Master 2 Recherche Informatique

Rapport de Stage

Systèmes Multi-Agents et Answer Set Programming : Raisonnement en Environnement Dynamique

Tony Ribeiro
Université d'Angers
ribeiro@info.univ-angers.fr

Résumé

Dans nos travaux nous nous intéressons aux systèmes multi-agents en environnement dynamique. Notre intérêt se porte tout particulièrement sur le raisonnement d'un agent dans un tel cadre. Raisonner en environnement dynamique implique de manipuler des connaissances de façon non monotone. Afin de compléter ses observations, un agent pourra émettre des hypothèses par abduction et mettre à profit les connaissances de ses pairs pour affiner celles-ci. Pour évoluer dans un environnement changeant, un agent doit être capable de s'adapter. Notre objectif est l'établissement d'une méthode facilitant la modélisation et l'utilisation des connaissances d'un agent en environnement dynamique. Afin de représenter le savoir de nos agents, nous utilisons la puissante expressivité de l'answer set programming. Pour modéliser le raisonnement d'un agent, nous proposons une méthode basée sur les modules ASP. Nous proposons également un framework permettant d'implémenter et d'utiliser cette méthode dans un système multi-agents.

Mots Clef

Answer Set Programming, Systèmes Multi-Agents, module ASP, métaconnaissance, résilience

Abstract

In this work, we focus on multi-agent systems in dynamic environment. Our interest is about individual agent reasoning in such environment. For reasoning in dynamic environment, an agent needs to be able to manage his knowledge in a non-monotonic way. To complete his observations, an agent can use abduction to produce hypotheses and refines them with the help of other agents. To reach his goals in a changing environment, an agent needs some capacities like adaptability. Our objective is to define a method which makes easier to design agent knowledge and reasoning in such environment. We use the expressivity of answer set programming to represent agent knowledge. To design agent reasoning, we propose a method based on ASP modules. We also propose a framework to implement and use this method in multi-agent systems.

Keywords

Answer Set Programming, Multi-Agent Systems, ASP module, meta-knowledge, resilience

Table des matières

Chapitre 1

Introduction

1.1 Contexte

Ces dernières années la recherche dans le domaine des Systèmes Multi-Agents (SMA) est particulièrement active. Avec l'évolution matérielle de l'informatique en matière de calculs distribués, les applications des SMA deviennent des problèmes concrets. Les interactions d'agents au sein d'un SMA soulèvent d'intéressantes problématiques, aussi bien au niveau de l'agent que au niveau du groupe. Certains travaux se concentrent sur les communications, étudiant différents protocoles et la propagation de l'information. D'autres s'intéressent aux connaissances des agents sur leurs pairs et l'impact de ces connaissances sur le raisonnement de ces derniers. D'autres encore se penchent sur le raisonnement de groupe, comme l'apprentissage distribué ou bien l'argumentation. Enfin le raisonnement individuel d'un agent au sein d'un SMA est également le sujet de nombreuses recherches, et c'est également ce dernier point qui nous intéresse ici. Nous considérons donc des agents qui, en temps réel, raisonnent et agissent en parallèle. Pour atteindre leurs objectifs, nos agents doivent être réactifs et posséder une certaine flexibilité au niveau du raisonnement. Il est également nécessaire pour eux de s'adapter aux évolutions de leur environnement.

1.2 Objectifs

Notre intérêt se porte donc sur les SMA en environnement dynamique et plus précisément sur le raisonnement d'un agent au sein d'un tel réseau. En environnement dynamique, un agent doit être capable de gérer ses connaissances de façon non monotone afin d'atteindre ses objectifs. Dans un tel cadre, un agent doit constamment mettre à jour ses observations, cela comprend aussi bien l'ajout de nouvelles informations, que la révision de ses connaissances passées. Entre autres choses, un agent pourra également recourir à l'abduction, afin d'émettre des hypothèses lui permettant de compléter ses connaissances. Le caractère changeant de l'environnement dans lequel évolue nos agents, nécessite de leur part une certaine adaptabilité. En fonction de la situation, un agent doit être capable d'adapter son comportement et ses actions en conséquence. La modélisation de ce type de raisonnement est des plus compliquée, c'est pourquoi nous portons nos efforts sur la simplification de cette tâche.

Chapitre 2

État de l'art

Afin de pouvoir discuter de notre contribution, il convient dans un premier temps d'explorer l'état de l'art. Dans ce chapitre, nous nous intéresserons aux différents champs de recherche en relation avec notre problématique. Dans un premier temps, nous introduirons quelques bases à propos des systèmes multi-agents. Ensuite, nous présenterons les principales formes de raisonnement logique et discuterons brièvement des particularités liées à leur utilisation en environnement dynamique. Enfin notre intérêt se portera sur l'answer set programming, nous poserons alors les bases nécessaires à sa compréhension et son utilisation.

2.1 Systèmes multi-agents

Notre première et principale source d'information sur le domaine des systèmes multi-agents provient de l'étude de la thèse de Gauvain Bourgne [Bourgne, 2008]. Dans sa thèse, l'auteur s'intéresse au raisonnement hypothétique et à la propagation du savoir au sein des SMA. Les définitions qui suivent sont en grande partie basées sur les travaux de cette thèse.

Définition 1 (Agent) *Un agent est une entité autonome dotée de capacités de raisonnement et de communication. Le savoir d'un agent peut être définie par deux ensembles :*

- *K un ensemble de connaissances non révisables composé de :*
  - *T la théorie commune, base commune des agents du réseau.*
  - *O l'ensemble des observations reçues par l'agent.*
- *B un ensemble de croyances révisables.*

La théorie commune représente le savoir de base que tout agent possède au sein du SMA, ce savoir est considéré comme certain et n'est donc pas révisable. Une théorie commune peut également se limiter à un groupe d'agents qui n'englobe pas tout le système. Les observations proviennent de l'environnement de l'agent et représentent l'état actuel du monde dans lequel il évolue. Si on considère que les capteurs et la mémoire de l'agent sont parfaits, ces informations sont certaines et donc non révisables.

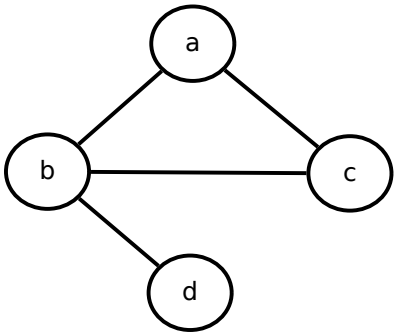

FIGURE 2.1 – Un système multi-agents de 4 agents et 4 liens de communications.

Dans un environnement dynamique le monde est en perpétuelle évolution, ce qui implique qu'une observation n'est certaine qu'à l'instant T où elle est observée. Selon la représentation utilisée, la mémoire de l'agent peut constituer un savoir révisable ou non. La mémoire est révisable lorsque la notion temporelle est absente des observations qu'elle contient.

Les croyances représentent les informations supposées vraies par l'agent. Ces connaissances proviennent d'inférences non monotones et constituent donc un savoir révisable. Ce savoir est constitué d'hypothèses construites par l'agent afin de compléter ses connaissances. Ces hypothèses permettent d'expliquer les observations vis à vis de la théorie commune.

Définition 2 (Système multi-agents) *Un système multi-agents (SMA) est un réseau d'agents dans lequel ils peuvent interagir entre eux. Un tel système peut être représenté par un graphe où :*
- *les noeuds représentent des agents*
- *les arcs représentent les liens de communications.*

Exemple 1 *La figure 2.1 représente un système multi-agents avec communications bilatérales. Ce système se compose de quatre agents a, b, c, d et de quatre liens de communications entre : a et b, a et c, b et c, b et d. Dans cet exemple, l'arc (a,b) représente le fait que a peut envoyer des messages à l'agent b et vice versa.*

Considérer des agents au sein d'un système multi-agents soulève des questions intéressantes. Notamment au niveau des communications, dans [Bourgne, 2008], [Bourgne et al., 2010b] et [Bourgne et al., 2010a] les auteurs s'intéressent particulièrement à la propagation de l'information au sein d'un groupe d'agents. Nous pouvons également citer d'autres travaux tels que [Sakama et al., 2011], où les agents ont la possibilité de mentir pour atteindre leurs buts. Ici les agents sont en compétition et usent de persuasion pour atteindre leurs buts. D'autres travaux encore s'intéressent à la planification de groupe, dans [Nieuwenborgh et al., 2006] on considère une hiérarchie au sein du SMA. Cette hiérarchie permet une certaine flexibilité du planning en préférant la satisfaction des plus haut gradés et permet d'intéressantes simplifications des communications : un supérieur pourra par exemple imposer son avis à ses subordonnés sans avoir à le justifier.

2.2 Raisonnement

Le raisonnement d'un agent est d'autant plus intéressant lorsqu'il est considéré dans un système multi-agents, car il doit alors prendre en compte la présence de ses pairs. Avant de s'intéresser à l'impact de la présence et des interactions des autres agents sur le raisonnement d'un seul, il convient d'introduire quelques notions de base concernant le raisonnement logique. Nous abordons ici trois méthodes de raisonnement usuellement employées par un agent : la déduction, l'induction et l'abduction.

Définition 3 (Déduction) *La déduction est l'application de la règle d'inférence du modus ponens, qui consiste à inférer la conclusion d'une règle d'implication logique lorsque ses prémisses sont vérifiées.*

Le raisonnement déductif permet d'inférer de nouveaux faits à partir de règles et de faits existants. Un agent combinera sa théorie et ses observations afin de déduire de nouveaux faits, qui viendront enrichir ses connaissances certaines. Alors qu'un savoir déduit de connaissances certaines est également non révisable, un savoir déduit de connaissances incertaines est révisable. Dans un environnement dynamique, les observations ne sont certaines qu'à l'instant où elles sont perçues et il en va donc de même pour le savoir qui en est déduit.

Définition 4 (Induction) *L'induction consiste à inférer des règles à partir d'un ensemble d'exemples.*

Le raisonnement inductif permet de produire des règles générales en se basant sur des cas particuliers, pour un agent il s'agira de généraliser ses observations. Ces règles sont révisables, qu'elles soient induites de connaissances certaines ou non. Ce raisonnement est entre autre utilisé en apprentissage automatique. Dans [Nicolas and Duval, 2001] les auteurs utilisent l'induction pour apprendre simultanément un concept et sa négation. L'induction est une forme de raisonnement non monotone car produisant des connaissances révisables. Dans un environnement dynamique, elle permet à un agent d'apprendre de son environnement, de s'adapter à son évolution.

Définition 5 (Abduction) *L'abduction consiste à inférer les prémisses d'une règle lorsque sa conclusion est vérifiée.*

Le raisonnement abductif permet de produire des hypothèses afin d'expliquer un ensemble d'observations au regard de nos connaissances. Une hypothèse valide est une conjonction de faits qui combinée à notre savoir implique logiquement nos observations. Un agent aura recours à cette forme de raisonnement pour compléter ses connaissances. Au sein d'un SMA, un agent pourra mettre à profit ses congénères afin d'affiner la formation d'hypothèses. La distinction entre les concepts d'abduction et d'induction est parfois ténue, c'est pourquoi nous avons choisi ici de les définir ainsi. En effet ces deux raisonnements peuvent permettre d'inférer des règles révisables, or ici, nous limitons l'abduction à la production d'ensemble de faits.

2.3 Answer set programming

L'answer set programming (ASP) est une forme de programmation déclarative basée sur la sémantique des modèles stables de la programmation logique. On utilise également le terme answer set pour désigner un modèle stable, d'où l'appellation answer set programming. On parle de programmation déclarative car on décrit un problème et non comment le résoudre. Un programme ASP se compose de faits, de règles et de contraintes. En ASP la recherche de solutions se limite au calcul des modèles stables du programme, des ensembles de faits déduits des règles et respectant les contraintes. Pour cela on utilise un solver qui retournera l'ensembles des answer sets du problème. De nombreux travaux de recherche ont pour sujet l'ASP ou l'utilise pour modéliser les connaissances. Plusieurs travaux de recherche tel que [Baral et al., 2010], [Nieuwenborgh et al., 2006] utilisent l'ASP. Dans le premier les auteurs s'en servent afin de modéliser le savoir d'un agent sur ses homologues. Dans le second il est question de fléxibilité du raisonnement avec l'introduction de contrainte souples : des contraintes violables sous certaines conditions. D'autres travaux tels que [Costantini, 2010] s'évertuent à développer des méthodes ASP offrant certaines propriétés au raisonnement d'un agent, ici il est question de rendre les agents réactifs par l'utilisation de modules ASP. Dans ces modules, les contraintes sont vues comme les conditions nécessaires à l'activation de celui-ci. Dans [Costantini, 2009], l'auteur statue sur l'intégration de modules ASP au sein du raisonnement d'un agent. D'autres encore, tels que [Baral et al., 2006] ou [Faber and Woltran, 2011], s'intéressent aux possibles extensions du paradigme ASP.

Définition 6 (Règle) *Soit H, a_i et b_i des littéraux et la règle*

$$H \leftarrow a_1 \wedge \ldots \wedge a_n \wedge \neg b_1 \ldots \wedge \neg b_m.$$

Si tout les a_i sont vrais et tout les b_i ne sont pas vrais alors H est vrai. H est la tête de la règle et la conjonction des a_i, $\neg b_i$ en est le corps.

Note 1 *Dans les exemples qui suivent nous représentons :*
- $\leftarrow$ *par :-*
- $\neg$ *par not*
- $\wedge$ *par ,*
- *la fin d'une règle par .*

Ces symboles sont utilisés dans les programmes ASP et autres langages de programmation logique à des fins de commodités d'écriture.

Exemple 2 *Ici nous avons trois règles ASP :*
parapluie :- pluie.
mouillé :- pluie, not parapluie.
ouvrir :- not verrouillé.

Définition 7 (Fait) *Un fait est une règle sans corps*

$$H.$$

signifiant que H est vrai.

Exemple 3 *Ici nous avons trois faits :*
pluie.
parapluie.
verrouillé.

Définition 8 (Contrainte) *Une contrainte est une règle sans tête*

$$\leftarrow a_1 \wedge \ldots \wedge a_n \wedge \neg b_1 \ldots \wedge \neg b_m.$$

Si le corps de la règle est vrai cela implique $\perp$: une contradiction.

Exemple 4 *Ici nous avons deux contraintes :*
:- pluie, not parapluie.
:- arc(X,Y), couleur(X,C), couleur(Y,C).

Définition 9 (Programme ASP) *Un programme ASP est un ensemble de règles.*

Exemple 5 *Un programme ASP composé d'un fait, de trois règles et d'une contrainte :*

pluie.
rester :- not sortir.
sortir :- not rester.
mouillé :- pluie, sortir, not parapluie.
:- mouillé.

Définition 10 (Answer set) *Un answer set d'un programme P est un ensemble de faits constituant un modèle stable de P.*

Exemple 6 *Soit $P =$*

pluie.
rester :- not sortir.
sortir :- not rester.
mouillé :- pluie, sortir, not parapluie.
:- mouillé.

L'ensemble de fait {pluie, rester} est un answer set de P alors que {pluie, sortir, mouillé} n'en est pas un car il viole une contrainte.

L'answer set programming est un paradigme de programmation expressif adéquat pour représenter la connaissance d'un agent. Les derniers solvers ASP sont très efficaces : clasp, un solver ASP basé sur des technologies SAT, à été classé premier toutes catégories lors d'une compétition SAT en 2009. Le lecteur pourra se référer à [Gebser et al., 2011] pour plus de détails sur clasp et le projet potassco dont il fait parti. L'évolution récente des technologies SAT permettent d'utiliser l'ASP dans des applications réelles.

2.4 Métaconnaissance

Définition 11 (Métaconnaissance) *Métaconnaissance est le terme qui désigne de la connaissance sur la connaissance.*

Exemple 7 *La métaconnaissance peut par exemple être une connaissance :*
- *qui décrit des connaissances*
- *sur l'utilisation des connaissances*
- *sur la véracité d'une connaissance*
- *sur les connaissances des autres*

Parmi les recherches entreprises sur le raisonnement, beaucoup concernent de près ou de loin la métaconnaissance. Dans [Baral et al., 2010], les auteurs s'intéressent à l'influence sur le raisonnement d'un agent de la connaissance qu'il possède sur le savoir des autres. À leur tour les auteurs de [Sakama et al., 2011] se penchent également sur la métaconnaissance : il y est question d'agents malhonnêtes qui communiquent des informations qu'ils savent fausses afin d'atteindre leurs buts. Le savoir sur autrui est très important ici car il permet à l'agent d'adapter son discours en conséquence. Dans la thèse [Bourgne, 2008], l'auteur met l'accent sur l'intérêt pour un agent de connaître les croyances des autres. Connaître une partie des connaissances de ses congénères permet à un agent d'améliorer son argumentaire lorsque il communiquent avec eux. Cela permet d'optimiser les communications en limitant la quantité d'information échangée au strict nécessaire.

Chapitre 3

Propositions

Dans nos travaux, notre intérêt se porte sur la représentation du raisonnement d'un agent évoluant en environnement dynamique au sein d'un système multi-agents. Afin de faciliter la compréhension du lecteur nous suivrons un exemple intuitif au fil de nos propositions, il s'agit d'un exemple de SMA dans un environnement dynamique : un jeu de survie. Nous avons donc des loups, des lapins et des fleurs. Nos agents peuvent appartenir à l'un de ces trois groupes. Chaque agent a alors des objectifs et des comportements propres à son genre. Les agents peuvent coopérer avec leurs semblables et sont en compétition avec les autres type d'agents. Pour faire simple : les loups cherchent à manger les lapins, qui eux cherchent à manger les fleurs.

3.1 Jeu de survie

Les loups n'ont qu'un seul objectif : se nourrir. Pour ce faire, ils doivent attraper leurs proies, à savoir les lapins. Ici les loups sont au sommet de la chaîne alimentaire, leurs comportements se concentre donc sur la chasse. Nos prédateurs peuvent se trouver dans deux cas de figure : une proie est en vue ou non. Lorsque aucune source de nourriture n'est dans son champ de vision, notre loup devra explorer son environnement. Si un loup repère un lapin, il tentera d'abord de s'en approcher sans se faire lui même repérer. Une fois découvert le

FIGURE 3.1 – Un SMA dans un environnement dynamique où les agents sont soit des loups, des lapins ou des fleurs. Les loups mangent les lapins, qui eux mangent les fleurs.

loup passera à l'attaque, ne se souciant plus alors d'être vu ou non.

Les loups ont donc trois types de comportements : l'exploration, l'approche et l'attaque. Ils auront également tendance à coopérer avec les autres loups pour attraper les lapins.

Les lapins ont deux objectifs : se nourrir et survivre. Ils doivent donc d'une part manger des fleurs et d'autre part échapper aux loups. Ce qui est intéressant chez ces agents c'est le fait que leurs objectifs soient conflictuels. Comme le loup, si aucune proie n'est en vue, notre lapin doit étendre son champ d'action en explorant son environnement. Cependant notre ami peut aussi bien trouver son bonheur que faire celui de ses prédateurs. L'exploration sera donc plus réfléchie chez le lapin que pour notre loup. Par exemple, pour aller se nourrir il privilégiera les positions lui offrant le meilleur champs de vision afin de facilement repérer ses prédateurs. Une fois un prédateur détecté, la survie prend le dessus sur la faim et la fuite s'impose alors.

Les fleurs sont des agents passifs : leur comportement n'est pas influencé par l'environnement. Dans notre exemple, les fleurs ne font que se reproduire au fil du temps. Ce sont des agents statiques, indifférent aux changements de leur environnement et totalement indépendant au sein de leur groupe.

3.2 Module ASP

Un module ASP est un programme ASP qui se distingue de part son contenu et son utilisation. Le premier intérêt des modules est leur simplicité, il s'agit de petits programmes dédiés à une utilisation spécifique. Un module ASP peut contenir des observations sur les alentours, un autre contiendra des règles définissant ce qu'est une proie, un autre encore sera dédié au calcul de chemin selon plusieurs critères d'optimisation. Pour obtenir les chemins menant aux proies alentours, on combinera les connaissances de ces trois modules. En combinant nos modules et en utilisant la déduction nous sommes à même de produire un nouveau savoir, c'est la que réside l'intérêt de cette modélisation. Avec la déduction, un ensemble de modules représente plus de connaissances que individuellement : le tout est plus que la somme des parties.

3.2.1 Observations

Dans notre approche nous considérons un système multi-agents dans un environnement dynamique. Un environnement dynamique est non monotone vis à vis du temps, une information peut être vraie à un instant T et devenir fausse à $T+1$. Moins formellement, si nous considérons qu'un agent peut se déplacer, alors sa position constitue une information qui peut changer en fonction du temps. Si un agent a observe un agent b à une position p au temps T alors le fait $position(b, p)$ est actuellement vrai. Admettons que b ait changé de position à $T+1$, alors le fait $position(b, p)$ est maintenant faux. Pour évoluer dans un tel environnement un agent a besoin de régulièrement mettre à jour ses observations. Il convenait donc de définir un type de module dédié à la représentation et l'organisation des observations.

Définition 12 (Module d'observations) *Un module d'observations est constitué d'un ensemble de faits. Son contenu est dynamique : il change en fonction du temps. L'agent utilise ces modules comme des bases de données spécialisées. L'objectif premier est d'organiser les observations afin de faciliter leur consultation et leur mise à jour.*

Exemple 8 *Un module d'observations dédié aux positions des loups :*

position(loup,0,3).
position(loup,2,5).
position(loup,2,6).

Un module d'observations décrivant l'agent :

moi("Bugs Bunny").
position("Bugs Bunny",0,0).
type("Bugs Bunny",lapin).

3.2.2 Théorie

Dans la section précédente, nous avons défini un type de module qui nous permet de représenter l'ensemble des observations reçues par l'agent. Pour représenter la théorie commune de nos agents, nous avons besoin d'un autre type de module : le module de règles.

Définition 13 (Module de règles) *Un module de règles est constitué d'un ensemble de règles. Son contenu est établit à la création de l'agent et ne change pas au fil du temps. Ces modules contiennent des connaissances générales sur un domaine spécifique. Ils permettent d'organiser le savoir et de faciliter son exploitation.*

Exemple 9 *Un module de règles concernant la chaîne alimentaire pour un lapin :*

prédateur(loup).
congénère(lapin).
proie(fleur).

Un module de règles dédié au calcul de chemin :

chemin(X,Y) :- arc(X,Y).
chemin(X,Y) :- chemin(X,Z), arc(Z,Y).

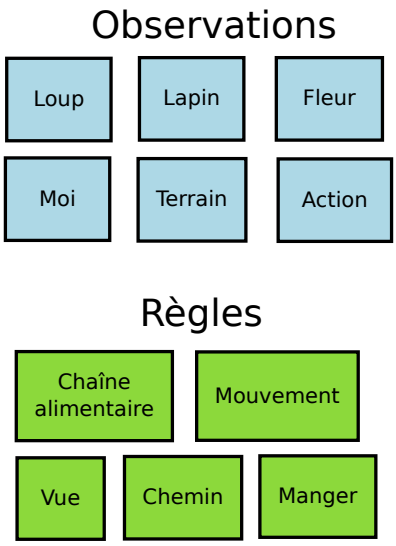

FIGURE 3.2 – Un exemple d'organisation des connaissances utilisant des modules ASP. En bleu clair les modules d'observations et en vert les modules de règles.

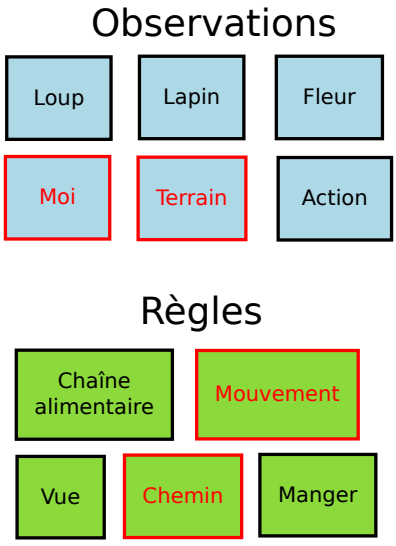

FIGURE 3.3 – Ici la combinaison de modules {Moi, Terrain, Mouvement, Chemin} retournera toutes les positions que l'agent peut atteindre.

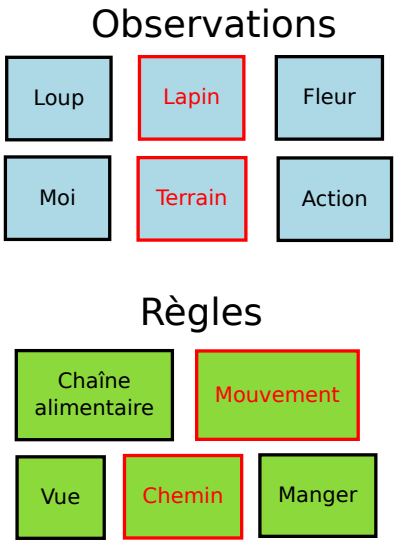

FIGURE 3.4 – Ici la combinaison de modules {Lapin, Terrain, Mouvement, Chemin} retournera toutes les positions que les lapins en vue peuvent atteindre.

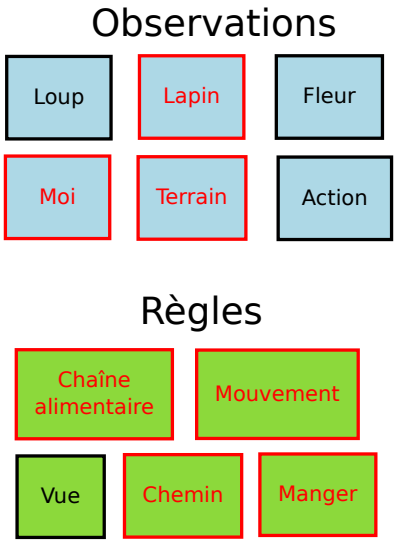

FIGURE 3.5 – Ici notre agent est un loup. La combinaison de modules {Moi, Terrain, Lapin, Mouvement, Chemin, Chaîne alimentaire, Manger} retournera les chemins permettant de manger un lapin.

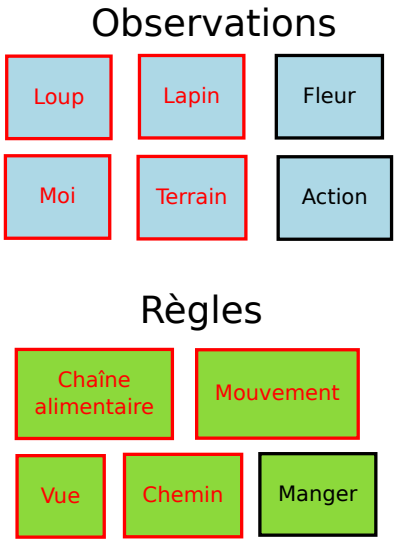

FIGURE 3.6 – Ici encore notre agent est un loup. La combinaison de modules {Moi, Loup, Lapin, Terrain, Mouvement, Chemin, Chaîne alimentaire, Vue} retournera les chemins permettant de s'approcher d'un lapin tout en évitant de se faire repérer par la cible.

La figure 3.2 représente l'organisation des connaissances d'un agent. Nous avons d'une part les modules d'observations et d'autre part les modules de règles. L'intérêt de cette modélisation est qu'en combinant les modules, on obtient par déduction un savoir qui diffère selon la combinaison. Prenons par exemple les modules *Mouvement* et *Chemin* qui permettent de calculer les déplacements d'un agent. Un agent peut les utiliser pour calculer ses propre possibilités de mouvement, mais également celles des autres. On voit ici que cette modélisation permet entre autres de factoriser le code : un même module peut être utilisé de façons différentes. Les figures 3.3, 3.4, 3.5 et 3.6 montrent différentes façons de combiner nos modules.

3.2.3 Métaconnaissance

Dans la section précédente nous avons présenté différentes manières de combiner nos modules d'observations et de règles. Mais ces combinaisons représentent également du savoir et peuvent par conséquent être connues par nos agents. Un agent possède alors des métaconnaissances sur ses propres connaissances, plus précisément sur l'utilisation de ses connaissances. Cette métaconnaissance fait partie de la théorie commune des agents et pourrait être représentée dans les modules de règles. Cependant le caractère singulier de ce savoir nécessite qu'on lui réserve un type de module. C'est pourquoi nous introduisons maintenant les modules de métaconnaissance.

Définition 14 (Module de métaconnaissance) *Un module de métaconnaissance est constitué d'un ensemble de règles ayant pour conclusion l'utilisation d'un module. Leur contenu est établit à la création de l'agent et ne changent pas*

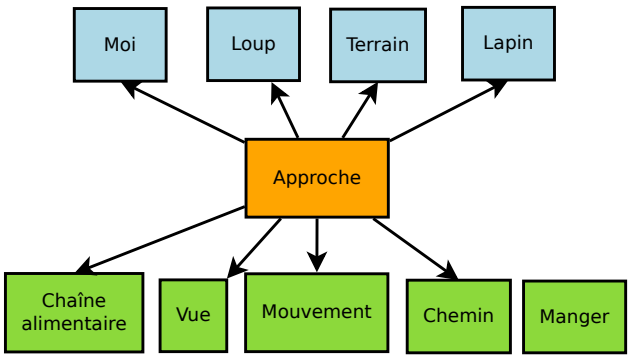

FIGURE 3.7 – En orange un module de métaconnaissance sur l'approche. Une flèche représente le fait qu'un module contient de la connaissance sur l'utilisation du module pointé. Ici le module *Approche* définit la combinaison de module à utiliser pour calculer les chemins permettant à un loup d'approcher un lapin sans être repéré.

au fil du temps. Ces modules contiennent des connaissances sur les combinaisons de modules. Ils permettent d'organiser le savoir et d'orienter le raisonnement.

Exemple 10 *Un module de métaconnaissance définissant le comportement d'approche :*

```
utiliser_module("Moi").
utiliser_module("Loup").
utiliser_module("Lapin").
utiliser_module("Terrain").
utiliser_module("Chaîne alimentaire").
utiliser_module("Chemin").
utiliser_module("Mouvement").
utiliser_module("Vue").
```

Exemple 11 *Un module de métaconnaissance définissant le comportement de chasse :*

```
utiliser_module("Attaque") :- repéré.
utiliser_module("Approche") :- not repéré.
```

Les modules de métaconnaissance les plus triviaux ne représentent qu'une combinaison de modules, comme 3.7 et 3.8. Cela peut être suffisant pour représenter un comportement simple. Pour les comportements plus complexes, il est tout à fait possible d'avoir des modules de métaconnaissance ayant des connaissances sur d'autres modules de métaconnaissance.

Pour notre loup nous avons les comportements d'approche et d'attaque qui définissent en fait deux phases d'un comportement plus complexe : la chasse. La figure 3.9 et l'exemple 11 représentent ce comportement : si notre loup est repéré il utilisera le module d'attaque sinon il utilisera celui d'approche.

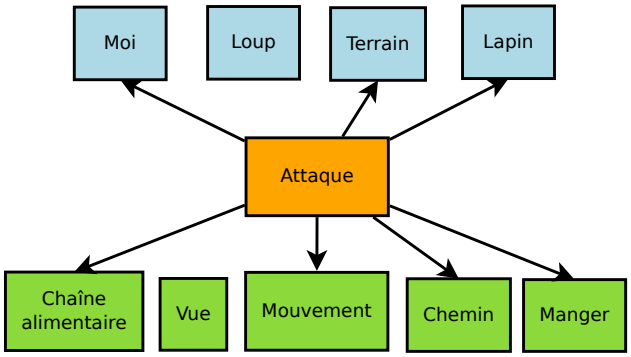

FIGURE 3.8 – Ici le module *Attaque* définit la combinaison de modules à utiliser pour calculer les chemins permettant à un loup de manger un lapin.

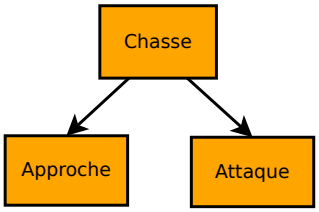

FIGURE 3.9 – Représentation graphique du module de métaconnaissance *Chasse* de l'exemple 11, qui contient de la connaissance sur des modules de métaconnaissances. Selon que le loup soit repéré ou non il utilisera le module d'approche ou celui d'attaque.

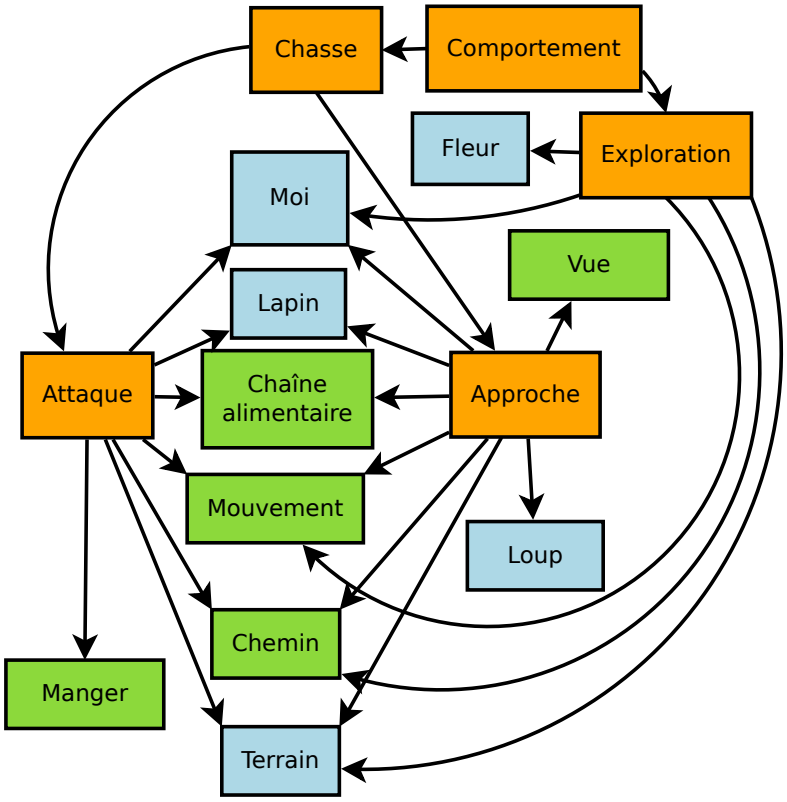

FIGURE 3.10 – Organisation des modules de connaissances d'un loup.

3.2.4 Organisation

Avec ces trois types de modules nous sommes à même de représenter les connaissances d'un agent. La figure 3.10 représente le savoir d'un loup sous forme d'un graphe de modules. Les noeuds du graphes sont des modules, les arcs ont pour origine un module de métaconnaissance et signifient qu'il contient de la connaissance sur celui qui est ciblé. On représente ici les deux comportements de notre animal : l'exploration et la chasse. La chasse se décomposant elle même en deux comportements : l'approche et l'attaque. Les modules de métaconnaissance permettent ici de décomposer les différents comportements de nos agents, facilitant ainsi leur représentation et leur implémentation.

Jusqu'à maintenant nous avons définit trois types de modules ASP : le module d'observations, le module de règles et le module de métaconnaissance. Ces deux derniers constituent la théorie commune de nos agents et restent inchangés au cours du temps. Le cas des modules d'observations est différent : ils doivent être mis à jour en fonction des dernières observations.

Dans la première implémentation de notre framework, la mise à jour des modules d'observations était effectuée du côté application. Les senseurs de nos agents maintenaient à jour les modules qui leurs étaient assignés. En fonction du type d'observation le senseur l'ajoutait dans le module d'observation corre-

spondant. Comme la combinaison de module, cette opération peut être connue de l'agent et constitue donc une métaconnaissance : une connaissance sur la gestion des connaissances. Cette métaconnaissance permet à l'agent de gérer ses observations via le raisonnement.

Nous avons donc également dédié un type de module à la représentation de cette métaconnaissance : le module d'acquisition.

3.2.5 Acquisition

Définition 15 (Module d'acquisition) *Un module d'acquisition est constitué d'un ensemble de règles ayant pour conclusion une commande de mise à jour. Ces commandes de mise à jour concernent un module d'observation et peuvent être :*

- *Ajouter un fait*
- *Retirer un fait*

Comme pour les modules de règles et de métaconnaissances, leur contenus est établit à la création de l'agent et ne changent pas au fil du temps. Ces modules contiennent des connaissances sur la mise à jour des modules d'observations. Ils permettent d'organiser les observations et de les interpréter avant de les stocker.

Exemple 12 *Un module d'acquisition sur les positions des fleurs :*

ajouter("Fleur",position(fleur,X,Y)) :- observation(position(fleur,X,Y)), not position(fleur,X,Y).
retirer("Fleur",position(fleur,X,Y)) :- position(fleur,X,Y), not observation(position(fleur,X,Y)).

Exemple 13 *Un module d'acquisition sur les positions des lapins :*

ajouter("Lapin",position(lapin,X,Y)) :- observation(position(lapin,X,Y)), not position(lapin,X,Y).
retirer("Lapin",position(lapin,X,Y)) :- position(lapin,X,Y), observation(position(E,X,Y)), not observation(position(lapin,X,Y)).

Dans l'exemple 12, nous avons un module d'acquisition qui limite le contenu du module d'observations *Fleur* aux observations courantes. La première règle ajoute les positions des fleurs qui ne sont pas déjà connues, d'où le *not*. La seconde retire les positions des fleurs qui ne sont plus dans le champs de vision.

Un module d'acquisition un peu plus élaboré comme celui de l'exemple 13, peut permettre de gérer une mémoire plus étendue dans le temps. Dans cet exemple, la position d'un lapin est conservée jusqu'a ce que notre agent constate qu'il n'y a plus de lapin à cette position. La seconde règle se traduit par : si une position connue d'un lapin est actuellement observée et qu'aucun lapin n'est présent, alors retirer le fait qu'un lapin est à cette position.

La figure 3.11 montre un module d'acquisition dédié aux positions des autres agents, celui-ci consulte et met à jour les modules d'observation *Loup*, *Lapin* et *Fleur* avec les nouvelles observations reçues par l'agent. Un module d'acquisition peut également générer de nouvelles informations par déduction. Par exemple, si un agent observe de la fumée il peut en déduire qu'il y a un incendie et donc

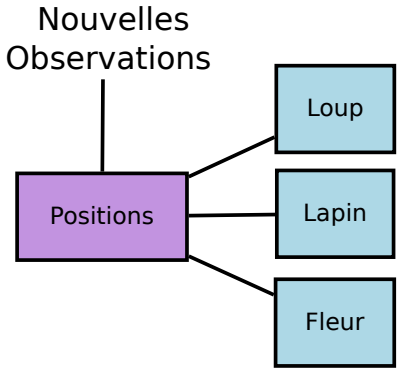

FIGURE 3.11 – En violet un module d'acquisition dédié à la mise à jour des positions des agents.

directement stocker le fait qu'il y a un incendie. C'est pourquoi nous utilisons le terme acquisition, ces modules permettent à l'agent d'acquérir de nouvelles connaissances : non seulement des nouvelles observations mais également leurs conséquences.

Avec l'ajout des modules d'acquisition, nos agents sont à même d'évoluer dans un environnement dynamique. Il peuvent à présent réviser leur mémoire au gré de leurs observations. Avec ces quatre types de modules, un agent peut raisonner sur des faits certains : ses observations courantes. Et également sur des faits incertains : ses observations passées. Le raisonnement de nos agents est donc, pour l'instant, uniquement basé sur leurs seules observations. Pour améliorer leur capacité de raisonnement, il convient de leur permettre de compléter leurs connaissances.

Prenons un exemple concret : un loup qui attaque un lapin. Au temps T, notre loup connaît la position du lapin car sa proie est dans son champ de vision. Au temps $T+1$, le lapin à disparu du champ de vision du loup en passant derrière un rocher. Si le raisonnement du loup se limite à celui de nos quatre modules, il abandonnera la poursuite. Pour continuer sa traque, notre prédateur à besoin d'un but : il lui faut la nouvelle position de sa proie même si cette position est hypothétique. Dans notre exemple, le loup doit supposer la nouvelle position du lapin : faire une hypothèse et pour la construire il peut avoir recours au raisonnement abductif.

Afin de permettre à nos agents de compléter leurs connaissances, nous avons consacré un module à cette fin : le module d'abduction.

3.2.6 Abduction

Définition 16 (Module d'abduction) *Un module d'abduction est constitué de trois éléments : un ensemble d'abducibles, de règles et une contrainte. Les abducibles sont les faits que l'on autorise à être supposés et qui constitueront donc les hypothèses. Ces modules contiennent une règle qui permet de générer toutes les hypothèses possibles : tout les sous-ensembles d'abducibles. Enfin, la contrainte définie une hypothèse valide comme étant un ensemble minimal d'ab-*

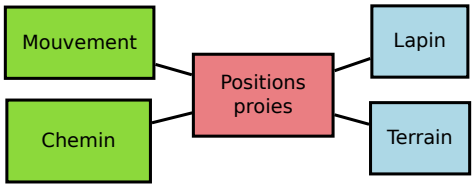

FIGURE 3.12 – Un module d'abduction combiné à deux modules de règles et deux modules d'observations.

ducibles expliquant toutes les observations. Ces modules font partie de la théorie commune : leur contenu est établit à la création de l'agent et ne change pas au fil du temps. Les modules d'abduction permettent de compléter le savoir d'un agent par le raisonnement hypothétique.

Exemple 14 *Un module d'abduction sur la disparition d'un lapin :*

abducible(bouger).
abducible(mort).

conclusion(disparu) :- hypothese(bouger).
conclusion(disparu) :- hypothese(mort).

0{hypothese(X) : abducible(X)}.

:- observation(X), conclusion(X).

L'exemple 14 définie un module d'abduction d'un loup concernant la disparition d'un lapin. Ce module se compose de deux abducibles : *bouger* et *mort*. Il contient également deux règles concluant sur la disparition de la proie et un agrégat générant toutes les combinaisons possibles d'abducibles. Enfin, la contrainte limite les answer sets aux hypothèses permettant d'expliquer les observations.

Ici trois hypothèses sont possibles :
- Le lapin a bouger
- Le lapin est mort
- Le lapin a bouger et est également mort

Cependant, la troisième hypothèse est non minimal puisque la première et la deuxième en sont des sous-ensembles. Notre framework ne conservera que les deux premières hypothèses. L'agent doit alors explorer les conséquences des différentes hypothèses afin de choisir celle qui lui semble préférable. Dans notre exemple, le lapin a effectivement bougé puisqu'il est passé derrière un rocher. Notre loup n'a aucune observation venant appuyer le fait que sa proie soit morte : il ne l'a pas mangé et à sa connaissance, aucun de ses semblables n'est aux alentours. De plus, si ce lapin est le seul en vue, supposer sa mort revient pour le loup à perdre son seul objectif. Si finalement le prédateur choisit de supposer que sa proie a bougé, il pourra utiliser ses connaissances sur les lapins pour déterminer les positions possibles de sa proie.

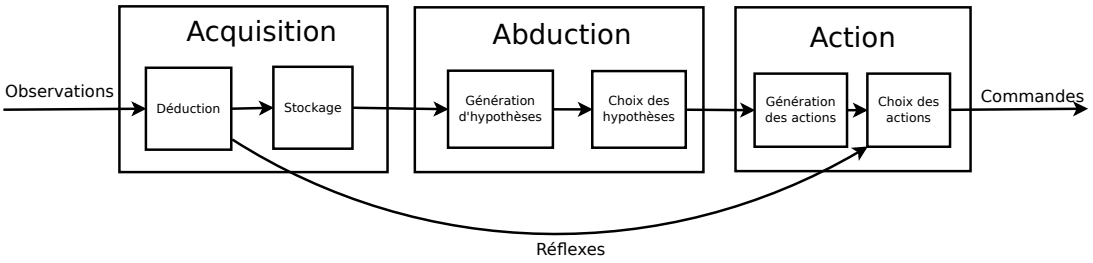

FIGURE 3.13 – Les différentes étapes de raisonnement d'un agent.

Avec ce cinquième module, nos agents sont maintenant capables de raisonner de façon non monotone. Nous pouvons maintenant définir la structure générale du raisonnement d'un agent. La figure 3.13 présente les différentes étapes qui constituent le raisonnement de nos agents. Nous avons dans un premier temps la phase d'acquisition : l'agent reçoit de nouvelles observations de ses senseurs et met alors à jour ses observations. Puis vient la phase d'abduction, pendant laquelle l'agent émet des hypothèses afin de compléter son savoir. Il raisonnera ensuite sur les interactions qu'il peut avoir avec son environnement : c'est la phase d'action. L'agent combine alors sa théorie, sa mémoire, ses nouvelles observations et ses hypothèses afin de déterminer son champs d'action. Enfin, il choisira les actions à effectuer en fonction de ses objectifs.

Le raisonnement hypothétique permet à un agent de compléter ses connaissances et d'enrichir son raisonnement, mais selon la situation il peut être aussi bien vitale qu'une perte de temps. Reprenons notre exemple du jeu de survie, si l'action se passe en temps réel, ce raisonnement à un coût en temps qu'il faut considérer. Ici la réactivité est une propriété importante, un lapin repérant un loup ne peut pas perdre son temps à émettre des hypothèses sur les positions des fleurs : son raisonnement doit être entièrement consacré à sa fuite. Pour donner à nos agents cette réactivité, nous utilisons les modules de réflexes.

Ces modules sont grandement inspirés des modules réactifs de [Costantini, 2010]. Si ils sont similaires de par leur forme, ils diffèrent cependant de par l'utilisation que nous en faisons. Un agent aura un ensemble de modules de réflexes qui seront vérifiés à chaque nouvelle observation. Si l'observation déclenche un réflexe, le raisonnement passe directement à la phase d'action.

3.2.7 Réflexes

Définition 17 (Module de réflexes) *Un module de réflexes est constitué d'un ensemble de règles qui peuvent avoir pour conclusion une action et de contraintes définissant les conditions nécessaires à l'activation du réflexe. Ces modules sont établis à la création de l'agent et ne changent pas au fil du temps. Ces réflexes doivent être vérifiés à chaque nouvelle observation.*

Exemple 15 *Un module de réflexes d'un lapin concernant la fuite :*

fuir.
:- not observation(position(loup,X,Y)).

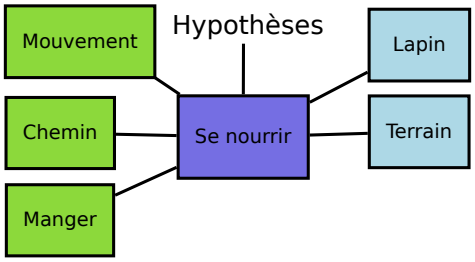

FIGURE 3.14 – Un module de buts combiné avec trois modules de règles, deux modules d'observations et des hypothèses.

Exemple 16 *Un module de réflexes d'un loup concernant l'attaque :*

action(tuer(X)) :- type(X,lapin), a_portée(X).
:- not a_portée(X).

L'exemple 15 montre un module de réflexes produisant un nouveau fait. Lorsque le lapin repère un loup ce module génère le fait *fuir* et le lapin passe directement à la génération des actions. Considérant ce nouveau fait il préférera les actions assurant sa fuite. Le module de l'exemple 16 génère une action : quand un lapin est à portée, l'action *tuer* devient un réflexe. Ici le loup passe directement d'une nouvelle observation à une action bien définie, alors que le lapin évite simplement la phase d'abduction.

Nos agents sont à présent dotés de modules leur permettant de réagir rapidement aux stimuli de l'environnement. Avec les modules de métaconnaissance un agent peut adapter son comportement en fonction de la situation. Les deux premières phases du raisonnement, à savoir l'acquisition et l'abduction, sont assurées par nos modules ASP. Reste donc la phase d'action à qui nous consacrons également un module spécifique : le module de buts.

3.2.8 Buts

Définition 18 (Module de buts) *Un module de buts est constitué d'un ensemble de règles qui ont pour conclusion une action. Ces modules sont établis à la création de l'agent et ne changent pas au fil du temps. Un module de but contient des commandes que l'agent peut exécuter sur son environnement.*

Exemple 17 *Un module de but d'un loup lui permettant de se nourrir :*

manger(P) :- proie(P), à_portée(P), mort(P).
aller_en(X,Y) :- proie(P), position(P,X,Y), peut_atteindre(X,Y).

Les modules de buts représentent le savoir de l'agent sur ses possibilités d'interaction avec l'environnement. Chaque module de buts est dédié à l'accomplissement d'un objectif et définie des actions qui permettent de l'atteindre. Un module de buts doit être combiné avec des modules de règles et d'observations

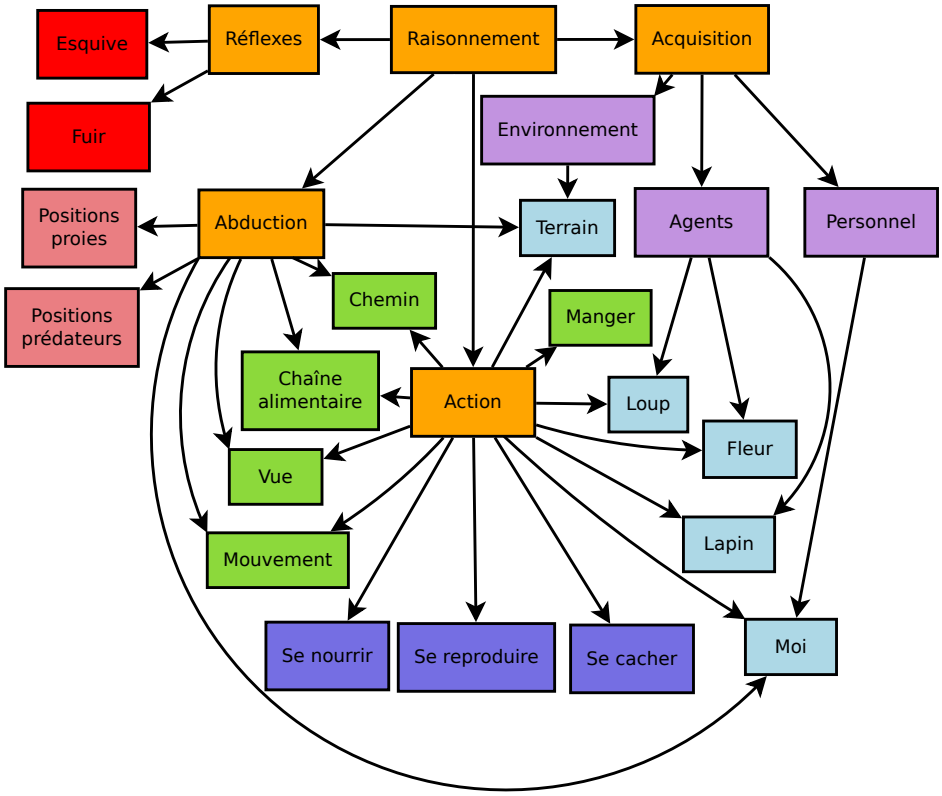

FIGURE 3.15 – Organisation des connaissances d'un lapin sous forme de graphe de module ASP.

afin de générer les actions possibles. Ces modules peuvent également prendre en considérations les hypothèses émises lors de la phase d'abduction.

La figure 3.14 représente un module de buts dont l'objectif est de nourrir l'agent. Supposons que notre agent soit un loup, ce module produira alors des actions permettant de s'approcher, de tuer et de se repaître d'une proie. Pour générer ces actions, notre module à besoin des observations concernant les autres types d'agents et des règles définissant ces trois actions. Comme expliquer dans la section consacrée aux modules d'abductions, les hypothèses serviront ici à déterminer l'emplacement d'une proie perdue de vue afin de continuer la poursuite.

Avec ces sept modules nous pouvons modéliser les connaissances de nos agents. La figure 3.15 représente les connaissances de nos lapin sous forme d'un graphe de modules. Le raisonnement est organisé par un réseaux de module de méta-connaissances. Nous avons donc la partie dédiée à l'acquisition des observations en haut à droite du graphe. La partie supérieure gauche quand à elle, représente les réflexes. Juste en dessous de celle-ci nous retrouvons les modules dédiés à l'abduction. Enfin on trouve en bas du graphe la partie consacrée aux actions sur l'environnement avec les modules de buts.

Chapitre 4

Framework

Afin d'utiliser la méthode que nous proposons, nous avons implémenté un framework. Celui-ci fait office d'interface entre l'agent et ses connaissances. L'intérêt de cet outil est que l'ensemble du raisonnement est effectué du côté ASP, en effet notre framework se contente d'exécuter les commandes qu'il récupère dans les answer sets. Pour calculer ces answer sets nous utilisons clingo, qui comprend le solver ASP clasp dont nous avons déjà discuté précédemment et le grounder gringo. La sortie de clingo est redirigée dans un fichier texte qui est ensuite analysé afin de récupérer les answer sets.

4.1 Combinaison de module

La première fonctionnalité de notre framework est la combinaison de modules ASP. Formellement un module ASP est un ensemble de règles et combiner plusieurs de ces modules résulte en l'union de ces ensembles de règles. Une combinaison de modules ASP est donc un programme ASP et les answer sets d'une combinaison de modules sont celles de ce programme. Les modules de métaconnaissance contiennent des règles concluant sur l'utilisation d'autres modules. C'est la conclusion de ces règles qui se retrouve dans les answer sets sous la forme de mots clefs, qui sont ensuite capturés et interprétés par notre framework. Le raisonnement s'effectue donc par étapes, chacune retournant la combinaison de modules à utiliser pour la prochaine. Le cycle de raisonnement s'interrompt lorsque la combinaison de modules n'a pas d'answer set ou que la liste des modules à utiliser est vide.

Dans l'exemple de la figure 3.15, la combinaison du module *Raisonnement* et des observations courantes, produira un answer set spécifiant que l'on doit utiliser le module *Acquisition* dans la prochaine itération du raisonnement. Le module *Acquisition* retournera ensuite les modules d'acquisition à utiliser en fonctions du type des observations à traiter. Pour effectuer cette tâche, notre framework capture la commande *utiliser_module(Module)* dans les answer set et récupère ainsi les noms des modules à utiliser. Une fois l'answer set entièrement analysé il exécute clingo sur l'ensemble des modules à combiner.

4.2 Observations dynamiques

Pour raisonner sur la mise à jour de sa mémoire, un agent utilisera ses modules d'acquisition. Les modules d'acquisition retourneront des commandes spécifiant l'ajout ou le retrait d'observations. Ici notre framework capture les commandes *ajouter(Module, Fait)* et *retirer(Module, Fait)*. Une fois le nom du module récupéré, il s'agit alors d'ajouter ou de retirer le fait du module, cela consiste simplement en l'ajout ou la suppression d'une chaîne de caractère dans un fichier texte.

Afin d'optimiser les mises à jour, nous conservons les modules d'observations dans un buffer mémoire et réservons une commande spécifique à l'utilisation de ces modules : *utiliser_observation(Module)*. Dans un soucis de commodité d'écriture, nous capturons également la commande *remplacer(Module, Ancien_fait, Nouveau_fait)* qui permet de remplacer un fait par un autre, ce qui revient à supprimer le premier et ajouter le second.

4.3 Raffinement d'hypothèses

Formellement, un answer set est un ensemble minimal de faits, cependant ceux retournés par clingo ne le sont pas. Cela peut notamment poser problème lors de la phase d'abduction, en effet une hypothèse qui n'est pas minimale peut contenir des faits qui ne sont impliqués dans aucune règles expliquant nos observations. Lors de l'utilisation de modules d'abduction, notre framework récupère dans un premier temps les hypothèses puis les raffines de façon à conserver uniquement celles qui sont minimales.

4.4 Actions

Une action est une commande exécutable par l'agent, dans notre modélisation ces commandes apparraissent en conclusions des règles des modules de buts. Notre framework récupére une commande dans les answer sets : *action(Commande)*. L'argument de cette commande en est également une, mais qui elle, sera exécutée par l'agent sur son environnement. L'envoie des commandes d'actions est la dernière phase du raisonnement de nos agents. Pour résumer, dans un premier temps le framework récupère les informations glanées par les senseurs de l'agent. Ensuite il les combine avec un premier module ASP, dans l'exemple de la figure 3.15 il s'agira du module *Raisonnement*. Il exécutera ensuite les différentes commandes extraites des answer sets jusqu'à récupérer les actions que l'agent doit exécuter. Enfin, ces dernières seront envoyées à la partie système de l'agent qui se chargera du reste.

4.5 Algorithme

L'algorithme 1 est une version très simplifiée de celui sur lequel est basé notre framework. Nous avons donc en entrée une combinaison de modules et en sortie un ensemble d'actions que l'agent peut exécuter. Nous commençons par récupérer les answer sets de notre combinaison de modules en utilisant clingo. De ces answer sets nous tirons les différentes commandes dédiées à notre framework.

Nous récupérons dans un premier temps la combinaison de modules à utilisée dans le prochain cycle de raisonnement. Ensuite les commandes de mise à jour des modules d'observations sont extraites et exécutées. Enfin nous récupérons également les différentes actions que notre agent peut effectuer. Une fois que les mises à jour des observations sont faites et que les actions sont stockées, le cycle reprend avec la nouvelle combinaison de modules. Enfin, lorsqu'il n'y a plus de modules à combiner, nous renvoyons l'ensemble des actions afin que l'agent les exécute.

Algorithm 1 Raisonnement

```
1: INPUT : M un ensemble de modules ASP
2: OUTPUT : A un ensemble d'actions

3: A un ensemble d'actions
4: AS un answer set

5: A ← ∅
6: AS ← ∅

7: repeat
8:    // Récupération d'une answer set de la combinaison de modules
9:    AS ← clingo(M)
10:   M ← ∅

11:   // Extraction des commandes du framework
12:   for chaque faits F de AS do
13:      if F == "utiliser_module(Module)" then
14:         ajouter Module à M
15:      end if
16:      if F == "ajouter(Module,Fait)" then
17:         ajouter Fait dans le module Module
18:      end if
19:      if F == "retirer(Module,Fait)" then
20:         retirer Fait du module Module
21:      end if
22:      if F == "action(Action)" then
23:         ajouter Action à A
24:      end if
25:   end for
26: until M == ∅ // Tant qu'il y a une combinaison de modules à utiliser

27: return A
```

Chapitre 5

Implémentation

Afin de mieux cerner les problèmes liés aux SMA en environnement dynamique, nous avons développé une application jouet basée sur notre exemple du jeu de survie. Au fil de notre étude, cette application nous à permis d'appréhender concrètement les difficultés qu'implique le raisonnement en milieu dynamique. Nous avons affiné notre méthode et notre framework en regard de cette application. Chaque loup, lapin et fleur est un agent qui raisonne selon notre méthode des modules ASP, par le biais de notre framework. L'action se déroule sur un plateau de jeu divisé en cases, l'action se joue en tour par tour. Les agents sont activés les uns après les autres et ne peuvent effectuer qu'un nombre limité d'actions pendant leur tour. Chaque action consomme une certaine quantité de point d'actions qui sont recouvrés au début de chaque tour et dont la quantité est fixée selon que l'agent soit un loup, un lapin ou une fleur.

5.1 Déplacements

Les mouvements s'effectuent case par case : un agent ne peut bouger que sur une case adjacente à celle où il se situe. Les déplacements sont limités par l'environnement et par la distance parcourue. L'eau, les arbres et autres rochers sont autant d'obstacles que nos agents ne peuvent franchir. La distance qu'un agent peut parcourir dépend de son genre : un loup pourra parcourir une plus longue distance qu'un lapin et les fleurs ne peuvent pas se déplacer. Chaque déplacement consomme des point d'actions, ce qui permet entre autre de limiter la distance parcourue. Sur une même case nous pouvons avoir plusieurs loups, plusieurs lapins et plusieurs fleurs.

5.2 Observations

Les spécificités du terrain ont non seulement une incidence sur les mouvements des agents, mais également sur leur perception visuelle. Les arbres et les rochers limiteront la vue de nos agents et comme pour les déplacements, la distance est également un facteur limitant. L'eau et les autres agents ne sont pas des facteurs de limitation de la vue. Il est du ressort du système de récupérer les observations et donc de considérer le champs de vision de l'agent. Cependant l'agent peut savoir qu'il ne voit pas une case, en se basant sur sa distance de

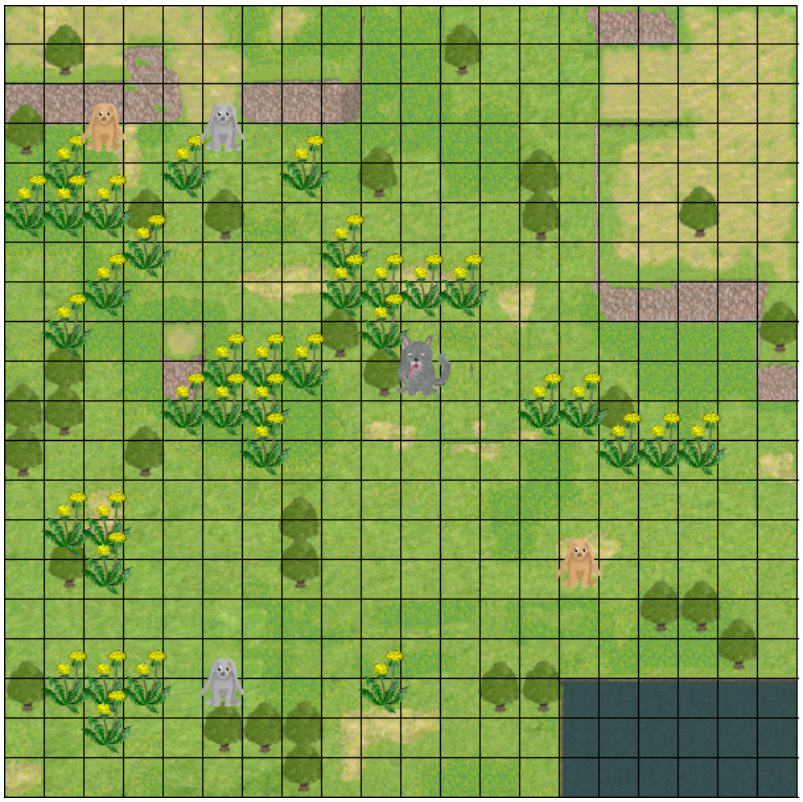

FIGURE 5.1 – Implémentation du jeu de survie impliquant des loups, des lapins et des fleurs.

vue et ses observations. Ces observations peuvent concernées la position d'un agent ou d'un élément du décors : un arbre, de l'eau...

5.3 Nourriture

Nos agent peuvent se nourrir de leurs proies lorsque ils se trouvent sur la même case que celle-ci. Un agent ne peut manger qu'une proie à la fois et cela consomme des points d'action. Un loup devra d'abord parvenir à tuer un lapin avant de pouvoir s'en repaître. Dans notre exemple, se nourrir n'est pas nécessaire à la survie de l'agent mais constitue néanmoins un but pour les loups et les lapins.

5.4 Reproduction

Seuls les lapins et les fleurs sont capables de procréer. Afin de se reproduire, un agent à besoin d'accumuler un certain nombre de points d'énergie. Alors qu'une fleur accumulera de l'énergie à chaque tour, un lapin devra se nourrir de ces fleurs afin d'en accumuler. Lorsque que la quantité de point d'énergie est suffisante, un clone de l'agent apparaît à proximité de la case de son géniteur.

FIGURE 5.2 – Représentation du champs de vision d'un loup, ici notre animal ne perçoit pas ce qui se passe sur les cases grisées.

Les agents ainsi produits n'héritent de leur parent que la théorie commune de leur groupe, qui peut être considérée ici comme l'instinct de l'espèce.

5.5 Survie

L'unique moyen pour qu'un agent meurt est qu'il serve de repas à un de ses prédateurs. Les lapins tenteront d'éviter de tomber sous les griffes des loups, alors que les fleurs se contenterons de se multiplier. Un loup ne peut pas mourir, mais ne peut également pas se reproduire. Les loups servent à réguler la population des lapins qui eux temporisent la propagation des fleurs. Nous avons pu observer plusieurs fin possibles, la première est la victoire totale des loups. Si les lapins mangent toutes les fleurs de la carte, ils ne pourront plus se reproduire et finiront par être tous dévorés par les loups. La seconde fin est la conquête de la carte par les fleurs. Si les loups tuent tout les lapins, les fleurs se reproduiront jusqu'a envahir la totalité du plateau de jeu. Enfin il est possible d'arriver dans un état stable dans lequel la population des fleurs arrive à se renouveler et où la population de lapins arrive à compenser les victimes faites par les loups.

Conclusion

Nous avons proposer une modélisation basée sur les modules ASP afin de représenter les connaissances d'un agent et implémenter un framework permettant à un agent de raisonner via cette représentation. Le premier atout de cette modélisation est sa simplicité de représentation et d'implémentation. Notre représentation modulaire permet de facilité la modélisation des comportements complexes d'un agent. Cette représentation est également dédiée aux environnements dynamiques : les modules d'acquisition permettent une gestion non monotone des observations.

Un des avantages de l'utilisation de ces modules ASP est l'amélioration de la survivabilité des agents. La réactivité, la flexibilité et l'adaptabilité d'un agent peuvent être améliorées par l'utilisation de nos modules. La réactivité est assurée par l'utilisation des modules de réflexes qui permettent à nos agents de réagir rapidement en cas d'urgence. Une organisation intelligente des modules de métaconnaissance permet d'utiliser de façon optimale les connaissances et de gagner ainsi du temps de raisonnement. L'abduction permet aux agents d'émettre des hypothèses et de raisonner dans l'incertain. Afin de débloquer certaine situation, un agent a besoin de flexibilité dans son raisonnement : ne pas se limiter au savoir certain, utiliser des inférences non monotones afin d'obtenir au moins une solution même si celle-ci n'est pas parfaite. Les modules de métaconnaissance et de buts permettent à un agent d'adapter son comportement à la situation. Les premiers assurent l'adaptabilité du raisonnement et les seconds celui des actions.

Enfin, il reste des améliorations à faire du côté du framework, entre autres sur la gestion des hypothèses. Nos expérimentations sur le jeu de survie n'ont pas touchées à tout les aspects présentés ici. Les phases d'acquisition et d'action sont elles, totalement fonctionnelles, ce qui n'est par contre pas le cas de la phase d'abduction et des réflexes. Ces deux éléments ont été implémentés, cependant sur la fin le temps nous aura manqué pour réellement les tester.

Conclusion Personnelle

Ces travaux sont le fruit de mon stage de deuxième année de master, que j'ai effectué dans le laboratoire du professeur Katsumi Inoue au National Institute of Informatics (NII) de Tokyo. Ce stage fût une expérience très enrichissante, tant du point de vue scientifique, que culturel et personnel. Vivre à Tokyo m'a permis de découvrir différents aspects de la culture nippone. Le cadre de travail était très stimulant, chaque jour j'avait la chance d'être au contact d'étudiants et de chercheurs passionnés par leurs recherches. J'ai eu la chance d'assister

et de participer à plusieurs séminaires qui m'ont permis d'enrichir ma culture scientifique.

C'est à Kanazawa que le premier séminaire a eu lieu, quelques jours après mon arrivée au Japon, il avait pour sujet les systèmes multi-agents et la Bio-informatique. Hormis ces séminaires, nous avions également des meetings chaque semaines, durant lesquels moi et les autres membres du laboratoire présentions l'avancée de nos travaux. Outre la possibilité d'avoir l'avis des autres sur notre travail, ces séances permettaient de nous entraîner pour les présentations du prochain séminaire. C'est donc à Kyoto que le second séminaire a eu lieu, il a été l'occasion pour moi de présenter pour la première fois mon travail à d'autres chercheurs. Objet principal de ce séminaire : la Bio-informatique et la complexité. Bien que mon travail ne concernait pas directement la biologie, j'ai pu présenter ma méthode au travers de l'exemple du jeu de survie, qui se rapproche assez d'une simulation d'évolution de population. Il m'a également servit d'exemple pour le troisième séminaire qui s'est tenu à Kobe au mois de mai. SAT, ASP et CSP étaient les sujets de ce séminaire. Outre l'opportunité de présenter mon travail, ces séminaires ont surtout été l'occasion de découvrir les travaux de chercheurs et d'étudiants de tout les horizons. Une présentation durait généralement vingt minutes et était suivie de quelques questions.

S'adonner à cet exercice m'a permis d'apprendre à clarifier mon travail afin de le rendre compréhensible aux autres, chaque préparation de présentation me permettait de prendre du recul sur mes recherches. Les retours de ceux qui assistaient à mes présentations m'ont permis d'orienter et d'affiner la méthode que je vous présente ici. Les séminaires m'ont permis d'en apprendre plus sur la culture japonaise, chaque sortie était une occasion de découvrir les spécialités culinaires et les lieux culturels de la région. Effectuer mon stage au NII m'a permis de rencontrer et de travailler avec des gens du monde entier : des japonais bien sûr, mais également des étudiants et chercheurs venus de Chine, du Vietnam, de Thaïlande, des États-Unis, d'Italie, ainsi que beaucoup de compatriotes français.

Ces six mois de stage à Tokyo m'ont beaucoup apporté, cela à conforté mon attrait pour la recherche et mon envie de continuer en doctorat.

Bibliographie

[Baral et al., 2006] Baral, C., Anwar, S., and Dzifcak, J. (2006). Macros, macro calls and use of ensembles in modular answer set programming. In *AAAI Spring Symposium : Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, pages 1–9.

[Baral and Gelfond, 2011] Baral, C. and Gelfond, G. (2011). On representing actions in multi-agent domains. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, pages 213–232.

[Baral et al., 2010] Baral, C., Gelfond, G., Son, T. C., and Pontelli, E. (2010). Using answer set programming to model multi-agent scenarios involving agents' knowledge about other's knowledge. In *AAMAS*, pages 259–266.

[Bourgne, 2008] Bourgne, G. (2008). *Hypotheses refinement and propagation with communication constraints.* PhD thesis, University Paris IX Dauphine.

[Bourgne et al., 2010a] Bourgne, G., Inoue, K., and Maudet, N. (2010a). Abduction of distributed theories through local interactions. In *ECAI*, pages 901–906.

[Bourgne et al., 2010b] Bourgne, G., Inoue, K., and Maudet, N. (2010b). Towards efficient multi-agent abduction protocols. In *LADS*, pages 19–38.

[Costantini, 2009] Costantini, S. (2009). Integrating answer set modules into agent programs. In *LPNMR*, pages 613–615.

[Costantini, 2010] Costantini, S. (2010). Answer set modules for logical agents. In *Datalog*, pages 37–58.

[Faber and Woltran, 2011] Faber, W. and Woltran, S. (2011). Manifold answer-set programs and their applications. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, pages 44–63.

[Fisher et al., 2007] Fisher, M., Bordini, R. H., Hirsch, B., and Torroni, P. (2007). Computational logics and agents : A road map of current technologies and future trends. *Computational Intelligence*, 23(1) :61–91.

[Gebser et al., 2011] Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., and Schneider, M. T. (2011). Potassco : The potsdam answer set solving collection. *AI Commun.*, 24(2) :107–124.

[Hatano and Hirayama, 2011] Hatano, D. and Hirayama, K. (2011). Dynamic sat with decision change costs : Formalization and solutions. In *IJCAI*, pages 560–565.

[Kowalski and Sadri, 1999] Kowalski, R. A. and Sadri, F. (1999). From logic programming towards multi-agent systems. *Ann. Math. Artif. Intell.*, 25(3-4) :391–419.

[Kowalski and Sadri, 2011] Kowalski, R. A. and Sadri, F. (2011). Abductive logic programming agents with destructive databases. *Ann. Math. Artif. Intell.*, 62(1-2) :129–158.

[Nakamura et al., 2000] Nakamura, M., Baral, C., and Bjäreland, M. (2000). Maintainability : A weaker stabilizability like notion for high level control. In *AAAI/IAAI*, pages 62–67.

[Nicolas and Duval, 2001] Nicolas, P. and Duval, B. (2001). Representation of incomplete knowledge by induction of default theories. In *LPNMR*, pages 160–172.

[Nieuwenborgh et al., 2006] Nieuwenborgh, D. V., Vos, M. D., Heymans, S., and Vermeir, D. (2006). Hierarchical decision making in multi-agent systems using answer set programming. In *CLIMA*, pages 20–40.

[Sakama et al., 2011] Sakama, C., Son, T. C., and Pontelli, E. (2011). A logical formulation for negotiation among dishonest agents. In *IJCAI*, pages 1069–1074.

[Tran and Baral, 2009] Tran, N. and Baral, C. (2009). Hypothesizing about signaling networks. *J. Applied Logic*, 7(3) :253–274.