Answer Set Modules for Logical Agents

Stefania Costantini

Università degli Studi di L'Aquila
Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
Stefania.Costantini@univaq.it

Abstract. Various approaches exist to the application of Answer Set Programming (ASP) in the agent realm. Nonetheless, a controversial point is how to combine answer set modules with the other modules an agent is composed of, considering that an agent can be seen as a set of "capabilities" that in suitable combination produce the overall agent behavior as an emergent behavior. In this paper, we outline a possible fruitful integration of ASP into many agent architectures, by introducing two kinds of modules: one that allows for complex reaction, the other one that allows for reasoning about necessity and possibility.

1 Introduction

Logic programming under the answer set semantics (Answer Set Programming, for short ASP) is nowadays a well-established programming paradigm, with applications in many areas, including problem solving, configuration, information integration, security analysis, agent systems, semantic web, and planning (see among many [1,2,3,4,5] and the references therein).

The application of ASP in agents has been advocated since long, with ASP mainly taking the form of Action Description Languages. These kind of ASP-based languages were first introduced in [6] and [7] and have been since then extended and refined in many subsequent papers by several authors. Action Description Languages are formal models used to describe dynamic domains, by focusing on the representation of effects of actions. In particular, an action specification represents the direct effects of each action on the state of the world, while the semantics of the language takes care of all the other aspects concerning the evolution of the world (e.g., the ramification problem).

The first approaches have been extended in many ways, recently also in order to cope with, interpret, and recover from, exogenous events and unexpected observations, on the line of [8]. In this direction we mention [7], [9], and the recent work presented in [10]. In this work, an architecture (called AAA) is described where both the description of the domain's behavior and the reasoning components are written in Answer Set Programming, selected because of its ability to represent various forms of knowledge including defaults, causal relations, statements referring to incompleteness of knowledge, etc. An AAA agent executes a main cycle according to the *Observe-Think-Act* model proposed in the seminal paper [11]. Unexpected observations are coped with by hypothesizing the undetected occurrence of exogenous actions. In [12], this notion of

an agent is extended to enable communication between agents through the introduction of special named sets of fluents known as "requests".

In other directions, we mention a different line of work, focusing upon modeling agent decisions in an extended ASP by means of game theory [13]. In [14,15] and other papers by the same group, ASP is exploited to model dynamic updates of an agent's knowledge base. We are also aware of ongoing work about modeling properties of multi-agent systems in ASP, e.g., [16].

Despite this corpus of work is technically and conceptually very well-developed, the view of an agent based upon having an ASP program as its "core" does not appear to be fully convincing. One reason is that the basic feature of ASP, which is that a program may have several answer sets that correspond to alternative coherent views of the world, is in our opinion not fully suitable for the agent main cycle, while it can be very useful for many of the agent reasoning activities. Another reason is that the architecture outlined above appears to be too rigid with respect to the other approaches to defining agent architectures in computational logic, among which one has to mention at least MetateM, 3APL, AgentSpeak, Impact, KGP and DALI [17,18,19,20,21,22,23,24,25] (for a recent survey the reader may refer to [26]). All these architectures, and their operational models, are in practice or at least in principle more dynamic and flexible. If we consider for instance the KGP [24,25] architecture, we find many modules ("capabilities") and many knowledge bases, integrated by *control theories* that can be interchanged according to the agent's present context and tasks. In KGP, capabilities are supposed to be based upon abductive logic programming [27] but the architecture might in principle accommodate modules defined in different ways.

We believe that an "ideal" agent architecture should exploit the potential of integrating several modules/components representing different behaviors/forms of reasoning, with these modules possibly based upon different formalisms. The "overall agent" should emerge from dynamic, non-deterministic combination of these behaviors that should occur also in consequence of the evolution of the agent's environment. Therefore, in our view an important present and future direction of ASP is that of being able to encapsulate ASP programs into modules suitable to be integrated into an overall agent program, the latter expressed in whatever languages/formalisms. There is a growing corpus of literature about modules in ASP (see Section 3). However, the existing approaches mainly refer to traditional programming techniques and to software engineering methodologies. To the best of our knowledge, except for the approach of [28] in the context of action theories, there is no existing approach to modules which is tailored for the agent realm.

Building upon our long-termed experience in logical agents, involving the definition and implementation of the DALI agent-oriented logic language [22,23,29,30], in this paper we propose two kinds of ASP modules to be possibly integrated into a variety of agent architectures. A first perspective is that of "Reactive ASP modules", aimed at defining complex reaction strategies to cope with external events and establish what could be done. An ASP module will determine the different possibilities, among which the agent will choose according either to preferences or to an overall planning strategy. A second particularly relevant perspective is that of "Modal ASP modules", that exploit the multi-model nature of answer set semantics to allow for reasoning about

possibility and necessity in agents, at a comparatively low complexity. The proposed approach allows for interesting forms of reasoning suitable for real applications. From the implementation point of view, we implemented and we have been experimenting ASP modules within the DALI multi-agent system [31].

The paper is structured as follows. In Section 2 we briefly introduce answer set programming to the non-expert reader. In Sections 3 and 4 we review the existing research about modules in ASP and we quickly discuss logical agents. In Sections 5 and 6 we introduce Reactive and Modal ASP modules respectively, of which we propose a possible operative usage and some examples of application. Finally, we conclude in Section 7.

2 Answer Set Programming in a Nutshell, and some Terminology

"Answer set programming" (ASP) is the well-established logic programming paradigm adopting logic programs with default negation under the *answer set semantics*, shortly summarized below. For the applications of ASP, the reader can refer for instance to [1,2,3,4,5]. Several well-developed answer set solvers [32] that compute the answer sets of a given program can be freely downloaded by potential users [32].

In the rest of the paper, whenever it is clear from the context, by "a (logic) program Π" we mean a datalog program Π (for datalog the reader may refer for instance to [33]), and we will implicitly refer to the "ground" version of Π. The ground version of Π is obtained by replacing in all possible ways the variables occurring in Π with the constants occurring in Π itself, and is thus composed of *ground atoms*, i.e., atoms which contain no variables. The Herbrand base $\mathbf{B}_\Pi$ of a ground ASP program Π is composed of all ground atoms that can be constructed out of the set of predicate symbols and the set of constant symbols occurring in Π. We indicate with $\mathcal{B}_\Pi$ the restriction of $\mathbf{B}_\Pi$ to the atoms actually occurring in the ground version of Π. This assumption is due to the fact that ASP solvers produce the grounding of the given program as a first step. In fact, they are presently able to find the answer sets of ground programs only (though work is under way to overcome at least partially this limitation, cf.,e.g., [34,35]).

Let $\mathcal{V}$ be a set of variables. To the purposes of this paper, we will call *abstract atom* (referring to program Π) any non-ground atom built out of a ground atom $A \in \mathcal{B}_\Pi$ by substituting some of the constants occurring in it by variables in $\mathcal{V}$. We will call $\mathcal{B}^a_\Pi$ the set of all the abstract atoms obtained from $\mathcal{B}_\Pi$. Vice versa, a *proper* instantiation (w.r.t. program Π) of an abstract atom $B \in \mathcal{B}^a_\Pi$ is an instantiation (ground instance) A of B such that $A \in \mathcal{B}_\Pi$. If S is a set of abstract atoms, we let $Ground(S)$ be the set of its proper instantiations. Instantiations and proper instantiations of a conjunction of abstract atoms have the obvious definition.

A normal program (or, for short, just "program") Π is a collection of *rules* of the form

$$H \leftarrow \ L_1, \ldots, L_m, \ not \ L_{m+1}, \ldots, not \ L_{m+n}.$$

where H is an atom, $m \geqslant 0$ and $n \geqslant 0$, and each L_i is an atom. An atom L_i and its negative counterpart $not\, L_i$ are called *literals*. In the examples, $\leftarrow$ will often be indicated with $:-$, which is the symbol adopted in practical programming systems. In

the version of Π defined by a programmer, all atoms will be in general abstract atoms. In the ground version of Π they become ground atoms, as each original rule is substituted by all its ground instantiations. Various extensions to the basic paradigm exist, that we do not consider here as they are not essential in the present context. The left-hand side and the right-hand side of the clause are called *head* and *body*, respectively. A rule with empty body is called a *fact*. A rule with empty head is a *constraint*, where a constraint of the form

$$\leftarrow L_1, ..., L_n.$$

states that literals $L_1, \ldots, L_n$ cannot be simultaneously true in any answer set.

The answer sets semantics [36,37] is a view of logic programs as sets of inference rules (more precisely, default inference rules). Alternatively, one can see a program as a set of constraints on the solution of a problem, where each answer set represents a solution compatible with the constraints expressed by the program. Consider the simple program $\{q \leftarrow not\, p.\ \ p \leftarrow not\, q.\}$. For instance, the first rule is read as "assuming that p is false, we can *conclude* that q is true." This program has two answer sets. In the first one, q is true while p is false; in the second one, p is true while q is false. The programming paradigm based upon logic programs under the answer set semantics is called "Answer Set Programming" (ASP), and programs are called "answer set programs" (ASP programs).

A subset M of $\mathcal{B}_\Pi$ is an answer set of Π if M coincides with the least model of the reduct P^M of P with respect to M. This reduct is obtained by deleting from Π all rules containing a condition $not\, a$, for some a in M, and by deleting all negative conditions from the other rules. Answer sets are minimal supported models, and form an anti-chain. Referring to the original terminology of [36], answer sets are sometimes called *stable models*. Unlike other semantics, a program may have several answer sets, or may have no answer set, because conclusions are included in an answer set only if they can be justified. The following program has no answer set:
$\{a \leftarrow not\, b.\ b \leftarrow not\, c.\ c \leftarrow not\, a.\}$
The reason is that in every minimal model of this program there is a true atom that depends (in the program) on the negation of another true atom, which is strictly forbidden in this semantics, where every answer set can be considered as a self-consistent and self-supporting set of consequences of a given program. Whenever a program has no answer sets, we will say that the program is *inconsistent*. Correspondingly, checking for consistency (or stability) means checking for the existence of answer sets.

By some abuse of notation, given program Π and a set of facts and rules I, by $\Pi \cup I$ we indicate the new program obtained by adding the atoms and rules occurring in I to Π . Also, if a consistent program Π has a number k of answer sets, we will assume an arbitrary enumeration $M_1, \ldots, M_k$ of these answer sets, and we will refer to M_h ($h \leq k$) as the $h - th$ answer set. Given answer set program Π which is inconsistent, we call a set of atoms $R \subseteq \mathcal{B}_\Pi$ a *trigger* for Π whenever $\Pi \cup R$ is consistent.

As it is well known (cf., e.g., [38]), an ASP program is inconsistent whenever there is some *odd cycle*, like for instance the above one
$\{a \leftarrow not\, b.\ b \leftarrow not\, c.\ c \leftarrow not\, a.\}$

For obtaining a potentially consistent program from one including such a cycle, the cycle should *constrained* by adding, in the terminology of [38], some *handle* for the cycle. A handle for the above cycle can consist of, e.g., rule $\{a \leftarrow d.\}$. Or, it can consist of an additional literal, e.g., $not\, r$, added to any of the rules of the cycle, say for instance the second one. If at least one handle of an odd cycle is *active*, then the program fragment including the odd cycle and the handles is consistent. The former handle is active if d occurs somewhere in the overall program. Thus, the head a of the rule becomes true. The latter handle is active if r occurs somewhere in the overall program. Thus, $not\, r$ is false and then the head b of the rule where this literal occurs becomes true. In both cases, the circularity is broken, i.e., if there are active handles the cycle becomes (again in the terminology of [38]) *actually constrained*. For the overall program to be consistent, every odd cycle must be actually constrained. This requires that, if there are several odd cycles, they admit handles which are *compatible*, i.e., that do not expect opposite truth values for the same atom.

An inconsistent program Π necessarily involves some "problematic" odd cycle which is not actually constrained. Therefore, a trigger R for Π includes a set of atoms that make all the odd cycles in Π actually constrained. I.e., a trigger includes atoms that make at least one handle for each problematic odd cycle active, where these handles are compatible among themselves and with those already present in Π.

In the following sections, triggers will be exploited as a "control" device to manage modules consisting of an ASP program. Such a module will be supposed to provide some kind of answer to an agent which "invokes" it by providing suitable input. We will assume the ASP program defining a module to be inconsistent on purpose, and to be designed so that a trigger must include the significant input the module needs in order to provide meaningful answers. Then, providing a trigger will be the way for an agent to invoke a module and get answers.

As mentioned before, ASP has the peculiarity that an ASP program may have none, one or several answer sets. These answer sets can be interpreted in various possible ways. If the program formalizes a search problem, e.g., a colorability problem or a path finding problem for graphs, then the answer sets represents the possible solutions to the problem, namely, in the examples, the possible colorings or the existing paths for given graph. In knowledge representation, an ASP program may represent a formal definition of the known features of a situation/world of interest. In this case, the answer sets represent the possible consistent states of this world, that can be several whenever the formalization involves some kind of uncertainty. Also, and ASP program can be seen as the formalization of the knowledge and beliefs of a rational agent about a situation/world, and the answer sets represent the possible belief states of such an agent, that can be several if either uncertainty or alternative possible choices are involved in the description. Such an agent can exploit an ASP module for several purposes, such as answering questions, building plans, explaining observations, making choices, etc. Some potential uses of ASP modules in agents will be proposed and discussed in the rest of the paper.

3 Related Work on ASP Modules

There are several approaches to modularization of ASP programs with software engineering purposes, i.e., to govern the complexity of programs and their development process. For a review of the state of the art in this field the reader may refer for instance to [39] and to the references therein.

In the approach of [40,39], in conformance with programming-in-the-large principles, a suitable input-output interface for ASP modules is defined, in order to compute the combination of compatible answer sets of joinable modules. By providing a notion of equivalence for modules, the approach tackles the issue of the replacement of a module with another one without altering the semantics of the program when seen as an overall entity.

This proposal is related to that of [41], then evolved into [42], as each one can be rephrased in terms of the other. However, in the latter proposal the point of view is different, as modules are seen as "procedures" that can invoke each other, even recursively, by providing input parameters. An overall program is composed of several modules where a "main" module without input can be identified. Providing the semantics of a program requires to identify, via a *call graph*, the relevant modules, i.e., those that are actually invoked. Complexity ranges from exponential to double exponential, due to the complex module interaction that the approach admits.

In [43], modules import answer sets from other modules in order to compute the overall solution, where no cycles are admitted among modules. [44] provides modules specification with information hiding, where modules exchange information with a global state.

Some approaches exist [45,46] that, in order to encourage code reusability, define modules in terms of macros or "templates" that factorize predefined definitions, again with no cycle allowed among these entities.

In [47], a technique is proposed that allows an answer set program to access the brave or cautious consequences of another answer set program. The technique is based upon joining the two programs into a single one and then performing a suitable rewriting with the addition of weak constraints.

In the following sections we will propose *Reactive ASP Modules*, where complex forms of reaction can be specified in an agent program, in contrast to the simple "condition-action rules" that are often adopted. Namely, an ASP module will describe how an agent might behave upon the occurrence of certain events, also depending upon particular circumstances and/or the agent's past experiences (e.g., when and why lend or not lend a certain resource upon request). When provided with information about the present context, the answer sets of such a module will encode the possible courses of action that the agent might undertake. We will also propose *Modal ASP Modules*, where an agent will be enabled to reason about possibility and necessity. I.e., such a module will describe what an agent knows or believes about some situation, and the agent will be enable to inspect its answer sets so as to "bring to consciousness" its own mental states and understand what is possible in that situation (because it occurs in some answer set) and/or what is mandatory (because it occurs in every answer set). In previous example, lending some resource to a certain requester might be possible given some

conditions, or even mandatory if for instance the agent has previously contracted an obligation. This will imply encoding in an ASP module a fragment of the domain of interest of the agent and examining the answer sets of the module. In the present proposal, ASP modules do not interact with each other. For future extensions in the direction of interacting modules, the techniques presented in [47] might be of use for a principled implementation.

The first idea of exploiting possibility and necessity in ASP is due to Michael Gelfond and presented in [48]. In this proposal, possibility and necessity operators can occur in ASP programs, thus called "epistemic logic programs", in the body of rules. Therefore, concluding or not the head of these rules will depend upon the contents of a program's own answer sets. A suitable extension of the answer set semantics is introduced to cope with the enhanced expressivity. The work presented in [49,50] investigates computational complexity of this approach by redefining its semantics as *world view semantics*. On the one hand it is concluded that the consistency check problem under this semantics is PSPACE-complete. On the other hand however, non-trivial classes of programs where the complexity is Σ_2^P-complete or even NP-complete are identified. In [16], the authors adopt a different perspective and employ meta-programming techniques to model in an ASP program multi-agent systems involving agents with knowledge about other agent's knowledge.

Related to the present work is ASP-PROLOG [51], that proposes an integration between prolog and ASP where prolog programs are enabled to invoke ASP modules and examine the answers sets. These modules can be customized by adding and removing rules prior to invocation. The similarity with the approach presented in this paper lays in the fact the the prolog program invoking ASP modules can be seen as analogous to a logical agent program exploiting ASP modules, though the kind of application and the envisaged use of modules is different. ASP-PROLOG is procedural in nature and extends the standard prolog notation. It might be a good implementation tool for many kinds of ASP modules, included those presented here.

In Section 6.1 we will show how to exploit possibility and necessity to perform interesting forms of meta-reasoning. An approach to meta-reasoning within ASP programs is that of [52], which proposes "template" rules with variables in place of predicates (to be suitable instantiated to actual predicate symbols occurring in the program), in the style of Reflective Prolog [53,54]. The work presented in [55] interprets ASP programs as agents and allows for various forms of reasoning by introducing deontic operators (such as for instance *Obligation*) in such programs.

4 Logical Agents in short

Recently, the computing landscape has changed from a focus on standalone computer systems to a situation characterized by distributed, open and dynamic heterogeneous systems that must interact, and must operate effectively within rapidly changing circumstances and with increasing quantities of available information. In this context, agents constitute a suitable design metaphor, that provides designers and developers with a way of structuring an application around autonomous, communicative and flexible elements [56].

Agents should be *intelligent* so as to face changing situations by modifying their behavior, or their goals, or the way to achieve their goals. This requires agents to be able to perform, interleave and combine various forms of commonsense reasoning, possibly based upon different kinds of representation. Several agent-oriented languages and architecture exist and in particular several computational logic-based agent architectures and models. A common feature is the aim at building agents that are able to adapt or change their behavior when they encounter a new or different situation.

A logical agent is based upon an "agent program" which consists of a knowledge base and of a set of rules aimed at providing the entity with the needed capabilities. Rules may include object-level rules and meta-(meta-. . .)rules that determine the agent behavior. The knowledge base may itself include rules, which either define knowledge (and meta-knowledge) in an abstract way or constitute part of the agent knowledge. The knowledge base constitutes in fact the agent "memory" while rules define the agent behavior. An underlying inference engine, or more generally a control mechanism, puts an agent at work. Agents in general evolve in time as a result of both their interaction with the environment and their own self-modifications. Despite the differences, all logical agent-oriented architectures and languages, or "agent models", exhibit at least the following basic features (for a general discussion about logical agent models the reader may see, e.g., [57] and [58], and for a general logical semantics for evolving agents [29]):

- A logical "core", that for instance in both KGP and DALI is a resolution-based logic program (prolog-like for DALI and abductive for KGP).
- Reactivity, i.e., the capability of managing external stimuli.
- Proactivity, i.e., the capability of managing internal "initiatives".
- The capability of performing actions.
- The capability of recording what has happened and has been done in the past.
- The capability of managing communication with other agents.
- A basic cycle that interleaves the application of formerly specified capabilities. E.g., in DALI the basic cycle is integrated within the logical core into an extended resolution, while in KGP the basic cycle has a meta-level definition and thus can be varied.

Taking for instance KGP and DALI, which are two well-known and fully implemented agent models based upon logic programming, we can identify the following more specific features.

KGP agents are equipped with the following components.

(1) A set of beliefs, equipped with a set of *reasoning capabilities*, for reasoning with the information available in the agent state. These capabilities include Planning, Temporal Reasoning, Reactivity, Goal Decision, and Temporal Constraint Satisfiability. Beliefs include a records of the information sensed from the environment, as well as a history of executed actions.

(2) A set of *goals* and *plans* to which the agent is committed.

(3) A sensing capability, allowing agents to observe their environment and actions (including utterances) by other agents.

(3) An actuating capability, allowing agents to affect their environment (including by performing utterances).

(4) Control information, including a set of *transition rules*, changing the agent's state and a set of *selection functions* to select inputs to transitions.

(5) A control component, for deciding which enabled transition should be next [59].

The DALI agent model includes:

(i) A set of beliefs, including reactive rules, support for proactivity and reasoning, planning, constraint satisfiability. Beliefs also include *past events* that record what has happened in the past: events perceived and reacted to, proactive initiatives, goals reached, etc. Past events can be organized into histories on which properties can be verified by means of constraints.

(ii) A sensing capability, allowing agents to observe their environment and actions by other agents.

(iii) A set of constraints for verifying that the agent's course of actions respects some properties and does not present anomalies.

(iv) A learning component for recording past events and building histories; a belief revision component for removing old information based on conditions and for either incorporating or dropping knowledge acquired from other agents.

(v) Control information that may influence proactive behavior and the recording of past events.

Both KGP and DALI are by their very natural modular architectures, as agents are composed of various modules. ASP modules may be exploited in these architectures to implement various capabilities, for instance planning. In subsequent sections, we will propose however some kinds of ASP modules that may actually under some respects empower these agent models.

5 Reactive ASP Modules

Since [60], it is universally recognized that *reactivity* is an essential feature in logical agents, in the sense of an agent being able to respond in a timely and appropriate way to the reception of stimuli coming from an external environment which is in general subject to change and that can generate events in an unforeseeable sequence. Reactions are often expressed in condition-action rules, say e.g. of the form

$IF\ \langle Conditions \rangle\ DO\ \langle Actions \rangle$

which are also present in ASP-based action languages, where however they are not meant to be triggered by the conditions, but are rather processed contextually to the rest of the program. The problem was tackled in [28] where *reactive control modules* composed of condition-action rules were introduced and the problem of their correctness w.r.t. the overall program (action theory) was discussed.

Here, we intend to introduce modules that allow for "complex" reactivity, where some kind of reasoning has to be performed in order to devise suitable reactions. These modules are intended to "sleep" in the background and enter into play when activated by the occurrence of external events. In order to choose among the different actions

that is possible to perform, corresponding to different answer sets of a reactive module, we build upon previous work [61], where we introduced priorities among (conditional) actions in logic agent-oriented languages.

In the rest of this section, we propose a formulation, a possible operational behavior and some examples of use of reactive ASP modules. Technically, we will specify reactive ASP modules by exploiting a distinguished, ASP feature, i.e., the constraints. We also make the reactive behavior parametric w.r.t. context conditions that may be different in different module invocations.

The basic idea is that of constructing a reactive ASP module around an inconsistent ASP program, where inconsistency is due to one or more constraints of the form

$:\!-not\ A_1, \ldots, not\ A_n$

where the A_i's are atoms, which represent the events that must happen in order to *activate* the module. In fact, if no event has happened, all the A_i's are false which implies that the constraint is violated (as all the $not\ A_i$'s are true) and therefore the module is inconsistent. A module will stay "asleep" until one or more events happen: events which have occurred will be asserted as facts, thus acting as triggers that make the module consistent. The module will now have answer sets which encompass the possible reactions to these events. The proposed formulation of reactive ASP modules is aimed at their introduction in the basic cycle of the agent architecture at hand. In this basic cycle, there will be at some stage a check of reactive modules that, whenever active, will generate possible reactions one of which will be chosen and put into play either nondeterministically or based on preferences.

A reactive ASP module will have an input/output interface. The input interface specifies the events that may trigger the module. The output interface specifies the actions that the module answer sets (if any) can possibly encompass. However, at the *invocation* the module will return not only the actions, but also the conditions (if any) for their being actually performed. For instance, if the module performs some form of default reasoning [62], the output may include the normality/abnormality assumptions.

We introduce below the definition that specifies an ASP reactive module after giving some guidelines about the logic program which constitutes its "core".

Definition 1. *A* completed logic program Π *is obtained from an inconsistent logic program* Π_{given} *containing at least one constraint of the form* $:\!-not\ A_1, \ldots, not\ A_n$ *by adding, for each atom* A *that in* Π_{given} *does not occur in a constraint and does not occur as the head of a rule, the even cycle (composed of two rules):* $A :\!-not\ noA$, $noA :\!-not\ A$ *where* noA *is a fresh atom.*

A will be called an *assumption* (w.r.t. program Π) and the set of all the assumptions will be called $\mathcal{A}_\Pi$. The purpose of assumptions will be illustrated below in relation to an example.

Definition 2. *A reactive ASP module* $\mathcal{M}$ *is a triple* $\langle In, \Pi, Out\rangle$ *where* Π *is a completed logic program and* $In, Out \subseteq \mathcal{B}^a_\Pi$ *are sets of abstract atoms, called the* abstract inputs *and* abstract outputs *respectively, where* $\mathcal{A}_\Pi \subseteq Ground(Out)$.

A reactive ASP module can be invoked by providing an input including the proper instantiations of (some of) the atoms in In (i.e., it is not mandatory to provide *all* the specified inputs). Symmetrically, it may be the case that only part of the outputs is returned. However, the input may also include facts and rules that represent additional contextual knowledge useful for the evaluation of the reaction. These facts and rules are here required to be ground.

Given *actual input* I, an invocation implies to determine the answer sets of $\Pi \cup I$ and to extract proper instantiations for the outputs. If $\Pi \cup I$ is consistent, there may be different results corresponding to the different answer sets. Otherwise, no result will be returned. We may notice that the assumptions which belong to each answer set are returned in the output by definition, unless they have been provided as input. In fact, no input atom is returned as output.

Definition 3. *An* invocation result *of a reactive ASP module* $\mathcal{M} = \langle In, \Pi, Out \rangle$ *is a triple* $\langle I, \Pi, O \rangle$*, where:* I*, called the* actual input*, is a set of ground facts and rules*[1]*, including proper instantiations of (some of the) atoms in* In*;* $O \subseteq \mathcal{B}_\Pi$*, called the* actual output*, includes proper instantiations of (some of the) atoms in* Out*, where either* $\Pi \cup I$ *is inconsistent and* $O = \emptyset$ *or* $O \subseteq (M \setminus I)$ *where* M *is an answer set of* $\Pi \cup I$ *and* O *is composed of all the proper instantiations of atoms in* Out *which occur in* M*, except those given in the input.*

It is easy to see that, given input I, there are as many invocation results as the answer sets of $\Pi \cup I$ (among which the actual course of action must be somehow selected by the agent), and that $O \neq \emptyset$ only if I includes a trigger R for Π.

Operationally, invocation of ASP modules can explicitly occur in an agent program, where the precise way to invoke a module will depend upon the agent language at hand. In DALI for instance, the simple reactive rules of the language can be used to directly resort to a reactive module whenever the relevant events occur together (where DALI provides a way of specifying what does it mean to happen together for a given set of events, e.g., in the same day, same second, etc.). Other methods for invocation are also possible: e.g., the inputs related to an invocation can be written on a blackboard which is examined from time to time by an underlying control component which performs the invocation, and puts the results on the blackboard.

The ASP modules so defined are suitable for specifying the reaction to external stimuli, where, in an invocation, the inputs include the external stimuli and the outputs include a set of actions to be executed in response to the stimuli according to the assumptions. In our view in fact, reactive ASP modules should be used to describe knowledge and beliefs concerning how an agent would cope with some events in a given situation. The answer sets of a reactive module are meant to represent the possible courses of action that the agent might undertake whenever these events actually occur, given the present context. They will in general contain plans that the agent might execute to cope with the events together with the assumptions these plans are based

[1] Facts and rules composed of ground atoms. Notice, here and in what follows, that they are not required to be composed only of atoms in $\mathbf{B}_\Pi$, i.e., fresh predicate and constant symbols are allowed to occur.

upon. In simple cases, like in the examples below, a plan may plainly consist of few actions whose order does not matter. The module "core" is a completed program so that whatever is not known and is not provided as input can possibly be assumed. An agent invokes an ASP module by providing an actual input including a trigger for the module and all the relevant information which is available. Among the resulting answer sets, the agent will have to choose according to some criteria and put the selected course of action into operation.

Below we propose an example of an ASP module. For the sake of clarity, here and in the rest of the paper we adopt the DALI syntax, and thus we assume to indicate predicates denoting actions with suffix 'A' and those denoting external stimuli with suffix 'E'. The external stimulus to be coped with is $bell\_ringsE$. Program Π is the following. It is a completed program, where $good\_weather$ is an assumption, i.e., if not provided as input, the agent may assume that the weather is good or not. The agent will open the door if the bell rings whenever the situation does not look dangerous (i.e., we are not at night with strangers around). It opens the window whenever the weather good, or, precisely, whenever either it is known to be good (because this information has been received in input) or it has been assumed to be good. Notice that the former action is generated by means of a very simple form of *proactivity*, i.e., on the agent's own initiative. In fact, it is not a reaction to an external event and it is not necessarily a consequence of what is known (the weather being good can arbitrarily be assumed if not known). Proactivity is commonly assumed to be a main feature of agents.

```
:- not bell_ringsE.
openA(door) :- bell_ringsE.
:- openA(door), at_night, strangers_around.
openA(window):- good_weather.
good_weather :- not nogood_weather.
nogood_weather :- not good_weather.
```

The trigger that makes Π consistent is the external event $bell\_ringsE$ and the resulting program $\Pi \cup bell\_ringsE$ has a number of answer sets which depends upon whether both at_night and $strangers\_around$ are given, and whether $good\_weather$ is either given or assumed. If we do not either have as input or assume $good\_weather$ we can possibly conclude $openA(door)$ but not $openA(window)$, which has $good\_weather$ as a condition. If we have both at_night and $strangers\_around$, we cannot conclude $openA(door)$ but, if we assume $good\_weather$, then we can possibly conclude $openA(window)$.

A reactive ASP module associated to Π can be for instance:

$\langle \{bell\_ringsE, at\_night, strangers\_around\}, \Pi, \{openA(X)\} \rangle$

Among the possible invocation results, each one corresponding to an answer set of $\Pi \cup I$, we have the following :

$\langle \{bell\_ringsE\}, \Pi, \{openA(door)\} \rangle$

$\langle \{bell\_ringsE, good\_weather\}, \Pi, \{openA(door), openA(window)\} \rangle$

$\langle \{bell\_ringsE\}, \Pi, \{good\_weather, openA(door), openA(window)\} \rangle$

$\langle\{bell\_ringsE, at\_night, strangers\_around\}, \Pi, \emptyset\rangle$

$\langle\{bell\_ringsE, at\_night\}, \Pi, \{good\_weather, openA(window)\}\rangle$

As another example, program Π below states that the agent may or may not lend money to somebody, however: (s)he never lends money to unreliable persons; (s)he normally lends money to friends, unless this friend is an unreliable person. Notice that lending/not lending money is chosen arbitrarily, unless conditions occur (stated in the constraints) to force an agent to make a certain choice. Going back to the definition of completed program (Definition 1), it may be noticed that every item of information occurring in the program can be assumed if not provided in input, except the conditions occurring in the constraints. For instance, in the module below the agent will force itself to lend the money if the request comes from a friend, but the requester being a friend must be explicitly specified in input (of course, if at the invocation the agent believes that this is the case).

```
:- not requestE.
lend_moneyA :- not no_land, requestE.
no_land :- not lend_moneyA.
:- lend_moneyA, unreliable_person.
:- not lend_moneyA, requestE, friend.
```

A reactive ASP module associated to Π can be for instance:

$\langle\{requestE, friend, unreliable\_person\}, \Pi, \{lend\_moneyA\}\rangle$

We may have for instance the following invocation results, where notice that, if the requester is stated to be unreliable, the output is empty (thus no action is prescribed) as the module is inconsistent.

$\langle\{requestE, friend\}, \Pi, \{lend\_moneyA\}\rangle$

$\langle\{requestE, friend, unreliable\_person\}, \Pi, \emptyset\rangle$

$\langle\{requestE\}, \Pi, \emptyset\}\rangle$ and $\langle\{requestE\}, \Pi, \{lend\_moneyA\}\rangle$

With input $requestE$, one of the two possible outcomes must be chosen as the actual course of action.

6 Reasoning on Possibility and Necessity: Modal ASP Modules

In this section, we propose another kind of ASP module, defined so as to allow forms of reasoning to be expressed on possibility and necessity analogous to those of modal logic. As it is well-known, in classical modal logic (see [63]) a proposition is said to be possible if and only if it is not necessarily false (regardless of whether it is actually true or actually false), and to be *necessary* if and only if it is not possibly false. The meaning of these terms refers to the existence of multiple "possible worlds": something "necessary" is true in all possible worlds, something "possible" is true in at least one possible world. These "possible world semantics" are formalized with Kripke semantics. Either the notion of possibility or that of necessity may be taken to be basic, where the other one is defined in terms of it. Possibility and necessity are related to *credulous*

and *skeptical* (or *brave* and *cautious*) reasoning in non-monotonic reasoning, where in the credulous (brave) approach a proposition is believed if it is possible, while in the skeptical (cautious) approach it is believed only if it is necessary.

In our setting, the "possible worlds" that we consider refer to an ASP program Π and are its answer sets. Therefore, given $A \in \mathcal{B}_\Pi$, we will say that A is possible if it belongs to some answer set, and that A is necessary if it belongs to the intersection of all the answer sets.

A comment is in order about why we do not choose to refer to the well-founded model (wfm) [64]. In fact, as it is well-known every answer set M of a given program Π is a superset of the wfm of the program. This means that, given $WFM = \langle T; F \rangle$ where atoms in T are considered to be true, atoms in F are considered to be false and all the other atoms are considered to be undefined (i.e., the WFM is a three-valued semantics) we have $T \subseteq M$. However, T is in general smaller than the intersection of all the answer sets, as it includes only the consequences derivable from the acyclic part of the program. Therefore, T does not include consequences deriving from assumptions, even when these assumptions lead to the same conclusion in all the answer sets[2].

6.1 Definition, Use and Applications of Modal ASP Modules

We introduce below an operator of possibility, that we indicate with P (instead of the traditional $\diamond$, or M), and an operator of necessity, that we indicate with N (instead of the classical $\Box$, or L). We change the terminology as we re-define the operators w.r.t. the answer sets of a program considered as a theory. In this specific setting, properties of the operators can be proved rather than defined axiomatically. These operators define *Modal ASP Expressions* that can be either *possibility expressions* or *necessity expressions*.

Definition 4. *Given answer set program Π with answer sets enumerated as $M_1, \ldots, M_k$, and an atom A, the* possibility *expression $P(w_i, A)$ is deemed to hold (w.r.t. Π) whenever $A \in M_{w_i}$, $w_i \in \{1, \ldots, k\}$. The possibility operator $P(A)$ is deemed to hold whenever $\exists M \in \{M_1, \ldots, M_k\}$ such that $A \in M$.*

Definition 5. *Given answer set program Π with answer sets $M_1, \ldots, M_k$, and an atom A, the* necessity *expression $N(A)$ is deemed to hold (w.r.t. Π) whenever $A \in (M_1 \cap \ldots \cap M_k)$.*

We are now able to define the negation of possibility and necessity operators.

Definition 6. *Given answer set program Π with answer sets enumerated as $M_1, \ldots, M_k$, and an atom A: the* possibility *expression $\neg P(w_i, A)$ is deemed to hold (w.r.t. Π) whenever $A \notin M_{w_i}$, $w_i \in \{1, \ldots, k\}$; the expression $\neg P(A)$ is deemed to hold whenever $\neg\exists M \in \{M_1, \ldots, M_k\}$ such that $A \in M$; the* necessity *expression $\neg N(A)$ is deemed to hold (w.r.t. Π) whenever $A \notin (M_1 \cap \ldots M_k)$.*

[2] For the interested reader, in previous work [38] we have discussed the role of cycles and of connections between cycles for the consistency of the program.

It is easy to see that, given answer set program Π:

Proposition 1. *$N(A)$ implies $P(A)$ and implies that $\exists w_i$ such that $P(w_i, A)$.*

Proposition 2. *$\neg P(A)$ implies $\neg N(A)$.*

The extension of the above operators to conjunctions is straightforward, where a conjunction is deemed to be possible in a certain answer set w_i (resp. possible in general) whenever all conjuncts belong to w_i (resp. to the same answer set) and a conjunction is deemed to be necessary whenever all conjuncts belong to the intersection of the answer sets.

We now extend the definition of modal ASP expressions to include a context for their evaluation

Definition 7. *Let $E(Args)$ be either a possibility or a necessity expression. The corresponding* contextual *expression has the form $E(Args) : Context$ where $Context$ is a set of ground facts and rules. $E(Args) : Context$ is deemed to hold whenever $E(Args)$ holds w.r.t. $\Pi \cup Context$.*

The abstract counterparts of modal ASP expressions are expressions of the form $P(I, X)$, $P(X)$ and $N(X)$ (resp. $P(I, X) : C$, $P(X) : C$ and $N(X) : C$ for their contextual version) where: I is a variable ranging over natural numbers; X can be either an abstract atom or a conjunction of abstract atoms or also a metavariable intended to denote either an abstract atom or a conjunction of abstract atoms; C can be either a set of abstract atoms or a metavariable intended to denote a set of abstract atoms.

Possibility and necessity expressions are evaluated w.r.t. an underlying *modal ASP module* of the following form.

Definition 8. *A modal ASP module $\mathcal{M}$ is a tuple $\langle Module\_name, AbstrQuery, AbstrContext, \Pi, AbstrPos, AbstrNec\rangle$ where:*

- *$Module\_name$ is the name of the module;*
- *Π is a logic program;*
- *$AbstrQuery$ is either an abstract atom or a conjunction of abstract atoms (that can be intended as a set), i.e., $AbstrQuery \subseteq \mathcal{B}^a_\Pi$ and $AbstrQuery \neq \emptyset$;*
- *$AbstrContext$ is a metavariable denoting a set of ground facts and rules;*
- *$AbstrPos$ is a metavariable denoting a set of abstract possibility expressions of the form $P(I, AbstrQuery)$;*
- *Nec is a metavariable denoting either a necessity expressions of the form $N(AbstrQuery)$ or the empty set.*

A modal ASP module is invoked whenever a modal ASP expression has to be evaluated, by providing a proper instantiation of the abstract query and of the context by means of the arguments of the modal ASP expression at hand.

Definition 9. *An* invocation result *of a modal ASP module $\mathcal{M}$ is a tuple $\langle Module\_name, Query, Context, \Pi, Pos, Nec\rangle$ where:*

- *$Query \subseteq \mathcal{B}_{\Pi}$, $Query \neq \emptyset$, is composed of proper instantiations of (some of) the abstract atoms in $AbstrQuery$;*
- *$Context$ is a set of ground facts and rules;*
- *Pos is the set of the expressions $P(w_i, Query)$ that hold w.r.t. $\Pi \cup Context$, or the expression $\neg P(Query)$ if no possibility has been found to hold;*
- *Nec is either $N(Query)$ or $\neg N(Query)$ depending upon which of the two holds w.r.t. $\Pi \cup Context$.*

Notice that, from the practical point of view, once the module has been invoked on some input, its invocation result can be stored for subsequent use.

For the case where there are several modal ASP modules, the straightforward extension of the above-defined modal ASP expressions can be $E(T, Args)$ (resp. $E(T, Args) : Context$ for the contextual form) where the given expression is meant to be evaluated w.r.t. module (theory) T (precisely, w.r.t. program Π included in T).

The Kripke structure that we propose is simple, but yet it allows significant forms of reasoning to be performed. For instance, one is able to define meta-axioms, like, e.g., the following, which states that a proposition is plausible w.r.t. theory T if, say, it is possible in at least two different worlds:

$$plausible(T, Q) :- P(T, I, Q), P(T, J, Q), I \neq J.$$

We can also formulate the contextual counterpart of the above:

$$plausible(T, Q, C) :- P(T, I, Q) : C, P(T, J, Q) : C, I \neq J.$$

As we were mentioning before, to evaluate an instance of the meta-axioms above one has to invoke module T on query Q just once.

Among the relevant realms of possible application of modal ASP expressions are in our view normative reasoning and negotiation. Consider for instance the famous example proposed in the seminal work about meta-interpreters [65]:

$$guilty(X) :- demo(Facts, guilty(X))$$

meaning that one can be considered to be guilty only if (s)he is provably guilty within theory $Facts$ representing both the laws/regulations and the evidence. We can generalize this kind of reasoning by allowing $Facts$ to be an answer set program, i.e., by allowing non-monotonic reasoning and multiple possible solutions. In our setting, we might rephrase this example as follows:

$$guilty(X) :- N(Facts, guilty(X))$$

We might also allow evidence that one proposes to her/his excuse, e.g.,

$$innocent(X) :- \neg P(Facts, guilty(X)) : Evidence$$

Here, we have used a contextual expression where we say that one has to be considered innocent if it is impossible that (s)he is not, assuming to accept the $Evidence$ (s)he proposes as excuse.

6.2 Extension to Multi-Agent Setting

It can be interesting to extend our setting so as to allow an agent to reason not only about what is possible or necessary for herself/himself, but also about what is possible or necessary for other agents.

In this discussion, we assume that there are several agents, which are able to communicate with each other. We will however abstract from the details of the communication mechanism, assuming the existence of two primitives: $tell(Ag, Prop)$, to signify that the agent in which it occurs communicates proposition $Prop$ to agent Ag; $told(Ag, Prop)$, to signify that the agent in which it occurs receives proposition $Prop$ from agent Ag.

As a first simple example, let us assume for instance that agent Mary includes a modal ASP module where she decides whether to spend the evening going out (e.g, to cinema) or not. Let us also assume that there exists another agent, say John, who would like to invite Mary to cinema. In our approach, John can reason about Mary's possibilities, e.g., by means of a condition-action rule that might look like the following:

$$told(mary, P(go\_to\_cinema))\ OR$$
$$told(mary, P(go\_to\_cinema)\ :\ lend\_money)$$
$$DO\ tell(mary, lend\_money\_if\_needed, invite\_to\_cinema)$$

stating that if John is told by Mary that she would possibly go to cinema, either at her own expenses or upon the condition she can borrow some money, he offers to lend the money and invites her to go.

The next example refers to negotiation between agents. In the example, a benevolent agent accepts the justification of a partner agent for a contract violation if the partner is known to be reliable and offers a justification which is plausible w.r.t. a theory describing the negotiation domain, given a context (that presumably includes common knowledge about what has been going on). We refer to the above definition of $plausible$.

$$excused(Ag, Viol, Context\_facts) :-$$
$$N(Reputation\_theory, reliable(Ag)), told(Ag, Justification),$$
$$plausible(Domain\_theory, Justification, Context\_facts)$$

6.3 Complexity

As it is well-known, deciding the existence of an answer set has been proved NP-complete and the same for deciding whether an atom is a member of some answer

set, while the problem whether a given atom is in the intersection of all stable models is co-NP-complete (see [66] and [67])[3].

It is useful to remark that complexity of epistemic logic programs [48] recalled in Section 3 is not related to the complexity of the approach presented here. In fact, in epistemic logic programs necessity and possibility operators may occur *within* a theory, while here we reason about an inner theory which is a plain ASP program.

We state here the complexity of reasoning about possibility and necessity with the above-mentioned operators, that is in accordance with the above results. In fact, we may notice that there is no real difference between computing the answer sets and enumerating them. Therefore, deciding whether an atom is a member of the i-th answer set has the same complexity of deciding whether an atom is a member of some answer set. We can then easily state the following.

Proposition 3. *Given atom A, the problem of deciding whether $P(w_i, A)$ holds w.r.t. program Π is NP-complete.*

Proposition 4. *Given atom A, the problem of deciding whether $N(A)$ holds w.r.t. program Π is co-NP-complete.*

Notice however that the above complexity results refers to the ground version of the logic program included in an ASP module: in fact, this is always the case when one adopts ASP. Therefore, either one bases a module upon ground programs as we have assumed up to now, or it is necessary to be careful about possible exponential blowup of program size (e.g., by stating constraints on the program structure or by avoiding to introduce too many new constants in the input). In the present setting, the input and the context of modal ASP expressions are provided by the overall agent (logic) program, and we have seen in the examples that several modal expressions may occur in the same rule. However, we assume (cf. Definition 9) that every modal ASP expression can be evaluated whenever all its arguments are ground. Then, no interaction is possible due to conjunctions of modal atoms. Relaxing at least to some extent this limitation, i.e., allowing modal ASP expressions to return results rather than simply evaluate to true or false will be a subject of future work.

Notice also that the increase of complexity that one can find, e.g., in the approach of [42] due to the interaction between modules that invoke each other even recursively cannot be found here. This because in the present setting modules cannot be nested and do not interact: in fact, an agent uses the possibility and necessity operator without nesting in its main agent program. Thus, there is no possible interaction among different invocations of such operators. The topic of allowing the use of possibility and necessity operators within modal ASP modules, where a module is allowed to refer to another one (presumably in an acyclic fashion) and the topic of nesting of possibility and necessity will be interesting subjects of future work, but have not been tackled here.

Whenever a logical agent uses only possibility in the body of its rules, the resulting system fits into the framework of [68]. In fact, rules with the possibility operator in

[3] These results hold for normal ASP programs as defined in Section 2. If one considers additional constructs such as for instance disjunction, the complexity increases.

the body can easily be seen in their terminology as "bridge rules". The agent program under, e.g., the semantics defined in [29] and the invoked modules under the answer set semantics form, again in their terminology, a set of logics. Finally, the belief state composed of the semantics of the agent program and the answer sets of the invoked modules and selected by the possibility operator constitutes what they call an "equilibrium". However, the complexity is lower that the more general case that they consider, because bridge rules can be used in the agent program only.

7 Concluding Remarks

We have proposed a framework for integrating ASP modules into virtually any agent architecture so as to allow for complex reactivity, and for hypothetical reasoning based upon possibility and necessity. From the implementation point of view, the integration of such modules into logic-based architectures is straightforward. In fact, we have implemented the approach within the DALI interpreter. The implementation is described in detail in [69], and uses the DLV answer set solver [70].

Our approach is different from previous ones under several respects. To the best of our knowledge in fact, except for the approach of [28] in the context of action theories, there is no existing approach which is comparable with the proposed one. On the first place, ASP modules are adopted for empowering reasoning capabilities of logical intelligent agents. Then, they are exploited to introduce forms of complex reactivity. Finally, we allow an agent to reason about what is possible given the corpus of agent's knowledge. The forms of hypothetical reasoning which are allowed are interesting, and may be used to design real applications at a comparatively low complexity.

References

1. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)
2. Anger, C., Schaub, T., Truszczyński, M.: ASPARAGUS – the Dagstuhl Initiative. ALP Newsletter **17**(3) (2004) See `http://asparagus.cs.uni-potsdam.de`.
3. Leone, N.: Logic programming and nonmonotonic reasoning: From theory to systems and applications. In Baral, C., Brewka, G., Schlipf, J., eds.: Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007. (2007)
4. Truszczyński, M.: Logic programming for knowledge representation. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd International Conference, ICLP 2007. (2007) 76–88
5. Gelfond, M.: Answer sets. In: Handbook of Knowledge Representation, Chapter 7. Elsevier (2007)
6. Gelfond, M., Lifschitz, V.: Action languages. ETAI, Electronic Transactions on Artificial Intelligence (6) (1998)
7. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In Minker, J., ed.: Workshop on Logic-Based Artificial Intelligence, Kluwer Academic Publishers (2001) 257–279
8. Baral, C., McIlraith, S., Son, T.C.: Formulating diagnostic problem solving using an action language with narratives and sensing. In: Proc. of the Int. Conference on the Principles of Knowledge Representation and Reasoning (KRR'00). (2000) 311–322

9. Balduccini, M.: Answer Set Based Design of Highly Autonomous, Rational Agents. PhD thesis (2005)
10. Balduccini, M., Gelfond, M.: The AAA architecture: An overview. In: AAAI Spring Symposium 2008 on Architectures for Intelligent Theory-Based Agents (AITA08). (2008)
11. Kowalski, R.A., Sadri, F.: From logic programming towards multi-agent systems. Annals of Mathematics and Artificial Intelligence **25**((3-4)) (1999) 391–419
12. Gelfond, G., Watson, R.: Modeling cooperative multi-agent systems. In Costantini, S., Watson, R., eds.: Proc. of ASP2007, 4th International Workshop on Answer Set Programming at ICLP07. (2007)
13. De Vos, M., Vermeir, D.: Extending answer sets for logic programming agents. Annals of Mathematics and Artifical Intelligence, Special Issue on Computational Logic in Multi-Agent Systems **42**(1-3) (2004) 103–139
14. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In: Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002. LNAI 2424, Springer-Verlag, Berlin (2002) 50–61
15. Alferes, J.J., Dell'Acqua, P., Pereira, L.M.: A compilation of updates plus preferences. In Flesca, S., Greco, S., Leone, N., Ianni, G., eds.: Logics in Artificial Intelligence. LNAI 2424, Berlin, Springer-Verlag (2002) 62–74
16. Baral, C., Gelfond, G., Son, T.C., Pontelli, E.: Using answer set programming to model multi-agent scenarios involving agents' knowledge about other's knowledge. In: Proc. of the 9th Int. Conference on Autonomous Agents and Multiagent Systems (AAMAS-10), Copyright 2010 by the International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS) (2010)
17. Rao, A.S., Georgeff, M.: Modeling rational agents within a bdi-architecture. In: Proc. of the Second Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR'91), Morgan Kaufmann (1991) 473–484
18. Rao, A.S.: Agentspeak(l): BDI agents speak out in a logical computable language. In: Agents Breaking Away: Proc. of the Seventh Europ. Worksh. on Modelling Autonomous Agents in a Multi-Agent World. Number 1038 in Lecture Notes in Artificial Intelligence, Springer-Verlag (1996)
19. Hindriks, K.V., de Boer, F., van der Hoek, W., Meyer, J.C.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems **2**(4) (1999)
20. Fisher, M.: Metatem: The story so far. In Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E., eds.: PROMAS. Volume 3862 of Lecture Notes in Computer Science., Springer (2005) 3–22
21. Subrahmanian, V.S., Bonatti, P., Dix, J., Eiter, T., Kraus, S., Ozcan, F., Ross, R.: Heterogeneous Agent Systems. MIT Press/AAAI Press, Cambridge, MA, USA (2000)
22. Costantini, S., Tocchio, A.: A logic programming language for multi-agent systems. In: Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf.,JELIA 2002. LNAI 2424, Springer-Verlag, Berlin (2002)
23. Costantini, S., Tocchio, A.: The DALI logic programming agent-oriented language. In: Logics in Artificial Intelligence, Proc. of the 9th European Conference, Jelia 2004. LNAI 3229, Springer-Verlag, Berlin (2004)
24. Kakas, A.C., Mancarella, P., Sadri, F., Stathis, K., Toni, F.: The KGP model of agency. In: Proc. ECAI-2004. (2004)
25. Bracciali, A., Demetriou, N., Endriss, U., Kakas, A., Lu, W., Mancarella, P., Sadri, F., Stathis, K., Terreni, G., Toni, F.: The KGP model of agency: Computational model and prototype implementation. In: Global Computing: IST/FET International Workshop, Revised Selected Papers. LNAI 3267. Springer-Verlag, Berlin (2005) 340–367

26. Fisher, M., Bordini, R.H., Hirsch, B., Torroni, P.: Computational logics and agents: a road map of current technologies and future trends. Computational Intelligence Journal **23**(1) (2007) 61–91
27. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. In Gabbay, D., Hogger, C., Robinson, A., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming. Volume 5. Oxford University Press (1998) 235–324
28. Baral, C., Son, T.: Relating theories of actions and reactive control. ETAI (Electronic transactions of AI) **2**((3-4)) (1998) 211–271
29. Costantini, S., Tocchio, A.: About declarative semantics of logic-based agent languages. In Baldoni, M., Torroni, P., eds.: Declarative Agent Languages and Technologies. LNAI 3904. 106–123
30. Costantini, S., Tocchio, A.: DALI: An architecture for intelligent logical agents. In: Proc. of the Int. Workshop on Architectures for Intelligent Theory-Based Agents (AITA08). AAAI Spring Symposium Series, Stanford, USA, AAAI Press (2008)
31. Costantini, S., D'Alessandro, S., Lanti, D., Tocchio, A.: Dali web site, download of the interpreter (2010) With the contribution of many undergraduate and graduate students of Computer Science, L'Aquila.
32. implementations, A.: Web references for some ASP solvers
ASSAT: `http://assat.cs.ust.hk`;
Ccalc: `http://www.cs.utexas.edu/users/tag/ccalc`;
Clasp: `http://www.cs.uni-potsdam.de/clasp`;
Cmodels: `http://www.cs.utexas.edu/users/tag/cmodels`;
DeReS and aspps: `http://www.cs.uky.edu/ai/`;
DLV: `http://www.dbai.tuwien.ac.at/proj/dlv`;
Smodels: `http://www.tcs.hut.fi/Software/smodels`.
33. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering **1**(1) (1989) 146166
34. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: Answer set programming with constraints using lazy grounding. In Hill, P., Warren, D., eds.: Logic Programming, 25st International Conference, ICLP 2009, Proceedings. Volume 5649 of LNCS., Springer (2009)
35. Lefèvre, C., Nicolas, P.: A first order forward chaining approach for answer set computing. In: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR'09. LNCS, Springer-Verlag (2009)
36. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proc. of the 5th Intl. Conference and Symposium on Logic Programming, The MIT Press (1988) 1070–1080
37. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
38. Costantini, S.: On the existence of stable models of non-stratified logic programs. J. on Theory and Practice of Logic Programming **6**(1-2) (2006)
39. Oikarinen, E.: Modularity in Answer Set Programs. Doctoral dissertation, TKK Dissertations in Information and Computer Science TKK-ICS-D7, Helsinki University of Technology, Faculty of Information and Natural Sciences, Department of Information and Computer Science, Espoo, Finland (2008) ISBN 978-951-22-9581-4.
40. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. In Baral, C., Brewka, G., Schlipf, J., eds.: Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007. (2007)
41. Veith, H., Eiter, T., Eiter, T., Gottlob, G.: Modular logic programming and generalized quantifiers. In: Proc. of the 4th Int. Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-97). Number 1265 in LNCS, Berlin, Springer-Verlag (1997) 290–309

42. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Modular nonmonotonic logic programming revisited. In: Proc. of the 25th International Conference on Logic Programming, ICLP 2009. (2009) 145–159
43. Tari, L., Baral, C., Anwar, S.: A language for modular answer set programming: Application to acc tournament scheduling. In: Proc. of the Int.Workshop on Answer Set Programming ASP'05. (2005) 277–292 Volume 142 of CEUR Workshop Proceedings.
44. Balduccini, M.: Modules and signature declarations for a-prolog: Progress report. In: Proc. of the Software Engineering for Answer Set Programming Workshop (SEA07). (2007)
45. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls, and use of ensembles in modular answer set programming. In: Proc. of Int. Conference on Logic Programming, ICLP06. Number 4079 in LNCS, Berlin, Springer-Verlag (2006) 376–390
46. Calimeri, F., Ianni, G.: Template programs for disjunctive logic programming: An operational semantics. AI Communications **19**
47. Faber, W., Woltran, S.: Manifold answer-set programs for meta-reasoning. In: Proc. of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR09. (2009) 115–128
48. Gelfond, M.: Logic programming and reasoning with incomplete information. Annals of Mathematics and Artificial Intelligence **12** (1994)
49. Zhang, Y.: Computational properties of epistemic logic programs. In: Principles of Knowledge Representation and Reasoning: Proc. of the 10th Int. Conference (KR2006), AAAI Press (2006) 308–317
50. Zhang, Y.: Epistemic reasoning in logic programs. In: Proc. of the 20th International Joint Conference On Artificial Intelligence, IJCAI07. (2007) 647–652
51. El-Khatib, O., Pontelli, E., Son, T.C.: Asp-prolog: a system for reasoning about answer set programs in prolog. In: Proc. of the 10th International Workshop on Non-Monotonic Reasoning, NMR2004. (2004) 155–163
52. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: Proc. of the 19th International Joint Conference On Artificial Intelligence, IJCAI05. (2005) 90–96
53. Costantini, S., Lanzarone, G.A.: A metalogic programming language, Cambridge, Mass., The MIT Press (1989) 218–233
54. Barklund, J., Dell'Acqua, P., Costantini, S., Lanzarone, G.A.: Reflection principles in computational logic. J. of Logic and Computation **10**(6) (2000) 743–786
55. Eiter, T., Subrahmanian, V., Pick, G.: Heterogeneous active agents, i: Semantics. Artificial Intelligence **108**(1-2) (1999) 179–255
56. Luck, M., McBurney, P., Preist, C.: A manifesto for agent technology: Towards next generation computing. Autonomous Agents and Multi-Agent Sytems **9** (2004) 203–252
57. Fisher, M., Bordini, R.H., Hirsch, B., Torroni, P.: Computational logics and agents: a road map of current technologies and future trends. Computational Intelligence Journal **23**(1) (2007) 61–91
58. Costantini, S., Tocchio, A., Toni, F., Tsintza, P.: A multi-layered general agent model. In: AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, 10th Congress of the Italian Association for Artificial Intelligence. LNCS 4733, Springer-Verlag, Berlin (2007)
59. Kakas, A.C., Mancarella, P., Sadri, F., Stathis, K., Toni, F.: Declarative agent control. In: Post-proc. CLIMA V. Volume 3487 of LNAI., Springer Verlag (2005)
60. Kowalski, R., Sadri, F.: Towards a unified agent architecture that combines rationality with reactivity. In: Logic in Databases. Number 1154 in Lecture Notes in Computer Science. Springer-Verlag (1996) 135–149
61. Costantini, S., Dell'Acqua, P., Tocchio, A.: Expressing preferences declaratively in logic-based agent languages. In: Proc. of Commonsense'07, the 8th International Symposium

on Logical Formalizations of Commonsense Reasoning. AAAI Spring Symposium Series, AAAI Press (2007) a special event in honor of John McCarthy, Stanford University, March 2007.
62. Gabbay, D.M., Smets, P.: Handbook of Defeasible Reasoning and Uncertainty Management Systems. Kluwer Academic Publishers (2000) edited collection.
63. Blackburn, P., van Benthem, J., Wolter, F.: Handbook of Modal Logic. Elsevier, Amsterdam (2006) collection of contributions.
64. Van Gelder, A., Ross, K.A., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM (3) (1990)
65. Bowen, K.A., Kowalski, R.A.: Amalgamating language and metalanguage in logic programming. In Clark, K.L., Tärnlund, S.Å., eds.: Logic Programming. Academic Press, London (1982) 153–172
66. Marek, V.W., Truszczyński, M.: Autoepistemic logic. Journal of the ACM **38**(3) (1991) 587–618
67. Marek, V.W., Truszczyński, M.: Computing intersection of autoepistemic expansions. In: Proceedings of the First International Workshop on Logic Programming and Non Monotonic Reasoning, The MIT Press (1991) 35–70
68. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proc. of the 22nd Conference on Artificial Intelligence, AAAI-07. (2007) 385–390
69. Nisar, M.A.: Integration of answer set programming modules with logical agents. Master's thesis, University of the Punjab, Lahore, Pakistan (2010) Supervisor Prof. Stefania Costantini, University of L'Aquila, Italy.
70. Eiter, T., Faber, W., Leone, N., Pfeifer, G. In: Declarative problem-solving using the DLV system. Kluwer Academic Publishers, USA (2000) 79–103