre de naissances est grandement influencé par ce choix. Dans la section qui suit, nous présentons plusieurs approches existantes permettant de remédier à ce problème.

7.2.1 Approches existantes

La première approche permettant de remédier au problème de la participation simultanée à plusieurs interactions consiste à modifier directement le contenu du modèle. Lawson et Park [LP00] en fournissent un exemple, en complétant le modèle SugarScape d'Epstein et Axtell [EA96] dont l'un des objectif est de décrire l'évolution de populations dans un écosystème. Dans le modèle proposé par Epstein et Axtell, un agent peut se reproduire plus de deux fois lors d'un pas de temps : une fois en tant que source de l'interaction, et plusieurs autres fois en tant que cible. Pour corriger ce problème, Lawson et Park proposent de compléter le modèle en ajoutant une période de gestation aux agents pouvant se reproduire. La solution proposée par Lawson et Park est ad-hoc et se limite à ce cas particulier du modèle SugarScape. Ainsi, ce type d'approche remédiant au problème de participation simultanée à plusieurs interactions repose uniquement sur la capacité des concepteurs à analyser eux-mêmes et sans aucune aide le modèle qu'ils conçoivent, et à trouver par eux-mêmes la solution à ces problèmes.

D'autres approches fournissent des outils théoriques aidant l'identification des actions initiées par les agents qui peuvent être sujettes au problème de la simultanéité. L'approche de Michel *et al.* [MGF03] en est un exemple.

Interactions fortes et interactions faibles [MGF03]. Dans leur approche, Michel *et al.* font la distinction entre le terme *action*, qui qualifie une action qu'un agent est capable d'initier dont les effets ne concernent que cet agent, et le terme *interaction*, qui qualifie une relation existant entre plusieurs

actions. Selon cette approche, la reproduction entre un agent mâle et un agent femelle est considérée comme le résultat d'une action de reproduction initiée par un agent mâle, conjointement à une action de reproduction initiée par un agent femelle.

Ils définissent alors un ensemble d'actions initiées par des agents comme étant en *interaction forte* si l'objectif qu'un agent cherche à atteindre en initiant l'une de ces actions ne peut être atteint que :

- si d'autres agents effectuent certaines actions précises de cette interaction forte ;
- ou si d'autres agents n'effectuent pas certaines autres actions précises de cette interaction forte.

Par exemple, l'action de reproduction est en interaction forte avec d'autres actions de reproduction, puisque deux agents ne peuvent avoir une descendance que s'ils exécutent tous deux cette action. De même l'action de fuite est en interaction forte avec l'action de reproduction. En effet, la fuite est plus urgente que la reproduction. Ainsi, un agent ne peut se reproduire avec un autre agent que si ce dernier n'effectue pas l'action de fuite. Au contraire, des actions sont considérées en *interaction faible* s'il n'existe pas cette dépendance entre les actions.

Dans la méthodologie de conception qu'ils proposent, déterminer si deux actions sont en interaction forte ou en interaction faible permet d'identifier dans quels cas la simultanéité nécessite un traitement spécifique. Toutefois, l'approche proposée se limite à la phase d'analyse du modèle. Elle ne fournit pas les outils permettant de construire des modèles reposant sur ces concepts.

Enfin, certaines approches fournissent des moyens théoriques et techniques aidant à la conception de modèles en faisant explicitement mention des choix effectués concernant la simultanéité des interaction. C'est par exemple le cas de l'extension du modèle Influence/Réaction de Ferber et Müller [FM96] proposée par Weyns et Holvoet [WH03] .

Extension par Weyns et Holvoet [WH03] du modèle Influences/Réaction. L'approche de Weyns et Holvoet repose sur la qualification de la relation existant entre les actions initiées par les agents de la simulation. Ils qualifient deux actions :

- d'*indépendantes* si leurs effets sont strictement indépendants ;
- de *jointes* si les actions sont complémentaires et ne peuvent être initiées séparément. Il s'agit par exemple du cas de l'action de reproduction initiée par un mâle et l'action de reproduction initiée par une femelle ;
- de *concurrentes* si seule l'une des deux actions peut être effectuée à un temps t ;
- d'*influentes* si ces actions peuvent être initiées séparément, mais que leur initiation conjointe renforce leur effet. C'est par exemple cas de l'action pousser un objet lourd.

La méthodologie qu'ils défendent consiste alors à établir de manière exhaustive les relations existant entre les actions pouvant être initiées par les agents et à traiter ces actions dans le moteur de simulation selon la nature de leur relation. Ce traitement se fait selon des lois énoncées par les concepteurs.

Bien que dans cette dernière approche aucun modèle précis n'est donné aux lois, elle a l'avantage indéniable d'obliger les concepteurs à traiter de manière explicite tous les cas possibles liés à la simultanéité dans une simulation. Toutefois, le modèle proposé suppose de déterminer de manière exhaustive la relation existant entre chaque action pouvant survenir dans la simulation : si n actions différentes sont utilisées dans la simulation, au moins n^2 relations doivent être étudiées dans la méthodologie. Ce point devient rapidement un problème dans des simulations large échelle, puisque dans ces simulations n prend des valeurs élevées. Dans cette section, nous étudions comment il serait possible de tirer parti de l'approche Ioda afin d'éviter ce problème.

Dans Ioda, les actions jointes de Weyns et Holvoet peuvent être modélisées sous la forme d'une interaction dont le déclencheur spécifie que la cible doit accepter d'interagir. Elles ne nécessitent donc pas de traitement spécifique. De plus, les actions influentes peuvent être modélisées à l'aide du principe de modélisation par agrégation d'agents décrit dans la section 6.1.2 du chapitre 6. Dans cette section, nous faisons donc le choix de porter nos études sur la caractérisation de l'indépendance et de la concurrence d'interactions et sur l'intégration de cette caractérisation dans la méthodologie Ioda ainsi que dans ses algorithmes de simulation. Nous nous basons pour cela sur des cas d'étude qui nous permettent d'illustrer différents choix de conception. Ces cas d'étude nous ont aussi permis d'identifier la solution envisagée permettant de spécifier explicitement les choix de conception.

7.2.2 Illustration expérimentale des biais

Dans cette section, nous nous basons sur des cas d'étude afin de construire progressivement les principes et le modèle permettant de rendre explicites les choix de conceptions liés à la gestion de la simultanéité dans l'unité d'activation. Pour cela, nous nous reposons sur deux expériences.

Dans la première, nous étudions un premier type d'erreur pouvant survenir dans l'implémentation naïve de l'unité d'activation utilisée dans une grande partie des simulations multi-agents. Ces résultats nous amènent à considérer une nouvelle implémentation de l'unité d'activation permettant d'éviter cette erreur.

Dans la seconde, nous étudions un second type d'erreur pouvant survenir dans l'implémentation naïve, mais aussi dans la nouvelle implémentation proposée lors de la première expérience. Ces résultats nous amènent à considérer une nouvelle implémentation de l'unité d'activation permettant d'éviter cette nouvelle erreur.

Nous déduisons alors de ces expériences une classification des interactions survenant dans une simulation. Elle permet de faire apparaître explicitement les choix de conception effectués dans l'unité d'activation sans pour autant nécessiter les description exhaustives rendant l'approche proposée par Weyns et Holvoet difficilement praticable dans des simulations large échelle.

Caractérisation des unités fonctionnelles utilisées

Puisque nous cherchons à évaluer l'impact de modifications de l'unité d'activation sur une simulation, chaque cas d'étude reposera sur la même unité de définition, et la même unité de sélection. De plus, afin de pouvoir implémenter et tester les différentes unités d'activation, nous avons été amenés à contraindre la nature de l'unité d'activation. Dans cette section, nous décrivons ces contraintes.

L'unité de définition spécifie la plupart des données dépendantes du phénomène simulé. Par conséquent, son contenu va changer d'une étude à l'autre. Toutefois, afin de faciliter la compréhension de nos exemples, chaque cas d'étude se focalise sur un même problème de simulation : la représentation d'un écosystème contenant des prédateurs et des proies.

L'unité de sélection permet de définir le modèle de sélection d'interaction utilisé par les agents dans nos expériences. Nous faisons ici le choix d'utiliser le modèle réactif de sélection d'interaction décrit dans le chapitre 3. Dans nos études, l'unité de sélection repose donc sur l'attribution de priorités aux différentes interactions pouvant être initiées par un agent, ainsi que sur l'algorithme 8, qui repose sur ces priorités pour sélectionner un tuple réalisable.

Algorithme 8 : Algorithme sur lequel repose l'unité de sélection dans les expériences effectuées dans la section 7.2. Cet algorithme définit comment un agent noté a choisit le tuple réalisable représentant l'interaction qu'il initie à un temps z.

```
début
    ℝ_z(a) ⇐ l'ensemble des tuples réalisables ayant a pour source;
    ℙ ⇐ l'ensemble décroissant des priorités que a fournit aux interactions qu'il peut initier;
    ℒ ⇐ ∅;
    pour p ∈ ℙ faire
        pour t ∈ ℝ_z(a) faire
            si l'élément d'assignation de t a pour priorité p alors
                ℒ ⇐ ℒ ∪ {t};
        si ℒ ≠ ∅ alors
            Le tuple réalisable sélectionné est choisi au hasard parmi les éléments de 𝕃;
            L'algorithme de sélection s'arrête;
    Aucun tuple réalisable n'est sélectionné;
fin
```

Enfin, dans les expériences effectuées dans cette section, nous contraignons l'unité d'activation afin de pouvoir réaliser des expériences. Pour cela, nous supposons que le temps utilisé dans ces simulations est discret et que, durant chaque pas de temps, l'ordonnanceur donne séquentiellement la parole aux agents

dans un ordre aléatoire redéfini à chaque pas de temps. Ces choix de conception sont les plus courants en simulation multi-agents. Nous supposons de plus d'un agent ne peut initier qu'au mieux une seule interaction par pas de temps.

Bien que les études soient menées dans ce cadre précis, l'approche de conception que nous proposons par la suite est plus générale. Elle peut être utilisée en particulier dans des simulations reposant sur une représentation du temps par évènements discrets. Nous imposons uniquement qu'un agent ne soit capable d'initier qu'une seule interaction à la fois.

Les expériences réalisées dans cette section sont volontairement simples afin d'identifier clairement là où les erreurs surviennent. Ces problèmes surviennent aussi dans des simulations plus complexes.

Étude d'une erreur liée à la participation simultanée à plusieurs interactions

Dans le cas d'étude de cette section, nous caractérisons les limites de l'implémentation naïve de l'unité d'activation, qui est utilisée dans une grande partie des simulations multi-agents. Nous dégageons de ces limites un algorithme alternatif à l'algorithme naïf permettant de dépasser les limites rencontrées.

Modèle utilisé Cette expérience étudie l'évolution des populations d'un écosystème contenant de l'herbe et des moutons. Dans cet écosystème, l'environnement est un espace continu, torique en deux dimensions, et organisé en parcelles carrées de côté 1. Chaque parcelle $\mathcal{P}$ dispose d'un attribut quantité d'herbe $q(\mathcal{P})$ qui augmente d'une unité chaque fois que l'environnement évolue. Lorsque $q(\mathcal{P}) > 0$, on considère que $\mathcal{P}$ contient de l'herbe. Si une parcelle est vidée par un agent, alors $q(\mathcal{P})$ est initialisé à $1 - r_{herb}$, (*i.e.* r_{herb} est le délai nécessaire à l'herbe pour pousser).

Un mouton $\mathcal{M}$ a une énergie $e(\mathcal{M})$ représentant sa santé, et peut effectuer les interactions :

1. Mourir si $e(\mathcal{M}) \leq 0$. Dans ce cas :
   - Il est supprimé de l'environnement.
2. Se reproduire avec un mouton $\mathcal{M}'$ se situant à une distance au plus de 1 si $e(\mathcal{M}) > 0$ et $e(\mathcal{M}') > 0$. Dans ce cas :
   - Un nouveau mouton $\mathcal{M}''$ est créé à la même position que $\mathcal{M}$, avec $e(\mathcal{M}'') = Min(e(\mathcal{M}), e_{repr}) + Min(e(\mathcal{M}'), e_{repr})$. e_{repr} représente l'énergie consommée lors de la reproduction.
   - $e(\mathcal{M})$ et $e(\mathcal{M}')$ sont diminuées de e_{repr}.
   - $\mathcal{M}$, $\mathcal{M}'$ et $\mathcal{M}''$ sont déplacés dans l'environnement en effectuant les mêmes actions que l'interaction Errer (voir 4).
3. Manger l'herbe sur la parcelle où il se situe, s'il y en a. Dans ce cas :
   - Son énergie augmente de e_{mang}, représentant l'énergie récupérée en mangeant.
   - Il vide la parcelle sur laquelle il se situe.
4. Errer. Dans ce cas :
   - Il se tourne d'un angle aléatoire de [-π, π[
   - Il avance de 1.
   - Son énergie diminue de e_{depl}.

La figure 7.2 décrit les différentes priorités que nous fournissons à chacune de ces interactions. Puisque nous nous intéressons ici uniquement au comportement des moutons, nous avons fait le choix de modéliser les parcelles par des attributs de l'environnement plutôt que comme des agents. Ce choix est fait afin que la présence des parcelles ne perturbe pas l'effet de l'unité d'activation sur le comportement des moutons.

Source \ Cible	$\emptyset$	Mouton
Mouton	(Mourir, p = 5) (Errer, p = 0) (Manger, p = 3)	(Se reproduire, d = 1, p = 4)

Figure 7.2 – Matrice d'interaction raffinée de la première expérience traitant de la participation simultanée à plusieurs interactions.

Dans cette expérience, nous évaluons deux algorithmes d'unité d'activation différents. Le premier est l'algorithme 9. Il correspond à l'implémentation naïve de l'unité d'activation, qui est utilisée dans une grande partie des simulations multi-agents. Le second est l'algorithme 10. Il constitue une alternative

Algorithme 9 : Algorithme naïf de l'unité d'activation, qui est utilisée dans une grande partie des simulations multi-agents. MAX y représente la durée de la simulation, exprimée en nombre de pas de temps.

```
début
    pour i ∈ [1, MAX] faire
        Mise à jour des parcelles de l'environnement;
        pour tous les agents a présents dans l'environnement faire
            ℝ_i(a) ⇐ l'ensemble des tuples réalisables ayant a pour source;
            T ⇐ le tuple sélectionné par l'unité de sélection;
            si T ≠ null alors
                Initier l'interaction représentée par le tuple T;
fin
```

à l'algorithme naïf que nous considérons dans cette expérience. Nous l'appelons « algorithme mono-interaction ».

Algorithme 10 : Algorithme mono-interaction de l'unité d'activation, dans lequel un agent ne participe qu'à une seule interaction par pas de temps, soit en tant que source, soit en tant que cible. MAX y représente la durée de la simulation, exprimée en nombre de pas de temps.

```
début
    pour i ∈ [1, MAX] faire
        Mise à jour des parcelles de l'environnement;
        pour tous les agents a présents dans l'environnement faire
            Marquer l'agent a comme activable;
        pour tous les agents a présents dans l'environnement faire
            ℝ_i(a) ⇐ l'ensemble des tuples réalisables ayant a pour source;
            pour T ∈ ℝ_i(a) faire
                I ⇐ L'interaction associée à T;
                si I est non-dégénérée, alors
                    si a n'est pas activable, ou au moins une cible de T n'est pas activable alors
                        ℝ_i(a) ⇐ ℝ_i(a)\{T};
                sinon
                    si a n'est pas activable alors
                        ℝ_i(a) ⇐ ℝ_i(a)\{T};
            T ⇐ le tuple sélectionné par l'unité de sélection;
            si T ≠ null alors
                Initier l'interaction représentée par le tuple T;
                Marquer l'agent a comme non activable;
                I ⇐ L'interaction associée à T;
                si I est non dégénérée alors
                    Marquer chaque cible de T comme non activable;
fin
```

Cadre expérimental Nous avons simulé ce modèle dans un environnement de dimensions 33×33 contenant :

- 1089 parcelles, dont 30% ont $q(\mathcal{P}) = 0$ et 70% $q(\mathcal{P}) \in]\text{-}r_{herb}, \text{-}1]$.
- 70 moutons tels que $e(\mathcal{M}) = 2 \times e_{repr}$

Nous imposons $r_{herb} = 10$, $e_{repr} = 15$, $e_{depl} = 2$ et $e_{mang} = 7$.

La figure 7.3 compare l'évolution de la population en moutons de ce modèle implémenté en utilisant respectivement les algorithmes *naïf* (voir algorithme 9) et *mono-interaction* (voir algorithme 10) en tant qu'unité d'activation, et en posant $MAX = 5000$.

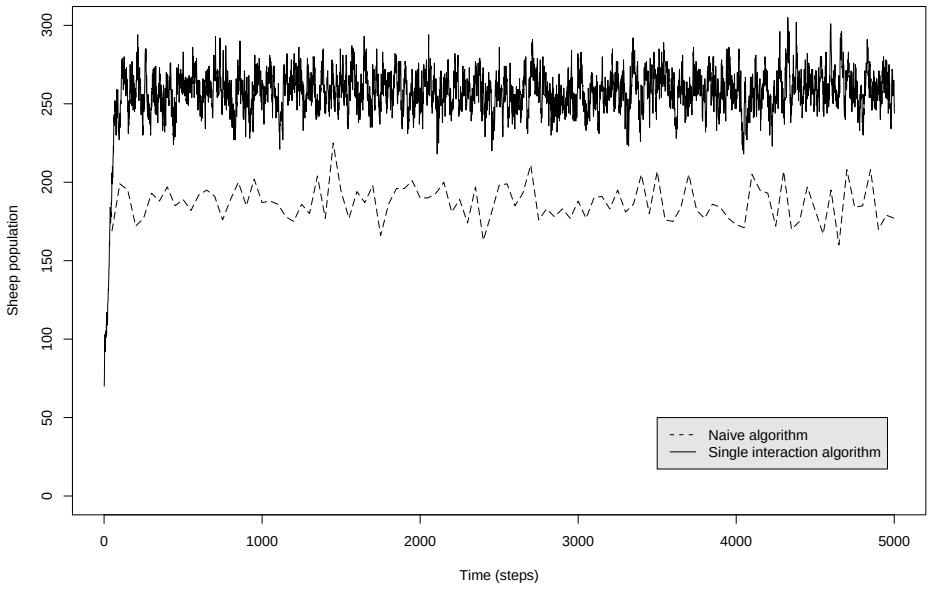


FIGURE 7.3 – Évolution de la population en Moutons en fonction du temps (exprimé en pas de temps) dans le cas de la première expérience traitant de la participation simultanée à plusieurs interactions. La courbe affichée en pointillés représente les résultats obtenus avec l'algorithme *naïf* (voir figure 9) en tant qu'unité d'activation. La courbe affichée en train plein représente les résutats obtenus avec l'algorithme *mono-interaction* (voir figure 10) en tant qu'unité d'activation.

Analyse des résultats L'algorithme de l'unité d'activation influe nettement les résultats de simulation : la population moyenne en moutons est d'environ 188 pour l'algorithme 9, et de 256 pour l'algorithme 10. Dans cette expérience, puisque le sujet d'étude est l'évolution des populations, la différence de résultats fait état d'une erreur ou d'un artefact. L'analyse des unités d'activation nous permet alors d'identifier le choix de conception à l'origine de cette différence.

Dans l'algorithme 9 (l'algorithme naïf), rien n'empêche un mouton $\mathcal{M}$ d'être la source d'une interaction de reproduction avec un autre mouton $\mathcal{M}'$ pour cible, puis d'être lui-même la cible d'autres interactions de reproduction au cours du même pas de temps. Il est même possible que $\mathcal{M}'$ soit en plus la source d'une interaction de reproduction avec $\mathcal{M}$ pour cible ! À l'opposé, dans l'algorithme 10 (l'algorithme mono-interaction), un mouton ne peut participer à une interaction que s'il est *activable*. De plus, une fois qu'il participe à une interaction en tant que source ou en tant que cible, il devient *non activable* jusqu'au début du prochain pas de temps. Par conséquent, un agent ne peut participer lors d'un pas de temps qu'à une seule interaction, que cela soit en tant que source ou en tant que cible. Les résultats obtenus s'expliquent alors par le fait que l'algorithme naïf permet aux agents de se reproduire beaucoup plus, et favorise donc d'une part la baisse de leur énergie (et donc leur mort), mais accentue aussi la compétition entre moutons pour l'accès à la nourriture, ce qui favorise les famines. Dans l'autre implémentation, la reproduction moins intensive entre moutons leur permet un meilleur partage des ressources, ce qui explique que la population moyenne de moutons dans ce cas est plus élevée qu'avec l'algorithme naïf.

Le fait qu'un mouton puisse se reproduire plusieurs fois lors d'un pas de temps se révèle gênant

dans certains cas (par exemple dans le cas de la simulation décrite par Lawson [LP00]). Le nombre d'interactions auxquelles un agent peut participer simultanément au cours d'un pas de temps doit donc être restreint.

Étude d'une erreur liée à la restriction à une unique interaction

Dans la section précédente, nous avons caractérisé un algorithme mono-interaction permettant de restreindre à une le nombre d'interactions auxquelles peut participer un agent dans un pas de temps. Cet algorithme seul n'est toutefois pas suffisant pour modéliser toute simulation où un agent peut initier une seule interaction. En effet, il est par exemple impossible de reproduire l'unité d'activation naïve.

Dans le cas d'étude de cette section, nous caractérisons plus précisément les limites de l'implémentation mono-interaction de l'unité d'activation, et nous en dégageons un algorithme d'unité d'activation n'y étant pas sujette.

Modèle utilisé L'expérience que nous considérons ici consiste à introduire une nouvelle espèce animale `Loup` à l'expérience que nous avons définie dans la section précédente. Dans cette nouvelle expérience, un loup est un animal pouvant uniquement se déplacer dans l'environnement. Le comportement des moutons est changé en présence d'un loup, afin qu'ils puissent le fuir s'il se trouve être trop proche d'eux. Afin d'assurer leur survie, les moutons préféreront fuir plutôt que d'initier l'interaction de reproduction. Dans cette simulation, nous nous attendons donc à voir apparaître une zone vide autour des loups.

Le modèle utilisé ici est donc similaire à celui de l'expérience précédente, mais un agent loup y est ajouté, et un mouton $\mathcal{M}$ peut de plus y effectuer une nouvelle interaction :

5. Fuir un loup $\mathcal{L}$ se situant à une distance au plus de 10. Dans ce cas :
   - $\mathcal{M}$ se tourne pour faire dos à $\mathcal{L}$.
   - $\mathcal{M}$ avance de 1.
   - L'énergie de $\mathcal{M}$ diminue de e_{depl}

Les agents loup ne font que se déplacer en utilisant l'interaction Errer.

La figure 7.4 décrit les différentes priorités que nous fournissons à chacune de ces interactions. Dans cette matrice, nous spécifions qu'un mouton décide de se reproduire avec un autre mouton uniquement s'il n'avait pas à fuir un loup situé à une distance d'au plus 10.

Source \ Cible	$\emptyset$	Mouton	Loup
Mouton	(Mourir, p = 5) (Errer, p = 0) (Manger, p = 2)	(Se reproduire, d = 1, p = 3)	(Fuir, d = 10, p = 4)
Loup	(Errer, p = 0)		

Figure 7.4 – Matrice d'interaction raffinée de la seconde expérience traitant de la participation simultanée à plusieurs interactions.

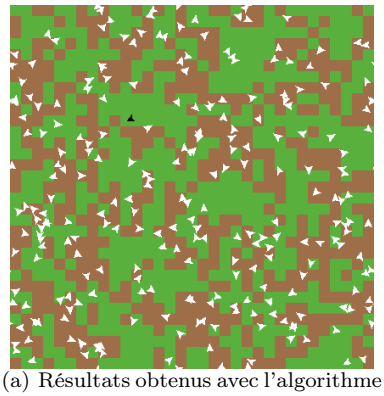
Dans cette expérience, nous évaluons deux algorithmes d'unité d'activation différents. Le premier est l'algorithme mono-interaction 10, et le second est l'algorithme 11, que nous appelons *algorithme à interactions parallèles*. Ce second algorithme base le fonctionnement de l'unité d'activation sur l'attribution d'une classe parmi $\{\mathcal{I}_e, \mathcal{I}_p\}$ aux différentes interactions de la simulation. Dans le cadre de cette expérience, nous attribuons la classe $\mathcal{I}_p$ à l'interaction Fuir , et la classe $\mathcal{I}_e$ aux autres interactions.

Cadre expérimental L'initialisation de cette expérience se fait de la même manière que dans l'expérience précédente, en ajoutant toutefois un unique loup dans la simulation, placé initialement à une position déterminée aléatoirement.

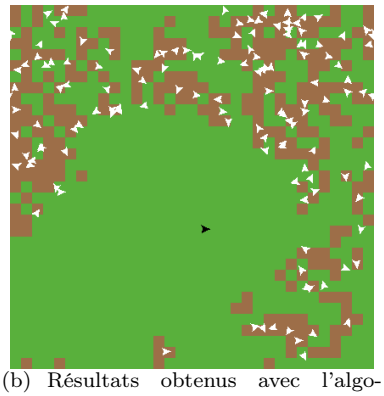
Nous évaluons alors les résultats de l'algorithme mono-interaction et de l'algorithme à interactions parallèles en observant le positionnement des différents agents de la simulation après l'exécution de 500 pas de temps. Ces informations sont résumées par les captures d'écran de la figure 7.5.

Algorithme 11 : Algorithme à interactions parallèles de l'unité d'activation. Dans cet algorithme, un agent ne peut initier qu'une interaction par pas de temps, ne peut participer qu'à une seule interaction de classe $\mathcal{I}_e$ par pas de temps, mais peut être la cible d'autant d'interactions de classe $\mathcal{I}_p$ que nécessaire. MAX y représente la durée de la simulation, exprimée en nombre de pas de temps.

```
début
    pour i ∈ [1, MAX] faire
        Mise à jour des parcelles de l'environnement;
        pour tous les agents a présents dans l'environnement faire
            Marquer l'agent a comme activable;
        pour tous les agents a présents dans l'environnement faire
            ℝ_i(a) ⇐ l'ensemble des tuples réalisables ayant a pour source;
            pour T ∈ ℝ_i(a) faire
                I ⇐ L'interaction associée à T;
                si I est non-dégénérée alors
                    si I a pour classe ℐ_e alors
                        si a ou une des cibles de T n'est pas activable alors
                            ℝ_i(a) ⇐ ℝ_i(a)\{T};
                    sinon
                        si I a pour classe ℐ_p alors
                            si a n'est pas activable alors
                                ℝ_i(a) ⇐ ℝ_i(a)\{T};
                sinon
                    si a n'est pas activable alors
                        ℝ_i(a) ⇐ ℝ_i(a)\{T};
            T ⇐ le tuple sélectionné par l'unité de sélection;
            I ⇐ L'interaction associée à T;
            si T ≠ null alors
                Initier l'interaction représentée par le tuple T;
                si I est non-dégénérée alors
                    si I a pour classe ℐ_e alors
                        Marquer l'agent a comme non activable;
                        Marquer chaque cible de T comme non activable;
                    sinon
                        si I a pour classe ℐ_p alors
                            Marquer l'agent a comme non activable;
                sinon
                    Marquer l'agent a comme non activable;
fin
```



(a) Résultats obtenus avec l'algorithme mono-interaction



(b) Résultats obtenus avec l'algorithme à interactions parallèles

FIGURE 7.5 – Résultats de l'étude d'une erreur liée à la restriction à une unique interaction. La capture d'écran gauche illustre les résultats obtenus avec l'algorithme mono-interaction (*c.f.* algorithme 10). Celle de droite illustre les résultats obtenus avec l'algorithme à interactions parallèles (*c.f.* algorithme 11). Dans l'algorithme mono-interaction, un `Loup` (triangle noir) n'est FUI que par un seul `Mouton` (triangle blanc) par pas de temps. Par conséquent, la zone vide attendue autour du `Loup` n'apparaît pas, contrairement aux résultats obtenus avec l'algorithme à interactions parallèles.

Analyse des résultats La capture d'écran de la figure 7.5(a) ne fait pas apparaître la zone vide autour du loup. Les résultats attendus ne sont donc pas obtenus, ce qui implique que l'utilisation de l'algorithme mono-interaction induit des erreurs ou des artefacts dans ces simulations. Ce problème est dû au nombre d'interactions FUIR qu'un loup peut subir au cours d'un même pas de temps. Dans l'algorithme mono-interaction, puisqu'un agent est au mieux la cible d'une seule interaction par pas de temps, un loup n'est fuit que par un seul mouton par pas de temps. Les autres moutons se comportent alors comme si le loup n'existait pas.

La restriction fournie par l'algorithme mono-interaction est donc trop forte pour simuler cet exemple. Toutefois, nous ne pouvons pas non plus envisager l'implémentation naïve, puisque même si elle résout le problème lié à la fuite, le problème lié à la reproduction persiste. Un traitement spécifique doit donc être effectué selon les interactions.

L'algorithme à interactions parallèles repose sur ce principe, et fournit à chaque interaction une classe $\mathcal{I}_e$ ou $\mathcal{I}_p$, qui permet de traiter spécifiquement les deux cas : les interactions de classe $\mathcal{I}_e$, déjà présentes dans la première expérience, imposent des contraintes fortes à propos des agents source et cibles, et les interactions de classe $\mathcal{I}_p$ n'imposent des contraintes que sur la source. Les résultats obtenus en utilisant cet algorithme, résumés par la capture d'écran sur la figure 7.5(b), font bien apparaître la zone vide autour du loup, qui est le résultat attendu de cette simulation. Les résultats attestent donc que l'algorithme à interactions parallèles est approprié pour simuler ce phénomène.

7.2.3 Comment gérer les interactions simultanées ?

Au travers des deux expériences présentées ici, nous avons fait apparaître la nécessité de fournir une réponse explicite à la question « la source (ou la cible) d'une interaction est-elle autorisée à être en même temps la source ou la cible d'une autre interaction ? ». Dans le cas contraire, le manque de spécifications aboutit aux erreurs et artefacts que nous avons illustrés dans les deux expériences.

La solution que nous proposons au problème de la simultanéité des interactions dans une simulation repose sur les concepts d'interaction, d'agent initiant une interaction et d'agent subissant une interaction. Nous avons donc naturellement choisi de la concrétiser comme une extension du corps de l'approche IODA présentée dans la partie II.

Notre approche repose sur l'attribution de classes aux interactions [KMP08b]. Ces dernières sont exploitées par l'unité d'activation, afin de déterminer quelles interactions peuvent être initiées par un

agent à un instant t de la simulation, en fonction des interactions déjà en cours au moment où la parole est donnée à cet agent.

Afin d'utiliser cette façon de modéliser, le concepteur doit faire explicitement deux choix qui fixeront de manière explicite les hypothèses de simulation émises. Le premier choix consiste à déterminer la représentation du temps utilisée, et plus particulièrement quel sens prend l'expression « interactions déjà en cours au moment où la parole est donnée à cet agent ». Le second choix consiste à attribuer à chaque interaction non dégénérée de la simulation une des classes que nous décrivons par la suite dans cette section.

Quel sens donner à « interactions déjà en cours au moment où la parole est donnée à cet agent » ?

Le sens de l'expression « interactions déjà en cours au moment où la parole est donnée à cet agent » dépend de la façon dont l'unité d'activation donne la parole aux agents, et donc de la façon dont le temps est représenté dans la simulation.

Dans le cas d'une modélisation du temps par évènements discrets, par exemple avec DEVS [ZKP00], la durée d'une interaction est mentionnée explicitement dans le modèle, puisqu'elle est calculée afin de déterminer la prochaine date où l'agent aura la parole. Dès lors, « interactions déjà en cours au moment où la parole est donnée à cet agent » se réfère à toutes les interactions initiées à un temps $t' < t$ et n'étant toujours pas terminées au temps t (*i.e.* leur durée est supérieure à $t - t'$).

Dans le cas d'une simulation en temps discret reposant sur des pas de temps, deux approches sont utilisées, ce qui implique que l'expression « interactions déjà en cours au moment où la parole est donnée à cet agent » peut prendre deux sens.

La première approche consiste à simuler le parallélisme en demandant dans un premier temps à tous les agents de sélectionner l'interaction qu'ils souhaitent initier et, dans un second temps, à appliquer l'effet de ces interactions. Cette seconde partie consiste plus précisément à déterminer :

- Quelles interactions sélectionnées par les agents sont indépendantes de toutes les autres et peuvent être initiées telles quelles ;
- Quelles interactions sélectionnées par les agents sont concurrentes et nécessitent donc un traitement spécifique.

Dans un tel cas, puisque toutes les interactions sont initiées en même temps, l'ensemble des « interactions déjà en cours à l'instant t » est toujours vide. La solution que nous décrivons dans cette section n'est donc pas utile dans ce cas là.

La seconde approche consiste à donner successivement la parole une fois à chaque agent, en suivant une séquence réordonnée au début de chaque pas de temps. Dans ce cas, l'ensemble des « interactions déjà en cours à l'instant t » au moment où la parole est donnée à un agent correspond à l'ensemble des interactions initiées par tous les agents se situant avant cet agent dans la séquence de prise de parole.

Le premier choix explicite devant être fait par les concepteurs consiste à déterminer quelle représentation du temps ils utilisent parmi les trois représentation mentionnées, et à pleinement comprendre l'implication cette représentation sur le sens de l'expression « interactions déjà en cours au moment où la parole est donnée à cet agent ».

Définition des classes d'interaction

Le second choix devant être fait explicitement par les concepteurs consiste à attribuer une classe aux interactions non dégénérées de la simulation. Nous définissons à cet effet deux classes d'interactions, que nous appelons *interactions exclusives* et *interactions parallèles*.

Une *interaction exclusive* est une interaction imposant de fortes contraintes sur sa source et ses cibles.

Définition 38. ***Interaction exclusive***
Une **interaction exclusive** est une interaction ne pouvant être initiée par un agent source S avec un (ou des) agent(s) cible(s) C que si les deux conditions suivantes sont vérifiées :
1. S et C ne sont pas la source d'une interaction déjà en cours ;
2. S et C ne sont pas la cible d'une interaction exclusive déjà en cours.

$\Big)$ *ET*

Un agent peut donc au mieux être impliqué dans une seule interaction exclusive à un instant t. De plus, tant qu'il est impliqué dans une interaction exclusive, un agent ne peut pas initier d'autres interactions, et tant qu'il est la source d'une interaction, il ne peut être impliqué dans une interaction exclusive.

Exemple.
Par exemple, un animal n'est impliqué que dans une seule interaction Se Reproduire *à un instant t. De plus, si un animal est en train de manger (i.e. il est la source d'une interaction* Manger *en cours), alors il ne peut pas se reproduire (i.e. il ne peut être ni la source, ni la cible d'une interaction* Se Reproduire*). Inversement, si un agent est en train de se reproduire (i.e. il est la source ou la cible d'une interaction* Se Reproduire *en cours), alors il ne peut pas manger (i.e. il ne peut pas être la source d'une interaction* Manger*).*

Se Reproduire *est donc une interaction exclusive.*

Une *interaction parallèle* est une interaction n'imposant de fortes contraintes que sur sa source.

Définition 39. ***Interaction parallèle***
Une **interaction parallèle** est une interaction ne pouvant être initiée par un agent source S avec un (ou des) agent(s) cible(s) C que si les deux conditions suivantes sont vérifiées :
1. S n'est pas la source d'une interaction déjà en cours ;
2. S n'est pas la cible d'une interaction exclusive déjà en cours.

$\Big)$ *ET*

Un agent peut donc être impliqué en tant que cible dans un nombre arbitraire d'interactions parallèles. Par contre, un agent ne peut être la source d'une interaction parallèle tant qu'il est la source d'une autre interaction, ou tant qu'il est la cible d'une interaction exclusive.

Exemple.
Par exemple, un loup peut être la cible d'une interaction Fuir *autant de fois que nécessaire à un instant t. Toutefois, si un mouton est en train de manger, il ne peut pas fuir un loup. De même, tant qu'un autre mouton est en train de se reproduire avec lui, un mouton ne peut pas fuir un loup.*

Fuir *est donc une interaction parallèle.*

Intégration dans la méthodologie

L'identification de la classe d'une interaction I consiste à répondre à l'une des deux questions qui suivent :
- une cible de I peut-elle être la source d'une autre interaction lors du même pas de temps ?
- un agent peut-il être deux fois la cible de l'interaction I avec des sources différentes lors du même pas de temps ?

Si la réponse à l'une des deux questions est oui, alors l'interaction est parallèle. Dans le cas contraire, il s'agit d'une interaction exclusive.

Implémentation dans une unité d'activation

L'algorithme permettant d'exploiter les classes d'interaction dans une unité d'activation repose sur les classes d'interaction, et la notion d'agent activable.

Un agent est défini comme activable si sa participation à toute interaction n'est pas restreinte. Un agent activable est donc caractérisé par le fait qu'il n'est la source d'aucune interaction en cours, et qu'il n'est la cible d'aucune interaction exclusive en cours. Un agent est défini comme non activable s'il est la source d'une interaction en cours, ou s'il est la cible d'une interaction exclusive en cours.

Les interactions auxquelles peut participer un agent dépendent alors de sa nature activable ou non activable. Un agent activable peut par définition participer à n'importe quelle interaction en tant que source ou en tant que cible. Un agent non activable ne peut être que la cible d'interactions parallèles. L'algorithme général permettant d'implémenter l'unité d'activation prend donc la forme de deux fonctions décrites dans l'algorithme 12. La première permet de savoir si un agent est activable ou non, et la seconde décrit ce en quoi consiste la prise de parole d'un agent. Cet algorithme est indépendant de la façon dont la parole est donnée à un agent, et peut aussi bien être utilisé dans une représentation du temps par

Algorithme 12 : Algorithme général permettant d'utiliser les classes d'interaction définies dans la section 7.2.3 afin de résoudre les problèmes liés à la participation simultanée à plusieurs interactions. Cet algorithme décrit le comportement de l'unité d'activation lorsque la parole est donnée à un agent a. Il repose sur la constitution de deux ensembles $\mathbb{S}(a)$ et $\mathbb{T}(a)$, contenant respectivement les interactions dans lesquelles a est la source, et les interactions dans lesquelles a est la cible.

```
estActivable(a)
début
    si 𝕊(a) ≠ ∅ alors
        retourner faux;
    sinon
        pour I ∈ 𝕋(a) faire
            si I est de classe exclusive alors
                retourner faux;
        retourner vrai;
fin
donnerParole(a)
début
    si ¬estActivable(a) alors
        % Si a n'est pas activable, il ne peut pas initier d'interactions
    sinon
        ℝz(a) ⇐ le potentiel d'interaction de a;
        % On retire de ℝz(a) les interactions que les voisins non activables de a ne peuvent pas subir.
        pour T ∈ ℝz(a) faire
            I ⇐ l'interaction associée à T;
            si I n'est pas dégénérée alors
                si il existe une cible contenue dans T étant non activable alors
                    si I est une interaction exclusive alors
                        ℝz(a) ⇐ ℝz(a)\{T};
        a sélectionne un tuple réalisable T dans ℝz(a) avec son modèle de sélection d'interaction;
        si T ≠ null alors
            I ⇐ l'interaction associée à T;
            𝕊(a) ⇐ 𝕊(a) ∪ {I};
            si I est exclusive alors
                pour chaque cible c contenue dans T faire
                    𝕋(c) ⇐ 𝕋(c) ∪ {I};
fin
```

évènements discrets, que dans une représentation par pas de temps, où les agents agissent en séquence. L'algorithme 11 de la page 198 constitue d'ailleurs un exemple d'implémentation de ce principe (utilisant des optimisations) pour les simulations en temps discret où les agents agissent en séquence.

L'algorithme défini ici permet de reproduire le comportement des trois algorithmes décrits dans les cas d'étude. L'algorithme à interactions parallèles est une implémentation de l'algorithme 12 optimisée pour les simulations en temps discret où les agents ont la parole séquentiellement, où la classe $\mathcal{I}_e$ correspond aux interactions exclusives, et la classe $\mathcal{I}_p$ correspond aux interactions parallèles. L'algorithme mono-interaction (*i.e.* l'algorithme 10 de la page 195) est équivalent un l'algorithme à interactions parallèles dans lequel toutes les interactions sont exclusives. Enfin, l'algorithme naïf (*i.e.* l'algorithme 9 de la page 195) est équivalent un l'algorithme à interactions parallèles dans lequel toutes les interactions sont parallèles. Par conséquent, l'utilisation des classes d'interactions que nous avons définies dans cette section permet de modéliser tous les cas étudiés ici, et permet de faire apparaître explicitement des choix de conception concernant l'unité d'activation.

7.3 Spécification de comportements stochastiques

Les simulations que nous considérons dans cette thèse ont pour principal objectif d'expliquer l'apparition de phénomènes émergents par les interactions entretenues par les agents, et par le comportement de ces agents. Dans ce cadre, les modèles équationnels ont un intérêt moindre, car ils ne permettent que de prédire le comportement macroscopique du système simulé, et fournissent donc uniquement des résultats quantitatifs, alors que l'objectif de simulations explicatives est d'obtenir des résultats qualitatifs sur le fonctionnement du modèle étudié. Toutefois, les modèles équationnels ne doivent pas pour autant être totalement écartés lors de la conception de simulations explicatives. En effet, ces derniers peuvent être utilisés comme bases statistiques pour construire un modèle multi-agents. Cette approche favorise l'obtention de résultats similaires au modèle équationnel (dont la validité est déjà prouvée), et favorise donc la construction de modèles valides. De plus, le modèle laisse alors place à l'ajout de nouvelles hypothèses de fonctionnement, ce qu'il n'est pas possible de faire dans le modèle équationnel.

Concevoir un modèle multi-agents en se basant sur un modèle équationnel est complexe en soi, car cela nécessite de traduire des lois portant sur l'évolution d'aspects macroscopiques de la simulation en lois régissant le comportement individuel de chaque agent. Par exemple, dans le modèle de Lotka-Volterra [J.L25, Vol28], cela consiste à utiliser les équations décrivant l'évolution des populations de plusieurs espèces animales afin de décrire le comportement de chaque espèce animale. Cette traduction nécessite de plus d'intégrer des données liées au positionnement des agents, qui au pire ne sont pas modélisées dans les équations, et au mieux sont modélisées sous la forme d'équations exprimant le nombre d'entités se diffusant d'une parcelle de l'environnement à l'autre. Dans ce cas, la pratique courante consiste à baser le modèle de sélection d'interaction des agents sur des probabilités extraites de l'étude du modèle équationnel. Tout le problème d'une telle approche revient alors à identifier quelles probabilités utiliser et comment intégrer ces probabilités au comportement des agents.

7.3.1 Problématique traitée

Comme nous l'avons mentionné ci-avant, le problème sous-jacent à la construction d'un modèle multi-agents à l'aide d'un modèle équationnel est double. Il faut dans un premier temps pouvoir extraire du modèle équationnel les probabilités régissant le comportement des agents. Cela nécessite une analyse fine des équations afin de faire apparaître les agents utilisés pour modéliser le phénomène, et comment utiliser les différents coefficients et facteurs des équations afin d'exprimer les probabilités utilisées dans le modèle multi-agents.

Exemple.
Par exemple, les équations de Lokta-Volterra [J.L25, Vol28] permettant de modéliser l'évolution de la population de deux espèces x et y dans un écosystème, où une espèce x est la proie d'une espèce y. Ces

équations prennent la forme :

$$\frac{dx(t)}{dt} = x(t)\Big(\alpha - \beta y(t)\Big)$$
$$\frac{dy(t)}{dt} = -y(t)\Big(\gamma - \delta x(t)\Big)$$

Dans ces équations, γ représente le taux de mortalité des prédateurs en absence de proies. Par conséquent, l'interaction Mourir *est initiée par un prédateur selon une probabilité exprimée en fonction de la valeur de γ.*

Quelles probabilités utiliser ?

Actuellement, aucune méthodologie ne supporte complètement ce procédé complexe. Certains travaux préliminaires à ce propos sont toutefois particulièrement prometteurs. C'est par exemple le cas de la méthodologie présentée par Nguyen *et al.* [NDA08], qui propose de construire un modèle multi-agents en deux phases :

1. introduire la spatialité dans le modèle équationnel ;
2. distribuer les paramètres présents dans les équations aux différents agents.

Ce problème n'est pas étudié dans cette thèse.

Comment intégrer les probabilités au comportement des agents ?

Le second problème consiste à déterminer comment exploiter dans le modèle comportemental des agents les probabilités extraites du modèle équationnel. À ce jour, différentes approches permettent de décrire des comportements stochastiques. On peut en particulier mentionner :

- les processus de décision markoviens qui sont dédiés, entre autres, à la description de processus de sélection d'action stochastiques ;
- des architectures multi-agents telles qu'InteRRaP [MP94] ou l'algorithme de Gillespie [Gil76]. Dans ces approches, il est possible de décrire un choix parmi un ensemble d'alternatives en pondérant chaque alternative et en effectuant une sélection pondérée de l'alternative.

Toutefois, ces approches se fondent sur les actions qu'un agent est capable d'initier pour prendre ses décisions plutôt que sur la notion d'interaction. Le choix de la cible d'une interaction n'y apparaît donc que de manière implicite.

Dans cette section, nous proposons d'étudier l'importance de la prise en compte explicite de la cible des interactions lors de description du modèle de sélection d'interaction, et mettons en exergue différents choix de conception pouvant survenir volontairement ou involontairement lors de cette description. Puisque le cas des agents cognitifs est mieux traité que celui des agents réactifs, nous portons ici l'étude aux modèles de comportement réactifs. Pour cela, nous analysons différentes façons d'implémenter l'unité de sélection d'un modèle dont l'objectif est de reproduire les deux réactions chimiques décrites dans la section 7.3.2. Cette analyse se base sur deux cas d'étude, confrontant chacun deux manières d'implémenter le comportement stochastique recherché. Nous dégageons de ces analyses des principes de sélection d'interaction différents, reflétant des choix de conception différents. Nous soutenons que le concepteur doit mentionner explicitement et sciemment ces différents choix, afin d'éviter autant d'erreurs que possible dans le modèle.

7.3.2 Illustration des biais

Nous considérons une simulation décrivant deux réactions chimiques. Dans de telles simulations, le comportement des agents se résume presque entièrement par des règles, s'interprétant comme des équations différentielles portant sur la concentration des espèces chimiques. Dans nos cas d'étude, nous considérons les réactions chimiques qui suivent (décrites à gauche), dont l'interprétation en termes d'équations

est décrite sur la droite [19] :

$$\begin{array}{ll} \text{(R1)} & \mathrm{A} + \mathrm{B} \xrightarrow{k_1} \mathrm{C} \\ \text{(R2)} & \mathrm{A} + \mathrm{D} \xrightarrow{k_2} \mathrm{E} \end{array} \qquad \left\{ \begin{array}{rcl} -\frac{d[B]}{dt} & = & k_1 \times [A] \times [B] \\ -\frac{d[D]}{dt} & = & k_2 \times [A] \times [D] \\ \frac{d[A]}{dt} & = & \frac{d[B]}{dt} + \frac{d[D]}{dt} \\ \frac{d[C]}{dt} & = & -\frac{d[B]}{dt} \\ \frac{d[E]}{dt} & = & -\frac{d[D]}{dt} \end{array} \right.$$

La réaction $R1$ spécifique qu'une molécule A peut se combiner avec une molécule B afin de produire une molécule C. Dans cette réaction, k_1 est une constante de réaction décrivant la vitesse à laquelle la réaction a lieu. Afin de simplifier nos propos, nous considérons que la constante de réaction k_1 dans la réaction chimique $R1$ représente aussi la probabilité que cette réaction survienne lors de la simulation. Dans le cas général, ce postulat est faux, mais il nous permet de nous détacher du premier problème mentionné dans la section précédente, qui sort du cadre de notre étude.

Dans les deux sous-sections qui suivent, nous caractérisons différentes façons d'implémenter l'unité de sélection de l'espèce chimique A. Dans chaque sous-section, nous confrontons deux implémentations de l'unité de sélection, et analysons leurs différences afin de dégager différents principes de sélection (et donc différents choix de conception) d'une interaction et de sa cible. Les implémentations étudiées sont pour la plupart issues de la librairie de modèles fournie par la plateforme de simulation Netlogo [WC99].

Contrairement à la section 7.2, où nous avions effectué les simulations et observé les résultats obtenus pour faire état de différences, nous nous focalisons ici sur l'analyse formelle des comportement spécifiés.

Étude d'un choix lié à l'interprétation d'une règle macroscopique

Dans cette section, nous illustrons un premier choix de conception pouvant aboutir à deux implémentations de l'unité de sélection fournissant des résultats différents, alors qu'ils se reposent tous deux sur la même constante de réaction. Nous supposons pour cela que la simulation que nous modélisons ne contient que la réaction $R1$. Nous cherchons alors à modéliser le comportement d'un agent A.

D'après les équations, le comportement des instances de la famille d'agents `A` consiste à initier l'interaction ReactionR1 avec une instance de la famille d'agents `B` pour cible. La constante de réaction k_1 représente la probabilité que la réaction ait lieu dans la simulation. Cette spécification peut donner lieu à deux implémentations différentes de l'unité de sélection, que nous présentons respectivement sur les algorithmes 13(a) et 13(b).

Interprétation des deux implémentations. Posons $P(R1)$ la probabilité que l'interaction R1 soit initiée par un agent (avec une cible indifférente), $P(\overline{R1})$ la probabilité que l'interaction R1 ne soit pas initiée par un agent, et n_B le nombre d'instances de la famille d'agents `B` situés à proximité de l'agent effectuant la sélection d'interaction. Dans notre étude, nous supposons $n_B \neq 0$.

Dans la première implémentation de l'unité de sélection (voir algorithme 13(a)), le test de probabilité portant sur k_1 n'est effectué qu'une seule fois pour toutes les cibles potentielles de l'interaction ReactionR1. Si cette probabilité est vérifiée, alors la cible de l'interaction est choisie aléatoirement parmi les instances de la famille d'agents `B` situés à proximité de l'agent effectuant la sélection d'interaction. Par conséquent, dans cet exemple, $P(R1) = \frac{k_1}{100}$ et $P(\overline{R1}) = 1 - k_1$.

Dans la seconde implémentation de l'unité de sélection (voir algorithme 13(b)), le test de probabilité portant sur k_1 est effectué une fois par cible potentielle de l'interaction Reaction1. L'interaction ReactionR1 n'est initiée que si au moins une cible a vérifié la probabilité. La cible de la réaction est alors choisie aléatoirement parmi les cibles ayant vérifié la probabilité. Par conséquent, le seul cas dans lequel l'interaction ReactionR1 n'a pas lieu est celui où toutes les n_b instances de la famille d'agents `B` ne vérifient pas la probabilité. Ainsi, dans cet exemple, $P(R1) = 1 - (1 - k_1)^{n_b}$ et $P(\overline{R1}) = (1 - k_1)^{n_b}$.

En résumé, on a donc :

19. dans ces équations, $[A]$ représente la concentration de l'espèce chimique A dans l'environnement

Algorithme 13 : Algorithmes de deux implémentations différentes de l'unité de sélection décrivant comment un agent de l'espèce chimique A détermine s'il effectue la réaction chimique ReactionR1. On suppose que $random([0;1[)$ retourne un nombre à virgule flottante de l'intervalle $[0;1[$.

```
selection(a)
début
    V ⇐ l'ensemble des instances de la famille d'agents B se situant à une distance de 0 de a;
    si V ≠ null alors
        si random([0;1[) < k1 alors
            c ⇐ un élément de V pris au hasard;
            retourner (ReactionR1, a, c);
        sinon
            retourner null;
    sinon
        retourner null;
fin
```

(a) Premier algorithme de l'unité de sélection

```
selection(a)
début
    V ⇐ l'ensemble des instances de la famille d'agents B se situant à une distance de 0 de a;
    T ⇐ ∅;
    si V ≠ null alors
        pour c ∈ V faire
            si random([0;1[) < k1 alors
                T ⇐ T ∪ {c};
        si T ≠ ∅ alors
            retourner (ReactionR1, a, c);
        sinon
            retourner null;
    sinon
        retourner null;
fin
```

(b) Second algorithme de l'unité de sélection

	Algorithme 13(a)	Algorithme 13(b)
$P(R1)$	k_1	$1-(1-k_1)^{n_b}$
$P(\overline{R1})$	$1-k_1$	$(1-k_1)^{n_b}$

Les deux implémentations représentent donc bien deux implémentations différentes de l'unité de sélection. Leur différence est liée à l'interprétation donnée à la probabilité k_1. Dans la première unité de sélection, k_1 représente la probabilité d'initier l'interaction ReactionR1 indépendamment du nombre d'agents pouvant subir cette interaction dans le voisinage. Dans la seconde unité de sélection, k_1 représente la probabilité d'initier l'interaction ReactionR1 avec une cible particulière.

Pour déterminer l'unité de sélection appropriée à la simulation que nous cherchons à décrire, et donc déterminer quel choix de conception effectuer, il nous faut alors comparer les probabilités que nous venons de calculer avec la probabilité théorique que l'on peut extraire des équations. Cette comparaison sort du contexte de cette thèse et nous n'en donnons donc pas les détails. Si l'on compare les deux implémentations de l'unité de sélection avec l'implémentation de modèles similaires et validés, l'unité de sélection se révélant être la plus appropriée est la première, décrite dans l'algorithme 13(a).

Extraction des principes de sélection d'interaction utilisés Nous pouvons extraire plusieurs principes de sélection différents à partir des deux implémentations de l'unité de sélection étudiées ici.

La première unité de sélection fonctionne en deux temps. Elle consiste dans un premier temps à déterminer si l'interaction ReactionR1 est sélectionnée (si la probabilité est vérifiée) ou pas (si la probabilité n'est pas vérifiée). Dans un second temps, si ReactionR1 est sélectionnée, elle consiste à choisir aléatoirement l'une des instances de la famille d'agents `B` vérifiant les conditions de ReactionR1 (en l'occurrence, être situé à proximité de la source).

La seconde unité de sélection consiste à vérifier pour chaque couple (ReactionR1, agent cible) si la probabilité est vérifiée. Puisque l'interaction ReactionR1 ne peut être initiée avec un agent cible que si cette probabilité est vérifiée, la vérification de la probabilité fait partie des conditions de l'interaction. La sélection fonctionne donc en un seul temps : elle sélectionne aléatoirement un couple (ReactionR1, agent cible) qui vérifie les conditions de l'interaction ReactionR1.

Par conséquent, dans un modèle réactif de sélection d'interaction, au moins deux politiques de sélection peuvent être utilisées. La première consiste à sélectionner une interaction selon certains critères, puis à sélectionner une cible selon d'autres critères. La seconde consiste à sélectionner directement un tuple

(interaction, cible) selon un critère particulier. De plus, chaque sélection effectuée dans une politique peut se faire au moins selon deux natures : soit en effectuant un choix aléatoire, soit en effectuant un choix pondéré. Il est ainsi possible d'effectuer la sélection d'une seule interaction et de ses cibles au moins de six manières différentes.

Étude d'un choix lié à des probabilités conditionnelles

Dans cette section, nous illustrons un second choix de conception, apparaissant de manière implicite lorsqu'il n'y a pas de séparation entre l'unité de définition (qui spécifie les interactions que les agents ont la capacité d'initier ou de subir) et l'unité de sélection (qui spécifie comment sélectionner l'interaction initiée). Pour illustrer ce problème, nous cherchons à modéliser le comportement d'un agent A dans le cas où les deux réactions chimiques $R1$ et $R2$ peuvent avoir lieu.

D'après les équations, le comportement des instances de la famille d'agents `A` consiste à initier l'interaction ReactionR1 avec une instance de la famille d'agents `B` pour cible. Lorsque cette interaction est initiée, ces deux instances sont retirées de l'environnement, et sont remplacées par une instance de la famille d'agents `C`. L'interaction ReactionR2 a un effet similaire, mais en prenant pour cible la famille d'agents `D`, et en remplaçant source et cible par une instance de la famille d'agents `E`. Les constantes de réaction k_1 et k_2 représentent la probabilité que les réactions aient lieu dans la simulation. Puisque les interactions ReactionR1 et ReactionR2 détruisent toutes les deux leur source, nous savons que k_1 et k_2 sont disjointes. Ces spécifications peuvent donner lieu à plusieurs implémentations différentes de l'unité de sélection, dont nous fournissons deux exemples respectivement dans les algorithmes 14(a) et 14(b).

Algorithme 14 : Algorithmes de deux implémentations différentes de l'unité de sélection décrivant comment un agent de l'espèce chimique A détermine s'il effectue les réactions chimiques ReactionR1 ou ReactionR2. On suppose que $random([0;1[)$ retourne un nombre à virgule flottante de l'intervalle $[0;1[$.

```
selection(a)
début
    V_B ⇐ l'ensemble des instances de la famille d'agents B se situant à une distance de 0 de a;
    V_D ⇐ l'ensemble des instances de la famille d'agents D se situant à une distance de 0 de a;
    si V_B ≠ null et random([0;1[) < k_1 alors
        c ⇐ un élément de V_B pris au hasard;
        retourner (ReactionR1, a, c);
    sinon
        si V_D ≠ null et random([0;1[) < k_2 alors
            c ⇐ un élément de V_D pris au hasard;
            retourner (ReactionR2, a, c);
        sinon
            retourner null;
fin
```

(a) Premier algorithme de l'unité de sélection

```
selection(a)
début
    V_B ⇐ l'ensemble des instances de la famille d'agents B se situant à une distance de 0 de a;
    V_D ⇐ l'ensemble des instances de la famille d'agents D se situant à une distance de 0 de a;
    prob ⇐ random([0;1[);
    si prob < k_1 alors
        si V_B ≠ ∅ alors
            c ⇐ un élément de V_B pris au hasard;
            retourner (ReactionR1, a, c);
    sinon
        si prob < k_1 + k_2 alors
            si V_D ≠ ∅ alors
                c ⇐ un élément de V_D pris au hasard;
                retourner (ReactionR2, a, c);
    retourner null;
fin
```

(b) Second algorithme de l'unité de sélection

Interprétation des deux implémentations. Posons $P(R1)$ la probabilité que R1 soit initiée par un agent, p_B la probabilité qu'au moins une instance de la famille d'agents `B` se situe à proximité de l'agent effectuant la sélection d'interaction, et p_D la probabilité qu'au moins une instance de la famille d'agents `D` se situe à proximité de l'agent effectuant la sélection d'interaction.

Dans la première implémentation de l'unité de sélection (voir algorithme 14), le test de probabilité portant sur k_2 n'est fait qu'après le test de probabilité portant sur k_1. De plus, ces tests de probabilité sont indépendants. Par conséquent, cette unité de sélection introduit des probabilités conditionnelles. On a donc $P(R1) = k_1 * p_B$ et $P(R2) = (1 - k_1 * p_B) \times k_2 * p_D$.

Dans la seconde implémentation de l'unité de sélection (voir algorithme 14), le test de probabilité portant sur k_1 et k_2 est fait en divisant l'intervalle $[0, 100[$ en trois parties, de longueurs respectives k_1, k_2 et $1 - k_1 - k_2$. Ainsi, dans cette unité de sélection, $p(R1) = k_1 * p_B$ et $p(R2) = k_2 * p_D$.

En résumé, on a donc :

	Algorithme 14(a)	Algorithme 14(b)
$P(R1)$	$k_1 * p_B$	$k_1 * p_B$
$P(R2)$	$(1 - k_1 * p_B) \times k_2 * p_D$	$k_2 * p_D$

Les deux algorithmes représentent donc bien deux implémentations différentes de l'unité de sélection. Leur différence est liée à la présence dans un cas de probabilités conditionnelles, que l'on ne retrouve pas dans l'autre. Sauf si les probabilités ont été calculées à cet effet lors de l'analyse des équations, la présence de probabilités conditionnelles dans le modèle constitue une erreur. Les biais qu'elle occasionne dépend de la valeur de k_1 : plus sa valeur est élevée, plus le biais sera important.

Puisque, dans la simulation que nous cherchons à décrire, l'implémentation définie par l'algorithme 14(a) contient une erreur, le choix d'implémentation à privilégier ici est évident : il ne faut pas utiliser de probabilités conditionnelles, et il faut donc plutôt reposer sur l'algorithme 14(b).

Extraction des principes de sélection d'interaction utilisés Dans la première unité de sélection, l'interaction ReactionR2 n'est initiée que si l'interaction ReactionR1 ne l'a pas été. Il existe donc une relation d'ordre entre ces interactions : ReactionR1 est plus prioritaire que ReactionR2. Le modèle de sélection d'interaction utilisé pour la ReactionR1 est identique à celui présenté dans la section précédente : il y a d'abord sélection de l'interaction ReactionR1 par pondération (un poids k_1 est donné à cette interaction, et un poids $1 - k_1$ est donné au fait de ne rien initier), puis sélection d'une cible aléatoirement. Dans le cas où ReactionR1 n'est pas initiée, la sélection portant sur l'interaction ReactionR2 fonctionne de la même manière : il y a d'abord sélection de l'interaction ReactionR2 par pondération (un poids k_2 est donné à cette interaction, et un poids $1 - k_2$ est donné au fait de ne rien initier), puis sélection d'une cible aléatoirement.

On peut donc généraliser le modèle réactif de sélection d'interaction ébauché à la fin de la section précédente, attribuant une priorité aux interaction, et en définissant une politique d'interaction pour chaque priorité.

Limites de l'étude directe des algorithmes

Dans les deux expériences que nous avons détaillées dans cette section, nous avons identifié l'implémentation la plus appropriée du comportement stochastique d'un agent en analysant directement du code. Cette analyse n'a été possible que grâce à la taille réduite des problèmes étudiés, et du nombre restreint des interactions pouvant être initiées par un agent. La complexité de cette tache croît exponentiellement avec le nombre d'interactions pouvant être initiées par les agents, si bien qu'en pratique, il est difficile de mener ce genre d'étude. Afin d'éviter ce problème, le modèle de sélection d'interaction des agents doit faire apparaître explicitement les choix de conception effectués implicitement dans ces algorithmes.

En nous basant sur les résultats expérimentaux obtenus dans cette section, nous sommes en mesure d'étendre le modèle de l'approche Ioda afin de faire apparaître explicitement de tels choix de conception.

7.3.3 Spécification fine de la sélection d'interactions réactive

Pour mettre en exergue tous les choix de modélisation étant apparu dans cette section et pousser les concepteurs à les mentionner explicitement, nous proposons ici une extension du modèle réactif de sélection d'interaction décrit dans le chapitre 3. Cette extension se base aussi sur la notion de priorités. Dans le modèle original la sélection pour une priorité donnée consiste à choisir aléatoirement un tuple du potentiel d'interaction de cette priorité. Nous étendons ici la sélection à d'autres politiques de sélection

d'interaction. L'algorithme général permettant d'effectuer la sélection d'interaction devient alors celui présenté dans l'algorithme 15.

Algorithme 15 : Algorithme décrivant comment une entité utilisant l'extension du modèle réactif de sélection d'interaction détermine le tuple dont il initie l'interaction. Dans cet algorithme, on suppose disposer d'une fonction nommée *élément* permettant de connaître l'élément d'assignation d'un tuple.

```
sélection(x)
début
    R ⇐ potentiel d'interaction(x);
    % O contiendra l'ensemble des priorités des éléments d'assignation où e est une source
    O ⇐ ∅;
    pour tous les a ∈ ligne(famille(x)) faire
        si M_raff(a) ∉ O alors
            O ⇐ O ∪ {M_raff(a)};
    élémentChoisi ⇐ null;
    tant que élémentChoisi == null et O ≠ ∅ faire
        % Récupération de la plus grande priorité contenue dans O
        p ⇐ max(O);
        O ⇐ O\{p};
        % Récupération de tous les tuples de priorité p
        R_p ⇐ ∅;
        pour tous les T ∈ R faire
            si M_raff(élément(T)) == p alors
                R_p ⇐ R_p ∪ {T};
        si R_p ≠ ∅ alors
            élémentChoisi ⇐ élément R_p choisi selon la politique de sélection associée à la priorité p;
    retourner élémentChoisi;
fin
```

Nous définissons trois *politiques de sélection d'interaction*, qui se différencient par le procédé utilisé pour sélectionner un tuple réalisable au sein du potentiel d'interaction de priorité p [KMP08d, KMP09] :

1. nous appelons *sélection par interaction puis par cible* la politique qui consiste à sélectionner dans un premier temps une interaction présente dans au moins un tuple réalisable, puis à en sélectionner la cible ;
2. nous appelons *sélection par cible puis par interaction* la politique qui consiste à sélectionner dans un premier temps un agent cible présent dans au moins un tuple réalisable, puis à sélectionner une interaction prenant cet agent pour cible ;
3. nous appelons *sélection par tuple* la politique qui consiste à sélectionner directement un tuple du potentiel d'interaction.

Dans chaque politique, les sélections effectuées ont une *nature*, qui définit comment sélectionner un tuple, un élément d'assignation, ou une cible parmi un ensemble d'alternatives. Dans notre modèle, nous définissons les trois natures de sélection qui suivent :

- la *sélection par préférence* consiste à attribuer une valeur à chaque élément pouvant être sélectionné, à l'aide d'une *fonction d'attribution de préférence*, et à sélectionner l'élément ayant la valeur la plus élevée. Si plusieurs éléments ont la même valeur, alors l'élément sélectionné est choisi aléatoirement parmi eux ;
- la *sélection pondérée* consiste à attribuer un poids à chaque élément pouvant être sélectionné, à l'aide d'une *fonction d'attribution de poids*, et à effectuer une sélection pondérée d'un élément ;

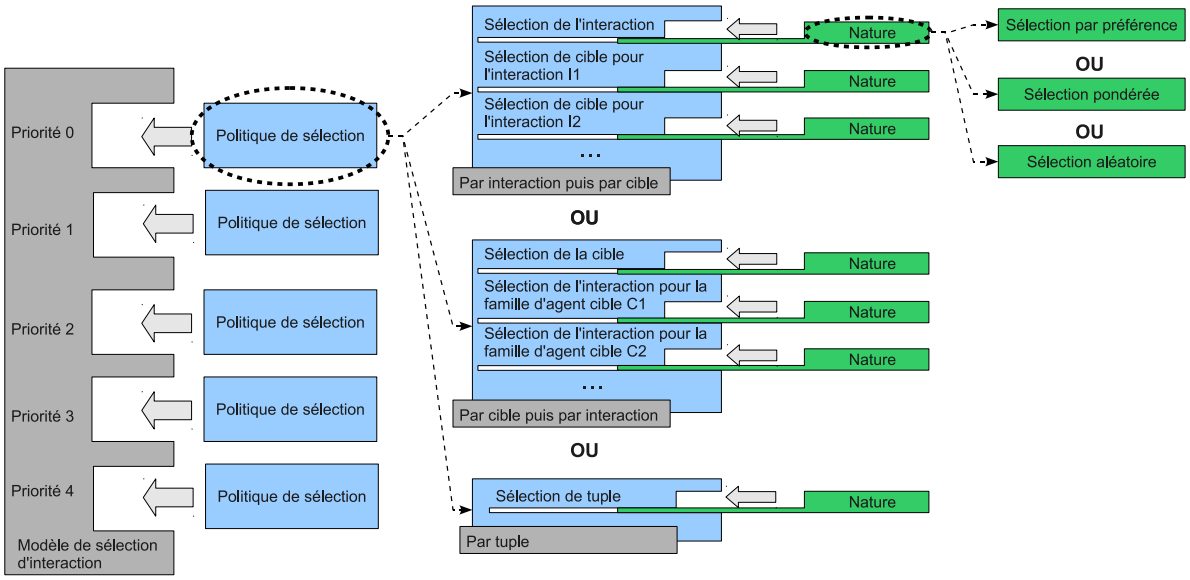


FIGURE 7.6 – Modèle de sélection d'interaction étendue présenté dans la section 7.3. La construction du modèle de sélection d'interaction d'un agent source consiste à attribuer une politique de sélection d'interaction à chaque priorité figurant dans sa ligne de la matrice d'interaction raffinée. Chaque politique définit un ordre spécifique dans quel effectuer la sélection parmi les interaction et les cibles (*i.e.* sélectionner d'abord une interaction, puis une cible, ou d'abord une cible puis une interaction, ou directement un couple interaction/cible). Chacune de ces sélections est régie par une nature, caractérisant comment la sélection a lieu (*i.e.* par la maximisation d'une fonction d'évaluation, par pondération, ou aléatoirement).

– la *sélection aléatoire* est un cas particulier de la sélection pondérée, où chaque élément se voit attribuer le même poids.

La figure 7.6 fournit une vue d'ensemble sur le modèle que nous proposons.

Dans le modèle réactif de sélection d'interaction décrit dans le chapitre 3, la politique de sélection utilisée implicitement est une politique de sélection par tuples, dont la nature est une sélection aléatoire. Ce principe de sélection réactive d'interaction est suffisant dans la majorité des simulations impliquant des agents réactifs.

Modèle

Plus formellement, une nature de sélection est définie au minimum comme une fonction prenant en paramètres une liste d'interactions, une liste de cibles ou une liste de tuples, et qui retourne soit l'un des éléments de la liste, soit *null* (si aucun élément n'est sélectionné).

Définition 40. ***Nature d'une sélection***
La **nature d'une sélection** est définie comme une fonction $selection_{nature}$ permettant de sélectionner une interaction parmi une liste d'interactions, une cible parmi une liste de cibles, ou un tuple parmi une liste de tuples. Il est possible que la nature ne sélectionne rien, dans quel cas la valeur retournée est *null*. Elle prend donc les formes :
– $selection_{nature} : \mathcal{P}(\mathbb{E}) \to \mathbb{E} \cup \{null\}$, si la sélection porte sur des cibles ;
– $selection_{nature} : \mathcal{P}(\mathbb{I}_{(1,0)} \cup \mathbb{I}_{(1,1)}) \to \mathbb{I}_{(1,0)} \cup \mathbb{I}_{(1,1)} \cup \{null\}$, si la sélection porte sur des interactions ;
– $selection_{nature} : \mathcal{P}\Big(\big((\mathbb{I}_{(1,0)} \times \mathbb{F}) \times \mathbb{E}\big) \cup \big((\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}) \times \mathbb{E} \times \mathbb{E}\big)\Big) \to \big((\mathbb{I}_{(1,0)} \times \mathbb{F}) \times \mathbb{E}\big) \cup \big((\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}) \times \mathbb{E} \times \mathbb{E}\big) \cup \{null\}$, si la sélection porte sur des tuples réalisables ;

La *sélection par préférence* base la sélection sur l'attribution d'une note aux différents éléments pouvant être sélectionnés. Elle définit pour cela une *fonction d'attribution de valeur*, qui prend en paramètre

un élément, et retourne un nombre à virgule flottante correspondant à la valeur attribuée à cet élément.

Exemple. Suivi de phéromones
Plaçons-nous dans le cadre d'une simulation où des fourmis se déplacent en fonction des phéromones déposées par les autres fourmis. Une `Fourmi` SE DÉPLACE *toujours* VERS *la* `Parcelle` *de l'environnement située à proximité contenant la plus grande concentration en phéromones. La sélection de la cible de l'interaction* SE DÉPLACER VERS *se fait donc par préférence. La fonction d'attribution de valeur retourne dans ce cas la concentration en phéromones de l'agent* `Parcelle` *qu'elle prend en paramètre.*

Les valeurs retournées par cette fonction dépendent des éléments sur lesquels portent la sélection. Par conséquent, lorsqu'une sélection par préférence est utilisée dans une politique de sélection, la fonction d'attribution de valeurs lui étant associée doit être spécifiée.

Définition 41. ***Sélection par préférence***
Une **sélection par préférence** attribue une valeur à chaque élément pouvant être sélectionné à l'aide d'une **fonction d'attribution de valeur**. Cette nature de sélection s'assure alors de sélectionner l'élément ayant une valeur maximale. Si plusieurs éléments ont cette valeur alors un élément est sélectionné aléatoirement parmi eux.
Plus formellement, une sélection par préférence est définie comme un couple $(selection_{nature}, attr_{val})$, où $selection_{nature}$ est la fonction permettant de sélectionner un élément selon cette nature, qui se comporte tel que décrit dans l'algorithme 16, et $attr_{val}$ est la fonction d'attribution de valeur de cette nature, avec :
- $attr_{val} : \mathbb{E} \to \mathbb{R}$, si la sélection porte sur des cibles ;
- $attr_{val} : \mathbb{I}_{(1,0)} \cup \mathbb{I}_{(1,1)} \to \mathbb{R}$, si la sélection porte sur des interactions ;
- $attr_{val} : \big((\mathbb{I}_{(1,0)} \times \mathbb{F}) \times \mathbb{E}\big) \cup \big((\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}) \times \mathbb{E} \times \mathbb{E}\big) \to \mathbb{R}$, si la sélection porte sur des tuples réalisables.

Algorithme 16 : Algorithme de la fonction $selection_{nature}$ dans le cas d'une sélection par préférence. Cette fonction décrit comment s'effectue la sélection d'un tuple, d'une cible ou d'une interaction selon cette nature. Elle se base sur la fonction d'attribution de valeur $attr_{val}$ définie par la sélection par préférence.

```
% liste représente la liste des éléments parmi lesquels la sélection est faite.
selection_nature(liste)
début
    S ⇐ ∅;
    max ⇐ −∞;
    pour elt ∈ liste faire
        val ⇐ attr_val(elt);
        si val > max alors
            max ⇐ val;
            S ⇐ {elt};
        sinon
            si val == max alors
                S ⇐ S ∪ {elt};
    si S ≠ ∅ alors
        retourner random(S);
    sinon
        retourner null;
fin
```

La *sélection pondérée* se fonde sur un principe proche de l'algorithme de Gillespie [Gil76], aussi appelé méthode de Monte-Carlo dynamique. Cette nature base la sélection sur l'attribution d'un poids aux différents éléments pouvant être sélectionnés. Elle définit pour cela une *fonction d'attribution de poids*, qui prend en paramètre un élément, et retourne un nombre à virgule flottante de l'intervalle $[0; 1[$

correspondant au poids attribué à cet élément. Le principe de cette sélection consiste alors à attribuer une portion de l'intervalle $[0;1[$ à chaque élément pouvant être sélectionné, dont la longueur est le poids de l'élément. La sélection consiste alors à tirer un nombre aléatoire, puis à identifier l'élément dont la portion de l'intervalle $[0;1[$ contient le nombre aléatoire. Cet élément est alors l'élément sélectionné par la *sélection pondérée*. Si jamais la somme des poids de tous les éléments pouvant être sélectionnés dépasse 1, alors les poids sont diminués proportionnellement afin que leur somme soit égale à 1. Si jamais cette somme est inférieure à 1, alors il y a une probabilité égale à 1 moins cette somme qu'aucun élément ne soit sélectionné.

Exemple. Suivi de phéromones
Plaçons nous à nouveau dans le cadre d'une simulation de fourmis fourragères, où des fourmis se déplacent en fonction des phéromones déposées par les autres fourmis. Considérons que la propension d'une `Fourmi` *à* Se Déplacer Vers *une* `Parcelle` *est fonction de la quantité de phéromones qu'elle contient. Plus une* `Parcelle` *contient une forte concentration en phéromones, plus une* `Fourmi` *tend à* Se Déplacer Vers *elle. La sélection de la cible de l'interaction* Se Déplacer Vers *se fait alors par pondération. La fonction d'attribution de poids retourne dans ce cas la concentration en phéromones de l'agent* `Parcelle` *qu'elle prend en paramètre.*

Les poids retournés par cette fonction dépendent des éléments sur lesquels portent la sélection. Par conséquent, lorsqu'une sélection pondérée est utilisée dans une politique de sélection, la fonction d'attribution de poids lui étant associée doit être spécifiée.

Définition 42. ***Sélection pondérée***
Une **sélection pondérée** attribue un poids à chaque élément pouvant être sélectionné à l'aide d'une **fonction d'attribution de poids**. Cette nature sélectionne de manière pondérée un élément en fonction des poids qui leur ont été attribués. Pour cela, elle attribue une portion de l'intervalle $[0;1[$ à chaque élément en fonction de leur poids, puis tire un nombre aléatoire dans l'intervalle $[0;1[$ et sélectionne l'élément dont la portion d'intervalle contient ce nombre. Si la somme des poids est supérieure à 1, alors elle est ramenée à a en divisant tous les poids par cette somme.
Plus formellement, une sélection par préférence est définie comme un couple $(selection_{nature}, attr_{poids})$, où $selection_{nature}$ est la fonction permettant de sélectionner un élément selon cette nature, qui se comporte tel que décrit dans l'algorithme 17 et $attr_{poids}$ est la fonction d'attribution de poids de cette nature, avec :
- $attr_{poids} : \mathbb{E} \to [0;1[$, si la sélection porte sur des cibles ;
- $attr_{poids} : \mathbb{I}_{(1,0)} \cup \mathbb{I}_{(1,1)} \to [0;1[$, si la sélection porte sur des interactions ;
- $attr_{poids} : \big((\mathbb{I}_{(1,0)} \times \mathbb{F}) \times \mathbb{E}\big) \cup \big((\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}) \times \mathbb{E} \times \mathbb{E}\big) \to [0;1[$, si la sélection porte sur des tuples réalisables.

Enfin, la *sélection aléatoire* est une nature de sélection pondérée particulière retrouvée couramment dans les simulations réactives. Sa particularité est que sa fonction d'attribution de poids fournit un poids de 1 à chaque élément pouvant être sélectionné. Elle assure donc que la sélection de l'élément soit équitable.

Définition 43. ***Sélection aléatoire***
Une **sélection aléatoire** est sélection pondérée particulière dans laquelle la **fonction d'attribution de poids** retourne systématiquement la valeur 1. Elle assure donc un choix équitable entre tous les éléments pouvant être sélectionnés.
Plus formellement, une sélection par préférence est définie comme un couple $(selection_{nature}, attr_{poids})$, où $selection_{nature}$ est la fonction permettant de sélectionner un élément selon cette nature, et $attr_{poids}$ est une fonction d'attribution de poids retournant systématiquement 1.

Une politique de sélection est aussi définie au minimum comme une fonction, prenant toutefois en paramètres un potentiel d'interaction de priorité p, *i.e.* un ensemble de tuples réalisables, et retournant le tuple sélectionné, s'il y en a un.

Algorithme 17 : Algorithme de la fonction $selection_{nature}$ dans le cas d'une sélection pondérée. Cette fonction décrit comment s'effectue la sélection d'un tuple, d'une cible ou d'une interaction selon cette nature. Elle se base sur la fonction d'attribution de poids $attr_{poids}$ définie par la sélection pondérée.

```
% liste représente la liste des éléments parmi lesquels la sélection est faite.
selection_nature(liste)
début
    % poids est une structure de données associant un poids à un élément
    poids ⇐ ∅;
    % ordre est une structure de données associant un index à un élément
    ordre ⇐ ∅;
    poidsTotal ⇐ 0;
    index ⇐ 0;
    pour elt ∈ liste faire
        poids[elt] ⇐ attr_poids(elt);
        ordre[index] ⇐ elt;
        poidsTotal ⇐ poidsTotal + poids[elt];
        index ⇐ index + 1;
    si poidsTotal > 1 alors
        pour elt ∈ liste faire
            poids[elt] ⇐ poids[elt] / poidsTotal;
    r ⇐ random([0; 1[);
    min ⇐ 0;
    pour i ∈ [0; index[ faire
        max ⇐ min + poids[ordre[i]];
        si r ∈ [min; max[ alors
            retourner ordre[i];
        min ⇐ max;
    retourner null;
fin
```

Définition 44. ***Politique de sélection***
Une **politique de sélection** est définie comme une fonction $selection_{politique}$ permettant de sélectionner un tuple réalisable parmi le potentiel d'interaction de priorité p. Il est possible que la politique ne sélectionne rien, dans quel cas la valeur retournée est *null*. Elle prend donc la forme : $selection_{politique} : \mathcal{P}\Big(\big((\mathbb{I}_{(1,0)} \times \mathbb{F}) \times \mathbb{E}\big) \cup \big((\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}) \times \mathbb{E} \times \mathbb{E}\big)\Big) \to \big((\mathbb{I}_{(1,0)} \times \mathbb{F}) \times \mathbb{E}\big) \cup \big((\mathbb{I}_{(1,1)} \times \mathbb{R}^+ \times \mathbb{F} \times \mathbb{F}) \times \mathbb{E} \times \mathbb{E}\big) \cup \{null\}$

La politique de *sélection par tuples* consiste à directement sélectionner un tuple réalisable parmi le potentiel d'interaction de priorité p, à l'aide d'une nature de sélection portant sur les tuples.

Définition 45. ***Politique de sélection par tuple***
La **politique de sélection par tuple** consiste à sélectionner un tuple réalisable parmi le potentiel d'interaction en utilisant une nature de sélection par préférence, pondérée, ou aléatoire portant sur les tuples de ce potentiel d'interaction.
Plus formellement, une politique de sélection par tuple est définie comme un couple $(selection_{politique}, nature)$ où $selection_{politique}$ est la fonction permettant de sélectionner un tuple réalisable selon cette politique, qui se comporte tel que décrit dans l'algorithme 18, et *nature* est une nature de sélection.

Algorithme 18 : Algorithme de la fonction $selection_{politique}$ dans le cas d'une politique de sélection par tuple. Cette fonction décrit comment s'effectue la sélection d'un tuple réalisable du potentiel d'interaction selon cette politique. Elle se base sur la fonction $selection_{nature}$ définie par la *nature* associée à la politique.

```
% R_p représente le potentiel d'interaction de priorité p.
selection_politique(R_p)
début
  | retourner nature.selection_nature(R_p);
fin
```

La politique de *sélection par interaction puis par cible* consiste à sélectionner un tuple réalisable parmi le potentiel d'interaction de priorité p d'un agent en deux temps. Dans un premier temps, l'ensemble des interactions figurant dans les divers tuples du potentiel d'interaction sont recensées, puis une sélection est effectuée parmi ces dernières selon une première nature. Une fois la sélection d'une interaction effectuée, les tuples réalisables ne contenant pas cette interaction sont retirés du potentiel d'interaction. Dans un second temps, l'ensemble des cibles figurant dans les divers tuples du potentiel d'interaction réduit sont recensées. Une sélection est alors effectuée parmi ces dernières selon une seconde nature définie par la politique pour cette interaction. Le tuple sélectionné est alors le tuple contenant à la fois l'interaction et la cible sélectionnés.

Définition 46. ***Politique de sélection par interaction puis par cible***
La **politique de sélection par interaction puis par cible** consiste à sélectionner un tuple réalisable parmi le potentiel d'interaction en deux temps. Dans un premier temps, la sélection se fait parmi les interactions selon une nature notée $nature_{interaction}$. Dans un second temps, la sélection est faite parmi les cibles des tuples réalisables dont l'interaction est celle sélectionnée lors de la première étape. Cette seconde sélection se fait selon une nature propre à l'interaction I sélectionnée, notée $nature_{cible}(I)$. Le tuple sélectionné est alors le tuple contenant à la fois l'interaction et la cible sélectionnés.
Plus formellement, une politique de sélection par interaction puis par cible est définie comme un triplet $(selection_{politique}, nature_{interaction}, nature_{cible})$ où $selection_{politique}$ est la fonction permettant de sélectionner un tuple réalisable selon cette politique, qui se comporte tel que décrit dans l'algorithme 19, où $nature_{interaction}$ est une nature de sélection utilisée pour sélectionner une interaction, et $nature_{cible}$ est une fonction associant une nature de sélection d'une cible à chaque interaction pouvant être sélectionnée par $nature_{interaction}$.

Algorithme 19 : Algorithme de la fonction $selection_{politique}$ dans le cas d'une politique de sélection par tuple. Cette fonction décrit comment s'effectue la sélection d'un tuple réalisable du potentiel d'interaction selon cette politique. Elle se base sur la fonction $selection_{nature}$ définie par la nature $nature_{interaction}$ associée à la politique, ainsi que sur la fonction $selection_{nature}$ définie par l'une des natures retournée par la fonction $nature_{cible}$ associée à la politique.

```
% R_p représente le potentiel d'interaction de priorité p.
selection_politique(R_p)
début
    interactions ⇐ ∅;
    % Récupération de l'ensemble des interactions
    pour T ∈ R_p faire
        I ⇐ l'interaction associée au tuple réalisable T;
        si I ∉ interactions alors
            interactions ⇐ interactions ∪ {I};
    % Sélection d'une interaction à l'aide de la nature de sélection nature_interaction
    I ⇐ nature_interaction.selection_nature(interactions);
    si I ≠ null alors
        % Retrait de tous les tuples ne contenant pas l'interaction I
        pour T ∈ R_p faire
            si l'interaction associée au tuple réalisable T n'est pas I alors
                R_p ⇐ R_p\{T};
        % Récupération de l'ensemble des cibles
        pour T ∈ R_p faire
            cibles ⇐ ∅;
            pour c ∈ l'ensemble des cibles de T faire
                si c ∉ cibles alors
                    cibles ⇐ cibles ∪ {c};
        % Sélection d'une cible à l'aide de la nature de sélection nature_cible(I)
        c ⇐ nature_cible(I).selection_nature(cibles);
        si c ≠ null alors
            % Retrait de tous les tuples ne contenant pas la cible c
            pour T ∈ R_p faire
                si le tuple réalisable T ne contient pas la cible c alors
                    R_p ⇐ R_p\{T};
            % On retourne un tuple réalisable au hasard parmi ceux qui restent
            retourner random(R_p);
        sinon
            retourner null;
    sinon
        retourner null;
fin
```

Enfin, la politique de *sélection par cible puis par interaction* consiste à sélectionner un tuple réalisable parmi le potentiel d'interaction de priorité p d'un agent en deux temps. Dans un premier temps, les cibles figurant dans les divers tuples du potentiel d'interaction sont recensées, puis une sélection est effectuée parmi ces dernières selon une première nature définie par la politique. Le potentiel d'interaction est alors réduit aux tuples contenant la cible sélectionnée. Dans un second temps, les interactions figurant dans les divers tuples du potentiel d'interaction réduit sont recensés, puis une sélection est effectuée parmi ces dernières selon une seconde nature dépendant de la famille d'agents de la cible sélectionnée. Le tuple sélectionné est alors le tuple contenant à la fois l'interaction et la cible sélectionnés.

Définition 47. ***Politique de sélection par cible puis par interaction***
La **politique de sélection par cible puis par interaction** consiste à sélectionner un tuple réalisable parmi le potentiel d'interaction en deux temps. Dans un premier temps, la sélection se fait parmi les cibles selon une nature notée $nature_{cible}$. Dans un second temps, la sélection est faite parmi les interactions des tuples réalisables dont les cibles contiennent la cible sélectionnée lors de la première étape. Cette seconde sélection se fait selon une nature propre à la famille d'agents F de l'agent sélectionné, notée $nature_{interaction}(F)$. Le tuple sélectionné est alors le tuple contenant à la fois l'interaction et la cible sélectionnés.
Plus formellement, une politique de sélection par interaction puis par cible est définie comme un triplet $(selection_{politique}, nature_{interaction}, nature_{cible})$ où $selection_{politique}$ est la fonction permettant de sélectionner un tuple réalisable selon cette politique, qui se comporte de manière similaire à l'algorithme 19 (il suffit d'inverser sélection d'interaction et sélection de cible), où $nature_{cible}$ est une nature de sélection utilisée pour sélectionner une cible, et $nature_{interaction}$ est une fonction associant une nature de sélection d'une interaction à chaque famille d'agents dont une instance peut être sélectionnée par $nature_{cible}$.

Intégration dans la méthodologie

Le modèle décrit ici s'intègre à la méthodologie de conception lors de l'étape de construction de la matrice d'interaction raffinée. Au cours de cette étape, une priorité est attribuée à chaque élément d'assignation figurant dans la ligne associée à une famille d'agents. Dans cette extension, cette étape de la méthodologie consiste aussi à attribuer une politique de sélection pour chaque priorité définie dans une ligne de la matrice d'interaction.

La construction de chaque politique consiste alors dans un premier temps à identifier quelle politique utiliser parmi la *politique de sélection par tuple*, la *politique de sélection par interaction puis par cible* et la *politique de sélection par cible puis par tuple*. Selon le choix effectué ici, les spécifications devant être fournies diffèrent.

Si le choix du concepteur s'est porté sur la *politique de sélection par tuple*, le concepteur doit définir quelle nature de sélection utiliser pour le choix d'un tuple. S'il s'agit d'une sélection aléatoire, la spécification s'arrête ici. S'il s'agit d'une sélection pondérée, une fonction d'attribution de poids doit être définie par le concepteur pour sélectionner les tuples de cette priorité. Enfin, s'il s'agit d'une sélection par préférences, une fonction d'attribution de valeur doit être définie par le concepteur pour sélectionner les tuples de cette priorité.

Si le choix concepteur s'est porté sur la *politique de sélection par interaction puis par cible*, la spécification de la politique se fait en deux temps. Dans un premier temps, le concepteur doit définir la nature de sélection utilisée pour sélectionner une interaction, selon les mêmes modalités que la spécification de la nature de sélection décrite dans le paragraphe précédent. Dans un second temps, le concepteur doit associer une nature de sélection à chaque interaction pouvant être sélectionnée lors de la première étape. Cette nature permet de décrire comment sélectionner la cible de l'interaction sélectionnée.

Enfin, si le choix concepteur s'est porté sur la *politique de sélection par cible puis par interaction*, la spécification de la politique se fait aussi en deux temps. Dans un premier temps, le concepteur doit définir la nature de sélection utilisée pour sélectionner une cible, selon les mêmes modalités que la spécification de la nature de sélection décrite dans le paragraphe précédent. Dans un second temps, le concepteur doit associer une nature de sélection à chaque famille d'agent cible dont une instance peut être sélectionnée

lors de la première étape. Cette nature permet de décrire comment sélectionner l'interaction prenant pour cible l'instance sélectionnée.

L'utilisation combinée des priorités, des trois politiques d'interactions et des trois natures de sélection permet de spécifier des comportements réactifs finement et de manière explicite dans le modèle. En effet, il devient possible d'exprimer par exemple « un prédateur préfère manger des proies saines » à l'aide d'une politique de sélection par tuple, dont la nature est une sélection par préférences. Il est également possible d'exprimer qu'« une réaction chimique est initiée par une source avec une probabilité k » à l'aide d'une politique de sélection par interaction puis par cible, où :

- le choix de l'interaction se fait selon une sélection pondérée donnant un poids k à la interaction de réaction chimique ;
- le choix de la cible se fait selon une sélection aléatoire.

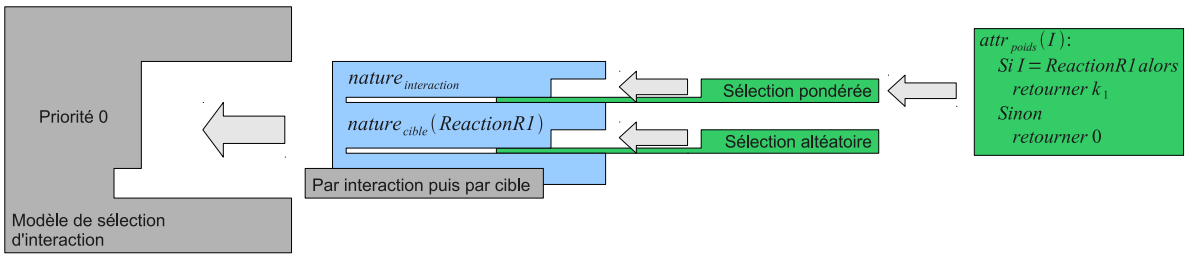
7.3.4 Retour sur les expériences

L'utilisation du modèle, de la méthodologie et des algorithmes décrits ici permet de mettre en avant les choix de modélisation effectués par le concepteur. Nous illustrons ici ce point en reprenant les quatre implémentations de l'unité de sélection que nous avons considérées dans cette section, et en les modélisant suivant l'approche défendue ici. La figure 7.7 résume les différents modèles obtenus.

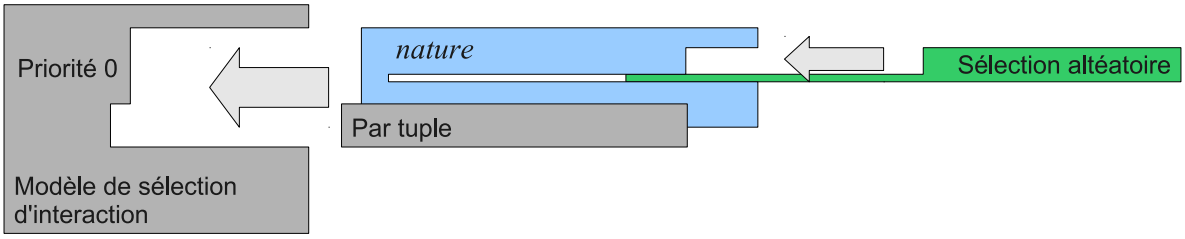
Unité de sélection de l'algorithme 13(a) du premier cas d'étude Dans cette unité de sélection, il n'y a qu'une seule interaction. Par conséquent, le modèle réactif de sélection d'interaction ne définit qu'une seule priorité, que nous fixons arbitrairement à 0. La probabilité d'initier la réaction ReactionR1 est testée indépendamment du nombre d'agents différents pouvant être la cible de la réaction. Cette unité de sélection repose par conséquent sur une politique de sélection par interaction puis par cible. Le choix de l'interaction se fait selon une probabilité. Il s'agit donc d'une sélection pondérée, où le poids de l'interaction ReactionR1 est k_1. Si l'interaction ReactionR1 est sélectionnée, la cible est choisie aléatoirement. La nature de la sélection de la cible associée à l'interaction ReactionR1 dans la politique de sélection est donc une sélection aléatoire. Ce modèle est alors résumé par le schéma de la figure 7.7(a).

Unité de sélection de l'algorithme 13(b) du premier cas d'étude Dans cette unité de sélection, il n'y a qu'une seule interaction. Par conséquent, le modèle réactif de sélection d'interaction ne définit qu'une seule priorité, que nous fixons arbitrairement à 0. La probabilité d'initier la réaction ReactionR1 est testée pour chaque cible, et l'interaction ne peut être initiée que si cette probabilité est vérifiée. Par conséquent, le test de la probabilité est partie intégrante des conditions de l'interaction. Dans ce cas, le potentiel d'interaction de priorité 0 ne contient que les tuples ayant vérifié cette probabilité. La sélection consiste alors à sélectionner aléatoirement une cible et l'interaction. Cette unité de sélection repose par conséquent sur une politique de sélection par tuples, dont la nature de sélection est une sélection aléatoire. Ce modèle est alors résumé par le schéma de la figure 7.7(b).

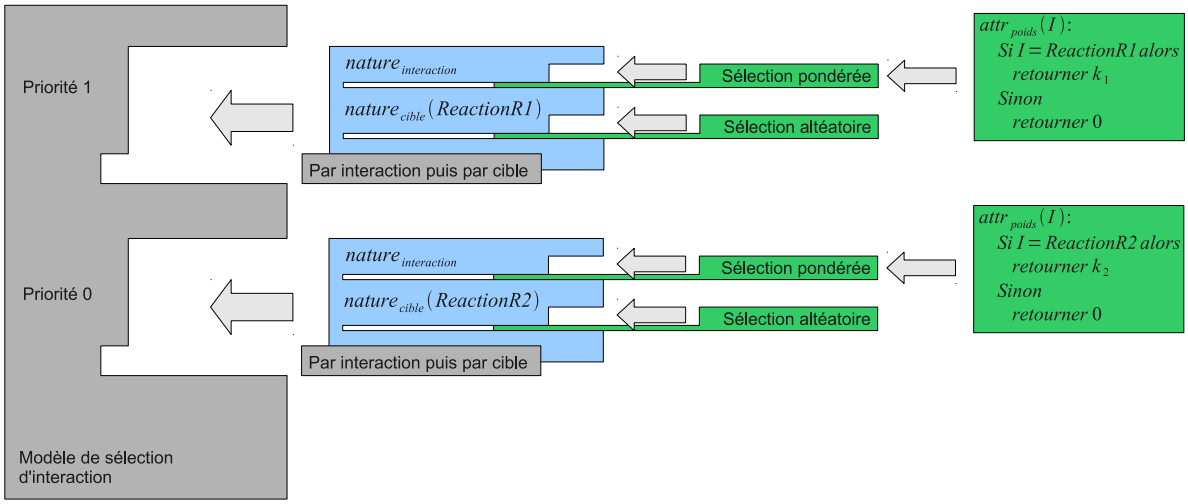
Unité de sélection de l'algorithme 14(a) du second cas d'étude Dans cette unité de sélection, plusieurs interactions sont considérées. La probabilité d'initier ReactionR1 et la probabilité d'intier ReactionR2 sont testées indépendamment dans l'unité de sélection, puisqu'il y a deux appels différents à la fonction $random([0;1[)$. Puisque la probabilité d'initier ReactionR1 est vérifiée en premier, ReactionR2 n'est initiée que si ReactionR1 n'est pas initiée. ReactionR1 est donc plus prioritaire. Nous choisissons d'attribuer la priorité 1 à l'interaction ReactionR1 dans la matrice d'interaction raffinée, et la priorité 0 à l'interaction ReactionR2. Puisqu'il y a deux priorités différentes, nous devons définir deux politiques de sélection différentes. Si l'on fait abstraction de la présence de l'interaction ReactionR2, la sélection d'interaction se fait exactement comme décrit dans l'algorithme 13(a) du premier cas d'étude. Par conséquent, cette unité de sélection repose sur une politique de sélection par interaction puis par cible pour la priorité 1, où la nature de la sélection de l'interaction est une sélection pondérée fournissant un poids k_1 à l'interaction ReactionR1, et la nature de la sélection des cibles est aléatoire. Les trois lignes propres à l'interaction Reaction2 dans l'algorithme 14(a) sont similaires aux trois lignes propres à l'interaction Reaction1. Nous sommes donc en droit de penser que la politique de sélection pour



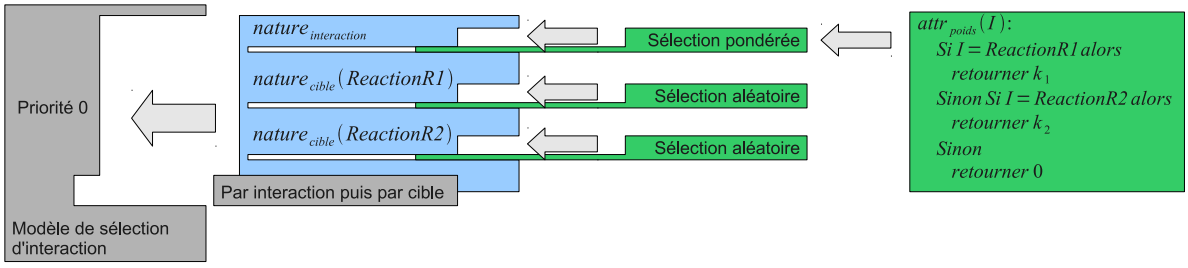
(a) Politique de sélection équivalente à l'unité de sélection décrite dans l'algorithme 13(a) de la page 206.



(b) Politique de sélection équivalente à l'unité de sélection décrite dans l'algorithme 13(b) de la page 206. Cette politique de sélection est utilisée implicitement dans la partie II.



(c) Politique de sélection équivalente à l'unité de sélection décrite dans l'algorithme 14(a) de la page 207.



(d) Politique de sélection équivalente à l'unité de sélection décrite dans l'algorithme 14(b) de la page 207.

FIGURE 7.7 – Modélisation, à l'aide de politiques de sélection, des unités de sélection utilisées dans les cas d'étude de la section 7.3.

la priorité 0 est similaire à la politique de sélection pour la priorité 1 (en modifiant les occurrences de ReactionR1 par ReactionR2, et les occurrences de k_1 par k_2). Ce modèle est alors résumé par le schéma de la figure 7.7(c).

Unité de sélection de l'algorithme 14(b) du second cas d'étude Dans cette unité de sélection, plusieurs interactions sont considérées. La probabilité d'initier ReactionR1 et la probabilité d'intier ReactionR2 sont testées avec le même appel à la fonction $random([0;1[)$. Il y a par conséquent une sélection pondérée entre ces deux interactions, ce qui implique qu'une seule et même priorité est attribuée à l'interaction ReactionR1, et à l'interaction ReactionR2. Puisque la pondération ne change pas selon le nombre de cibles, cette unité repose sur une politique de sélection par interaction puis par cible. La nature de la sélection d'une interaction est une sélection pondérée, où le poids attribué à la réaction ReactionR1 est k_1, et où le poids attribué à la réaction ReactionR2 est k_2. Dans les deux cas, la sélection de la cible de la réaction est fait aléatoirement, et repose donc sur une sélection de nature aléatoire. Ce modèle est alors résumé par le schéma de la figure 7.7(d).

La structure du modèle que nous proposons fait apparaître les choix de conception à l'origine des différences existant entre les quatre unités de sélection que nous avons étudiées. Puisque construire une simulation selon ce modèle ne peut être fait sans spécifier précisément quelles politiques de sélections sont utilisées, ce modèle oblige les concepteurs à effectuer explicitement des choix de conception relatifs à l'unité de sélection, ce qui est l'objectif recherché dans cette section.

7.4 Synthèse du chapitre

Une même simulation implémentée par deux personnes différentes peut aboutir dans certains cas à des résultats très différents. Cette différence est liée à des choix effectués implicitement par les concepteurs lors de l'implémentation. Afin d'éviter d'introduire des biais dans une simulation, une méthodologie de conception doit donc pousser ses utilisateurs à spécifier explicitement un maximum de choix de conception.

La construction d'une telle méthodologie nécessite la résolution de trois problèmes différents :

- Identifier les choix de conception apparaissant de manière explicite ou implicite lors de la construction du modèle et de l'implémentation ;
- Fournir un modèle permettant d'exprimer explicitement de tels choix ;
- Fournir une méthodologie de conception aidant les utilisateurs à exprimer le plus simplement possible ces choix, afin de ne pas complexifier excessivement la construction de modèles.

Trois unités fonctionnelles Pour identifier plus aisément les choix de conception, toute simulation peut être décomposée en trois unités fonctionnelles disjointes :

- l'unité d'activation qui décrit tous les éléments liés à la représentation du temps dans la simulation ;
- l'unité de définition qui décrit quelles interactions peuvent avoir lieu entre les agents présents dans la simulation, dans quelles conditions elles peuvent survenir, et quels sont leurs effet ;
- l'unité de sélection qui décrit le procédé réactif ou cognitif utilisé par un agent pour sélectionner les interactions qu'il initie lorsque l'unité d'activation lui donne la parole.

Puisque la conception d'une simulation implique des choix cruciaux concernant ces trois unités, nous soutenons qu'il est fondamental de faire apparaître explicitement cette séparation, même dans le cas de simulations contenant des agents réactifs.

En effet, en menant des études en suivant cette séparation, nous avons dégagé deux extensions de Ioda permettant de rendre explicites les choix de conception concernant deux problèmes de simulation :

- la participation simultanée à plusieurs interactions ;
- la spécification de comportements stochastiques.

Participation simultanée à plusieurs interactions L'unité d'activation a entre autres pour rôle de déterminer si un agent a le droit de participer simultanément à plusieurs interactions. Les choix de conception relatifs à ce problème peuvent être mis en évidence et spécifiés facilement grâce à Ioda.

En effet, nous avons montré qu'il suffisait d'attribuer la classe « parallèle » ou la classe « exclusive » à chaque interaction pouvant être initiée par un agent pour caractériser si cet agent peut participer

simultanément à plusieurs interactions. La classe d'une interaction I est identifiée par un procédé simple consistant à répondre à l'une des deux questions :

- une cible de I peut-elle être la source d'une autre interaction lors du même pas de temps ?
- un agent peut-il être la cible de deux interactions I ayant des sources différentes lors du même pas de temps ?

Si la réponse à la question est oui, l'interaction est parallèle. Sinon, elle est exclusive. Des algorithmes tirent alors parti de ces classes afin d'empêcher les agents d'initier des interactions entrant en conflit avec des interactions en cours d'exécution au moment où ils prennent la parole.

Spécification de comportements stochastiques Un comportement stochastique peut être implémenté par des unités de sélection très différentes. L'analyse de ces implémentations permet d'interpréter les choix de conceptions effectués et ainsi déterminer si l'implémentation est correcte. De telles analyses sont toutefois fastidieuses voire impossibles sans une représentation des connaissances facilitant l'identification des choix de conception effectués dans l'unité de sélection utilisée.

Nous soutenons que l'expression de tels choix est facilitée si l'on utilise un modèle de sélection d'interaction réactif :

- caractérisant explicitement dans le modèle comment est sélectionnée l'interaction initiée ainsi que l'agent qu'elle prend pour cible ;
- fournissant une représentation unifiée de comportements stochastiques, dirigés des préférences ou par des priorités.

En effet, une telle approche permet d'exprimer grande variété de comportements réactifs et fait donc apparaître un grand nombre de choix de conception.

Nous définissons à cet effet une extension du modèle réactif de sélection d'interaction de Ioda. Cette extension consiste à construire le comportement d'un agent en associant une politique de sélection à chaque priorité apparaissant dans sa ligne de la matrice d'interaction raffinée. Chaque politique de sélection est construite à l'aide de six briques comportementales élémentaires. Les trois premières décrivent en quoi consiste la politique de sélection :

- sélectionner un couple interaction/cible OU ;
- sélectionner une interaction puis une cible selon des modalités différentes OU ;
- sélectionner une cible puis une interaction selon des modalités différentes.

Les trois autres briques décrivent la nature de la sélection d'un couple, d'une interaction ou d'une cible :

- une sélection aléatoire OU ;
- une sélection basée sur l'attribution de préférences OU ;
- une sélection basée sur une pondération.

La spécification fine du comportement des agents consiste alors à associer à chaque priorité une brique comportementale décrivant en quoi consiste la sélection, et à associer à chacune de ces briques un ensemble de briques élémentaires décrivant comment la sélection des couples, des interactions, ou des cibles est effectuée.

Chapitre 8

Héritage et réutilisation de modèle dans IODA

Plan du chapitre :

La construction de simulations large échelle ne peut être envisagée sans profiter d'outils d'ingénierie logicielle similaires à l'héritage. En effet, ces outils permettraient d'éviter de fournir une spécification systématiquement exhaustive de la source et la cible de chaque interaction.

Dans ce chapitre, nous étudions sous quelles conditions il est possible d'intégrer des concepts proches de l'héritage dans l'approche IODA *afin de faciliter l'ingénierie des connaissances. A cette occasion, nous définissons une extension de* IODA *fondée sur la relation de spécialisation qui facilite l'ingénierie des connaissances dans une simulation. En effet, la spécialisation exprime une relation sémantique entre familles d'agents qui permet :*

- *de représenter des matrices d'interaction brutes exprimant un grand nombre d'interactions entre agents à l'aide d'une quantité réduite d'éléments ;*
- *d'exprimer qu'une famille d'agents est une extension d'une autre famille d'agents. Par exemple « une plante carnivore est une plante sachant de plus manger des insectes » ;*
- *d'exprimer qu'une famille d'agents est une exception à une autre famille d'agents. Par exemple « un client aveugle est un client ne sachant pas lire de panneaux publicitaires ».*

D'une manière générale, plus le modèle d'une simulation contient d'informations, plus on est susceptible d'y retrouver des redondances, *i.e.* des portions de modèle réutilisées en plusieurs endroits. L'un des principaux enjeux de la conception de telles simulations est alors de structurer le modèle afin de faciliter la spécification de ces portions du modèle et, si possible, à les réutiliser au maximum.

Une première approche favorisant la réutilisation dans le modèle consiste à le diviser en un maximum d'unités disjointes les plus indépendantes possible et à réifier chacune de ces unités en un élément logiciel. Cette séparation permet de constituer des bibliothèques d'éléments pouvant être réutilisés autant lors de la construction du modèle que lors de son implémentation. C'est par exemple le cas dans la plateforme Volcano [Dem95], où un agent est conçu comme une agrégation de briques logicielles décrivant respectivement : son comportement, comment il communique avec les autres agents, comment il communique avec l'environnement et quels sont ses liens sociaux avec les autres agents. Il en va de même pour la plateforme JADE [BPR99], où le comportement d'un agent est décomposé en activités, implémentées chacune à l'aide d'une instance de la classe Behavior. et où le comportement est décrit par l'utilisateur en spécifiant l'activation séquentielle, cyclique ou parallèle de ces activités. On retrouve aussi ce principe de décomposition dans la plateforme Jack [BHRH00], où les agents sont définis par des plans, des ensembles de croyances et des buts. Ces éléments sont tous représentés sous la forme d'une entité logicielle autonome (une classe) permettant ainsi leur réutilisation pour des agents différents. L'approche IODA décrite dans cette thèse se place aussi dans cette perspective, mais va plus loin en fournissant les outils permettant non seulement de réifier les interactions en unités logicielles autonomes, mais aussi de les décrire de manière générique et indépendante des spécificités des agents, ce qui favorise leur réutilisation dans des agents

aux spécificités très différentes.

Chaque unité élémentaire constituant le modèle représente une unité sémantique de base permettant de décrire le phénomène. Bien que ces unités soient décrites de manière disjointe, certaines peuvent être regroupées et concourir à la description d'un concept plus complexe. Par exemple :

- le fait de savoir chasser un autre agent et de le manger caractérise un prédateur ;
- le fait de vieillir et mourir caractérise un être vivant ;
- le fait de savoir se déplacer caractérise une entité mobile, *etc.*

Une deuxième approche favorisant la réutilisation dans le modèle consiste à décrire les agents à l'aide de ces regroupements sémantiques. Dans cette section, nous étudions cette question du point de vue du modèle et de l'implémentation d'une simulation, en nous appuyant sur les concepts développés dans IODA.

8.1 Héritage en simulation multi-agents

En génie logiciel, le concept s'approchant le plus de ce type de descriptions est l'héritage, utilisé en premier lieu en programmation orientée-objet. L'héritage est une relation binaire liant un objet "père" à un objet "fils", statuant que l'objet "fils" est une instance particulière de l'objet "père". En termes de réutilisation, cela implique que tous les attributs et toutes les méthodes figurant dans l'objet "père" doivent figurer dans l'objet "fils".

Un grand nombre de plateformes permettant d'implémenter des systèmes multi-agents reposent sur des langages orientés-objets. Dans ces plateformes, un agent est en général implémenté sous la forme d'une classe ou un objet. En conséquence, le concept d'héritage peut leur être appliqué, permettant ainsi d'une part de fournir une structure aux liens sémantiques existant entre les divers agents d'une simulation, mais aussi de permettre dans une certaine mesure leur réutilisation. Caractériser une telle relation entre deux agents au niveau du modèle s'avère plus problématique. En effet, l'héritage des langages orientés-objet s'exprime en termes de méthodes et attributs réutilisés, alors que dans le cas d'agent, on aimerait exprimer la réutilisation en termes de façon de percevoir, en termes de contenu de la mémoire, en termes d'interactions qu'un agent est capable d'initier, ou en termes de modèle de sélection d'interaction.

La construction de la sémantique d'un modèle commence inévitablement par l'identification des identifiants des familles d'agents, ainsi que les identifiants des interactions impliquées dans le phénomène. Lors de cette étape, les familles d'agents peuvent être utilisées pour exprimer des concepts du phénomène très différents :

1. une espèce, un genre, une famille, un ordre, une classe ou un embranchement, au sens des sciences du vivant. Par exemple `Végétaux`, `Animaux`, `Minéraux`, *etc.*
2. une catégorie particulière d'individus ayant des caractéristiques communes. Par exemple `AgentAvecInventaire`, qui représente l'ensemble des agents pouvant contenir quelque chose. Cette famille d'agents caractérise alors autant un `Facteur` qu'une `Remorque` ou une `Corbeille` ;
3. une catégorie particulière d'individus ayant des capacités communes. Par exemple, un `Être vivant` est capable de MOURIR et VIEILLIR. Cette famille d'agents caractérise alors autant un `Loup` qu'une `Personne` ou une `Plante` ;
4. une catégorie particulière d'individus ayant leur comportement en commun. Par exemple, un `Virus` a pour comportement de SE DÉPLACER dans un corps humain, PÉNÉTRER dans une `Cellule`, puis SE DUPLIQUER dans cette `Cellule`. Cette famille d'agents caractérise des virus pouvant avoir des effets très différents sur la cellule infectée, allant de l'INHIBITION de certaines de ses fonctions (par exemple dans le cas du `VIH`), à la FUSION de cellules (par exemple dans le cas du `Virus de Sendai`) ;
5. toute combinaison des quatre concepts précédents.

La notion de *capability* de la plateforme Jack [BHRH00] est particulièrement intéressante de ce point de vue. En effet, ce concept permet de structurer les éléments relatifs au raisonnement des agents (plans, croyances et buts) en blocs sémantiques. Un agent associé à une *capability* dispose de tous les plans, croyances et buts qu'elle contient. Une *capability* peut de plus hériter d'autres *capabilities* et donc disposer de tous les plans, croyances et buts qu'elle contient. Un agent peut alors être construit en décrivant dans un premier temps des fonctionnalités abstraites à l'aide de *capabilities*, puis en attribuant ces fonctionnalités

aux agents de la simulation. Ainsi, des librairies de *capabilities* réutilisables sont construites. Ces dernières simplifient la conception d'applications pouvant contenir un grand nombre d'agents.

Cette approche ne fait pas apparaître les interactions ayant lieu entre agents. La solution proposée est donc susceptible d'éluder un certain nombre de problèmes, que nous proposons d'étudier ci-après.

8.1.1 Nature des problèmes liés aux interactions

Deux types de problèmes de modélisation sont rencontrés lors de la spécification de grandes matrices d'interactions. L'un d'entre eux est lié à l'énumération exhaustive de chaque cible d'une interaction pour une famille d'agents source donnée, l'autre est lié à la re-spécification systématique des interactions pouvant être initiées par différentes familles d'agents sources ayant pourtant une souche commune. Dans cette section, nous illustrons en quoi consistent ces problèmes et fournissons l'idée générale de la solution que nous employons pour y remédier.

Problèmes liés aux cibles

Exprimer explicitement quelles sont les cibles d'une interaction induit un double problème. Lorsque qu'une famille d'agent source a la capacité d'initier une interaction individuelle ou de multicast avec un grand nombre de famille d'agents différentes pour cible, spécifier la matrice d'interaction devient fastidieux, puisque **l'interaction doit figurer dans la colonne de chacune des familles d'agents cibles**. De plus, cette approche est peu robuste à l'ajout de nouvelles familles d'agents devant elles aussi être la cible de l'interaction. Prenons l'exemple d'un `Lymphocyte`, dont le rôle dans le corps est de PHAGOCYTER le `Virus Rhume`, `Virus Grippe` et `Virus Varicelle`. Ces trois virus ont tous trois un comportement différent que nous supposons déjà spécifié. Une telle simulation est modélisée à l'aide de la matrice d'interaction brute décrite dans la figure 8.1(a). Si jamais une nouvelle famille d'agents VIRUS ROUGEOLE devait être ajoutée, alors la ligne de la matrice d'interaction brute du Lymphocyte doit être modifiée (voir figure 8.1(b)), bien que son comportement reste rigoureusement identique : la phagocytose des virus.

Source \ Cible	$\emptyset$	Virus Rhume	Virus Grippe	Virus Varicelle
Lymphocyte	(SE DÉPLACER)	(PHAGOCYTER, d = 0)	(PHAGOCYTER, d = 0)	(PHAGOCYTER, d = 0)

(a) Matrice décrivant la base d'un comportement de Lymphocyte

Source \ Cible	$\emptyset$	Virus Rhume	Virus Grippe	Virus Varicelle	Virus Rougeole
Lymphocyte	(SE DÉPLACER)	(PHAGOCYTER, d = 0)	(PHAGOCYTER, d = 0)	(PHAGOCYTER, d = 0)	(PHAGOCYTER, d = 0)

(b) Matrice décrivant l'extension du comportement de Lymphocyte à la phagocytose d'un nouveau virus

FIGURE 8.1 – Matrices d'interaction brutes décrivant d'une part le comportement d'un `Lymphocyte` vis-à-vis du `Virus Rhume`, du `Virus Grippe` et du `Virus Varicelle` (figure 8.1(a)) et d'autre part l'extension de ce comportement au `Virus Rougeole` (figure 8.1(b)).

Pour remédier à ce problème, nous étendons l'approche IODA en lui ajoutant la *relation de spécialisation* entre une famille d'agent "fille" et une famille d'agents "mère". Définir une famille d'agents "mère" en tant que cible d'une interaction dans la matrice d'interaction brute est alors un raccourci spécifiant que cette famille d'agents, ainsi que toutes les familles d'agents spécialisant cette famille d'agents "mère" peuvent être la cible de cette interaction. Dans l'exemple du `Lymphocyte`, la famille d'agents `Virus` est spécialisée par les familles d'agents `Virus Rhume`, `Virus Grippe`, `Virus Varicelle` et `Virus Rougeole`. Cette relation simplifie la spécification de la matrice d'interaction brute (voir figure 8.2) et la rend plus robuste aux modifications. En effet, l'ajout d'une nouvelle espèce de `Virus` ne nécessite pas la modification du comportement du `Lymphocyte`.

Problèmes liés aux sources

Dans IODA, une ligne de la matrice d'interaction brute doit être définie pour chaque famille d'agents ayant un comportement autonome. Certaines de ces familles peuvent avoir beaucoup en commun et

Source \ Cible	∅	Virus
Lymphocyte	(SE DÉPLACER)	(PHAGOCYTER, d = 0)

FIGURE 8.2 – Matrice d'interaction brute décrivant le comportement d'un `Lymphocyte` vis à vis d'une famille d'agents `Virus`, qui est spécialisée par les familles d'agents `Virus Rhume`, `Virus Grippe`, `Virus Varicelle` et `Virus Rougeole`. Cette matrice est équivalente à celle présentée dans la figure 8.1

l'adaptation d'un concept proche des *capabilities* de Jack permettrait de simplifier leur conception. Prenons l'exemple d'une simulation en éthologie où évoluent trois espèces animales et une espèce végétale, représentées respectivement à l'aide d'une famille d'agents `Aigle`, `Faucon`, `Rongeur` et `Végétation`. Cette simulation modélise les relation de proie et prédation entre les différentes familles d'agents constituant la simulation, où les entités peuvent de plus vieillir et mourir de vieillesse. La matrice d'interaction brute et la matrice de mise à jour d'une telle simulation sont résumées sur la figure 8.3. Dans ce modèle, il y a beaucoup de redondances : toutes les familles d'agents peuvent initier l'interaction MOURIR et l'interaction de mise à jour VIEILLIR, trois familles d'agents peuvent initier l'interaction de mise à jour AUGMENTER SENSATION FAIM ainsi que l'interaction SE DÉPLACER. Deux plus, deux familles d'agents représentent des espèces animales apparentées, qui partagent une partie de leur régime alimentaire.

Source \ Cible	∅	Aigle	Faucon	Rongeur	Végétation
Aigle	(SE DÉPLACER) (MOURIR)	(SE REPRODUIRE, d = 0)	(MANGER, d = 0)	(MANGER, d = 0)	
Faucon	(SE DÉPLACER) (MOURIR)		(SE REPRODUIRE, d = 0)	(MANGER, d = 0)	
Rongeur	(SE DÉPLACER) (MOURIR)			(SE REPRODUIRE, d = 0)	(MANGER, d = 0)
Végétation	(MOURIR) (SE PROPAGER)				

(a) Matrice d'interaction brute

	Aigle	Faucon	Rongeur	Végétation
Interactions de mise à jour	VIEILLIR AUGMENTER SENSATION FAIM	VIEILLIR AUGMENTER SENSATION FAIM	VIEILLIR AUGMENTER SENSATION FAIM	VIEILLIR

(b) Matrice de mise à jour

FIGURE 8.3 – Matrice d'interaction brute et matrice de mise à jour d'une simulation en éthologie contenant les familles d'agents AIGLE, FAUCON, RONGEUR et VÉGÉTATION.

Il serait donc possible de le structurer afin d'y favoriser l'ajout futur d'autres espèces animales. La relation de spécialisation mentionnée précédemment peut jouer ce rôle. Définir qu'une famille d'agents "mère" est capable d'initier une interaction est alors un raccourci spécifiant que toutes les familles d'agents spécialisant la famille d'agents "mère" sont aussi capables d'initier cette interaction. Dans l'exemple de l'écosystème mentionné ici, cela reviendrait par exemple à :

- créer une famille d'agents `Rapace` qui rassemble les interactions communes aux familles d'agents `Faucon` et `Aigle` ;
- créer une famille d'agents `Animal` qui rassemble les interactions communes aux familles d'agents `Aigle`, `Faucon` et `Rongeur` ;
- créer une famille d'agents `Vivant` qui rassemble les interactions communes aux familles d'agents `Aigle`, `Faucon`, `Rongeur` et `Végétation`.

Cela aboutirait alors à une matrice d'interaction similaire à celle présentée sur la figure 8.4.

Problèmes inhérents à la spécialisation

L'utilisation de la spécialisation telle que schématisée dans les deux sections précédentes allège la construction d'une matrice d'interaction brute. Elle favorise de plus la réutilisation logicielle de lignes de la matrice déjà spécifiées. Toutefois, ces notations induisent un certain nombre de problèmes qu'il est nécessaire de résoudre afin de pouvoir utiliser ce concept sans ambiguïtés.

Source \ Cible	$\emptyset$	Aigle :: Rapace	Faucon :: Rapace	Rongeur :: Animal	Végétation :: Vivant
Vivant	(Mourir)				
Végétation :: Vivant	(Se Propager)				
Animal :: Vivant	(Se Déplacer)				
Rongeur :: Animal				(Se Reproduire, d = 0)	(Manger, d = 0)
Rapace :: Animal				(Manger, d = 0)	
Aigle :: Rapace		(Se Reproduire, d = 0)	(Manger, d = 0)		
Faucon :: Rapace			(Se Reproduire, d = 0)		

(a) Matrice d'interaction brute

	Vivant	Animal :: Vivant
Interactions de mise à jour	Vieillir	Augmenter Sensation Faim

(b) Matrice de mise à jour

Figure 8.4 – Matrice d'interaction brute et matrice de mise à jour d'une simulation en éthologie équivalente à celle décrite dans la figure 8.3. Toutefois, le modèle y est exprimé à l'aide de la spécialisation de familles d'agents, permettant l'ajout plus simple de nouvelles familles d'agents. Dans cette figure, la relation "::" spécifie que la famille d'agents présente dans le membre de gauche spécialise les familles d'agents présentes dans le membre de droite. "`Aigle` :: `Rapace`" signifie alors qu'un `Aigle` est une sorte de `Rapace` pouvant en plus Manger des `Faucons` et Se Reproduire avec d'autres `Aigles`.

Classes abstraites et méthodes abstraites Le concept de spécialisation pose d'abord problème au niveau des primitives à spécifier. En effet, certaines familles d'agents sont trop générales pour pouvoir fournir une spécification sensée des primitives abstraites des interactions qu'elle peut subir ou effectuer. C'est le cas de l'exemple fourni dans la section 8.1.1, où la famille `Virus` doit fournir une spécification aux primitives liées aux cibles de l'interaction Phagocyter, alors que ce code peut être spécifique à chacune des familles `Virus Rhume`, `Virus Grippe`, *etc.* Il faudrait donc pouvoir ne pas fournir de spécification à certaines primitives, ce qui n'est actuellement pas possible dans le modèle Ioda présenté dans le chapitre 3.

Héritage multiple Le second problème est aussi lié aux primitives. Il est rencontré lorsqu'une famille d'agents spécialise deux familles d'agents pouvant initier la même interaction, ou subir la même interaction. Dans cette situation, si les deux familles d'agents "mères" fournissent une spécification aux primitives, il faut alors pouvoir déterminer laquelle de ces spécifications doit être utilisée dans la famille d'agents "fille". Ce problème est similaire au problème du losange rencontré en conception orientée-objet. Il faut donc trouver un moyen permettant de décider quelles spécifications des primitives sont réutilisées par la famille d'agents "fille".

Dispersion de la connaissance Le troisième problème survient lorsqu'une famille d'agents spécialise un nombre élevé de familles d'agents. Dans de tels cas, la réutilisation de modèles existants est grandement favorisée, au prix de la facilité de compréhension du comportement des agents. En effet, les interactions que cette famille d'agents peut initier ou subir sont éparpillées dans des familles d'agents très différentes, si bien qu'il est difficile d'avoir une vue d'ensemble sur toutes les interactions auxquelles peut participer cette famille d'agents. Il faudrait donc trouver un compromis entre expressivité de la matrice d'interaction brute et possibilité de réutiliser des comportements déjà existants.

8.1.2 Héritage et ingénierie des connaissances

Une transposition naïve du concept d'héritage des langages orientés-objet à la simulation consiste à construire les familles d'agents en :

– récupérant l'ensemble des spécifications effectuées dans les familles d'agents "mères" ;
– complétant éventuellement ces spécifications par l'ajout de nouveaux éléments d'assignation et primitives.

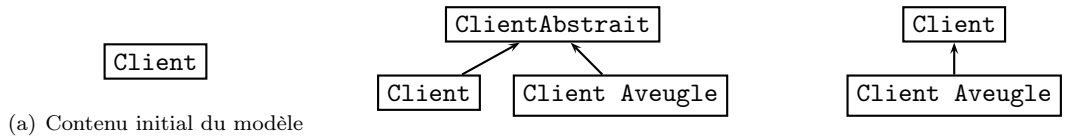
Selon cette approche, si deux familles d'agents ont en commun une partie des interactions qu'elles sont capables d'initier, alors une famille d'agents intermédiaire doit être créée. Son rôle est de définir le tronc commun de ces deux familles d'agents. Dans cette section, nous illustrons succinctement certains problèmes sous-jacents à cette approche et décrivons une approche différente qui permettrait de s'en détacher.

Considérons une simulation reproduisant le comportement (simplifié) de `Clients` dans un magasin. Ce comportement consiste à SE DÉPLACER, LIRE les informations affichées sur un `Panneau Publicitaire`, RAMASSER des `Articles` et PAYER à une `Caisse` les articles achetés. Dans cette simulation, les `Caisses` peuvent de plus SIGNALER LEUR OUVERTURE à un `Client`. La matrice d'interaction brute de cette simulation est résumée sur la figure 8.5.

Source \ Cible	∅	Panneau Publicitaire	Article	Caisse	Client
Client	(SE DÉPLACER)	(LIRE, d = 5)	(RAMASSER, d = 0)	(PAYER, d = 0)	
Caisse					(SIGNALER OUVERTURE, d = 3)

FIGURE 8.5 – Matrice d'interaction brute décrivant ce qu'un `Client` et une `Caisse` sont capables de faire dans un magasin.

Supposons maintenant qu'un raffinement du modèle amène à l'ajout au modèle de `Clients Aveugles`. Ces derniers sont des clients ne pouvant pas LIRE de `Panneaux Publicitaires` et percevant leur voisinage dans des modalités différentes des clients ne souffrant pas de ce handicap. Dans la suite, nous décrivons deux approches différentes permettant de modéliser cette modification du modèle. Nous nous focalisons pour cela sur la relation de existant entre la famille d'agents `Client` et la famille d'agents `Client Aveugle`. Les idées principales de ces deux approches sont résumées sur la figure 8.6.



(a) Contenu initial du modèle

(b) Modification du modèle en utilisant l'approche classique, qui implique la modification de la famille d'agents `Client`

(c) Modification souhaitable du modèle, qui n'implique pas la modification de la famille d'agents `Client`

FIGURE 8.6 – Illustration du problème de modélisation lié à l'ajout d'une famille d'agents dont le comportement est une exception au comportement d'une autre famille d'agents. Ce problème est illustré dans le cas de la modélisation du comportement d'un `Client Aveugle` dans un magasin, dont le comportement est identique à celui d'un `Client`, hormis le fait qu'il n'est pas capable d'initier des interactions telles que LIRE avec un `Panneau Publicitaire` pour cible. Dans cette figure, une flèche allant de A vers B signifie que A réutilise tout (figure 8.6(b)) ou partie (figure 8.6(c)) du comportement et des interactions de B.

La pratique classique du génie logiciel voudrait la création d'une famille d'agents intermédiaire (par exemple `ClientAbstrait`) qui regroupe les capacités d'interagir communes aux familles d'agents `Client` et `Client Aveugle`. Il faudrait de plus s'assurer que les deux familles d'agents "héritent" de la famille `ClientAbstrait`. Cette façon d'organiser les familles d'agents est résumée sur la figure 8.6(b). Cette description aboutirait alors à la matrice d'interaction brute présentée sur la figure 8.7.

Dans cet exemple, l'ajout de la famille d'agents `Client Aveugle` implique :
– la modification de la ligne de la famille d'agents `Client` dans la matrice d'interaction ;
– la création d'une famille d'agents `ClientAbstrait` qui regroupe ces interactions communes à `Client` et `Client Aveugle`.

Cette modification a pour conséquence que les `Caisses` ne peuvent pas signaler leur ouverture aux `Clients Aveugles`.

Source \ Cible	∅	Panneau Publicitaire	Article	Caisse	ClientAbstrait
ClientAbstrait	(SE DÉPLACER)		(RAMASSER, d = 0)	(PAYER, d = 0)	
Client :: ClientAbstrait	—	(LIRE, d = 5)	—	—	
Client Aveugle :: ClientAbstrait					
Caisse					(SIGNALER OUVERTURE, d = 3)

FIGURE 8.7 – Matrice d'interaction brute décrivant ce qu'un `Client`, un `Client Aveugle` et une `Caisse` sont capables de faire dans un magasin. Cette matrice est construite en factorisant les capacités d'interaction communes aux familles d'agents `Client` et `Client Aveugle` au sein d'une troisième famille d'agents `ClientAbstrait`. Les modifications du modèle initial présenté dans la figure 8.5 apparaissent en rouge dans la présente figure.

Pour corriger ce problème, toute occurrence de la famille d'agents `Client` en tant que cible d'une interaction a du être remplacée par la nouvelle famille d'agents `ClientAbstraite`. Cela implique la modification des lignes de la matrice d'interaction associées à chaque famille d'agents pouvant auparavant interagir avec des `Clients`. Enfin, la création d'une famille d'agents supplémentaire et l'introduction de relations de spécialisation supplémentaires dispersent encore plus la connaissance des agents.

De telles situations surviennent dès que l'on cherche à définir le comportement d'un agent qui constitue une exception. Dans de tels cas, plutôt que de tenter de factoriser les points communs des deux familles d'agents, il peut s'avérer fructueux d'utiliser une approche inverse où l'on caractérise une famille d'agents par les interactions n'étant pas héritées de la famille d'agents mère. Dans l'exemple développé ici, cela reviendrait à mettre en exergue qu'un `Client Aveugle` est un `Client` n'étant pas capable de LIRE des `Panneaux Publicitaires`. Cette façon de modéliser se traduirait alors par une matrice d'interaction brute prenant la forme illustrée sur la figure 8.8. Cette façon d'organiser les familles d'agents est résumée sur la figure 8.6(c).

Source \ Cible	∅	Panneau Publicitaire	Article	Caisse	Client
Client	(SE DÉPLACER)	(LIRE, d = 5)	(RAMASSER, d = 0)	(PAYER, d = 0)	
Client Aveugle :: Client		−(LIRE, d = 5)			
Caisse					(SIGNALER OUVERTURE, d = 3)

FIGURE 8.8 – Matrice d'interaction brute décrivant ce qu'un `Client`, un `Client Aveugle` et une `Caisse` sont capables de faire dans un magasin. Dans cette matrice, le signe − peut apparaître devant un élément d'assignation, ce qui signifie que l'interaction lui étant associée est retirée des interactions pouvant être initiées ou subies par la famille d'agents. Ainsi, la ligne associée à la famille d'agents `Client Aveugle` se lit « Un `Client Aveugle` est un `Client` particulier (*i.e.* Client Aveugle ::Client) » ne sachant pas LIRE de `Panneaux Publicitaires` (*i.e.* −(LIRE, d = 5)). Les modification du modèle initial présenté dans la figure 8.5 apparaissent en rouge dans la présente figure.

Ce changement de perspective se révèle particulièrement intéressant lorsque l'on souhaite concevoir des familles d'agents dont le comportement constitue une exception du comportement d'une autre famille d'agents, par exemple lorsqu'il s'agit de modéliser le comportement d'une `Personne Aveugle` à partir du comportement d'une `Personne` qui n'est pas aveugle, le comportement d'un `Animal Stérile` à partir du comportement d'un `Animal` qui n'est pas stérile *etc.* En effet, il ne nécessite ni l'ajout d'une nouvelle famille d'agents intermédiaire, ni la modification de la matrice d'interaction brute.

8.2 Spécialisation de familles d'agents dans IODA/JEDI

Pour répondre aux diverses problématiques mentionnées ci-avant, nous transposons dans un premier temps à IODA les principales notions liées à l'héritage dans les langages orientés-objets. Nous décrivons ensuite comment profiter de la spécification synthétique des agents permise par l'héritage sans pour autant souffrir de la dispersion des connaissances. Cette approche permet ainsi concilier ingénierie des

connaissances et ingénierie logicielle. Enfin, nous décrivons quels concepts permettent de spécifier les familles d'agents selon une approche radicalement opposée à l'héritage des langages orientés-objets : concevoir des exceptions à certaines familles d'agents, en modifiant ou en supprimant certaines de leurs capacités à interagir.

8.2.1 Spécialisation et héritage

Dans cette extension du corps de l'approche IODA, la relation d'héritage des langages orientés-objets est transposée en une relation que nous appelons *spécialisation*.

Définition 48. ***Spécialisation***
La **spécialisation** est une relation binaire entre une famille d'agents appelée **fille** et une famille d'agents appelée **mère**. Elle exprime le fait que la famille d'agents fille constitue un sous-ensemble particulier de la famille d'agents mère.

Cette relation est exprimée dans la méthodologie IODA lors de la construction de l'ensemble des identifiants des familles d'agents. En plus de la construction de la liste des identifiants, cette étape établit les relations de spécialisation entre familles d'agents sous la forme d'un graphe similaire à un diagramme de classes : les noeuds y sont des identifiants de familles d'agents et les arcs représentent une relation de spécialisation. Les arcs ont pour origine une famille d'agents fille et pointent vers une famille d'agents mère. Un tel graphe est illustré sur la figure 8.2.1 page 228 pour une simulation d'un écosystème dont la matrice d'interaction est décrite dans la figure 8.4.

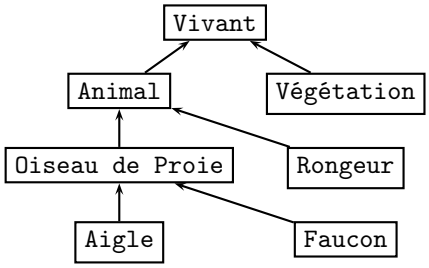


FIGURE 8.9 – Graphe dirigé résumant le lien de spécialisation entre familles d'agents, dans le cas de la simulation d'un écosystème déjà décrit sous la forme d'une matrice d'interaction brute dans la figure 8.2.1 page 228. Une flèche y représente une relation de spécialisation entre deux familles d'agents, qui pointe vers la famille d'agent mère de la relation.

La spécialisation s'exprime aussi dans la matrice d'interaction brute, dans la matrice de mise à jour, et dans la matrice de mise à jour ordonnée, en postfixant le nom de chaque famille d'agents par les caractères "::", suivis d'une liste de familles d'agents mères. Par exemple, dans la figure 8.4(b), l'expression « `Animal`::`Vivant` » spécifie que la famille d'agents `Animal` spécialise la famille d'agents mère `Vivant`. Si jamais la famille d'agents ne spécialise aucune famille d'agents particulière, les caractères "::" n'apparaissent pas. La relation de spécialisation est transitive. Par conséquent, pour éviter de surcharger le contenu de la matrice d'interaction brute, seules les familles d'agents mères étant directement en relation avec la famille d'agents fille sont mentionnées dans l'entête de chaque ligne et de chaque colonne de la matrice.

Cette relation a deux conséquences dans la matrice d'interaction. Premièrement, **une famille d'agents *fille* peut subir toutes les interactions que la famille d'agents *mère* peut subir**. Des regroupements sémantiques de familles d'agents sont donc créés. Ils permettent de désigner plus simplement un ensemble de familles d'agents cibles à l'aide d'un seul élément d'assignation. Cela allège d'une part la description du contenu de la matrice d'interaction brute et cela facilite d'autre part l'intégration de nouvelles familles d'agents au modèle sans avoir à modifier les familles déjà existantes. Cette première utilisation de la spécialisation est illustré dans la section 8.1.1 pour une simulation où un `Lymphocyte` peut PHAGOCYTER des `Virus` tels que le `Virus Rougeole` ou le `Virus Rhume`.

La seconde conséquence de cette relation est qu'**une famille d'agents *fille* récupère par défaut toutes les interactions de la matrice d'interaction brute que la famille d'agents *mère* est capable d'initier**. Le même principe s'applique aux interactions présentes dans la matrice de mise à jour. Cette seconde utilisation de la spécialisation est illustrée dans la section 8.1.1 pour la simulation d'un écosystème.

Dans les deux cas, la famille d'agents fille doit spécifier les primitives issues des interactions que la famille d'agents mère peut subir ou effectuer. Dans le cas le plus simple, la spécification de ces primitives consiste à réutiliser les spécifications fournies dans la famille d'agents mère ou, s'il en est besoin, de surcharger certaines primitives pour leur fournir un comportement plus spécifique. En pratique, deux cas récurrents échappent à ces principes simples de spécification.

Familles d'agents abstraites

Le premier cas posant problème dans la relation de spécialisation est rencontré si la famille d'agent mère ne peut pas fournir de spécification pertinente à toutes les primitives, car elle représente un groupement trop abstrait d'agents (il s'agit du premier problème inhérent à la spécialisation mentionné dans la section 8.1.1). Pour traiter ce genre de cas, nous introduisons le concept de *famille d'agents abstraite*, où certaines primitives peuvent ne pas être spécifiées.

Définition 49. ***Famille d'agents abstraite***
Une **Famille d'agents abstraite** est une famille d'agents ne fournissant pas de spécifications à au moins l'une de ses primitives. Ce type de famille d'agents ne peut être instanciée directement.

Dans la méthodologie IODA, déterminer si une famille d'agents est abstraite ou non se fait lors de l'étape de spécification de leurs primitives. Pendant cette étape, le concepteur peut décider de ne pas fournir de spécifications à une primitive, dans quel cas la famille d'agents devient obligatoirement abstraite. Si jamais le concepteur spécifie toutes les primitives de la famille d'agents, alors cette famille n'est plus abstraite.

Problème de l'héritage multiple

Le second cas posant problème dans la relation de spécialisation est rencontré lorsque la famille d'agents fille spécialise plus d'une famille d'agents, et que ces familles fournissent des spécifications différentes à une même primitive (il s'agit du deuxième problème inhérent à la spécialisation mentionné dans la section 8.1.1). Ce problème, aussi connu sous le nom de l'héritage multiple en programmation orientée-objet, consiste à déterminer quel moyen utiliser pour identifier quelle spécification fournir aux primitives d'une famille d'agents fille. De nombreuses techniques permettent de résoudre ce problème en programmation orientée-objet.

La première est trouvée par exemple dans le langage Perl [Wal10]. Elle consiste à ordonner l'ensemble des classes mères lors de la déclaration de l'objet et à utiliser la première spécification de la primitive rencontrée en itérant sur cet ensemble. Une autre spécification, retrouvée par exemple dans Scala [LAM10] ou Python [Fou10], consiste à remonter dans l'arbre d'héritage en effectuant un parcours en largeur d'abord et à utiliser la spécification de la première primitive rencontrée.

Dans le cas de la simulation, de telles sélections automatisées sont à éviter. En effet, elles reviennent à définir de manière arbitraire la sémantique du modèle. La solution considérée dans IODA consiste à faire ce choix manuellement dans la méthodologie de conception lors de l'étape de spécification des primitives. Ce choix de conception se retrouve par exemple dans le langage C++ [SKM02].

Puisque certaines familles d'agents mères peuvent être abstraites, elles peuvent ne pas fournir de spécification à certaines primitives. Par conséquent, avant de pouvoir choisir la spécification d'une primitive dans une famille d'agents fille, il faut au préalable effectuer l'étape de spécification des primitives de toutes ses familles d'agents mère. Pour cela, nous imposons un ordre dans lequel les agents se voient spécifier leurs primitives. Il s'agit d'un parcours en largeur d'abord des liens de spécialisation, dont le point de départ sont les familles d'agents ne spécialisant aucune autre famille d'agents.

Synthèse

Dans cette section, nous avons caractérisé la spécialisation comme un moyen permettant de construire les familles d'agents à l'aide d'un moyen proche de l'héritage dans le systèmes multi-agents, que l'on retrouve par exemple avec la notion de capability dans Jack. Cette caractérisation de la spécialisation répond aux problèmes relatifs aux sources et aux cibles ainsi qu'aux deux premiers problèmes inhérents à la spécialisation mentionnés dans la section 8.1.1.

8.2.2 Forme synthétique et forme étendue

La spécialisation permet d'exprimer sous forme synthétique l'ensemble des interactions que des agents sont capables d'initier ou subir ainsi que l'ensemble de leurs interactions de mise à jour. Cette relation facilite ainsi la conception de la matrice d'interaction brute et la factorisation de code dans les agents. Toutefois, cet apport logiciel a un coût du point de vue de l'ingénierie des connaissances : la matrice d'interaction brute devient moins facile à interpréter. En effet, les interactions pouvant être initiées ou subies par une famille d'agents sont éparpillées parmi ses différentes familles d'agents mères. Ingénierie des connaissances et ingénierie logicielle sont toutes deux fondamentales pour la conception de simulations contenant un grand nombre d'informations. Par conséquent, la méthodologie de conception doit fournir le meilleur compromis possible entre l'utilisation de la relation de spécialisation et la possibilité d'interpréter facilement le modèle construit.

Dans IODA, nous choisissons de reposer sur deux représentations équivalentes d'une matrice d'interaction brute, se focalisant respectivement sur l'aspect représentation des connaissances et l'aspect ingénierie logicielle de la simulation. Nous appelons ces deux formes des matrices d'interaction *forme synthétique* et *forme étendue*.

Principes

La spécialisation est un moyen permettant de construire facilement de grandes matrices d'interaction brutes à l'aide d'un faible nombre d'éléments d'assignation. Elle transpose à une approche centrée-interactions la notion d'héritage telle qu'on la rencontre en programmation orientée-objets. Les matrices d'interaction reposant sur cette relation afin de réduire le nombre d'éléments d'assignations qu'elles contiennent sont dites sous *forme synthétique*. Une matrice d'interaction brute sous une telle forme est illustrée sur la figure 8.4(a) de la page 225 pour une simulation simple d'un écosystème.

Définition 50. ***Forme synthétique***
Une matrice d'interaction brute est dite sous **forme synthétique** si elle fait apparaître des liens de spécialisation entre familles d'agents et le moins possible d'interactions.

L'interprétation d'une matrice d'interaction n'est pas aisée lorsqu'un nombre important de relations de spécialisation entre familles d'agents apparaissent. En effet, les interactions pouvant être effectuées ou subies par un agent sont éparpillées entre les très nombreuses familles d'agents mères. Afin de faciliter cette interprétation, nous considérons dans IODA une forme équivalente à la matrice d'interaction brute sous forme synthétique, appelée *forme étendue*. Cette forme fait apparaître de manière exhaustive les interactions pouvant survenir entre toutes les familles d'agents de la simulation. La forme étendue de la simulation d'un écosystème simple mentionnée dans le paragraphe précédent est illustrée sur la figure 8.10 de la page 231. Dans le cas de la simulation de la phagocytose de virus dont la matrice d'interaction brute sous forme synthétique est décrite dans la figure 8.2, la forme étendue correspond à la matrice présente sur la figure 8.11.

Définition 51. ***Forme étendue***
La **Forme étendue** d'une matrice d'interaction brute sous forme synthétique est une matrice faisant état de manière exhaustive de toutes les interactions pouvant avoir lieu entre les différentes familles d'agents d'une simulation.

Dans la méthodologie IODA, nous prenons le parti de construire les matrices d'interaction brutes en utilisant la forme synthétique. La forme étendue y est utilisée ponctuellement afin de vérifier que les capacités d'interaction sont héritées correctement, et décrivent bien le modèle voulu. Reposer sur

Source \ Cible	∅	Aigle :: Oiseau de Proie	Faucon :: Oiseau de Proie	Rongeur :: Animal	Végétation :: Vivant
Vivant	**(MOURIR)**				
Végétation :: Vivant	*(*MOURIR*)* **(SE PROPAGER)**				
Animal :: Vivant	*(*MOURIR*)* **(SE DÉPLACER)**				
Rongeur :: Animal	*(*MOURIR*)* *(*SE DÉPLACER*)*			**(SE REPRODUIRE, d = 0)**	**(MANGER, d = 0)**
Oiseau de Proie :: Animal	*(*MOURIR*)* *(*SE DÉPLACER*)*			**(MANGER, d = 0)**	
Aigle :: Rapace	*(*MOURIR*)* *(*SE DÉPLACER*)*	**(SE REPRODUIRE, d = 0)**	**(MANGER, d = 0)**	*(*MANGER, *d = 0)*	
Faucon :: Rapace	*(*MOURIR*)* *(*SE DÉPLACER*)*		**(SE REPRODUIRE, d = 0)**	*(*MANGER, *d = 0)*	

FIGURE 8.10 – Matrice d'interaction brute sous forme étendue d'une simulation d'un écosystème contenant des `Aigles`, des `Faucons`, des `Rongeurs` et de la `Végétation`, dont la matrice d'interaction brute sous forme synthétique est définie sur la figure 8.4(a). Dans cette matrice, les éléments d'assignation présents explicitement dans la forme synthétique sont affichés en gras. Ceux ajoutés afin de compléter la forme étendue apparaissent en italique.

Source \ Cible	∅	Virus	Virus Rhume
Lymphocyte	**(SE DÉPLACER)**	**(PHAGOCYTER, d = 0)**	*(*PHAGOCYTER, *d = 0)*

Source \ Cible	Virus Grippe	Virus Varicelle	Virus Rougeole
Lymphocyte	*(*PHAGOCYTER, *d = 0)*	*(*PHAGOCYTER, *d = 0)*	*(*PHAGOCYTER, *d = 0)*

FIGURE 8.11 – Matrice d'interaction brute sous forme étendue de la matrice d'interaction brute sous forme synthétique décrite sur la figure 8.2. Dans cette simulation, un `Lymphocyte` peut phagocyter des `Virus` tels que`Virus Rhume`, le `Virus Grippe`, le `Virus Varicelle` ou le `Virus Rougeole`. Dans la forme étendue, les éléments d'assignation présents explicitement dans la forme synthétique sont affichés en gras. Ceux ajoutés afin de compléter la forme étendue appariassent en italique. Pour des raisons de lisibilité, cette matrice d'interactions a été divisée en deux morceaux.

cette dualité dans la méthodologie permet alors à la fois de profiter du génie logiciel et de la facilité de compréhension du modèle ainsi construit. Pour profiter de cette dualité, il faut pouvoir construire la forme étendue d'une matrice d'interaction brute à partir de sa forme synthétique. Dans la section qui suit, nous décrivons le procédé général permettant d'y parvenir.

Principes de la construction d'une forme étendue

La forme étendue d'une matrice d'interaction brute est composée de deux types d'éléments d'assignations. Les premiers, qui apparaissent en gras sur la forme étendue donnée en exemple dans la figure 8.11, sont les éléments que l'on retrouve dans la forme synthétique. Nous qualifions ces éléments d'assignation d'*absolus*. Les seconds, qui apparaissent en italique sur la forme étendue donnée en exemple, sont les éléments d'assignation qui ont été déduits des éléments d'assignation *absolus*. Nous les appelons *éléments d'assignation relatifs*. Dans cette section, nous illustrons comment construire la forme étendue de la matrice d'interaction brute décrite sous forme synthétique dans la figure 8.12.

Source \ Cible	∅	Chèvre :: Herbivore	Herbivore	Mouton :: Herbivore
Vivant				
Carnivore :: Vivant			(MANGER, d=1)	
Loup :: Carnivore				
Louveteau :: Loup				

FIGURE 8.12 – Forme synthétique de la matrice d'interaction brute servant d'exemple pour illustrer la construction de sa forme étendue. Pour simplifier la compréhension de l'exemple, nous ne faisons apparaître qu'un seul élément d'assignation.

La construction de la forme étendue d'une matrice d'interaction brute est un procédé itératif pouvant être automatisé, lors duquel la présence de chaque élément d'assignation dans la forme synthétique de la matrice d'interaction brute est interprétée, afin de déterminer de manière exhaustive toutes les interactions entre une famille d'agents source et une famille d'agents cible qu'elle désigne. Ce procédé consiste donc à traiter séquentiellement chaque élément d'assignation a présent dans la forme synthétique de la matrice d'interaction brute. Le traitement de a est alors formé des trois étapes qui suivent :

1. La première étape consiste à ajouter a dans la forme étendue de la matrice d'interaction brute, dans la même ligne L et la même colonne C que dans la forme synthétique. Cette étape est illustrée dans la figure 8.13 sous la forme de flèches à double traits marquées par le chiffre 1.
2. Si la colonne C est associée à une famille d'agents ayant une ou plusieurs filles, alors a est aussi ajouté à l'intersection de la ligne L et de la colonne de toutes ces familles d'agents filles. Cette étape est illustrée dans la figure 8.13 sous la forme de flèches à double traits marquées par le chiffre 2.
3. Si la ligne L est associée à une famille d'agents ayant une ou plusieurs filles, alors chaque élément d'assignation ajouté lors des étapes 1 et 2 dans la ligne L sont aussi ajoutés dans la ligne associée à chacune de ces familles d'agents filles. Cette étape est effectuée récursivement pour toutes les familles d'agents filles rencontrées. Elle est illustrée dans la figure 8.13 sous la forme de flèches à trait simple marquées par le chiffre 3.

Par exemple, dans le cas où la forme synthétique de la matrice d'interaction correspond à ce qui est présenté sur la figure 8.12, le procédé permettant de construire sa forme étendue est celui décrit sur la figure 8.13.

8.2.3 Opérateurs de spécialisation

La relation de spécialisation que nous caractérisons dans cette section ne se limite pas au simple héritage des capacités d'interaction ou de mise à jour. Nous y permettons de plus de rendre ces connaissances

Source \ Cible	∅	Chèvre :: Herbivore	Herbivore	Mouton :: Herbivore
Vivant				
Carnivore :: Vivant				
Loup :: Carnivore				
Louveteau :: Loup				

(a) Première étape : construction de la ligne associée à `Vivant`

Source \ Cible	∅	Chèvre :: Herbivore	Herbivore	Mouton :: Herbivore
Vivant				
Carnivore :: Vivant		*(*MANGER*, d=1)*	**(MANGER, d=1)**	*(*MANGER*, d=1)*
Loup :: Carnivore				
Louveteau :: Loup				

1 (Herbivore) ; 2 (Chèvre :: Herbivore) ; 2 (Mouton :: Herbivore)

(b) Deuxième étape : construction de la ligne associée à `Carnivore`

Source \ Cible	∅	Chèvre :: Herbivore	Herbivore	Mouton :: Herbivore
Vivant				
Carnivore :: Vivant		*(*MANGER*, d=1)*	**(MANGER, d=1)**	*(*MANGER*, d=1)*
Loup :: Carnivore		*(*MANGER*, d=1)*	*(*MANGER*, d=1)*	*(*MANGER*, d=1)*
Louveteau :: Loup				

3 (Chèvre :: Herbivore) ; 3 (Herbivore) ; 3 (Mouton :: Herbivore)

(c) Troisième étape : construction de la ligne associée à `Loup`

Source \ Cible	∅	Chèvre :: Herbivore	Herbivore	Mouton :: Herbivore
Vivant				
Carnivore :: Vivant		*(*MANGER*, d=1)*	**(MANGER, d=1)**	*(*MANGER*, d=1)*
Loup :: Carnivore		*(*MANGER*, d=1)*	*(*MANGER*, d=1)*	*(*MANGER*, d=1)*
Louveteau :: Loup		*(*MANGER*, d=1)*	*(*MANGER*, d=1)*	*(*MANGER*, d=1)*

3 (Chèvre :: Herbivore) ; 3 (Herbivore) ; 3 (Mouton :: Herbivore)

(d) Quatrième étape : construction de la ligne associée à `Louveteau`

FIGURE 8.13 – Illustration du procédé permettant de construire la forme étendue d'une matrice d'interaction brute, à partir de la forme synthétique présentée sur la figure 8.12. Dans cette figure, chaque flèche décrit l'effet d'une des étapes du processus de construction de la forme étendue. Le nombre associé à chaque flèche correspond à l'étape de ce procédé ayant produit un tel résultat.

plus spécifiques à la famille d'agents fille, en autorisant d'autres opérations :
- la modification de la garde de distance d'un élément d'assignation ;
- le retrait d'éléments d'assignation permettant de retirer à un agent la capacité d'initier une interaction avec une cible particulière.

Ces deux opérateurs peuvent être utilisés pour altérer ou supprimer tout élément d'assignation présent dans la forme étendue de la matrice d'interaction.

Caractérisation des opérateurs de spécialisation

Un *opérateur de spécialisation* est un élément placé dans la forme synthétique de la matrice d'interaction brute. Il caractérise l'ajout d'un élément d'assignation (on parle alors d'opérateur d'ajout), la modification de la garde de distance d'un élément d'assignation (on parle alors d'opérateur de modification), ou la suppression pure et simple d'un élément d'assignation (on parle alors d'opérateur de retrait).

L'*opérateur d'ajout* permet d'ajouter un nouvel élément d'assignation dans la forme synthétique de la matrice d'interaction brute. Cet opérateur est utilisé implicitement dans toutes les matrices d'interaction brute sous forme synthétique que nous avons définies jusqu'à présent.

Définition 52. ***Opérateur d'ajout***
Un **opérateur d'ajout** est un opérateur figurant dans une cellule de la matrice d'interaction brute sous forme synthétique, qui caractérise l'ajout d'un élément d'assignation. Il permet donc de décrire le fait qu'un agent gagne la capacité d'initier une interaction.

Lorsqu'il est placé dans la colonne $\emptyset$ de la matrice, l'opérateur d'ajout est noté $'+(I)'$ ou $'(I)'$, où I est l'identifiant d'une interaction. Dans le cas contraire, il est noté $'+(I, d = \delta)'$ ou $'(I, d = \delta)'$, avec δ un nombre réel positif.

L'*opérateur de retrait* permet de retirer à un agent la capacité d'initier une interaction avec une cible particulière. Pour cela, cet opérateur permet de retirer tout élément d'assignation apparaissant dans la forme étendue de la matrice d'interaction brute.

Définition 53. ***Opérateur de retrait***
Un **opérateur de retrait** est un opérateur figurant dans une cellule de la matrice d'interaction brute sous forme synthétique, qui caractérise le retrait d'un élément d'assignation. Il permet donc de décrire le fait qu'un agent ne possède plus la capacité d'interagir avec une cible particulière.

Lorsqu'il est placé dans la colonne $\emptyset$ de la matrice, l'opérateur de retrait est noté $'-(I)'$, où I est l'identifiant d'une interaction. Dans le cas contraire, il est noté $'-(I, d = \delta)'$, avec δ un nombre réel positif. Bien entendu, on ne peut retirer à un agent la capacité d'initier une interaction avec une cible spécifique que si ce dernier possédait cette capacité. Par conséquent, un tel opérateur ne peut être ajouté dans une cellule de la matrice d'interaction brute sous forme synthétique que si un élément d'assignation de la forme $(I, d = \delta)$ (ou (I)) est présent dans la même cellule dans la matrice d'interaction brute sous forme étendue. Le retrait peut être utilisé de deux façons : retirer purement et simplement à un agent sa capacité à interagir avec une cible, ou restreindre l'ensemble des cibles pouvant subir une interaction particulière. Par exemple, la figure 8.14 montre comment utiliser l'opérateur de retrait pour exprimer qu'un loup peut manger tout type d'herbivore hormis les chèvres.

Source \ Cible	$\emptyset$	Herbivore	Mouton :: Herbivore	Chèvre :: Herbivore	Cerf :: Herbivore
Loup		+(MANGER, d = 1)		-(MANGER, d = 1)	

FIGURE 8.14 – Matrice d'interaction exprimant qu'un `Loup` est capable de MANGER tout `Herbivore` mis à part les `Chèvres`.

Le dernier opérateur est l'*opérateur de modification*. Il permet de modifier la garde de distance de n'importe quel élément d'assignation apparaissant dans la forme étendue de la matrice d'interaction.

Définition 54. ***Opérateur de modification***
Un **opérateur de modification** est un opérateur figurant dans une cellule de la matrice d'interaction brute sous forme synthétique, qui caractérise la modification de la garde de distance d'un élément d'assignation.

Cet opérateur ne peut pas apparaître dans la colonne $\emptyset$, puisqu'aucune garde de distance ne peut y être définie. Nous notons cet opérateur $'*(I, d = \delta \rightarrow \delta')'$, où δ' est la nouvelle garde de distance. Bien entendu, on ne peut modifier la capacité d'un agent à initier une interaction avec une cible spécifique que si ce dernier possédait cette capacité. Par conséquent, un tel opérateur ne peut être ajouté dans une cellule de la matrice d'interaction brute sous forme synthétique que si un élément d'assignation de la forme $(I, d = \delta)$ est présent dans la même cellule dans la matrice d'interaction brute sous forme étendue.

L'utilisation conjointe de ces trois opérateurs permet de spécifier des simulations non seulement par héritage, mais aussi par spécialisation. Il devient alors possible d'utiliser le schéma de conception décrit dans la figure 8.6(c) page 226. Nous illustrons ici comment pratiquer cette autre approche de la conception pour construire la matrice d'interaction brute d'une simulation reproduisant un comportement basique de clients dans un magasin.

Exemple.
Considérons une simulation reproduisant le comportement de `Clients` *et de* `Clients Aveugles` *dans un magasin. Un* `Client` *y est une entité pouvant* SE DÉPLACER, LIRE *les informations affichées sur un* `Panneau Publicitaire`, LIRE *l'étiquette d'un* `Article`, RAMASSER *un* `Article` *et* PAYER *à une* `Caisse` *les articles achetés. Dans cette simulation, les* `Clients Aveugles` *se comportent comme des* `Clients`*, hormis le fait qu'il sont incapables de* LIRE *les informations d'un* `Panneau Publicitaire`*. De plus, il ne peuvent* LIRE *les informations concernant un* `Article` *qu'en braille et donc à une distance de 0. La spécification de la matrice d'interaction brute de cette simulation se fait en deux phases.*

La première phase consiste à :

1. *établir la ligne de la matrice d'interaction brute associée la famille d'agents* `Client` *à l'aide d'opérateurs d'ajout ;*
2. *déclarer que la famille d'agents* `Client Aveugle` *spécialise la famille d'agents* `Client`*.*

La forme étendue de cette matrice (voir figure 8.15(b)) permet alors de déterminer quelles interactions peuvent être initiées par la famille d'agents `Client Aveugle`*. Cette première étape est aussi l'occasion d'identifier les éléments d'assignation devant être modifiés ou supprimés pour que les* `Clients Aveugles` *correspondent bien aux descriptions de l'énoncé.*

Source \ Cible	$\emptyset$	Panneau Publicitaire	Article	Caisse
Client	+(SE DÉPLACER)	+(LIRE, d = 5)	+(LIRE, d = 2) +(RAMASSER, d = 0)	+(PAYER, d = 0)
Client Aveugle :: Client				

(a) Forme synthétique de la matrice d'interaction brute

Source \ Cible	$\emptyset$	Panneau Publicitaire	Article	Caisse
Client	**(SE DÉPLACER)**	**(LIRE, d = 5)**	**(LIRE, d = 2)** **(RAMASSER, d = 0)**	**(PAYER, d = 0)**
Client Aveugle :: Client	*(SE DÉPLACER)*	*(LIRE, d = 5)*	*(LIRE, d = 2)* *(RAMASSER, d = 0)*	*(PAYER, d = 0)*

(b) Forme étendue de la matrice d'interaction brute

FIGURE 8.15 – Matrice d'interaction brute sous forme synthétique et sous forme étendue décrivant une simulation reproduisant le comportement de `Clients` et de `Clients Aveugles` dans un magasin. Dans cette matrice, le comportement spécifique des `Clients Aveugles` n'est pas encore décrit.

Dans une seconde phase, des opérateurs de modification et de retrait sont utilisés afin de caractériser la différence entre les familles `Client Aveugle` *et* `Client`*. Dans cet exemple, cela consiste à lui retirer la*

capacité de LIRE *des* `Panneaux Publicitaires` *en utilisant l'opérateur de retrait sur l'élément d'assignation* (LIRE, $d = 5$). *Cela consiste d'autre part à modifier la distance à partir de laquelle il est capable de* LIRE *l'étiquette des* `Articles`, *en modifiant la garde de distance de l'élément d'assignation* (LIRE, $d = 2$). *On aboutit alors à la matrice d'interaction sous forme synthétique de la figure 8.16(a). Sa forme étendue (voir figure 8.16(b)) représente bien les* `Clients Aveugles` *tels que décrits au début de cet exemple.*

Source \ Cible	∅	Panneau Publicitaire	Article	Caisse
Client	+(SE DÉPLACER)	+(LIRE, d = 5)	+(LIRE, d = 2) +(RAMASSER, d = 0)	+(PAYER, d = 0)
Client Aveugle :: Client		-(LIRE, d = 5)	*(LIRE, d = 2 → 0)	

(a) Forme synthétique de la matrice d'interaction brute

Source \ Cible	∅	Panneau Publicitaire	Article	Caisse
Client	**(SE DÉPLACER)**	**(LIRE, d = 5)**	**(LIRE, d = 2)** **(RAMASSER, d = 0)**	**(PAYER, d = 0)**
Client Aveugle :: Client	*(SE DÉPLACER)*		**(LIRE, d = 0)** *(RAMASSER, d = 0)*	*(PAYER, d = 0)*

(b) Forme étendue de la matrice d'interaction brute

FIGURE 8.16 – Matrice d'interaction brute sous forme synthétique et sous forme étendue décrivant une simulation reproduisant le comportement de `Clients` et de `Clients Aveugles` dans un magasin. Les opérateurs figurant dans la ligne associée à la famille d'agents `Client Aveugle` ont été déterminés en identifiant les modifications à apporter à la ligne associée à la famille d'agents `Client Aveugle` dans la figure 8.15(b) afin de parvenir à la ligne représentant les capacités d'interaction attendues affichées dans la figure 8.16(b).

Nous avons décrit ici l'usage des opérateurs de spécialisation dans un cas simple ne faisant intervenir que deux familles d'agents et une seule relation de spécialisation. Lorsque plusieurs relations de spécialisation apparaissent, la construction de la forme étendue de la matrice d'interaction brute doit prendre en compte non seulement les modifications ayant lieu dans la cellule où a été placé l'opérateur de retrait ou de modification, mais aussi la propagation de leur effet aux autres cellules de la matrice. Ce point nécessite une attention particulière, car selon la façon dont sont propagées les effets des différents opérateurs, des formes étendues très différentes peuvent être obtenues.

Propagation de l'effet des opérateurs de spécialisation

L'effet d'un opérateur de spécialisation n'est pas restreint à la cellule de la matrice d'interaction brute où il est placé : il se propage aux familles d'agents filles de sa source et de sa cible. Dans le cas de l'opérateur d'ajout, cette propagation consiste à ajouter des éléments d'assignation dans la forme étendue de la matrice d'interaction.

Tout comme l'opérateur d'ajout, l'opérateur de suppression et l'opérateur de modification ont un effet se propageant parmi les familles d'agents filles. Le traitement de ces opérateurs pose toutefois problème, car l'ordre dans lequel ils sont évalués influe grandement sur la forme étendue obtenue de la matrice d'interaction brute et donc sur la sémantique du modèle créé. Pour préserver la cohérence du modèle, le procédé permettant de propager l'effet des opérateurs doit en particulier s'assurer du traitement de deux problèmes, décrits par les matrices d'interaction brutes sous forme synthétique décrites dans les figures 8.17(a) et 8.17(b).

Caractérisation de problèmes rencontrés. Lorsque plusieurs familles d'agents entretiennent une relation de spécialisation, certains opérateurs peuvent avoir un effet entrant en concurrence. Par exemple, la figure 8.17(a) décrit qu'un `Carnivore` est capable de MANGER des `Herbivores` à une distance de 0, sauf dans le cas des `Herbivores Malade` dont la faible mobilité permet aux `Carnivores` de les chasser puis les MANGER à plus grande distance (une distance de 5). Dans cette matrice, un LOUP est un `Carnivore`

Source \ Cible	∅	Herbivore	Herbivore Malade :: Herbivore
Carnivore		+(MANGER, d = 0)	*(MANGER, d=0 → 5)
Loup :: Carnivore		*(MANGER, d=0 → 1)	?

(a) Premier problème rencontré : la collision de modifications. L'opérateur "*" est utilisé afin de modifier la garde de distance d'un élément d'assignation. Cette modification est valable pour tous les agents spécialisant la famille d'agent source et la famille d'agent cible de l'élément d'assignation modifié. Le problème est ici de déterminer si les `Loups` mangent les `Herbivores malades` à une distance de 1 ou à une distance de 5.

Source \ Cible	∅	Loup
Loup	+(SE DÉPLACER) +(MOURIR)	+(SE REPRODUIRE, d=0)
Loup Agressif :: Loup		+(ATTAQUER, d=1) *(SE REPRODUIRE, d=4)
Loup Stérile :: Loup		-(SE REPRODUIRE, d=0)
Loup Agressif et Stérile :: Loup Agressif Loup Stérile		?

(b) Deuxième problème rencontré : le retrait d'une interaction dans une famille d'agents mère qui est présente dans une autre famille d'agents mère. Dans cette figure, le problème consiste à déterminer si l'interaction SE REPRODUIRE est présente ou non dans la ligne associée à la famille d'agents `Loup Agressif et Stérile`

FIGURE 8.17 – Illustration de deux problèmes de cohérence devant être pris en compte lors de la propagation de l'effet des opérateurs de spécialisation.

particulier, suffisamment agile et rapide pour chasser et MANGER des `Herbivores` à une plus grande distance que la plupart des `Carnivores` (une distance de 1). Dans ce cas, deux valeurs peuvent être attribuées à la distance à laquelle un LOUP peut MANGER des `Herbivores Malades`. Se pose alors le problème du choix de la valeur à prendre en compte.

La solution à ce problème se trouve dans l'interprétation de la relation de spécialisation. La ligne de la matrice d'interaction associée à la famille d'agents CARNIVORE s'interprète en « Un carnivore est un agent qui est capable de manger des herbivores à une distance de 0, sauf les herbivores malades, qu'il est capable de manger à une distance d'au plus 5 ». L'opérateur de modification $*(Manger, d = 0 \rightarrow 1)$ situé à l'intersection de la ligne associée à LOUP et de la colonne associée à HERBIVORE se lit « Un loup est un carnivore particulier qui mange tout herbivore à une distance d'au plus 1 ». **Cette seconde interprétation est plus spécifique que celle fournie pour les CARNIVORES.** La solution nous étant apparue comme la plus appropriée est alors :

- d'ignorer la modification induite par l'opérateur $*(Manger, d = 0 \rightarrow 5)$ situé dans la ligne associée à la famille d'agents HERBIVORE ;
- de prendre en compte la modification la plus spécifique à la famille d'agents LOUP.

Dans ce cas, l'élément d'assignation présent implicitement dans la cellule marquée par point d'interrogation dans la figure 8.17(a) est donc $(\text{MANGER}, d = 1)$.

Le second problème est aussi relatif à la collision de la propagation de deux opérateurs de spécialisation. Il est lié au fait qu'un opérateur de retrait et un opérateur de modification peuvent avoir un effet sur un même élément d'assignation dans deux familles d'agents différentes. Dans ce cas, si ces deux familles d'agents sont spécialisées par une même famille d'agents fille, il faut pouvoir déterminer si l'élément d'assignation est présent ou pas dans la ligne associée à la famille d'agents fille. La figure 8.17(b) fournit un exemple de ce problème à l'aide d'une matrice d'interaction brute sous forme synthétique. Cette matrice décrit trois déclinaisons différentes du comportement d'un `Loup` : les `Loups Agressifs`, les `Loups Stériles` et les `Loup Agressifs et Stériles`. Les `Loups Agressifs` sont des `Loups` particuliers

pouvant ATTAQUER d'autres `Loups` et chercher plus loin leurs partenaires de reproduction. Les `Loups Stériles` sont des `Loups` particuliers n'étant pas capables de se reproduire. Enfin, les `Loups Agressifs et Stériles` cumulent les spécificités des `Loups Agressifs` et des `Loups Stériles`. Ce diagramme d'héritage amène à un paradoxe : un `Loup Agressif et Stérile` est un `Loup Agressif` particulier, et, à ce titre, il doit être capable de SE REPRODUIRE à une distance d'au plus 4. Il est toutefois aussi un `Loup Stérile` particulier et doit donc ne pas pouvoir SE REPRODUIRE.

La solution la plus appropriée à ce paradoxe pour cet exemple consiste à ignorer les modifications de garde de distance induites par la famille d'agents `Loup Agressif` et à ne prendre en compte que le retrait effectué par la famille d'agents `Loup Stérile`. **L'opérateur de suppression serait donc plus prioritaire que l'opérateur de modification.**

Les solutions envisagées ici ne dépendent pas de la sémantique de l'interaction, mais de la position des interaction dans la forme étendue de la matrice d'interaction brute, ainsi que de la relation d'ordre établie entre les opérateurs. La construction de la forme étendue d'une matrice d'interaction brute sous forme synthétique peut donc être automatisée.

Principes de la propagation de l'effet des opérateurs de spécialisation. Nous proposons dans cette section un procédé automatique permettant de passer d'une forme synthétique à une forme étendue d'une matrice d'interaction brute. Pour cela, nous faisons le choix de suivre les deux principes évoqués dans le paragraphe précédent :

1. **toujours prendre en compte l'effet de l'opérateur de modification le plus spécifique à la famille d'agents F considérée.** Il consiste à préférer les modifications apportées directement par un opérateur de modification figurant dans la ligne associée à F (par exemple l'opérateur $*(\text{MANGER}, d = 0 \rightarrow 1)$ dans la figure 8.17(b) si F est la famille d'agents `Loup`) plutôt que les modifications apportées par un opérateur de modification présent dans la ligne d'une famille d'agents mère de F (par exemple l'opérateur $*(\text{MANGER}, d = 0 \rightarrow 5)$ présent dans la ligne associée à la famille d'agents `Carnivore`). Il consiste aussi à préférer les modifications apportées par un opérateur de modification prenant directement pour cible une famille d'agents C plutôt que les modifications induites par un opérateur de modification prenant pour cible une famille d'agent mère de C.
2. **toujours préférer les modifications induites par un opérateur de retrait plutôt que les modifications induites par un opérateur de modification.** L'effet des opérateurs de retrait doit donc être appliqué en dernier.

Automatisation de la construction d'une forme étendue. L'automatisation du procédé permettant de construire la forme étendue d'une matrice d'interaction à partir de sa forme synthétique repose en premier lieu sur le principe de la propagation verticale et la propagation horizontale de l'effet d'un opérateur de spécialisation. Dans ce paragraphe, nous posons :

- a un élément d'assignation ;
- *op* un opérateur de spécialisation permettant d'ajouter, de modifier, ou de supprimer un élément d'assignation a ;
- S la famille d'agents associée à la ligne de la matrice d'interaction brute sous forme synthétique où figure *op* ;
- C la famille d'agents associée à la colonne où se situe a.

La *propagation horizontale* d'un opérateur *op* consiste à appliquer son effet uniquement dans la ligne associée à S dans la forme étendue de la matrice d'interaction brute. Ce type de propagation apparaît sous la forme de flèches à double trait dans la figure 8.19. Si *op* est un opérateur d'ajout, l'effet consiste à ajouter l'élément d'assignation a à l'intersection de la ligne associée à S et de la colonne associée à C . De plus, si a n'est pas un élément d'assignation dégénéré, la propagation horizontale de son effet consiste à dupliquer l'élément d'assignation a dans chaque colonne associée à une famille d'agents spécialisant C. Si *op* est un opérateur de retrait, son effet consiste à retirer l'élément d'assignation a de la cellule située à l'intersection de la ligne associée à S et de la colonne associée à C . De plus, si a n'est pas un élément d'assignation dégénéré, la propagation horizontale de son effet consiste à aussi retirer l'élément d'assignation a dans chaque colonne associée à toute famille d'agents spécialisant C. Enfin, si *op* est un opérateur de modification, son effet consiste à modifier la garde de distance de l'élément d'assignation

a de la cellule située à l'intersection de la ligne associée à S et de la colonne associée à C, ainsi que de toutes les occurrences de a dans une colonne associée à une famille d'agents spécialisant C.

Dans les trois cas, nous qualifions de dépendances de *op* l'ensemble des éléments d'assignation ayant été ajoutés, modifiés ou supprimés lors de la propagation horizontale de l'opérateur *op*.

La *propagation verticale* d'un opérateur *op* consiste à appliquer l'effet de *op* dans la ligne associée à chaque famille d'agents spécialisant S, ainsi que dans la ligne de toutes les familles d'agents spécialisant transitivement S. La propagation verticale de l'effet de *op* dans une de ces lignes consiste à y reproduire l'effet qu'a eu *op* dans la ligne associée à S. Dans le cas d'un opérateur d'ajout, cela consiste à ajouter une copie dans la ligne de chaque élément d'assignation apparaissant dans les dépendances de *op*. Dans le cas d'un opérateur de retrait, cela consiste à retirer de la ligne tous les éléments d'assignation présents dans les dépendances de *op*. Enfin, dans le cas d'un opérateur de modification, cela consiste à modifier la garde de distance de tous les éléments d'assignation de la ligne présents dans les dépendances de *op*. Ce type de propagation apparaît sous le forme de flèches à simple trait dans la figure 8.19.

La construction de la forme étendue de la matrice d'interaction se fait alors en trois phases :

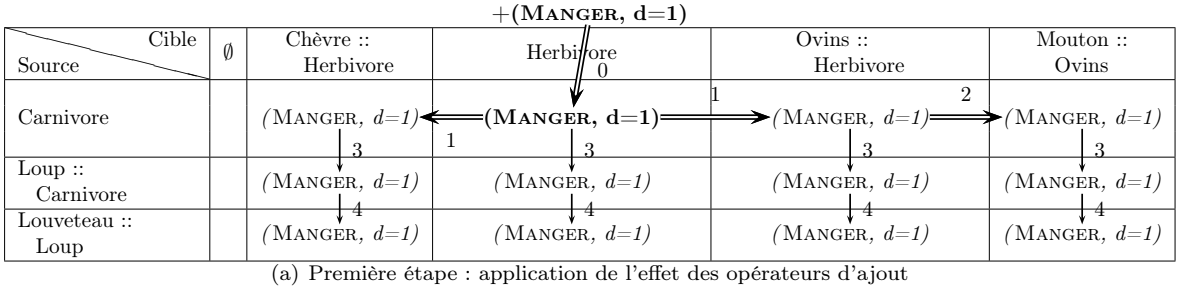
1. **propager l'effet de tous les opérateurs d'ajout contenus dans la forme synthétique de la matrice d'interaction brute.** Cette propagation est illustrée plus en détails dans la section 8.2.2.
2. **propager l'effet des opérateurs de modification dans un ordre précis, afin que la modification la plus spécifique soit prise en compte.** Pour cela, l'effet d'un opérateur de modification présent dans une ligne associée à une famille d'agents F n'est propagé que si l'effet des opérateurs de modification présents dans la ligne associée aux les familles d'agents mères de F a déjà été propagé. La propagation de l'effet de la modification repose sur le même principe que le retrait : la propagation horizontale dans la ligne où l'opérateur apparaît, puis la modification par propagation verticale de tous les éléments d'assignation ayant été modifiés dans cette ligne.
3. **propager l'effet de tous les opérateurs de retrait**, en effectuant dans un premier temps une propagation horizontale dans la ligne où l'opérateur apparaît, puis par le retrait par propagation verticale de tous les éléments d'assignation ayant été retirés de cette ligne.

Nous illustrons ce procédé de construction dans la figure 8.19, pour une matrice d'interaction brute sous forme synthétique telle que décrite dans la figure 8.18.

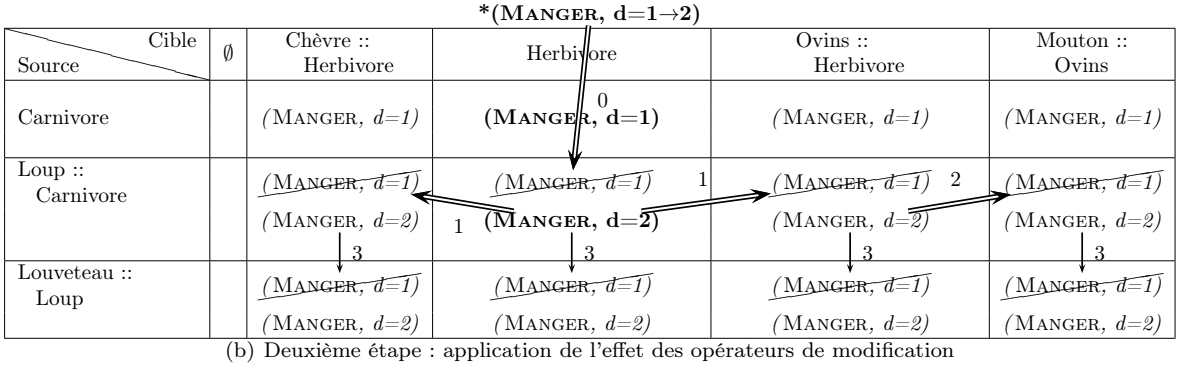
Source \ Cible	∅	Chèvre :: Herbivore	Herbivore	Ovins :: Herbivore	Mouton :: Ovins
Carnivore			+(MANGER, d=1)		
Loup :: Carnivore			*(MANGER, *d=1→2*)	-(MANGER, d=1)	
Louveteau :: Loup					

FIGURE 8.18 – Matrice d'interaction brute sous forme synthétique utilisée pour illustrer la propagation de l'effet des opérateurs. On exprime ici que les `Carnivores` peuvent manger des `Herbivores`. Il est de plus exprimé que les `Loups` peuvent manger des `Herbivores` à une distance plus grande que les autres `Carnivores` et qu'ils ne peuvent pas manger d'`Ovins`.

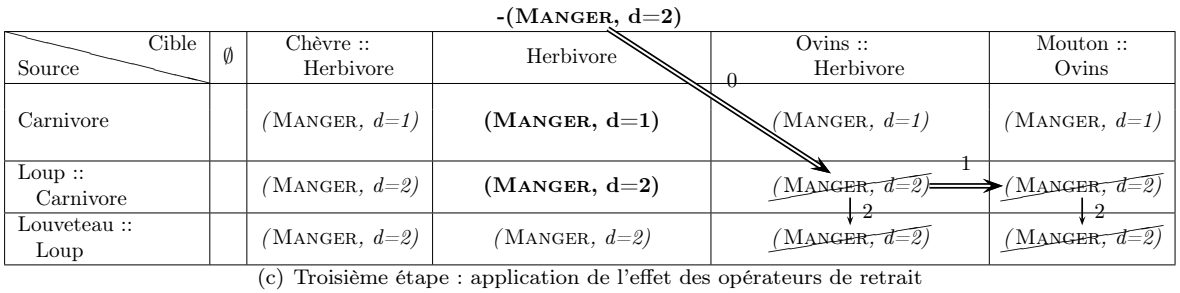
Limites de l'automatisation de la construction d'une forme étendue. Les principes que nous décrivons ici permettent d'automatiser la transformation d'une forme synthétique vers une forme étendue. Cette transformation ne peut toutefois pas être automatique dans tous les cas. En effet, puisque la spécialisation permet de pratiquer l'héritage "classique" des systèmes multi-agents, elle souffre aussi des mêmes limites. L'opérateur de modification permet de "surcharger" la garde de distance d'un élément d'assignation. Il est donc sujet au même problème que le polymorphisme des méthodes dans le contexte de l'héritage multiple : si une famille d'agents spécialise deux familles d'agents différentes, appliquant toutes deux un opérateur de modification au même élément d'assignation, une ambiguïté survient dans le modèle. La figure 8.20 illustre ce type d'ambiguïté.



(a) Première étape : application de l'effet des opérateurs d'ajout



(b) Deuxième étape : application de l'effet des opérateurs de modification



(c) Troisième étape : application de l'effet des opérateurs de retrait

Cible / Source	∅	Chèvre :: Herbivore	Herbivore	Ovins :: Herbivore	Mouton :: Ovins
Carnivore		*(*MANGER, *d=1)*	**(MANGER, d=1)**	*(*MANGER, *d=1)*	*(*MANGER, *d=1)*
Loup :: Carnivore		*(*MANGER, *d=2)*	**(MANGER, d=2)**		
Louveteau :: Loup		*(*MANGER, *d=2)*	*(*MANGER, *d=2)*		

(d) Apparence finale de la forme étendue de la matrice d'interaction brute

FIGURE 8.19 – Illustration des trois étapes du procédé permettant de propager l'effet d'un opérateur de modification dans une matrice d'interaction brute, ainsi que du résultat de la propagation. La flèche à double trait (resp. trait simple) représente la propagation horizontale (resp. verticale) de l'effet de l'opérateur.

Cible / Source	∅	E
A		$+(\mathrm{I},\ d=\delta)$
B :: A		$*(\mathrm{I},\ d=\delta \rightarrow \delta'_1)$
C :: A		$*(\mathrm{I},\ d=\delta \rightarrow \delta'_2)$
D :: B, C		?

FIGURE 8.20 – Illustration d'un problème lié à la spécialisation de plusieurs familles d'agents issu du problème du losange en héritage multiple. Dans cette figure, la détermination de la garde de distance dans la ligne associée à la famille d'agents `D` ne peut être automatisée. Elle requiert l'intervention du concepteur.

Contrairement aux deux autres problèmes mentionnés dans le paragraphe intitulé « Caractérisation de problèmes rencontrés » dans la page 236, ce dernier problème ne peut être résolu automatiquement. En effet, la solution des deux problèmes évoqués précédemment se fondent soit sur l'interprétation d'opérateurs de même nature se trouvant dans des situations différentes (voir figure 8.17(a)), soit sur des opérateurs de natures différentes se trouvant dans une situation similaire (voir figure 8.17(b)). Dans ce dernier cas, il n'y a aucune différence d'opérateur, ou de situation.

Une solution de ce problème lié à l'héritage multiple serait de **détecter automatiquement les collisions** dans l'implémentation de la méthodologie (par exemple JEDI-BUILDER) et **demander au concepteur de choisir** parmi différentes alternatives la garde de distance utilisée. Ce cas particulier de la spécialisation de plusieurs familles d'agents, où des modifications contradictoires sont fournies à un même élément d'assignation, n'est pas étudié dans le cadre de cette thèse.

Exploitation de la spécialisation dans le simulateur

Jusqu'à présent, nous nous sommes focalisés sur la spécification de la relation de spécialisation entre les familles d'agents d'une simulation et sur les moyens permettant de réutiliser tout ou partie du modèle d'une famille d'agents. Puisque l'introduction de la relation de spécialisation a amené à changer la structure de la matrice d'interaction brute, les algorithmes de simulation décrits hors du cadre de la spécialisation peuvent être devenus obsolètes. Dans cette section, nous étudions cette question et nous déterminons différents moyens permettant d'implémenter la relation de spécialisation.

La différence principale entre les algorithmes présentés dans le chapitre 4 et les algorithmes prenant en compte la relation de spécialisation réside dans la façon de mesurer le potentiel d'interaction d'un agent. En effet, là on n'avions que la matrice d'interaction brute, nous avons maintenant deux formes pour cette matrice : une forme synthétique et une forme étendue. Deux problèmes doivent alors être résolus :

1. déterminer comment mesurer le potentiel d'interaction d'un agent (*i.e.* l'ensemble des interactions qu'il est susceptible d'initier avec les agents de son voisinage) à l'aide des formes étendue et synthétique d'une matrice d'interaction brute ;
2. déterminer :
   - à quelle forme correspond la ligne de la matrice d'interaction brute du modèle d'un agent ;
   - comment cette ligne est en pratique implémentée dans un simulateur.

Mesure du potentiel d'interaction La forme synthétique de la matrice d'interaction permet de factoriser le code des agents. Toutefois, pour qu'il puisse exécuter son processus comportemental, un agent doit avoir la connaissance de toutes les interactions qu'il est susceptible d'initier, afin de pouvoir construire son potentiel d'interaction. La forme étendue de la matrice d'interaction brute fait état de toutes ces interactions. Elle peut donc être utilisée pour construire le comportement des agents, selon les principes énoncés dans l'algorithme 20. Cet algorithme fonctionne de la même manière que l'algorithme décrit dans le chapitre 4 en se basant toutefois sur la forme étendue de la matrice d'interaction brute.

Implémentation d'une ligne de la matrice d'interaction brute. L'algorithme présenté précédemment repose sur la forme étendue de la matrice d'interaction, alors que la spécification d'une simulation repose sur sa forme synthétique. Trois approches peuvent implémenter ces matrices et en attribuer les lignes aux familles d'agents.

La première approche consiste à attribuer à chaque famille d'agents leur ligne de la forme synthétique de la matrice d'interaction. À chaque fois que l'ordonnanceur de la simulation donne la parole à l'agent, la forme étendue de la matrice d'interaction est déduite de la ligne associée à la famille d'agents, ainsi que de la ligne associée à ses diverses familles d'agents mères. Cette approche est peu efficace en termes de temps de calculs, mais permet de profiter de la factorisation de la description des lignes de la matrice.

La seconde approche consiste à attribuer à chaque famille d'agents leur ligne de la forme étendue de la matrice d'interaction. Dans ce cas, la simulation fonctionne exactement comme énoncé dans le chapitre 4. Cette approche est très efficace en termes de temps de calculs, mais la factorisation de la description des différentes lignes de la matrice d'interaction brute disparaît. La matrice d'interaction brute sous forme synthétique n'est alors qu'un moyen utilisé dans la méthodologie afin de faciliter la

Algorithme 20 : Algorithme de calcul du potentiel d'interaction d'un agent nommé x, dans le cas où la relation de spécialisation peut être utilisée. Dans cet algorithme, nous notons $\mathcal{M}^{etendue}$ la matrice d'interaction sous forme étendue et $\mathcal{M}^{etendue}(S, C)$ l'ensemble des éléments d'assignation ayant la famille d'agents S pour source et la famille d'agents C pour cible.

potentiel d'interaction(x)
début
  $\mathcal{R} \Leftarrow \emptyset$;
  $\mathcal{S} \Leftarrow$ famille(x);
  % Recensement des tuples contenant des interactions de multicast
  pour tous les $\mathcal{F} \in \mathbb{F}$ **faire**
    pour tous les $a \in \mathcal{M}^{etendue}(\mathcal{S}, \mathcal{F})$ **faire**
      si $card(\mathcal{I}(a)) = (1, *)$ **alors**
        $\mathcal{T} \Leftarrow \emptyset$;
        pour tous les $y \in \mathcal{V}(x)$ **faire**
          si $accepterCible\big(\mathcal{I}(a)\big)\big(x, y\big)$ ***et*** $environnement.distance(x, y) \leq dist(a)$ **alors**
            $\mathcal{T} \Leftarrow \mathcal{T} \cup \{y\}$;
        si $(a, x, \mathcal{T})$ *est réalisable* **alors**
          $\mathcal{R} \Leftarrow \mathcal{R} \cup \{(a, x, \mathcal{T})\}$;
  % Recensement des tuples contenant des interactions individuelles
  pour tous les $y \in \mathcal{V}(x)$ **faire**
    pour tous les $a \in \mathcal{M}^{etendue}\Big(famille(x), famille(y)\Big)$ **faire**
      si (a, x, y) *est réalisable* **alors**
        $\mathcal{R} \Leftarrow \mathcal{R} \cup \{(a, x, y)\}$;
  % Recensement des tuples contenant des interactions dégénérées
  pour tous les $a \in \mathcal{M}\Big(famille(x), \emptyset\Big)$ **faire**
    si (a, x) *est réalisable* **alors**
      $\mathcal{R} \Leftarrow \mathcal{R} \cup \{a, x\}$;
  retourner $\mathcal{R}$;
fin

conception de matrices de taille conséquente. Elle n'apparaît pas dans l'implémentation. Il s'agit de l'approche implémentée actuellement dans JEDI-BUILDER.

La dernière approche consiste à attribuer à chaque famille d'agents leur ligne de la forme synthétique de la matrice d'interaction. Toutefois, au lieu de réévaluer la ligne de la forme étendue de la matrice d'interaction brute à chaque sollicitation du processus comportemental de l'agent, cette approche construit la ligne de la forme étendue de la matrice d'interaction brute lors de l'initialisation de la simulation. Cette dernière approche fournit un compromis cumulant à la fois généricité et efficacité en termes de temps de calcul, et constitue donc l'approche à privilégier lors de l'implémentation de la méthodologie. Actuellement, cette approche n'est pas implémentée dans JEDI et JEDI-BUILDER par manque de temps.

8.3 Spécialisation et modèle de sélection d'interaction

La section précédente a décrit comment utiliser la relation de spécialisation afin de construire la matrice d'interaction brute et la matrice de mise à jour. Un agent ne se limite toutefois pas à ces deux matrices. Pour qu'il puisse être implémenté et qu'il puisse participer à une simulation, un agent doit aussi définir entre autres son modèle de sélection d'interaction et sa matrice de mise à jour ordonnée. La matrice d'interaction raffinée et la matrice de mise à jour ordonnée ont des structures similaires. Les propos que nous tenons dans cette section concernant la matrice d'interaction raffinée s'appliquent donc aussi à la matrice de mise à jour ordonnée.

Dans cette section, nous nous intéressons plus particulièrement au modèle réactif de sélection d'interaction de IODA.

8.3.1 Principes

Le modèle réactif de sélection d'interaction que nous considérons ici repose sur le concept de matrice d'interaction raffinée défini dans le chapitre 3. Cette matrice attribue une priorité à chaque élément d'assignation présent dans la matrice d'interaction brute. Si l'on se place dans le cas où la spécialisation peut être utilisée, la matrice d'interaction peut prendre soit une forme synthétique, soit une forme étendue. Afin de prendre en compte la spécialisation dans le modèle réactif de sélection d'interaction, il faut donc, dans un premier temps, identifier sur quelle forme de la matrice d'interaction brute se baser.

Dans le modèle réactif, les priorités sont utilisées pour déterminer quel tuple présent dans le potentiel d'interaction représente l'interaction initiée par l'agent. Puisque, dans le cas où l'on utilise la spécialisation, la mesure du potentiel d'interaction se base sur la forme étendue de la matrice d'interaction brute, la spécification minimale permettant de définir le comportement d'un agent serait d'associer une priorité à chaque élément d'assignation présent dans la forme étendue de la matrice d'interaction brute. Toutefois, attribuer des priorités dans une matrice faisant apparaître de manière exhaustive toutes les interactions pouvant survenir dans une simulation peut s'avérer fastidieux.

Pour remédier à ce problème, la matrice d'interaction raffinée possède elle aussi une forme synthétique et une forme étendue.

Formes synthétique et étendue La forme étendue de la matrice d'interaction raffinée attribue une priorité à chaque élément d'assignation présent dans la forme étendue de la matrice d'interaction brute. Tout comme dans le cas de la matrice d'interaction brute, l'intérêt de la forme étendue de la matrice d'interaction raffinée est double :

- Elle permet d'interpréter la forme synthétique de la matrice d'interaction raffinée. Elle pallie donc au problème de dispersion des interactions dans les familles d'agents mères.
- Elle sert de support à l'implémentation. En effet, la forme étendue correspond à la matrice d'interaction raffinée utilisée dans les algorithmes du cœur de IODA décrits dans le chapitre 4. Il est donc possible de réutiliser les algorithmes de simulation du cœur de IODA sans interprétation spécifique à la spécialisation.

La forme synthétique de la matrice d'interaction raffinée repose sur des principes identiques à la forme synthétique d'une matrice d'interaction brute. Ses cellules contiennent des opérateurs décrivant chacun l'attribution d'une priorité à un élément d'assignation pour une famille d'agents source et cible

particulières. L'effet de cette attribution se propage alors de la même manière que l'effet d'un opérateur de modification dans la matrice d'interaction brute.

Opérateur d'attribution de priorité Dans le cas d'une matrice d'interaction raffinée, seul un opérateur appelé *opérateur d'attribution de priorité* est utilisé. Cet opérateur permet d'associer une priorité à un élément d'assignation présent dans la forme étendue de la matrice d'interaction brute.

Définition 55. ***Opérateur d'attribution de priorité***
Un **opérateur d'attribution de priorité** est un opérateur de spécialisation figurant dans une cellule de la forme synthétique de la matrice d'interaction raffinée, qui caractérise l'attribution d'une priorité à un élément d'assignation.

Lorsqu'il est placé dans la colonne $\emptyset$ de la matrice, il est noté $!(I, p = \rho)$ ou $(I, p = \rho)$, où ρ est la priorité attribuée à l'élément d'assignation (I) et I est l'identifiant d'une interaction dégénérée. Dans le cas contraire, il est noté $!(I, d = \delta, p = \rho)$ ou $(I, d = \delta, p = \rho)$, où ρ est la priorité attribuée à l'élément d'assignation $(I, d = \delta)$, I est l'identifiant d'une interaction qui n'est pas dégénérée et δ est un nombre réel positif.

Pour qu'un opérateur puisse figurer dans la forme synthétique de la matrice d'interaction raffinée, l'opérateur doit attribuer une priorité à un élément d'assignation existant dans la matrice d'interaction brute. Par conséquent, un opérateur $!(I, d = \delta, p = \rho)$ se situe à l'intersection d'une ligne associée à une famille d'agents S et de la colonne associée à une famille d'agents C dans la forme synthétique de la matrice d'interaction raffinée uniquement si l'élément d'assignation $(I, d = \delta)$ est présent dans la cellule à l'intersection de la ligne associée à S et de la colonne associée à C dans la forme étendue de la matrice d'interaction brute. Il en va de même pour l'opérateur $!(I, p = \rho)$ dans la colonne $\emptyset$.

L'utilisation de cet opérateur est suffisante pour construire une matrice d'interaction raffinée supportant les schémas de conception décrits dans la figure 8.6. Nous illustrons ci-après comment pratiquer cette approche de conception pour construire une matrice d'interaction raffinée pour une simulation reproduisant le comportement basique de clients dans un magasin, dont la matrice d'interaction brute est décrite dans la figure 8.15 page 235.

Exemple.
Considérons à nouveau une simulation reproduisant le comportement de `Clients` *et* `Clients Aveugles` *dans un magasin décrite dans la figure 8.15 page 235. Pour illustrer l'usage des priorités, nous introduisons dans cet exemple une nouvelle famille d'agents* `Caisse Invalides`*, qui est une* `Caisse` *particulière refusant d'encaisser le paiement d'un* `Client` *non aveugle, si jamais un* `Client Aveugle` *se situe à proximité.*

Dans cette simulation, si un `Client` *a fini ses achats, alors il cherche ne priorité à* PAYER *ses achats en* `Caisse`*. On attribue donc à l'élément d'assignation* (PAYER, $d = 0$) *la priorité maximale dans la ligne de* `Client` *(en l'occurrence 10 dans cet exemple). Si ses achats ne sont pas terminés, alors le* `Client` SE DÉPLACE *dans le magasin tant qu'il ne rencontre pas de* `Panneaux Publicitaires` *ou d'*`Articles`*. On attribue donc une priorité minimale à l'élément d'assignation* (SE DÉPLACER) *(en l'occurrence 0). Si le* `Client` *rencontre un* `Panneau Publicitaire`*, alors il prend connaissance des informations qu'il affiche en effectuant l'interaction* LIRE*. Si le* `Client` *a déjà lu ce* `Panneau Publicitaire` *et qu'il aperçoit un* `Article`*, alors il* LIT *l'étiquette de l'*`Article` *s'il ne l'a pas encore lue. Il rassemble ainsi des informations sur l'article, par exemple si l'*`Article` *fait partie d'une promotion. Dans le cas contraire, il peut décider de* RAMASSER *l'*`Article` *selon des critères qui lui sont propres, définis dans le déclencheur de l'interaction. On aboutit donc à la relation d'ordre :* (LIRE, $d = 5$) *prenant pour cible* `Panneau Publicitaire` $>$ (LIRE, $d = 2$) *prenant pour cible* `Article` $>$ (RAMASSER, $d = 0$) *prenant pour cible* `Article` *. Nous leur attribuons donc respectivement les priorités 3, 2 et 1. La forme synthétique de la matrice d'interaction raffinée faisant état de ces priorités est résumée sur la figure 8.21(a).*

Dans cette simulation, un `Client Aveugle` *se comporte comme un* `Client`*, mis à part le fait qu'il préférera aller* PAYER *ses achats dans une* `Caisse Invalides` *plutôt que dans une* `Caisse` *classique. Une priorité* 11 *est donc attribuée à l'élément d'assignation* (PAYER, $d = 0$) *se situant dans la colonne associée à la famille d'agents* `Caisse Invalides`*. La forme synthétique de la matrice d'interaction raffinée faisant état de ces priorités est résumée sur la figure 8.22(a).*

Source \ Cible	$\emptyset$	Panneau Publicitaire	Article	Caisse	Caisse Invalides :: Caisse
Client	!(Se Déplacer, p = 0)	!(Lire, d = 5, p = 3)	!(Lire, d = 2, p = 2) !(Ramasser, d = 0, p = 1)	!(Payer, d = 0, p = 10)	
Client Aveugle :: Client					

(a) Forme synthétique de la matrice d'interaction raffinée

Source \ Cible	$\emptyset$	Panneau Publicitaire	Article	Caisse	Caisse Invalides :: Caisse
Client	**(Se Déplacer, p = 0)**	**(Lire, d = 5, p = 3)**	**(Lire, d = 2, p = 2)** **(Ramasser, d = 0, p = 1)**	**(Payer, d = 0, p = 10)**	*(Payer, d = 0, p = 10)*
Client Aveugle :: Client	*(Se Déplacer, p = 0)*		*(Lire, d = 0, p = ?)* *(Ramasser, d = 0, p = 1)*	*(Payer, d = 0, p = 10)*	*(Payer, d = 0, p = 10)*

(b) Forme étendue de la matrice d'interaction raffinée

FIGURE 8.21 – Matrice d'interaction raffinée sous forme synthétique (a) et sous forme étendue (b) d'une simulation de magasin. Cette simulation reproduit le comportement de `Clients` et de `Clients Aveugles`. Dans ces matrices, le comportement spécifique des `Clients Aveugles` n'est pas encore décrit.
L'opérateur !(Lire, $d = 2, p = 2$) défini pour la famille `Client` ne peut modifier la priorité que de l'élément d'assignation (Lire, $d = 2$). Or, dans la matrice d'interaction brute de cette simulation, un opérateur de modification est utilisé pour faire passer la garde de distance de l'élément d'assignation (Lire, $d = 2$) à 0. Aucune priorité n'est donc attribuée à l'élément d'assignation (Lire, $d = 0$) de la ligne associée à la famille d'agents `Clients Aveugles`. Sa priorité est donc non définie et est représentée par un point d'interrogation.

Source \ Cible	$\emptyset$	Panneau Publicitaire	Article	Caisse	Caisse Invalides :: Caisse
Client	!(Se Déplacer, p = 0)	!(Lire, d = 5, p = 3)	!(Lire, d = 2, p = 2) !(Ramasser, d = 0, p = 1)	!(Payer, d = 0, p = 10)	
Client Aveugle :: Client			!(Lire, d = 0, p = 2)		!(Payer, d = 0, p = 11)

(a) Forme synthétique de la matrice d'interaction raffinée

Source \ Cible	$\emptyset$	Panneau Publicitaire	Article	Caisse	Caisse Invalides :: Caisse
Client	**(Se Déplacer, p = 0)**	**(Lire, d = 5, p = 3)**	**(Lire, d = 2, p = 2)** **(Ramasser, d = 0, p = 1)**	**(Payer, d = 0, p = 10)**	*(Payer, d = 0, p = 10)*
Client Aveugle :: Client	*(Se Déplacer, p = 0)*		*(Lire, d = 0, p = 2)* *(Ramasser, d = 0, p = 1)*	*(Payer, d = 0, p = 10)*	**(Payer, d = 0, p = 11)**

(b) Forme étendue de la matrice d'interaction raffinée

FIGURE 8.22 – Matrice d'interaction brute sous forme synthétique et sous forme étendue décrivant le comportement de `Clients` et de `Clients Aveugles` dans un magasin.

La forme étendue de ces deux matrices est fournie dans les figures 8.21(b) et 8.22(b).

La spécification minimale d'une telle matrice consiste à ajouter un opérateur d'attribution de priorité pour chaque opérateur d'ajout apparaissant dans la forme synthétique de la matrice d'interaction brute, afin de caractériser la priorité par défaut d'une première moitié des éléments d'assignation. Ce principe est illustré par les opérateurs d'attribution de priorité présents dans la ligne associée à la famille d'agents `Client` dans la figure 8.22(a). La seconde moitié des priorités par défaut est attribuée aux éléments d'assignation en ajoutant un opérateur d'attribution de priorité pour chaque opérateur de modification appraissant dans la forme synthétique de la matrice d'interaction brute. Ce principe a abouti à l'ajout de l'opérateur !(LIRE, $d = 0, p = 2$) dans la ligne associée à la famille d'agents `Client Aveugle` dans la figure 8.22(a). Ensuite, d'autres opérateurs peuvent être ajoutés afin de rendre le comportement de l'agent plus spécifique. Ce principe est par exemple utilisé dans la figure 8.22(a) pour spécifier que PAYER a une priorité plus forte si l'interaction prend pour cible une `Caisse Invalides`.

8.3.2 Construction d'une forme étendue et implémentation du simulateur

La construction de la forme étendue d'une matrice d'interaction raffinée à partir de sa forme synthétique repose sur le même principe que l'application de l'effet des opérateurs de modification dans la matrice d'interaction brute.

La construction de la forme étendue d'une matrice d'interaction raffinée est un procédé itératif, lors duquel la présence de chaque opérateur dans la forme synthétique de la matrice d'interaction raffinée est interprétée, afin de déterminer de manière exhaustive toutes les interactions entre une famille d'agents source et une famille d'agents cible concernées par l'attribution de priorité. Ce procédé consiste dans un premier temps à remplir la forme étendue de la matrice d'interaction raffinée d'éléments de valeurs par défaut. Cette première étape consiste à :

- ajouter un élément $(I, p =?)$ dans la forme étendue de la matrice d'interaction raffinée pour chaque élément d'assignation (I) apparaissant dans la forme étendue de la matrice d'interaction brute ;
- ajouter dans la forme étendue de la matrice d'interaction raffinée un élément $(I, d = \delta, p =?)$ pour chaque élément d'assignation (I) apparaissant dans la forme étendue de la matrice d'interaction brute.

La seconde étape de ce procédé consiste à traiter séquentiellement chaque opérateur d'attribution de priorité en commençant la ligne des familles d'agents ne spécialisant aucune autre famille d'agents et en suivant progressivement leurs relations de spécialisation. Le traitement de l'attribution d'une priorité ρ à un élément d'assignation $(I, d = \delta)$ est alors formé des trois étapes qui suivent :

1. Soit L la ligne et C la colonne où se situe l'opérateur d'attribution de priorité dans la forme synthétique de la matrice d'interaction. La première étape est de remplacer l'élément $(I, d = \delta, p =?)$ situé à l'intersection de L et C dans la forme étendue par $(I, d = \delta, p = \rho)$.

2. Si la colonne C est associée à une famille d'agents ayant une ou plusieurs filles, alors le remplacement est aussi effectué à l'intersection de la ligne L et de la colonne de toutes ces familles d'agents filles.

3. Si la ligne L est associée à une famille d'agents ayant une ou plusieurs filles, alors chaque élément remplacé lors des étapes 1 et 2 dans la ligne L sont aussi remplacés dans la ligne associée à chacune de ces familles d'agents filles. Cette étape est effectuée récursivement pour toutes les familles d'agents filles rencontrées.

Ce procédé permet d'automatiser la construction de la forme étendue de la matrice d'interaction raffinée à partir de sa forme synthétique.

Tout comme la matrice d'interaction brute, la matrice d'interaction raffinée repose sur une forme synthétique et une forme étendue. Par conséquent, les principes suivis pour l'implémenter sont similaires à ceux énoncés dans le paragraphe intitulé « Implémentation d'une ligne de la matrice d'interaction brute. » à la page 241.

8.3.3 Limites de la spécialisation dans un modèle réactif de sélection d'interaction

L'attribution de priorités dans une matrice d'interaction raffinée sous forme synthétique repose sur le principe de la propagation de l'effet de l'opérateur d'attribution des priorités.

Dans le cas général, cette propagation permet de construire une matrice d'interaction raffinée en utilisant un nombre d'opérateurs bien moins important que le nombre d'éléments d'assignation auxquels il faut attribuer une priorité. Toutefois, dans certains cas, réutiliser les priorités définies dans des familles d'agents mères peut poser problème, voire s'avérer inutile.

Dans le cas d'une spécialisation de familles d'agents prenant la forme d'un losange, un même élément d'assignation peut se voir attribuer des priorités différentes dans deux familles d'agents mères. On se retrouve alors dans un cas similaire à celui décrit dans la figure 8.20 page 8.20, qui se résout de la même façon : par un choix manuel effectué par le concepteur.

Un second problème survient aussi dans le cas où une famille d'agents spécialise plusieurs familles d'agents mères. Ce problème est lié à différence sémantique existant dans l'utilisation de la spécialisation dans la matrice d'interaction brute et dans la matrice d'interaction raffinée. La matrice d'interaction brute définit les interactions que les agents sont capables d'initier ou de subir. Dans la forme étendue de cette matrice, le comportement par défaut de la spécialisation consiste à construire chaque cellule de la ligne associée à la famille d'agents fille en lui ajoutant les éléments d'assignation présents dans la ligne associée à ses familles d'agents mères. Ce procédé permet d'exprimer que si une famille d'agents mère est capable d'initier une interaction, alors ses familles d'agents filles en sont aussi capables par défaut. Il peut donc être automatisé.

La matrice d'interaction raffinée utilise les priorités afin d'établir une relation d'ordre entre des éléments d'assignation *situés sur une même ligne*. Par conséquent, lorsqu'une famille d'agents spécialise deux familles d'agents mères pouvant initier au moins une interaction, elle doit mettre en commun les relations d'ordres définies pour la ligne de chacune de ses familles d'agents mères. Dans un tel cas, rien ne garantit que les priorités utilisées dans les familles d'agents mères peuvent être réutilisées directement dans la familles d'agents fille afin d'établir une relation d'ordre valide parmi ses éléments d'assignation. Illustrons ce point sur un exemple.

Exemple.
Considérons la matrice d'interaction raffinée définie dans les figures 8.23(a) et 8.23(b). Dans cette matrice, un `Loup` *est à la fois un agent* `Mobile` *et un être* `Vivant`*. Un être* `Vivant` *est caractérisé par le fait qu'il peut* Mourir*. Puisqu'il s'agit de la seule interaction apparaissant dans cette ligne, nous faisons le choix de lui attribuer la priorité* 0*. Un agent* `Mobile` *est quant à lui caractérisé par le fait qu'il peut* Se Déplacer *dans l'environnement et, si ça n'est pas possible, d'*Attendre *dans l'environnement qu'il puisse à nouveau se déplacer. La relation d'ordre entre ces deux interactions est alors* Se Déplacer > Attendre*, ce que nous traduisons par deux priorités* 1 *et* 0*. Ces descriptions sont résumées par la forme synthétique de la matrice d'interaction raffinée présente dans la figure 8.23(a).*

Une telle spécification aboutit à la forme étendue résumée sur la figure 8.23(b), dans laquelle on peut remarquer que l'interaction Mourir *a une priorité moins élevée que l'interaction* Se Déplacer*, ce qui a pour conséquence qu'un* `Loup` *est capable de* Se Déplacer *alors qu'il était sensé* Mourir*.*

Source \ Cible	∅
Vivant	!(Mourir, p = 0)
Mobile	!(Se Déplacer, p = 1) !(Attendre, p = 0)
Loup :: Mobile Vivant	

(a) Forme synthétique de la matrice d'interaction raffinée

Source \ Cible	∅
Vivant	(Mourir, p = 0)
Mobile	(Se Déplacer, p = 1) !(Attendre, p = 0)
Loup :: Mobile Vivant	(Mourir, p = 0) (Se Déplacer, p = 1) !(Attendre, p = 0)

(b) Forme étendue de la matrice d'interaction raffinée

Figure 8.23 – Illustration du problème lié à la réutilisation de priorités définies dans des familles d'agents mères différentes.

La solution du problème mentionné dans l'exemple précédent est évidente : il faut donner une priorité plus élevée à l'interaction MOURIR. Cette solution prend en compte dans la famille d'agents `Vivant` les priorités attribuées dans la ligne associée à la famille d'agents `Mobile`, alors que ces familles ne sont pas corrélées. En effet, une `Plante` est un `Vivant` mais pas un `Mobile`, un `Robot` est un `Mobile` mais pas un `Vivant`. Attribuer des priorités dans la ligne associée à `Mobile` et dans la ligne associée à `Vivant` satisfaisant tous les situations est impossible. Par conséquent, la seule solution à ce problème consiste à réattribuer la priorité de tous les éléments d'assignation présents dans la ligne associée à la famille d'agents `Loup`. Dans ce cas, à défaut de pouvoir être utilisées directement, les priorités fournies dans la ligne de chacune des familles d'agents mères de `Loup` peuvent servir d'indicateurs permettant de construire une relation d'ordre, qui sera par la suite convertie en un ensemble de priorités.

La réutilisation de comportements dans le modèle réactif de sélection d'interaction est donc moins facile à caractériser que la réutilisation dans une matrice d'interaction brute.

8.4 Synthèse du chapitre

Les outils de conception de simulation informatique doivent à la fois faciliter la description d'une explication à un phénomène et garantir la robustesse de son implémentation face aux révisions de modèle. Ils doivent donc concilier ingénierie logicielle et ingénierie des connaissances. Dans ce chapitre, nous montrons que IODA facilite l'obtention d'un tel compromis.

Modèle exhaustif et spécification synthétique

La construction de simulations large échelle nécessite un compromis entre deux problèmes semblant s'opposer :

- le modèle doit faire exhaustivement état de toutes les interactions ayant lieu entre les agents de la simulation afin d'interpréter plus facilement le modèle conçu ;
- la construction d'un modèle se révèle être fastidieuse, peu robuste aux modifications et parfois impossible s'il faut décrire manuellement tous les cas d'interactions entre agents.

Ce problème n'était jusqu'à présent pas traité pour deux raisons. D'une part, la plupart des approches existantes ne représentent pas toute action impliquant simultanément plusieurs agents en tant qu'interaction. Elles ne peuvent donc pas bénéficier d'un modèle offrant une vue d'ensemble exhaustive sur toutes les interactions ayant lieu entre les agents. D'autre part, même si certaines approches fournissent une représentation des connaissances s'en approchant, elles se limitent à la description d'un modèle formel. Elles ne prennent pas en compte les problèmes liés à la méthodologie permettant de spécifier de tels modèles.

IODA repose sur des concepts facilitant la résolution d'un tel dilemme. Pour étayer ce point, nous décrivons dans ce chapitre une extension de l'approche IODA reposant sur le concept de **spécialisation** similaire à la relation « sorte de » rencontrée dans les langages orientés objets.

La spécialisation exprime une relation sémantique entre deux familles d'agents ayant deux conséquences sur la matrice d'interaction brute :

- une famille d'agents peut être la cible de toutes les interactions pouvant être subies par une famille d'agents qu'elle spécialise. Par exemple, si `Mouton` spécialise `Herbivore` et qu'un `Loup` peut MANGER un `Herbivore`, alors un `Loup` peut aussi MANGER un `Mouton` ;
- une famille d'agents peut être la source de toutes les interactions pouvant être initiée par une famille d'agents qu'elle spécialise. Par exemple, si `Mouton` spécialise `Vivant` et qu'un `Vivant` peut MOURIR, alors `Mouton` peut aussi MOURIR.

La matrice d'interaction brute est donc décrite à l'aide d'un nombre beaucoup plus réduit d'interactions et permet donc la spécification synthétique recherchée.

Pour concilier spécification non-exhaustive et vue exhaustive sur les interactions ayant lieu dans la simulation, la matrice d'interaction brute est représentée sous deux formes.

- La **forme synthétique** est une version de la matrice ne faisant apparaître qu'un nombre restreint d'interactions. Elle est utilisée à des fins de spécification.

– La **forme étendue** est un version de la matrice faisant apparaître de manière exhaustive toutes les interactions pouvant avoir lieu entre les agents de la simulation. Elle est utilisée non seulement à des fins d'interprétation, mais aussi à des fins d'implémentation. En effet, elle peut être directement utilisée dans les algorithmes décrits dans le cœur de l'approche IODA et ainsi aboutir à une implémentation.

Des algorithmes permettent de déduire automatiquement une forme étendue à partir d'une forme synthétique et permettent donc d'atteindre le compromis recherché.

La spécialisation caractérisée ici correspond à la transposition de l'héritage à la conception de simulations large échelle. Elle permet de concevoir des familles d'agents comme des extensions d'autres familles d'agents.

Dépasser les limites de l'héritage

Certains principes d'ingénierie logicielle de l'héritage constituent une limite du point de vue de l'ingénierie des connaissances dans les simulations. L'approche usuelle en ingénierie logicielle est de concevoir les objets par ajouts et surcharge de méthodes, ce qui se transpose dans les simulations par l'ajout ou la surcharge d'interaction. Un problème se pose alors s'il y a ajout d'une exception à une famille d'agents existante, par exemple l'ajout de `Clients aveugles` modélisés comme des `Clients` particuliers ne pouvant pas LIRE. En effet, cet ajout ne peut être fait qu'en créant une famille d'agents intermédiaire factorisant leurs interactions communes. L'ajout d'une exception implique donc la modification de familles d'agents existantes.

Afin de remédier à ce problème, nous nous détachons de ce point de vue de l'héritage en permettant d'exprimer les exceptions. Dans ce but, la forme synthétique de la matrice d'interaction brute est construite à l'aide de trois **opérateurs de spécialisation** :

– l'opérateur d'**ajout** qui permet d'ajouter à un agent la capacité d'initier une interaction avec un autre agent pour cible. Cet opérateur était utilisé implicitement jusqu'à présent ;
– l'opérateur de **modification** qui permet de modifier la garde de distance d'une interaction que l'agent peut initier ;
– l'opérateur de **retrait** qui permet de retirer à un agent la capacité d'initier une interaction avec un agent particulier pour cible.

Ces opérateurs permettent à la fois de reproduire les schémas de conception de l'héritage et d'exprimer les exceptions aux familles d'agents existantes. En effet, l'ajout d'une interaction s'exprime avec un opérateur d'ajout, la surcharge d'une interaction est exprimée soit en redéfinissant une primitive abstraite, soit en utilisant un opérateur de modification et les exceptions sont exprimées avec des opérateurs de retrait.

Limites de la spécialisation

Afin de faciliter encore plus la conception de telles simulations, la conception du modèle de sélection d'interaction pourrait être facilitée en usant de concepts similaires.

La conception de modèles de comportements réactifs basés sur des priorités peut effectivement être simplifiée en définissant une forme synthétique et étendue de la matrice d'interaction raffinée et un opérateur d'attribution de priorités. Nous prouvons ce point par la description d'algorithmes automatisant l'attribution de priorités dans une famille d'agents spécialisant d'une autre famille d'agents.

Toutefois, le cas de l'héritage multiple nécessite une attention particulière. En effet, une priorité n n'a pas systématiquement le même sens dans deux familles d'agents indépendantes. Par conséquent, les priorités fournies automatiquement à une famille d'agents héritant de plusieurs familles d'agents différentes n'est pas satisfaisante dans tous les cas. Une vérification sémantique de ces priorités est nécessaire de la part du concepteur. Cette limite est inhérente à la spécification de comportements réactifs en utilisant une adaptation de l'héritage.

En conclusion, IODA fournit un compromis entre ingénierie logicielle et ingénierie des connaissances ne pouvant être obtenu dans d'autres approches actuelles. En effet, elle offre une vue d'ensemble sur les interactions d'une simulation sans pour autant avoir à en faire une description exhaustive. Il y est de plus possible de décrire autant des extensions que des exceptions au comportement d'une famille d'agents. La

conception des agents s'en trouve facilitée tout en limitant les modifications nécessaires lors de révisions du modèle.

Conclusion

1 Synthèse

Le paradigme agent repose sur une métaphore ayant permis de grandes avancées scientifiques durant ces deux dernières décennies, en particulier dans le cas de la simulation explicative. En effet, la notion d'agent autonome est un outil de choix permettant d'explorer et de trouver les comportements individuels à l'origine d'un phénomène macroscopique étudié.

Problématique : faciliter la conception de simulations large échelle

La construction de simulations multi-agents n'est pas aisée. Elle nécessite en effet de passer d'hypothèses exprimées en langage naturel à leur implémentation dans un langage de programmation en s'assurant :

- de retranscrire aussi fidèlement que possible les spécifications des experts du domaine dans l'implémentation ;
- de compléter les spécifications des experts du domaine par des informations nécessaires à la constitution du programme informatique ;
- de faciliter la modification des hypothèses et donc de l'implémentation.

Du point de vue méthodologique, les solutions fournissant le meilleur compromis aux problèmes mentionnés ici sont les approches liées à l'ingénierie de systèmes multi-agents dirigée par les modèles. Elles ne sont toutefois pas satisfaisantes du point de vue de la structure du modèle et de l'implémentation. En effet, elles se focalisent uniquement sur les entités et assujettissent les interactions à ces entités. Les seules interactions y sont des échanges de messages, alors qu'en simulation cette notion représente toute autre action impliquant plusieurs entités comme les réactions chimiques, les comportements d'animaux, *etc.*

Du point de vue de la structure du modèle et de l'implémentation, les solutions fournissant le meilleur compromis sont celles issues de la théorie des affordances. Ces approches sont toutefois dédiées aux agents cognitifs et ne sont pas adaptées pour modéliser des agents réactifs. De plus, elles ne sont pas satisfaisantes du point de vue méthodologique. En effet, la structure du modèle est connue, mais aucun moyen permettant de parvenir à une description du phénomène dans ce modèle n'est fourni.

Ainsi, aucune approche actuelle n'est adéquate à la fois du point de vue méthodologique et du point de vue de la représentation des connaissances.

Solution défendue : utilisation d'une approche centrée sur les interactions

Dans cette thèse, nous montrons que cinq principes suffisent à établir le compromis recherché :

- chaque entité du phénomène doit être concrétisée par un agent ;
- chaque comportement d'un agent doit être concrétisé par une interaction ;
- une interaction doit pouvoir représenter tout ensemble d'actions impliquant simultanément plusieurs agents ;
- les interactions doivent être réifiées indépendamment des spécificités des agents ;
- il doit y avoir séparation explicite entre agents, interactions et sélection d'interaction, que cela soit dans le modèle ou dans l'implémentation.

En effet, la concrétisation des agents et des interactions en élément logiciels disjoints favorise la constitution de bibliothèques d'agents et d'interactions réutilisables, réduisant ainsi les efforts de modification lors des révisions du modèle. La séparation explicite entre agents, interactions et sélection d'interaction a permis de construire une méthodologie de conception complétant graduellement les modèles, en passant de description abstraites où les agents se voient attribuer dans une matrice d'interaction des interactions qu'ils peuvent effectuer ou subir, à des descriptions fines où le comportement spécifique des agents est caractérisé. Une telle méthodologie facilite ainsi la construction de modèles conséquents, contenant un grand nombre d'agents interagissant de manière diversifiée. La représentation des connaissances est unifiée par les notions d'agents et d'interactions, réduisant ainsi la complexité du modèle utilisé.

IODA : l'approche centrée sur les interactions

Afin de concevoir des simulations selon ces principes, nous avons développé une approche de conception de simulations multi-agents centrée sur les interactions appelée IODA (Interaction Oriented Design of Agent simulations). Elle est composée d'un modèle formel, d'algorithmes de simulation et d'une méthodologie fournissant les outils théoriques permettant de concevoir graduellement un modèle selon notre approche, et de conserver sa structure lors de l'implémentation. Afin de confirmer la faisabilité de l'approche IODA, nous avons développé :

- une implémentation fidèle des principes de IODA dans une plateforme de simulation intitulée JEDI (Java Environment for the Design of agent Interactions) ;
- une implémentation fidèle de la méthodologie nommée JEDI-BUILDER permettant de créer un modèle IODA et de le transformer en du code pour JEDI.

Nous nous sommes appuyés sur l'approche IODA, et son implémentation dans JEDI et JEDI-BUILDER, afin d'explorer selon trois thématiques différentes les possibilités offertes par l'approche centrée sur les interactions que nous défendons.

Faut-il nécessairement des interactions complexes pour créer des comportements complexes ?

Dans un premier temps, nous avons cherché à savoir s'il fallait nécessairement intégrer au modèle formel des interactions complexes faisant intervenir un grand nombre d'agents afin de modéliser des comportements complexes. Nous avons alors identifié deux patrons de conception permettant de décomposer la plupart des interactions complexes en un ensemble d'interactions simples, étant soit réflexives (*i.e.* l'action d'un agent sur son propre état), soit n'impliquant que deux agents à la fois, montrant ainsi que la majorité des simulations peuvent être modélisées à l'aide d'interactions simples. Nous avons toutefois aussi montré que ces patrons de conception n'étaient pas suffisants pour modéliser les interactions ayant un effet nécessairement simultané sur plusieurs agents. Nous avons donc formalisé une extension de l'approche IODA aux interactions multicast, qui impliquent simultanément un nombre d'agents déterminé dynamiquement.

Dans un deuxième temps, nous avons exploré comment utiliser l'approche IODA afin d'éviter la construction de modèles et d'implémentations biaisés. Nous nous reposons pour cela sur l'approche IODA, et son implémentation dans JEDI et JEDI-BUILDER, afin de faire apparaître explicitement des choix de conception apparaissant autrement de manière implicite ou diffuse dans la plupart des autres approches de conception.

Comment éviter d'introduire des biais liés à la participation simultanée à plusieurs interactions ?

Le postulat simplificateur « tout agent peut subir un nombre quelconque d'interactions simultanément » se retrouve implicitement dans une majorité de simulations. Nous avons étudié si ce postulat permettait de conduire une simulation dans le cas général ou s'il constitue un biais et avons conclu que certaines simulations ne peuvent pas être modélisées sur ce postulat. Nous avons alors déterminé un modèle simple poussant à fournir explicitement une réponse à la question « un agent peut-il participer à une interaction, s'il est déjà impliqué dans un ensemble d'autres interactions ? ». Ce modèle consiste

à étiqueter les interactions d'une simulation par le terme « exclusive » ou le terme « parallèle ». Des algorithmes interprètent alors cet étiquetage pour définir précisément comment la simultanéité est gérée. Cela permet d'éviter d'introduire certains biais liés à la participation simultanée d'un agent à plusieurs interactions.

Comment éviter d'introduire des biais liés à la spécification de comportements stochastiques ?

Il existe des façons très différentes d'implémenter un comportement stochastique basé sur des décisions réactives. Seule une des façons d'implémenter permet d'aboutir à des résultats corrects. Cette façon d'implémenter n'est pas toujours facilement identifiable. En effet, en l'absence de connaissances théoriques *a priori*, les seuls moyens de la déterminer consistent :

- soit à comparer statistiquement les résultats expérimentaux obtenus par simulation avec des résultats considérés comme viables ;
- soit à interpréter l'implémentation et à en déduire les probabilités réellement utilisées.

Afin de remédier à ce problème, nous explorons comment utiliser l'approche IODA afin de faire apparaître explicitement dans le modèle les différentes façons d'implémenter un tel comportement, sans avoir recours à des représentations complexes ou à des solutions propres à un domaine d'application.

Nous soutenons que le comportement d'agents réactifs doit reposer sur ce que nous appelons des **politiques de sélection d'interaction**. De telles politiques doivent être définies en deux parties :

1. établir si le choix porte :
   - d'abord sur une interaction puis sur l'agent qui subira l'interaction ;
   - ou d'abord sur un agent qui subira une interaction puis sur l'interaction ;
   - ou enfin directement sur un couple composé d'une interaction et d'un agent qui subira l'interaction ;
2. établir pour chaque sélection si le choix est effectué :
   - aléatoirement ;
   - ou en cherchant à maximiser une fonction d'évaluation ;
   - ou en pondérant chaque élément pouvant être sélectionné.

Un tel modèle fait apparaître explicitement les choix d'implémentation effectués. Il permet de plus d'interpréter plus simplement le comportement des agents que par l'analyse directe du code.

Définition de comportements par extension et par exception.

Pour concevoir des simulations large échelle, un compromis entre deux problèmes semblant s'opposer doit être trouvé :

- le modèle doit faire exhaustivement état de toutes les interactions ayant lieu entre les agents de la simulation afin d'interpréter plus facilement le modèle conçu ;
- la construction d'un modèle se révèle être fastidieuse, peu robuste aux modifications et parfois impossible s'il faut décrire manuellement tous les cas d'interactions entre agents.

IODA repose sur des concepts facilitant la résolution d'un tel dilemme. Pour étayer ce point, nous avons défini un concept de **spécialisation** similaire à la relation « sorte de » rencontrée dans les langages orientés-objet. En utilisant cette relation, la matrice d'interaction brute est décrite à l'aide d'un nombre beaucoup plus réduit d'interactions et permet donc la spécification synthétique recherchée.

Pour concilier spécification non-exhaustive et vue exhaustive sur les interactions ayant lieu dans la simulation, la matrice d'interaction brute est représentée sous deux formes.

- La **forme synthétique** est une version de la matrice ne faisant apparaître qu'un nombre restreint d'interactions. Elle est utilisée à des fins de spécifications.
- La **forme étendue** est un version de la matrice faisant apparaître de manière exhaustive toutes les interactions pouvant avoir lieu entre les agents de la simulation. Elle est utilisée non seulement à des fins d'interprétation, mais aussi à des fins d'implémentation. En effet, elle peut être directement utilisée dans les algorithmes décrits dans le cœur de l'approche IODA et ainsi aboutir à une implémentation.

Des algorithmes permettent de déduire automatiquement une forme étendue à partir d'une forme synthétique et donc d'atteindre le compromis recherché.

De plus, nous définissons des opérateurs de spécialisation permettant de dépasser les limites inhérentes à l'héritage et d'exprimer des agents non seulement comme une extension, mais aussi comme une exception d'autres agents.

Favoriser le principe de parcimonie avec IODA.

Nous avons ainsi défini quatre extensions disjointes de IODA qui fournissent des concepts et des outils méthodologiques permettant de traiter plus efficacement quatre problèmes de conception spécifiques :

1. Des interactions complexes sont-elles nécessaires pour modéliser des comportements complexes ?
2. Comment éviter d'introduire des biais dans une simulation ? Ce problème a été étudié selon deux perspectives :
   (a) Une entité peut-elle participer simultanément à plusieurs interactions ?
   (b) Comment s'assurer que le comportement d'un agent est implémenté correctement, sans avoir recours à des représentations complexes ou à des solutions propres à un domaine d'application ?
3. Comment transposer l'héritage à la simulation afin de faciliter l'ingénierie des connaissances ?

L'approche IODA reste toutefois valide en l'absence de telles extensions et permet déjà d'implémenter un grand nombre de simulations dans des domaines très différents. La complexité de l'approche que nous proposons dans ce manuscrit doit en pratique être adaptée à la complexité des problèmes traités. Cela permet de ne complexifier les modèles utilisés que lorsque cela se révèle nécessaire et facilite donc la conception de modèles respectant le principe de parcimonie.

2 Applications

Les contributions décrites dans ce manuscrit sont actuellement mises en pratique dans différentes applications, à l'occasion de projets auxquels l'équipe SMAC participe.

Le cœur de l'approche IODA a été utilisé afin de modéliser les rythmes circadiens de l'*Ostreococcus tauri* [PCSB07], ce qui témoigne de l'intérêt de l'approche centrée sur les interactions afin de concevoir des simulations explicatives, dans le domaine d'application de la biochimie cellulaire.

Les extensions de IODA décrites dans ce manuscrit sont également utilisées dans un projet intitulé FormatStore [20] réalisé en collaboration avec l'entreprise Idées-3com et l'école de commerce ENACO. Ce projet a pour objectif de modéliser des clients virtuels dans un magasin. La simulation du magasin est alors utilisée comme moyen d'évaluation des performances d'étudiants en école de commerce, que cela soit au poste de manager de magasin ou de commercial auprès de clients.

L'approche IODA a aussi servi de support à la construction d'un explorateur de l'espace des simulations appelé LEIA (LEIA lets you Explore your Interactions for your Agents) [GKMP10, GKMP08]. LEIA repose sur la génération aléatoire de modèles, rendue possible par IODA en attribuant aléatoirement des interactions aux agents dans la matrice d'interaction. L'exploration de l'espace des simulations consiste alors à générer aléatoirement un ensemble de n modèles et à en exécuter la simulation. À l'aide de fonctions d'évaluation se basant entre autre sur l'interprétation de la matrice d'interaction, une note est attribuée à chaque modèle afin de déterminer son intérêt. Des déclinaisons du modèle considéré comme le plus intéressant sont alors explorées de la même manière, en générant aléatoirement n variantes par l'ajout, la modification ou la suppression de capacité d'interagir. Cela n'est évidemment pas réalisable sans la séparation agents/interactions d'une part ni la séparation déclaratif/procédural d'autre part.

3 Perspectives

Les perspectives aux travaux présentés dans cette thèse consistent à poursuivre l'exploration des problématiques de simulation informatique selon l'approche IODA. Ces études permettront de construire de

20. dont une description courte est disponible à l'url `http://www2.lifl.fr/SMAC/projects/formatstore/`

nouvelles extensions méthodologiques à IODA pour faciliter la résolution de problématiques actuellement traitées de manière non satisfaisante par une approche « classique » de la simulation.

JEDI-BUILDER et extensions méthodologiques

L'approche IODA s'est enrichie d'extensions méthodologiques et logicielles permettant d'adapter la complexité de la méthodologie à la complexité des problèmes traités. Toutefois, le choix des extensions à utiliser ou non pour modéliser un phénomène n'est pas aisé pour un non informaticien. Il serait donc intéressant d'automatiser ces choix en intégrant les critères de décision correspondants dans JEDI-BUILDER, ce qui n'est pas le cas actuellement.

Modélisation d'agents réflexifs

Nous avons présenté des situations où la matrice d'interaction est fixe. Or, le comportement de certaines entités peut évoluer au fil du temps dans un phénomène, soit par un apprentissage effectué par l'entité, soit parce que l'entité a subi un changement d'état affectant son comportement (par exemple attraper une maladie). En l'état, IODA permet à des agents d'apprendre et de s'adapter dans une certaine mesure à leur environnement par exemple via les interactions de mise à jour et les primitives des déclencheurs. Toutefois, notre approche permet un apprentissage de plus grande ampleur, dans la mesure où elle offre aux agents la capacité de connaître, interpréter ou modifier leur propre modèle.

Il suffit en effet :

- soit de fournir des primitives abstraites aux agents leur permettant l'introspection (*i.e.* la lecture des matrices d'interaction brute et raffinée) et l'intercession (*i.e.* la modification des matrices d'interaction brute et raffinée) ;
- soit de décrire plusieurs matrices d'interactions, représentant le comportement des agents selon l'état dans lequel ils se situent.

Séparation modèle du phénomène/modèle d'une expérience

Lors de l'implémentation de IODA dans JEDI nous avons identifié non seulement les classes qui permettent d'implémenter tous les concepts de IODA, mais aussi un ensemble de classes qu'il est nécessaire de définir afin de pouvoir réaliser des expériences. Sur cette base, il serait possible d'établir la structure d'un modèle précis permettant de décrire des expériences conformes à l'approche IODA, ce qui fait défaut à toutes les approches de conception de simulations disponibles actuellement. On pourrait alors :

- assurer la reproductibilité des expériences par des personnes différentes et favoriser ainsi la validation et la diffusion des résultats expérimentaux ;
- générer automatiquement des expériences sans pour autant avoir à modifier l'implémentation du modèle du phénomène.

Modélisation de phénomènes ayant lieu à plusieurs échelles

Dans de nombreux systèmes complexes, plusieurs échelles spatiales et temporelles coexistent. Par exemple en cosmologie (où des galaxies sont composées de systèmes planétaires en interaction, eux mêmes composés d'astres pouvant interagir), en simulation de foules (dans un centre commercial composé d'un emboîtement de magasins), ou encore encore en biologie cellulaire (où les métabolites et les protéines passent du noyau au cytoplasme, puis à l'organe et vice-versa). De plus, dans certains cas, le comportement d'un agent peut varier en fonction de l'endroit où il se situe. Par exemple, le comportement d'un client n'est pas le même dans les allées de la galerie marchande, dans un magasin, ou dans un rayon particulier de ce magasin. De même une enzyme peut n'agit que dans le cytoplasme et pas dans le noyau cellulaire.

La modélisation de tels phénomènes se trouve facilité s'il est possible :

- d'exprimer une relation d'imbrication entre les agents, par exemple qu'une entité est composée de sous-entités ;
- d'exprimer simplement des comportements différents pour un agent selon l'environnement dans lequel il se situe ;
- d'exprimer simplement les interactions survenant entre des agents situés à des échelles différentes ;

– d'exprimer simplement des comportements ayant lieu à des échelles temporelles différentes.

Un modèle nommé PADAWAN (*Pattern for the Accurate Design of Agent Worlds in Agent Nests*) [PMK10] développé par Sébastien Picault et auquel nous avons contribué se base sur l'approche IODA afin d'exprimer de telles propriétés. Il se repose pour cela sur une extension de l'approche IODA et de sa matrice d'interaction, selon cinq principes :

– Il existe plusieurs environnements dotés chacun d'une période et d'une fréquence déterminant le rythme de la prise de parole des agents.
– Chaque environnement se voit associer une matrice d'interaction.
– Le comportement d'un agent est défini par la ligne de la matrice d'interaction de (ou des) environnement(s) où il se situe.
– Chaque agent peut héberger un environnement, *i.e.* constituer un environnement pour d'autres agents.
– Chaque agent peut être situé dans plusieurs environnements.

Ces multiples pistes sont rendues possibles par le choix que nous avons fait de séparer, d'une part, interactions et agents et d'autre part, la partie déclarative des modèles de la partie procédurale. C'est cela qui fait de IODA une approche fructueuse dont le potentiel applicatif, déjà attesté dans des domaines variés, commence seulement à être utilisé. De plus, cette approche centrée sur les interactions ouvre la voie à une relecture générale des concepts fondamentaux de l'approche orientée-agent.

Annexe A

Modélisation et implémentation d'environnements euclidiens avec IODA

Dans cet annexe, nous décrivons comment compléter le modèle IODA afin qu'il intègre les données nécessaires l'utilisation d'un environnement euclidien en deux dimensions. Cette description est principalement constituée de la spécification :

- des différentes primitives de l'environnement ;
- d'un halo particulier ;
- des primitives abstraites des agents issues de la signature des entités dans l'environnement ou de la signature des entités dans le halo proposé.

Puisque nous cherchons ici à compléter le modèle présenté dans le chapitre 3, nous décrivons précisément à la fois les concepts utilisés et la spécification précise de la plupart des primitives de l'environnement. Si jamais le lecteur ne s'intéresse qu'à la partie émergée de cet environnement, *i.e.* la partie manipulée par les concepteurs pour concevoir des simulations utilisant ce type d'environnement, il peut directement se référer à la section A.6.

A.1 Caractérisation de l'environnement

L'environnement que nous modélisons ici est un espace euclidien continu en deux dimensions, pouvant être torique selon son axe des abscisses, son axe des ordonnées, les deux ou aucun des deux. Nous imposons que l'environnement soit un rectangle, caractérisé en particulier par une largeur et une hauteur, que nous supposons définies lorsque l'environnement est créé, de même que le fait que l'environnement est un tore ou pas. Cette description aboutit à l'identification de plusieurs primitives de l'environnement :

- $\langle largeurEnvironnement, Nombre\ Entier, \emptyset \rangle \in primitives_{env}$
- $\langle hauteurEnvironnement, Nombre\ Entier, \emptyset \rangle \in primitives_{env}$
- $\langle toriqueEnAbscisses, Booleen, \emptyset \rangle \in primitives_{env}$
- $\langle toriqueEnOrdonnees, Booleen, \emptyset \rangle \in primitives_{env}$

La spécification de ces primitives de l'environnement décrite sur l'algorithme 21 amène à identifier une partie de la structure de données de l'environnement. Nous identifions en effet quatre attributs :

- la longueur de l'environnement ;
- la largeur de l'environnement ;
- un booléen déterminant si l'environnement est un tore sur l'axe des abscisses ;
- un booléen déterminant si l'environnement est un tore sur l'axe des ordonnées.

Afin de caractériser plus précisément cet environnement, il nous faut déterminer :

- Comment est calculée la notion de distance et par conséquent comment est gérée la position des entités ;
- Comment ajouter les entités dans l'environnement, comment les en retirer et comment avoir connaissance des entités dans l'environnement ;
- Comment déplacer les entités dans l'environnement ;

Algorithme 21 : Spécification des primitives de l'environnement intitulées « largeurEnvironnement », « longueurEnvironnement », « environnementToriqueEnAbscisses » et « environnementToriqueEnOrdonnees ».

```
largeurEnvironnement()
début
  | retourner this.largeur;
fin
```

```
hauteurEnvironnement()
début
  | retourner this.hauteur;
fin
```

```
environnementToriqueEnAbscisses()
début
  | retourner this.environnementToriqueEnAbscisses;
fin
```

```
environnementToriqueEnOrdonnees()
début
  | retourner this.environnementToriqueEnOrdonnees;
fin
```

– Comment définir le halo des entités dans l'environnement.

A.2 Distance entre entités et position des entités

Nous considérons que les entités sont représentées dans l'environnement par une position, *i.e.* un couple de nombres à virgule flottante, ainsi que par une surface qu'elles occupent au sol. Pour des raisons de performances, nous imposons que cette surface soit un rectangle centré sur la position de l'entité dont les bords sont parallèles aux bords de l'environnement.

La distance entre deux entités e_1 et e_2 se mesure de manières différentes selon la nature torique ou non de l'environnement (voir figure A.1). Si l'environnement est non-torique, il s'agit de la distance entre e_1 et e_2 à leurs positions réelles. Si l'environnement est torique selon l'axe des abscisses, cette distance est le minimum entre :

– la distance de e_1 à e_2 ;
– la distance de e_1 au translaté de e_2 vers la gauche, d'une distance égale à la largeur de l'environnement ;
– la distance entre e_1 et le translaté de e_2 vers la droite, d'une distance égale à la largeur de l'environnement.

Le calcul de la distance dans un environnement torique selon l'axe des ordonnées est similaire, en effectuant des translations vers le haut et vers le bas, d'une longueur égale à la hauteur de l'environnement. Enfin, le calcul de la distance dans environnement torique selon les deux axes consiste à prendre le minimum des distances entre e_1 et toutes les combinaisons possibles de translations vers le haut, la gauche, la droite et le bas de e_2.

Une spécification possible de la distance dans ce type d'environnement selon l'approche IODA est décrite sur l'algorithme 22. Cet algorithme manipule les quatre primitives de perception qui suivent, qui sont donc ajoutées à la signature des entités dans l'environnement :

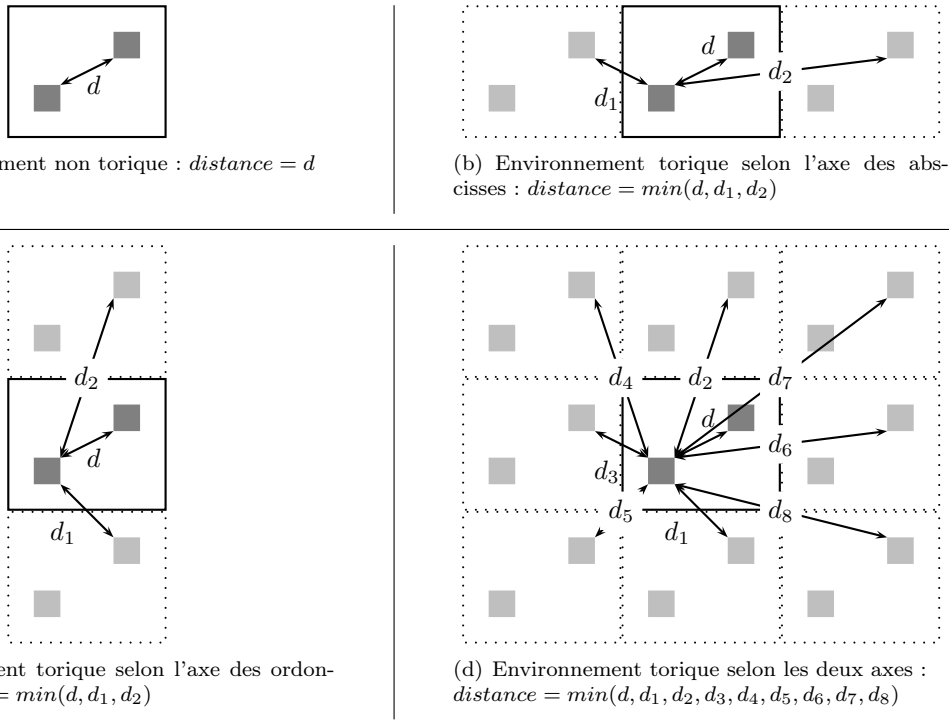
– $\langle abscisse, Nombre\ a\ Virgule\ Flottante, \emptyset\rangle \in sign_{entites}(environnement)$
– $\langle ordonnee, Nombre\ a\ Virgule\ Flottante, \emptyset\rangle \in sign_{entites}(environnement)$
– $\langle largeur, Nombre\ a\ Virgule\ Flottante, \emptyset\rangle \in sign_{entites}(environnement)$
– $\langle hauteur, Nombre\ a\ Virgule\ Flottante, \emptyset\rangle \in sign_{entites}(environnement)$

Ces dernières permettent de connaître la position d'une entité, ainsi que les dimensions de sa surface. Nous utilisons des primitives de perception plutôt que de mémoriser ces informations dans l'environnement afin d'optimiser les simulateurs qui seront construits à l'aide de ce modèle. Ces primitives permettent d'accéder à la valeur d'un attribut éponyme se situant dans les entités.

L'algorithme précédent manipule de plus une primitive de l'environnement intitulée « distanceNonTorique », permettant de mesurer la distance entre deux surfaces dans un environnement non-torique. Son calcul se fait de la manière qui suit : si les surfaces ont une intersection vide, la distance entre elles est mesurée à l'aide de la distance de Hausdorff (*i.e.* il s'agit de la distance séparant les points les plus proches des deux rectangles). Sinon, elle la distance entre les entités est nulle. Dans notre cas, la distance de Hausdorff entre deux surfaces A et B est calculée de quatre manières différentes, en fonction de la position relative de A par rapport à B (voir figure A.2). L'algorithme 23 spécifie la primitive

Algorithme 22 : Spécification de la primitive de l'environnement $\langle distance, Nombre\ a\ Virgule\ Flottante, \{Entite, Entite\}\rangle$ pour un espace euclidien continu en deux dimensions pouvant être torique selon l'axe des abscisses ou des ordonnées.

distance(e_1, e_2)
début
  $d \Leftarrow$ **environnement.***distanceNonTorique*($e_1.abscisse(), e_1.ordonnee(), e_1.largeur(), e_1.hauteur(), e_2.abscisse(), e_2.ordonnee(), e_2.largeur(), e_2.hauteur()$);
  si ***environnement.****toriqueEnAbscisses()* **alors**
    pour $x \in \{-$***environnement.****largeur*$(),$***environnement.****largeur*$()\}$ **faire**
      $d' \Leftarrow$ **environnement.***distanceNonTorique*($e_1.abscisse(), e_1.ordonnee(), e_1.largeur(), e_1.hauteur(), e_2.abscisse() + x, e_2.ordonnee(), e_2.largeur(), e_2.hauteur()$);
      si $d' < d$ **alors**
        $d \Leftarrow d'$;
  si ***environnement.****toriqueEnOrdonnees()* **alors**
    pour $y \in \{-$***environnement.****.hauteur*$(),$***environnement.****.hauteur*$()\}$ **faire**
      $d' \Leftarrow$ **environnement.***distanceNonTorique*($e_1.abscisse(), e_1.ordonnee(), e_1.largeur(), e_1.hauteur(), e_2.abscisse(), e_2.ordonnee() + y, e_2.largeur() e_2.hauteur()$);
      si $d' < d$ **alors**
        $d \Leftarrow d'$;
  si ***environnement.****toriqueEnAbscisses()* ***et environnement.****toriqueEnOrdonnees()* **alors**
    pour $x \in \{-$***environnement.****.largeur*$(),$***environnement.****.largeur*$()\}$ **faire**
      pour $y \in \{-$***environnement.****.hauteur*$(),$***environnement.****.hauteur*$()\}$ **faire**
        $d' \Leftarrow$ **environnement.***distanceNonTorique*($e_1.abscisse(), e_1.ordonnee(), e_1.largeur(), e_1.hauteur(), e_2.abscisse() + x, e_2.ordonnee() + y, e_2.largeur() e_2.hauteur()$);
        si $d' < d$ **alors**
          $d \Leftarrow d'$;
  retourner d;
fin



(a) Environnement non torique : $distance = d$

(b) Environnement torique selon l'axe des abscisses : $distance = min(d, d_1, d_2)$

(c) Environnement torique selon l'axe des ordonnées : $distance = min(d, d_1, d_2)$

(d) Environnement torique selon les deux axes : $distance = min(d, d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8)$

FIGURE A.1 – Principes régissant le calcul de la distance entre deux entités, en fonction de la nature torique ou non de l'environnement.

« distanceNonTorique » en se basant sur ces informations.

A.3 Ajout et retrait d'entités

Puisque l'environnement est un espace euclidien continu en deux dimensions pouvant être borné, une entité ne peut pas tout le temps être placée à une position particulière. Par conséquent, la primitive permettant d'ajouter une entité à l'environnement doit retourner une valeur, spécifiant si l'ajout à l'environnement a été effectif ou pas.

Au regard de ces informations, la primitive de l'environnement permettant d'ajouter une entité prend la forme de $\langle ajouter, Boolean, \{Entite, Nombre\ a\ Virgule\ Flottante, Nombre\ a\ Virgule\ Flottante\}\rangle$. De plus, la primitive de l'environnement permettant le retrait d'une entité prend la forme de $\langle retrirer, \emptyset, \{Entite\}\rangle$.

Notre approche se situe dans un cadre où une entité ne peut modifier sa ligne de la matrice d'interaction brute. Par conséquent, le maintien de l'ensemble $\mathbb{E}_{active}$ des entités actives de la simulation peut être fait lors de l'appel aux primitives d'ajout et de retrait d'une entité à l'environnement. Il en va de même pour l'ensemble $\mathbb{E}_{labile}$ des entités labiles et pour l'ensemble $\mathbb{E}_{passive}$ des entités passives. L'algorithme 24 décrit la spécification de ces deux primitives.

La spécification de ces primitives amène à identifier une nouvelle primitive de l'environnement, intitulée « rectangleDansLesBornes », dont le rôle est de vérifier que le rectangle dont le centre et les dimensions sont fournis en arguments ne se situe pas partiellement ou totalement en dehors de bornes de l'environnement. Puisque sa spécification ne nécessite pas l'introduction de nouvelles primitives abstraites ou de primitives de l'environnement, nous ne décrivons pas sa spécification dans ce chapitre. Les primitives d'ajout et de retrait d'entités dans l'environnement introduisent de plus trois primitives de perception à la signature des entités dans l'environnement, intitulées « estActive », « estPassive » et « estLabile ». Elles permettent de connaître la nature active, passive ou labile d'une entité, et leur spécification prend la forme présentée sur l'algorithme 25.

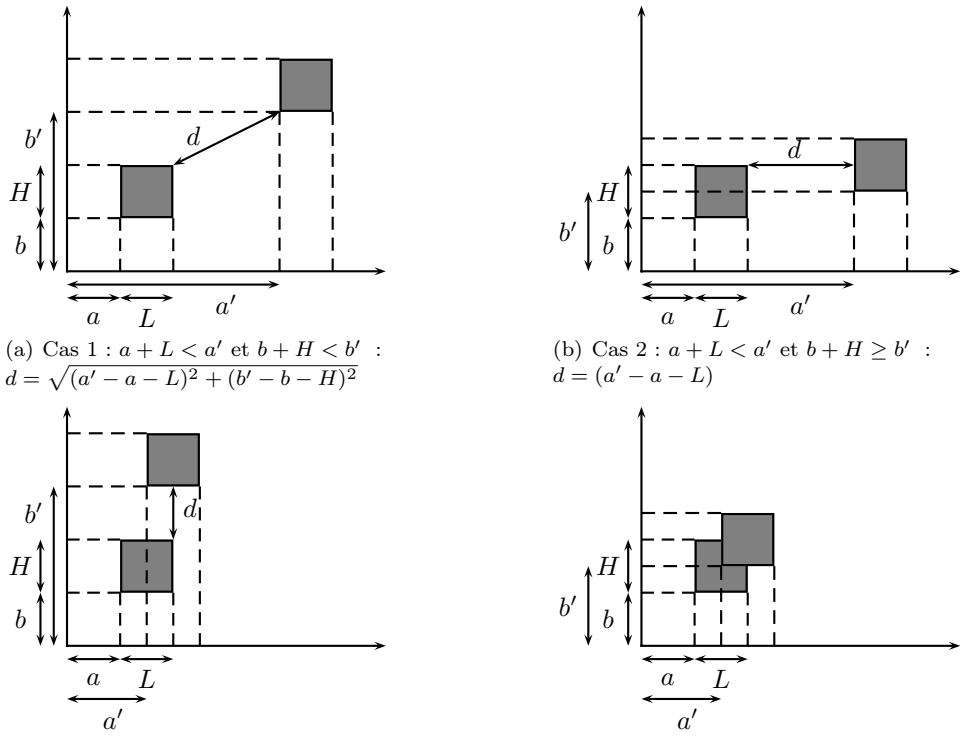
Algorithme 23 : Calcul de la distance séparant deux entités A et B, selon le principe décrit dans la figure A.2, où A (resp. B) a pour position (x_A, y_A) (resp. (x_B, y_B)) et a pour surface un rectangle centré sur cette position, dont les dimensions sont $L_A \times H_A$ (resp. $L_B \times H_B$).

distanceNonTorique(x_A,Y_A,L_A,H_A,x_B,Y_B,L_B,H_B)
début
  $a \Leftarrow min(x_A - \frac{L_A}{2}, x_B - \frac{L_B}{2})$;
  si $a == x_A - \frac{L_A}{2}$ **alors**
    $L \Leftarrow L_A$;
    $a' \Leftarrow x_B - \frac{L_B}{2}$;
  sinon
    $L \Leftarrow L_B$;
    $a' \Leftarrow x_A - \frac{L_A}{2}$;
  $b \Leftarrow min(y_A - \frac{H_A}{2}, y_B - \frac{H_B}{2})$;
  si $b == y_A - \frac{H_A}{2}$ **alors**
    $H \Leftarrow H_A$;
    $b' \Leftarrow y_B - \frac{H_B}{2}$;
  sinon
    $H \Leftarrow H_B$;
    $b' \Leftarrow y_A - \frac{H_A}{2}$;
  si $a + L < a'$ **alors**
    si $b + H < b'$ **alors**
      retourner $\sqrt{\left(a' - (a+L)\right)^2 + \left(b' - (b+H)\right)^2}$;
    sinon
      retourner $a' - (a + L)$;
  sinon
    si $b + H < b'$ **alors**
      retourner $b' - (b + H)$;
    sinon
      retourner 0;
fin

Algorithme 24 : Spécification des primitives de l'environnement permettant d'ajouter ou de retirer une entité de l'environnement.

ajouter(e_1,x,y)
début
  si ***environnement.****rectangleDansLesBornes(x,y,*e_1*.largeur,* e_1*.hauteur)* **alors**
    $e_1.definirAbscisse(x)$;
    $e_1.definirOrdonnee(y)$;
    $\mathbb{E} \Leftarrow \mathbb{E} \cup \{e_1\}$;
    si $e_1.estActive()$ **alors**
      $\mathbb{E}_{active} \Leftarrow \mathbb{E}_{active} \cup \{e_1\}$;
    si $e_1.estPassive()$ **alors**
      $\mathbb{E}_{labile} \Leftarrow \mathbb{E}_{passive} \cup \{e_1\}$;
    si $e_1.estLabile()$ **alors**
      $\mathbb{E}_{labile} \Leftarrow \mathbb{E}_{labile} \cup \{e_1\}$;
    retourner Vrai;
  sinon
    retourner Faux;
fin

retirer(e_1)
début
  $\mathbb{E} \Leftarrow \mathbb{E} \backslash \{e_1\}$;
  si $e_1.estActive()$ **alors**
    $\mathbb{E}_{active} \Leftarrow \mathbb{E}_{active} \backslash \{e_1\}$;
  si $e_1.estPassive()$ **alors**
    $\mathbb{E}_{labile} \Leftarrow \mathbb{E}_{passive} \backslash \{e_1\}$;
  si $e_1.estLabile()$ **alors**
    $\mathbb{E}_{labile} \Leftarrow \mathbb{E}_{labile} \backslash \{e_1\}$;
fin



(a) Cas 1 : $a + L < a'$ et $b + H < b'$: $d = \sqrt{(a' - a - L)^2 + (b' - b - H)^2}$

(b) Cas 2 : $a + L < a'$ et $b + H \geq b'$: $d = (a' - a - L)$

(c) Cas 3 : $a + L \geq a'$ et $b + H < b'$: $d = (b' - b - H)$

(d) Cas 4 : $a + L \geq a'$ et $b + H \geq b'$: $d = 0$

FIGURE A.2 – Principes du calcul de la distance entre deux entités, en fonction de leur position.

Algorithme 25 : Spécification des primitives de perception permettant de savoir si une entité est active, labile ou passive.

```
estActive()
début
    pour F ∈ 𝔽 ∪ {∅} faire
        si this.ligne(F) ≠ ∅ alors
            retourner Vrai;
    retourner Faux;
fin
```

```
estPassive()
début
    pour F ∈ 𝔽 faire
        si this.colonne(F) ≠ ∅ alors
            retourner Vrai;
    retourner Faux;
fin
```

```
estLabile()
début
    retourner this.U_ord ≠ ∅;
fin
```

A.4 Déplacement des entités

Nous faisons le choix de représenter les déplacements des entités dans l'environnement de deux manières. La première consiste à modifier directement la position d'une entité. La seconde consiste à déplacer l'entité selon une direction qui lui est propre, sur une certaine distance. Pour chacun de ces déplacements, deux primitives de l'environnement sont ajoutées. Elles permettent de vérifier qu'un tel déplacement est possible et d'effectuer le déplacement. Cela implique l'ajout à $primitives_{env}$ des primitives :

- $\langle peutDeplacerVers, Booleen, \{Entite, (Nombre\ a\ Virgule\ Flottante, Nombre\ a\ Virgule\ Flottante\}\rangle$, qui permet de vérifier qu'une entité peut être placée à une position particulière de l'environnement ;
- $\langle deplacerVers, \emptyset, \{Entite, Nombre\ a\ Virgule\ Flottante, Nombre\ a\ Virgule\ Flottante\}\rangle$, qui déplace une entité vers une position particulière de l'environnement ;
- $\langle peutAvancerVers, Booleen, \{Entite, Nombre\ a\ Virgule\ Flottante, Nombre\ a\ Virgule\ Flottante\}\rangle$, qui permet de vérifier que l'entité peut être placée dans une certaine distance, selon la direction fournie en argument. Cette direction est exprimée en radians dans le sens trigonométrique (*i.e.* sens anti-horaire), par rapport au vecteur $(0; 1)$;
- $\langle avancer, \emptyset, \{Entite, Nombre\ a\ Virgule\ Flottante\}\rangle$, qui permet de déplacer une entité vers l'avant, sur une certaine distance. La notion de « avant » est définie par la direction de l'entité dans l'environnement. Cette direction est exprimée en radians dans le sens trigonométrique (*i.e.* sens anti-horaire), par rapport au vecteur $(0; 1)$.

À ces primitives, nous ajoutons deux primitives $\langle tournerADroite, \emptyset, \{Entite, Nombre\ a\ Virgule\ Flottante\rangle$ et $\langle tournerADroite, \emptyset, \{Entite, Nombre\ a\ Virgule\ Flottante\rangle$, qui permettent aux entités de tourner sur elles-mêmes ainsi qu'une primitive $\langle faireFaceA, \emptyset, \{Entite, Nombre\ a\ Virgule\ Flottante\rangle$ qui permet de redéfinir la direction d'une entité dans l'environnement. Leur utilisation, conjointement à la primitive « avancerVers », permet de spécifier tout déplacement en fonction de la direction d'une entité.

La spécification de ces primitives figure dans l'algorithme 26. Elles introduisent :

- la primitive de l'environnement intitulée « rectangleDansLesBornes » qui est déjà décrite dans la section précédente ;
- trois primitives d'action intitulées « definirAbscisse », « definirOrdonnee » et « definirDirection » ;
- une primitive de perception intitulée « direction ».

Nous postulons que le comportement de ces primitives au sein des entités consiste à retourner ou modifier la valeur d'un attribut éponyme. Par conséquent, toute entité présente dans ce type d'environnement dispose de trois attributs dont l'identifiant est « abscisse », « ordonnee » et « direction » : $\forall \mathcal{F} \in \mathbb{F}, "abscisse" \in attributs(\mathcal{F}), "ordonnee" \in attributs(\mathcal{F})$ et $"direction" \in attributs(\mathcal{F})$. On peut aussi remarquer dans ces algorithmes l'apparition de fonctions « sinus » et « cosinus ». Elles sont considérées comme des fonctions arithmétiques de base et ne sont donc pas décrites comme des primitives de l'environnement.

A.5 Halo des entités

Dans ce chapitre, nous introduisons un halo particulier, où la perception est définie à l'aide d'une surface de l'environnement. Selon ce halo, une entité e_2 est perçue par une entité e_1 si l'intersection entre la surface de l'environnement perçue par e_1 et la surface de l'entité e_2 est non vide. La surface de l'environnement perçue par une entité dépend à la fois de sa position, mais aussi de sa direction. Par conséquent, le halo d'une entité est caractérisé par une surface référente, qui représente, dans un repère relatif à l'entité, la surface qu'elle perçoit si jamais sa direction est de 0 radians (*i.e.* si sa direction est le « nord » dans l'environnement). Nous considérons que la surface de perception référente est obtenue à l'aide de la primitive de perception $\langle surfaceDePerceptionReferente, Surface, \emptyset\rangle$. La surface de l'environnement perçue par une entité est obtenue par la rotation de la surface référente d'un angle égal à la direction de l'entité, puis en faisant passer la surface de perception d'un repère local à un repère global, par une translation de vecteur la position de cette entité (ce procédé est résumé sur la figure A.3).

Puisque l'environnement peut être torique, la spécification d'un tel halo suit un algorithme similaire

Algorithme 26 : Spécification des sept primitives de l'environnement permettant de déplacer les entités dans un environnement.

peutDeplacerVers(e_1,x,y)
début
| **retourner environnement.**rectangleDansLesBornes(x,y,$e_1.largeur()$,$e_1.hauteur()$);
fin

deplacerVers(e_1,x,y)
début
| $e_1.definirAbscisse(x)$;
| $e_1.definirOrdonnee(y)$;
fin

faireFaceA(e_1,α)
début
| $e_1.definirDirection(\alpha)$
fin

tournerAGauche(e_1,α)
début
| $e_1.definirDirection(e_1.direction() + \alpha)$
fin

tournerADroite(e_1,α)
début
| $e_1.definirDirection(e_1.direction() - \alpha)$
fin

peutAvancerVers(e_1,α,d)
début
| *% Calcul du point atteint par un déplacement de d unités dans la direction α par rapport au*
| *% vecteur $(0;1)$, exprimé dans le sens trigonométrique.*
| $x \Leftarrow e_1.abscisse() - d \times$ **sinus**(α);
| $y \Leftarrow e_1.ordonnee() + d \times$ **cosinus**(α);
| **retourner environnement.**rectangleDansLesBornes(x,y,$e_1.largeur()$,$e_1.hauteur()$);
fin

avancer(e_1,α,d)
début
| *% Calcul du point atteint par un déplacement de d unités dans la direction α par rapport au*
| *% vecteur $(0;1)$, exprimé dans le sens trigonométrique.*
| $x \Leftarrow e_1.abscisse() - d \times$ **sinus**$(e_1.direction())$;
| $y \Leftarrow e_1.ordonnee() + d \times$ **cosinus**$(e_1.direction())$;
| **retourner environnement.**deplacerVers(x,y);
fin

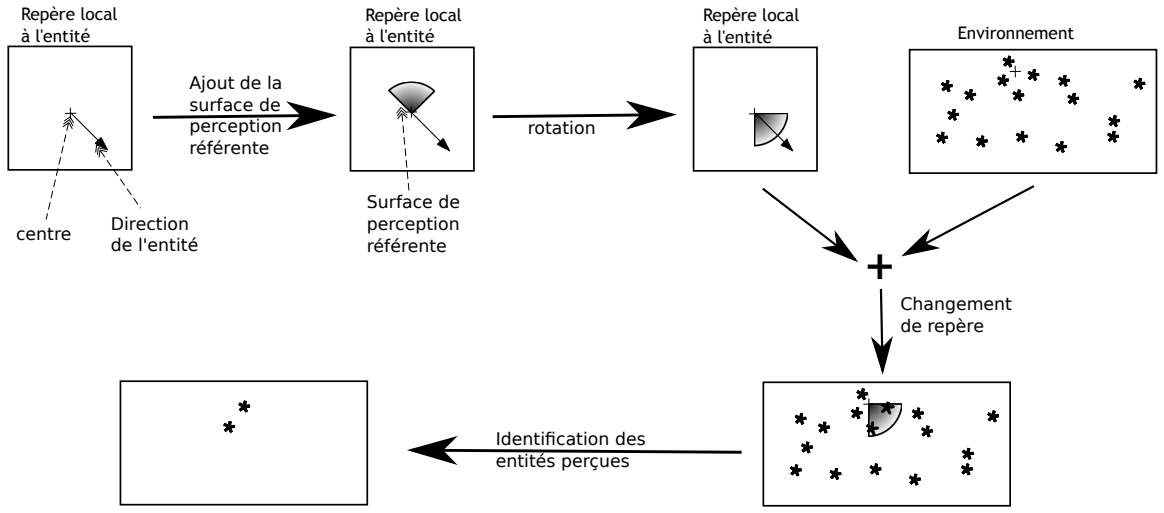


FIGURE A.3 – Procédé suivi pour identifier les entités perçues dans un environnement non-torique, à l'aide d'une surface de perception. La surface de perception référente d'une entité est décrite par la primitive de perception $\langle surfaceDePerceptionReferente, Surface, \emptyset \rangle$.

à celui du calcul de la distance : déterminer si une entité e_1 perçoit une entité e_2 revient à vérifier si e_1 perçoit directement e_2, ou si e_1 perçoit la translation de e_2 selon l'axe des abscisses et des ordonnées. La spécification du halo est alors illustrée par l'algorithme 27.

La signature des entités dans un tel halo contient les primitives abstraites dont l'identifiant est « distance », « abscisse », « ordonnee », « largeur » et « hauteur », dont la spécification est déjà fournie, ainsi qu'une primitive de perception $\langle surfaceDePerceptionReferente, Surface, \emptyset \rangle$, qui retourne la surface perçue de l'environnement si la direction de l'entité est 0 radians. Cette surface est exprimée dans un repère local à l'entité. La signature de l'environnement dans ce halo fait apparaître trois primitives de

Algorithme 27 : Spécification du halo d'une entité, comme présenté sur la figure A.3

```
halo(e1,e2)
début
    % Récupération de la surface de perception référente de l'entité e1 et passage en repère absolu.
    surfacePerçue ⇐ e1.surfaceDePerceptionReferente();
    environnement.rotation(surfacePerçue,e1.direction());
    environnement.translation(surfacePerçue,e1.abscisse(), e1.ordonnee());
    % La primitive « intersectionEstVide » vérifie si l'intersection de la surface représentée par
    % son premier argument avec le rectangle dont le centre
    % et les dimensions sont identifiées par ses quatre
    % derniers paramètres.
    si ¬environnement.intersectionEstVide(surfacePerçue, e1.abscisse(), e1.ordonnee(),
    e1.largeur(), e1.hauteur()) alors
        retourner Vrai;
    si environnement.toriqueEnAbscisses() alors
        pour x ∈ {−environnement.largeur(),environnement.largeur()} faire
            si ¬environnement.intersectionEstVide(surfacePerçue, e1.abscisse() + x, e1.ordonnee(),
            e1.largeur(), e1.hauteur()) alors
                retourner Vrai;
    si environnement.toriqueEnOrdonnees() alors
        pour y ∈ {−environnement.hauteur(),environnement.hauteur()} faire
            si ¬environnement.intersectionEstVide(surfacePerçue, e1.abscisse(), e1.ordonnee() + y,
            e1.largeur(), e1.hauteur()) alors
                retourner Vrai;
    si environnement.toriqueEnAbscisses() et environnement.toriqueEnOrdonnees() alors
        pour x ∈ {−environnement.largeur(),environnement.largeur()} faire
            pour y ∈ {−environnement.hauteur(),environnement.hauteur()} faire
                si ¬environnement.intersectionEstVide(surfacePerçue, e1.abscisse() +
                x, e1.ordonnee() + y, e1.largeur(), e1.hauteur()) alors
                    retourner Vrai;
    retourner Faux;
fin
```

l'environnement, dont l'objectif est de manipuler ou analyser des surfaces :

- La primitive $\langle rotation, \emptyset, \{Surface, Nombre\ a\ Virgule\ Flottante\} \rangle$ tourne la surface donnée en argument dans le sens trigonométrique, d'un angle égal au second argument de cette primitive ;
- La primitive $\langle translation, \emptyset, \{Surface, Nombre\ a\ Virgule\ Flottante, Nombre\ a\ Virgule\ Flottante\} \rangle$ a effectue la translation de la surface fournie par le premier argument de la primitive, suivant un vecteur défini par les second et troisième paramètres de la primitive ;
- La primitive $\langle intersectionEstVide, Boolean, \{Surface, (Nombre\ a\ Virgule\ Flottante)^4\} \rangle$ déter-

mine si l'intersection de la surface représentée par le premier argument et le rectangle dont le centre et les dimensions sont définis par les quatre derniers arguments est vide.

Nous ne décrivons pas les spécifications de ces primitives, ni de la structure de données utilisée pour représenter les surfaces dans ce manuscrit, puisque le principe de ces calculs n'est pas ambigu, contrairement au calcul de la distance entre deux surfaces.

A.6 Synthèse

Dans cette section, nous avons identifié et spécifié les différentes primitives de l'environnement, primitives abstraites et attributs permettant de représenter un environnement euclidien continu en deux dimensions. Cet environnement peut être torique ou non torique et les entités y sont représentées sous la forme de rectangles. Le tableau A.4 résume l'ensemble des primitives de l'environnement pouvant être utilisées par les concepteurs afin de décrire les interactions ayant lieu dans ce type d'environnement. Les primitives de l'environnement utilisées comme outils de calcul, par exemple la primitive dont l'identifiant est « distanceNonTorique », ne sont pas mentionnées dans cette synthèse.

Dans un tel environnement, nous proposons de représenter le halo d'une entité par une surface de l'environnement, qui détermine le voisinage d'une entité par son intersection avec la surface des entités de l'environnement. La signature des entités dans ce halo est définie sur le tableau A.5b. Plus généralement, la signature des entités dans l'environnement euclidien défini dans cette section est résumée dans le tableau A.5a.

« environnement » Environnement euclidien en deux dimensions				
Identifiant	Valeur retournée	Attribut		Description
		Identifiant	Type	
Signature des entités dans l'environnement	Entités dans Environnement euclidien en deux dimensions (voir figure A.5)			
distance	Nombre à Virgule Flottante	e_1	Entité	Retourne la distance existant entre deux entités de l'environnement
		e_2	Entité	
largeurEnvironnement	Nombre Entier	—	—	Retourne la largeur du rectangle constituant l'environnement
hauteurEnvironnement	Nombre Entier	—	—	Retourne la hauteur du rectangle constituant l'environnement
toriqueEnAbscisses	Booléen	—	—	Retourne vrai si l'environnement est un tore selon l'axe des abscisses
toriqueEnOrdonnees	Booléen	—	—	Retourne vrai si l'environnement est un tore selon l'axe des ordonnées
ajouter	Booléen	e	Entité	Ajoute une entité dans l'environnement à la position (x, y). Retourne Vrai si l'entité pouvait être placée à la position souhaitée et Faux si la surface de l'entité sort de l'environnement
		x	Nombre à Virgule Flottante	
		y	Nombre à Virgule Flottante	
retirer	—	e	Entité	Retire une entité de l'environnement
peutDeplacerVers	Booléen	e	Entité	Vérifie si l'entité peut être placée à la position (x, y).
		x	Nombre à Virgule Flottante	
		y	Nombre à Virgule Flottante	
deplacerVers	—	Identiques à *peutDeplacerVers*		Déplace l'entité e vers la position (x, y).
peutAvancerVers	Booléen	e	Entité	Vérifie si l'entité peut être placée à la position se situant à une distance de d dans la direction α (en radians, exprimé par rapport au vecteur $(0, 1)$, dans le sens anti-horaire) par rapport à sa position actuelle.
		α	Nombre à Virgule Flottante	
		d	Nombre à Virgule Flottante	
avancer	—	Identiques à *peutAvancerVers*		Déplace l'entité e à la position se situant à une distance de d devant elle (*i.e.* dans sa direction).
faireFaceA	—	e	Entité	Définit la direction d'une entité comme égale à α, exprimé en radians par rapport au vecteur $(0, 1)$, dans le sens anti-horaire.
		α	Nombre à Virgule Flottante	
tournerAGauche	—	Identiques à *faireFaceA*		Tourne l'entité vers la gauche (*i.e.* dans le sens horaire) d'un angle égal à α (en radians).
tournerADroite	—	Identiques à *faireFaceA*		Tourne l'entité vers la droite (*i.e.* dans le sens anti-horaire) d'un angle égal à α (en radians).

FIGURE A.4 – Synthèse des différentes primitives de l'environnement fournies par l'environnement décrit dans ce chapitre. Nous ne mentionnons ici que les primitives de l'environnement susceptibles d'être utilisées lors de la modélisation d'un phénomène. Les primitives de l'environnement utilisées comme outil par d'autres primitives de l'environnement ne sont pas mentionnées.

« signature d'entité » Entités dans Environnement euclidien en deux dimensions				
Identifiant	Valeur retournée	Attribut		Description
		Identifiant	Type	
estActive	Booléen	—	—	Retourne vrai si la ligne de la matrice d'interaction raffinée associée à l'entité est vide
estLabile	Booléen	—	—	Retourne vrai si la cellule de la matrice de mise à jour ordonnée associée à l'entité est vide
abscisse	Nombre à Virgule Flottante	—	—	Retourne l'abscisse de la position de l'entité dans l'environnement
ordonnee	Nombre à Virgule Flottante	—	—	Retourne l'ordonnée de la position de l'entité dans l'environnement
largeur	Nombre à Virgule Flottante	—	—	Retourne la largeur de la surface rectangulaire de l'entité
hauteur	Nombre à Virgule Flottante	—	—	Retourne la hauteur de la surface rectangulaire de l'entité
direction	Nombre à Virgule Flottante	—	—	Retourne la direction de l'entité dans l'environnement, exprimée en radians par rapport au vecteur $(0, 1)$, dans le sens anti-horaire
definirAbscisse	—	x	Nombre à Virgule Flottante	Change l'abscisse de la position de l'entité dans l'environnement, pour la valeur x
definirOrdonnee	—	y	Nombre à Virgule Flottante	Change l'abscisse de la position de l'entité dans l'environnement, pour la valeur y
definirDirection	—	x	Nombre à Virgule Flottante	Change la direction de l'entité dans l'environnement, pour la valeur α, exprimée en radians par rapport au vecteur $(0, 1)$, dans le sens anti-horaire

(a) Signature des entités dans l'environnement

« signature d'entité » Entités dans le halo d'un Environnement euclidien en deux dimensions				
Identifiant	Valeur retournée	Attribut		Description
		Identifiant	Type	
surfacePerception-Referente	Surface	—	—	Retourne la surface de perception référente de l'entité, *i.e.* la surface de perception, exprimée dans un repère centré sur l'entité, dans le cas où la direction de l'entité est 0

(b) Signature des entités dans leur halo

FIGURE A.5 – (a) Signature des entités dans l'environnement euclidien défini dans l'annexe A et (b) signature des entités dans le halo défini pour cet environnement.

Bibliographie

[AAEC96] Robert Axtell, Robert Axelrod, Joshua M. Epstein, and Michael D. Cohen. Aligning simulation models : A case study and results. *Computational and Mathematical Organization Theory*, 1 :123–141, 1996.

[ABB+04] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychol Rev*, 111(4) :1036–1060, October 2004.

[And74] Jock R. Anderson. Simulation : Methodology and application in agricultural economics. *Review of Marketing and Agricultural Economics*, 42(01), March 1974.

[AS07] Francesco Amigoni and Viola Schiaffonati. Multiagent-based simulation in biology : A critical analysis. In *Proceedings of Model-Based Reasoning in Science, Technology, and Medicine conference (MBR'06)*, volume 64 of *Studies in Computational Intelligence*, Guangzhou, P.R. China, 2007.

[ATL09] ATLAS group. ATLAS Transformation Language. Site de référence : `http://www.sciences.univ-nantes.fr/lina/atl/`, 2009. Dernier accès le 12/07/2009.

[Axe97] Robert Axelrod. Advancing the art of simulation in the social sciences. *Simulating Social Phenomena, Lecture Notes in Economics and Mathematical Systems*, 456 :21–40, 1997.

[BB90] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Proceedings of the 17th symposium on Principles of programming languages (POPL '90)*, pages 81–94, New York, NY, USA, 1990. ACM.

[BCGP05] C. Bernon, V. Camps, M.-P. Gleizes, and G. Picard. *Engineering Self-Adaptive Multi-Agent Systems : the ADELFE Methodology*, chapter 7, pages 172–202. Idea Group Publishing, 2005.

[BF97] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2) :281–300, 1997.

[BH06] Rafael H. Bordini and Jomi F. Hübner. BDI agent programming in AgentSpeak using Jason. In *Computational Logic in Multi-Agent Systems*, pages 143–164, 2006.

[BHRH00] Paolo Busetta, Nicholas Howden, Ralph Rönnquist, and Andrew Hodgson. Structuring BDI agents in functional clusters. In *Proceedings of the 6th International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL'99)*, pages 277–289, London, UK, 2000. Springer.

[BM97] Nouredine Bensaid and Philippe Mathieu. Magique : A hybrid and hierarchical multi-agent architecture model. In *Proceedings of the second International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi Agent Technology (PAAM'97)*, pages 145–155, 1997.

[BM07] Jean-Pierre Briot and Thomas Meurisse. An experience in using components to construct and compose agent behaviors for agent-based simulation. In *Proceedings of the International Modeling and Simulation Multiconference (IMSM'07)*, pages 207–212. The Society for Modeling & Simulation International (SCS), 2007.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos : An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3) :203–236, May 2004.

[BPL05] Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Jadex : A bdi agent system combining middleware and reasoning. In M. Klusch R. Unland, M. Calisti, editor, *Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168. Birkhäuser-Verlag, September 2005.

[BPR99] F. Bellifemine, A. Poggi, and G. Rimassa. JADE – a FIPA-compliant agent framework. In *Proceedings of the fourth International Conference on the Practical Application of Intelligent Agents and Multi Agent Technology (PAAM'99)*, pages 97–108, London, April 1999.

[Bro86] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1) :14–23, Mar 1986.

[CCM08] Nicola Cannata, Flavio Corradini, and Emanuela Merelli. Multiagent modelling and simulation of carbohydrate oxidation in cell. *International Journal of Modelling, Identification and Control (IJMIC)*, 3(1) :17–28, 2008.

[Che56] Michel-Eugène Chevreul. *Lettres adressées à M. Villemain sur la méthode en général et sur la définition du mot FAIT, relativement aux sciences, aux lettres, aux beaux-arts, etc.* Garnier Frères, 1856.

[Col90] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7), juillet 1990.

[Cos05] Massimo Cossentino. *From Requirements to Code with PASSI Methodology*, chapter 4, pages 79–106. Idea Group Publishing, 2005.

[COST03] Jason Cornwell, Kevin O'Brien, Barry Silverman, and Jozseph Toth. Affordance theory for improving the rapid generation, composability, and reusability of synthetic agents and objects. In *Proceedings of the 12^{t}h Conference on Behavior Representation in Modeling and Simulation (BRIMS, formerly CGF)*, May 2003.

[DDG09] Remy Courdier Daniel David, Denis Payet and Yassine Gangat. XELOC : un support générique pour la configuration et l'initialisation de systèmes multi-agents. In Zahia GUESSOUM et Salima HASSAS, editor, *Actes des 17e Journées Francophones sur les Systèmes Multi-Agents (JFSMA'2009)*, pages 233–236. Cépaduès, 2009.

[Dem95] Yves Demazeau. From interactions to collective behaviour in agent-based systems. In *Proceedings of the 1st. European Conference on Cognitive Science. Saint-Malo*, pages 117–132, 1995.

[Des06] Gireg Desmeulles. *Réification des interactions pour l'expérience in virtuo de systèmes biologiques multi-modèles*. PhD thesis, Université de Bretagne Occidentale, Décembre 2006.

[DF92] A. Drogoul and J. Ferber. Multi-agent simulation as a tool for modeling societies : Application to social differentiation in ant colonies. In *Proceedings of MAAMAW'92*, 1992.

[DMR05] Damien Devigne, Philippe Mathieu, and Jean-Christophe Routier. Interaction-based approach for game agents. In Yuri Merkuryev, Richard Zobel, and Eugene Kerckhoffs, editors, *Proceedings of the 19th European Conference on Modelling and Simulation (ECMS'05)*, pages 705–714. TODO, 2005.

[Dro93] Alexis Drogoul. *De La Simulation Multi-Agent A La Résolution Collective de Problèmes - Une Étude De l'Émergence De Structures D'Organisation Dans Les Systèmes Multi-Agents*. PhD thesis, Universite Paris VI, Novembre 1993.

[DTH05] Takuo Doi, Yasuyuki Tahara, and Shinichi Honiden. IOM/T : an interaction description language for multi-agent systems. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems (AAMAS'05)*, pages 778–785, Utrecht, The Netherlands, 2005. ACM.

[EA96] Joshua M. Epstein and Robert Axtell. *Growing artificial societies : social science from the bottom up*. The Brookings Institution, Washington, DC, USA, 1996.

[Edm05] Bruce Edmonds. Simulation and complexity - how they can relate. In *Virtual Worlds of Precision - computer-based simulations in the sciences and social sciences*, pages 5–32. Lit Verlag, feldmann, valerie and mühlfeld, katrin edition, 2005.

[EH03] Bruce Edmonds and David Hales. Replication, replication and replication : Some hard lessons from model alignment. *J. Artificial Societies and Social Simulation*, 6(4), 2003.

[Fer92] Innes A. Ferguson. Touring machines : Autonomous agents with attitudes. *Computer*, 25(5) :51–55, 1992.

[Fer99] Jacques Ferber. *Multi-Agent System : An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[FG98] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings. International Conference on Multi Agent Systems, 1998.*, pages 128–135, Jul 1998.

[FHK+97] Pat Ferguson, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya. Results of applying the personal software process. *Computer*, 30 :24–31, 1997.

[FIP10a] FIPA. Fipa contract net interaction protocol specification. Site de référence : `http://www.fipa.org/specs/fipa00029/SC00029H.pdf`, 2010. Dernier accès le 12/03/2010.

[FIP10b] FIPA. Foundation for intelligent physical agents (FIPA) specification. Site de référence : `http://www.fipa.org/`, 2010. Dernier accès le 12/03/2010.

[Fis94] Paul A. Fishwick. Simulation model design. In *Proceedings of the 26th conference on Winter simulation (WSC '94)*, pages 173–175, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[FM96] Jacques Ferber and Jean-Pierre Müller. Influences and reaction : a model of situated multi-agent systems. In Mario Tokoro, editor, *Proceedings of the Second International Conference on Multiagent Systems (ICMAS'96)*, pages 72–79, 1996.

[FMP95] Klaus Fischer, Jörg P. Müller, and Markus Pischel. Unifying control in a layered agent architecture. In *In IJCAI95, Agent Theory, Architecture and Language Workshop 95*, 1995.

[Fou10] Python Software Foundation. Python programming language - official website. Site de référence : `http://www.python.org/`, 2010. Dernier accès le 28/07/2010.

[fra10] Académie française. Dictionnaire de l'académie française, neuvième édition. Site de référence : `http://www.academie-francaise.fr/dictionnaire/`, 2010. Dernier accès le 15/03/2010.

[Fre98] Michael Freed. Managing multiple tasks in complex, dynamic environments. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence (AAAI '98/IAAI '98)*, pages 921–927, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[GBB+06] Volker Grimm, Uta Berger, Finn Bastiansen, Sigrunn Eliassen, Vincent Ginot, Jarl Giske, John Goss-Custard, Tamara Grand, Simone K. Heinz, Geir Huse, Andreas Huth, Jane U. Jepsen, Christian Jørgensen, Wolf M. Mooij, Birgit Müller, Guy Pe'er, Cyril Piou, Steven F. Railsback, Andrew M. Robbins, Martha M. Robbins, Eua Rossmanith, Nadja Rüger, Espen Strand, Sami Souissi, Richard A. Stillman, Rune Vabø, Ute Visser, and Donald L. Deangelis. A standard protocol for describing individual-based and agent-based models. *Ecological modelling*, 198(1-2) :115–126, 2006.

[GFM01] Olivier Gutknecht, Jacques Ferber, and Fabien Michel. Integrating tools and infrastructures for generic multi-agent systems. In Jörg P. Müller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, *Proceedings of the 5^{th} International Conference on Autonomous Agents (AGENTS'01)*, Montreal, Canada, 2001. ACM.

[GGB08] Emilia Garcia, Adriana Giret, and Vicente Botti. Towards an evaluation framework for mas software engineering. In *Proceedings of the 11th Pacific Rim International Conference on Multi-Agents (PRIMA '08)*, pages 197–205, Berlin, Heidelberg, 2008. Springer-Verlag.

[Gib77] J. J. Gibson. *The Theory of Affordances*. Lawrence Erlbaum, 1977.

[GII+09] José Manuel Galán, Luis R. Izquierdo, Segismundo S. Izquierdo, José Ignacio Santos, Ricardo del Olmo, Adolfo López-Paredes, and Bruce Edmonds. Errors and artefacts in agent-based modelling. *Journal of Artificial Societies and Social Simulation*, 12(1) :1, 2009.

[Gil76] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4) :403–434, December 1976.

[GKMP08] François Gaillard, Yoann Kubera, Philippe Mathieu, and Sébastien Picault. A reverse engineering form for multi agent systems. In Alexander Artikis, Gauthier Picard, and Laurent Vercouter, editors, *Proceedings of the 9th International Workshop Engineering Societies in the Agents World (ESAW'2008)*, volume 5485 of *Lecture Notes on Artificial Intelligence*, pages 137–153. Springer, 2008.

[GKMP10] François Gaillard, Yoann Kubera, Philippe Mathieu, and Sébastien Picault. Une forme de rétro ingénierie pour systèmes multi agents : explorer l'espace des simulations. *Revue d'Intelligence Artificielle*, 2010. À paraître.

[GL87] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87)*, pages 677–682, Seattle, WA, 1987.

[GM02] Jean-Louis Giavitto and Olivier Michel. Data structure as topological spaces. In *Proceedings of the Third International Conference on Unconventional Models of Computation (UMC '02)*, pages 137–150, London, UK, 2002. Springer-Verlag.

[GMGSFF09] Iván García-Magariño, Jorge J. Gómez-Sanz, and Rubén Fuentes-Fernández. Model transformations for improving multi-agent systems development in ingenias. In *The 10th International Workshop on Agent-Oriented Software Engineering (AOSE'09)*, 2009. May 11, 2009, Budapest Hungary, (with the annex).

[GODR+08] Juan C. Garcia-Ojeda, Scott A. DeLoach, Robby, Walamitien H. Oyenan, and Jorge Valenzuela. O-MaSE : A customizable approach to developing multiagent development processes. In *Proceedings of the 8th International Workshop on Agent Oriented Software Engineering (AOSE 2007)*, pages 1–15, 2008.

[GODR09] Juan C. Garcia-Ojeda, Scott A. DeLoach, and Robby. agentTool III : from process definition to code generation. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '09)*, pages 1393–1394, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[GRHT07] Laurent Gaubert, Pascal Redou, Fabrice Harrouet, and Jacques Tisseau. A first mathematical model of brood sorting by ants : Functional self-organization without swarm-intelligence. *Ecological Complexity*, December 2007.

[Gru10] Grupo de Investigación en Agentes Software : Ingeniería y Aplicaciones (GARCIA). IDK : Ingenias Development Kit. Site de référence : `http://grasia.fdi.ucm.es/main/node/127`, 2010. Dernier accès le 17/02/2010.

[GT05] Nigel Gilbert and Klaus G. Troitzsch. *Simulation for the Social Scientist*. Open University Press, Maidenhead and New York, 2 edition, 2005.

[HTW04] A. Helsinger, M. Thome, and T. Wright. Cougaar : a scalable, distributed multi-agent architecture. In *IEEE International Conference on Systems, Man and Cybernetics 2004*, volume 2, pages 1910–1917, Oct. 2004.

[Hub99] Marcus J. Huber. JAM : a BDI-theoretic mobile agent architecture. In *Proceedings of the third annual conference on Autonomous Agents (AGENTS'99)*, pages 236–243, New York, NY, USA, 1999. ACM.

[Hum91] P. Humphreys. Computer simulation. In & L. Wessels A. Fine, M. Forbes, editor, *PSA 1990*, volume 2, pages 497–506, East Lansing, USA, 1991.

[Jen99] Nicholas R. Jennings. Agent-based computing : Promise and perils. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 1429–1436, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[JFB05] F. Michel J. Ferber and J. Baez. Agre : Integrating environments with organizations. In *Proceedings of Environments for Multi-Agent Systems Conference (E4MAS'05)*, volume 3374, pages 48–56, 2005.

[J.L25] Alfred J.Lotka. *Elements of Physical Biology*. Williams and Wilkins Company, 1925.

[KHF06] F. Klügl, R. Herrler, and M. Fehler. SeSAm : implementation of agent-based simulation using visual programming. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems (AAMAS '06)*, pages 1439–1440, New York, NY, USA, 2006. ACM.

[KMP07] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. La complexité dans les simulations multi-agents. In Valérie Camps and Philippe Mathieu, editors, *Actes des 15e Journées Francophones sur les Systèmes Multi-Agents (JFSMA'2007)*, pages 139–148. Cépaduès, 2007.

[KMP08a] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. Formalisation et implémentation des interactions pour la simulation centrée individu. *L'objet*, 14(1-2) :9–33, Janvier-Juin 2008. Numéro spécial Architectures Logicielles.

[KMP08b] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. Interaction biases in multi-agent simulations : An experimental study. In Alexander Artikis, Gauthier Picard, and Laurent Vercouter, editors, *Proceedings of the 9th International Workshop Engineering Societies in the Agents World (ESAW'2008)*, volume 5485 of *Lecture Notes on Artificial Intelligence*, pages 229–247. Springer, 2008.

[KMP08c] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. Interaction-oriented agent simulations : From theory to implementation. In Malik Ghallab, Constantine Spyropoulos, Nikos Fakotakis, and Nikos Avouris, editors, *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, pages 383–387. IOS Press, 2008.

[KMP08d] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. Interaction selection ambiguities in multi-agent systems. In Chengqi Zhang, Nick Cercone, and Lakhmi Jain, editors, *Proceedings of IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'08)*, pages 75–78, 2008.

[KMP09] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. How to avoid biases in reactive simulations. In Yves Demazeau, Juan Pavòn, Juan Corchado, and Javier Bajo, editors, *Proceedings of the 7th International conference on Practical Applications of Agents and Multi-Agents Systems (PAAMS'2009)*, volume 55 of *Practical Advances in Intelligent and soft computing*, pages 100–109. Springer, 2009.

[KMP10] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. Everything can be agent ! In *Proceedings of the ninth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2010)*, 2010.

[LAM10] LAMP group. Scala. Site de référence : `http://www.scala-lang.org/`, 2010. Dernier accès le 28/07/2010.

[LM91] A.M. Law and M.G. McComas. Secrets of successful simulation studies. In *Simulation Conference, 1991. Proceedings., Winter*, pages 21–27, Dec 1991.

[LNR87] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar : an architecture for general intelligence. *Artificial Intelligence*, 33(1) :1–64, 1987.

[Lor84] K Lorenz. *Les Fondements de l'Ethologie*. Flammarion, Paris, 1984.

[LP00] B.G. Lawson and S. Park. Asynchronous time evolution in an artificial society mode. *Journal of Artificial Societies and Social Simulation*, 2000.

[MBLA96] Nelson Minar, Rogert Burkhart, Chris Langton, and Manor Askenazi. The swarm simulation system : A toolkit for building multi-agent simulations. Working papers, Santa Fe Institute, June 1996.

[MEU04] Thomas MEURISSE. *Simulation multi-agent : du modèle à l'opérationnalisation*. PhD thesis, Universite de Paris 6, Juillet 2004.

[MGF03] Fabien Michel, Abdelkader Gouaïch, and Jacques Ferber. Weak interaction and strong interaction in agent based simulations. In David Hales, Bruce Edmonds, Emma Norling, and Juliette Rouchier, editors, *Multi-Agent-Based Simulation III*, volume 2927 of *Lecture Notes in Computer Science*, pages 43–56. Springer Berlin / Heidelberg, 2003.

[Mic00] F. Michel. Une approche méthodologique pour l'analyse et la conception de simulateur multi-agents. In *Cinquièmes rencontres des Jeunes Chercheurs en Intelligence Artificielle (RJCIA'00)*, Lyon, France, 2000.

[Mic04] Fabien Michel. *Formalisme, outils et éléments méthodologiques pour la modélisation et la simulation multi-agents.* PhD thesis, Universite Montpellier II, December 2004.

[Mic07] Fabien Michel. Le modèle IRM4S : De l'utilisation des notions d'influence et de réaction pour la simulation de systèmes multi-agents. *Revue d'intelligence artificielle*, 21(5-6) :757–779, 2007.

[MP94] J. P. Müller and M. Pischel. Modelling interacting agents in dynamic environments. In *Proceedings of ECAI'94*, Amsterdam, The Netherlands, 1994.

[MP05] Philippe Mathieu and Sébastien Picault. Towards an interaction-based design of behaviors. In *Proceedings of the The Third European Workshop on Multi-Agent Systems (EUMAS'2005)*, 2005.

[MP06] Philippe Mathieu and Sébastien Picault. Vers une représentation des comportements centrée interactions. In *Actes du 15e congrès francophone AFRIF-AFIA Reconnaissance des Formes et Intelligence Artificielle (RFIA'2006)*, 2006.

[MPR07] Philippe Mathieu, Sébastien Picault, and Jean-Christophe Routier. Donner corps aux interactions. In *Actes des 4e Journées Francophones sur les Modèles Formels de l'Interaction (MFI'07)*, pages 333–340. Université de Paris Dauphine, 2007.

[MRU01] Philippe Mathieu, Jean-Christophe Routier, and Pascal Urro. Un modèle de simulation agent basé sur les interactions. In *Actes des Premières Journées Francophones sur les Modèles Formels de l'Interaction (MFI'01)*, pages 407–417, Toulouse,France, 2001.

[MVDJ09] G. Morvan, A. Veremme, D. Dupont, and D. Jolly. Stratégies d'observation de simulations orientées agent. In Zahia Guessoum et Salima Hassas, editor, *Actes des 17e Journées Francophones sur les Systèmes Multi-Agents (JFSMA'2009)*, pages 233–236. Cépaduès, 2009.

[NCLMA99] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. SHOP : Simple hHierarchical Ordered Planner. In *Proceedings of the 16th international joint conference on Artificial intelligence (IJCAI'99)*, pages 968–973, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[NDA08] Ngoc Doanh Nguyen, Alexis Drogoul, and Pierre Auger. Methodological steps and issues when deriving individual based-models from equation-based models : A case study in population dynamics. In *Proceedings of the 11th Pacific Rim International Conference on Multi-Agents (PRIMA '08)*, pages 295–306, Berlin, Heidelberg, 2008. Springer-Verlag.

[NM09] Cynthia Nikolai and Gregory Madey. Tools of the trade : A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation*, 12(2) :2, 2009.

[Nor88] Donald A. Norman. *The Psychology of Everyday Things.* Basic Books, 1988.

[NTCO07] M. North, E. Tatara, N. Collier, and J. Ozik. Visual agent-based model development with repast simphony. In *Proceedings of the Agent 2007 Conference on Complex Interaction and Social Emergence*, pages 173–192, November 2007.

[OPB00] J. Odell, H.V.D. Parunak, and B Bauer. Extending uml for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop (AOIS'00)*, pages 3–17, 2000.

[ORV08] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3) :432–456, 2008.

[Pap09] Michael Papasimeon. *Modelling Agent-Environment Interaction in Multi-Agent Simulations with Affordances.* PhD thesis, The University of Melbourne, Victoria 3010 Australia, January 2009.

[Par01] Miles Parker. What is Ascape and why should you care ? *Journal of Artificial Societies and Social Simulation*, 4 :1, 2001.

[PCSB07] Sébastien Picault, Florence Corellou, Christian Schwartz, and François-Yves Bouget. Simulation multi-agent de réseaux génétiques : les rythmes circadiens d'Ostreococcus tauri. In Valérie Camps and Philippe Mathieu, editors, *Actes des 15e Journées Francophones sur les Systèmes Multi-Agents (JFSMA'2007)*, pages 149–158. Cépaduès, 2007.

[PGS03] Juan Pavón and Jorge J. Gómez-Sanz. Agent oriented software engineering with INGENIAS. In *Proceedings of the third International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 2003)*, volume 2691, pages 394–403, Prague, Czech Republic, June 2003. Springer-Verlag.

[PMK10] Sébastien Picault, Philippe Mathieu, and Yoann Kubera. PADAWAN, un modèle multi-échelles pour la simulation orientée interactions. In Octobre, editor, *Actes des 18e Journées Francophones sur les Systèmes Multi-Agents (JFSMA'2010)*, Mahdia (Tunisie), 2010. Cépaduès.

[PMR+05] D. Payet, J.M. Medoc, T. Ralambondrainy, F. Guerrin, and R. Courdier. Outils d'observation et d'analyse de simulations multi-agents : l'expérience de la plate-forme Geamas/Biomas. In *Conference on Multi-agent modelling for environmental management*, Bourg Saint Maurice - Les Arcs, France, 2005.

[PTW05] Lin Padgham, John Thangarajah, and Michael Winikoff. Tool support for agent development using the Prometheus methodology. In *Proceedings of the Fifth International Conference on Quality Software (QSIC '05)*, pages 383–388, Washington, DC, USA, 2005. IEEE Computer Society.

[Pŏ00] Gheorghe Păun. Computing with membranes. *J. Comput. Syst. Sci.*, 61(1) :108–143, 2000.

[PW84] William Poundstone and Robert T. Wainwright. *The Recursive Universe ; Cosmic Complexity and the Limits of Scientific Knowledge.* William Morrow & Co., Inc., New York, NY, USA, 1984.

[Que02] Ronan Querrec. *Les Systèmes Multi-Agents pour les Environnements Virtuels de Formation – Application à la sécurité civile.* PhD thesis, Université de Bretagne Occidentale, Octobre 2002.

[RCP06] Tiana Ralambondrainy, Remy Courdier, and Denis Payet. An ontology for observation of multiagent based simulation. In *Proceedings of the 2006 IEEE/WIC/ACM international conference on Web Intelligence and Intelligent Agent Technology (WI-IAT'06)*, pages 351–354, Washington, DC, USA, 2006. IEEE Computer Society.

[Res97] M. Resnick. *Turtles, Termites, and Traffic Jams : Explorations in Massively Parallel Microworlds.* The MIT Press, Cambridge, MA, 1997.

[Rey02] Craig Reynolds. Opensteer. Site de référence : `http://opensteer.sourceforge.net/`, 2002. Dernier accès le 10/02/2010.

[RG91] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc., 1991.

[Rob06] S. Robinson. Conceptual modeling for simulation : Issues and research requirements. In *Proceedings of the Winter Simulation Conference 2006 (WSC 06)*, pages 792–800, Dec. 2006.

[Sar98] Robert G. Sargent. Verification and validation of simulation models. In *Proceedings of the 30th Winter Simulation Conference (WSC '98)*, pages 121–130, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[Sch71] Thomas C. Schelling. Dynamic models of segregation. *Journal of Mathematical Sociology*, 1 :143–186, 1971.

[Sci10] Cap Sciences. Clim'way. Site de référence : `http://climcity.cap-sciences.net/`, 2010. Dernier accès le 29/01/2010.

[SE 09] SE Division at Fondazione Bruno Kessler. TAOM4E : Tool for Agent Oriented Modeling. Site de Référence : `http://sra.itc.it/tools/taom4e/`, 2009. Dernier accès le 12/07/2009.

[SFT09] Tiberiu Stratulat, Jacques Ferber, and John Tranier. MASQ : towards an integral approach to interaction. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '09)*, pages 813–820, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[Sha76] Robert E. Shannon. Simulation modeling and methodology. In *Proceedings of the '76 Bicentennial conference on Winter simulation (WSC '76)*, pages 9–15. Winter Simulation Conference, 1976.

[Sha98] Robert E. Shannon. Introduction to the art and science of simulation. In *Proceedings of the 30th Winter Simulation Conference (WSC '98)*, pages 7–14, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[SKM02] B. Stroustrup, A. Koenig, , and B. Moo. The C++ programming language (2nd ed.). In *Encyclopedia of Software Engineering*, volume 2. Addison-Wesley, 2002.

[SPGS06] Candelaria Sansores, Juan Pavòn, and Jorge Gómez-Sanz. Visual modeling for complex agent-based simulation systems. In *Proceedings of the Multi-Agent-Based Simulation VI international workshop*, pages 174–189, 2006.

[SU01] B. Schattenberg and A.M. Uhrmacher. Planning agents in james. *Proceedings of the IEEE*, 89(2) :158–173, Feb 2001.

[Ter98] Pietro Terna. Simulation tools for social scientists : Building agent based models with swarm. *Journal of Artificial Societies and Social Simulation*, 1(2), 1998.

[Tri09] Triskell Team. Kermeta : Triskell Metamodeling Kernel. Site de référence : `http://www.kermeta.org/`, 2009. Dernier accès le 12/07/2009.

[VCP+95] Manuela Veloso, Jaime Carbonell, Alicia Perez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning : The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7 :81–120, 1995.

[Vol28] Vito Volterra. Variations and Fluctuations of the Number of Individuals in Animal Species living together. *ICES J. Mar. Sci.*, 3(1) :3–51, 1928.

[Wal10] Larry Wall. The Perl programming language. Site de référence : `http://www.perl.org/`, 2010. Dernier accès le 26/07/2010.

[WC99] Uri Wilenski and Center for Connected Learning and Computer-Based Modeling. Netlogo. Site de référence : `http://ccl.northwestern.edu/netlogo/`, 1999. Dernier accès le 29/01/2010.

[WH03] Danny Weyns and Tom Holvoet. Model for simultaneous actions in situated multi-agent systems. In *Proceedings of MATES 2003*, Erfurt, Germany, 2003.

[WJ95] M. J. Wooldridge and N. R. Jennings. Intelligent agents : Theory and practice. *The Knowledge Engineering Review*, 10(2) :115–152, 1995.

[Woo01] Michael Woolridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[WP04] Michael Winikoff and Lin Padgham. *Developing Intelligent Agent Systems : A Practical Guide*. Halsted Press, New York, NY, USA, 2004.

[ZJW03] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems : The Gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3) :317–370, 2003.

[ZKP00] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation, 2nd Edition*. Academic Press, Inc., Orlando, FL, USA, 2000.

[Öre87] T.I. Ören. *Simulation : Taxonomy*, pages 4411–4414. Systems and Control Encyclopedia. Pergamon Press, Boston, 1987.

Résumé

Les simulations multi-agents permettent de reproduire un phénomène en modélisant intuitivement son fonctionnement au niveau microscopique. Ce fonctionnement est décrit par le comportement d'entités autonomes qui agissent dans un environnement commun. Toutefois, les approches actuelles restreignent les interactions à des effets de bord ou ne fournissent aucune méthodologie réifiant la notion d'interaction. La conception de simulations contenant un grand nombre d'agents interagissant de manière variée s'en trouve complexifiée.

Nous soutenons que pour faciliter la conception des simulations, il est préférable de considérer que *toute entité est concrétisée par un agent* et *tout comportement par une interaction*. Le moteur de simulation doit de plus être clairement séparé des agents et interactions, de sorte que *tout le système multi-agents soit régi par le même algorithme de simulation*. Une telle approche procure de nombreux avantages tels que l'automatisation de l'implémentation, la réutilisabilité des interactions ou la conception graduelle du modèle du phénomène.

En nous fondant sur ces principes, nous avons développé une *approche centrée sur les interactions* (IODA) composée d'une pyramide d'outils : un *modèle formel* décrivant agents et interactions, un ensemble d'algorithmes de simulation unifiant le traitement de l'activité des agents et de la sélection d'actions et une *méthodologie* permettant de spécifier graduellement un modèle. Nous confirmons la faisabilité de cette approche par *une plateforme de simulation paramétrable* (JEDI) fidèle au modèle formel et un environnement de développement intégré (JEDI-BUILDER) qui automatise le passage du modèle IODA au code JEDI.

Nous montrons ainsi que la concrétisation logicielle des interactions a conduit à une unification du concept d'agent et à une simplification du processus de conception de simulations.

Mots-clés: Simulation Multi-Agents, Interaction, Modélisation, Méthodologie

Abstract

This thesis focuses on the design of Multi-Agent Based Simulations (MABS). MABS are aimed at reproducing real phenomena of intuitively modeling their inner mechanisms using autonomous entities behaving in a synthetic environment. For either of the following two reasons, current MABS design techniques are unsatisfactory : the interactions are limited to exchanging messages or MABS are restricted to formal specifications. As a consequence, large-scale simulations – *e.g.* featuring many agents and many interactions – are still hard to design.

In this context, our research promotes a more homogeneous knowledge representation, resulting in practice into *the reification of each entity by an agent and each behaviour by an interaction*. That way, the simulation engine has to be separated from both the agents and the interactions, enabling the MABS to be *ruled by a single and accurate algorithm*. Besides, we show that the interactions can be re-used, the models incrementally designed and the implementation automated.

Based on these principles, we have developped IODA, an interaction-oriented approach to simulation design. IODA provides a formal model for describing the agents and their interactions, a set of algorithms for unifying their action selection and a methodology for incremental design. IODA also features JEDI, a parameterizable simulation framework faithful to IODA's principles, and JEDI-BUILDER, an automated tool for translating IODA models into JEDI code.

As a result of this research, it is shown that the reification of the interaction leads not only to the unification of the agent concept, but also to a simplification of the simulation design process.

Keywords: MultiAgent Based Simulation, Interaction, Modeling, Methodology