EVOLP: Tranformation-Based Semantics*

Martin Slota[1,2] and João Leite[2]

[1] Katedra aplikovanej informatiky, Univerzita Komenského, Slovakia
[2] CENTRIA, Universidade Nova de Lisboa, Portugal

Abstract. Over the years, Logic Programming has proved to be a good and natural tool for expressing, querying and manipulating explicit knowledge in many areas of computer science. However, it is not so easy to use in dynamic environments. Evolving Logic Programs (EVOLP) are an elegant and powerful extension of Logic Programming suitable for Multi-Agent Systems, planning and other uses where information tends to change dynamically. In this paper we characterize EVOLP by transforming it into an equivalent normal logic program over an extended language, that serves as a basis of an existing implementation. Then we prove that the proposed transformation is sound and complete and examine its computational complexity.

1 Introduction

Construction of intelligent agents is one of the main matters of artificial intelligence. Computational Logic has shown to be a good tool for both symbolic knowledge representation and reasoning, with fruitful application in Multi-Agent Systems.

Examples of the success of Computational Logic in Multi-Agent Systems include IMPACT [1,2], 3APL [3,4], Jason [5], DALI [6], ProSOCS [7], FLUX [8] and ConGolog [9], to name a few. For a survey on some of these systems, as well as others, see [10,11,12].

Computational Logic, and Logic Programming in particular, can be seen as a good representation language for static knowledge. However, agents must be capable of operating independently in a partially observable environment that may change unexpectedly. Therefore, they need to be able to evolve, both due to self-updates and updates from the environment, and change their model of the world accordingly. If we are to move to such more open and dynamic environments, 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

---

* This research has been funded by the European Commission within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`). It was also supported by the Slovak Agency for Promotion Research and Development under the contract No. APVV-20-P04805.

F. Sadri and K. Satoh (Eds.): CLIMA VIII, LNAI 5056, pp. 117–136, 2008.

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 the above mentioned systems.

Much research in the last decade has been devoted to finding a good way of updating knowledge bases represented by logic programs [13,14,15,16,17,18,19]. A sequence of logic programs where each program represents a supervenient state of the world was called a Dynamic Logic Program (DLP). Finding a suitable semantics for DLPs became the first step on one of the paths to using Logic Programming in Multi-Agent Systems. Quite a number of semantics with different properties were introduced [16,17,18,19]. We will only mention the Dynamic Stable Model semantics [17] that was later improved and called Refined Dynamic Stable Models [19]. This is also the semantics used throughout this work. For a more comprehensive overview of semantics for DLPs see [18,20,21].

Although Dynamic Logic Programming provides a semantics for a sequence of states of the world expressed as logic programs, it doesn't offer a mechanism for constructing these programs. Update languages like LUPS [22], EPI [23], KUL and KABUL [18] were developed for the purpose of specifying transitions between the states of the world. Each of them defines special types of rules for adding and deleting rules from programs in the sequence. Evolving Logic Programs (EVOLP) [24] also comes from this line of work, but while its predecessors were becoming more and more complicated as more constructs were being added, EVOLP is a simple, yet very powerful extension of traditional logic programming.

EVOLP generalizes Answer-set Programming [25] to allow for the specification of a program's own evolution, in a single unified way. Furthermore, EVOLP also permits, besides internal or self updates, for updates arising from the environment. The resulting language provides a simpler, and more general, formulation of logic program updating, running close to traditional LP doctrine, setting itself on a firm formal basis in which to express, implement, and reason about dynamic knowledge bases, opening up several interesting research topics.

Indeed, 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. Syntactically, evolving logic programs are just generalized logic programs[1]. 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.

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

---

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

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).

The aim of this work is to provide the basis for an operational semantics for EVOLP, based on a sound and complete transformational semantics for EVOLP, i.e. define a transformation that, given an evolving logic program and a sequence of events, produces an equivalent normal logic program over an extended language. Such a transformation, together with an ASP solver, is the basis of our implementation of EVOLP under the evolution stable model semantics[2]. Currently, the only somehow similar implementation appears in [26] and only for a limited constructive view of EVOLP. More information about the implementation can be found in [27].

We also examine the complexity of the defined transformation. This is performed by inferring both a lower and an upper bound for the size of the transformed program.

The remainder of this work is structured as follows: in Sect. 2 we introduce the syntax and semantics of EVOLP; in Sect. 3 we define the transformation; in Sect. 4 we show that the proposed transformation is sound and complete; in Sect. 5 we examine the complexity of the transformation; in Sect. 6 we conclude and sketch some possible directions of future work.

2 Background: Concepts and Notation

We start with the usual preliminaries: Let $\mathcal{L}$ be a set of propositional atoms. A *default literal* is an atom preceded by **not**. A *literal* is either an atom or a default literal. A *rule* r is an ordered pair $(H(r), B(r))$ 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 $B(r) = \{ L_1, L_2, \ldots, L_n \}$ will simply be written as

$$L_0 \leftarrow L_1, L_2, \ldots, L_n. \tag{1}$$

If $H(r) = A$ (resp. $H(r) = \mathbf{not}\, A$) then $\mathbf{not}\, H(r) = \mathbf{not}\, A$ (resp. $\mathbf{not}\, H(r) = A$). Two rules r, r' are *conflicting*, denoted by $r \bowtie r'$, iff $H(r) = \mathbf{not}\, H(r')$. We will say a literal L appears in a rule (1) iff the set $\{ L, \mathbf{not}\, L \} \cap \{ L_0, L_1, L_2, \ldots, L_n \}$ is non-empty.

A *generalized logic program* (GLP) over $\mathcal{L}$ is a set of rules. A literal appears in a GLP iff it appears in at least one of its rules.

An *interpretation* of $\mathcal{L}$ is any set of atoms $I \subseteq \mathcal{L}$. An atom A is true in I, denoted by $I \models A$, iff $A \in I$, and false otherwise. A default literal $\mathbf{not}\, A$ is true

---

[2] A web based demo is available at `http://centria.di.fct.unl.pt/evolp/`

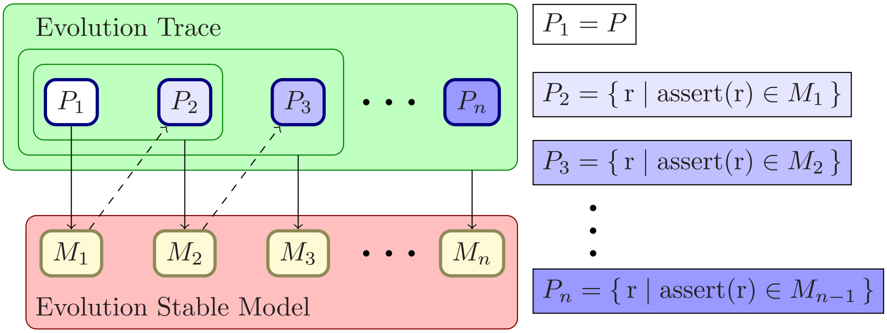

Fig. 1. Semantics of EVOLP (without events)

in I, denoted by $I \models \mathbf{not}\, A$, iff $A \notin I$, and false otherwise. A set of literals B is true in I iff each literal in B is true in I. Given an interpretation I we also define $I^- \overset{\text{def}}{=} \{ \mathbf{not}\, A \mid A \in \mathcal{L} \setminus I \}$ and $I^* \overset{\text{def}}{=} I \cup I^-$. An interpretation M is a *stable model* of a GLP P iff $M^* = \mathrm{least}(P \cup M^-)$ where $\mathrm{least}(\cdot)$ denotes the least model of the definite program obtained from the argument program by treating all default literals as new atoms.

Definition 1. *A* dynamic logic program *(DLP) is a sequence of GLPs. Let $\mathcal{P} = (P_1, P_2, \ldots, P_n)$ be a DLP. We use $\rho(\mathcal{P})$ to denote the multiset of all rules appearing in the programs $P_1, P_2, \ldots, P_n$ and $\mathcal{P}^i$ $(1 \leq i \leq n)$ to denote the i-th component of $\mathcal{P}$, i.e. P_i. Given a DLP $\mathcal{P}$ and an interpretation I we define*

$$\mathrm{Def}(\mathcal{P}, I) \overset{\text{def}}{=} \{ \mathbf{not}\, A \mid (\nexists r \in \rho(\mathcal{P}))(H(r) = A \wedge I \models B(r)) \} \quad , \tag{2}$$

$$\mathrm{Rej}^j(\mathcal{P}, I) \overset{\text{def}}{=} \left\{ r \in \mathcal{P}^j \;\middle|\; (\exists k, r') \left(k \geq j \wedge r' \in \mathcal{P}^k \wedge r \bowtie r' \wedge I \models B(r')\right) \right\} \quad , \tag{3}$$

$$\mathrm{Rej}(\mathcal{P}, I) \overset{\text{def}}{=} \bigcup_{i=1}^{n} \mathrm{Rej}^i(\mathcal{P}, I) \quad . \tag{4}$$

An interpretation M is a (refined) dynamic stable model *of a DLP $\mathcal{P}$ iff $M^* = \mathrm{least}([\rho(\mathcal{P}) \setminus \mathrm{Rej}(\mathcal{P}, M)] \cup \mathrm{Def}(\mathcal{P}, M))$.*

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

1. *All propositional atoms in $\mathcal{L}$ are propositional atoms in $\mathcal{L}_{\mathrm{assert}}$.*
2. *If r is a rule over $\mathcal{L}_{\mathrm{assert}}$ then $\mathrm{assert}(r)$ is a propositional atom in $\mathcal{L}_{\mathrm{assert}}$.*
3. *Nothing else is a propositional atom in $\mathcal{L}_{\mathrm{assert}}$.*

An evolving logic program *over a language $\mathcal{L}$ is a GLP over $\mathcal{L}_{\mathrm{assert}}$. An* event sequence *over $\mathcal{L}$ is a sequence of evolving logic programs over $\mathcal{L}$.*

Table 1. Evolution of the program in Example 1 ("assert" is shortened to "ass")

Time	Program	Event	Model
1	P	E_1	$\{\text{no\_coffee}, \text{write\_thesis}, \text{ass}(\text{tired} \leftarrow)\}$
2	$\{\, \text{tired} \leftarrow .\}$	E_2	$\{\text{tired}, \text{no\_coffee}, \text{make\_coffee}\}$
3	$\emptyset$	E_3	$\{\text{tired}, \text{drink\_coffee}, \text{ass}(\textbf{not}\, \text{tired} \leftarrow)\}$
4	$\{\, \textbf{not}\, \text{tired} \leftarrow .\}$	E_4	$\{\text{write\_thesis}, \text{ass}(\text{tired} \leftarrow),$ $\text{ass}(\textbf{not}\, \text{drink\_coffee} \leftarrow),$ $\text{ass}(\text{sleep} \leftarrow \text{tired}),$ $\text{ass}(\text{ass}(\textbf{not}\, \text{tired} \leftarrow) \leftarrow \text{sleep})\}$
5	$\{\text{tired} \leftarrow .,$ $\textbf{not}\, \text{drink\_coffee} \leftarrow .,$ $\text{sleep} \leftarrow \text{tired}.,$ $\text{ass}(\textbf{not}\, \text{tired} \leftarrow) \leftarrow \text{sleep}.\}$	E_5	$\{\text{tired}, \text{sleep}, \text{ass}(\textbf{not}\, \text{tired} \leftarrow)\}$

Definition 3. *An* evolution interpretation *of length* n *of an evolving program* P *over* $\mathcal{L}$ *is a finite sequence* $\mathcal{I} = (I_1, I_2, \ldots, I_n)$ *of interpretations of* $\mathcal{L}_{\text{assert}}$*. The* evolution trace *associated with an evolution interpretation* $\mathcal{I}$ *of* P *is the sequence of programs* $(P_1, P_2, \ldots, P_n)$ *where* $P_1 = P$ *and* $P_{i+1} = \{\, r \mid \text{assert}(r) \in I_i \,\}$ *for all* $i \in \{\, 1, 2, \ldots, n-1 \,\}$*.*

Definition 4. *An evolution interpretation* $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ *of an evolving logic program* P *with evolution trace* $(P_1, P_2, \ldots, P_n)$ *is an* evolution stable model *of* P *given an event sequence* $(E_1, E_2, \ldots, E_n)$ *iff for every* $i \in \{\, 1, 2, \ldots, n \,\}$ M_i *is a dynamic stable model of* $(P_1, P_2, \ldots, P_{i-1}, P_i \cup E_i)$*.*

Example 1. Consider the following evolving logic program:

$$P: \quad \text{write\_thesis} \leftarrow \textbf{not}\, \text{tired}. \tag{5}$$

$$\text{drink\_coffee} \leftarrow \text{tired}, \textbf{not}\, \text{no\_coffee}. \tag{6}$$

$$\text{make\_coffee} \leftarrow \text{tired}, \text{no\_coffee}. \tag{7}$$

$$\text{assert}(\text{tired} \leftarrow) \leftarrow \text{write\_thesis}. \tag{8}$$

$$\text{assert}(\textbf{not}\, \text{tired} \leftarrow) \leftarrow \text{drink\_coffee}. \tag{9}$$

P could be an initial program of a simple agent (e.g. Mary) who is trying to write a thesis. Mary can do 3 things: write the thesis, drink coffee or make coffee. She also relies on a sensor that sends her the fact ($\text{no\_coffee} \leftarrow .$) as an event in case no coffee is available. The meaning of the rules is as follows: Rule (5) says Mary's writing the thesis as long as she's not tired. Rules (6) and (7) tell her what to do when she's tired. Rules (8) and (9) specify whether she will be tired in the next evolution step. If she's writing the thesis, she will get tired. Drinking coffee has an opposite effect. If she's making coffee, no change will take place. Table 1 shows the evolution of P given the sequence of events $\mathcal{E} = (E_1, E_2, E_3, E_4, E_5)$ where $E_1 = E_2 = \{\, \text{no\_coffee} \leftarrow .\}$, $E_3 = E_5 = \emptyset$ and

$$\begin{aligned} E_4 : \quad & \text{assert}(\textbf{not}\,\text{drink\_coffee} \leftarrow) \leftarrow . \\ & \text{assert}(\text{sleep} \leftarrow \text{tired}) \leftarrow . \\ & \text{assert}(\text{assert}(\textbf{not}\,\text{tired} \leftarrow) \leftarrow \text{sleep}) \leftarrow . \end{aligned}$$

We start off with P and E_1 and compute the first model. It says there is no coffee, Mary is writing her thesis and in the next step she will get tired. We infer the second program from the model, add the second event and compute the second model. Now Mary is tired and makes coffee. This makes the sensor stop complaining in the third step (i.e. $E_3 = \emptyset$) and Mary, still tired, drinks coffee. In the fourth step Mary is writing her thesis again and she is reprogrammed – when she gets tired she will take a nap instead of drinking coffee. In the fifth step the new rules are used – Mary is tired and sleeping.

The previous example is very simple and its main purpose is to demonstrate how the definitions of semantics of EVOLP work. It is by no means exhaustive and doesn't demonstrate the full power of the language. In the next example we will briefly show more complex rules that are a part of a more complex example where EVOLP is used as to implement a fairly sophisticated email agent. For the full version of the example the reader is referred to [28]. Another different example that makes use of an agent architecture based on EVOLP can be found in [26].

Example 2. The email agent example is composed of an evolving logic program P and a sequence of events $(E_1, E_2, \ldots, E_16)$. We will only pick some specific rules from the example to demonstrate the constructs that can be specified using EVOLP.

First let's consider the following two rules:

$$\text{assert}(\text{in}(M, F_{to}) \leftarrow) \leftarrow \text{move}(M, F_{from}, F_{to}), \text{in}(M, F_{from}).$$
$$\text{assert}(\textbf{not}\,\text{in}(M, F_{from}) \leftarrow) \leftarrow \text{move}(M, F_{from}, F_{to}), \textbf{not}\,\text{in}(M, F_{to}).$$

They are used in [28] as a part of the base program of an email agent and encode a message moving mechanism. The first rule specifies that if a command received to move a message M from folder F_{from} to folder F_{to} and it is currently in folder F_{from} (i.e. the command is a valid one), then in the next evolution step the message will be in folder F_{to}. The second rule deletes M from folder F_{from} in case it is different from the destination folder F_{to}. Similar rules are used to remember new messages and delete them from folders and remember sent messages.

Another three rules encode an evolving predicate that decides whether a message is spam or not:

$$\begin{aligned} r_1 : \quad & \text{spam}(F, S, B) \leftarrow \text{contains}(S, \text{``credit''}). \\ r_2 : \quad \textbf{not}\, & \text{spam}(F, S, B) \leftarrow \text{contains}(F, \text{``accountant''}). \\ r_3 : \quad & \text{spam}(F, S, B) \leftarrow \text{contains}(S, \text{``credit''}), \text{contains}(S, \text{``Fwd''}). \end{aligned}$$

They are asserted one by one in the events, i.e. $\mathrm{assert}(r_1) \in E_i$, $\mathrm{assert}(r_2) \in E_j$ and $\mathrm{assert}(r_3) \in E_k$ for some $i < j < k$. The first rule defines a spam message as any message having "credit" in its subject. This is further updated by the second rule – messages whose sender contains the word "accountant" are not considered as spam (even if they contain "credit" in their body). The third rule further updates the way messages are classified – messages containing both "credit" and "Fwd" in their subject are considered spam (even if they come from the accountant).

The last rule we are going to mention encodes a more complex behaviour:

$$\mathrm{assert}(\mathrm{send}(R, S, B) \leftarrow \mathrm{newmsg}(M, F, S, B), \mathrm{contains}(S, ID), \mathrm{assign}(ID, R)) \leftarrow \\ \mathrm{newmsg}(M, R, ID, B), \mathrm{contains}(B, \text{“accept”}).$$

The meaning of the rule is as follows: If a message is received from a reviewer R that contains a paper identification ID in the subject and the word "accept" in its body, then all future messages regarding this paper will be forwarded to the reviewer R in case he has been assigned the paper ID. Multiple rules of this kind can be used to configure a simple paper submission system that keeps track of papers, deadlines, authors and reviewers and manages the communication between them.

3 Transformation into a Normal Logic Program

Now we will define a transformation which turns an evolving logic program P together with an event sequence $\mathcal{E}$ of length n into a normal logic program $P_{\mathcal{E}}$ over an extended language. We will prove later that the stable models of $P_{\mathcal{E}}$ are in one-to-one correspondence with the evolution stable models of P given $\mathcal{E}$.

The transformation is essentially a multiple parallel usage of a similar transformation for DLPs introduced in [29]. First we need to define the extended language over which we will construct the resulting program:

$$\begin{aligned} \mathcal{L}_{\mathrm{trans}} \overset{\mathrm{def}}{=} & \left\{ A^j, A^j_{\mathrm{neg}} \mid A \in \mathcal{L}_{\mathrm{assert}} \wedge 1 \leq j \leq n \right\} \\ & \cup \left\{ \mathrm{rej}(A^j, i), \mathrm{rej}(A^j_{\mathrm{neg}}, i) \mid A \in \mathcal{L}_{\mathrm{assert}} \wedge 1 \leq j \leq n \wedge 0 \leq i \leq j \right\} \\ & \cup \left\{ u \right\} \ . \end{aligned}$$

Atoms of the form A^j and A^j_{neg} in the extended language allow us to compress the whole evolution interpretation (consisting of n interpretations of $\mathcal{L}_{\mathrm{assert}}$, see Def. 3) into just one interpretation of $\mathcal{L}_{\mathrm{trans}}$. Atoms of the form $\mathrm{rej}(A^j, i)$ and $\mathrm{rej}(A^j_{\mathrm{neg}}, i)$ are needed for rule rejection simulation. The atom u will serve to formulate constraints needed to eliminate some unwanted models of $P_{\mathcal{E}}$.

To simplify the notation in the transformation's definition, we'll use the following conventions: Let L be a literal over $\mathcal{L}_{\mathrm{assert}}$, *Body* a set of literals over $\mathcal{L}_{\mathrm{assert}}$ and j a natural number. Then:

- If L is an atom A, then L^j is A^j and L^j_{neg} is A^j_{neg}.
- If L is a default literal $\mathbf{not}\, A$, then L^j is A^j_{neg} and L^j_{neg} is A^j.
- $\mathit{Body}^j = \{\, L^j \mid L \in \mathit{Body} \,\}$.

Definition 5. *Let P be an evolving logic program and $\mathcal{E} = (E_1, E_2, \ldots, E_n)$ an event sequence. By a* transformational equivalent *of P given $\mathcal{E}$ we mean the normal logic program $P_{\mathcal{E}} = P^1_{\mathcal{E}} \cup P^2_{\mathcal{E}} \cup \ldots \cup P^n_{\mathcal{E}}$ over $\mathcal{L}_{\text{trans}}$, where each $P^j_{\mathcal{E}}$ consists of these six groups of rules:*

1. ***Rewritten program rules.*** *For every rule $(L \leftarrow \mathit{Body}.) \in P$, $P^j_{\mathcal{E}}$ contains the rule*
$$L^j \leftarrow \mathit{Body}^j, \mathbf{not}\, \mathrm{rej}(L^j, 1).$$
2. ***Rewritten event rules.*** *For every rule $(L \leftarrow \mathit{Body}.) \in E_j$, $P^j_{\mathcal{E}}$ contains the rule*
$$L^j \leftarrow \mathit{Body}^j, \mathbf{not}\, \mathrm{rej}(L^j, j).$$
3. ***Assertable rules.*** *For every rule $r = (L \leftarrow \mathit{Body}.)$ over $\mathcal{L}_{\text{assert}}$ and all i, $1 < i \leq j$, such that $(\mathrm{assert}(r))^{i-1}$ is in the head of some rule of $P^{i-1}_{\mathcal{E}}$, $P^j_{\mathcal{E}}$ contains the rule*
$$L^j \leftarrow \mathit{Body}^j, (\mathrm{assert}(r))^{i-1}, \mathbf{not}\, \mathrm{rej}(L^j, i).$$
4. ***Default assumptions.*** *For every atom $A \in \mathcal{L}_{\text{assert}}$ such that A^j or A^j_{neg} appears in some rule of $P^j_{\mathcal{E}}$ (from the previous groups of rules), $P^j_{\mathcal{E}}$ also contains the rule*
$$A^j_{\text{neg}} \leftarrow \mathbf{not}\, \mathrm{rej}(A^j_{\text{neg}}, 0).$$
5. ***Rejection rules.*** *For every rule of $P^j_{\mathcal{E}}$ of the form*
$$L^j \leftarrow \mathit{Body}, \mathbf{not}\, \mathrm{rej}(L^j, i).^3$$
$P^j_{\mathcal{E}}$ also contains the rules
$$\mathrm{rej}(L^j_{\text{neg}}, p) \leftarrow \mathit{Body}. \tag{10}$$
$$\mathrm{rej}(L^j, q) \leftarrow \mathrm{rej}(L^j, i). \tag{11}$$
where:
   - *(a) $p \leq i$ is the largest index such that $P^j_{\mathcal{E}}$ contains a rule with the literal $\mathbf{not}\, \mathrm{rej}(L^j_{\text{neg}}, p)$ in its body. If no such p exists, then (10) is not in $P^j_{\mathcal{E}}$.*
   - *(b) $q < i$ is the largest index such that $P^j_{\mathcal{E}}$ contains a rule with the literal $\mathbf{not}\, \mathrm{rej}(L^j, q)$ in its body. If no such q exists, then (11) is not in $P^j_{\mathcal{E}}$.*

---

[3] It can be a rewritten program rule, a rewritten event rule or an assertable rule (default assumptions never satisfy the further conditions). The set *Body* contains all literals from the rule's body except the $\mathbf{not}\, \mathrm{rej}(L^j, i)$ literal.

6. ***Totality constraints.*** *For all* $i \in \{1, 2, \dots, j\}$ *and every atom* $A \in \mathcal{L}_{\mathrm{assert}}$ *such that* $P^j_{\mathcal{E}}$ *contains rules of the form*

$$A^j \leftarrow Body_p, \mathbf{not}\, \mathrm{rej}(A^j, i).$$
$$A^j_{\mathrm{neg}} \leftarrow Body_n, \mathbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, i).$$

$P^j_{\mathcal{E}}$ *also contains the constraint*

$$u \leftarrow \mathbf{not}\, u, \mathbf{not}\, A^j, \mathbf{not}\, A^j_{\mathrm{neg}}.$$

Each $P^j_{\mathcal{E}}$ contains rules for simulating the DLP $(P, P_2, P_3, \dots, P_{j-1}, P_j \cup E_j)$ from the definition of evolution stable model (Definition 4). For the simulation we use the transformational semantics from [29]. We also rewrite all atoms from the original rules as a new set of j-indexed atoms.

The first two groups of rules in $P^j_{\mathcal{E}}$ (rewritten program rules and rewritten event rules) contain the rewritten forms of rules from P and E_j. However, we don't know the exact contents of $P_2, P_3, \dots, P_j$, so the group of assertable rules contains all rules that can possibly occur in those programs. Each of these rules also has an atom of the form $(\mathrm{assert}(r))^{i-1}$ in its body. We will call it the *assertion guard* of the rule and it assures the rule is only used in case it was actually asserted. These atoms are also the only connection between the rules of $P^j_{\mathcal{E}}$ and the rules in $P^1_{\mathcal{E}} \cup P^2_{\mathcal{E}} \cup \dots \cup P^{j-1}_{\mathcal{E}}$.

The default assumptions are defined similarly as in [29], and they have the same function – they simulate the set of defaults defined in Def. 1.

Rewritten program rules, rewritten event rules, assertable rules and default assumptions also contain a default literal of the form $\mathbf{not}\, \mathrm{rej}(L^j, i)$ in their bodies. We will call this literal the *rejection guard* of the rule and the natural number i the *level* of the rule. Together with the rejection rules, the rejection guard provides a means of rejecting a rule by a higher level rule, similarly as in the set of rejected rules (4).

Rejection rules are responsible for inferring the correct $\mathrm{rej}(L^j, i)$ atoms. The first kind of rules introduces the rejection of rules with a conflicting literal in their head and a level that is the maximum that is also less or equal to i. The second kind of rules takes care of propagating the rejection to rules with an even lower level.

Totality constraints are important in the case that rules of the same level reject each other and no rule of higher level resolves their conflict. An interpretation causing such a situation is not a refined dynamic stable model (more details can be found in [19]). Totality constraints are needed to eliminate the superfluous stable models of $P_{\mathcal{E}}$ originating from these situations.

The following example illustrates how the transformation works:

Example 3. Let's take the evolving logic program

$$P: \qquad \mathrm{assert}(\mathrm{a} \leftarrow) \leftarrow \mathbf{not}\, \mathrm{a}.$$
$$\mathrm{assert}(\mathbf{not}\, \mathrm{a} \leftarrow) \leftarrow \mathrm{a}.$$

and a sequence of two empty events $\mathcal{E}$. The defined transformation would produce the following transformed program:

$$\begin{aligned}
P_{\mathcal{E}}: \quad (\mathrm{assert}(\mathrm{a} \leftarrow))^1 &\leftarrow \mathrm{a}^1_{\mathrm{neg}}, \mathbf{not}\, \mathrm{rej}((\mathrm{assert}(\mathrm{a} \leftarrow))^1, 1). && (12)\\
(\mathrm{assert}(\mathbf{not}\, \mathrm{a} \leftarrow))^1 &\leftarrow \mathrm{a}^1, \mathbf{not}\, \mathrm{rej}((\mathrm{assert}(\mathbf{not}\, \mathrm{a} \leftarrow))^1, 1). && (13)\\
\mathrm{a}^1_{\mathrm{neg}} &\leftarrow \mathbf{not}\, \mathrm{rej}(\mathrm{a}^1_{\mathrm{neg}}, 0). && (14)\\
(\mathrm{assert}(\mathrm{a} \leftarrow))^2 &\leftarrow \mathrm{a}^2_{\mathrm{neg}}, \mathbf{not}\, \mathrm{rej}((\mathrm{assert}(\mathrm{a} \leftarrow))^2, 1). && (15)\\
(\mathrm{assert}(\mathbf{not}\, \mathrm{a} \leftarrow))^2 &\leftarrow \mathrm{a}^2, \mathbf{not}\, \mathrm{rej}((\mathrm{assert}(\mathbf{not}\, \mathrm{a} \leftarrow))^2, 1). && (16)\\
\mathrm{a}^2 &\leftarrow (\mathrm{assert}(\mathrm{a} \leftarrow))^1, \mathbf{not}\, \mathrm{rej}(\mathrm{a}^2, 2). && (17)\\
\mathrm{a}^2_{\mathrm{neg}} &\leftarrow (\mathrm{assert}(\mathbf{not}\, \mathrm{a} \leftarrow))^1, \mathbf{not}\, \mathrm{rej}(\mathrm{a}^2_{\mathrm{neg}}, 2). && (18)\\
\mathrm{a}^2_{\mathrm{neg}} &\leftarrow \mathbf{not}\, \mathrm{rej}(\mathrm{a}^2_{\mathrm{neg}}, 0). && (19)\\
\mathrm{rej}(\mathrm{a}^2_{\mathrm{neg}}, 2) &\leftarrow (\mathrm{assert}(\mathrm{a} \leftarrow))^1. && (20)\\
\mathrm{rej}(\mathrm{a}^2, 2) &\leftarrow (\mathrm{assert}(\mathbf{not}\, \mathrm{a} \leftarrow))^1. && (21)\\
\mathrm{rej}(\mathrm{a}^2_{\mathrm{neg}}, 0) &\leftarrow \mathrm{rej}(\mathrm{a}^2_{\mathrm{neg}}, 2). && (22)\\
\mathrm{u} &\leftarrow \mathbf{not}\, \mathrm{u}, \mathbf{not}\, \mathrm{a}^2, \mathbf{not}\, \mathrm{a}^2_{\mathrm{neg}}. && (23)
\end{aligned}$$

The rules (12) to (14) simulate the first evolution step – they are 2 rewritten program rules and one default assumption. Rules (15) and (16) are rewritten program rules for the second evolution step. In this step, two new rules can be asserted – (17) and (18) are the corresponding assertable rules. (19) is a default assumption, (20) to (22) are rejection rules and (23) is a totality constraint.

$P_{\mathcal{E}}$ has exactly one stable model

$$M = \{\, \mathrm{a}^1_{\mathrm{neg}}, (\mathrm{assert}(\mathrm{a} \leftarrow))^1, \mathrm{a}^2, (\mathrm{assert}(\mathbf{not}\, \mathrm{a} \leftarrow))^2, \mathrm{rej}(\mathrm{a}^2_{\mathrm{neg}}, 2), \mathrm{rej}(\mathrm{a}^2_{\mathrm{neg}}, 0) \,\} \enspace .$$

It directly corresponds to the single evolution stable model $\mathcal{M} = (M_1, M_2)$ of P given $\mathcal{E}$ where $M_1 = \{\, \mathrm{assert}(\mathrm{a} \leftarrow) \,\}$ and $M_2 = \{\, \mathrm{a}, \mathrm{assert}(\mathbf{not}\, \mathrm{a} \leftarrow) \,\}$.

4 Soundness and Completeness

The following 2 theorems show how the stable models of the transformed program correspond to the evolution stable models of the input program. Only sketches of proofs are provided, their full versions can be found in [30].

Theorem 1 (Soundness). *Let P be an evolving logic program, $\mathcal{E} = (E_1, E_2, \ldots, E_n)$ an event sequence, N a stable model of $P_{\mathcal{E}}$,*

$$M_i = \{\, A \in \mathcal{L}_{\mathrm{assert}} \mid A^i \in N \,\} \text{ for all } i \in \{\, 1, 2, \ldots, n \,\} \enspace .$$

Then $(M_1, M_2, \ldots, M_n)$ is an evolution stable model of P given $\mathcal{E}$.

Proof (sketch). Let $(P_1, P_2, \ldots, P_n)$ be the evolution trace associated with the evolution interpretation $\mathcal{M} = (M_1, M_2, \ldots, M_n)$. According to Def. 4, $\mathcal{M}$ is an evolution stable model of P given $\mathcal{E}$ iff for every $i \in \{\, 1, 2, \ldots, n \,\}$ M_i is a dynamic stable model of $(P_1, P_2, \ldots, P_{i-1}, P_i \cup E_i)$. Hence we choose one arbitrary but fixed $j \in \{ 1, 2, \ldots, n \}$ and show that M_j is a dynamic stable model of $\mathcal{P} = (P_1, P_2, \ldots, P_{j-1}, P_j \cup E_j)$.

M_j contains exactly those atoms that have their corresponding j-indexed counterpart inferred by rules in $P^j_{\mathcal{E}}$ as defined in Def. 5. What we need to show is that each rule of $P^j_{\mathcal{E}}$ either corresponds to some rule in $P_1, P_2, \ldots, P_j, E_j$, or is used to simulate the rule-rejection mechanism behind Dynamic Logic Programming, or has no effect on the model.

It can be seen quite easily that rewritten program rules and rewritten event rules correspond to rules in $P_1 = P$ and E_j, respectively. They just contain one extra literal in their body – the rejection guard that is used to block them in case they are rejected.

An assertable rule, added as a rewritten form of an original rule r, can only be fired in case an atom of the form $(\mathrm{assert}(r))^{i-1}$ is true in N. But then $\mathrm{assert}(r)$ is true in M_{i-1} and thus $r \in P_i$. On the other hand, if $r \in P_i$ for some $i \in \{ 2, 3, \ldots, j \}$, then $\mathrm{assert}(r) \in M_{i-1}$ and hence $(\mathrm{assert}(r))^{i-1} \in N$. So each rewritten program rule, rewritten event rule and assertable rule either corresponds to some rule in the dynamic logic program $\mathcal{P}$, or has no effect on the resulting model because it cannot be fired.

Default assumptions in $P^j_{\mathcal{E}}$ are present for all atoms of the program. They simulate the set of defaults from Def. 1 and contain, just like all the other rules before, the rejection guard in their body that can block their usage in case a higher level rule rejects them by having an opposite literal in its head and its body satisfied in N.

The rejection rules together with the totality constraints can be proved to behave as follows:

1. For each atom A^j appearing in $P^j_{\mathcal{E}}$ they force exactly one of A^j and A^j_{neg} to be a member of N.
2. They infer an atom $\mathrm{rej}(L^j, i)$ with $i > 0$ iff some rule $r \in \mathrm{Rej}^i(\mathcal{P}, M_j)$ has L in its head.
3. They infer an atom $\mathrm{rej}(L^j, 0)$ iff L is a default literal $\mathbf{not}\, A$ and $\mathbf{not}\, A \notin \mathrm{Def}(\mathcal{P}, I)$.

The first point implies that the resulting model will be consistent with respect to the j-indexed versions of original literals. Correct simulation of the rule-rejection mechanism is a consequence of the second point. The third point ensures that only the appropriate subset of default assumptions is used.

Using the propositions from the previous paragraphs, it can be proved (by induction on the number of applications of the immediate consequence operator) that M_j is indeed a dynamic stable model of $\mathcal{P}$. □

Theorem 2 (Completeness). *Let P be an evolving logic program, $\mathcal{E} = (E_1, E_2, \ldots, E_n)$ an event sequence, $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ an evolution stable model of P given $\mathcal{E}$, $(P_1, P_2, \ldots, P_n)$ the evolution trace associated with $\mathcal{M}$ and*

$$\mathcal{P}_i = (P_1, P_2, \ldots, P_{i-1}, P_i \cup E_i) \textit{ for all } i \in \{ 1, 2, \ldots, n \} .$$

Furthermore, let

$$\begin{aligned} N = &\{\, L^i \mid i \in \{\, 1, 2, \ldots, n \,\} \wedge M_i \models L \wedge L^i \textit{ appears in } P_{\mathcal{E}} \,\} \\ &\cup \{\, \mathrm{rej}(L^i, k) \mid 1 \leq k \leq i \leq n \wedge (\exists r \in \mathrm{Rej}^k(\mathcal{P}_i, M_i))(H(r) = L) \,\} \\ &\cup \{\, \mathrm{rej}(A^i_{\mathrm{neg}}, 0) \mid i \in \{\, 1, 2, \ldots, n \,\} \wedge \mathbf{not}\, A \notin \mathrm{Def}(\mathcal{P}_i, M_i) \,\} \enspace . \end{aligned}$$

Then N is a stable model of $P_{\mathcal{E}}$.

Proof (sketch). Let $R = \mathrm{least}(P_{\mathcal{E}} \cup N^-)$. We need to prove that $N^* = R$. This can be proved in three steps:

1. In the first step we must prove for every literal L of $\mathcal{L}_{\mathrm{assert}}$ and all $j \in \{\, 1, 2, \ldots, n \,\}$ that $L_j \in N \iff L_j \in R$. This can be proved by complete induction on j, using ideas very similar to those in the proof of soundness.
2. The second step is to prove that N and R are identical on the set of atoms of the form $\mathrm{rej}(L^j, i)$ for all $L \in \mathcal{L}_{\mathrm{assert}}$, every $j \in \{\, 1, 2, \ldots, n \,\}$ and every $i \in \{\, 0, 1, \ldots, j \,\}$. If $\mathrm{rej}(L^j, i) \in N$, then some rule $r \in \mathrm{Rej}^i(\mathcal{P}_j, M_j)$ has L in its head. This rule must have been rejected by some other rule r'. $P^j_{\mathcal{E}}$ must contain a rule corresponding to r' that will cause the presence of appropriate rejection rules. Consequently, $\mathrm{rej}(L^j, i)$ will eventually be added to R. A similar idea can be used to prove the converse implication.
3. The last matter that needs to be proved is that none of the totality constraints in $P_{\mathcal{E}}$ has been broken, i.e. that $u \notin R$. This can be proved by contradiction: consider one of the constraints if broken. Then for some atom $A \in \mathcal{L}_{\mathrm{assert}}$ we have $\mathbf{not}\, A^j, \mathbf{not}\, A^j_{\mathrm{neg}} \in R$ and also that both A^j and A^j_{neg} appear in $P_{\mathcal{E}}$. Furthermore, $\mathbf{not}\, A^j, \mathbf{not}\, A^j_{\mathrm{neg}} \in N^-$ and hence $A^j, A^j_{\mathrm{neg}} \notin N$. But then we have both $M_j \not\models A$ and $M_j \not\models \mathbf{not}\, A$ – a contradiction. □

5 Complexity of the Transformation

The computational complexity of the proposed transformation is interesting from multiple viewpoints:

- it directly influences the computational complexity of the implementation of EVOLP that is based on it [27],
- it allows to identify the most time-consuming parts of the transformation which can in turn be optimized to perform better,
- it reveals the branching factor that EVOLP is capable of, i.e. it demonstrates the expressivity of EVOLP.

The rules for generating the transformed program are quite simple, so the algorithm performing the transformation will also be reasonably simple. What really matters is the size and number of rules of the transformed program. The bigger the transformed program will be, the longer it will take to generate it and perform any further processing. We are also interested in which group of rules is the biggest and how it can be made smaller.

The size of each generated rule is either constant (default assumptions, totality constraints and propagating rejection rules) or just constantly bigger than the corresponding original rule. Therefore, we will concentrate on the number of generated rules. First we will derive both a lower and an upper bound for the number of rules of the transformed program. After we have the bounds, we will draw some conclusions. For the rest of this section we will assume P is a finite evolving logic program and $\mathcal{E} = (E_1, E_2, \ldots, E_n)$ is a sequence of finite events.

5.1 Lower Bound

We know the transformed program $P_{\mathcal{E}}$ contains $n|P|$ rewritten program rules and $\sum_{j=1}^{n} |E_j|$ rewritten event rules. So a very simple lower bound for $|P_{\mathcal{E}}|$ is:

$$|P_{\mathcal{E}}| \geq n|P| + \sum_{j=1}^{n} |E_j| \enspace . \tag{24}$$

Equality can be achieved only if $P = E_1 = E_2 = \ldots = E_n = \emptyset$. Otherwise, $P_{\mathcal{E}}$ will also contain some default assumptions and rejection rules.

5.2 Number of Assertable Rules

In order to derive an upper bound for $|P_{\mathcal{E}}|$, we will first need to make an approximation of the number of assertable rules. Let A be the set of all assertable rules in $P_{\mathcal{E}}$. In Appendix A it is shown that

$$|A| \leq |P| \frac{n^3 - n}{6} + \sum_{j=1}^{n} |E_j| \frac{(n-j)^3 + 5(n-j)}{6} \enspace . \tag{25}$$

It is also shown that in case we disallow nested asserts (i.e. a rule within an assert atom must not contain another assert atom in its head), we have

$$|A| \leq |P| \frac{n^2 - n}{2} + \sum_{j=1}^{n} (n-j)|E_j| \enspace . \tag{26}$$

5.3 Upper Bound

We already know the number of rewritten program rules and rewritten event rules in the transformed program and an upper bound for the number of assertable rules. Now we need to deal with the default assumptions, rejection rules and totality constraints.

How many default assumptions can there be? Both P and the events are finite so only a finite set of atoms from $\mathcal{L}_{\text{assert}}$ can be used in them. Let this set be $\mathcal{L}_{P,\mathcal{E}}$. Each atom in this set can generate up to n default assumptions.

Each rewritten program rule, rewritten event rule and assertable rule can generate at most 2 rejection rules. Two of these rules are needed to generate a totality constraint.

Taken together, we have

$$|P_{\mathcal{E}}| \leq \frac{7}{2}\left(n|P| + \sum_{j=1}^{n}|E_j| + |A|\right) + n|\mathcal{L}_{P,\mathcal{E}}| \ . \tag{27}$$

If we use the approximation of $|A|$ (25), we get the following inequality:

$$|P_{\mathcal{E}}| \leq \frac{7}{2}\left(n|P| + \sum_{j=1}^{n}|E_j| \right.$$
$$\left. + |P|\frac{n^3-n}{6} + \sum_{j=1}^{n}|E_j|\frac{(n-j)^3+5(n-j)}{6}\right) + n|\mathcal{L}_{P,\mathcal{E}}|$$

which can be further simplified to

$$|P_{\mathcal{E}}| \leq \frac{7}{2}\left(|P|\frac{n^3+5n}{6} + \sum_{j=1}^{n}|E_j|\left(\frac{(n-j)^3+5(n-j)}{6}+1\right)\right) + n|\mathcal{L}_{P,\mathcal{E}}| \ .$$

When n is large and program sizes are considered as parameters, we can use the big-oh notation to get

$$|P_{\mathcal{E}}| = |P| \cdot \mathcal{O}\big(n^3\big) + \sum_{j=1}^{n}|E_j| \cdot \mathcal{O}\big((n-j)^3\big) + n|\mathcal{L}_{P,\mathcal{E}}| \ . \tag{28}$$

In case of programs without nested asserts we can use (26) to derive

$$|P_{\mathcal{E}}| \leq \frac{7}{2}\left(|P|\frac{n^2+n}{2} + \sum_{j=1}^{n}(n-j+1)|E_j|\right) + n|\mathcal{L}_{P,\mathcal{E}}| \ ,$$

or, for large n,

$$|P_{\mathcal{E}}| = |P| \cdot \mathcal{O}\big(n^2\big) + \sum_{j=1}^{n}|E_j| \cdot \mathcal{O}(n-j) + n|\mathcal{L}_{P,\mathcal{E}}| \ . \tag{29}$$

5.4 Conclusion

In this section we examined how big the transformed program can get. Probably the most obvious and also a very important observation is that the lower bound (24) for $|P_{\mathcal{E}}|$ implies that the transformed program grows with n, no matter how big the events are. Hence for large values of n and small events this can lead to an intractably large transformed program even for tractably large inputs.

The main reason for this is that the transformed program captures all the possible evolutions of the input program. The expressivity EVOLP encompasses, especially the possibility of arbitrary branching based on intermediate models, makes it intractable to compute all the possible evolutions of even small programs. In case we are not interested in all or many of the possible evolutions, this transformation is not suitable as a basis for an implementation. For example when using EVOLP as an executable specification of a Multi-Agent System, a constructive view of the language as taken in [26] is more appropriate.

There are, however, also other possible uses of EVOLP where we do care about the possible evolutions. An example is reasoning about possible futures or on-line planning as a part of a deliberation of an agent. Verification of a Multi-Agent System specified in EVOLP is another. We believe the transformation is appropriate for such applications of EVOLP because the high computational complexity inherent in these problems is delegated to an answer set solver which is already optimized to deal with it.

From the upper bound (28) for $|P_{\mathcal{E}}|$ we can also see that the size of the transformed program depends on the size of the input program, size of events and n at most polynomially. Furthermore, if we use only (or mostly) rules without nested asserts, (29) implies we can lower the power of n that $|P_{\mathcal{E}}|$ grows with.

6 Conclusion and Future Work

We have defined a transformational semantics for Evolving Logic Programs and proved that it is sound and complete. We also examined the computational complexity of the transformation and identified situations in which it is practically applicable. These include reasoning about possible futures, on-line planning and verification of systems specified in EVOLP.

Future work can be devoted to optimizations of the transformation. In particular, the current transformation generates a number of unnecessary default assumptions and rejection rules. This was useful because it made its definition and proofs of soundness and completeness simpler. Now that these proofs are ready, we can concentrate on optimizing the transformation and prove more easily what optimizations are safe to perform. In many situations it is also possible to share rules among evolution steps which is another source of possible future optimizations. The third issue worth examining is the possibility of having a larger transformed program that performs better with the current answer set solvers.

Another line of work that can be followed involves generalizing the transformation. The definition can be extended to a language with classical negation. Another interesting issue is that of identifying a class of evolving logic programs with variables that is groundable with intuitive results and is general enough to be usable in practise.

References

1. Subrahmanian, V.S., Bonatti, P., Dix, J., Eiter, T., Kraus, S., Ozcan, F., Ross, R.: Heterogeneous Agent Systems. MIT Press, Cambridge (2000)
2. Dix, J., Zhang, Y.: IMPACT: a multi-agent framework with declarative semantics. In: Bordini, et al. (eds.) [11], ch. 3

3. Hindriks, K., de Boer, F., van der Hoek, W., Ch. Meyer, J.-J.: Agent programming in 3APL. Int. J. of Autonomous Agents and Multi-Agent Systems 2(4), 357–401 (1999)
4. Dastani, M., van Riemsdijk, M.B., Ch.Meyer, J.-J.: Programming multi-agent systems in 3APL. In: Bordini, et al. (eds.) [11], ch. 2
5. Bordini, R., Hübner, J., Vieira, R.: Jason and the Golden Fleece of agent-oriented programming. In: Bordini, et al. (eds.) [11], ch. 1
6. Costantini, S., Tocchio, A.: A logic programming language for multi-agent systems. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 1–13. Springer, Heidelberg (2002)
7. Bracciali, A., Demetriou, N., Endriss, U., Kakas, A., Lu, W., Stathis, K.: Crafting the mind of a PROSOCS agent. Applied Artificial Intelligence 20(4-5) (2006)
8. Thielscher, M.: Reasoning Robots: The Art and Science of Programming Robotic Agents. Springer, Heidelberg (2005)
9. De Giacomo, G., Lesprance, Y., Levesque, H.J.: ConGolog, a concurrent programming language based on the situation calculus. Artificial Intelligence 121(1-2), 109–169 (2000)
10. Mascardi, V., Martelli, M., Sterling, L.: Logic-based specification languages for intelligent software agents. Theory and Practice of Logic Programming 4(4), 429–494 (2004)
11. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Platforms and Applications. Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15. Springer, Heidelberg (2005)
12. Bordini, R.H., Braubach, L., Dastani, M., Seghrouchni, A.E.F., Gomez-Sanz, J.J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. Informatica 30(1), 33–44 (2006)
13. Sakama, C., Inoue, K.: Updating extended logic programs through abduction. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) LPNMR 1999. LNCS (LNAI), vol. 1730. Springer, Heidelberg (1999)
14. Zhang, Y., Foo, N.Y.: Updating logic programs. In: Procs. of ECAI 1998. John Wiley & Sons, Chichester (1998)
15. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: On properties of update sequences based on causal rejection. Theory and Practice of Logic Programming 2(6) (2002)
16. Leite, J.A., Pereira, L.M.: Generalizing updates: From models to programs. In: Dix, J., Moniz Pereira, L., Przymusinski, T.C. (eds.) LPKR 1997. LNCS (LNAI), vol. 1471. Springer, Heidelberg (1998)
17. Alferes, J.J., Leite, J.A., Pereira, L.M., Przymusinska, H., Przymusinski, T.C.: Dynamic updates of non-monotonic knowledge bases. The Journal of Logic Programming 45(1-3), 43–70 (2000)
18. Leite, J.A.: Evolving Knowledge Bases. IOS Press, Amsterdam (2003)
19. 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)
20. Leite, J.A.: On some differences between semantics of logic program updates. In: Lemaître, C., Reyes, C.A., González, J.A. (eds.) IBERAMIA 2004. LNCS (LNAI), vol. 3315, pp. 375–385. Springer, Heidelberg (2004)
21. Homola, M.: Various semantics are equal on acyclic programs. In: Leite, J.A., Torroni, P. (eds.) Procs. of the 5th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA V). LNCS, vol. 3487, pp. 78–95. Springer, Heidelberg (2004)
22. Alferes, J.J., Pereira, L.M., Przymusinska, H., Przymusinski, T.C.: LUPS – a language for updating logic programs. Artificial Intelligence 138(1&2) (June 2002)

23. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: A framework for declarative update specifications in logic programs. In: IJCAI 2001, pp. 649–654. Morgan-Kaufmann, San Francisco (2001)
24. 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)
25. 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)
26. Leite, J.A., Soares, L.: Adding evolving abilities to a multi-agent system. In: Inoue, K., Satoh, K., Toni, F. (eds.) CLIMA 2006. LNCS (LNAI), vol. 4371, pp. 246–265. Springer, Heidelberg (2007)
27. Slota, M., Leite, J.A.: EVOLP: an implementation. In: Sadri, F., Satoh, K. (eds.) Proceedings of the 8th Workshop on Computational Logic in Multi-Agent Systems (CLIMA VIII) (2008); (System Description, in this volume)
28. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Logic programming for evolving agents. In: Klusch, M., Omicini, A., Ossowski, S., Laamanen, H. (eds.) CIA 2003. LNCS (LNAI), vol. 2782, pp. 281–297. Springer, Heidelberg (2003)
29. Banti, F., Alferes, J.J., Brogi, A.: Operational semantics for DyLPs. In: Bento, C., Cardoso, A., Dias, G. (eds.) EPIA 2005. LNCS (LNAI), vol. 3808, pp. 43–54. Springer, Heidelberg (2005)
30. Slota, M.: Transformational semantics and implementation of evolving logic programs. Master's thesis, Univerzita Komenského (May 2007), `http://slotik.info/`

A Upper Bound for the Number of Assertable Rules

In this Appendix we derive an upper bound for the number of assertable rules in the transformed program. We will assume P is a finite evolving logic program and $\mathcal{E} = (E_1, E_2, \ldots, E_n)$ is a sequence of finite events. Let A be the set of all assertable rules in the transformational equivalent $P_{\mathcal{E}}$ of P given $\mathcal{E}$. We will need some more declarative characterization of the rules in A in order to work with its cardinality. The following Definition, Lemmas and Theorem provide such characterization:

Definition 6. *Let $E_0 = \emptyset$. We define*

$$A_1 \overset{\text{def}}{=} \{\, r \mid (\exists r_1 \in P)(H(r_1) = \text{assert}(r)) \,\} \quad , \tag{30}$$

for all $i \in \{\, 2, 3, \ldots, n-1 \,\}$

$$\begin{aligned} A_i \overset{\text{def}}{=} & \{\, r \mid (\exists r_1 \in A_{i-1})(H(r_1) = \text{assert}(r)) \,\} \\ & \cup \{\, r \mid (\exists r_2 \in E_{i-1})(H(r_2) = \text{assert}(r_1) \wedge H(r_1) = \text{assert}(r)) \,\} \end{aligned} \tag{31}$$

and for all $j \in \{\, 1, 2, \ldots, n-1 \,\}$ also

$$\overline{A_j} \overset{\text{def}}{=} \bigcup_{i=1}^{j} A_i \cup \{\, r \mid (\exists r_1 \in E_j)(H(r_1) = \text{assert}(r)) \,\} \quad . \tag{32}$$

Remark 1. Let $j \in \{1, 2, \dots, n\}$. Each assertable rule in $P^j_{\mathcal{E}}$ is fully determined by its assertion guard, i.e. if we know that it has the assertion guard $(\mathrm{assert}(r))^{i-1}$ and $r = (L \leftarrow Body.)$, then the assertable rule must be:

$$L^j \leftarrow Body^j, (\mathrm{assert}(r))^{i-1}, \mathbf{not}\, \mathrm{rej}(L^j, i).$$

We will make use of this fact in order to make some formulations simpler.

Lemma 1. *Let* $i \in \{1, 2, \dots, n-1\}$, $j \in \{i, i+1, \dots, n-1\}$ *and* $r \in A_i$. *Then* $P^{j+1}_{\mathcal{E}}$ *contains an assertable rule with the assertion guard* $(\mathrm{assert}(r))^j$.

Proof. We will prove by induction on i.

1 Let $r \in A_1$. Then some rule $r_1 \in P$ exists such that $H(r_1) = \mathrm{assert}(r)$. Let $j \in \{1, 2, \dots, n-1\}$. Then $P^j_{\mathcal{E}}$ must contain a rewritten program rule with $(\mathrm{assert}(r))^j$ in its head and therefore $P^{j+1}_{\mathcal{E}}$ must contain an assertable rule with the assertion guard $(\mathrm{assert}(r))^j$.

2 We assume the claim holds for i and prove it for $i+1$. Let $r \in A_{i+1}$ and let $j \in \{i+1, i+2, \dots, n-1\}$. Two cases are possible:
  - (a) Some rule $r_1 \in A_i$ exists such that $H(r_1) = \mathrm{assert}(r)$. By the induction hypothesis we have that $P^j_{\mathcal{E}}$ contains an assertable rule with the assertion guard $(\mathrm{assert}(r_1))^{j-1}$. This rule has $(\mathrm{assert}(r))^j$ in its head. Hence $P^{j+1}_{\mathcal{E}}$ contains an assertable rule with the assertion guard $(\mathrm{assert}(r))^j$.
  - (b) Some rule $r_2 \in E_i$ exists such that $H(r_2) = \mathrm{assert}(r_1)$ and $H(r_1) = \mathrm{assert}(r)$. Then $P^i_{\mathcal{E}}$ contains a rewritten event rule with $(\mathrm{assert}(r_1))^i$ in its head. Hence $P^j_{\mathcal{E}}$ will contain an assertable rule with the assertion guard $(\mathrm{assert}(r_1))^i$ and $(\mathrm{assert}(r))^j$ in its head. Therefore $P^{j+1}_{\mathcal{E}}$ must contain an assertable rule with the assertion guard $(\mathrm{assert}(r))^j$. □

Lemma 2. *Let* $j \in \{1, 2, \dots, n-1\}$ *and* $r \in \overline{A_j}$. *Then* $P^j_{\mathcal{E}}$ *contains a rule with* $(\mathrm{assert}(r))^j$ *in its head.*

Proof. Assume that $r \in \overline{A_j}$. Two cases are possible:

a) $r \in A_i$ for some $i \in \{1, 2, \dots, j\}$. Then by Lemma 1 we have that $P^{j+1}_{\mathcal{E}}$ contains an assertable rule with the assertion guard $(\mathrm{assert}(r))^j$. Hence $P^j_{\mathcal{E}}$ must contain a rule with $(\mathrm{assert}(r))^j$ in its head.

b) Some rule $r_1 \in E_j$ exists such that $H(r_1) = \mathrm{assert}(r)$. Then $P^j_{\mathcal{E}}$ contains a rewritten event rule with $(\mathrm{assert}(r))^j$ in its head. □

Lemma 3. *Let* $j \in \{1, 2, \dots, n-1\}$ *and let* r *be a rule over* $\mathcal{L}_{\mathrm{assert}}$. *If* $P^j_{\mathcal{E}}$ *contains a rule with* $(\mathrm{assert}(r))^j$ *in its head, then* $r \in \overline{A_j}$.

Proof. We will prove by complete induction on j.

1 The basis can be inferred from the inductive step with $j = 1$ (the third case doesn't have to be examined because $P^1_{\mathcal{E}}$ contains no assertable rules).

2 We assume the proposition holds for all $i \in \{1, 2, \dots, j-1\}$ and prove it for j. Let's consider three cases:

(a) If $P_{\mathcal{E}}^j$ contains a rewritten program rule r_1^* with $(\text{assert}(r))^j$ in its head, then P contains a rule r_1 such that $H(r_1) = \text{assert}(r)$. Hence $r \in A_1 \subseteq \overline{A_j}$.

(b) If $P_{\mathcal{E}}^j$ contains a rewritten event rule r_1^* with $(\text{assert}(r))^j$ in its head, then E_j contains a rule r_1 such that $H(r_1) = \text{assert}(r)$. Hence $r \in \overline{A_j}$.

(c) If $P_{\mathcal{E}}^j$ contains an assertable rule with $(\text{assert}(r))^j$ in its head, then it must be of the form

$$(\text{assert}(r))^j \leftarrow \mathit{Body}^j, (\text{assert}(r_1))^{i-1}, \mathbf{not}\, \text{rej}((\text{assert}(r))^j, i).$$

where $r_1 = (\text{assert}(r) \leftarrow \mathit{Body}.)$ and $i \le j$. So $P_{\mathcal{E}}^{i-1}$ must contain a rule with $(\text{assert}(r_1))^{i-1}$ in its head and by the induction hypothesis we have $r_1 \in \overline{A_{i-1}}$. Two cases are possible again:

i. $r_1 \in A_h$ for some $h \in \{1, 2, \ldots, i-1\}$. Then $r \in A_{h+1} \subseteq \overline{A_j}$.

ii. Some rule $r_2 \in E_{i-1}$ exists such that $H(r_2) = \text{assert}(r_1)$. We also have $H(r_1) = \text{assert}(r)$. So $r \in A_i \subseteq \overline{A_j}$. □

Theorem 3. *Let* $j \in \{1, 2, \ldots, n-1\}$ *and let* r *be a rule over* $\mathcal{L}_{\text{assert}}$. $P_{\mathcal{E}}^j$ *contains a rule with* $(\text{assert}(r))^j$ *in its head iff* $r \in \overline{A_j}$.

Proof. Follows directly from Lemmas 2 and 3. □

As a consequence of the theorem we have

$$|A| = \sum_{j=1}^{n} (n-j) \left|\overline{A_j}\right| \tag{33}$$

because each rule $r \in \overline{A_j}$ will generate $n-j$ assertable rules, one in each of $P_{\mathcal{E}}^{j+1}, P_{\mathcal{E}}^{j+2}, \ldots, P_{\mathcal{E}}^{n}$. Now we can make an approximation of $|A|$. According to (30), (31) and (32) we have for all $j \in \{1, 2, \ldots, n-1\}$

$$|A_j| \le |P| + \sum_{i=1}^{j-1} |E_i| \ , \qquad \left|\overline{A_j}\right| \le j|P| + |E_j| + \sum_{i=1}^{j} (j-i)|E_i| \ .$$

Furthermore, by (33) we have

$$\begin{aligned} |A| = \sum_{j=1}^{n} (n-j) \left|\overline{A_j}\right| &\le \sum_{j=1}^{n} (n-j) \left(j|P| + |E_j| + \sum_{i=1}^{j} (j-i)|E_i| \right) \\ &= |P| \sum_{j=1}^{n} j(n-j) + \sum_{j=1}^{n} (n-j)|E_j| + \sum_{j=1}^{n} (n-j) \sum_{i=1}^{j} (j-i)|E_i| \ . \end{aligned} \tag{34}$$

First let's solve the first sum:

$$\sum_{j=1}^{n} j(n-j) = n \sum_{j=1}^{n} j - \sum_{j=1}^{n} j^2 = \frac{n^3 - n}{6} \ . \tag{35}$$

The third sum can be simplified as follows:

$$\begin{aligned}\sum_{j=1}^{n}(n-j)\sum_{i=1}^{j}(j-i)|E_i| &= \sum_{i=1}^{n}|E_i|\sum_{j=1}^{n-i}j((n-i)-j)\\ &= \sum_{i=1}^{n}|E_i|\frac{(n-i)^3-(n-i)}{6} \enspace .\end{aligned} \tag{36}$$

By (34), (35) and (36) we now have

$$|A| \leq |P|\frac{n^3-n}{6} + \sum_{j=1}^{n}|E_j|\frac{(n-j)^3+5(n-j)}{6} \enspace .$$

We can also put some extra restrictions on the input program and then look at the number of assertable rules. For example, if we disallow nested asserts (i.e. a rule within an assert atom must not contain an assert atom in its head), then we have $|A_1| \leq |P|$ and $|A_j| = 0$ for all $j \in \{ 2, 3, \dots, n-1 \}$. Hence $\left|\overline{A_j}\right| \leq |P| + |E_j|$ for all $j \in \{ 1, 2, \dots, n-1 \}$ and

$$|A| \leq \sum_{j=1}^{n}(n-j)(|P|+|E_j|) = |P|\frac{n^2-n}{2} + \sum_{j=1}^{n}(n-j)|E_j| \enspace .$$