Operational Semantics for EVOLP*

Martin Slota[1] 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 written in an extended language, that serves as the 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, and Logic Programming in particular, have 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].

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.

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

* 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.

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.

Syntactically, evolving logic programs are just generalized logic programs[3]. But semantically, they permit to reason about assertions of new rules to the program. The language of Evolving Logic Programs 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 assertions within assertions allows rule updates to be themselves updated down the line. The ability to include assert literals in rule bodies allows for looking ahead on some program changes and acting on that knowledge before the changes occur. 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 written in 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, available through two frontends, a web form[4] and a command line interface[5]. Currently, the only somehow similar implementation appears in [25] and only for a limited constructive view of EVOLP.

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;

---

[3] logic programs that allow for rules with default negated literals in their heads.

[4] runs at `http://www.ii.fmph.uniba.sk/~slota/evolp-prop-prototype/`

[5] downloadable from `http://slotik.medovnicek.sk/2006/thesis/results/`

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 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 $I^- \stackrel{\text{def}}{=} \{\mathbf{not}\, A \mid A \in \mathcal{L} \setminus I\}$ and $I^* \stackrel{\text{def}}{=} I \cup I^-$. An interpretation M is a *stable model* of a GLP P iff $M^* = \text{least}(P \cup M^-)$ where $\text{least}(\cdot)$ 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*

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

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

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

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

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

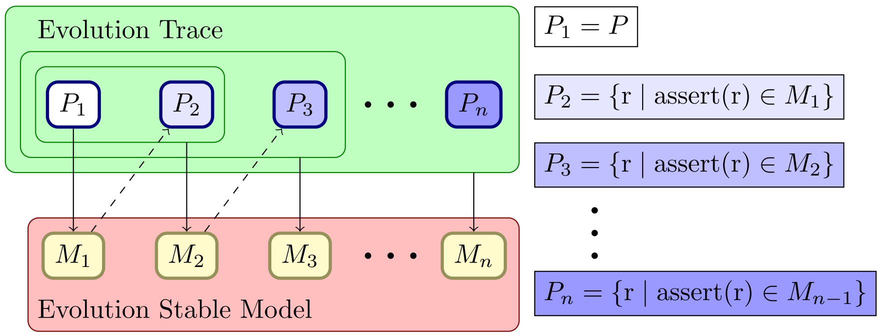

Fig. 1. Semantics of EVOLP (without events)

All propositional atoms in $\mathcal{L}$ *are propositional atoms in* $\mathcal{L}_{\text{assert}}$*; – If* r *is a rule over* $\mathcal{L}_{\text{assert}}$ *then* $\text{assert}(r)$ *is a propositional atom in* $\mathcal{L}_{\text{assert}}$*; – Nothing else is a propositional atom in* $\mathcal{L}_{\text{assert}}$*. An* evolving logic program *over a language* $\mathcal{L}$ *is a GLP over* $\mathcal{L}_{\text{assert}}$*. An* event sequence *over* $\mathcal{L}$ *is a sequence of evolving logic programs over* $\mathcal{L}$*.*

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 (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

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

Time	Program	Event	Model
1	P	E_1	{no_coffee, write_thesis, ass(tired ←)}
2	{tired ← .}	E_2	{tired, no_coffee, make_coffee}
3	$\emptyset$	E_3	{tired, drink_coffee, ass(**not** tired ←)}
4	{**not** tired ← .}	E_4	{write_thesis, ass(tired ←), ass(**not** drink_coffee ←), ass(sleep ← tired), ass(ass(**not** tired ←) ← sleep)}
5	{tired ← ., **not** drink_coffee ← ., sleep ← tired., ass(**not** tired ←) ← sleep.}	E_5	{tired, sleep, ass(**not** tired ←)}

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

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

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.

For more examples see [26].

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 [27]. First we need to define the extended language over which we will construct the resulting program:

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

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 $\mathcal{L}_{\text{assert}}$, see Def. 3) into just one interpretation of $\mathcal{L}_{\text{trans}}$. Atoms of the form $\text{rej}(A^j, i)$ and $\text{rej}(A^j_{\text{neg}}, i)$ 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 $\mathcal{L}_{\text{assert}}$, *body* a set of literals over $\mathcal{L}_{\text{assert}}$ 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 $\textbf{not}\, A$, 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 $(L \leftarrow \mathit{body}.) \in P$ it contains the rule*
$$L^j \leftarrow \mathit{body}^j, \textbf{not}\, \text{rej}(L^j, 1).$$
2. ***Rewritten event rules.*** *For every rule $(L \leftarrow \mathit{body}.) \in E_j$ it contains the rule*
$$L^j \leftarrow \mathit{body}^j, \textbf{not}\, \text{rej}(L^j, j).$$
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 $(\text{assert}(r))^{i-1}$ is in the head of some rule of $P^{i-1}_{\mathcal{E}}$ it contains the rule*
$$L^j \leftarrow \mathit{body}^j, (\text{assert}(r))^{i-1}, \textbf{not}\, \text{rej}(L^j, i).$$
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) it also contains the rule*
$$A^j_{\text{neg}} \leftarrow \textbf{not}\, \text{rej}(A^j_{\text{neg}}, 0).$$
5. ***Rejection rules.*** *For every rule of $P^j_{\mathcal{E}}$ of the form*
$$L^j \leftarrow \mathit{body}, \textbf{not}\, \text{rej}(L^j, i).^6$$

---

[6] 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 $\textbf{not}\, \text{rej}(L^j, i)$ literal.

it also contains the rules

$$\mathrm{rej}(L^j_{\mathrm{neg}}, p) \leftarrow \mathit{body}. \tag{10}$$

$$\mathrm{rej}(L^j, q) \leftarrow \mathrm{rej}(L^j, i). \tag{11}$$

where:

(a) $p \leq i$ *is the largest index such that* $P^j_{\mathcal{E}}$ *contains a rule with the literal* $\mathbf{not}\,\mathrm{rej}(L^j_{\mathrm{neg}}, p)$ *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* $P^j_{\mathcal{E}}$*.*

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

$$A^j \leftarrow \mathit{body}_p, \mathbf{not}\,\mathrm{rej}(A^j, i).$$

$$A^j_{\mathrm{neg}} \leftarrow \mathit{body}_n, \mathbf{not}\,\mathrm{rej}(A^j_{\mathrm{neg}}, i).$$

it 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 $(P, P_2, P_3, \ldots, P_{j-1}, P_j \cup E_j)$ from the definition of evolution stable model (Definition 4). For the simulation we use the transformational semantics from [27]. 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 $P_2, P_3, \ldots, P_j$, 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. 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 $P^1_{\mathcal{E}} \cup P^2_{\mathcal{E}} \cup \ldots \cup P^{j-1}_{\mathcal{E}}$.

The default assumptions are defined similarly as in [27], 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. Together with the rejection rules, this literal 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 the next less or equally preferred rule with a conflicting literal in its head. The second kind of rules takes care of propagating the rejection to even less preferred rules with the same head.

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

The following example illustrates how the transformation works:

Example 2. Let's take the evolving logic program

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

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

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

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 [28].

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 has no effect on the model, or is used to simulate the rule-rejection mechanism behind Dynamic Logic Programming.

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 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 $(\text{assert}(r))^{i-1}$ is true in N. But then $\text{assert}(r)$ 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 $\text{assert}(r) \in M_{i-1}$ and hence $(\text{assert}(r))^{i-1} \in N$. 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, a literal 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 $\text{rej}(L^j, i)$ with $i > 0$ iff some rule $r \in \text{Rej}^i(\mathcal{P}, M_j)$ has L in its head.
3. They infer an atom $\text{rej}(L^j, 0)$ iff L is $\mathbf{not}\, A$ for some atom $A \in \mathcal{L}_{\text{assert}}$ and $\mathbf{not}\, A \notin \text{Def}(\mathcal{P}, I)$.

The first point implies that the resulting model will be consistent w.r.t. the rewritten 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*

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

Furthermore, let

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

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 similar ideas as 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 [29],
- 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:

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

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

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

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

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

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:

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

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

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

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,

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

5.4 Conclusion

The lower bound (24) for $|P_{\mathcal{E}}|$ implies that the transformed program grows with n, no matter how big the events are. For large values of n and many empty events this can be a problem. The main reason for this is that the expressivity EVOLP encompasses and the possibility of arbitrary branching based on the intermediate models makes it difficult to share rules among evolution steps. However, sharing is possible in many situations and it could significantly reduce the number of rules. The current transformation also 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. But now that these proofs are ready, it should be easy to prove the extra rules can be removed.

One possibility of using the current transformation efficiently is to disallow large values of n, i.e. use it for bounded lookahead of up to n steps forward and then choose only one of the possible evolutions. A special case of this approach with lookahead of length 1 was also used in the implementation used in [25].

The good news regarding the transformation is that, according to (28), the size of the transformed program depends on the size of the input program, size of events and n only polynomially. So the transformation can be performed in polynomial time and for small values of n the transformed program will be of reasonable size (comparing to the size of input). Furthermore, if we use only (or mostly) rules without nested asserts, (29) implies that 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.

Future work can be devoted to optimizations and extensions of the current transformation. In particular, the current transformation can generate a number of default assumptions and rejection rules that are actually not needed. In many situations we can also share the rules among evolution steps which can result in a much smaller transformed program. The definition can also be extended to a language with classical negation.

References

1. V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press, 2000.
2. J. Dix and Y. Zhang. IMPACT: a multi-agent framework with declarative semantics. In Bordini et al. [11], chapter 3.
3. K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Int. J. of Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
4. M. Dastani, M. B. van Riemsdijk, and J.-J. Ch. Meyer. Programming multi-agent systems in 3APL. In Bordini et al. [11], chapter 2.
5. R. Bordini, J. Hübner, and R. Vieira. Jason and the Golden Fleece of agent-oriented programming. In Bordini et al. [11], chapter 1.

6. S. Costantini and A. Tocchio. A logic programming language for multi-agent systems. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*, pages 1–13. Springer, 2002.
7. A. Bracciali, N. Demetriou, U. Endriss, A. Kakas, W. Lu, and K. Stathis. Crafting the mind of a PROSOCS agent. *Applied Artificial Intelligence*, 20(4-5), 2006.
8. M. Thielscher. *Reasoning Robots: The Art and Science of Programming Robotic Agents*. Springer, 2005.
9. G. De Giacomo, Y. Lesprance, and H.J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.
10. V. Mascardi, M. Martelli, and L. Sterling. Logic-based specification languages for intelligent software agents. *Theory and Practice of Logic Programming*, 4(4):429–494, 2004.
11. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, 2005.
12. R. H. Bordini, L. Braubach, M. Dastani, A. El F. Seghrouchni, J. J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1):33–44, 2006.
13. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In *Procs. of LPNMR'99*, volume 1730 of *LNAI*. Springer, 1999.
14. Y. Zhang and N. Y. Foo. Updating logic programs. In *Procs. of ECAI'98*. John Wiley & Sons, 1998.
15. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2(6), 2002.
16. J. A. Leite and L. M. Pereira. Generalizing updates: From models to programs. In *Procs. of LPKR'97*, volume 1471 of *LNAI*. Springer, 1997.
17. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1-3):43–70, September/October 2000.
18. J. A. Leite. *Evolving Knowledge Bases*. IOS Press, 2003.
19. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1):7–32, 2005.
20. J. A. Leite. On some differences between semantics of logic program updates. In C. Lemaître, C. A. Reyes, and J. A. González, editors, *Procs. of 9th Ibero-American Conference on AI, (IBERAMIA'04)*, volume 3315 of *LNCS*, pages 375–385. Springer, 2004.
21. M. Homola. Various semantics are equal on acyclic programs. In J. A. Leite and P. Torroni, editors, *Procs. of the 5th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA V)*, volume 3487 of *LNCS*, pages 78–95. Springer, 2004.
22. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. LUPS – a language for updating logic programs. *Artificial Intelligence*, 138(1&2), June 2002.
23. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. In *IJCAI'01*, pages 649–654. Morgan-Kaufmann, 2001.
24. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*, pages 50–61. Springer, 2002.

25. J. A. Leite and L. Soares. Adding evolving abilities to a multi-agent system. In K. Satoh K. Inoue and F. Toni, editors, *Procs. of the 7th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VII)*, volume 4371 of *LNAI*, pages 246–265. Springer-Verlag, 2007.
26. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Logic programming for evolving agents. In M. Klusch, S. Ossowski, A. Omicini, and H. Laamanen, editors, *Proceedings of the 7th International Workshop on Cooperative Information Agents CIA'03*, volume 2782 of *LNCS*, pages 281–297. Springer, 2003.
27. F. Banti, J. J. Alferes, and A. Brogi. Operational semantics for DyLPs. In C. Bento A. Cardoso and G. Dias, editors, *Progress in Artificial Intelligence, Procs. 12th Portuguese Int. Conf. on Artificial Intelligence (EPIA'05)*, pages 43–54. Springer, 2005.
28. M. Slota. Transformational semantics and implementation of evolving logic programs. Master's thesis, Univerzita Komenského, May 2007. Available at http://slotik.medovnicek.sk/2006/thesis/.
29. M. Slota and J. A. Leite. EVOLP – transformation-based implementation. In F. Sadri and K. Satoh, editors, *Pre-Proceedings of the 8th Workshop on Computational Logic in Multi-Agent Systems*, 2007. System Description.

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 and Theorem provide such characterization:

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

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

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

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

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

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

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

The proof of the theorem is not included because of limited space, but it can be found in [28]. 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 $j \in \{1, 2, \ldots, n-1\}$

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

Furthermore, by (33) we have

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

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:

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

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

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

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 $j \in \{2, 3, \ldots, n-1\}$. Hence $\left|\overline{A_j}\right| \leq |P| + |E_j|$ for all $j \in \{1, 2, \ldots, n-1\}$ and

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