The AAA Architecture: An Overview

Marcello Balduccini[*] and **Michael Gelfond**
Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
marcello.balduccini@gmail.com, michael.gelfond@ttu.edu

Abstract

This paper describes the AAA architecture for intelligent agents reasoning about, and acting in, a changing environment. The architecture is based on a simple control loop. Both the description of the domain's behavior and the reasoning components are written in Answer Set Prolog. The architecture is designed to make the agents capable of planning and of detecting, interpreting, and recovering from, unexpected observations. Overall, the design and the knowledge bases are *elaboration tolerant*. Another distinguishing feature of the architecture is that *the same domain description is shared by all the reasoning components*.

Introduction

In this paper we describe the *AAA architecture* for intelligent agents capable of reasoning about, and acting in, a changing environment.[1]

The AAA architecture is used for the design and implementation of software components of such agents and is applicable if: (1) The world (including an agent and its environment) can be modeled by a transition diagram whose nodes represent physically possible states of the world and whose arcs are labeled by actions. The diagram therefore contains all possible trajectories of the system; (2) The agent is capable of making correct observations, performing actions, and remembering the domain history; (3) *Normally* the agent is capable of observing all relevant exogenous events occurring in its environment. The agent, whose memory contains knowledge about the world and agents' capabilities and goals,

1. Observes the world, explains the observations, and updates its knowledge base;
2. Selects an appropriate goal, G;
3. Finds a plan (sequence of actions $a_1, \ldots, a_n$) to achieve G;
4. Executes part of the plan, updates the knowledge base, and goes back to step 1.

[*]Present address: Eastman Kodak Company, Rochester, NY 14650-2204 USA

[1]AAA stands for "Autonomous Agent Architecture."

The loop is called the *Observe-Think-Act Loop*. The knowledge of the AAA agent is encoded by a knowledge base in knowledge representation language Answer Set Prolog (ASP) or its extensions (Gelfond & Lifschitz 1991; Balduccini & Gelfond 2003b; Baral 2003). ASP is selected because of its ability to represent various forms of knowledge including defaults, causal relations, statements referring to incompleteness of knowledge, etc. This contributes to making the overall design and knowledge bases *elaboration tolerant* (McCarthy 1998). A knowledge base (or program) of ASP describes a collection of *answer sets* – possible sets of beliefs of a rational agent associated with it. The agent's reasoning tasks, including those of explaining unexpected observations and planning, can be reduced to computing (parts of) answer sets of various extensions of its knowledge base. In the case of original ASP, such computation can be rather efficiently performed by ASP solvers, which implement sophisticated grounding algorithms and suitable extensions of the Davis-Putnam procedure. Solvers for various extensions of ASP expand these reasoning mechanisms by abduction (Balduccini 2007a), constraint solving algorithms and resolution (Mellarkod & Gelfond 2007), and even some forms of probabilistic reasoning (Gelfond, Rushton, & Zhu 2006).

This architecture was suggested in (Baral & Gelfond 2000). Most of its refinements were modular (Balduccini & Gelfond 2003a; Balduccini, Gelfond, & Nogueira 2006; Balduccini 2007b). Throughout the paper we illustrate the architecture and its use for agent design using the scenarios based on the electrical circuit described below. The example is deliberately simple but we hope it is sufficient to illustrate the basic ideas of the approach. It is important to note, though, that the corresponding algorithms are scalable. In fact, they were successfully used in rather large, industrial size applications (Balduccini, Gelfond, & Nogueira 2006).

The rest of the paper is organized as follows. We begin by describing the behavior of a simple circuit. Next, we discuss how the agent finds plans and explanations for unexpected observations. Finally, we give a brief summary of the semantics of ASP and conclude the paper.

Building the Action Description

The electrical circuit used in this paper is depicted in Figure 1. Circuit C_0 consists of a battery (*batt*), two safety switches (sw_1, sw_2), and two light bulbs (b_1, b_2). By *safety switches*

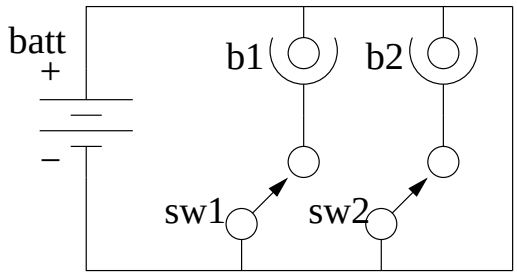

Figure 1: C_0: A simple electrical circuit

we mean switches with a locking device. To move a switch from its current position, the switch must first be unlocked. The switch is automatically locked again after it is moved. If all the components are working properly, closing a switch causes the corresponding bulb to light up. Next, we describe in more detail how we model the circuit, and introduce some useful terminology.

The state of the circuit is modeled by the following *fluents* (properties whose truth changes over time): $closed(SW)$: switch SW is closed; $locked(SW)$: switch SW is locked; $on(B)$: bulb B is on; $ab(B)$: bulb B is malfunctioning; $down(BATT)$: battery $BATT$ has run down. When a fluent f is false, we write $\neg f$.

The agent interacts with the circuit by means of the following *actions*: $flip(SW)$: move switch SW from open to closed, or vice-versa; $unlock(SW)$: unlock SW; $replace(BATT)$; $replace(B)$: replace the battery or a bulb.

Sometimes actions occur in the domain that are not controlled by the agent (e.g., a bulb blowing up). These actions are called *exogenous*. Relevant exogenous actions for this domain are: $run\_down(BATT)$: battery $BATT$ runs down; $blow\_up(B)$: B blows up. Note that actions can occur concurrently. We distinguish between *elementary actions*, such as the ones listed above, and *compound actions*, i.e. sets of elementary actions, intuitively corresponding to the concurrent execution of their components. In the rest of this paper we abuse notation slightly and denote singletons by their unique component. Similarly, we use the term "action" to denote both elementary and compound actions.

The behavior of the domain is described by laws. Depending on the approach used, the laws can be written using *action languages* (Gelfond & Lifschitz 1998) and later translated to ASP, or encoded directly in ASP. For simplicity, in the examples in this paper we use the direct encoding in ASP. A possible encoding of the effects of actions $unlock(SW)$ and $close(B)$ is:

$$\neg holds(locked(SW), S+1) \leftarrow occurs(unlock(SW), S).$$

$$holds(closed(SW), S+1) \leftarrow occurs(flip(SW), S), \neg holds(closed(SW), S).$$

$$\neg holds(closed(SW), S+1) \leftarrow occurs(flip(SW), S), holds(closed(SW), S).$$

$$holds(on(B), S) \leftarrow holds(closed(SW), S), connected(SW, B), \neg holds(ab(B), S), \neg holds(down(batt), S).$$

where SW, B, S are variables ranging, respectively, over switches, bulbs, and non-negative integers denoting steps in the evolution of the domain. The first law intuitively states that unlocking SW causes it to become unlocked. Laws describing, such as this, the *direct effects* of actions are sometimes referred to as *dynamic laws*. The second and third laws encode the effect of flipping a switch. The last law says that, if SW is closed and connected to some bulb B in working order while the battery is not down, then B is lit. Note that this law describes the *indirect effect, or ramification*, of an action. Such laws are sometimes called *static laws* or *state constraints*.

Similar laws encode the effects of the other actions, as well as the behavior of malfunctioning bulbs and battery. The encoding of the model is completed by the following general-purpose axioms:

$$holds(F, S+1) \leftarrow holds(F, S), \text{not } \neg holds(F, S+1).$$
$$\neg holds(F, S+1) \leftarrow \neg holds(F, S), \text{not } holds(F, S+1).$$

where F ranges over fluents, S over steps. The rules encode the principle of inertia "things normally stay as they are."

Planning Scenario

Let us now look at a scenario in which the main reasoning component is planning. We will use this scenario to illustrate how planning is performed in the AAA architecture.

Example 1 *Initially, all the bulbs of circuit C_0 are off, all switches open and locked, and all the components are working correctly. The agent wants to turn on b_1.*

The intended agent behavior is the following. At step 1 of the Observe-Think-Act loop, the agent gathers observations about the environment. In general, the observations need not be complete, or taken at every iteration. Let us assume however for simplicity that at the first iteration of the agent's observations are complete. At step 2, the agent selects goal $G = on(b_1)$. At step 3, it looks for a plan to achieve G and finds $\langle unlock(sw_1), flip(sw_1)\rangle$. Next, the agent executes $unlock(sw_1)$, records the execution of the action, and goes back to observing the world.

Suppose the agent observes that sw_1 is unlocked. Then, no explanations for the observations are needed. The agent proceeds through steps 2 and 3, and selects the plan $\langle flip(sw_1)\rangle$. Next, it executes $flip(sw_1)$ and observes the world again. Let us assume that the agent indeed finds out that b_1 is lit. Then, the agent's goal is achieved.

The key feature that allows to exhibit the behavior described above is in the capability to find a sequence of actions $\langle a_1, \ldots, a_k\rangle$ that achieves G. The task involves both *selecting* the appropriate actions, and *ordering* them suitably. For example, the sequence of actions $\langle unlock(sw_2), flip(sw_1)\rangle$ is not a good selection, while $\langle flip(sw_1), unlock(sw_1)\rangle$ is improperly ordered.

To determine if a sequence of actions achieves the goal, the agent uses its knowledge of the domain to predict the effect of the execution of the sequence. This is accomplished by reducing planning to computing answer sets of an ASP program, consisting of the ASP encoding of the domain model, together with a set of rules informally stating

that the agent can perform any action at any time (see e.g. (Lifschitz 1999; Nogueira *et al.* 2001)).

This technique relies on the fact that the answer sets of the ASP encoding of the domain model together with facts encoding the initial situation and occurrence of actions are in one-to-one correspondence with the corresponding paths in the transition diagram. This result, as well as most of the results used in this and the next section, are from (Balduccini & Gelfond 2003a). We invite the interested reader to refer to that paper for more details. Simple iterative modifications of the basic approach allow one to find shortest plans, i.e. plans that span the smallest number of steps.

To see how this works in practice, let us denote by *AD* the action description from the previous section, and consider a simple encoding, O_1, of the initial state from Example 1, which includes statements

$$holds(locked(sw_1),0), \neg holds(closed(sw_1),0), \\ \neg holds(on(b_1),0)$$

(more sophisticated types of encoding are possible.). A simple yet general *planning module* PM_1, which finds plans of up to *n* steps, consists of the rule:

$$occurs(A,S) \text{ OR } \neg occurs(A,S) \leftarrow cS \leq S < cS+n.$$

where *A* ranges over agent actions, *S* over steps, and *cS* denotes the current step (0 in this scenario). Informally, the rule says that any agent action *A* may occur at any of the next *n* steps starting from the current one. The answer sets of the program $\Pi_1 = AD \cup O_1 \cup PM_1$ encode all of the possible trajectories, of length *n*, from the initial state. For example, the trajectory corresponding to the execution of $unlock(sw_1)$ is encoded by the answer set:

$$O_1 \cup \{occurs(unlock(sw_2),0), \neg holds(locked(sw_2),1), \\ \neg holds(on(b_1),1), \ldots\}.$$

Note that $\neg holds(on(b_1),1)$ is obtained by O_1 and the inertia axioms.

To eliminate the trajectories that do not correspond to plans, we add to Π_1 the following rules,

$$goal\_achieved \leftarrow holds(on(b1),S). \\ \leftarrow \text{not } goal\_achieved.$$

which informally say that goal $on(b1)$ must be achieved. Let us denote the new program by Π_1'. It is not difficult to see that the previous set of literals is not an answer set of Π_1'. On the other hand, (if $n \geq 2$) Π_1' has an answer set containing

$$O_1 \cup \{occurs(unlock(sw_1),0), \neg holds(locked(sw_1),1), \\ occurs(flip(sw_1),1), holds(closed(sw_1),2), \\ holds(on(b_1),2)\},$$

which encodes the trajectory corresponding to the execution of the sequence $\langle unlock(sw_1), flip(sw_1)\rangle$.

Interpretation Scenario

To illustrate how a AAA agent interprets its observations about the world, let us consider the following example.

Example 2 *Initially, all the bulbs of circuit C_0 are off, all switches open and locked, and all the components are working correctly. The agent wants to turn on b_1. After planning, the agent executes the sequence $\langle unlock(sw_1), flip(sw_1)\rangle$, and notices that b_1 is not lit.*

The observation is unexpected, as it contradicts the effect of the actions the agent just performed. A possible explanation for this discrepancy is that b_1 blew up while the agent was executing the actions (recall that all the components were initially known to be working correctly). Another explanation is that the battery ran down.[2]

To find out which explanation corresponds to the actual state of the world, the agent will need to gather additional observations. For example, to test the hypothesis that the bulb blew up, the agent will check the bulb. Suppose it is indeed malfunctioning. Then, the agent can conclude that $blow\_up(b_1)$ occurred in the past. The fact that b_1 is not lit is finally explained, and the agent proceeds to step 3, where it re-plans.

The component responsible for the interpretation of the observations, often called *diagnostic component*, is described in detail in (Balduccini & Gelfond 2003a). Two key capabilities are needed to achieve the behavior described above: the ability to detect unexpected observations, and that of finding sequences of actions that, had they occurred undetected in the past, may have caused the unexpected observations. These sequences of actions correspond to our notion of *explanations*.

The detection of unexpected observations is performed by checking the consistency of the ASP program, Π_d, consisting of the encoding of the domain model, together with the history of the domain, and the Reality Check Axioms and Occurrence-Awareness Axiom, both shown below. The history is encoded by statements of the form $obs(F,S,truth\_val)$ (where $truth\_val$ is either t or f, intuitively meaning "true" and "false") and $hpd(A,S)$, where F is a fluent and A an action. An expression $obs(F,S,t)$ (respectively, $obs(F,S,f)$) states that F was observed to hold (respectively, to be false) at step S. An expression $hpd(A,S)$ states that A was observed to occur at S. The Reality Check Axioms state that it is impossible for an observation to contradict the agent's expectations:

$$\leftarrow holds(F,S), obs(F,S,f). \\ \leftarrow \neg holds(F,S), obs(F,S,t).$$

Finally, the Occurrence-Awareness Axiom ensures that the observations about the occurrences of actions are reflected in the agent's beliefs:

$$occurs(A,S) \leftarrow hpd(A,S).$$

It can be shown that program Π_d is inconsistent if-and-only-if the history contains unexpected observations.

To find the explanations of the unexpected observations, the agent needs to search for sequences of exogenous actions that would cause the observations (possibly indirectly), if

[2] Of course, it is always possible that the bulb blew up *and* the battery ran down, but we do not believe this should be the first explanation considered by a rational agent.

they had occurred in the past. A simple diagnostic module DM_1 is the one consisting of the rule:

$$occurs(E,S) \text{ OR } \neg occurs(E,S) \leftarrow S < cS.$$

where E ranges over exogenous actions, and S, cS are as in the planning module. Informally, the rule says that any exogenous action E may have occurred at any time in the past.

To see how DM_1 works, consider the program Π_2 consisting of AD and the encoding of the initial state O_1 from the previous section[3], together with the Occurrence-Awareness Axiom, module DM_1, and the history

$$H_1 = \{hpd(unlock(sw_1),0), hpd(flip(sw_1),1), \\ obs(on(b_1),2,f)\}.$$

It can be shown that the answer sets of Π_2 are in one-to-one correspondence with all the trajectories from the initial state that include all the actions that the agent has observed, plus a number of additional exogenous actions. That is, the answer sets of Π_2 will encode the trajectories corresponding to the sequences of actions:

$\langle \{unlock(sw_1), run\_down(batt)\}, flip(sw_1)\rangle$
$\langle unlock(sw_1), \{flip(sw_1), blow\_up(b_2)\}\rangle$
$\langle \{unlock(sw_1), blow\_up(b_2)\}, \{flip(sw_1), blow\_up(b_1)\}\rangle$
$\vdots$

Note that the first and third sequences of actions explain the observation about b_1 from H_1, while the second one does not. The sequences of actions that do not explain the unexpected observations can be discarded by means of the Reality Check Axiom. Let Π_2' consist of Π_2 and the Reality Check Axiom. It is not difficult to see that no answer set of Π_2' encodes the second sequence, while there are answer sets encoding the first and third.

It should be noted that some of the explanations found by DM_1 are not minimal (in set-theoretic sense). For example, the third explanation above is not minimal, because removing $blow\_up(b_2)$ yields another valid explanation. Diagnostic algorithms have been developed that extend the approach shown here to find minimal diagnoses. Another technique, which we discuss in the next section, avoids the use of such algorithms by employing a recent extension of ASP for the formalization of exogenous actions.

It is worth noticing that the technique described here to interpret observations is remarkably similar to that used for planning. In fact, the interpretation of observations is essentially reduced to "planning in the past." More importantly, *the planning and diagnostic reasoning components share the same knowledge about the domain*.

To see the interplay between interpretation of the observations and planning, consider the following scenario.

Example 3 *Initially, all bulbs are off, all switches open and locked, and all the components are working correctly. The agent wants to turn on b_1 and b_2. The agent is also given the additional constraint that* bulbs cannot be replaced while they are powered.

[3]The initial state described by O_1 could be re-written to use statements $obs(F,S,truth\_val)$, but that is out of the scope of the paper.

A sequence of actions expected to achieve this goal is $\langle unlock(sw_1), flip(sw_1), unlock(sw_2), flip(sw_2)\rangle$. Let us suppose that this is the plan found at step 3 of the Observe-Think-Act loop. The agent will then execute part of the plan – suppose, $unlock(sw_1)$ – and observe the world again. Assuming that there are no unexpected observations, the agent proceeds with the rest of the plan[4] and executes $flip(sw_1)$. This time, the agent finds that b_1 is not lit and hypothesizes that b_1 blew up at step 1. To store this piece of information, it then adds a statement $hpd(blow\_up(b_1),1)$ to the history of the domain.[5]

The agent now looks for a new plan. Because of the additional constraint on bulb replacement, the agent will have to flip sw_1 open before replacing b_1. A possible plan that achieves the goal from the new state of the world is:

$$\langle flip(sw_1), replace(b_1), flip(sw_1), unlock(sw_2), flip(sw_2)\rangle.$$

The agent then proceeds with the execution of the plan, and, assuming that no other unexpected observations are encountered, will eventually achieve the goal.

CR-Prolog to Interpret the Observations

In this section we discuss a modification of the AAA architecture based on a different technique for the interpretation of the agent's observations. This technique allows a more elegant representation of exogenous actions, which includes the formalization of information about relative the likelihood of their occurrence, and guarantees that all of the explanations returned by the diagnostic module are minimal (in set-theoretic sense).

The approach is based on the use of the extension of ASP called CR-Prolog (Balduccini & Gelfond 2003b; Balduccini 2007a). In CR-Prolog, programs consist of regular ASP rules, and of *cr-rules* and preferences over cr-rules. A cr-rule is a statement of the form:

$$r : l_0 \stackrel{+}{\leftarrow} l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n.$$

where r is the name of the cr-rule and l_i's are ASP literals. The rule says "if $l_1, \ldots, l_m$ hold and there is no reason to believe $l_{m+1}, \ldots, l_n$, l_0 *may possibly* hold, but that happens rarely." Informally, this possibility should be used only if the regular rules alone are not sufficient to form a consistent set of beliefs. In the CR-Prolog terminology, we say that cr-rules are used to *restore consistency*. Preferences are atoms of the form $prefer(r_1, r_2)$, where r_1 and r_2 are cr-rule names. The statement informally means that r_2 should be considered only if using r_1 does not restore consistency.

To see how cr-rules work, consider the answer sets of the program $P_1 = \{p \leftarrow q.\ \ p \leftarrow u.\ \ s \leftarrow \text{not } s, \text{not } p.\ \ r_1 :$

[4]In the AAA architecture, checking for the need to re-plan can be reduced to checking if any unexpected observations were detected.

[5]Storing in the history the conclusions obtained during the interpretation of observations can cause problems if evidence collected at later iterations of the loop invalidates the hypothesis, but we will not discuss more sophisticated methods of recording history because of space considerations.

$q \stackrel{+}{\leftarrow}$ not $t.\}$. Because the regular rules of P_1 alone are inconsistent,[6] r_1 is applied, yielding the (unique) answer set $\{q, p\}$. On the other hand, let us consider the program, P_2, obtained by adding rule $\{u.\}$ to P_1. Now the regular rules are sufficient to form a consistent set of beliefs ($\{u, p\}$). Therefore, the cr-rule is not applied, and the answer set of P_2 is $\{u, p\}$. The program $P_4 = P_1 \cup \{r_2 : u \stackrel{+}{\leftarrow} .\}$ has two answer sets, $\{q, p\}$, $\{u, p\}$, because either cr-rule can be applied (but not both, because that would involve the unnecessary application of one of them). Finally, the program $P_5 = P_4 \cup \{prefer(r_1, r_2).\}$ has only one answer set, $\{q, p\}$, because the preference statement prevents r_2 from being considered if r_1 restores consistency.

Cr-rules and preferences are particularly useful in encoding information about unlikely events, such as exogenous actions. As described in (Balduccini, Gelfond, & Nogueira 2006), an exogenous action e can be formalized in CR-Prolog by one or more cr-rules of the form:

$$r(e,S) : occurs(e,S) \stackrel{+}{\leftarrow} \Gamma. \quad (1)$$

where Γ is a condition under which the exogenous action may occur. The rule informally states that, under those conditions, the exogenous action may possibly occur, but that is a rare event. Let us now see how it is possible to encode the relative likelihood of the occurrence of exogenous actions. Let us consider two exogenous actions e_1 and e_2. To formalize the fact that e_1 is more likely to occur than e_2, we write $prefer(r(e_1,S), r(e_2,S))$.

Because cr-rules are applied only if needed, testing for unexpected observations and generating an explanation can be combined in a single step. Going back to Example 2, let EX be the set of cr-rules:

$$r(run\_down(BATT),S) : occurs(run\_down(BATT),S) \stackrel{+}{\leftarrow} .$$
$$r(blow\_up(B),S) : occurs(blow\_up(B),S) \stackrel{+}{\leftarrow} .$$

informally stating that $run\_down(BATT)$ and $blow\_up(B)$ may possibly (but rarely) occur. Consider now program Π_3 obtained from Π_2 by replacing DM_1 by EX. It is not difficult to show that the answer sets of Π_3 correspond to the *minimal* explanations of the observations in H_1. If no unexpected observations are present in H_1, the answer sets will encode an empty explanation.

The preference statements of CR-Prolog can be used to provide information about the relative likelihood of the occurrence of the exogenous actions. For example, the fact that $run\_down(BATT)$ is more likely than $blow\_up(B)$ can be encoded by a statement

$$\{prefer(r(run\_down(BATT),S), r(blow\_up(B),S)).\}.$$

To see how all this works, consider program Π_4, obtained by adding the above preference statement to Π_3. It is not difficult to show that Π_4 has only two answer sets, encoding the *preferred, minimal* explanations corresponding to the sequences of actions:

$$\langle \{unlock(sw_1), run\_down(batt)\}, flip(sw_1) \rangle$$
$$\langle unlock(sw_1), \{flip(sw_1), run\_down(batt)\} \rangle.$$

[6] Inconsistency follows from the third rule and the fact that p is not entailed.

It is worth noticing that the non-monotonic nature of CR-Prolog makes it possible, for explanations, which had previously not been considered (because non-minimal or less-preferred), to be selected when new information becomes available. For example, if we update P_4 to include additional information that the battery is not down, we obtain two (different) answer sets, encode the explanations corresponding to the less-preferred explanations:

$$\langle \{unlock(sw_1), blow\_up(b_1)\}, flip(sw_1) \rangle$$
$$\langle unlock(sw_1), \{flip(sw_1), blow\_up(b_1)\} \rangle.$$

In the next section, we discuss the specification of policies in the AAA architecture.

Policies and Reactivity

In this paper, by *policy* we mean the description of those paths in the transition diagram that are not only possible but also *acceptable* or *preferable*.

The ability to specify policies is important to improve both the quality of reasoning (and acting) and the agent's capability to *react* to the environment.

In this paper we show how the AAA architecture, and in particular the underlying ASP language, allows one to easily specify policies addressing both issues.

We begin by considering policies allowing one to improve the quality of reasoning. More details can be found in (Balduccini 2004; Balduccini, Gelfond, & Nogueira 2006). In the circuit domain, one policy addressing this issue could be "do not replace a good bulb." The policy is motivated by the consideration that, although technically possible, the action of replacing a good bulb should in practice be avoided. A possible ASP encoding of this policy is:

$$\leftarrow occurs(replace(B),S), \neg holds(ab(B),S).$$

Note that, although the rule has the same form of the executability conditions, it is conceptually very different. Also note that this is an example of a *strict* policy because it will cause the action of replacing a good bulb to be *always* avoided.

Often it is useful to be able to specify *defeasible policies*, that is policies that are *normally* complied with but may be violated if really necessary. Such policies can be elegantly encoded using CR-Prolog. For example, a policy stating "if at all possible, do not have both switches in the closed position at the same time" can be formalized as:

$$\leftarrow holds(closed(sw_1),S), holds(closed(sw_2),S),$$
$$\quad \text{not } can\_violate(p_1).$$
$$r(p_1) : can\_violate(p_1) \stackrel{+}{\leftarrow} .$$

The first rule says that the two switches should not be both in the closed position *unless the agent can violate the policy*. The second rule says that it is possible to violate the policy, but only if strictly necessary (e.g., when no plan exists that complies with the policy).

Now let us turn our attention to policies improving the agent's *capability to react to the environment*. When an agent is interacting with a changing domain, it is often important for the agent to be able to perform some actions in

response to a state of the world. An example is leaving the room if a danger is spotted. Intuitively, selecting the actions to be performed should come as an immediate reaction rather than as the result of sophisticated reasoning. We distinguish two types of reaction: *immediate reaction* and *short reaction*. Immediate reaction is when actions are selected based solely on observations. An example from the circuit domain is the statement "if you observe a spark from closed switch *SW*, open it," which can be formalized as (assume *SW* is unlocked and fluent $spark\_from(SW)$ is available):

$$occurs(flip(SW),S) \leftarrow obs(spark\_from(SW),S), \\ obs(closed(SW),S).$$

A short reaction is the occurrence of an action triggered by the agent's beliefs. This includes conclusions *inferred* from observations and possibly other beliefs. An example is the statement "if a battery is failing, replace it," encoded as:

$$occurs(replace(BATT),S) \leftarrow holds(failing(BATT),S).$$

Appendix: Semantics of ASP

In this section we summarize the semantics of ASP. Recall that an ASP rule is a statement of the form

$$h_1 \text{ OR } \ldots \text{ OR } h_k \leftarrow l_1,\ldots,l_m,\text{not } l_{m+1},\ldots,\text{not } l_n.$$

where h_i's and l_i's are literals (atoms or their strong negation, e.g. $\neg a$). The intuitive meaning of the rule is "if $l_1 \ldots l_m$ hold and there is no reason to believe $l_{m+1} \ldots l_n$, a rational reasoner should believe one of h_i's." Given a rule r, we call $\{h_1 \ldots h_k\}$ the *head* of r, denoted by $head(r)$; $l_1,\ldots,l_m,l_{m+1},\ldots,l_n$ is called the *body* of r ($body(r)$); $pos(r)$ and $neg(r)$, respectively, denote $\{l_1,\ldots,l_m\}$ and $\{l_{m+1},\ldots,l_n\}$. A *program* is a set of rules. A *default-negation-free program* is a program whose rules do not contain default negation "not." We say that a set of literals S is *closed under a default-negation-free program* Π if, for every rule r of Π, $head(r) \cap S \neq \emptyset$ whenever $pos(r) \subseteq S$. A set of literals S is *consistent* if it does not contain two complementary literals $f, \neg f$. A consistent set of literals S is an answer set of a default-negation-free program Π if S is the smallest set closed under Π. Given an arbitrary program Π and a set S of literals, the *reduct* Π^S is obtained from Π by deleting: (1) each rule, r, such that $neg(r) \setminus S \neq \emptyset$, and (2) all formulas of the form not l in the bodies of the remaining rules. A set of literals S is an *answer set* of Π if it is an answer set of Π^S.

Conclusions

In this paper we have described the ASP-based AAA architecture for intelligent agents capable of reasoning about and acting in changing environments. The design is based on a description of the domain's behavior that is shared by all of the reasoning modules. We hope we demonstrated how the agent's ability to generate and execute plans is interleaved with detecting, interpreting, and recovering from, unexpected observations. Although in this paper we focused on explaining unexpected observations by hypothesizing the undetected occurrence of exogenous actions, the architecture has also been extended with reasoning modules capable of modifying the domain description by means of inductive learning (Balduccini 2007b). An initial exploration of the issues of inter-agent communication and cooperation can be found in (Gelfond & Watson 2007). A prototype of the implementation of the architecture can be found at: `http://krlab.cs.ttu.edu/~marcy/APLAgentMgr/`.

Acknowledgments: The development of the AAA architecture was supported in part by NASA contract NASA-NNG05GP48G and ATEE/DTO contract ASU-06-C-0143.

References

Balduccini, M., and Gelfond, M. 2003a. Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)* 3(4–5):425–461.

Balduccini, M., and Gelfond, M. 2003b. Logic Programs with Consistency-Restoring Rules. In Doherty, P.; McCarthy, J.; and Williams, M.-A., eds., *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, 9–18.

Balduccini, M.; Gelfond, M.; and Nogueira, M. 2006. Answer Set Based Design of Knowledge Systems. *Annals of Mathematics and Artificial Intelligence*.

Balduccini, M. 2004. USA-Smart: Improving the Quality of Plans in Answer Set Planning. In *PADL'04*, Lecture Notes in Artificial Intelligence (LNCS).

Balduccini, M. 2007a. CR-MODELS: An Inference Engine for CR-Prolog. In *LPNMR 2007*, 18–30.

Balduccini, M. 2007b. Learning Action Descriptions with A-Prolog: Action Language C. In Amir, E.; Lifschitz, V.; and Miller, R., eds., *Procs of Logical Formalizations of Commonsense Reasoning, 2007 AAAI Spring Symposium.*

Baral, C., and Gelfond, M. 2000. Reasoning Agents In Dynamic Domains. In *Workshop on Logic-Based Artificial Intelligence*, 257–279. Kluwer Academic Publishers.

Baral, C. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 365–385.

Gelfond, M., and Lifschitz, V. 1998. Action Languages. *Electronic Transactions on AI* 3(16).

Gelfond, G., and Watson, R. 2007. Modeling Cooperative Multi-Agent Systems. In *Proceedings of ASP'07*, 67–81.

Gelfond, M.; Rushton, N.; and Zhu, W. 2006. Combining Logical and Probabilistic Reasoning. In *AAAI 2006 Spring Symposium*, 50–55.

Lifschitz, V. 1999. *Action Languages, Answer Sets, and Planning*. The Logic Programming Paradigm: a 25-Year Perspective. Springer Verlag, Berlin. 357–373.

McCarthy, J. 1998. Elaboration Tolerance.

Mellarkod, V. S., and Gelfond, M. 2007. Enhancing ASP Systems for Planning with Temporal Constraints. In *LPNMR 2007*, 309–314.

Nogueira, M.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 2001. An A-Prolog decision support system for the Space Shuttle. In *PADL 2001*, 169–183.