Modular Reasoning in Multi-Agent Systems Using Meta-Knowledge and Answer Set Programming

Tony Ribeiro — Department of Informatics, The Graduate University for Advanced Studies (Sokendai), Tokyo, Japan
keijurosan@gmail.com

Katsumi Inoue — National Institute of Informatics, Tokyo, Japan
inoue@nii.ac.jp, http://research.nii.ac.jp/~inoue/

Gauvain Bourgne — Université Pierre et Marie Curie, Paris, France
gauvain.bourgne@gmail.com

keywords: multi-agents systems, answer set programming, meta-knowledge, modularity, dynamic environment

Summary

We are concerned with multi-agent systems (MAS) in dynamic environment, and focus on knowledge representation and reasoning of such systems.
In dynamic environment, an agent needs to be able to manage its knowledge according to context changes.
To achieve this goal, an agent has to adapt his beliefs and behaviour with respect to the current state of the world.
In this work, we define a method to design agents' knowledge and reason efficiently in dynamic contexts.
For this purpose, we use the expressibility of answer set programming (ASP) and propose a method based on combinations of modules that are represented in ASP.
Using meta-knowledge on these combinations, we can easily realise dynamic behaviour and meta-reasoning.
We propose an algorithm to combine modules and implement the framework in an example of multi-agent systems in a dynamic world.
Using experimental results, we show that modular knowledge can be used to optimize reasoning time.

1. INTRODUCTION AND MOTIVATION

In recent years, research about reasoning of multi-agent systems is very active. Some research works are concerned about knowledge representation like [Baral and Gelfond, 2011], where authors discuss about representing actions in dynamic environment. There are also interesting works on meta-knowledge like [Baral et al., 2010], where agents have knowledge about other knowledge. In [Sakama et al., 2011], using this kind of meta-knowledge allows agents to lie to manipulate their fellows. Other works proposed agent knowledge representation like [Rao and Georgeff, 1995], where agent knowledge is divide into beliefs, decisions and intentions (BDI). In [Kowalski and Sadri, 1999] authors are concerned about representing agent reasoning via logic programming and proposed an agent cycle based on three phases: observe, think, act. Regarding reasoning, there are works like [Costantini, 2010], [Costantini, 2009] and [Baral et al., 2006] where authors are interested in modularity.

In this work, our interest is about representing agent reasoning in dynamic environment. Our framework is based on modular knowledge representation. Agent knowledge is divided into different modules which are combined for reasoning. Using knowledge on modules combinations, an agent can adapt his reasoning regarding the situation. The purpose is to reduce reasoning search space by avoiding a part of agent knowledge.

First, we consider the knowledge of an agent in two parts:

- K: non-revisable knowledge which consists of:
  - C: the common theory
  - O: the current observations
- B: revisable knowledge which consists of:
  - M: past observations

In such environment, an agent has to adapt to the evolution of the world to achieve his goals. Agent's knowledge has to be updated regarding world changes by adding new observations and updating old ones.

Let C_T, O_T, M_T be C, O and M at time step T and $\diamond$ a belief update operator.

- $C_0 = C$, $O_0 = \emptyset$, $M_0 = \emptyset$
- $C_{T+1} = C$, $O_{T+1} = O$, $M_{T+1} = M_T \diamond O_{T+1}$

T is the common background knowledge of all agent of a multi-agent system, it is considered as certain and not revisable. A common theory can also be limited to a group of agent which does not contain all the system. Observations are informations retrieved from the environment, it represents the current state of the world. If we consider that sensors are perfect then a current observation is not revisable. At time step 0, the knowledge of an agent is his common theory. This knowledge is extended by adding observations and its consequences at each new time step.

B is agent's beliefs, it represents informations assumed to be true by the agent and which are revisable. In a dynamic environment, if current observations are certain, it is not the case of past ones. Here we assumes that past observations are correct until new ones prove the contrary. At time step $T+1$, agent past observations M_{T+1} is the knowledge base resulting from updating M_T by O_{T+1} using the knowledge update operator $\diamond$.

Figure 1: A MAS in a dynamic environment where an agent is a wolf, a rabbit or a flower: Wolves eat rabbits and rabbits eat flowers.

For example, let's suppose that an agent a sees another agent b at position p at time T. Now, at $T+1$, a sees neither the position p nor the agent b, then a can assume that b is always at position p at $T+1$. When a sees again the position p or the agent b, he will update M if needed.

To make our work more understandable we will follow an intuitive example along our propositions: a survival game which represent a MAS in a dynamic environment. In this game there are three groups of agents: wolves, rabbits and flowers. Each kind of agent have specific goals and behaviours. To be simple, wolves eat rabbits and rabbits eat flowers.

Wolves have only one goal: to feed. To reach this goal they have to catch and eat rabbits. A wolf can be in two situations: a prey is in sight or not. If there is no rabbit in the sight range of a wolf, the predator has to explore his environment to find one. When a prey is spotted, a wolf will try to perform a sneaky approach if it is not spotted himself, otherwise the predator will rush on his target. To summarise, a wolf have three behaviours: exploration, approach and attack.

Rabbits have two goals : to feed and not to be eaten. On a first hand, a rabbit has to eat flowers and on an other hand it has to escape from wolf. Like a wolf, if no prey is in sight, a rabbit needs to explore its environment, but it can find preys and predators. If a wolf is spotted, surviving is more important than feeding, therefore a rabbit has to hide or run away. To sum up, a rabbit has four behaviours: explore, feed, hide and run away. In this paper, examples and experiments focus on rabbit agent regarding knowledge representation and reasoning.

Flowers are passive agents: the environment has no effect on their behaviour. The only goal of a flower is to reproduce. A flower produces regularly a new one after a certain amount of time. Knowledge of a flower does not change regarding time. Flowers are independent, it does not care about other agents.

2. ANSWER SET PROGRAMMING

Answer set programming which is a specification and an implementation language, is very suitable to represent agent knowledge and reasoning. ASP is a form of declarative programming based on the stable model semantics of logic programming. In this section, we briefly introduce ASP paradigm and semantics.

We recapitulate the basic elements of ASP in the following. An answer set program is a finite set of rules of the form

$$a_0 \text{ :- } a_1,\ \ldots,\ a_m,\ \mathit{not}\ a_{m+1},\ \ldots,\ \mathit{not}\ a_n \quad (1)$$

where $n \geq m \geq 0$, a_0 is a propositional atom or $\bot$, all $a_1, \ldots, a_n$ are propositional atoms and the symbol "*not*" denotes default negation. If $a_0 = \bot$, then Rule (1) is a constraint (in which case a_0 is usually omitted). The intuitive reading of a rule of form (1) is that whenever $a_1, \ldots, a_m$ are known to be true and there is no evidence for any of the default negated atoms $a_{m+1}, \ldots, a_n$ to be true, then a_0 has to be true as well. Note that $\bot$ can never become true.

In the ASP paradigm, the search of solution consist to compute answer sets of ASP program. An answer set for a program is defined following Gelfond and Lifschitz [Gelfond and Lifschitz, 1988]. An interpretation I is a finite set of propositional atoms. An atom a is true under I if a $\in I$, and false otherwise. A rule r of form (1) is true under I if $\{a1, \ldots, a_m\} \cup I$ and $\{a_{m+1}, \ldots, a_n\} \cup I = \emptyset$ implies $a_0 \in I$. Interpretation I is a model of a program P if each rule $r \in P$ is true under I. Finally, I is an answer set of P if I is a subset-minimal model of P^I, where P^I is defined as the program that results from P by deleting all rules that contain a default negated atom from I, and deleting all default negated atoms from the remaining rules. Programs can yield no answer set, one answer set, or many answer sets. To compute answer set of an ASP program, we run an ASP solver.

EXAMPLE 1. *An ASP program composed of one fact, three rules and one constraint :*

```
rain.
stay :- not go_out.
go_out :- not stay.
wet :- rain, go_out, not umbrella.
:- wet.
```

In example 1, the set of facts {*rain*, *stay*} is an answer set of the ASP program, but {*rain*, *go_out*, *wet*} is not, because it contains *wet* which is not consistent with the constraint *:- wet.*

Many research work use ASP to represent knowledge and reasoning such as [Baral et al., 2010] or [Nieuwenborgh et al., 2006]. In the first one, the authors use it to represent agent knowledge about other agents knowledge. In the second one, they focus on the flexibility of reasoning by introducing soft constraints: constraints which can be violate in some case. Other works like [Costantini, 2010] design ASP methods to improve some specific property of agent reasoning. In this work the author focuses on reactivity by using ASP modules where constraints are used as actions trigger. In [Costantini, 2009], the author discuss about integration of ASP modules into agent reasoning. There are also works like [Baral et al., 2006] about modular ASP and [Faber and Woltran, 2011] where they discuss about possible extensions of the paradigm.

3. ASP MODULES

An ASP module is an ASP program which have a specific form and a specific use. The first advantage of these modules is their simplicity: a module is a little program which represents a specific knowledge. We can have a module which contains observations about surroundings, an other one to define what is a prey and a module dedicated to path computing. By combining these three modules, an agent can compute all paths to surrounding preys. To design agent knowledge we respect a module typology to represent background knowledge, observations and meta-knowledge.

3.1 Typology

Following previous distinction of agent knowledge we define a typology to represent common theory C and observations memory M, respectively by *theory modules* and *observations modules*. In the following figures, we represent *theory modules* by plain rectangles and *observations modules* by dot rectangles.

DEFINITION 1 (THEORY MODULE). *A theory module is a set of rules which represent knowledge about a specific domain. The set of all theory modules of an agent represent his background knowledge: the common theory C.*

EXAMPLE 2. *A theory module of a rabbit about movement and one about actions where A is an agent, P a position, D and N are integer:*

Move

```
can_reach(A,P,D/N) :- distance(A,P,D), nb_action(A,N).
can_reach(A,P) :- can_reach(A,P,0).
```

eat

```
food(me,flower).
food(wolf,me).

can_eat(A,Prey,D) :- food(A,Prey), position(Prey,P),
distance(A,P,D).
can_eat :- position(me,P), food(me,Prey), position(Prey,P).

will_eat(A,Prey,D) :- food(A,Prey), move(A,P),
distance(Prey,P,D).
will_eat(A,Prey,D) :- food(Predator,Prey), position(A,P1),
move(Prey,P2), distance(P1,P2,D).
```

Action

```
nb_action(wolf,4).
nb_action(rabbit,3).

0{action(move(P))}1 :- can_reach(me,P).
0{action(eat)}1 :- can_eat.
0{action(hide)}1 :- can_hide.
:- action(A1), action(A2), A1 != A2.

move :- action(move(P)).
wolf_range :- position(me,P), can_reach(wolf,P).
wolf_close :- position(me,P), distance(wolf,P,D), D <= 2.

danger_after(S) :- move(me,P), can_reach(wolf,P,S).
danger :- wolf_range, not move, not hide.
danger :- action(hide), wolf_close.
danger :- danger_after(0).
safe :- not danger.
```

Example 2 shows three theory modules, the first one represents knowledge about movement and the second one represents knowledge about actions. To simplify our examples, *distance/3* is supposed given and his third argument is the

distance between an agent A and a position P or two positions $P1$, $P2$. Predicate *nb_action/2* specifies the number of actions that an agent can perform in one time step. The first module can be used by a rabbit to compute his movement possibility and the ones of his predators. It specify the number of step an agent needs to reach a position P. The second module specify the number of actions an agent needs currently to eat a prey. It also consider it regarding predator and prey movement. The third module defines the three actions that a rabbit can perform: move, eat and hide. It specifies that only one action can be performed at once. This module contain knowledge about consequence of actions, here *danger/0* represent the possibility to be eaten by a wolf at next time step. Predicate *danger_after/1* represent the number of time step needed to be in danger. Let suppose that we have a fourth module similar to *eat* one but regarding hiding possibilities. By combining these four modules with his observations a rabbit will produce all actions he can perform and their consequences regarding safety, eating and hiding.

Definition 2 (Observations module). *An observations module is a set of facts which represents related observations. The content of such module changes regarding time. The set of all observations modules of an agent represents his past observations M.*

Example 3. *An observations module of a rabbit about wolves positions and one about himself :*

Wolf

```
position(wolf,coord(0,3)).
position(wolf,coord(-1,2)).
position(wolf,coord(2,-4)).
```

Myself

```
me("Bugs Bunny").
position(me,coord(0,2)).
nb_action(me,3). hidden.
```

Example 3 shows two *observations modules*, the first one regroup wolves positions and the second one contains agent personal observations. Using *theory* and *observations* modules we can represent agent background knowledge C and past observations M. The figure 2 represents the knowledge of a rabbit by a set of observations modules and theory modules. Regarding how we combined these modules with *theory* ones, we can produce different kinds of knowledge. Here, by combining modules *Myself*, *Field* and *Move* a rabbit will produce all his possibility of movements. The combination {*Rabbit*, *Field*, *Move*} will produce all possible movements that other rabbits can perform. By combining {*Myself*, *Field*, *Flower*, *Move*, *Eat*, *Action*} a rabbit will produce all possible actions to feed he can perform now and their influence on the number of steps needed to eat each flower. We can see here a first advantage of knowledge modularity: a module can be used for multiple purposes. For example, module *Move* can be used to reason about both agent's, fellows' and predators' movements.

We have presented different ways to use agent knowledge via module combination. But these combinations are also a kind of knowledge and it can be very useful to an agent to know it. For an agent, it is meta-knowledge on his knowledge, more precisely in this case it is knowledge about the use of his knowledge. This meta-knowledge is a part of the common theory of an agent and could be represented by theory modules. But to clarify the design of agent knowledge we dedicated a new kind of modules to represent it: *meta-knowledge modules*. We define two kinds of these modules: *combination modules* and *decision modules*. The first ones represent a unique module combination. The second ones use theory and observations to make decision on which module combination to use for reasoning. To distinguish *decision modules* and *combination modules*, we represent them respectively by plain and dot circles. Modules represented by dot form only contain facts and plain ones can contain rules and constraints.

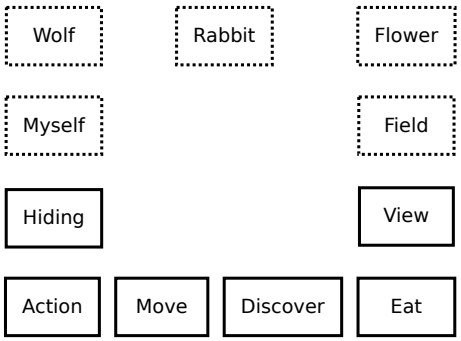

Figure 2: Knowledge base of a rabbit divided into observations modules (dot ones) and theory modules (plain ones).

Definition 3 (Combination module). *A combination module is a meta-knowledge module which represent a unique ASP modules combination. This combination can include theory, observations and combination modules.*

Example 4. *A combination module which represent the module combination to reason about movements:*

Movements

```
#include("Move").
#include("Field").
```

Meta-knowledge about modules combinations can be used to divide agent reasoning into multiple part to clarify its representation, like in figure 3. Furthermore, meta-knowledge can be used to optimize reasoning by reducing the use of knowledge. Using *decision modules* we can exploit this representation to control agent reasoning and optimize it.

Definition 4 (Decision module). *A decision module is a meta-knowledge module which defines the conditions to use ASP modules combinations. A graph of such module can represent agent reasoning and behaviours.*

Example 5. *A decision module which describes the* survive *behaviour of a rabbit:*

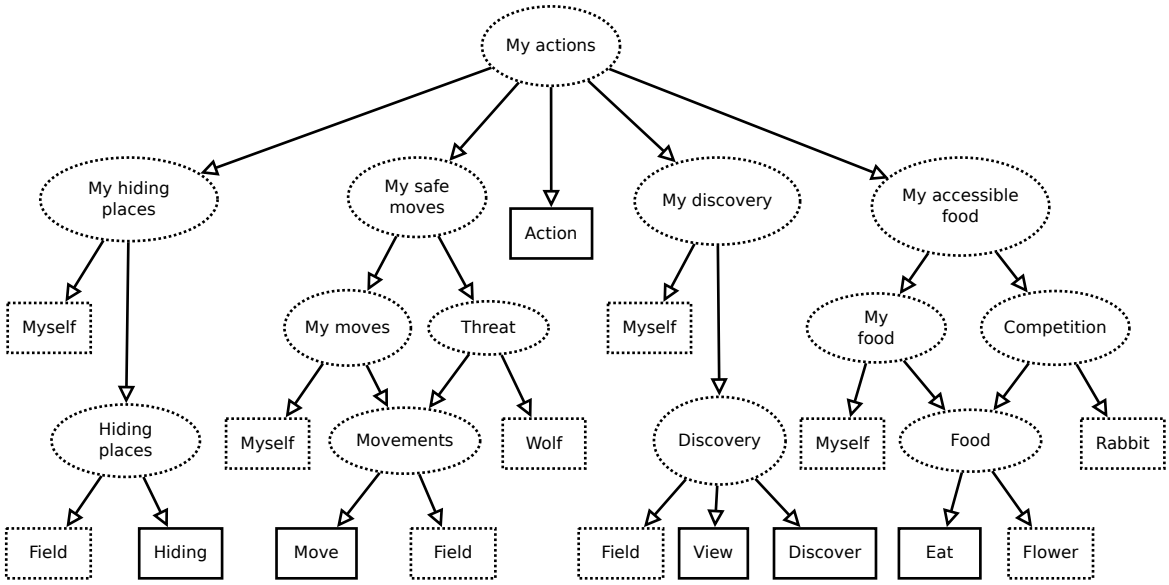

Figure 3: Rabbit meta-knowledge representation by *combination modules* (dot circle). Arrows represent module inclusion: keyword *include*(Module).

Survive

```
#include("Wolf").

predator :- position(wolf,Position).
next("Hunted") :- predator.
next("Hunter") :- not predator.
```

Decision modules are used to make decision on what it is useful to reason about regarding the situation. Figure 4 shows the introduction of *decision* modules into knowledge representation of figure 3. Here we replace *My actions* module by a *decision modules* graph based on rabbit behaviours. Example 5, show the content of *Survive* module which represent the main behaviour of a rabbit. A rabbit combine this *decision module* with observations about wolves to decide if it is hunted or not.

EXAMPLE 6. *A decision module which describes the* hunted *behaviour of a rabbit:*

Hunted

```
#include("Wolf").
#include("Myself").

hide :- hidden, position(me,P),
distance(wolf,P,D), D <= 2.
next("Hide") :- hide.
next("Run away") :- not hide.
```

If a predator is present, a rabbit will use *Hunted* module, shows in example 6, to choose regarding wolves distance, if it has to stay hidden or run away to find a place to hide.

EXAMPLE 7. *A decision module which describes the* hunter *behaviour of a rabbit:*

Hunter

```
#include("Flower").

prey :- position(flower,P).
next("Feed") :- prey.
next("Explore") :- not prey.
```

If there is no predator, a rabbit will use *Hunter* module, shows in example 7. Regarding flower presence, it will reason about how to eat one or find one. These *decision modules* are used to reduce the quantity of knowledge used for reasoning. In these example, when a rabbit reason about how to run away, it does not reason about eating or exploration and vice versa. Using modular representation we can avoid a part of agent knowledge to focus reasoning on current priority.

4. FRAMEWORK

The previous section shows how ASP modules can represent agent common theory, observations, meta-knowledge and reasoning. To use this representation we define a reasoning architecture and a framework to use it in agent applications. As shown in figure 5, we divide agent reasoning into a cycle of three phases : *acquisition-analyse-action*. Our framework is similar from the one of [Kowalski and Sadri, 1999] which is *observe-think-act*. One difference here, is that the acquisition phase is not limited to the storage of new observations.

Acquisition phase: during this phase, agent past observation M is updated to make it consistent with new observations O. Old ones are modified or deleted and new ones are then stored in corresponding observations modules. For example, if a rabbit sees a wolf at position P at time step T, the information *position(wolf,P)* is then stored in module *Wolf*. Now at T+1 the rabbit sees position P but no wolf is at P, therefore the rabbit has to remove the fact *position(wolf,P)* from module *Wolf*. An agent can also make direct deduc-

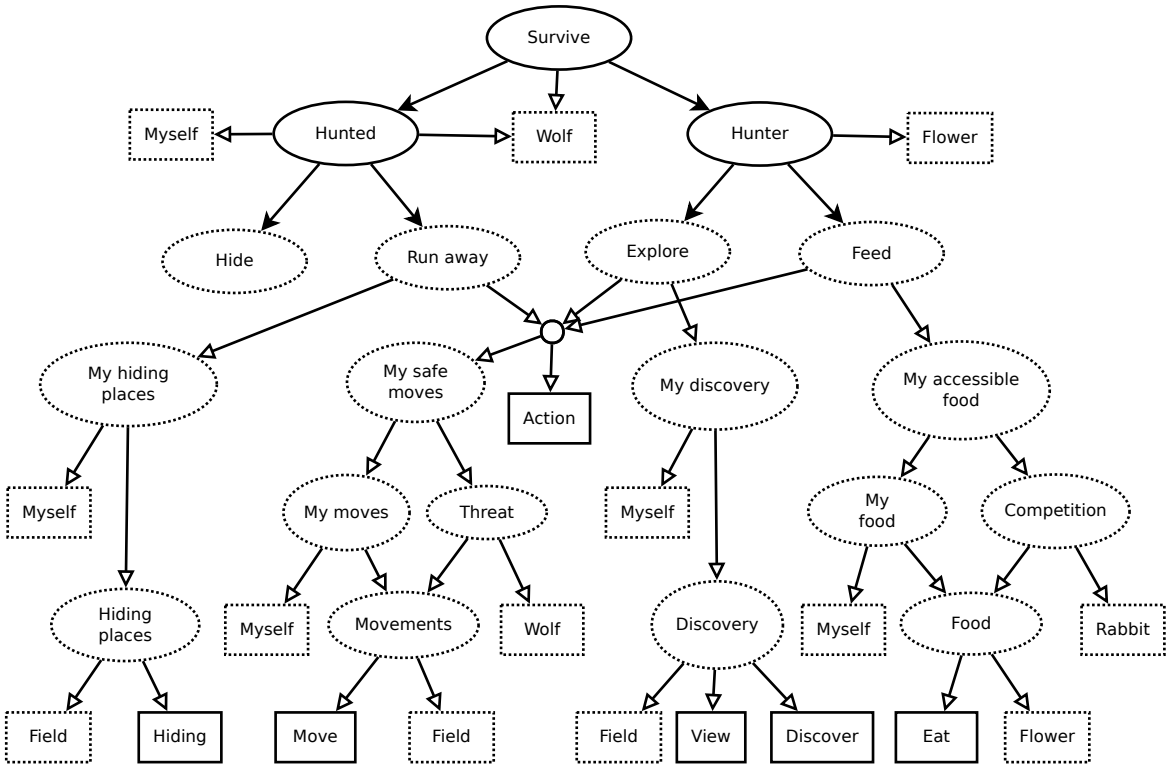

Figure 4: Representation of rabbit reasoning by *decision modules* (plain circle). Plain arrows represent the choice of next reasoning step: keyword *next*(Module).

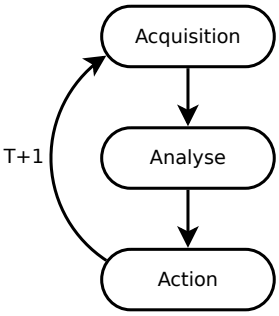

Figure 5: Agent reasoning cycle.

tion from observations: after a move, an agent will update his position regarding his previous one and the movement he performed. This phase is named acquisition, because the agent acquires observations by deducing their direct consequences on his past observations. In this framework, a part of agent common theory C is dedicated to updating past observations M.

Analyse phase: having acquired observations, an agent will reason on what he can do regarding the current state of the world. The purpose of this reasoning phase is to deduce what actions are possible to perform and their consequences. Here we use an agent reasoning architecture based on ASP modules where meta-knowledge modules are used for reasoning like in figure 4.

Here rabbit's reasoning is represented by an ASP modules graph. A node represents an ASP module and arrows represents module combinations. Edges with plain arrows represent reasoning decisions and empty ones represent module inclusion. In this representation, *decision modules* represent reasoning steps. In this example, the analyse phase will start by using *Survive* module with wolves observations. By combining these modules the rabbit will decide if he is hunted or has to hunt. When he is hunted, he will use *Hunted* module to decide if he has to run away or hide. Let's suppose that a wolf is too close to hide, then he will choose to run away, otherwise he will reason about where he can hide. In the case where there is no wolf, the rabbit will use *Hunter* module with flowers observations. If there is no flower he will use *Explore* module and when flowers are spotted he will use *Feed* module.

Action phase: now the agent has to choose which actions to perform, it means choosing an answer set which allows him to approach or reach his goals. To make this choice we can use constraints and use some modules to compute scores to rank answer sets. Here, module *Eat* computes the number of actions a predator has to perform to eat a prey. It can be used by a rabbit to rank movement actions to go far away from a wolf or to approach and eat a flower. After choosing the best answer set, corresponding actions are performed and the cycle continues with the acquisition of the real effect of these actions on the environment.

Let n be a reasoning step, regarding *observe-think-act* framework, ours can be sum up as:

- Acquisition phase: $observe, think, update$.

- Analyse phase: $think_0, decide_0, \ldots, decide_n, think_n$.
- Action phase: $decide, act$.

4.1 Implementation

In this section we focus on the deduction phase of the framework and discuss about it implementation based on algorithm 1. The input of this algorithm is a set of modules and the output is a set of answer sets which contains actions that an agent can perform on its environment. The algorithm start by computing the combination of input modules and retrieves its answer sets by using an ASP solver. Answer sets are exploited in a deep first exploration by extracting keywords "*next*" which define the combination of modules to use in the next reasoning step. The cycle continue until there is no more module to combine, then the algorithm returns a set of answer sets. The algorithm always finishes when an agent uses a module graph which is acyclic like in figure 4.

Let's take again the example of the figure 4 and let's suppose that the rabbit just finish to acquires new observations. This agent will start his analyse phase by running the algorithm with the set of modules {*Survive*, *Wolf*}. This combination will return one answer set which contains wolfs observations and two literals: *predator* and *next("Hunted")*. Interpreting keywords *next*, the cycle continue by calling recursively the algorithm with {*Hunted*, *wolf*, *Myself*} as input, because *Hunted* module specify to include *wolf* and *Myself*. This new combination produce one answer set which contains observations of module *Wolf* and *Myself* and one literals: *next("Run away")*. After extracting "*include*" keyword from *Run away* and from all *combination module* it include, the last combination can be sum up by {*Hiding*, *Move*, *Myself*, *Field*, *Wolf*, *Action*}. Finally, the algorithm returns a set of answer sets which contains a movement action and its consequences regarding safety and hiding possibilities.

In this example, only a part of agent knowledge is used for reasoning. *Meta-knowledge modules Hide*, *Hunter*, *Explore*, *Feed*, discovery ones and food ones are not used. Flowers and rabbits observations, as theory modules *Eat*, *View* and *Discover*, are ignored. The same knowledge can be represented by only one ASP program, but in this case, all knowledge is used for reasoning. Regarding this monolithic representation, modularity allows to reduce reasoning search space.

Algorithm 1 Combine

```
1: INPUT : <M> M a set of ASP modules
2: OUTPUT : AS a set of answer set

3: AS a set of answer set

4: AS ← ASPsolver(M)

5: for each answer set S of AS do
6:    M ← ∅

7:    // Extract keywords
8:    for every literal L of S do
9:       if L = "next(Module)" then
10:         M ← M ∪ {Module}
11:      end if
12:   end for

13:   if M ≠ ∅ then
14:      return combine(M)
15:   end if
16: end for

17: return AS
```

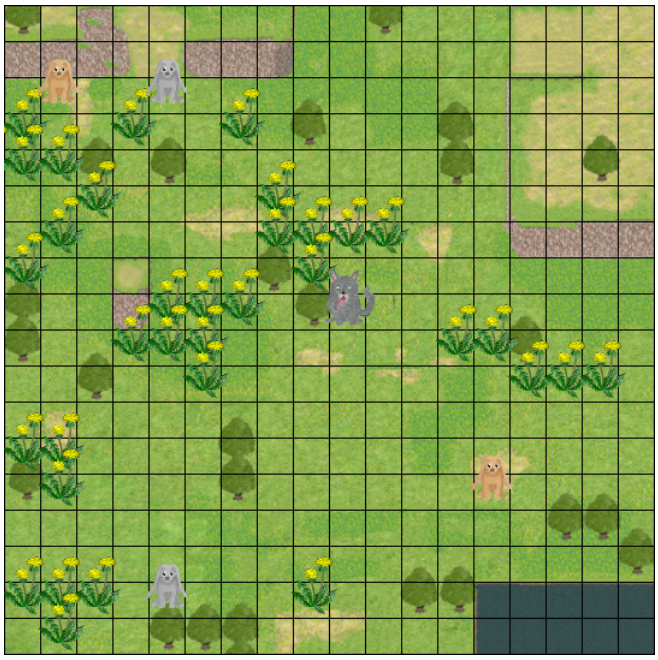

Figure 6: Experimental application based on the survival game example.

5. EXPERIMENTS

To evaluate our work, we implemented the algorithm 1 in a toy application based on the survival game example. In this application, the environment is a grid where each agent are located on a single square. Agents act turn by turn and have limited number of actions per turn. These actions can be: move to a square, eat an agent or hide. To eat his prey, a predator has to be on the same square, it is the same rule for hiding. These experimental results focus on the reasoning time of a single rabbit regarding a specific scenario. To evaluate our method we compare modular and monolithic reasoning time. The first one used algorithm 1 and exploit *meta-knowledge modules* to reduce the quantity of knowledge to use for reasoning. The second one directly runs the solver *clingo* on the entire knowledge base: all theory and observations modules.

5.1 Context

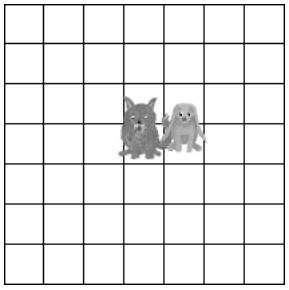

Figure 7: Environment 1, a rabbit and a wolf without flowers and no place to hide.

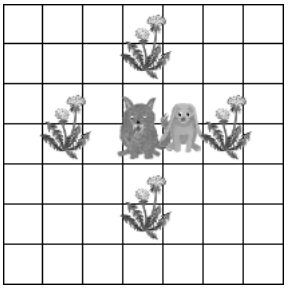

Figure 8: Environment 2, a rabbit, a wolf, 4 flowers and 4 hiding places (grass).

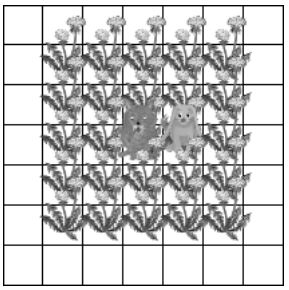

Figure 9: Environment 3, a rabbit and a wolf in a field of 25 flowers and 25 hiding places (same position as flower).

In these experiments, the rabbit's observations encompass the entire map, this agent knows what is on each square: that includes the position of wolves, flowers and hiding place. Here modular knowledge of the rabbit is the one of figure 4 and modules are almost the same as the ones presented in previous example. Monolithic representation always uses the entire knowledge base which correspond to the module combination {*Myself*, *Wolf*, *Rabbit*, *Flower*, *Field*, *View*, *Eat*, *Discover*, *Move*, *Action*, *Hiding*}.

For these experiments, we use 4 different environments represented by figure 7, 8 and 9. The fourth environment is equivalent to the third one but ten times bigger regarding the number of squares, flowers and hiding places. Our evaluation is based on eight scenarios: for each environment, we evaluate the rabbit reasoning time regarding wolf presence. In scenarios where there is a wolf, reasoning about flowers or exploration is useless because hiding has a bigger priority. When there is no wolf, reasoning about hiding or exploration is no use because eating flowers is more important. In these experiments, we show that modular reasoning can be used to optimize reasoning by avoiding a part of agent knowledge which is not necessary regarding the current situation. In all scenarios, monolithic reasoning considers movements regarding hiding and eating possibilities regardless the presence of a wolf. Modular reasoning considers movements to hide only when there is a wolf and considers eating only when there is no predator. It only consider exploration in the first scenario where there is no flower and when there is no wolf.

The figure 7 represents the first scenario where there is no flower and no hiding place. In this scenario the rabbit only considers the movement he can perform without being caught by his predator at the next turn. This scenario is used to evaluate the time we lose when modularity reasoning does not reduce search space. The figure 8 represents the second scenario where there are 4 flowers and 4 hiding places. In this scenario, when there is no wolf the time lost by multiple steps of reasoning is almost compensated by the reduction of search space. The figure 9 represents the third scenario: agents are in a field of 25 flowers and 25 hiding places which are on the same square as flowers. The fourth scenario is a ten times bigger extension of the third one where the rabbit is in a field of 250 flowers and 250 hiding places. In these two last scenarios, the number of flowers and hiding places is sufficient for modular representation to reduce reasoning time.

5.2 Results

These experiments focus on rabbit reasoning time regarding modular and monolithic representation on 8 scenarios. This time corresponds to the run of algorithm 1, which includes *clingo* solving and answer set parsing. The time we consider is the user one and to obtain a relevant value we compute the average running time on 1000 runs for each method on each scenario. These 16 sequences of 1000 runs tests have been performed consequently in the same running conditions on a Intel Core 2 Duo P8400 2.26GHz CPU. These experimental results are summarise in Table 5.2.

In scenario 1, figure 7, when there is a wolf, the monolithic reasoning run is about 0.135 seconds and the modular one is about 0.139 seconds. When there is no wolf, monolithic reasoning run is about 0.107 seconds and modular one is about 0.119 seconds. Here, the use of multiple reasoning steps by modular representation causes more time for reasoning because it does not reduce the search space. In scenario 2, figure 8, in the case of the presence of a wolf, run time of the monolithic reasoning up to 0.181 seconds because it now considers 4 flowers and 4 hiding places, and modular reasoning time just up to 0.169 seconds because it does not consider flowers. If there is no wolf, monolithic and modular reasoning time is almost the same. The time lost with reasoning about hiding in the monolithic representation is equivalent to the one lost by the multiple step reasoning. In this environment, modular reasoning is a little bit more efficient than monolithic one. For scenario 3, figure 9, modular reasoning is more efficient in both cases. The proportion of flowers and hiding places is sufficient to make search space reduction interesting. When reasoning about hiding is the priority, modular reasoning takes 30% less time than monolithic one and 18% less time when reasoning about eating. As the number of observations increases, the modular rep-

Environment	Flower	Hiding place	Wolf	Representation	Time average	T-test	Runs
1	0	0	yes	monolithic modular	0.135 s 0.149 s	0	1000
			no	monolithic modular	0.107 s 0.119 s	0	
2	4	4	yes	monolithic modular	0.181 s **0.169 s**	0	
			no	monolithic modular	0.151 s 0.152 s	$2.115E-5$	
3	25	25	yes	monolithic modular	0.416 s **0.277 s**	0	
			no	monolithic modular	0.383 s **0.314 s**	0	
4	250	250	yes	monolithic modular	3.838 s **1.963 s**	0	
			no	monolithic modular	3.819 s **2.739 s**	0	

Table 1: Experimental results of rabbit reasoning time on 8 scenarios of 4 different environments. For each method it shows the reasoning time average of 1000 runs and T-test result.

resentation becomes more interesting: in scenario 4, when there is a wolf it takes 49% less time to compute hiding actions and 32% less time for reasoning about eating when there is no predator.

These experiments show that modular knowledge representation and knowledge about modules combinations can be used to optimize reasoning by reducing search space. Here we show that this representation can be more efficient than monolithic one when entire knowledge base is not necessary to solve every situation. Designing such reasoning pattern implies to find the balance between the time lost by multiple reasoning steps and search space reduction.

6. CONCLUSIONS AND OUTLOOK

We provide a method based on ASP modules to design agent knowledge and reasoning. This representation allows to intuitively implements dynamic behaviours via meta-reasoning. We also provide a framework which allows agent reasoning by module combination. Regarding an equivalent monolithic representation, a first improvement of our method is the reduction of reasoning search space. It also causes reduction of code size because modules are reusable for multiple purposes.

In this paper, agent reasoning is very directed by meta-knowledge modules, an interesting outlook will be to give the possibility to the agent to really choose which module combination he wants to use. Making agent able to build themselves a reasoning architecture like the one on figure **??** is also an interesting research topic. Learning can also concern module content, an agent could choose to create new observations modules for storing specific information. Modular knowledge representation provides interesting perspectives for meta-reasoning and learning.

7. REFERENCES

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

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.

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

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

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

Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080.

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

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.

Rao, A. S. and Georgeff, M. P. (1995). Bdi agents: From theory to practice. In *ICMAS*, pages 312–319.

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