EVOLP: An Implementation*

Martin Slota\textsuperscript{1,2} and João Leite\textsuperscript{2}

\textsuperscript{1} Katedra aplikovanej informatiky, Univerzita Komenského, Slovakia
\textsuperscript{2} CENTRIA, Universidade Nova de Lisboa, Portugal

Abstract. In this paper we present an implementation of EVOLP under the Evolution Stable Model semantics, based on the transformation defined in [1]. We also discuss optimizations used in the implementation.

1 Introduction

Evolving Logic Programming (EVOLP) \cite{2} is a generalization of Answer Set Programming \cite{3} to allow for the specification of a program’s own evolution, in a single unified way, by permitting rules to indicate assertive conclusions in the form of program rules. Furthermore, EVOLP also permits, besides internal or self updates, for updates arising from the environment. The resulting language provides a simple and general formulation of logic program updating, particularly suited for Multi-Agent Systems \cite{4,5}.

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).

Elsewhere \cite{1}, we present a transformation that takes an evolving logic program \( P \) and a sequence of events \( \mathcal{E} \) as input and outputs an equivalent normal logic program \( P_{\mathcal{E}} \). The aim of this work is to present an implementation of EVOLP based on this transformation. The implementation can be easily integrated with other existing multi-agent programming frameworks such as Jason \cite{6}, 2APL \cite{7} and 3APL \cite{8}, among others.

---

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

\( \copyright \) Springer-Verlag Berlin Heidelberg 2008
The remainder of this work is structured as follows: in Sect. 2 we introduce the syntax and semantics of EVOLP and the transformation from [1]; in Sect. 3 we present the implementation; in Sect. 4 we conclude and discuss future work.

2 Preliminaries

First we present the syntax and semantics of Dynamic Logic Programs and Evolving Logic Programs (EVOLP) and also a simple example that shows how EVOLP can be used to program a simple agent.

Let $\mathcal{L}$ be a set of propositional atoms. A default literal is an atom preceded by $\neg$. 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 $B(r) = \{L_1, L_2, ..., L_n\}$ will simply be written as

$$L_0 \leftarrow L_1, L_2, ..., L_n.$$  \hspace{1cm} (1)

If $H(r) = A$ (resp. $H(r) = \neg A$) then $\neg H(r) = \neg A$ (resp. $\neg H(r) = A$).

Two rules $r, r'$ are conflicting, denoted by $r \triangleright \triangleright r'$, iff $H(r) = \neg H(r')$. We will say a literal $L$ appears in a rule (1) iff the set $\{L, \neg L\} \cap \{L_0, L_1, L_2, ..., L_n\}$ 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 $\neg A$ is true in $I$, denoted by $I \models \neg A$, 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 $I^− \stackrel{\text{def}}{=} \{\neg A \mid A \in \mathcal{L} \setminus I\}$ 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 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 $\mathcal{P} = (P_1, P_2, ..., P_n)$ be a DLP. We use $\rho(\mathcal{P})$ to denote the multiset of all rules appearing in the programs $P_1, P_2, ..., P_n$ 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

$$\text{Def}(\mathcal{P}, I) \stackrel{\text{def}}{=} \{\neg A \mid (\exists r \in \rho(\mathcal{P}))(H(r) = A \land I \models B(r))\},$$  \hspace{1cm} (2)

$$\text{Rej}^j(\mathcal{P}, I) \stackrel{\text{def}}{=} \{r \in \mathcal{P}^j \mid (\exists k, r')(k \geq j \land r' \in \mathcal{P}^k \land r \triangleright \triangleright r' \land I \models B(r'))\},$$  \hspace{1cm} (3)

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

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