Combining Answer Set Programs for Adaptive and Reactive Reasoning

Tony Ribeiro, Katsumi Inoue, Gauvain Bourgne

TPLP **13** (4-5): Online Supplement, July 2013.

Combining Answer Set Programs for Adaptive and Reactive Reasoning

TONY RIBEIRO
The Graduate University for Advanced Studies, Tokyo, Japan
(*e-mail:* tony_ribeiro@nii.ac.jp)

KATSUMI INOUE
National Institute of Informatics, Tokyo, Japan
(*e-mail:* inoue@nii.ac.jp)

GAUVAIN BOURGNE
University Pierre et Marie Curie, Paris, France
(*e-mail:* gauvain.bourgne@gmail.com)

submitted 10 April 2013; accepted 23 May 2013

Abstract

In this paper we present a method to improve efficiency of ASP based system in dynamic context. Our proposal is a reasoning framework based on combination of ASP modules. This framework can be seen as a tool to assist the development of ASP based systems. Our final goal is to provide a general design pattern to help construction of efficient ASP based reasoning systems. In this paper, we first define a method to divide the wide program into different kinds of modules. Then, we show how meta-knowledge on these decomposition can be used for meta-reasoning. We discuss the benefit of this modular reasoning method through an example of multi-agent systems in a dynamic world. Using this example, we show that this method can elegantly allow agents to realise dynamic behaviour via introspective meta-reasoning. Finally, through experimental results we show that the framework can be used to improve efficiency of ASP based system in terms of reactivity and adaptability.

KEYWORDS: Answer Set Programming, Developement Tool, Modular Logic Programming, Meta-Reasoning, Dynamic System

1 Introduction

Answer set programming (ASP) is a form of declarative programming that has been successively used in many knowledge representation and reasoning tasks (Niemelä 1999; Baral 2003; Nieuwenborgh et al. 2006; Fisher et al. 2007; Baral 2008). In ASP, a problem is represented by a logic program where the answer sets correspond to the solutions of the problem. Solving the problem is then reduced to computing stable models using answer set solvers like *clasp* (Gebser et al. 2007). This kind of modelling philosophy leads us to consider programs as monolithic entities. This fact is reflected in the answer set semantics itself: it has been originally defined for entire programs only (Gelfond and Lifschitz 1988). The indivisible nature of ASP programs is causing increasing difficulties

as program instances tend to grow in real applications. Soon, application areas such as bioinformatic, robotics and semantic web will imply to deal with huge program instances.

By contrast, modularity is widely used in many programming languages to manage the complexity of programs and their development process. It is usually easier to design a system as a set of many small interacting components rather than a huge monolithic system with obscure internal organisation. A modular structure is more suitable for implementation tasks that are devided by a team of developers. The possibility to re-use code is another interesting practical aspect of modular representation. To achieve similar advantages in ASP, previous works have been proposed to extend the ASP paradigm to incorporate modularity (Brogi et al. 1994; Oikarinen and Janhunen 2008). Other works already studied the consequences of dividing and combining knowledge in modular logic programming (Brogi et al. 1999; Brewka et al. 2011)

In this paper, contrary to these previous works, we do not intend to extend the ASP paradigm. Here, we propose a practical knowledge representation based on modular ASP and a reasoning framework based on this representation. Like (De Vos et al. 2012), our framework can be seen as a tool to assist the development of ASP based systems. Our final goal is to provide a general design pattern, i.e. a general guideline, to help the construction of efficient ASP based reasoning systems. The core idea of our reasoning framework is similar to (Monteiro and Porto 1989): like in their contextual logic programming, we define a reasoning system consisting of independent logic programs which can be combined together regarding context needs. In our representation, knowledge is divided into different modules which can be combined in multiple ways for reasoning. The novel point of our method is the use of meta-knowledge and meta-reasoning in the modular ASP context. Using meta-knowledge about different combinations of ASP modules, a system can perform introspective meta-reasoning to adapt its behaviour according to context change. It allows to represent reasoning by a hierarchy of modules organised like a tree where leaves represent elementary knowledge and internal nodes represent meta-knowledge. In that hierarchy, reasoning starts from the root node and is divided into multiple introspective meta-reasoning steps which define the behaviour of a system. The underlying idea is quite similar to the one of Minsky (Minsky 1991): *Is a mind composed of smaller and smaller minds, until the pieces become so small that they are no longer mind like?* To the best of our knowledge, there does not exist any previous work who proposed a reasoning method based on introspective meta-reasoning about ASP program combinations.

The purpose of this framework is to improve efficiency of ASP based systems in a dynamic context. In a dynamic environment, the adaptability and reactivity of a system has a major impact on its efficiency. Adaptability of a system is measured by its capacity to change its behaviour according to context change. Reactivity is the ability to respond quickly to environment stimuli. Adaptive reasoning can be divided in three phases (Kowalski and Sadri 1999). *Observe*: whenever a new event occurs, the system has to figure out "what is going on". For that, it needs to change his mind to accord his beliefs with new observations. *Think*: when the system is conscious of the current state of the world, he can reason about "what he can do" to achieve its tasks. *Act*: After reasoning about the consequences of its possible actions, the system can decide, "what he should do" according to the current context. Other works introduced the concept of context-aware reasoning (Loke 2004) to describe adaptive reasoning. During each reasoning phase, our

representation allows to exploit knowledge modularity and context-awareness to improve adaptability and reactivity of reasoning. In real-time applications, the ability to handle new observations on-line is a big concern regarding reactivity. Previous works like (Sadri and Toni 2006) already tackled it with the incorporation of belief update during reasoning. Dividing reasoning into multiple steps, our framework allows to handle such on-line consideration of new observations between two meta-reasoning steps. Our framework can also be used to perform introspective reasoning. Meta-reasoning can allow the system to perform introspection by providing an upper level of reasoning. To reduce reasoning time, meta-reasoning can be used to reason on "what is important to reason about". Combining such capacity with a modular knowledge representation, a reasoning system can enhance its reactivity in critical situations. Using meta-knowledge about modules, it can decide which part of its knowledge is really useful for reasoning in the current context. Reasoning time can then be reduced by avoiding non critical thinking.

To illustrate the interests of our work, we will use an intuitive game example where intelligent agents have to survive in a predator/prey survival game. In this game we consider three groups of agents: wolves, rabbits and flowers. In a nutshell, wolves eat rabbits and rabbits eat flowers. Rabbits have to feed and not to be eaten. Like a wolf, if no prey is in sight, a rabbit needs to explore its environment, but it can find predators as well as preys. If a wolf is spotted, surviving is more important than feeding and 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 the knowledge representation and reasoning of rabbit agents. Finally, we will discuss the benefit of our method using experimental results on this application.

2 Module Typology

From now on, we will use the term *ASP module* to designate a module within a modular ASP program. To make easier the design of a module hierarchy, an ASP module should be as simple as possible. A module should be a little program that represents a specific knowledge. For instance, we can have a module containing observations about surroundings, another one to define what is a prey and yet another dedicated to compute paths. Combining these three modules we compute all paths to surrounding preys. The simplicity of modules can improve the readability and the re-usability of the code.

To help the design of the knowledge hierarchy we propose a module typology to represent *background theory*, *observations* and *meta-knowledge* of the system. We use the term *background theory* to designate the initial knowledge of the system, i.e. a set of rules given by the programer. *Observations* are dynamic information about the environment of the system, i.e. a set of facts which represents the state of the world. Here, the *meta-knowledge* of the system is its knowledge about its own knowledge, i.e. knowledge of module combinations. In the following figures, we represent *theory modules* by plain rectangles and *observation modules* by dotted rectangles.

Definition 1 (*Theory module*)
A theory module is a set of rules representing knowledge about a specific domain. The set of all theory modules represents the background theory of the system.

Example 1
Some theory modules of a rabbit about eating, hiding and actions, where A is an agent,

P a position, D and N are integers:

Eat

```
%#extern position/2.
food(me,flower).
food(wolf,me).
...
can_eat(A,Prey) :- food(A,Prey),
position(A,P), position(Prey,P).
```

Hiding

```
%#extern position/2.
hiding_place(grass).
hiding_place(bush).
...
can_hide :- position(me,P),
position(Place,P), hiding_place(Place).
```

Action

```
%#extern can_reach/2, food/1 ...
%Generate action
0{action(move(P))}1 :- can_reach(me,P).
...
:- action(A1), action(A2), A1 != A2.
...
danger :- action(move(P)), food(Predator,me), can_reach(Predator,P).
```

Example 1 shows three theory modules, the first two represent knowledge about eating and hiding, the third represents knowledge about actions. The first module specifies if an agent can eat a prey. A rabbit can use it to consider what he can eat and if he can be eaten. The third module defines the three actions that a rabbit can perform: move, eat and hide, and specifies that atmost one action can be performed at once (using aggregate symbols $0\{\ldots\}1$ and a constraint). This module also contains knowledge about consequences of actions. Here, *danger*/*0* represents the possibility to be eaten by a wolf. Combining these modules with observations will allow to reason about which actions a rabbit can perform.

Definition 2 (*Observation module*)
An observation module is a set of facts which represents related observations. The content of such module changes regarding time. The set of all observation modules represents the beliefs of the system.

Example 2
An observation 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

```
position(me,coord(0,2)).
nb_move(me,3).
hidden.
```

Example 2 shows two *observations modules*, the first one regroups wolves positions and the second one contains observations about the system itself. Figure 1 represents the knowledge of a rabbit as a set of observations modules and theory modules. Depending on how we combine *observations* modules with *theory* ones, we can produce different kinds of knowledge. Here, by combining modules *Myself*, *Field* and *Move* a rabbit will determine all his possible movements. In this combination, by replacing module *Myself* by module *Rabbit*/*Wolf* the agent will reason about rabbit/wolves movements. This is an example of code re-use and factorisation of knowledge: a module can be used for multiple purposes via different combinations. Here, module *Move* can be used to reason about the agent's, fellows' and predators' movements.

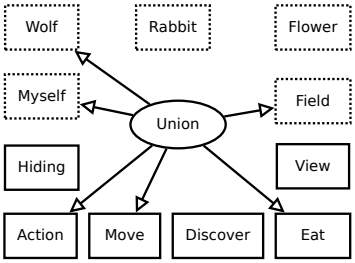

Fig. 1. Knowledge base of a rabbit divided into observations modules (dotted ones) and theory modules (plain ones). The module combination *Union* computes move and eat actions regarding danger.

2.1 Meta-knowledge and Meta-reasoning

Module combinations themselves can also be considered as a kind of knowledge. The idea is that we can represent module combinations inside the system knowledge by using meta-knowledge. This meta-knowledge is a part of the background theory. But to clarify the knowledge representation we dedicated a new kind of modules to represent it: *meta-knowledge modules*. We distinguish two kinds of *meta-knowledge modules*: *combination modules* and *decision modules*. The first ones represent a unique module combination as a list of modules. These modules contain specific commands which specify module inclusion. Like (De Vos et al. 2012), we exploit the comments syntax of ASP to hide our commands from ASP solvers. But contrary to this work, our commands have an impact on ASP solving. The second type of *meta-knowledge modules* use theory and observations to make decisions on which module combinations to use for reasoning. To differentiate *decision modules* and *combination modules*, we represent them respectively by plain and dotted circles.

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

Example 3
Combination modules which represents module combinations about movements:

Movements
```
%#include "Move".
%#include "Field".
```

My moves
```
%#include "Movements".
%#include "Myself".
```

My safe moves
```
%#include "My moves".
%#include "Threat".
```

Meta-knowledge about modules combinations can be used to divide agent reasoning into multiple parts to clarify its representation, like in Figure 2. Furthermore, meta-knowledge can be used to improve efficiency of reasoning by reducing the use of knowledge. Using *decision modules* we can exploit this representation to perform meta-reasoning.

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

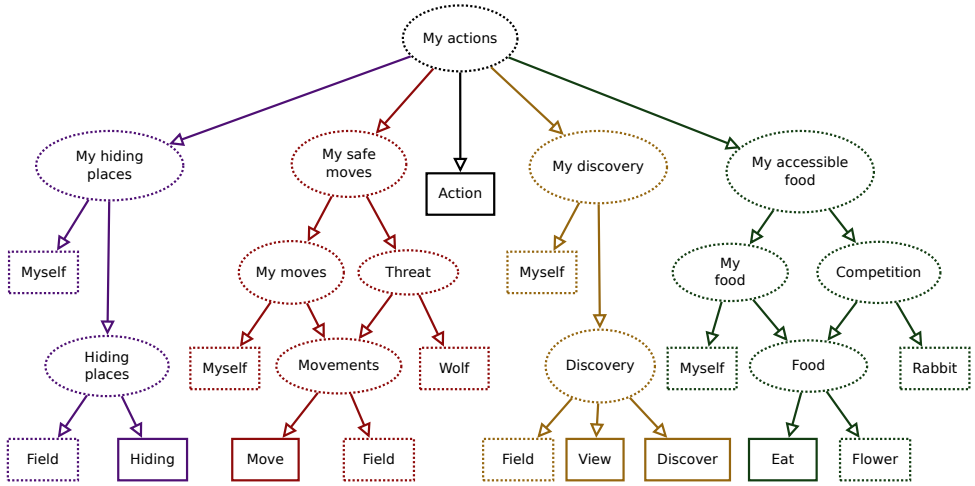

Fig. 2. Representation of the meta-knowledge of a rabbit as a tree of combination modules (dot circles) where arrows represent module inclusion. The knowledge based is divided into two principal part corresponding to rabbit goals: eat (right) and not be eaten (left). From the left, we have the knowledge about hiding, safe movements (including how to run away), exploration and eating which correspond to the four behaviours of the agent.

Decision modules are used to decide which modules should be used to reason depending on the situation. We propose to use these modules to divide reasoning into multiple phase of meta-reasoning. Here we apply the concept developed by (Minsky 1991) in the Society of Mind: starting from an abstract level, step-by-step, the reasoning becomes more and more elementary. To clarify the reasoning representation, decision modules should be as simple as possible. One simple way to use them is to represent a binary meta-reasoning choice. In that case the module will represents a simple rule "IF *conditions* THEN *combination*" or "IF *condition* THEN *next decision*". A similar approach to design *reactives* ASP modules have been studied in (Costantini 2010). In this work, the author proposes simple ASP modules which represent a unique rule "IF condition THEN action" and where constraints are used as trigger. To ensure reactivity of meta-reasoning phases we apply this idea to decision modules. But the novelty in our approach is that we consider module combination or decision in place of actions and use constraints to terminate reasoning. In our examples, we use the predicate *next*/*1*, where the argument is a module name, as a keyword to represent the fact that the next meta-reasoning step has to use this module.

If decision modules respect this pattern, then meta-reasoning can be represented by a decision tree where leaves are combination modules. Figure 3 shows the introduction of decision modules into the knowledge representation of Figure 2. Here the module *My actions* is replaced by a decision modules tree based on rabbit behaviours. Example 4, shows the content of *Survive* module which represents the main behaviour of a rabbit. A rabbit combines it with observations about wolves to decide if he is hunted or not.

Example 4

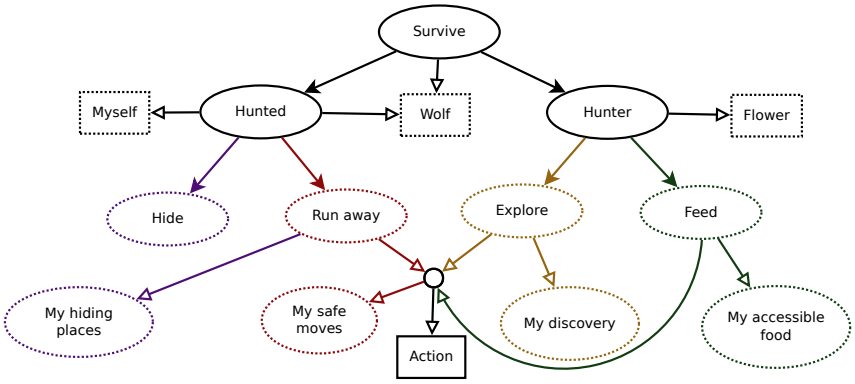

Fig. 3. Representation of rabbit meta-reasoning using decision modules (plain circles). Plain arrows represent the choice of next meta-reasoning step.

Decision modules describing the *survive*/*hunted*/*hunter* behaviour of a rabbit :

Survive	Hunted	Hunter
%#include "Wolf". %#extern position/2. wolf :- position(wolf,Position). **next**("Hunted") :- wolf. **next**("Hunter") :- not wolf.	%#include "Wolf", "Myself". %#extern position/2. hide :- hidden, position(me,P), distance(wolf,P,D), D >= 2. **next**("Hide") :- hide. **next**("Run away") :- not hide.	%#include "Flower". %#extern position/2. prey :- position(flower,P). **next**("Feed") :- prey. **next**("Explore") :- not prey.

If a predator is present, a rabbit will use the *Hunted* module, shown in example 4. According to wolves distance, this module decides if the rabbit should hide or run away. If there is no predator, a rabbit will use *Hunter* module, shown in Example 4. Depending on the presence or not of a visible flower, a rabbit will reason about how to go eat one or search for one. These decision modules are used to reduce the quantity of knowledge used for reasoning. In these example, when a rabbit reasons about how to run away, he does not reason about eating or exploration and vice versa. This representation allows us to avoid a part of knowledge and focus reasoning on the current priority.

3 Introspective meta-reasoning in practice

Algorithm 1 describe how meta-reasoning can be performed using our knowledge representation. The input of this algorithm is a set of modules and the output is a set of actions that the system can perform on its environment. The algorithm starts by computing the combination of input modules and retrieves its answer sets by using an ASP solver. Answer sets are exploited in a depth-first search by extracting keywords "*next*" which define the combination of modules to use in the next reasoning step. The cycle continues until there is no more module to combine, then the algorithm returns a set of actions. The algorithm always finishes when the ASP module graph is acyclic.

Let us take the example of Figure 3 and suppose that the rabbit has just finished to acquire new observations. Reasoning will start by running the algorithm with the set of modules $\{Survive, Wolf\}$ as input. Suppose that there is one observation about a wolf. This combination will return one answer set which contains observations about wolves and two literals: *predator* and *next("Hunted")*. Interpreting keywords *next*, the cycle continue by calling recursively the algorithm with $\{Hunted, wolf, Myself\}$ as input This

Algorithm 1 Combine

```
INPUT : M a set of ASP modules
OUTPUT : A a set of actions
// Solve the union of ASP modules
AS ← all answer set of M
// Explore answer sets
for each answer set S of AS do
    M ← ∅
    for every literal L of S do // Extract keywords
        if L = "next(Module)" then // Extract modules
            M ← M ∪ {Module}
            Recursively add into M modules specified to be included with Module
        if L = "action(Action)" then // Extract actions
            A ← A ∪ Action
    if M ≠ ∅ then // Continue reasoning
        combine(M)
return A
```

new combination gives one answer set which contains observations of module *Wolf* and *Myself* and one literal: *next("Run away")*. After extracting "*include*" keywords from *Run away* and recursively from all combination module it includes, the last combination will be {*Hiding*, *Move*, *Myself*, *Field*, *Wolf*, *Action*}. Finally, the algorithm returns a set of answer sets, each of them containing 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*, as the ones about discovery and food are not used. Flowers and rabbits observations, as theory modules *Eat*, *View* and *Discover*, are ignored. Here, combination of knowledge modularity and meta-reasoning allows us to reduce reasoning search space. The algorithm we propose here captures actions and returns them as output. One can easily adapt this algorithm to application needs, and make it capture other predicates or even return every deduced fact. Again, in this algorithm we do not keep the deduced facts from a meta-reasoning step to another one. To ensure reactivity of the system it could be interesting to reduce redundancy of reasoning. Keeping knowledge produced by previous meta-reasoning is one way to do it. Exploration of answer sets can also allows to backtrack in meta-reasoning. In this paper, for the sake of simplicity, we will not discuss into detail all possibilities offered by this algorithm. Our intention here is only to provide a simple way to exploit our knowledge representation in practice.

4 Experiments

To evaluate our work, we implemented Algorithm 1 in a toy application based on the survival game example. In this application, the environment is a grid where each agent is located on a single square. Figure 4 represents an environment of 13x18 squares with four agents: three rabbits and one wolf. Agents act turn by turn and can perform a limited number of actions per turn. These actions can be: move to a square, eat an agent or hide into bush/grass. To eat his prey, a predator has to be on the same square, it is the same rule for hiding. Following experiments focus on the reasoning time of a single rabbit regarding a specific scenario. To evaluate our method we compare reasoning time with

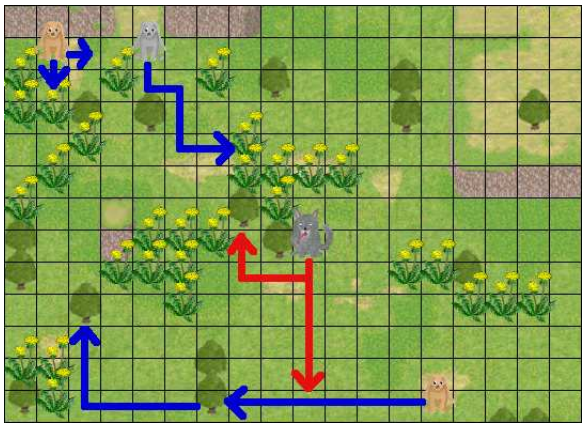

Fig. 4. Experimental application based on the survival game example. Arrows represent some movements that agents are considering in order to explore/feed/hide/escape.

and without meta-reasoning. Without meta-reasoning we directly run the solver *clingo* on the entire knowledge base: all theory and observations modules. With meta-reasoning, we use algorithm 1 and exploit decision modules to reduce reasoning search space.

4.1 Context

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 3 and modules are almost the same as the ones presented in previous examples. Without meta-reasoning we always use the entire knowledge base: the module combination {*Myself*, *Wolf*, *Rabbit*, *Flower*, *Field*, *View*, *Eat*, *Discover*, *Move*, *Action*, *Hiding*}.

For these experiments, the environment is a grid of size 25x25 similar to the one in figure 4. Our evaluation is based on 7 scenarios where we evaluate rabbit reasoning time regarding wolf presence and the number of flower/bush. The number of observations about flower and bush is used in these experiment to measure the quantity of knowledge about of the environment. In these experiments, we show that our knowledge representation can be used to improve reasoning by avoiding a part of agent knowledge which is not necessary in the current situation. In all scenarios, without meta-reasoning we consider actions corresponding to movements, hiding, exploring and eating possibilities regardless of the presence of a wolf. Using meta-reasoning we consider movement to hide only when there is a wolf and considers eating only when there is no predator.

In the first scenario, there is no flower and no hiding place. In the other scenarios, there are respectively {50,100,150,200,250,300} flowers and hiding places. In these scenarios, the number of flowers and hiding places is sufficient for modular representation to reduce reasoning time. The reduction of the reasoning time is mainly explained by the reduction of the quantity of grounding. Using the decision modules tree, meta-reasoning can only consider a part of the observations (either flowers or bush).

4.2 Results

Experimental results are shown in Figure 5, where x axis represent the quantity of knowledge in terms of observations about flower/bush and y axis represent time in

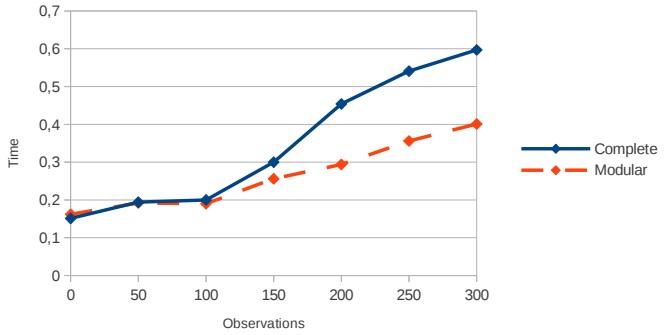

Fig. 5. Average reasoning time of a rabbit regarding the quantity of observations: the dot line correspond to reasoning with decision modules and the plain line without.

seconds. The two curves correspond to rabbit average reasoning time regarding meta-reasoning and complete representation for each scenario. 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 reasoning method for each scenario. These 28 sequences of 1000 run tests have been performed sequentially in the same running conditions on a Intel Core I7-3610QM, 2.3GHz CPU and 4GB of RAM.

As the number of observations increases, the meta-reasoning representation becomes more interesting than using the entire knowledge base, as shown in the diagram of Figure 5. In the last scenario (300 observations), when there is a wolf it takes around 45% less time to compute hiding actions and 30% less time for reasoning about eating when there is no predator. These experiments show that knowledge about modules combinations can be used to improve efficiency of reasoning by reducing search space. Here we show that this representation can be more efficient than a naïve complete reasoning 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. Our intention here is to show one intuitive way to improve reasoning time in ASP based reasoning systems: avoiding irrelevant knowledge/reasoning according to context needs. We can point out the interest of these possibility regarding system reactivity: for emergency contexts, one can design reasoning so that only critical knowledge is considered to reason and react as quickly as possible.

5 Conclusions and Outlook

We presented a knowledge representation and a reasoning framework based on combination of ASP modules. This method can be used to help the conception of efficient ASP based reasoning systems in a dynamic context. Using meta-knowledge, this method allows us to intuitively perform meta-reasoning to reduce reasoning search space. This form of meta-reasoning can be used to improve adaptability and reactivity of reasoning. An interesting outlook will be to incorporate more freedom in meta-reasoning: for example, give the possibility to make original module combinations. Making systems able to acquire meta-knowledge so that they can learn/build/modify themselves a reasoning architecture like the one on Figure 3 belongs to our future work. The next step of our research is to incorporate more dynamic aspects by involving time and extend our framework to fit with online incremental ASP (Gebser et al. 2011).

References

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge university press.

Baral, C. 2008. Using answer set programming for knowledge representation and reasoning: Future directions. In *ICLP*. 69–70.

Brewka, G., Eiter, T., and Fink, M. 2011. Nonmonotonic multi-context systems: A flexible approach for integrating heterogeneous knowledge sources. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*. 233–258.

Brogi, A., Contiero, S., and Turini, F. 1999. Programming by combining general logic programs. *J. Log. Comput. 9,* 1, 7–24.

Brogi, A., Mancarella, P., Pedreschi, D., and Turini, F. 1994. Modular logic programming. *ACM Trans. Program. Lang. Syst. 16,* 4, 1361–1398.

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

De Vos, M., Kisa, D. G., Oetsch, J., Pührer, J., and Tompits, H. 2012. Annotating answer-set programs in lana? *CoRR abs/1210.2195*.

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, M., Grote, T., Kaminski, R., and Schaub, T. 2011. Reactive answer set programming. In *LPNMR*. 54–66.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. 2007. *clasp* : A conflict-driven answer set solver. In *LPNMR*. 260–265.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *ICLP/SLP*. 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.

Loke, S. W. 2004. Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. *The Knowledge Engineering Review 19,* 03, 213–233.

Minsky, M. 1991. Society of mind: A response to four reviews. *Artif. Intell. 48,* 3, 371–396.

Monteiro, L. and Porto, A. 1989. Contextual logic programming. In *ICLP*. 284–299.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell. 25,* 3-4, 241–273.

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*. 20–40.

Oikarinen, E. and Janhunen, T. 2008. Achieving compositionality of the stable model semantics for smodels programs. *TPLP 8,* 5-6, 717–761.

Sadri, F. and Toni, F. 2006. Interleaving belief updating and reasoning in abductive logic programming. In *ECAI*. 442–446.