EVOLP – Transformation-based Implementation*

Martin Slota[1] and João Leite[2]

[1] Katedra aplikovanej informatiky, Univerzita Komenského, Slovakia
[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 some extensions and optimizations that could be used to increase performance.

1 Introduction

Evolving Logic Programming (EVOLP) [2] is a generalization of Answer Set Programming [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 simpler, and more general, formulation of logic program updating, particularly suited for Multi-Agent Systems [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). The following example shows how EVOLP can be used to program a simple agent:

Example 1. Let's consider a simple agent that fills glasses with water. If it receives a request from the environment (e.g. when somebody presses a button), it starts filling a glass with water. When the glass is full, it stops filling it. This evolving logic program encodes the described behaviour[3]: $P =$

* 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.

[3] with a small difference: the agent's reactions are always delayed one step

$\{\text{assert}(\text{fill} \leftarrow) \leftarrow \text{request.}, \text{assert}(\textbf{not}\, \text{fill} \leftarrow) \leftarrow \text{full.}\}$. In each step the agent also receives an input from the environment, also in the form of an evolving logic program. These programs will be called *events*. Let's consider a sequence of four events $\mathcal{E} = (E_1, E_2, E_3, E_4)$ where $E_1 = \{\text{request} \leftarrow .\}$, $E_2 = E_4 = \emptyset$ and $E_3 = \{\text{full} \leftarrow .\}$. The semantics of P w.r.t. $\mathcal{E}$ is a single sequence of four models (M_1, M_2, M_2, M_4) where $M_1 = \{\text{request}, \text{assert}(\text{fill} \leftarrow)\}$, $M_2 = \{\text{fill}\}$, $M_3 = \{\text{fill}, \text{full}, \text{assert}(\textbf{not}\, \text{fill} \leftarrow)\}$ and $M_4 = \emptyset$. Its meaning is that in the first step the agent receives a request in E_1, in the second it starts filling the glass, in the third it receives the signal to stop filling it and in the fourth it does nothing, i.e. it stops filling it.

Now we can complicate things a bit. Let's say the water was too warm and a cooling system was installed into the agent. It's behaviour also needs to be changed – it should ignore the request button until the water is cold enough. This reprogramming can be done on the run, through the event $E_5 = \{\text{assert}(\textbf{not}\, \text{assert}(\text{fill} \leftarrow) \leftarrow \textbf{not}\, \text{cold}) \leftarrow .\}$. It causes the program to be updated with the rule from inside the assert/1 predicate above. This rule disallows filling the glass if the agent doesn't receive a signal that the water is cold enough. If, for example, the agent further receives the events $E_6 = \{\text{request} \leftarrow .\}$, $E_7 = \{\text{request} \leftarrow ., \text{cold} \leftarrow .\}$ and $E_8 = \emptyset$, then the corresponding models will be $M_6 = \{\text{request}\}$, $M_7 = \{\text{request}, \text{cold}, \text{assert}(\text{fill} \leftarrow)\}$ and $M_8 = \{\text{fill}\}$. In other words, the first request was ignored while the second was accepted because the water was already cold.

For more on EVOLP, namely its syntax and declarative semantics, the reader is referred to [2].

2 Transformational Semantics for EVOLP

Elsewhere [1], we present a transformation that takes an evolving logic program P and a sequence of events $\mathcal{E} = (E_1, E_2, \ldots, E_n)$ as input and outputs an equivalent normal logic program $P_{\mathcal{E}}$.

Let's take a closer look at $P_{\mathcal{E}}$. It consists of n subprograms, i.e. $P_{\mathcal{E}} = P_{\mathcal{E}}^1 \cup P_{\mathcal{E}}^2 \cup \ldots \cup P_{\mathcal{E}}^n$. Each $P_{\mathcal{E}}^j$ is used to simulate the j-th evolution step on a separate set of atoms. In the first evolution step, the only rules that need to be simulated are the rules of P and E_1. Each occurrence of a literal L will be written as L^1.

The situation is more complicated in the further steps because the program can be updated by new rules. If we want to simulate the j-th evolution step, we need to include rules of P and E_j and also rules that could have been asserted in some previous evolution step. This means that whenever an atom $(\text{assert}(r))^i$ appears in the head of some rule of $P_{\mathcal{E}}^i$ for some $i \in \{1, 2, \ldots, j-1\}$, the rule r must be included in $P_{\mathcal{E}}^j$ in a special rewritten form. Apart from rewriting each literal L as L^j, we need to make sure the rule can only be used in case $(\text{assert}(r))^i$ is actually inferred by some rule of $P_{\mathcal{E}}^i$. This is achieved by adding $(\text{assert}(r))^i$ to the body of the rewritten form of r. This rewritten form is then called an *assertable rule*.

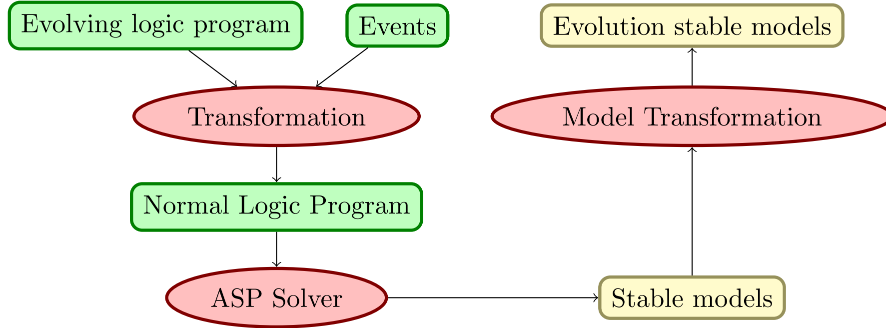

Fig. 1. Implementation of EVOLP using the transformation

Moreover, each of the mentioned rules will contain one extra literal in its body that marks it as a rule of a certain level. Rules with smaller levels can be rejected by higher level rules in case a conflict between them arises. Rules originating directly from P are of level 1 and rules from E_j are of level j, the highest level in $P^j_{\mathcal{E}}$. If a rule r was added because $(\text{assert}(r))^i$ appears in the head of some rule of $P^i_{\mathcal{E}}$ (i.e. the added rule is an assertable rule), it is marked with level $i+1$ because it can be asserted into the $(i+1)$-th step.

Taken all together, each $P^j_{\mathcal{E}}$ consists of six groups of rules. We are only interested in the first three groups, the other three are used to simulate the rule rejection mechanism behind the Refined Dynamic Stable Model semantics for Dynamic Logic Programming [6]. The first group contains rules originating from P, the second group contains rules originating from E_j and the third group contains all assertable rules.

3 Implementation of EVOLP

The described transformation, together with an ASP solver, can be used to implement EVOLP. Figure 3 shows how we can do this – first we take an evolving logic program and a sequence of events as input and use the transformation to produce an equivalent normal logic program. Then we use the ASP solver to find the stable models of the normal logic program and reconstruct the evolution stable models of the original input.

Our implementation is written in Java, this way it can be easily integrated with many of the existing agent programming frameworks. It uses Smodels[4] as an external ASP solver. A special version that supports only a propositional language is just a straightforward use of the transformation. Currently it can run as a simple web application[5] that can be used to enter an evolving logic program and compute some or all of its models.

[4] `http://www.tcs.hut.fi/Software/smodels/`

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

The language with variables is more interesting because using the transformation directly implies quite big restrictions on when and where it is possible to use the variables. The following are three types of rules that we believe should be fully acceptable, but are not allowed when the transformation is used directly:

$$\text{assert}(X) \leftarrow \text{says}(\text{joe}, X). \tag{1}$$

$$\text{assert}(\text{a}(X) \leftarrow) \leftarrow \text{b}(X). \tag{2}$$

$$\text{assert}(\text{a}(X,Y) \leftarrow \text{b}(Y)) \leftarrow \text{c}(X). \tag{3}$$

Their common property is that some variable inside the assert/1 predicate is bound in the body of the rule and hence the rule that can be asserted may take different forms. In each evolution step, depending on what atoms are inferred, a different set of assertable rules can be generated by such rules. Moreover, the set of inferred atoms may depend on what rules have been asserted before, just like in the program $\{\text{a}(1) \leftarrow ., \text{assert}(\text{b}(X) \leftarrow) \leftarrow \text{a}(X)., \text{assert}(\text{c}(X) \leftarrow) \leftarrow \text{b}(X).\}$.

The problems with rules (1) and (2) can be solved by constructing the transformed program incrementally, one evolution step at a time, and grounding the partially constructed program to find what rules can be asserted. Let's assume we already constructed the programs $P^1_{\mathcal{E}}, P^2_{\mathcal{E}}, \ldots, P^{j-1}_{\mathcal{E}}$. In order to construct $P^j_{\mathcal{E}}$, we need to know what assertable rules to include. These can be inferred from a grounded version of $P^1_{\mathcal{E}} \cup P^2_{\mathcal{E}} \cup \ldots \cup P^{j-1}_{\mathcal{E}}$. Consequently, $P^j_{\mathcal{E}}$ can be constructed and the process can be iterated.

However, from the practical point of view, it is easier and more efficient not to construct the whole $P^j_{\mathcal{E}}$ before grounding it. Instead, only the first three groups of rules can be constructed and changed syntactically so that the grounder produces an appropriate grounded version. There are two advantages of this approach. The first is that the second three groups of rules (those simulating the rule rejection mechanism) are much more difficult to construct when variables are still present in the first three groups. The second is that instead of grounding $P^1_{\mathcal{E}} \cup P^2_{\mathcal{E}} \cup \ldots \cup P^j_{\mathcal{E}}$ we only ground the first part of $P^j_{\mathcal{E}}$. On the other hand, we need to take care of remembering the level of rules when they come out of the grounder and also their status – whether they are assertable or not. This can be performed by adding dummy literals to their bodies before they are given to the grounder and filtering them out in their grounded versions.

The problem with rule (3) is more subtle – only a subset of variables inside the assert/1 predicate appears in the rule's body and the remaining variables will be rejected by the grounder. One possible solution is to encode Y as a constant and turn it back into a variable at the time of rewriting the rule into an assertable rule. But this brings a different problem illustrated in this example:

Example 2. Let's take the program

$$\{\text{assert}(\text{a}(X) \leftarrow \text{b}(X)) \leftarrow ., \text{b} \leftarrow \text{assert}(\text{a}(1) \leftarrow \text{b}(1)).\}$$

In case we encode X as a constant enc_X, then b will not be true in the model. If we would like it to be there we need to add the rule

$$\text{assert}(\text{a}(1) \leftarrow \text{b}(1)) \leftarrow \text{assert}(\text{a}(\text{enc_X}) \leftarrow \text{b}(\text{enc_X})).$$

to the transformed program.

The current implementation of EVOLP with variables supports variables outside the assert/1 predicate similarly as Lparse[6] – all variables in the input program must be bound by some domain predicate. It also handles rule (1) correctly, but in case of rules like (2) and (3), it considers variables inside the assert/1 predicate independently from variables in the rule's body. By the time of the workshop we hope to have a newer version that handles these cases more naturally. The current version is available through another web form[7].

4 Optimizations and Future Work

The mentioned implementation includes optimizations that prevent the generation of some unnecessary rules, even though these rules should be generated according to the definition of the transformation. There are also other ways of optimizing the resulting program, namely by sharing rules among evolution steps when possible. This can significantly reduce the size of the transformed program and such an optimization could be included in later versions of the implementation. The implementation can also be extended to support strong negation, weight constraints, arithmetic predicates and other practical features.

References

1. M. Slota and J. A. Leite. Operational semantics for EVOLP. In F. Sadri and K. Satoh, editors, *Pre-Proceedings of the 8th Workshop on Computational Logic in Multi-Agent Systems*, 2007.
2. 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.
3. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. Warren and P. Szeredi, editors, *Proceedings of the 7th international conference on logic programming*, pages 579–597. MIT Press, 1990.
4. 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.
5. 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.
6. 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.

[6] `http://www.tcs.hut.fi/Software/smodels/`
[7] runs at `http://www.ii.fmph.uniba.sk/∼slota/evolp-var/`