Playing with Rules

João Leite

CENTRIA, Universidade Nova de Lisboa, Portugal
jleite@di.fct.unl.pt

Abstract. In this paper we revisit Logic Programming under the answer-set semantics - or *Answer-Set Programming* - and its extension *Evolving Logic Programming*, two languages that use logic rules and rule updates and exhibit characteristics that make them suitable to be used for knowledge representation and reasoning within Agent Oriented Programming Languages. We illustrate the power of these rule based languages by means of examples showing how several of its features can be used to model situations faced by Agents.

1 Introduction

Recently, there has been considerable amount of research in designing Agent Oriented Programming Languages (AOPL). From the definition of the syntax and semantics to the development of the necessary tools and infrastructure (editors, debuggers, environments, etc.), the results of such efforts are key in turning these AOPLs into practical tools which exhibit the necessary level of maturity to make them compete with more established languages, and help widespread the Agent Oriented Programming paradigm. The collections of papers [8,9] reflect the state of the art on this subject.

In most cases, the designers of Agent Oriented Programming Languages have focused on developing an appropriate syntax which promotes to first class citizens notions such as beliefs, goals, intentions, plans, events, messages, etc., and defining a semantics which, for most cases, concentrates on the interaction between these notions and the definition of more or less complex transition rules which define the behaviour of the agent. These two tasks are not trivial ones. On the one hand, defining and fixing a particular syntax faces the big tension between flexibility and ease of use: fewer prescribed syntactical constructs, as often seen in declarative languages, usually provide greater flexibility but are harder to use, while more prescribed syntactical constructs are usually easier to use, at the cost of flexibility. On the other hand, defining a semantics that properly deals with these high level notions (beliefs, intentions, plans, goals, etc...), is efficient, to some extent declarative, and keeps some desirable (theoretical) principles is a very hard task.

When looking at existing AOPLs, one often sees that in order to achieve acceptable results in what concerns the appropriate balance between the definition of a syntax with high level agent oriented notions and an appropriate semantics, their developers need to make some compromises, often in the form of simplifications in the expressiveness of the languages used for specifying the underlying high level notions, with consequences in what can be expressed and how the agent can behave. For example, if

M. Baldoni et al. (Eds.): DALT 2009, LNAI 5948, pp. 1–19, 2010.

we consider the representation of beliefs, it is often the case that the knowledge representation languages used are not more expressive than a set of horn clauses, and belief revision/update mechanisms amount to the simple addition/retraction of facts. In practice, it is often the case that PROLOG is used to represent and reason about knowledge, and implementations diverge from the theoretical definitions, often with unforeseen, unintended and unintuitive results.

There is substantial literature illustrating the need for knowledge representation languages that are richer than those limited to horn clauses, e.g. exhibiting non-monotonicity [10,23], and richer belief change operators that permit not only the update of the extensional part of existing knowledge (the facts), but also the intensional part (the rules) [19,5]. And these should be accompanied by implementations that accurately implement the theory, which should preferably be (at least) decidable. The incorporation of richer forms of knowledge representation in current AOPLs should be investigated, not only to allow agent's beliefs to be more expressive, but also to specify other agent high level notions such as their goals (e.g. to represent conditional goals and their updates), richer forms of intentions, etc.

In this paper we revisit Logic Programming under the answer-set semantics [15] - or *Answer-Set Programming* - and its extension *Evolving Logic Programming*, [3], two languages with characteristics that make them suitable to be used for knowledge representation and reasoning within AOPLs, and illustrate some of their characteristics by means of examples.

Answer-Set Programming (ASP) is a form of declarative programming that is similar in syntax to traditional logic programming and close in semantics to non-monotonic logic, that is particularly suited for knowledge representation. Some of the most important characteristics of ASP include: the use of default negation to allow for reasoning about incomplete knowledge; a very intuitive semantics based on multiple answer-sets for reasoning about several possible consistent worlds; possibility to compactly represent all NP and coNP problems if non-disjunctive logic programs are used, while disjunctive logic programs under answer sets semantics capture the complexity class Σ_2^P [14]; fully declarative character in the sense that the program specification resembles the problem specification; the existence of a number of well studied extensions such as preferences, revision, abduction, etc. Enormous progress concerning the theoretical foundations of ASP (c.f. [7] for more) have been made in recent years, and the existence of very efficient ASP solvers (e.g. DLV and SMODELS) has made it possible to use it in real applications such as Decision Support for the Space Shuttle [24], Automated Product Configuration [26], Heterogeneous Data Integration [20], Inferring Phylogenetic Trees [11], Resource Allocation [17], as well as Reasoning about actions, Legal Reasoning, Games, Planning, Scheduling, Diagnosis, etc.

While ASP can be seen as a good representation language for static knowledge, if we are to move to a more open and dynamic environment, typical of the agency paradigm, we must consider ways and means of representing and integrating knowledge updates from external as well as internal sources. In fact, an agent should not only comprise knowledge about each state, but also knowledge about the transitions between states. The latter may represent the agent's knowledge about the environment's evolution, coupled to its own behaviour and evolution. The lack of rich mechanisms to

represent and reason about dynamic knowledge and agents i.e. represent and reason about environments where not only some facts about it change, but also the rules that govern it, and where the behaviours of agents also change, is common to most existing AOPLs.

To address this issue the paradigm of *Evolving Logic Programming* (*EVOLP*) was introduced in [3][1]. In a nutshell, *EVOLP* is a simple though quite powerful extension of ASP [15] that allows for the specification of a program's evolution, in a single unified way, by permitting rules to indicate assertive conclusions in the form of program rules. Syntactically, evolving logic programs are just generalized logic programs[2]. But semantically, they permit to reason about updates of the program itself. The language of *EVOLP* contains a special predicate $assert/1$ whose sole argument is a full-blown rule. Whenever an assertion $assert\,(r)$ is true in a model, the program is updated with rule r. The process is then further iterated with the new program. These assertions arise both from self (i.e. internal to the program) updating, and from external updating originating in the environment. EVOLP can adequately express the semantics resulting from successive updates to logic programs, considered as incremental specifications of agents, and whose effect can be contextual. Whenever the program semantics allows for several possible program models, evolution branching occurs, and several evolution sequences are made possible. This branching can be used to specify the evolution of a situation in the presence of incomplete information. Moreover, the ability of *EVOLP* to nest rule assertions within assertions allows rule updates to be themselves updated down the line. Furthermore, the *EVOLP* language can express self-modifications triggered by the evolution context itself, present or future – $assert$ literals included in rule bodies allow for looking ahead on some program changes and acting on that knowledge before the changes occur. In contradistinction to other approaches, *EVOLP* also automatically and appropriately deals with the possible contradictions arising from successive specification changes and refinements (via Dynamic Logic Programming[3] [19,5,2]).

In this paper we start by revisiting EVOLP and illustrate how its features, many of which inherited from ASP, seem appropriate to represent and reason about several aspects related to Agents and Agent Oriented Programming.

The remainder of the paper is structured as follows: In Sect. 2 we present a very simple agent architecture that will be used in the remainder of the paper. In Sect. 3 we illustrate many features of EVOLP and ASP in this context, and we draw some conclusions and pointers to future work in Sect. 4. In Appendix A we recap the syntax and semantics of EVOLP.

[1] There are implementations of EVOLP [25] available at http://centria.di.fct.unl.pt/~jja/updates.

[2] Logic programs that allow for rules with default negated literals in their heads.

[3] *Dynamic Logic Programming* determines the semantics of sequences of generalized logic programs representing states of the world at different time periods, i.e. knowledge undergoing successive updates. As individual theories may comprise mutually contradictory as well as overlapping information, the role of *DLP* is to employ the mutual relationships among different states to determine the declarative semantics, at each state, for the combined theory comprised of all individual theories. Intuitively, one can add newer rules at the end of the sequence, leaving to *DLP* the task of ensuring that these rules are in force, and that previous ones are valid (by inertia) only so far as possible, i.e. they are kept for as long as they are not in conflict with newly added ones, these always prevailing.

2 Simple Agent Architecture

In this Section we define a very simple agent architecture with the purpose of facilitating the illustration of EVOLP as a language for knowledge representation and reasoning to be used in the context of Agent Oriented Programming. Accordingly, we will only be concerned with the mental part of the agent, namely the representation of beliefs, agent behaviour and epistemic effects of actions.

To this purpose, we assume that an agent's initial specification will be given by an EVOLP program P over some language $\mathcal{A}$, which will be used to concurrently represent the beliefs, behaviour and epistemic effects of actions. To simplify our presentation, we assume the existence of a set $\mathcal{A}_{\mathcal{C}} \subseteq \mathcal{A}$ of propositions, each representing an action that the agent is capable of preforming, and a set $\mathcal{A}_{do(\mathcal{C})} \subseteq \mathcal{A}$ such that $\mathcal{A}_{do(\mathcal{C})} = \{do(\alpha) : \alpha \in \mathcal{A}_{\mathcal{C}}\}$ and $\mathcal{A}_{do(\mathcal{C})} \cap \mathcal{A}_{\mathcal{C}} = \emptyset$. Propositions of the form $do(\alpha)$ will be used to indicate that some current belief state prescribes (or permits) the execution of action α, somehow encoding the agent's behaviour. Propositions of the form $\alpha \in \mathcal{A}_{\mathcal{C}}$ will indicate the actual execution of action α, possibly causing some update representing the epistemic effects of executing it. Furthermore we assume that at each state the agent perceives a set of events which, besides any observations from the environment or incoming messages, will also include a sub-set of $\mathcal{A}_{\mathcal{C}}$ representing the actions that were just performed, effectively allowing the agent to update its beliefs accordingly. Finally, we assume some selection function ($Sel()$) that, given a set of stable models (at some state), selects the set of actions to be executed[4]. Accordingly, at each state, and agent mental state is characterised by its initial specification and the sequence of events it has perceived so far.

Definition 1. *An agent state is a pair $\langle P, \mathcal{E}\rangle$ where P is an evolving logic program and $\mathcal{E}$ an event sequence, both over language $\mathcal{A}$.*

An agent evolves into the next state as per the following observe-think-act cycle and definition:

$cycle(\langle P, \mathcal{E}\rangle)$
  $observe$ (percieve E from inbox)
  $think$ (determine $SM(\langle P, (\mathcal{E}, E)\rangle)$)
  act (execute actions $Sel(SM(\langle P, (\mathcal{E}, E)\rangle))$)
  $cycle(\langle P, (\mathcal{E}, E)\rangle)$

---

3 Playing with Rules

In this section we illustrate how some of the features of EVOLP, many of which inherited from ASP, can be used to represent and reason about several aspects related to

[4] Throughout this paper, we assume that the selection function returns all actions belonging to one of the stable models, non-deterministically chosen from the set of stable models provided as its input. Other possible selection functions include returning the actions that belong to all stable models, those that belong to at least one stable model, or some more complex selection procedure e.g. based on priorities between actions/models.

Agents and Agent Oriented Programming. This illustration will be done incrementally through the elaboration of the specification of an agent whose purpose is to control the access to a building of some company with several floors. Some basic facts about this scenario include:

– the existence of 4 floors, numbered 0 through 3, represented by the following facts in P:

$$floor\,(0)\,.\quad floor\,(1)\,.\quad floor\,(2)\,.\quad floor\,(3)\,. \tag{1}$$

– the existence of 4 people, named Birna, John, Jamal and Matteo, represented by the following facts in P:

$$person\,(birna)\,.\quad person\,(john)\,.\quad person\,(jamal)\,.\quad person\,(matteo)\,. \tag{2}$$

– John, Jamal and Matteo are employees, represented by the following facts in P:

$$employee\,(john)\,.\quad employee\,(jamal)\,.\quad employee\,(matteo)\,. \tag{3}$$

– John is the company's director, represented by the following fact in P:

$$director\,(john)\,. \tag{4}$$

Deductive Reasoning. Rules in ASP provide an easy way to represent deductive knowledge. Suppose, for example, that according to the initial access policy of the building, any person who is employed by the company has permission to access any floor. This can be represented in ASP through the following deductive rule in P:

$$permit(P, F) \leftarrow person(P), floor(F), employee(P). \tag{5}$$

It is trivial to see that P has the following stable model (given an empty event sequence)[5]:

$\{f\,(0)\,, f\,(1)\,, f\,(2)\,, f\,(3)\,, p\,(birna)\,, p\,(john)\,, p\,(jamal)\,, p\,(matteo)\,,$
$e\,(john)\,, e\,(jamal)\,, e\,(matteo)\,, d\,(john)\,,$
$permit\,(0, john)\,, permit\,(1, john)\,, permit\,(2, john)\,, permit\,(3, john)\,,$
$permit\,(0, jamal)\,, permit\,(1, jamal)\,, permit\,(2, jamal)\,, permit\,(3, jamal)\,,$
$permit\,(0, matteo)\,, permit\,(1, matteo)\,, permit\,(2, matteo)\,, permit\,(3, matteo)\}$

It is interesting to note how rules in ASP can be seen as a query language (or a way to define views). If we interpret predicates as relations (e.g. $person/1$ as representing the relation $person$ with one attribute and four tuples: $birna$, $john$, $jamal$ and $matteo$) then, intuitively, the head of a rule represents the relation (view) obtained from the natural inner join of the relations in its body (when no negation is present).

[5] With the obvious abbreviations.

Reactive Behaviour. The same kind of deductive rules, when coupled with the previously presented agent cycle, can be used to express the reactive behaviour of an agent. For example, to express that any request to access some floor should result in the action of opening the corresponding door if the requester has a permit for that floor, we can include the following rule in P:

$$do\left(open\_door\left(F\right)\right) \leftarrow person\left(P\right), floor\left(F\right), permit\left(P,F\right), request\left(P,F\right). \tag{6}$$

If the agent perceives the event sequence $\mathcal{E} = (E_1, E_2)$ with:

$$\begin{array}{l} E_1 = \{request\left(matteo,3\right). \qquad request\left(birna,1\right)\} \\ E_2 = \{request\left(john,2\right).\} \end{array}$$

Then, $do\left(open\_door\left(3\right)\right)$ will belong to the only stable model at time 1, resulting in the execution of action $open\_door\left(3\right)$, while $do\left(open\_door\left(2\right)\right)$ will belong to the only stable model at time 2, resulting in the execution of action $open\_door\left(2\right)$. Note that $request\left(birna,1\right)$ will not cause the execution of any action since $permit\left(birna,1\right)$ is not true. Furthermore, since events do not persist by inertia (i.e. they are only used to determine the stable models at the state they were perceived and are ignored at subsequent states) each only triggers the execution of the actions once.

Effects of Actions. According to the simple agent architecture being used, we assume that the agent is aware of the actions it (actually) executes, inserting them at the subsequent set of events[6]. In the previous example, the action $open\_door\left(3\right)$ would actually belong to E_2, while action $open\_door\left(2\right)$ would belong to E_3, together with other observed events. We can take advantage of the inclusion of these actions in the events to express their epistemic effects, through ASP rules. For example, if we want to express that while we open the door, the door is open, we can include the following rule in P:

$$open\left(F\right) \leftarrow open\_door\left(F\right). \tag{7}$$

It is easy to see that $open\left(3\right)$ is true at state 2 while $open\left(2\right)$ is true at state 3.

Persistent Effects of Actions. In the previous example, the effect of $open\_door\left(F\right)$ does not persist i.e. $open\left(F\right)$ is only true while the action $open\_door\left(F\right)$ is true. Often we need to make the effects of actions persist. This can be done using the $assert$ predicate provided by EVOLP. If we want $open\_door\left(F\right)$ to cause the door to be open and stay open, we include the following rule in P:

$$assert\left(open\left(F\right)\right) \leftarrow open\_door\left(F\right). \tag{8}$$

[6] Since the actions actually executed depend on the selection function, the presence of $do\left(\alpha\right)$ in some (even every) model is not a guarantee that the action is going to be executed, e.g. because the selection function only selects a sub-set of those actions. This can also be used to filter those actions whose execution failed (e.g. a hardware error when moving the arm of the robot) by not inserting them in the events.

Now, $open\,(3)$ is not only true at state 2, but also at subsequent states. If we wish to close the door when someone enters the open door, provided there is no other action to open it, we can add the rule:

$$do\,(close\_door\,(F)) \leftarrow open\,(F)\,, enters\,(P, F)\,, not\, open\_door\,(F)\,. \tag{9}$$

together with the effects of closing the door:

$$assert\,(not\, open\,(F)) \leftarrow close\_door\,(F)\,. \tag{10}$$

Considering also these new rules, if $enters\,(matteo, 3)$ is observed at state 5, then $do\,(close\_door\,(3))$ also belongs to the stable model at that state, causing the action $close\_door\,(3)$ to be performed with the consequent update with $not\, open\,(3)$ at state 6, so that $open\,(3)$ will not be true from state 7, until another action opens the door again.

Default Reasoning. One of the main characteristics of ASP is its use of default negation (not) permitting, among other things, to non-monotonically reason with the Closed World Assumption. This is useful to represent exceptions. For example, if we wish to permit access to the ground floor to all people, except those on some black list, we can add the following rule to P:

$$permit(P, 0) \leftarrow person(P), not\, black\_list(P). \tag{11}$$

The default $not\, black\_list(P)$ evaluates to true whenever $black\_list(P)$ is not in the stable model. For example, if $request\,(birna, 0)$ belongs to some event at state n, then $do\,(open\_door\,(0))$ will be true at that state causing the door to be open, because $black\_list(birna)$ is not true (by default).

Open and Closed World Assumptions. ASP permits, besides default negation (not), strong negation ($\neg$). Strong negation behaves just as positive atoms i.e. in order for some $\neg A$ to evaluate to true, there must be some rule with head $\neg A$ and whose body is true, unlike $not\, A$ whose only requirement is that A is not true in the model[7]. For example, if we wish to implement an access policy where anyone not known to be a terrorist could access the first floor, we could add the following rule to P:

$$permit(P, 1) \leftarrow person(P), not\, terrorist(P). \tag{12}$$

With this rule, Birna, not known to be a terrorist in our example, would have permission to access the first floor i.e. $permit(birna, 1)$ is true. If we wish to impose a stricter policy for granting access to the second floor, where one is only allowed if (s)he is known to not be a terrorist, then we could add the following rule to P:

$$permit(P, 2) \leftarrow person(P), \neg terrorist(P). \tag{13}$$

[7] Recall that for a language $\mathcal{A}$ with one propositional atom A, there are three possible interpretations: $\{A\}\,, \{\neg A\}$ and $\{\}$, indicating, respectively, that A is true, A is (strongly) false, and A is neither true nor (strongly) false. $\neg A$ evaluates to true in the second interpretation while $not\, A$ evaluates to true in both the second and the third interpretations.

With this rule, Birna, would not have permission to access the second floor i.e. $permit\ (birna, 2)$ is false since $\neg terrorist(birna)$ is not true. in order to gain access to the second floor, the agent would have to know that $\neg terrorist(birna)$ is true (e.g. by conducting an investigation to clear Birna, leading to the assertion of $\neg terrorist(birna)$).

It is possible to turn strong negation into default negation. In our example, if we wish to represent that we should conclude $\neg terrorist(P)$ whenever $not\ terrorist(P)$ is true, we simply add the rule:

$$\neg terrorist(P) \leftarrow person(P), not\ terrorist(P). \tag{14}$$

This would turn the policies for both the first and second floors equivalent.

Non-deterministic Choice of Actions. In ASP, default negation, together with a special pair of rules known as *even loop through default negation*, allow for the generation of two (or more) stable models, which can be exploited to generate possible alternative behaviours. For example, if, for every person entering the ground floor, we wish to either perform a body search or simply ask for an id, we can add the following pair of rules to P:

$$\begin{array}{l} do\,(body\_search(P)) \leftarrow enters\,(P, 0)\,, not\ do\,(ask\_id(P))\,. \\ do\,(ask\_id(P)) \leftarrow enters\,(P, 0)\,, not\ do\,(body\_search(P))\,. \end{array} \tag{15}$$

If $enters\,(jamal, 0)$ is observed at some state, then, at that state, there will be two stable models:

$$\begin{array}{c} \{enters\,(jamal, 0)\,, do\,(body\_search(jamal))\,, ...\} \\ \{enters\,(jamal, 0)\,, do\,(ask\_id(jamal))\,, ...\} \end{array}$$

encoding both possible agent behaviours.

Uncertainty. The *even loop through default negation* and the generation of several models can also be used to represent uncertainty. For example, if we wish to state that everyone is either a friend or a terrorist, but we do not know which, we can add the following two rules to P:

$$\begin{array}{l} terrorist(P) \leftarrow person(P), not\ friend(P). \\ friend(P) \leftarrow person(P), not\ terrorist(P). \end{array} \tag{16}$$

These rules will generate many stable models, with all possible combinations where each person is either a friend or a terrorist. Some od these models include:

$$\begin{array}{c} \{friend(jamal), friend(birna), friend(matteo), friend(john), ...\} \\ \{friend(jamal), terrorist(birna), friend(matteo), friend(john), ...\} \\ \{friend(jamal), friend(birna), terrorist(matteo), terrorist(john), ...\} \\ \{terrorist(jamal), friend(birna), friend(matteo), friend(john), ...\} \\ ... \end{array}$$

Each of these models can be seen as a possible state of the world. Each of these worlds could prescribe some particular behaviour i.e. have some predicates of the form $do\,(\alpha)$

true them. However, this use of ASP would require a different selection function since different models would no longer encode different possible courses of action, as before, but rather different possible worlds in which different courses of action would be appropriate. Whereas in the former a selection function that chooses one model would be appropriate (as each can be seen as one possible course of action), in the latter the agent is unsure about which model represents the real world and simply flipping a coin to choose one of them may yield the incorrect choice (we may choose the second model above and treat Birna as a terrorist when in fact it turns out not to be the case). A selection function that returns the actions that are true in all models is probably a more appropriate choice, with natural consequences in the way knowledge is represented.

Integrity Constraints. When using the *even loop through default negation* to generate several models, we may want to eliminate some of them given the knowledge we have. This is easily done through the use of integrity constraints, which are rules of the form

$$\leftarrow L_1, \ldots, L_n. \tag{17}$$

and prevent any interpretation in which $L_1, \ldots, L_n$ is true to be a stable model[8]. For example, if we wish to state that John is not a terrorist, we simply add the following integrity constraint to P:

$$\leftarrow terrorist\,(john)\,. \tag{18}$$

This would eliminate the third model above, as well as any other models in which $terrorist\,(john)$ was true.

Uncertainty, as encoded through the even loop through default negation, can also be eliminated through the existence of factual or deductive knowledge. For example, if we know that all employees are friends, then we add the following rule to P:

$$friend\,(P) \leftarrow employee(P). \tag{19}$$

This will make $friend(jamal)$,$friend(matteo)$ and $friend(john)$ true, eliminating some of the uncertainty generated by rules (16). The remanning models would be:

$$\{friend(jamal),friend(birna),friend(matteo),friend(john),...\}$$
$$\{friend(jamal),terrorist(birna),friend(matteo),friend(john),...\}$$

Problem Solving. The combination of the *even loop through default negation* and integrity constraints has been successfully employed as a general technique to use ASP to solve many problems (e.g. hamiltonian cycles, graph coloring, large clique, planning, resource allocation, etc...). It is possible to use such ASP programs within this agent architecture, providing agents with the ability to solve complex problems through simple declarative specifications. Given that ASP can encode planning problems, it is possible to encode, without requiring changes in the agent architecture, the implementation of a planning agent where each model would encode a plan and the agent would select one of them for execution. More about using ASP for planning and problem solving in general can be found in [21,7,12].

[8] The semantics of an Integrity Constraint of the form $\leftarrow L_1, \ldots, L_n$.is given first translating it into the (normal) rule $a \leftarrow not\ a, L_1, \ldots, L_n.$, where a is a reserved atom.

Rule Updates. So far, with the exception of persistent effects of actions, we have only focused on features of ASP. We now turn our attention to those features introduced by EVOLP that allow for updating the agent specification. The semantics of EVOLP provides an easy mechanism to update the knowledge of some agent. This is due, on the one hand, to the fact that events do not persist by inertia and, on the other hand, that rules inside assert predicates that are true at some state are used to update the agent's knowledge. Accordingly, if we wish to update some agent's program with some rule r, all we have to do is include, in the events, the predicate $assert(r)$. If we further want to make such update conditional on the current state of events e.g. on some condition $L_1, \ldots, L_n$ being true, we add, instead, the rule $assert(r) \leftarrow L_1, \ldots, L_n$. In our scenario, imagine that it has been decided that, from now on, unless for the director(s), no one should have permission to access the third floor. This can be achieved by adding the fact to the next set of external events[9]:

$$assert\,(not\,permit(P,3) \leftarrow person(P), not\,director(P))\,. \tag{20}$$

Note that it is substantially different to update the agent and to simply add the rule inside the assert predicate to P. Whereas if we simply add this rule to P we obtain a contradiction with rule (5), if we update P with this rule such contradiction is automatically detected and avoided by assigning preference to the rule for the director (the newer one) than to the rule for employees (the original one)[10]. After this update, no one but John will be allowed on the third floor, and permissions regarding other floors will still be governed by the previously stated policy.

To illustrate an update conditioned on the current state of events, let's suppose that, later on, we want to grant permission to all current directors to access the third floor, even after they cease to be directors. This can be done by including the following rule in the events:

$$assert\,(permit(P,3)) \leftarrow director(P). \tag{21}$$

With this rule, the knowledge base will be updated with $permit(P,3)$ for all people (P) that are currently directors of the company, in this case, John. note however that the verification of $director(P)$ is only performed at the current state, and the update is done with $permit(john,3)$. When John ceases to be a director, he will still be allowed in the third floor, despite the effect of rule (20) which would, before this latter update, mean that John would not be allowed in the third floor.

[9] Some security mechanisms should be added to prevent anyone from being able to update some agent from the outside. This can also be expressed in EVOLP, although outside the scope of this presentation.

[10] The reader may be thinking that the same effect could be achieved by replacing the original rule with one where there was an exception for the director. Despite being true in this case, it is arguably simpler, on the one hand, to state what we want to be the case from now on (i.e. no one except for the director should access the third floor), and let the semantics of EVOLP deal with the resulting conflicts, while on the other hand coming up with such refined rules may be, for more elaborate cases, a very difficult task requiring knowledge of all rules in the agent program.

External Agent Programming. Rule updates, as exemplified above, be them simple assertions (updates) of rules, or updates conditioned on the current state of affairs, provide a flexible mechanism to incrementally program agents after they have been deployed. All aspects of the agent that are programmed in EVOLP can be incrementally specified, and the agent specification completely changed as the agent progresses. Not only can we change the knowledge of the agent, but we can also update the agent behaviour (providing new, better, ways to react), the effects of the actions (even allowing the agent to learn new actions e.g. a robot was serviced with a new arm), and any other aspect specified in EVOLP. For example, we can update the behaviour of the agent exhibited when someone enters the ground floor (specified by rules 15) in a way such that directors are neither body searched nor asked for an id, by updating with the rules:

$$\begin{array}{l} not\, do\,(body\_search(P)) \leftarrow enters\,(P,0)\,, director(P). \\ not\, do\,(ask\_id(P)) \leftarrow enters\,(P,0)\,, director(P). \end{array} \tag{22}$$

We can update the effects of action *open_door* so that it does not open in case of malfunction, by updating with the rule (in the case where it's effect is persistent):

$$not\, assert\,(open\,(F)) \leftarrow open\_door\,(F)\,, malfunction(F). \tag{23}$$

Evolution. Assert predicates can be present in the agent program, specifying the way in which the agent should evolve (its knowledge, behaviour, etc...). Furthermore, the fact that assertions can be nested allows for expressing more complex evolutions. Imagine that we decide to implement a policy according to which, after the agent observes a (terrorist) attack, it starts behaving in a way that, after asking someone for an ID, it will treat that person as a terrorist if the person does not have an ID. Recall that $ask\_id(P)$ is an action (hence non-inertial), and $attack$ is an observation. this policy could be implemented with the following rule:

$$assert\,(assert\,(terrorist\,(P) \leftarrow not\, id(P)) \leftarrow ask\_id(P)) \leftarrow attack. \tag{24}$$

In the event of an attack (i.e. $attack$ belongs to some set of events), then the agent evolves to a new specification resulting from the update with the rule

$$assert\,(terrorist\,(P) \leftarrow not\, id(P)) \leftarrow ask\_id(P). \tag{25}$$

If, for example, Birna is asked for an ID, e.g. due to the non-deterministic choice of actions specified in rules (15), the agent is updated with the rule

$$terrorist\,(birna) \leftarrow not\, id(birna). \tag{26}$$

and, from then on, until revoked by some other update, Birna will be considered a terrorist if she doesn't have an ID.

Complex Behaviour. The possibility provided by EVOLP to include assert atoms in the body of rules, as well as actions, allows for the specification of more complex behaviours in a simple way. For example, if we wish to specify that some action should be

followed by some other action (e.g. that a body search should be reported), we simply include the rule:

$$do\left(report\left(P\right)\right) \leftarrow body\_search(P). \tag{27}$$

To specify that some action should always be performed together with some other action (e.g. arresting someone should be done together with informing the person of his rights), can be done with the rule:

$$do\left(read\_rights\left(P\right)\right) \leftarrow do\left(arrest\left(P\right)\right). \tag{28}$$

The ability to include assertive literals in rule bodies allows for looking ahead on some program changes and acting on that knowledge before the changes occur. For example, to inform someone that (s)he will be added to the company's black list, we use the rule:

$$do\left(inform\left(P\right)\right) \leftarrow assert\left(black\_list\left(P\right)\right). \tag{29}$$

It could be important to have a timer to allow for updates to depend on it. Such timer can be encoded in EVOLP with the rules

$$\begin{array}{l} assert\left(time\left(T+1\right)\right) \leftarrow time\left(T\right). \\ assert\left(not\, time\left(T\right)\right) \leftarrow time\left(T\right). \end{array} \tag{30}$$

together with the fact $time\left(0\right)$ in P.

With this clock, we can *easily* encode effects of actions (or reactive behaviours) that span over time, e.g. delayed effect of actions or behaviours, i.e. when the effect of some action (or observation) only occurs after a certain amount of time. For example, if we wish to close the door, ten time steps after it is opened (even if no one enters it as specified in rule (9)), we can add the rule:

$$assert\left(do\left(close\_door\left(F\right)\right) \leftarrow time\left(T+10\right)\right) \leftarrow open\_door\left(F\right), time\left(T\right). \tag{31}$$

Complex Effects of Actions. Many Agent Oriented Programming Languages are very limited in what concerns representing the effects of actions. It was shown that Logic Programming Update Languages, including EVOLP, are able to capture Action Languages $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, making them suitable for describing domains of actions [1]. This includes, for example, representing effects of executing parallel actions, non-deterministic effects of actions, etc.

Communication. Messages can be treated in a straight forward manner. On the one hand, sending a message is treated just as any other action i.e. through some predicate of the form $do\left(send\_msg\left(To, Type, Content\right)\right)$, possibly with some internal effect (e.g. recording the sent message). Similarly, incoming messages are just events (observations) of some agreed form, e.g. a predicate of the form $msg\left(From, Type, Content\right)$, and can be treated by the agent just as any other event. For example, to store (with a timestamp) all incoming messages sent by employees, the following rule could be used:

$$assert\left(msg\left(F, Ty, C, T\right)\right) \leftarrow msg\left(F, Ty, C\right), time\left(T\right), employee\left(F\right). \tag{32}$$

4 Conclusions, Current and Future Work

In this paper we revisited Logic Programming under the answer-set semantics - or *Answer-Set Programming* - and its extension *Evolving Logic Programming*, two languages that use logic rules and rule updates, and exhibit characteristics that make them suitable to be used for knowledge representation and reasoning within Agent Oriented Programming Languages. We illustrated the power of these rule based languages by means of examples showing how several of its features can be used to model situations faced by Agents.

The use of EVOLP has previously been illustrated in Role Playing Games to represent the dynamic behaviour of Non-Playing Characters [18] and Knowledge Bases to describe their update policies [13]. Furthermore, it was shown that Logic Programming Update Languages are able to capture Action Languages $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, making them suitable for describing domains of actions [1].

Even though we only presented EVOLP with its stable model based semantics, which means that it inherits the complexity of ASP i.e. it is in the NP class when only non-disjunctive programs are used, EVOLP also allows for the use of a well founded three valued semantics which is less expressive but permits more efficient top down polynomial proof procedures [27,6].

Extensions of EVOLP include the introduction of LTL-like temporal operators that allow more flexibility in referring to the history of the evolving knowledge base [4].

It was not the purpose of this paper to present yet another agent architecture and language to populate the already dense field of AOPLs. Instead, the purpose was to present several features of a well known language (ASP) and a more recent extension (EVOLP) in a way that brings forward the possible advantages of their usage within existing AOPLs. This is precisely the subject of ongoing research where ASP and rule updates are being used in conjunction with the AOPL GOAL [16].

References

1. Alferes, J.J., Banti, F., Brogi, A.: From logic programs updates to action description updates. In: Leite, J., Torroni, P. (eds.) CLIMA 2004. LNCS (LNAI), vol. 3487, pp. 52–77. Springer, Heidelberg (2005)
2. Alferes, J.J., Banti, F., Brogi, A., Leite, J.A.: The refined extension principle for semantics of dynamic logic programming. Studia Logica 79(1), 7–32 (2005)
3. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 50–61. Springer, Heidelberg (2002)
4. Alferes, J.J., Gabaldon, A., Leite, J.: Evolving logic programming based agents with temporal operators. In: IAT, pp. 238–244. IEEE, Los Alamitos (2008)
5. Alferes, J.J., Leite, J.A., Pereira, L.M., Przymusinska, H., Przymusinski, T.: Dynamic updates of non-monotonic knowledge bases. Journal of Logic Programming 45(1-3), 43–70 (2000)
6. Banti, F., Alferes, J.J., Brogi, A.: Well founded semantics for logic program updates. In: Lemaître, C., Reyes, C.A., González, J.A. (eds.) IBERAMIA 2004. LNCS (LNAI), vol. 3315, pp. 397–407. Springer, Heidelberg (2004)

7. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
8. Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.): Multi-Agent Programming: Languages, Platforms and Applications. Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15. Springer, Heidelberg (2005)
9. Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.): Multi-Agent Programming: Languages, Tools and Applications. Springer, Heidelberg (2009)
10. Brewka, G.: Nonmonotonic Reasoning: Logical Foundations of Commonsense. Cambridge University Press, Cambridge (1991)
11. Brooks, D.R., Erdem, E., Erdogan, S.T., Minett, J.W., Ringe, D.: Inferring phylogenetic trees using answer set programming. J. Autom. Reasoning 39(4), 471–511 (2007)
12. Garcia de la Banda, M., Pontelli, E. (eds.): ICLP 2008. LNCS, vol. 5366. Springer, Heidelberg (2008)
13. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: A framework for declarative update specifications in logic programs. In: Nebel, B. (ed.) IJCAI, pp. 649–654. Morgan Kaufmann, San Francisco (2001)
14. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Trans. Database Syst. 22(3), 364–418 (1997)
15. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: Warren, D., Szeredi, P. (eds.) Proceedings of the 7th international conference on logic programming, pp. 579–597. MIT Press, Cambridge (1990)
16. Hindriks, K.V.: Programming rational agents in goal. In: Bordini, et al. (eds.) [9], ch. 3
17. Leite, J., Alferes, J.J., Mito, B.: Resource allocation with answer-set programming. In: Sierra, C., Castelfranchi, C., Decker, K.S., Sichman, J.S. (eds.) AAMAS (1), IFAAMAS, pp. 649–656 (2009)
18. Leite, J., Soares, L.: Evolving characters in role playing games. In: Trappl, R. (ed.) Cybernetics and Systems, Proceedings of the 18th European Meeting on Cybernetics and Systems Research (EMCSR 2006), vol. 2, pp. 515–520. Austrian Society for Cybernetic Studies (2006)
19. Leite, J.A.: Evolving Knowledge Bases. IOS Press, Amsterdam (2003)
20. Leone, N., Greco, G., Ianni, G., Lio, V., Terracina, G., Eiter, T., Faber, W., Fink, M., Gottlob, G., Rosati, R., Lembo, D., Lenzerini, M., Ruzzi, M., Kalka, E., Nowicki, B., Staniszkis, W.: The infomix system for advanced integration of incomplete and inconsistent data. In: Özcan, F. (ed.) SIGMOD Conference, pp. 915–917. ACM, New York (2005)
21. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence 138(1-2), 39–54 (2002)
22. Lifschitz, V., Woo, T.Y.C.: Answer sets in general non-monotonic reasoning (preliminary report). In: Nebel, B., Rich, C., Swartout, W. (eds.) Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR 1992), pp. 603–614. Morgan Kaufmann, San Francisco (1992)
23. Makinson, D.: Bridges from Classical to Nonmonotonic Logic. College Publications (2005)
24. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An a-prolog decision support system for the space shuttle. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 169–183. Springer, Heidelberg (2001)
25. Slota, M., Leite, J.: Evolp: An implementation. In: Sadri, F., Satoh, K. (eds.) CLIMA VIII 2007. LNCS (LNAI), vol. 5056, pp. 288–298. Springer, Heidelberg (2008)
26. Tiihonen, J., Soininen, T., Niemelä, I., Sulonen, R.: A practical tool for mass-customising configurable products. In: Proceedings of the 14th International Conference on Engineering Design, pp. 1290–1299 (2003)
27. van Gelder, A., Ross, K.A., Schlipf, J.S.: Unfounded sets and well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)

A Evolving Logic Programming

A.1 Language

We start with the usual preliminaries. Let $\mathcal{A}$ be a set of propositional atoms. An objective literal is either an atom A or a strongly negated atom $\neg A$. A default literal is an objective literal preceded by not. A literal is either an objective literal or a default literal. A rule r is an ordered pair $H(r) \leftarrow B(r)$ where $H(r)$ (dubbed the head of the rule) is a literal and $B(r)$ (dubbed the body of the rule) is a finite set of literals. A rule with $H(r) = L_0$ and $B(r) = \{L_1, \ldots, L_n\}$ will simply be written as $L_0 \leftarrow L_1, \ldots, L_n$. A generalized logic program (*GLP*) P, in $\mathcal{A}$, is a finite or infinite set of rules. If $H(r) = A$ (resp. $H(r) = not\ A$) then $not\ H(r) = not\ A$ (resp. $not\ H(r) = A$). If $H(r) = \neg A$, then $\neg H(r) = A$. By the expanded generalized logic program corresponding to the GLP P, denoted by $\mathbf{P}$, we mean the GLP obtained by augmenting P with a rule of the form $not\ \neg H(r) \leftarrow B(r)$ for every rule, in P, of the form $H(r) \leftarrow B(r)$, where $H(r)$ is an objective literal. Two rules r and r' are conflicting, denoted by $r \bowtie r'$, iff $H(r) = not\ H(r')$.

An interpretation M of $\mathcal{A}$ is a set of objective literals that is consistent i.e., M does not contain both A and $\neg A$. An objective literal L is true in M, denoted by $M \vDash L$, iff $L \in M$, and false otherwise. A default literal $not\ L$ is true in M, denoted by $M \vDash not\ L$, iff $L \notin M$, and false otherwise. A set of literals B is true in M, denoted by $M \vDash B$, iff each literal in B is true in M.

An interpretation M of $\mathcal{A}$ is an answer set of a GLP P iff

$$M' = least\left(\mathbf{P} \cup \{not\ A \mid A \notin M\}\right) \tag{33}$$

where $M' = M \cup \{not\_A \mid A \notin M\}$, A is an objective literal, and $least(.)$ denotes the least model of the definite program obtained from the argument program by replacing every default literal $not\ A$ by a new atom not_A.

In order to allow for logic programs to evolve, we first need some mechanism for letting older rules be supervened by more recent ones. That is, we must include a mechanism for deletion of previous knowledge along the agent's knowledge evolution. This can be achieved by permitting default negation not just in rule bodies, as in extended logic programming, but in rule heads as well[22]. Furthermore, we need a way to state that, under some conditions, some new rule should be asserted in the knowledge base[11]. In EVOLP this is achieved by augmenting the language with a reserved predicate $assert/1$, whose sole argument is itself a full-blown rule, so that arbitrary nesting becomes possible. This predicate can appear both as rule head (to impose internal assertions of rules) as well as in rule bodies (to test for assertion of rules). Formally:

Definition 2. *Let $\mathcal{A}$ be a set of propositional atoms (not containing $assert/1$). The* extended language $\mathcal{A}_{assert}$ *is defined inductively as follows:*

[11] Note that asserting a rule in a knowledge base does not mean that the rule is simply added to it, but rather that the rule is used to update the existing knowledge base according to some update semantics, as will be seen below.

- *All propositional atoms in $\mathcal{A}$ are propositional atoms in $\mathcal{A}_{assert}$;*
- *If r is a rule over $\mathcal{A}_{assert}$ then $assert(r)$ is a propositional atom of $\mathcal{A}_{assert}$;*
- *Nothing else is a propositional atom in $\mathcal{A}_{assert}$.*

An evolving logic program *over a language $\mathcal{A}$ is a (possibly infinite) set of generalized logic program rules over $\mathcal{A}_{assert}$.*

Example 1. Examples of EVOLP rules are:

$$\begin{array}{l} assert\left(not\, a \leftarrow b\right) \leftarrow not\, c. \\ a \leftarrow assert\left(b \leftarrow\right). \\ assert\left(assert\left(a \leftarrow\right) \leftarrow assert\left(b \leftarrow not\, c\right), d\right) \leftarrow not\, e. \end{array} \tag{34}$$

Intuitively, the first rule states that, if c is false, then the rule $not\, a \leftarrow b$ must be asserted in the agent's knowledge base; the 2nd that, if the fact $b \leftarrow$ is going to be asserted in the agent's knowledge base, then a is true; the last states that, if e is false, then a rule must be asserted stating that, if d is true and the rule $b \leftarrow not\, c$ is going to be asserted then the fact $a \leftarrow$ must be asserted.

This language alone is enough to model the agent's knowledge base, and to cater, within it, for internal updating actions that change it. But self-evolution of a knowledge base is not enough for our purposes. We also want the agent to be aware of events that happen outside itself, and desire the possibility too of giving the agent update "commands" for changing its specification. In other words, we wish a language that allows for influence from the outside, where this influence may be: observation of facts (or rules) that are perceived at some state; assertion commands directly imparting the assertion of new rules on the evolving program. Both can be represented as EVOLP rules: the former by rules without the assert predicate in the head, and the latter by rules with it. Consequently, we shall represent outside influence as a sequence of EVOLP rules:

Definition 3. *Let P be an evolving program over the language $\mathcal{A}$. An* event sequence *over P is a sequence of evolving programs over $\mathcal{A}$.*

A.2 Semantics

In general, we have an EVOLP program describing an agent's initial knowledge base. This knowledge base may already contain rules (with asserts in heads) that describe some forms of its own evolution. Besides this, we consider sequences of events representing observation and messages arising from the environment. Each of these events in the sequence are themselves sets of EVOLP rules, i.e. EVOLP programs. The semantics issue is thus that of, given an initial EVOLP program and a sequence of EVOLP programs as events, to determine what is true and what is false after each of those events.

More precisely, the meaning of a sequence of EVOLP programs is given by a set of *evolution stable models*, each of which is a sequence of interpretations or states. The basic idea is that each evolution stable model describes some possible evolution of one initial program after a given number n of evolution steps, given the events in the sequence. Each evolution is represented by a sequence of programs, each program corresponding to a knowledge state.

The primordial intuitions for the construction of these program sequences are as follows: regarding head asserts, whenever the atom $assert(Rule)$ belongs to an interpretation in a sequence, i.e. belongs to a model according to the stable model semantics of the current program, then $Rule$ must belong to the program in the next state; asserts in bodies are treated as any other predicate literals.

The sequences of programs are treated as in Dynamic Logic Programming [19,5,2], a framework for specifying updates of logic programs where knowledge is given by a sequence of logic programs whose semantics is based on the fact that the most recent rules are set in force, and previous rules are valid (by inertia) insofar as possible, i.e. they are kept for as long as they do not conflict with more recent ones. In DLP, default negation is treated as in answer-set programming [15]. Formally, a *dynamic logic program* is a sequence $\mathcal{P} = (P_1, \ldots, P_n)$ of generalized logic programs and its semantic is determined by (c.f. [19,2] for more details):

Definition 4. *Let $\mathcal{P} = (P_1, \ldots, P_n)$ be a dynamic logic program over language $\mathcal{A}$. An interpretation M is a (refined) dynamic stable model of $\mathcal{P}$ at state s, $1 \leq s \leq n$ iff*

$$M' = least\left([\rho_s(\mathcal{P}) - Rej_s(M)] \cup Def_s(M)\right) \tag{35}$$

where:

$$\begin{array}{l} Def_s(M) = \{not\ A \mid \nexists r \in \rho(\mathcal{P}), H(r) = A, M \vDash B(r)\} \\ Rej_s(M) = \{r \mid r \in \mathbf{P}_i, \exists r' \in \mathbf{P}_j, i \leq j \leq s, r \bowtie r', M \vDash B(r')\} \end{array} \tag{36}$$

and A is an objective literal, $\rho_s(\mathcal{P})$ denotes the multiset of all rules appearing in the programs $\mathbf{P}_1, ..., \mathbf{P}_s$, and M' and $least(.)$ are as before. Let $DSM(\mathcal{P})$ denote the set of (refined) dynamic stable model of $\mathcal{P}$ at state n.

Intuitively, given an interpretation M, the set $Rej_s(M)$ contains those rules which are overridden by a newer conflicting rule whose body is true according to the interpretation M. The set $Def_s(M)$ contains default negations $not\ A$ of all unsupported atoms A, i.e., those atoms A for which there is no rule, in any program, whose body is true according to the interpretation M, which can thus be assumed false *by default*.

Before presenting the definitions that formalize the above intuitions of EVOLP, let us show some illustrative examples.

Example 2. Consider an initial program P containing the rules

$$\begin{array}{l} a. \\ assert(not\ a \leftarrow) \leftarrow b. \\ c \leftarrow assert(not\ a \leftarrow). \\ assert(b \leftarrow a) \leftarrow not\ c. \end{array} \tag{37}$$

and that all the events are empty EVOLP programs. The (only) answer set of P is $M = \{a, assert(b \leftarrow a)\}$ and conveying the information that program P is ready to evolve into a new program (P, P_2) by adding rule $(b \leftarrow a)$ at the next step, i.e. to P_2. In the only dynamic stable model M_2 of the new program (P, P_2), atom b is true as well as atom $assert(not\ a \leftarrow)$ and also c, meaning that (P, P_2) evolves into a new program (P, P_2, P_3) by adding rule $(not\ a \leftarrow)$ at the next step, i.e. in P_3. This negative fact in P_3

conflicts with the fact in P, and the older is rejected. The rule added in P_2 remains valid, but is no longer useful to conclude b, since a is no longer valid. So, $assert(not\ a \leftarrow)$ and c are also no longer true. In the only dynamic stable model of the last sequence both a, b, and c are false.

This example does not address external events. The rules that belong to the i-th event should be added to the program of state i, and proceed as in the example above.

Example 3. In the example above, suppose that at state 2 there is an external event with the rules, r_1 and r_2, $assert(d \leftarrow b) \leftarrow a$ and $e \leftarrow$. Since the only stable model of P is $I = \{a, assert(b \leftarrow a)\}$ and there is an outside event at state 2 with r_1 and r_2, the program evolves into the new program obtained by updating P not only with the rule $b \leftarrow a$ but also with those rules, i.e. $(P, \{b \leftarrow a;\ assert(d \leftarrow b) \leftarrow a;\ e \leftarrow\})$. The only dynamic stable model M_2 of this program is $\{b, assert(not\ a \leftarrow), assert(d \leftarrow b), e\}$.

If we keep with the evolution of this program (e.g. by subsequent empty events), we have to decide what to do, in these subsequent states, about the event received at state 2. Intuitively, we want the rules coming from the outside, be they observations or assertion commands, to be understood as events given at a state, that are not to persist by inertia. I.e. if rule r belongs to some set E_i of an event sequence, this means that r was perceived, or received, after $i-1$ evolution steps of the program, and that this perception event is not to be assumed by inertia from then onward. In the example, it means that if we have perceived e at state 2, then e and all its possible consequences should be true at that state. But the truth of e should not persist into the subsequent state (unless e is yet again perceived from the outside). In other words, when constructing subsequent states, the rules coming from events in state 2 should no longer be available and considered. As will become clear below, making these events persistent can be specified in EVOLP.

Definition 5. *An* evolution interpretation *of length n of an evolving program P over $\mathcal{A}$ is a finite sequence $\mathcal{I} = (I_1, I_2, \ldots, I_n)$ of interpretations $\mathcal{A}_{assert}$. The* evolution trace *associated with evolution interpretation $\mathcal{I}$ is the sequence of programs $(P_1, P_2, \ldots, P_n)$ where:*

- $P_1 = P$;
- $P_i = \{r \mid assert(r) \in I_{i-1}\}$, *for each* $2 \leq i \leq n$.

Definition 6. *An evolution interpretation $(I_1, I_2, \ldots, I_n)$, of length n, with evolution trace $(P_1, P_2, \ldots, P_n)$ is an* evolution stable model *of an evolving program P given a sequence of events $(E_1, E_2, \ldots, E_k)$, with $n \leq k$, iff for every i ($1 \leq i \leq n$), I_i is a dynamic stable model at state i of $(P_1, P_2 \ldots, (P_i \cup E_i))$.*

Notice that the rules coming from the outside do not persist by inertia. At any given step i, the rules from E_i are added and the (possibly various) I_i obtained. This determines the programs P_{i+1} of the trace, which are then added to E_{i+1} to determine the models I_{i+1}. The definition assumes the whole sequence of events given a priori. In fact this need not be so because the events at any given step n only influence the models in the evolution interpretation from n onward:

Proposition 1. *Let $M = (M_1, \ldots, M_n)$ be an evolution stable model of P given a sequence of events $(E_1, E_2, \ldots, E_n)$. Then, for any sets of events $E_{n+1}, \ldots, E_m$ $(m > n)$, M is also an evolution stable model of P given $(E_1, \ldots, E_n, E_{n+1}, \ldots, E_m)$.*

EVOLP programs may have various evolution models of given length, or none:

Example 4. Consider P with the following two rules, and 3 empty events:

$$\begin{array}{l} assert\,(a \leftarrow) \leftarrow not\, assert\,(b \leftarrow), not\, b. \\ assert\,(b \leftarrow) \leftarrow not\, assert\,(a \leftarrow), not\, a. \end{array} \tag{38}$$

The reader can check that there are 2 evolution stable models of length 3, each representing one possible evolution of the program after those empty events:

$$\begin{array}{l} M_1 = \langle \{assert(a \leftarrow)\}, \{a, assert(a \leftarrow)\}, \{a, assert(a \leftarrow)\} \rangle \\ M_2 = \langle \{assert(b \leftarrow)\}, \{b, assert(b \leftarrow)\}, \{b, assert(b \leftarrow)\} \rangle \end{array}$$

Since various evolutions may exist for a given length, evolution stable models alone do not determine a truth relation. A truth relation can be defined, as usual, based on the intersection of models:

Definition 7. *Let P be an evolving program, $\mathcal{E}$ an event sequence of length n, both over the language $\mathcal{A}$, and M an interpretation over $\mathcal{A}_{assert}$. M is a* Stable Model of P given $\mathcal{E}$ *iff $(M_1, \ldots, M_{n-1}, M)$ is an evolution stable model of P given $\mathcal{E}$ with length n, for some interpretations $M_1, \ldots, M_{n-1}$. Let $SM\,(\langle P, \mathcal{E} \rangle)$ denote the set of* Stable Model of P given $\mathcal{E}$*.We say that propositional atom A of $\mathcal{A}$ is:* true given $\mathcal{E}$*, denoted by $\langle P, \mathcal{E} \rangle \models A$, iff A belongs to all stable models of P given $\mathcal{E}$;* false given $\mathcal{E}$*, denoted by $\langle P, \mathcal{E} \rangle \models not\, A$, iff A does not belong to any stable models of P given $\mathcal{E}$;* unknown given $\mathcal{E}$ *otherwise.*

A consequence of the above definitions is that the semantics of EVOLP is, in fact, a proper generalization of the answer-set semantics, in the following sense:

Proposition 2. *Let P be a generalized (extended) logic program (without predicate $assert/1$) over a language $\mathcal{A}$, and $\mathcal{E}$ be any sequence with $n \geq 0$ of empty EVOLP programs. Then, M is a stable model of P given $\mathcal{E}$ iff the restriction of M to $\mathcal{A}$ is an answer set of P (in the sense of [15,22]).*

The possibility of having various stable models after an event sequence is of special interest for using EVOLP as a language for reasoning about possible evolutions of an agent's knowledge base. Like for answer-set programs, we define the notion of categorical programs as those such that, for any given event sequence, no "branching" occurs, i.e. a single stable model exists.

Definition 8. *An EVOLP program P is* categorical *given event sequence $\mathcal{E}$ iff there exists only one stable model of P given $\mathcal{E}$.*