EVOLP: an Implementation*

Martin Slota[1,2] 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 optimizations used in the implementation.

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 simple and 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).

Elsewhere [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 [6], 2APL [7] and 3APL [8], among others.

* 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.

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 **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 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 $M^* =$* 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: – 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* 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. 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]: $P = \{\text{assert}(\text{fill} \leftarrow) \leftarrow \text{request}., \text{assert}(\textbf{not}\ \text{fill} \leftarrow) \leftarrow \text{full}.\}$. In each step the agent also receives an event from the environment. Let's consider a sequence of four events $\mathcal{E} = (E_1, E_2, E_3, E_4)$ where $E_1 = \{\text{request} \leftarrow .\}$, $E_2 = \emptyset$, $E_3 = \{\text{full} \leftarrow .\}$ and $E_4 = \emptyset$. 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 will complicate things a bit. Let's say the water was too warm and a cooling system was installed into the agent. Its behaviour also needs to be changed – it should ignore the request button until the water is cold enough. In EVOLP events can be used to program the agents. In this particular case the reprogramming is done through the event $E_5 = \{\text{assert}(\textbf{not}\ \text{assert}(\text{fill} \leftarrow) \leftarrow \textbf{not}\ \text{cold}) \leftarrow .\}$. The rule asserted in E_5 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.

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

The previous example is just a very simple one. The rules can be much more complex and can be used to capture very complicated behaviours. Evolution branching may also occur, allowing us to reason about incomplete information. For more examples the reader is referred to [4,5].

Now we will present the 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. In [1] we show that there is a one-to-one correspondence between the stable models of $P_{\mathcal{E}}$ and the evolution stable models of P given $\mathcal{E}$. The computational complexity of the transformation is also examined there.

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 **not** A, then L^j is A^j_{neg} and L^j_{neg} is A^j.
- $Body^j = \{L^j \mid L \in Body\}$.

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 $(L \leftarrow Body.) \in P$, $P^j_{\mathcal{E}}$ contains the rule*
$$L^j \leftarrow Body^j, \textbf{not}\, \text{rej}(L^j, 1).$$
2. ***Rewritten event rules.*** *For every rule $(L \leftarrow Body.) \in E_j$, $P^j_{\mathcal{E}}$ contains the rule*
$$L^j \leftarrow Body^j, \textbf{not}\, \text{rej}(L^j, j).$$
3. ***Assertable rules.*** *For every rule $r = (L \leftarrow Body.)$ over $\mathcal{L}_{\text{assert}}$ and all $i \in \{1, 2, \ldots, j-1\}$ such that $(\text{assert}(r))^i$ is in the head of some rule of $P^i_{\mathcal{E}}$, $P^j_{\mathcal{E}}$ contains the rule*
$$L^j \leftarrow Body^j, (\text{assert}(r))^i, \textbf{not}\, \text{rej}(L^j, i+1).$$

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*

$$A^j_{\text{neg}} \leftarrow \mathbf{not}\, \text{rej}(A^j_{\text{neg}}, 0). \tag{5}$$

5. ***Rejection rules.*** *For every rule of* $P^j_{\mathcal{E}}$ *of the form*

$$L^j \leftarrow \mathit{Body}, \mathbf{not}\, \text{rej}(L^j, i).^4$$

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

$$\text{rej}(L^j_{\text{neg}}, p) \leftarrow \mathit{Body}. \tag{6}$$

$$\text{rej}(L^j, q) \leftarrow \text{rej}(L^j, i). \tag{7}$$

where:

(a) $p \leq i$ *is the largest index such that* $P^j_{\mathcal{E}}$ *contains a rule with the literal* $\mathbf{not}\, \text{rej}(L^j_{\text{neg}}, p)$ *in its body. If no such* p *exists, then* (6) *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* $\mathbf{not}\, \text{rej}(L^j, q)$ *in its body. If no such* q *exists, then* (7) *is not in* $P^j_{\mathcal{E}}$.

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

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

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

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

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

Let's take a closer look at $P_{\mathcal{E}}$. It consists of n subprograms $P^1_{\mathcal{E}}, P^2_{\mathcal{E}}, \ldots, P^n_{\mathcal{E}}$. Each $P^j_{\mathcal{E}}$ 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. They are simulated in the groups of rewritten program rules and rewritten event rules. Each occurrence of a literal L is written as L^1.

In the further steps there are more rules coming into play. In order to simulate the j-th evolution step, we need to simulate rules of P and E_j and also rules that could have been asserted in some previous evolution step. So apart from the rewritten program rules and rewritten event rules we also have assertable rules – whenever an atom $(\text{assert}(r))^i$ appears in the head of some rule of $P^i_{\mathcal{E}}$ for some $i \in \{1, 2, \ldots, j-1\}$, the rule r is included in $P^j_{\mathcal{E}}$ 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 $P^i_{\mathcal{E}}$. This is achieved by adding $(\text{assert}(r))^i$ to the body of the rewritten form of r.

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

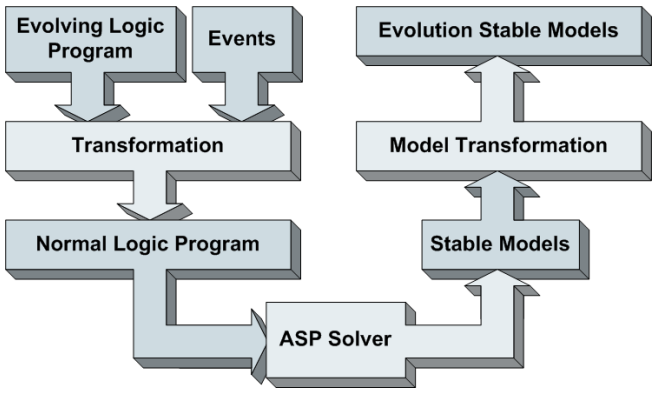

Fig. 1. Implementation of EVOLP using the transformation

Moreover, each of the mentioned rules contains one extra literal of the form **not** $\mathrm{rej}(L^j, i)$ in its body that marks it as a rule of level i. Rules with smaller levels can be rejected by higher level rules in case a conflict between their heads 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 $(\mathrm{assert}(r))^i$ 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.

The other three groups of rules (default assumptions, rejection rules and totality constraints) are used to simulate the rule rejection mechanism behind the Refined Dynamic Stable Model semantics for Dynamic Logic Programming [9]. For more information on the transformation the reader is referred to [1].

3 Implementation of EVOLP

The described transformation, together with an ASP solver, can be used to implement EVOLP. Figure 1 shows how we can do this – we use the transformation to turn an evolving logic program and a sequence of events into an equivalent normal logic program. Then we use an ASP solver to find its stable models and reconstruct the evolution stable models of the original input.

One of the objectives was to make the implementation easy to integrate with existing multi-agent programming frameworks. Since many of them are written in Java, EVOLP is also implemented in Java. The implementation, however, uses Lparse as an external grounder and Smodels[5] as an external ASP solver.

Variables are supported and restricted in the same way as in Lparse. More specifically, every variable must be bound in the positive part of the rule's body by some domain predicate[6].

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

[6] Domain predicates are those that are defined without recursion or using positive recursion only. A more detailed description with an example can be found

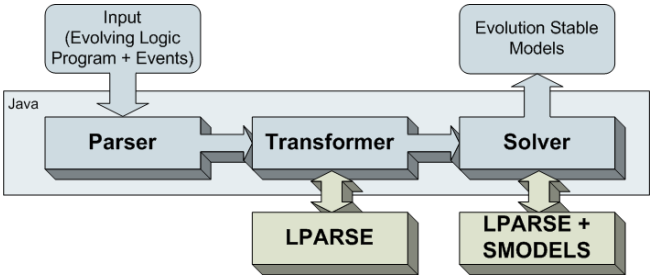

Fig. 2. Data processing steps

As illustrated in Fig. 2, the input program is processed in three steps, each implemented by a separate Java class:

1. The input text is parsed and an object representation of the evolving logic program and events is created.
2. An equivalent normal logic program is produced according to a slightly more optimized version of the transformation defined in Def. 5. Lparse is used during the transformation to ground any variables in the input program. A more detailed description of this step is given below.
3. Lparse and Smodels are executed on the transformed program in a separate process. The resulting stable models are parsed and evolution stable models of the original input are reconstructed.

The implementation can currently run as a simple web application[7]. It can be used to enter an evolving logic program and compute some or all of its evolution stable models. Figure 3 shows a screenshot of the web application with the source and computed evolution stable models of the program from Example 1.

3.1 Implementation of the Transformation

The transformed program is constructed in incrementally and the partially constructed program is always used to construct the next part of the result. Let's assume we already constructed the programs $P_{\mathcal{E}}^1, P_{\mathcal{E}}^2, \ldots, P_{\mathcal{E}}^{j-1}$. In order to construct $P_{\mathcal{E}}^j$, we need to know what assertable rules to include. These can be inferred from a grounded version of $P_{\mathcal{E}}^1 \cup P_{\mathcal{E}}^2 \cup \ldots \cup P_{\mathcal{E}}^{j-1}$. Consequently, $P_{\mathcal{E}}^j$ can be constructed and the process can be iterated.

However, from a practical point of view, it is not possible to use Lparse to perform the transformation exactly as described. The problem is that the predicate rej/2 might be a non-domain predicate and still might contain variables

in the Lparse Manual which is included in the Lparse source package available at `http://www.tcs.hut.fi/Software/smodels/`.

[7] a demo runs at `http://centria.di.fct.unl.pt/evolp/`

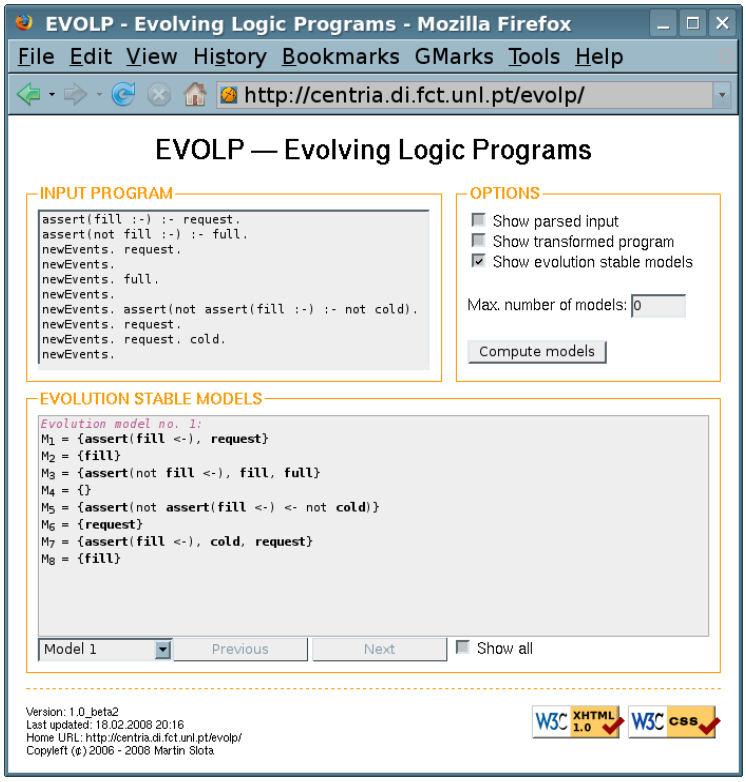

Fig. 3. Screenshot of the web application with the program from Example 1. The program source is showed in the bottom part and the single evolution stable model is listed in the upper part.

in some of the second type of rejection rules (7). Fortunately, there is a better solution of the whole problem – it avoids the mentioned problem and is also more efficient. Instead of constructing the whole $P_{\mathcal{E}}^j$ before grounding it, only the first three groups of rules can be constructed and changed syntactically so that the grounder produces an appropriate grounded version. This way we only ground a part of $P_{\mathcal{E}}^j$ instead of grounding $P_{\mathcal{E}}^1 \cup P_{\mathcal{E}}^2 \cup \ldots \cup P_{\mathcal{E}}^j$. 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 origin – whether they are assertable or not. This can be performed by adding dummy literals to their bodies before they are given to Lparse and filtering them out in their grounded versions.

3.2 Optimizations

The presented implementation includes some simple optimizations that prevent the generation of some unnecessary rules, even though these rules should be

generated according to the formal definition of the transformation. In particular, we don't generate all default assumptions as in the definition, we generate a default assumption (5) only in case the atom A^j_{neg} appears in a body of some rewritten program rule, rewritten event rule or assertable rule.

Another optimization involves the second type of rejection rules (7). They are only generated if they are "reachable", i.e. a rule $\text{rej}(L^j, q) \leftarrow \text{rej}(L^j, i)$ is generated only in case $P_{\mathcal{E}}$ contains another rejection rule (6) of the form $\text{rej}(L^j, p) \leftarrow \mathit{Body}$ for some $p \geq i$.

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. We plan to include such an optimization in later versions of the implementation. Other possible optimizations include minimizing the amount of data transmitted between Lparse and Java by giving shorter (numeric) names to predicates and experimenting with modifications of the transformation that would produce equivalent normal programs that perform better with the current answer set solvers. We also plan to design a set of benchmark tests and perform them with different versions of the implementation.

3.3 Using the Implementation as a Library

Our implementation can easily be used as an external library from Java. If the binary .jar file from `http://centria.di.fct.unl.pt/evolp/` is included in the CLASSPATH, then the easiest way to use the implementation is to override the `lp.ui.EvolpVarProcessor` class. Figure 4 shows a complete source code of a Java class that uses the library to print out the evolution stable model of the program from Example 1. It produces the following output containing the expected evolution stable model:

```
Evolution stable model no. 1
Step 1: assert(fill <-), request,
Step 2: fill,
Step 3: assert(not fill <-), fill, full,
Step 4:
Step 5: assert(not assert(fill <-) <- not cold),
Step 6: request,
Step 7: assert(fill <-), cold, request,
Step 8: fill,
```

4 Conclusion and Future Work

We presented an implementation of EVOLP that is based on a transformation into an equivalent normal logic program. We also discussed some basic optimizations performed in the implementation.

We are currently extending the implementation to support strong negation, domain declarations, weight constraints, arithmetic predicates and other practical features. Apart from that, we plan to examine further optimizations of the implementation and perform some benchmark tests.

```
class EvolpTest extends EvolpVarProcessor {
    protected void showMessage(String message) {
        // uncomment to see logging output
        // System.out.println("-- LOG MESSAGE: " + message);
    }

    public static void main(String [] args) {
        EvolpTest et = new EvolpTest();
        // set the program and events from Example 1 as input
        et.setInput(new StringReader(
            "assert(fill :-) :- request." +
            "assert(not fill :-) :- full." +
            "newEvents. request." +
            "newEvents." +
            "newEvents. full." +
            "newEvents." +
            "newEvents." +
                "assert(not assert(fill :-) :- not cold)." +
            "newEvents. request." +
            "newEvents. request. cold." +
            "newEvents."));
        // compute evolution stable models and give them one
        // by one to the TestConsumer.compute() method
        et.computeModels(new TestConsumer());
    }

    static class TestConsumer
        extends AbstractConsumer<EvolutionStableModel> {
        private int count = 0;

        public void consume(EvolutionStableModel model) {
            count++;
            System.out.print("Evolution stable model no. ");
            System.out.println(count);
            int step = 0;
            for (StableModel m : model) {
                step++;
                System.out.print("Step " + step + ": ");
                for (LpAtom a : m)
                    System.out.print(a.toString() + ", ");
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

Fig. 4. Simple use of the implementation directly from Java

References

1. M. Slota and J. A. Leite. EVOLP: Transformation-based semantics. In F. Sadri and K. Satoh, editors, *Proceedings of the 8th Workshop on Computational Logic in Multi-Agent Systems (CLIMA VIII)*, 2008. In this volume.
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. Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.
7. M. Dastani, D. Hobo, and J.-J. Ch. Meyer. Practical extensions in agent programming languages. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07)*. ACM Press, 2007.
8. 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.
9. 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.