COMENIUS UNIVERSITY
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
DEPARTMENT OF APPLIED INFORMATICS
BRATISLAVA, SLOVAKIA

Transformational Semantics and Implementation of Evolving Logic Programs

Master's Thesis

Martin Slota
author

João Alexandre Leite, MSc., PhD.
doc. PhDr. Ján Šefránek, PhD.
advisors

BRATISLAVA MAY 2007

Transformational Semantics and Implementation of Evolving Logic Programs

Master's Thesis

Martin Slota

COMENIUS UNIVERSITY
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
DEPARTMENT OF APPLIED INFORMATICS
BRATISLAVA, SLOVAKIA

Study Programme: 2508800 Informatics

João Alexandre Leite, MSc., PhD.
doc. PhDr. Ján Šefránek, PhD.

BRATISLAVA, MAY 2007

Declaration

I hereby declare that this thesis is my own work, only with the help of the referenced literature and under the careful supervision of my thesis advisors.

Bratislava, May 2007 Martin Slota

Acknowledgements

A very big thanks goes to my advisors João Alexandre Leite and Ján Šefránek for their supervision and priceless help with this work.

I would also like to thank my parents and my brother for their gentle guidance, patience and support ever since I was born.

Last but not least, my thanks goes to Martin Baláž, Peter Klimo, Michal Malý and Jozef Šiška for their questions and ideas.

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 multiagent systems, planning and other uses where information tends to change dynamically.

This work characterizes **EVOLP** by transforming it into an equivalent normal logic program. The proposed transformation is further examined and used to design and write the first freely available, extensible and reusable implementation of **EVOLP** under the evolution stable model semantics.

Keywords: Logic Programming, Stable Model Semantics, Evolving Logic Programs, Transformational Semantics, Implementation

Preface

In the last years, a lot of effort has been invested in finding a language suitable for specifying and programming the evolution of knowledge bases represented as logic programs. Such a language could be used to declaratively program intelligent agents and multiagent systems. Evolving logic programs (**EVOLP**) is one of the languages developed for this purpose and unlike its predecessors, it is just a simple, yet powerful extension of traditional logic programming.

The aim of this work is to examine the possibilities of implementing **EVOLP** under the stable model semantics. First, we focused on defining a sound and complete transformation that would produce an equivalent normal logic program for any given evolving logic program (a so-called transformational semantics). Then we used the transformation to implement propositional **EVOLP** and tried to face the problems with introducing variables into the language.

The result is a partial implementation of **EVOLP** with variables. It has been designed with maintainability, extensibility, and reusability in mind. In Chap. 4 we sketch what needs to be done to finish the support for variables. The implementation can also be extended with other practical features, e.g. support for weight constraints, arithmetic predicates and strong negation.

Bratislava, May 2007 Martin Slota

Contents

List of Figures

Chapter 1

Introduction and Motivation

1.1 Logic Programming and Intelligent Agents

Construction of intelligent agents is one of the main matters of artificial intelligence. Such agents should be capable of operating independently in a partially observable environment that may change unexpectedly. Therefore, they need to be able to update their model of the world according to the changes that take place in them and around them.

Logic programming showed as a good tool for both symbolic knowledge representation and hypothetical reasoning. Much research in the last decade has been devoted to finding a good way of updating knowledge represented by logic programs. A sequence of logic programs where each program is an update of the preceding programs was called a Dynamic Logic Program (**DLP**). Finding a suitable semantics for **DLP**s became the first step on the path to using logic programming in agent systems. Quite a number of semantics with different properties were introduced. We will only mention the Dynamic Stable Model semantics [1, 2, 3, 4] that was later improved and called Refined Dynamic Stable Models [5, 6, 7]. This is also the semantics used throughout this work. For a more comprehensive overview of semantics for **DLP**s see [8, 9].

Although dynamic logic programming provides a way of updating a logic program by another logic program, it still doesn't tell us how we should construct these programs. Update languages like LUPS [10, 11], EPI

[12], KUL and KABUL [8] were developed for the purpose of incrementally constructing a sequence of logic programs. Each of them specifies special types of rules for adding and deleting logic programming rules from programs in the sequence. Evolving Logic Programs (**EVOLP**) [13] also comes from this line of work, but while its predecessors were getting more and more complicated as more constructs were being added, **EVOLP** is a simple, yet powerful extension of traditional logic programming. Syntactically, evolving logic programs are just generalized logic programs. Semantically, they allow for arbitrary updates of the program by adding new rules to it, both by self-updates and by updates from the environment (through events).

1.2 The Roadmap

We believe **EVOLP** is an interesting language with a neat idea behind it and as such it is worth implementing. An implementation running under the well-founded semantics has been available for quite some time [14]. But an implementation of the evolution stable model semantics from the papers about **EVOLP** appears only in [15, 16] and only for a limited constructive view of propositional **EVOLP**.

The aim of this work is to examine the problems with implementing **EVOLP** and providing a freely available and reusable implementation of **EVOLP** under the evolution stable model semantics. The first steps lead to defining the transformational semantics for **EVOLP**, i.e. a sound and complete transformation that turns an arbitrary evolving logic program into an equivalent normal logic program. The proposed transformation is then used to implement propositional **EVOLP** and practical problems with introducing variables are examined and partially resolved.

The remainder of this work is structured as follows:

Chapter 2 – Preliminaries: This Chapter presents the syntax and semantics of logic programs, dynamic logic programs and evolving logic programs. It only contains the definitions and theorems needed later in the work. We also take a look at the transformational semantics for **DLP**s [17] because our transformational semantics for **EVOLP** is based on it.

Chapter 3 – Transformational Semantics for EVOLP: Here we define the transformational semantics for **EVOLP** and prove that it is sound and complete with respect to the evolution stable model semantics. We also infer the lower and upper bounds for the size of the transformed program.

Chapter 4 – Implementation of EVOLP: This Chapter contains a description of the implementation based on the defined transformation. It also presents the problems with variables in **EVOLP** and proposes some solutions.

Chapter 5 – Conclusion and Future Work: In the last Chapter we sum up this work and sketch some direction of future research.

On the last two pages there is a "Definition Index" containing concepts and notation introduced throughout this work together with the numbers of pages where they are defined.

Chapter 2

Preliminaries

This Chapter contains a collection of definitions and theorems that we will use subsequently. In Sect. 2.1 we briefly introduce the syntax and semantics of generalized logic programs and formulate some related propositions and theorems. We only consider a propositional language.

Dynamic logic programs (**DLP**s) are presented in Sect. 2.2. Special attention is paid to the transformational semantics for **DLP**s from [17] because our transformational semantics for **EVOLP** proposed in Chap. 3 is based on it.

Section 2.3 contains the definition of evolving logic programs.

2.1 Logic Programs

This section contains some definitions and theorems from the wide area of logic programming. For a more thorough overview see [18, 8]. We will start off by defining the syntax of propositional logic programs.

2.1.1 Syntax

Definition 2.1 (Atoms and literals)**.** Let $\mathcal{L}$ be an arbitrary denumerable set of propositional atoms (a language). An *atom of* $\mathcal{L}$ is any $A \in \mathcal{L}$. A *default literal over* $\mathcal{L}$ is an atom of $\mathcal{L}$ preceded by a "**not**" representing default negation. A *literal over* $\mathcal{L}$ is either an atom of $\mathcal{L}$ or a default literal over $\mathcal{L}$.

The set of all literals[1] is denoted by $\mathcal{L}^*$, i.e. $\mathcal{L}^* = \mathcal{L} \cup \{\mathbf{not}\, A \mid A \in \mathcal{L}\}$.

Let L be a literal. If L is a default literal $\mathbf{not}\, A$, then $\mathbf{not}\, L$ denotes the atom A. Similarly, if L is an atom A, then $\mathbf{not}\, L$ denotes the default literal $\mathbf{not}\, A$.

Definition 2.2 (Rules)**.** A *rule r over $\mathcal{L}$* is an ordered pair $(H(r), B(r))$ where $H(r)$ (dubbed the head of the rule) is a literal over $\mathcal{L}$ and $B(r)$ (dubbed the body of the rule) is a finite set of literals over $\mathcal{L}$. A rule with an empty body is called a *fact*.

A rule $r = (L_0, \{L_1, L_2, \ldots, L_n\})$ is usually written as

$$L_0 \leftarrow L_1, L_2, \ldots, L_n. \tag{2.1}$$

We say a literal L appears in a rule (2.1) iff the set

$$\{L, \mathbf{not}\, L\} \cap \{L_0, L_1, \ldots, L_n\}$$

is non-empty. Two rules r, r' are conflicting, denoted by $r \bowtie r'$, iff $H(r) = \mathbf{not}\, H(r')$.

A *definite rule over $\mathcal{L}$* is a rule containing only atoms of $\mathcal{L}$. A *normal rule over $\mathcal{L}$* is a rule with an atom in its head.

Definition 2.3 (Generalized logic program)**.** A *generalized logic program over $\mathcal{L}$* is a set of rules over $\mathcal{L}$. We say a literal L appears in a generalized logic program P iff L appears in some rule of P.

Definition 2.4 (Normal logic program)**.** A *normal logic program over $\mathcal{L}$* is a set of normal rules over $\mathcal{L}$.

Definition 2.5 (Definite logic program)**.** A *definite logic program over $\mathcal{L}$* is a set of definite rules over $\mathcal{L}$.

Remark. Every definite logic program is also a normal logic program. Every normal logic program is also a generalized logic program.

[1]in the remainder of the text, the words "over $\mathcal{L}$" will be dropped for the sake of readability where it is clear from the context which $\mathcal{L}$ we are talking about (just like here)

2.1.2 Semantics

First we will define a model-theoretic semantics of definite logic programs. Subsequently we will use this semantics to define the stable model semantics of generalized logic programs.

Definition 2.6 (Interpretation)**.** By an *interpretation over* $\mathcal{L}$ we mean any set of atoms $I \subseteq \mathcal{L}$. Given an interpretation I we define

$$I^- \overset{\text{def}}{=} \{\mathbf{not}\, A \mid A \notin I\} \quad ,$$
$$I^* \overset{\text{def}}{=} I \cup I^- \quad .$$

An atom A is true in an interpretation I, denoted by $I \models A$, if $A \in I$, and false otherwise. A default literal $\mathbf{not}\, A$ is true in I, denoted by $I \models \mathbf{not}\, A$, if $A \notin I$, and false otherwise. A set of literals B is true in I, denoted by $I \models B$, iff each literal in B is true in I.

Definition 2.7 (Model)**.** Interpretation M is a model of a generalized logic program P iff for every rule $r \in P$ the following condition holds: if $M \models B(r)$, then $M \models H(r)$.

Definition 2.8 (Minimal model)**.** A *minimal model of a generalized logic program* P is every model M of P such that no $I \subsetneq M$ is a model of P.

Theorem 2.9 (Least model of a definite logic program)**.** Let P be a definite logic program. Then P has a unique minimal model. This model is called the *least model of* P.

Remark. The least model is generally considered to be a good semantics for definite logic programs because it minimizes the set of atoms inferred by the program and all the other models are supersets of the least model.

Now let's take a look at how we can compute the least model.

Definition 2.10 (Immediate consequence operator)**.** The *immediate consequence operator* is for every definite logic program P and every interpretation I defined as

$$T_P(I) \overset{\text{def}}{=} \{A \mid (\exists r \in P)(H(r) = A \wedge B(r) \subseteq I)\} \quad .$$

Proposition 2.11 (Monotonicity of the immediate consequence operator)**.** The immediate consequence operator is monotone, i.e. for every definite logic program P and all interpretations I_1, I_2 such that $I_1 \subseteq I_2$ it holds that $T_P(I_1) \subseteq T_P(I_2)$

Proof. Follows easily from the definition. □

Theorem 2.12. Let P be a definite logic program, $M_0 = \emptyset$ and $M_{i+1} = T_P(M_i)$ for every $i \in \mathbb{N}$[2] . Then the least model of P is [3]

$$\bigcup_{i<\omega} M_i \ .$$

Example 2.13. Let's take the set of atoms $\mathcal{L} = \{\text{tired}, \text{sleepy}, \text{hungry}, \text{happy}\}$ and construct the definite logic program P over $\mathcal{L}$:

$$P: \quad \text{sleepy} \leftarrow \text{tired}. \tag{2.2}$$

$$\text{tired} \leftarrow . \tag{2.3}$$

$$\text{happy} \leftarrow \text{sleepy}, \text{hungry}. \tag{2.4}$$

These rules can be interpreted as follows: Rule (2.2) says that if I'm tired, then I'm also sleepy. Rule (2.3) says I'm tired. Rule (2.4) says that if I'm sleepy and hungry, I'm happy (because usually I eat too much before going to sleep and then I don't sleep very well). We can construct the least model M of P according to Thm. 2.12:

$$\begin{aligned}
M_0 &= \emptyset \ , \\
M_1 &= T_P(M_0) = \{A \mid (\exists r \in P)(H(r) = A \wedge B(r) \subseteq \emptyset\} \\
&= \{\text{tired}\} \ , \\
M_2 &= T_P(M_1) = \{A \mid (\exists r \in P)(H(r) = A \wedge B(r) \subseteq \{\text{tired}\}\} \\
&= \{\text{tired}, \text{sleepy}\} \ , \\
M_3 &= T_P(M_2) = \ldots = M_2 \ ,
\end{aligned}$$

[2]$\mathbb{N}$ is the set of all natural numbers, including 0

[3]ω is the first limit ordinal, for more details see [18]

and thus

$$\begin{aligned} M &= \bigcup_{i<\omega} M_i = \emptyset \cup \{\mathrm{tired}\} \cup \{\mathrm{tired}, \mathrm{sleepy}\} \cup \{\mathrm{tired}, \mathrm{sleepy}\} \cup \dots \\ &= \{\mathrm{tired}, \mathrm{sleepy}\} \quad . \end{aligned}$$

The following example shows that a normal logic program doesn't always have a least model and some of its minimal models can be "better" than the others:

Example 2.14. Let $\mathcal{L} = \{\mathrm{write\_thesis}, \mathrm{tired}\}$. We can construct the following normal logic program over $\mathcal{L}$:

$$P: \quad \mathrm{write\_thesis} \leftarrow \mathbf{not}\ \mathrm{tired}. \tag{2.5}$$

The interpretation of rule (2.5) is: If I have no evidence that I am tired, then I will continue writing the thesis. This program has 3 models, in particular

$$\begin{aligned} M_1 &= \{\mathrm{write\_thesis}\} \quad , \\ M_2 &= \{\mathrm{tired}\} \quad , \\ M_3 &= \{\mathrm{write\_thesis}, \mathrm{tired}\} \quad . \end{aligned}$$

Only M_1 and M_2 are minimal. We can also see that M_2 is not constructive – P contains no rule that could infer tired. On the other hand, M_1 is the most natural consequence of the program – we cannot infer tired, so we can use the rule (2.5) to infer $\mathrm{write\_thesis}$. Those minimal models that are constructive in this sense are called stable. Their definition follows.

Definition 2.15. Let S be a set of rules and literals over $\mathcal{L}$. By $\mathrm{least}(S)$ we'll denote the least model of the definite logic program P over $\mathcal{L}^*$ that consists of exactly these rules:

1. all rules from S [4],
2. the rule $(L \leftarrow .)$ for each literal $L \in S$.

[4] please note that although S can contain any rule, the rules in P are really definite because the language of P is $\mathcal{L}^* = \mathcal{L} \cup \{\mathbf{not}\ A \mid A \in \mathcal{L}\}$

Definition 2.16 (Stable model)**.** We say that an interpretation M is a *stable model* of a generalized logic program P iff

$$M^* = \text{least}\big(P \cup M^-\big) \quad .$$

Proposition 2.17. Let P be a generalized logic program, M its stable model and A an atom. Then

$$A \in M \iff (\exists r \in P)(H(r) = A \wedge M \models B(r)) \quad .$$

Proof. From Def. 2.16 and Thm. 2.12 we have that

$$M^* = \bigcup_{i<\omega} M_i$$

where $M_0 = \emptyset$ and $M_{i+1} = T_{P \cup M^-}(M_i)$ for every $i \in \mathbb{N}$.

Now let $A \in M$. Then $A \in M^*$ and therefore some $i \in \mathbb{N}$ exists such that $A \in M_{i+1}$. This means that a rule $r \in P$ exists such that $H(r) = A$ and $B(r) \subseteq M_i \subseteq M^*$. So $M \models B(r)$ and r is the rule we search for.

For the converse implication let's take some rule $r \in P$ such that $H(r) = A$ and $M \models B(r)$. This means that $B(r) \subseteq M^*$ and from the monotonicity of the immediate consequence operator we have that for some $i \in \mathbb{N}$ it must hold that $B(r) \subseteq M_i$. Therefore $A \in M_{i+1} \subseteq M^*$ and thus $A \in M$. $\square$

2.2 Dynamic Logic Programs

Syntactically, a dynamic logic program is simply a sequence of generalized logic programs. Semantically, the rules in each program of the sequence are preferred over rules from preceding programs.

Now we will introduce the syntax and the refined dynamic stable model semantics for **DLPs** (as it is defined in [7]). It is an improved version of the dynamic stable model semantics that can be found in [8].

Definition 2.18 (Dynamic logic program)**.** A *dynamic logic program over* $\mathcal{L}$ (**DLP**) is a sequence of generalized logic programs over $\mathcal{L}$. 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.

Definition 2.19 (Default assumptions)**.** Let $\mathcal{P}$ be a dynamic logic program and I an interpretation. Then

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

Definition 2.20 (Rejected rules)**.** Let $\mathcal{P}$ be a **DLP** of length n, I an interpretation and $j \in \{1, 2, \ldots, n\}$. Then:

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

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

Definition 2.21 (Refined dynamic stable model, [7])**.** Let $\mathcal{P}$ be a **DLP** and M an interpretation. M is a *(refined) dynamic stable model*[5] $\mathcal{P}$ iff

$$M^* = \mathrm{least}([\rho(\mathcal{P}) \setminus \mathrm{Rej}(\mathcal{P}, M)] \cup \mathrm{Def}(\mathcal{P}, M)) \quad .$$

Remark. The defined semantics is a generalization of the stable model semantics, i.e. the stable models of a generalized logic program P are the same as the dynamic stable models of the **DLP** $\mathcal{P} = (P)$. We can also see an analogy with the definition of stable models: $\rho(\mathcal{P}) \setminus \mathrm{Rej}(\mathcal{P}, M)$ plays the role of P and $\mathrm{Def}(\mathcal{P}, M)$ the role of M^-.

[5] in the remainder of this work we will always work with the refined dynamic stable model semantics but for the sake of readability we will drop the word "refined" everywhere

Example 2.22. Consider the following two generalized logic programs:

$$
\begin{aligned}
P_1 : \quad & \text{tired} \leftarrow . && (2.6)\\
& \text{drink\_coffee} \leftarrow \text{tired}. && (2.7)\\
& \text{write\_thesis} \leftarrow \textbf{not}\ \text{tired}. && (2.8)\\
P_2 : \quad & \textbf{not}\ \text{tired} \leftarrow . && (2.9)
\end{aligned}
$$

Rule (2.6) states that I'm tired. The other two rules in P_1 define what I will do depending on whether I'm tired or not. The rule (2.7) in P_2 states I'm not tired (any more).

P_1 has exactly one stable model: $M_1 = \{\text{tired}, \text{drink\_coffee}\}$. The dynamic logic program $\mathcal{P} = (P_1, P_2)$ has exactly one dynamic stable model: $M_2 = \{\text{write\_thesis}\}$. To verify that M_2 is really a dynamic stable model of $\mathcal{P}$ we need to check the following (according to Defs. 2.19, 2.20 and 2.21):

$$
\begin{aligned}
\text{Def}(\mathcal{P}, M_2) &= \{\textbf{not}\ \text{drink\_coffee}\} \quad ,\\
\text{Rej}(\mathcal{P}, M_2) &= \{\text{tired} \leftarrow .\} \quad ,\\
M_2^* &= \{\text{write\_thesis}, \textbf{not}\ \text{tired}, \textbf{not}\ \text{drink\_coffee}\}\\
&= \text{least}\left(\left\{\begin{array}{c}\text{drink\_coffee} \leftarrow \text{tired}.\\ \text{write\_thesis} \leftarrow \textbf{not}\ \text{tired}.\\ \textbf{not}\ \text{tired} \leftarrow .\end{array}\right\} \cup \{\textbf{not}\ \text{drink\_coffee}\}\right) \quad .
\end{aligned}
$$

2.2.1 Transformational Semantics

The transformational semantics for **EVOLP** that we will define in Chap. 3 is based on the transformational semantics for **DLP**s defined in [17]. On an example we will show how the transformation works.

Example 2.23. Let's take a **DLP** $\mathcal{P} = (P_1, P_2)$ where P_1 and P_2 are defined as in Ex. 2.22. If we use the transformation from [17], we will get the following normal logic program:

$$
\begin{aligned}
P : \quad & \text{tired}^- \leftarrow \textbf{not}\ \text{rej}(0, \text{tired}^-). && (2.10)\\
& \text{drink\_coffee}^- \leftarrow \textbf{not}\ \text{rej}(0, \text{drink\_coffee}^-). && (2.11)
\end{aligned}
$$

$$\mathrm{write\_thesis}^- \leftarrow \mathbf{not}\,\mathrm{rej}(0,\mathrm{write\_thesis}^-). \tag{2.12}$$
$$\mathrm{tired} \leftarrow \mathbf{not}\,\mathrm{rej}(1,\mathrm{tired}). \tag{2.13}$$
$$\mathrm{drink\_coffee} \leftarrow \mathrm{tired}, \mathbf{not}\,\mathrm{rej}(1,\mathrm{drink\_coffee}). \tag{2.14}$$
$$\mathrm{write\_thesis} \leftarrow \mathrm{tired}^-, \mathbf{not}\,\mathrm{rej}(1,\mathrm{write\_thesis}). \tag{2.15}$$
$$\mathrm{tired}^- \leftarrow \mathbf{not}\,\mathrm{rej}(2,\mathrm{tired}^-). \tag{2.16}$$
$$\mathrm{rej}(0,\mathrm{tired}^-) \leftarrow . \tag{2.17}$$
$$\mathrm{rej}(0,\mathrm{drink\_coffee}^-) \leftarrow \mathrm{tired}. \tag{2.18}$$
$$\mathrm{rej}(0,\mathrm{write\_thesis}^-) \leftarrow \mathrm{tired}^-. \tag{2.19}$$
$$\mathrm{rej}(1,\mathrm{tired}) \leftarrow . \tag{2.20}$$
$$\mathrm{rej}(0,\mathrm{tired}^-) \leftarrow \mathrm{rej}(2,\mathrm{tired}^-). \tag{2.21}$$

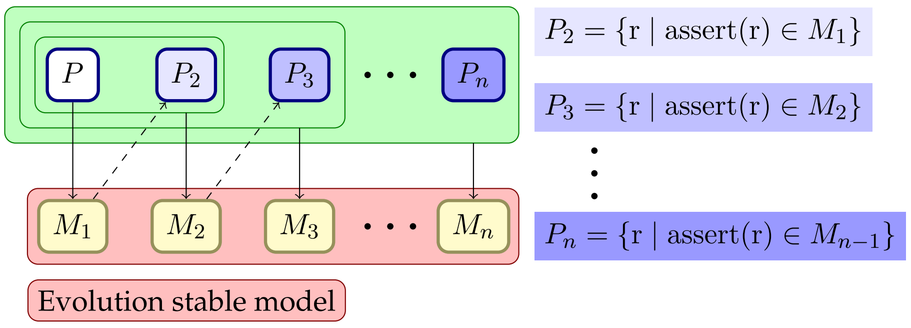
So what happened with the input program? The first big change is that all default literals were turned into new atoms. The reason is that in dynamic logic programming the set $\mathrm{Def}(\mathcal{P}, M)$ may be smaller than M^- (its counterpart in the stable model semantics), so we have to treat the default literals differently. In order to simulate the set of defaults, we add the rules (2.10) to (2.12). The next four rules (2.13) to (2.16) are just rewritten rules of the original programs. Now you must be asking: What are those "**not** $\mathrm{rej}(\dots)$" literals good for? They are employed as guards that either allow or disallow the use of each rule. And the remaining five rules (2.17) to (2.21) infer the correct $\mathrm{rej}(\dots)$ literals, also based on the original rules.

The program P has a unique stable model:

$$\begin{aligned} M = \{&\mathrm{write\_thesis}, \mathrm{tired}^-, \mathrm{drink\_coffee}^-,\\ &\mathrm{rej}(1,\mathrm{tired}), \mathrm{rej}(1,\mathrm{tired}^-), \mathrm{rej}(1,\mathrm{write\_thesis}^-)\}\ . \end{aligned}$$

M directly corresponds to the dynamic stable model $M_2 = \{\mathrm{write\_thesis}\}$ of the dynamic logic program $\mathcal{P}$.

Figure 2.1: Semantics of **EVOLP** (without events)



P
P_2
P_3
P_n
M_1
M_2
M_3
M_n
Evolution stable model
$P_2 = \{r \mid \text{assert}(r) \in M_1\}$
$P_3 = \{r \mid \text{assert}(r) \in M_2\}$
$P_n = \{r \mid \text{assert}(r) \in M_{n-1}\}$

2.3 Evolving Logic Programs

Evolving logic programs (**EVOLP**) can be seen as an extension of generalized logic programs. They can contain rules that add a new rule to the program in case they are fired. The newly added rule can also be of this type, so some rules can be added at the beginning and some only after the first set has been added. We can say the program evolves in steps. These steps are called evolution steps. In the semantics, the new rules after each evolution step are collected in a separate program and dynamic logic programming is used to prefer the rules that were added later.

In order to be usable in dynamic environments, there must also be some way of adding information coming from outside of the program. Otherwise an agent written in **EVOLP** would be unable to receive information from its sensors or to communicate with other agents. Therefore, after each evolution step an evolving logic program (called *event*) is added to the set of most recently added rules. The event can contain an arbitrary set rules that may even change or completely reprogram the behaviour of the agent.

Now we will define the syntax and semantics of **EVOLP** as it is defined in [13]. The semantics (with events excluded) is also illustrated in Fig. 2.1.

Definition 2.24 (Extended language)**.** Let $\mathcal{L}$ be a set of propositional atoms (not containing the predicate $\text{assert}/1$). The *extended language* $\mathcal{L}_\mathcal{A}$ is a minimal set of propositional atoms such that $\mathcal{L} \subseteq \mathcal{L}_\mathcal{A}$ and $\text{assert}(r) \in \mathcal{L}_\mathcal{A}$ for every rule r over $\mathcal{L}_\mathcal{A}$.

Definition 2.25 (Evolving logic program)**.** An *evolving logic program over $\mathcal{L}$* is a (possibly infinite) set of rules over $\mathcal{L}_{\mathcal{A}}$.

Definition 2.26 (Event sequence)**.** An *event sequence over $\mathcal{L}$* is a sequence of evolving logic programs over $\mathcal{L}$.

Definition 2.27 (Evolution interpretation and its evolution trace)**.** An *evolution interpretation of length n* of an evolving program P over $\mathcal{L}$ is a finite sequence $\mathcal{I} = (I_1, I_2, \dots, I_n)$ of interpretations over $\mathcal{L}_{\mathcal{A}}$. The *evolution trace* associated with an evolution interpretation $\mathcal{I}$ of P is the sequence of programs $(P_1, P_2, \dots, P_n)$ where

$$P_1 = P \text{ and } P_{i+1} = \{r \mid \mathrm{assert}(r) \in I_i\} \text{ for all } i \in \{1, 2, \dots, n-1\} \quad .$$

Definition 2.28 (Evolution stable model given an event sequence)**.** An evolution interpretation $\mathcal{M} = (M_1, M_2, \dots, M_n)$ with evolution trace $(P_1, P_2, \dots, P_n)$ is an *evolution stable model of an evolving program P given an event sequence* $(E_1, E_2, \dots, E_n)$ iff for every $i \in \{1, 2, \dots, n\}$ M_i is a dynamic stable model of $(P_1, P_2, \dots, P_{i-1}, P_i \cup E_i)$.

Example 2.29. Consider the following evolving logic program:

$$P: \qquad \mathrm{write\_thesis} \leftarrow \mathbf{not}\ \mathrm{tired}. \tag{2.22}$$

$$\mathrm{drink\_coffee} \leftarrow \mathrm{tired}, \mathbf{not}\ \mathrm{no\_coffee}. \tag{2.23}$$

$$\mathrm{make\_coffee} \leftarrow \mathrm{tired}, \mathrm{no\_coffee}. \tag{2.24}$$

$$\mathrm{assert}(\mathrm{tired} \leftarrow .) \leftarrow \mathrm{write\_thesis}. \tag{2.25}$$

$$\mathrm{assert}(\mathbf{not}\ \mathrm{tired} \leftarrow .) \leftarrow \mathrm{drink\_coffee}. \tag{2.26}$$

P could be a 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 the fact $(\mathrm{no\_coffee} \leftarrow .)$ as an event in case she runs out of coffee.

The meaning of the rules is as follows: Rule (2.22) says Mary's writing the thesis as long as she's not tired. Rules (2.23) and (2.24) say what she does when she's tired – depending on whether she has coffee she either drinks it or makes some more. Rules (2.25) and (2.26) specify whether she will be tired in the next evolution step. If she's writing the thesis, she will get tired. In case she's drinking coffee, the tiredness will wear off. If she's making coffee, no change will take place.

The following table shows the evolution of P (given the sequence of events that are also present in the table):

Time	Program	Event	Model
1	P	$\emptyset$	$\{\text{write\_thesis}, \text{assert}(\text{tired} \leftarrow .)\}$
2	$\{\text{tired} \leftarrow .\}$	$\emptyset$	$\{\text{tired}, \text{drink\_coffee}, \text{assert}(\textbf{not}\ \text{tired} \leftarrow .)\}$
3	$\{\textbf{not}\ \text{tired} \leftarrow .\}$	$\{\text{no\_coffee} \leftarrow .\}$	$\{\text{no\_coffee}, \text{write\_thesis}, \text{assert}(\text{tired} \leftarrow .)\}$
4	$\{\text{tired} \leftarrow .\}$	$\{\text{no\_coffee} \leftarrow .\}$	$\{\text{tired}, \text{no\_coffee}, \text{make\_coffee}\}$
5	$\emptyset$	$\emptyset$	$\{\text{tired}, \text{drink\_coffee}, \text{assert}(\textbf{not}\ \text{tired} \leftarrow .)\}$
6	$\{\textbf{not}\ \text{tired} \leftarrow .\}$	$\emptyset$	$\{\text{write\_thesis}, \text{assert}(\text{tired} \leftarrow .)\}$

We start off with P and an empty event and compute the first model. It says Mary's writing her thesis, and in the next step she should get tired. We infer the second program from the model, add another empty event and compute the second model. Now Mary is tired and drinks coffee. In the next step the sensor starts complaining that there's no more coffee, but Mary doesn't really care. She's not tired, so she's writing the thesis. In the fourth step she's tired again, and, as there is still no coffee, she makes some. This makes the sensor stop complaining in the fifth step and Mary, still tired, drinks coffee again. In the sixth step she continues writing her thesis again...

For more examples see [19, 20, 21].

Chapter 3

Transformational Semantics for EVOLP

A quick and convenient way of implementing some new language is to transform the input program (written in that language) into an equivalent program written in some already implemented language. As for declarative languages, it is much easier to formally define such transformations and prove they are sound and complete.

This Chapter introduces a similar transformation for **EVOLP**. It takes an evolving logic program and a sequence of events as input and outputs an equivalent normal logic program. In Sect. 3.1 we define the transformation and explain how it works. Sections 3.2 and 3.3 contain proofs of soundness and completeness of the transformation. In Sect. 3.4 we infer the lower and upper bounds for the size of the transformed program and deduce some of its implications.

3.1 Definition

In this Section 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 **DLP**s introduced in [17] and illustrated in Ex. 2.23. First we need to define the extended language over which we will construct the resulting program:

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

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 over $\mathcal{L}_{\mathcal{A}}$, see Def. 2.27) into just one interpretation over $\mathcal{L}_{\mathcal{T}}$. 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 and the following sections, we'll use the following conventions: Let L be a literal over $\mathcal{L}_{\mathcal{A}}$, B a set of literals over $\mathcal{L}_{\mathcal{A}}$ 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.
- $B^j = \{L^j \mid L \in B\}$.
- We will say that L *trans-appears* in $P_{\mathcal{E}}$ iff L^j or L^j_{neg} appears in $P_{\mathcal{E}}$.

Definition 3.1. 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 $\mathcal{L}_{\mathcal{T}}$, 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
$$L^j \leftarrow \mathit{body}^j, \mathbf{not}\, \mathrm{rej}(L^j, 1).$$

2. **Rewritten event rules.** For every rule $(L \leftarrow \mathit{body}.) \in E_j$ it contains the rule
$$L^j \leftarrow \mathit{body}^j, \mathbf{not}\, \mathrm{rej}(L^j, j).$$

3. **Assertable rules.** For every rule $r = (L \leftarrow \mathit{body}.)$ over $\mathcal{L}_\mathcal{A}$ and all i, $1 < i \leq j$, such that $(\mathrm{assert}(r))^{i-1}$ is in the head of some rule of $P_\mathcal{E}^{i-1}$ it contains the rule
$$L^j \leftarrow \mathit{body}^j, (\mathrm{assert}(r))^{i-1}, \mathbf{not}\, \mathrm{rej}(L^j, i).$$

4. **Default assumptions.** For every atom $A \in \mathcal{L}_\mathcal{A}$ such that A^j or A^j_{neg} appears in some rule of $P_\mathcal{E}^j$ (from the previous groups of rules) it also contains the rule
$$A^j_{\mathrm{neg}} \leftarrow \mathbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, 0).$$

5. **Rejection rules.** For every rule of $P_\mathcal{E}^j$ of the form
$$L^j \leftarrow \mathit{body}, \mathbf{not}\, \mathrm{rej}(L^j, i).[1] \tag{3.1}$$
it also contains the rules
$$\mathrm{rej}(L^j_{\mathrm{neg}}, p) \leftarrow \mathit{body}. \tag{3.2}$$
$$\mathrm{rej}(L^j, q) \leftarrow \mathrm{rej}(L^j, i). \tag{3.3}$$
where:

   (a) $p \leq i$ is the largest index such that $P_\mathcal{E}^j$ contains a rule with the literal $\mathbf{not}\, \mathrm{rej}(L^j_{\mathrm{neg}}, p)$ in its body. If no such p exists, then the rule (3.2) is not in $P_\mathcal{E}^j$.

[1]The set *body* contains literals from the original body of the rule in translated form and in case it is an assertable rule it also contains a literal of the form $(\mathrm{assert}(r))^{i-1}$ – later we will call this literal the *assertion guard* of the rule.

(b) $q < i$ is the largest index such that $P_{\mathcal{E}}^j$ contains a rule with the literal $\textbf{not}\,\text{rej}(L^j, q)$ in its body. If no such q exists, then the rule (3.3) is not in $P_{\mathcal{E}}^j$.

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

$$A^j \leftarrow body_p, \textbf{not}\,\text{rej}(A^j, i).$$
$$A^j_{\text{neg}} \leftarrow body_n, \textbf{not}\,\text{rej}(A^j_{\text{neg}}, i).$$

it also contains the constraint

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

Each $P_{\mathcal{E}}^j$ contains rules for simulating the "j-th **DLP**" from the definition of evolution stable model (Def. 2.28). For the simulation we use the transformational semantics from [17]. We also rewrite all atoms from the original rules as a new set of j-indexed atoms. The **DLP** looks like this:

$$(P, P_2, P_3, \ldots, P_{j-1}, P_j \cup E_j) \ .$$

However, we don't know the exact contents of $P_2, P_3, \ldots, P_j$. What we know are the rules in P and E_j. The first two groups of rules in $P_{\mathcal{E}}^j$ (rewritten program rules and rewritten event rules) contain their rewritten forms.

The group of assertable rules contains all rules that can possibly occur in $P_2, P_3, \ldots, P_j$. Each of these rules also has an atom of the form $(\text{assert}(r))^{i-1}$ in its body. This is its *assertion guard*, and it assures the rule is only used in case it was actually asserted. Assertion guards are also the only connection between the rules of $P_{\mathcal{E}}^j$ and the rules in $P_{\mathcal{E}}^1 \cup P_{\mathcal{E}}^2 \cup \ldots \cup P_{\mathcal{E}}^{j-1}$.

The default assumptions are defined similarly as in [17], and they have the same function – they simulate the set of defaults (Def. 2.19).

Rewritten program rules, rewritten event rules, assertable rules and default assumptions also contain a default literal of the form $\textbf{not}\,\text{rej}(L^j, i)$ in their bodies – we will call this literal the *rejection guard of the rule* and i the *level of the rule*. Together with the rejection rules, the rejection guard provides a means of rejecting a rule by a higher level rule, similarly as in the set of rejected rules defined in Def. 2.20. A body of a rewritten program rule, rewritten event rule or an assertable rule without the assertion and rejection guards is called its *guardless body*.

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 dynamic stable model (more details can be found in [7]) and totality constraints are needed to eliminate the superfluous stable models of $P_\mathcal{E}$ originating from such situations.

In the following two Sections we will prove the defined transformation is sound and complete.

3.2 Soundness

In this Section we will prove the defined transformation is sound, i.e. every stable model of the transformed program corresponds to an evolution stable model of the original evolving logic program. Therefore, we will assume P is an evolving logic program, $\mathcal{E} = (E_1, E_2, \dots, E_n)$ is an event sequence, N is a stable model of $P_\mathcal{E}$,

$$M_i = \{A \in \mathcal{L}_\mathcal{A} \mid A^i \in N\} \text{ for all } i \in \{1, 2, \dots, n\} \tag{3.4}$$

and $(P_1, P_2, \dots, P_n)$ is the evolution trace associated to the evolution interpretation $(M_1, M_2, \dots, M_n)$, i.e.

$$P_1 = P \text{ and } P_{i+1} = \{r \mid \mathrm{assert}(r) \in M_i\} \text{ for all } i \in \{1, 2, \dots, n-1\} \quad .$$

Using this notation, formulation of the soundness property boils down to: $(M_1, M_2, \ldots, M_n)$ is an evolution stable model of P given $\mathcal{E}$. According to the definition of evolution stable model (Def. 2.28), this holds iff each 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 $j \in \{1, 2, \ldots, n\}$, prove that M_j is a dynamic stable model of $\mathcal{P} = (P_1, P_2, \ldots, P_{j-1}, P_j \cup E_j)$ and the property will follow.

We will need a number of auxiliary propositions to prove this. To make their formulation (and also the formulation of their proofs) simpler and more comprehensible, we will use the notation introduced in Sect. 3.1 and in the previous paragraphs and also the following definitions:

- *Rewritten rules* are all rewritten program rules, all rewritten event rules and all assertable rules such that their assertion guard is true in N.

- A rewritten rule is *unrejected* iff its rejection guard is true in N.

- As N is a stable model of $P_{\mathcal{E}}$, we can use Def. 2.16 and Thm. 2.12 to obtain:
$$N^* = \bigcup_{i<\omega} N_i$$
where $N_0 = \emptyset$ and $N_{i+1} = T_{P_{\mathcal{E}} \cup N^-}(N_i)$ for all $i \geq 0$.

- We can use Thm. 2.12 once again to get
$$\text{least}([\rho(\mathcal{P}) \setminus \text{Rej}(\mathcal{P}, M_j)] \cup \text{Def}(\mathcal{P}, M_j)) = R = \bigcup_{i<\omega} R_i$$
where $R_0 = \emptyset$ and $R_{i+1} = T_{[\rho(\mathcal{P}) \setminus \text{Rej}(\mathcal{P}, M_j)] \cup \text{Def}(\mathcal{P}, M_j)}(R_i)$ for all $i \geq 0$.

According to Def. 2.21, M_j is a dynamic stable model of $\mathcal{P}$ iff $M_j^* = R$. This equality will be proved in Lemma 3.9. But first we will show some basic properties of N in Lemmas 3.2 and 3.3 and introduce a relation between N and M_j in Lemma 3.4. Then we will show how rewritten rules correspond to rules of $\mathcal{P}$ in Lemmas 3.5 and 3.6. Lemma 3.7 will reveal a correspondence between default assumptions and the set of defaults. Lemma 3.8 shows that j-indexed literals in N correspond to literals in R.

Lemma 3.2. Let L be a literal over $\mathcal{L}_\mathcal{A}$ and $k \in \{0, 1, \ldots, j\}$. $\mathrm{rej}(L^j, k) \in N$ holds iff some $i \in \{k, k+1, \ldots, j\}$ exists such that $P_\mathcal{E}$ contains rules of the form [2]

$$L^j \leftarrow \mathit{body}_p, \textbf{not}\, \mathrm{rej}(L^j, k).$$
$$L^j_{\mathrm{neg}} \leftarrow \mathit{body}_n, \textbf{not}\, \mathrm{rej}(L^j_{\mathrm{neg}}, i).$$

and $N \models \mathit{body}_n$.

Proof. First let's assume $\mathrm{rej}(L^j, k) \in N$. Then from Prop. 2.17 it follows that $P_\mathcal{E}$ contains some rule

$$r = (\mathrm{rej}(L^j, k) \leftarrow \mathit{body}.)$$

such that $N \models \mathit{body}$. This must be one of the two kinds of rejection rules:

1. If r is of the form (3.2), then a look at the definition of rejection rules tells us that the rules we search for must also be in $P_\mathcal{E}$.

2. If r is of the form (3.3), then body is of the form $\mathrm{rej}(L^j, k_1)$ where $k < k_1 \leq j$. As body is true in N, we can use Prop. 2.17 again to find a rule of $P_\mathcal{E}$ of the form
$$\mathrm{rej}(L^j, k_1) \leftarrow \mathit{body}_1.$$
such that $N \models \mathit{body}_1$. Two cases are possible again. In the first the proof ends and in the second we get an index k_2 such that $k_1 < k_2 \leq j$ and $\mathrm{rej}(L^j, k_2) \in N$. If the second case would occur forever, we would get an infinite increasing bounded sequence of natural numbers $k < k_1 < k_2 < \ldots \leq j$, which is not possible. Hence after a finite number of iterations the first case must occur, and the proof ends.

Now to the converse implication. Let $k = k_1 < k_2 < \ldots < k_s \leq i$ be all indices such that $P_\mathcal{E}$ contains a rule of the form

$$L^j \leftarrow \mathit{body}_t, \textbf{not}\, \mathrm{rej}(L^j, k_t).$$

[2]Similarly as before, the set body_p contains literals from the original body of the rule in translated form, and in case it is an assertable rule, it also contains its assertion guard. The same holds for the set body_n.

where $t \in \{1, 2, \dots, s\}$. From the definition of rejection rules (Def. 3.1) we have that $P_{\mathcal{E}}$ contains the rule

$$\mathrm{rej}(L^j, k_s) \leftarrow body_n.$$

and also the rules

$$rej(L_j, k_t) \leftarrow rej(L_j, k_{t+1}).$$

for all $t \in \{1, 2, \dots, s-1\}$. The claim now follows by s times applying Prop. 2.17. □

Lemma 3.3. The following holds for every atom $A \in \mathcal{L}_{\mathcal{A}}$ such that it trans-appears in $P_{\mathcal{E}}$:

$$A^j \in N \iff A^j_{\mathrm{neg}} \notin N \ .$$

Proof. First let's assume $A^j \in N$. Proposition 2.17 implies that $P_{\mathcal{E}}$ contains some rule of the form

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

such that $N \models body_p$ and $\mathrm{rej}(A^j, i) \notin N$. If r is any rule of $P_{\mathcal{E}}$ with A^j_{neg} in its head, then it must be of the form

$$A^j_{\mathrm{neg}} \leftarrow body_n, \mathbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, k).$$

We will consider two cases:

a) If $k \le i$, then from Lemma 3.2 we have that $\mathrm{rej}(A^j_{\mathrm{neg}}, k) \in N$.

b) If $k > i$, then, as $\mathrm{rej}(A^j, i) \notin N$, we can use Lemma 3.2 to get $N \not\models body_n$.

Both cases imply that the body of r is false in N. Hence there is no rule of $P_{\mathcal{E}}$ with A^j_{neg} in its head and a body true in N and from Prop. 2.17 it follows that $A^j_{\mathrm{neg}} \notin N$.

We will prove the converse implication by contradiction. Suppose $A^j \notin N$ and at the same time $A^j_{\mathrm{neg}} \notin N$. Let s be the highest index such that $P_{\mathcal{E}}$

contains a rule

$$A^j \leftarrow body_s, \textbf{not}\, \mathrm{rej}(A^j, s).$$

such that $N \models body_s$. If no such rule is in $P_\mathcal{E}$, let $s = -1$. Similarly let t be the highest index such that $P_\mathcal{E}$ contains a rule

$$A^j_{\mathrm{neg}} \leftarrow body_t, \textbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, t).$$

such that $N \models body_t$. As A trans-appears in $P_\mathcal{E}$, there is always at least one such rule between the default assumptions, so t is defined in all cases and $t \geq 0$. Let's consider two situations again:

a) $s \geq t$: If $\mathrm{rej}(A^j, s) \notin N$, then A^j would be (by Prop. 2.17) in N. So $\mathrm{rej}(A^j, s) \in N$ and Lemma 3.2 implies that $P_\mathcal{E}$ contains a rule

$$A^j_{\mathrm{neg}} \leftarrow body_u, \textbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, u).$$

such that $N \models body_u$ and $u \geq s$. According to the way t was constructed we also have $u \leq t \leq s$. Thus $u = s$. But then it can't be that $A^j, A^j_{\mathrm{neg}} \notin N$, because $P_\mathcal{E}$ contains a totality constraint forbidding this – a contradiction with the assumptions.

b) $s < t$: If $\mathrm{rej}(A^j_{\mathrm{neg}}, t) \notin N$, then A^j_{neg} would be (by Prop. 2.17) in N. So $\mathrm{rej}(A^j_{\mathrm{neg}}, t) \in N$ and from Lemma 3.2 we get that $P_\mathcal{E}$ contains a rule

$$A^j \leftarrow body_v, \textbf{not}\, \mathrm{rej}(A^j, v).$$

such that $N \models body_v$ and $v \geq t$. According to the way s was constructed we also have $v \leq s < t$, which is a contradiction. □

Lemma 3.4. Let B be a set of literals over $\mathcal{L}_\mathcal{A}$ which trans-appear in $P_\mathcal{E}$. Then

$$M_j \models B \iff N \models B^j \enspace .$$

Proof. Let $L \in B$. If L is an atom A, then

$$M_j \models L \iff A \in M_j \overset{(3.4)}{\iff} A^j \in N \iff N \models L^j \enspace .$$

If L is a default literal **not** A, then

$$M_j \models L \iff A \notin M_j \overset{(3.4)}{\iff} A^j \notin N \overset{\text{Lemma 3.3}}{\iff} A^j_{\text{neg}} \in N \iff N \models L^j \ . \ \square$$

Lemma 3.5. $P_{\mathcal{E}}$ contains a rewritten rule of level i with L^j in its head and a guardless body $body^j$ iff

$$(L \leftarrow body.) \in \mathcal{P}^i \ .$$

Proof. Let $P_{\mathcal{E}}$ contain a rewritten rule r^* of level i with L^j in its head and a guardless body $body^j$. Let $r = (L \leftarrow body.)$. We will consider three cases:

1. If r^* is a rewritten program rule, then by Def. 3.1 we have $r \in \mathcal{P}^1$.
2. If r^* is a rewritten event rule, then by Def. 3.1 we have $r \in \mathcal{P}^j$.
3. If r^* is an assertable rule, then $i \in \{2, 3, \dots, j\}$ and its assertion guard $(\text{assert}(r))^{i-1}$ is true in N. By (3.4) we have $\text{assert}(r) \in M_{i-1}$, and thus $r \in \mathcal{P}^i$.

For the converse implication let $r = (L \leftarrow body.) \in \mathcal{P}^i$ for some $i \in \{1, 2, \dots, j\}$. We will consider three cases again:

1. If $r \in P$, then $P_{\mathcal{E}}$ contains a rewritten program rule with L^j in its head and the guardless body $body^j$ by the definition.
2. If $r \in E_j$, then $P_{\mathcal{E}}$ contains a rewritten event rule with L^j in its head and the guardless body $body^j$ by the definition.
3. If $r \in P_i$ and $i > 1$, then $\text{assert}(r) \in M_{i-1}$ and by (3.4) we have $(\text{assert}(r))^{i-1} \in N$. Hence by Prop. 2.17 $P_{\mathcal{E}}$ must contain a rule with $(\text{assert}(r))^{i-1}$ in its head and by Def. 3.1 it must also contain the assertable rule

$$L^j \leftarrow body^j, (\text{assert}(r))^{i-1}, \textbf{not}\, \text{rej}(L^j, i). \qquad \square$$

Lemma 3.6. $P_{\mathcal{E}}$ contains an unrejected rewritten rule with L^j in its head and a guardless body $body^j$ iff

$$(L \leftarrow body.) \in \rho(\mathcal{P}) \setminus \mathrm{Rej}(\mathcal{P}, M_j) \enspace .$$

Proof. Let $P_{\mathcal{E}}$ contain an unrejected rewritten rule of level i with L^j in its head and a guardless body $body^j$. Then by Lemma 3.5 we have

$$r = (L \leftarrow body.) \in \mathcal{P}^i \enspace .$$

We will continue by contradiction – let's assume $r \in \mathrm{Rej}^i(\mathcal{P}, M_j)$. Then some $k \in \{i, i+1, \ldots, j\}$ exists such that $\mathcal{P}^k$ contains a rule

$$\mathbf{not}\, L \leftarrow body_n.$$

and $M_j \models body_n$. Then by Lemma 3.4 we have $N \models body_n^j$ and by Lemma 3.5 we have that $P_{\mathcal{E}}$ contains a rewritten rule with L^j_{neg} in its head and the guardless body $body_n^j$. Hence from Lemma 3.2 it follows that $\mathrm{rej}(L^j, i) \in N$, which is a contradiction with the assumption that r is unrejected. Therefore $r \in \mathcal{P}^i \setminus \mathrm{Rej}^i(\mathcal{P}, M_j)$.

For the converse implication let $r = (L \leftarrow body.) \in \mathcal{P}^i \setminus \mathrm{Rej}^i(\mathcal{P}, M_j)$ for some $i \in \{1, 2, \ldots, j\}$. By Lemma 3.5 we have that $P_{\mathcal{E}}$ contains a rewritten rule r of level i with L^j in its head and the guardless body $body^j$. We need to prove that r is unrejected. Assume it is rejected, i.e. $N \not\models \mathbf{not}\, \mathrm{rej}(L^j, i)$. Then $\mathrm{rej}(L^j, i) \in N$ and from Lemmas 3.2, 3.5 and 3.4 it follows that $r \in \mathrm{Rej}^i(\mathcal{P}, M_j)$ which is a contradiction with the assumption. Hence r is unrejected. □

Lemma 3.7. Let A be an atom of $\mathcal{L}_{\mathcal{A}}$ that trans-appears in $P_{\mathcal{E}}$. Then

$$\mathbf{not}\, A \in \mathrm{Def}(\mathcal{P}, M_j) \Longleftrightarrow \mathbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, 0) \in N^- \enspace .$$

Proof. According to the definition of default assumptions, $P_{\mathcal{E}}$ must contain a default assumption of the form

$$A^j_{\mathrm{neg}} \leftarrow \textbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, 0).$$

because A trans-appears in $P_{\mathcal{E}}$. We will prove both implications indirectly.

First let's assume $\textbf{not}\, A \notin \mathrm{Def}(\mathcal{P}, M_j)$. Then some rule $r = (A \leftarrow \mathit{body}.) \in \rho(\mathcal{P})$ exists such that $M_j \models \mathit{body}$. $P_{\mathcal{E}}$ contains a corresponding rewritten rule r^* (Lemma 3.5) and we also have $N \models \mathit{body}^j$ (Lemma 3.4). Now we can use Lemma 3.2, the existence of r^* and of the default assumption mentioned earlier to conclude that $\mathrm{rej}(A^j_{\mathrm{neg}}, 0) \in N$. Hence $\textbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, 0) \notin N^-$.

For the converse implication let's assume $\textbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, 0) \notin N^-$. Then $\mathrm{rej}(A^j_{\mathrm{neg}}, 0) \in N$ and we can use Lemma 3.2 to find a rule r of $P_{\mathcal{E}}$ of the form

$$A^j \leftarrow \mathit{body}, \mathrm{rej}(A^j, i).$$

such that $N \models \mathit{body}$. $\mathcal{P}^i$ must contain a corresponding rule with A in its head and its body true in M_j (Lemmas 3.5 and 3.4). Hence $\textbf{not}\, A \notin \mathrm{Def}(\mathcal{P}, M_j)$. □

Lemma 3.8. For every literal L over $\mathcal{L}_{\mathcal{A}}$ such that it trans-appears in $P_{\mathcal{E}}$ the following holds:

$$L^j \in N \iff L \in R \ .$$

Proof. In order to prove the first implication, we will prove

$$(\forall i \in \mathbb{N})(L^j \in N_i \implies L \in R_i)$$

by induction on i:

1° $N_0 = \emptyset$, so the claim holds.

2° We assume that

$$L^j \in N_i \implies L \in R_i$$

holds for all L trans-appearing in $P_{\mathcal{E}}$ and prove

$$L^j \in N_{i+1} \Longrightarrow L \in R_{i+1} \quad .$$

If $L^j \in N_{i+1}$, then $P_{\mathcal{E}}$ must contain some rule $r^* = (L^j \leftarrow \mathit{body}.)$ such that $\mathit{body} \subseteq N_i$. This means that all guards in the body of r^* are true in N (because $N_i \subseteq N$). If r^* is a default assumption, then by Lemma 3.7 we get $L \in \mathrm{Def}(\mathcal{P}, M_j) \subseteq R_{i+1}$. Otherwise it must be an unrejected rewritten rule, and, according to Lemma 3.6, the corresponding rule $r \in \rho(\mathcal{P}) \setminus \mathrm{Rej}(\mathcal{P}, M_j)$ has L in its head. Moreover, by the induction hypothesis we have that the body of r is a subset of R_i, so $L \in R_{i+1}$.

By another induction on i we will prove

$$(\forall i \in \mathbb{N})(L \in R_i \Longrightarrow (\exists k \in \mathbb{N})(L^j \in N_k))$$

from which the converse implication follows.

1° $R_0 = \emptyset$, so the claim holds.

2° We assume that

$$L \in R_i \Longrightarrow (\exists k \in \mathbb{N})(L^j \in N_k)$$

holds for all L that trans-appear in $P_{\mathcal{E}}$ and prove

$$L \in R_{i+1} \Longrightarrow (\exists k \in \mathbb{N})(L^j \in N_k) \quad .$$

If $L \in R_{i+1}$, then $[\rho(\mathcal{P}) \setminus \mathrm{Rej}(\mathcal{P}, M_j)] \cup \mathrm{Def}(\mathcal{P}, M_j)$ must contain some rule $r = (L \leftarrow \mathit{body}.)$ such that $\mathit{body} \subseteq R_i$. Two situations are possible:

(a) If $L \in \mathrm{Def}(\mathcal{P}, M_j)$, then by Lemma 3.7 we have $\mathbf{not}\, \mathrm{rej}(L^j, 0) \in N^- \subseteq N_1$. So the body of the default assumption for L^j is satisfied in N_1 and $L^j \in N_2$.

(b) If $r \in \mathcal{P}^i \setminus \mathrm{Rej}^i(\mathcal{P}, M_j)$ for some $i \in \{1, 2, \ldots, n\}$, then $P_{\mathcal{E}}$ contains an unrejected rewritten rule r^* with L^j in its head and the guardless body body^j (Lemma 3.6). For each literal $X \in \mathit{body}$ we can use the inductive assumption to find a natural number k_X such that $X^j \in N_{k_X}$. Let $k_1 = \max\{k_X \mid X \in \mathit{body}\}$.

If the rule r^* has an assertion guard in its body, then it must be true in N because r^* is a rewritten rule. Hence some $k_2 \in \mathbb{N}$ must exist such that the assertion guard is true in N_{k_2}. If it has no such guard, let $k_2 = 1$.

Now let $k = \max\{k_1, k_2\}$. The rejection guard of r^* is true in $N^- \subseteq N_k$ because r^* is unrejected. Hence from the monotonicity of the immediate consequence operator (Prop. 2.11) it follows that the whole body of r^* is true in N_k. Thus $L^j \in N_{k+1}$. □

Lemma 3.9. M_j is a dynamic stable model of $\mathcal{P}$.

Proof. M_j is a dynamic stable model of the dynamic logic program $\mathcal{P}$ iff $M_j^* = R$. Let L be a literal over $\mathcal{L}_{\mathcal{A}}$. Two situations are possible:

1. If L doesn't trans-appear in $P_{\mathcal{E}}$, then it cannot appear in $\rho(\mathcal{P})$ (if it appeared in a rule $r \in \rho(\mathcal{P})$, then it would trans-appear in the rule of $P_{\mathcal{E}}$ corresponding to r). So $L \in \{A, \textbf{not } A\}$ and $\textbf{not } A \in \mathrm{Def}(\mathcal{P}, M_j)$ and hence $\textbf{not } A \in R$. We also have $A \notin R$ as the program $[\rho(\mathcal{P}) \setminus \mathrm{Rej}(\mathcal{P}, M_j)] \cup \mathrm{Def}(\mathcal{P}, M_j)$ doesn't contain any rule with A in its head. Furthermore, A^j doesn't appear in $P_{\mathcal{E}}$, so by Prop. 2.17 we have $A^j \notin N$. Hence $A \notin M_j^*$ and also $\textbf{not } A \in M_j^*$. Taken all together, we proved
$$L \in M_j^* \iff L \in R \enspace .$$

2. If L trans-appears in $P_{\mathcal{E}}$, then:
$$L \in M_j^* \iff M_j \models L \overset{\text{Lemma 3.4}}{\iff} N \models L^j \iff L^j \in N$$
$$\overset{\text{Lemma 3.8}}{\iff} L \in R \enspace . \qquad \square$$

Theorem 3.10 (Soundness)**.** $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ is an evolution stable model of P given $\mathcal{E}$.

Proof. Follows by applying Lemma 3.9 for each $j \in \{1, 2, \ldots, n\}$. □

3.3 Completeness

In this Section we will prove the defined transformation is complete, i.e. every evolution stable model of the original evolving logic program corresponds to a stable model of the transformed program. Therefore, we will assume P is an evolving logic program, $\mathcal{E} = (E_1, E_2, \dots, E_n)$ is an event sequence, $\mathcal{M} = (M_1, M_2, \dots, M_n)$ is an evolution stable model of P given $\mathcal{E}$, $(P_1, P_2, \dots, P_n)$ is the evolution trace associated to $\mathcal{M}$ and

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

In order to prove that the transformation is complete, we have to find a stable model of $P_{\mathcal{E}}$ corresponding to $\mathcal{M}$. First let's define an interpretation N corresponding to $\mathcal{M}$:

$$\begin{aligned} N = &\{L^i \mid i \in \{1, 2, \dots, n\} \wedge M_i \models L \wedge L \text{ trans-appears in } P_{\mathcal{E}}\} \\ &\cup \{\mathrm{rej}(L^i, k) \mid 1 \leq k \leq i \leq n \wedge (\exists r \in \mathrm{Rej}^k(\mathcal{P}_i, M_i))(H(r) = L)\} \\ &\cup \{\mathrm{rej}(A^i_{\mathrm{neg}}, 0) \mid i \in \{1, 2, \dots, n\} \wedge \textbf{not } A \notin \mathrm{Def}(\mathcal{P}_i, M_i)\} \quad . \end{aligned} \tag{3.5}$$

The rest of this section is devoted to proving that N is a stable model of $P_{\mathcal{E}}$. The following definitions will be used throughout the proofs[3]:

- A *rewritten rule* is every rewritten program rule, rewritten event rule and every assertable rule of $P_{\mathcal{E}}$ with its assertion guard true in R.
- A rewritten rule is *unrejected* iff its rejection guard is true in R.
- Let $j \in \{1, 2, \dots, n\}$. As M_j is a dynamic stable model of $\mathcal{P}_j$, we can use Def. 2.21 and Thm. 2.12 to get

$$M_j^* = \bigcup_{i<\omega} M_{j,i}$$

where $M_{j,0} = \emptyset$ and $M_{j,i+1} = T_{[\rho(\mathcal{P}_j) \setminus \mathrm{Rej}(\mathcal{P}_j, M_j)] \cup \mathrm{Def}(\mathcal{P}_j, M_j)}(M_{j,i})$.

[3]please note that a part of this notation was already defined in Sect. 3.2, but those definitions were only valid for that Section and the following definitions are slightly different

- According to Thm. 2.12 we also have

$$\text{least}\big(P_{\mathcal{E}} \cup N^-\big) = R = \bigcup_{i<\omega} R_i$$

where $R_0 = \emptyset$ and $R_{i+1} = T_{P_{\mathcal{E}} \cup N^-}(R_i)$ for all $i \geq 0$.

According to Def. 2.16 N is a stable model of $P_{\mathcal{E}}$ iff

$$N^* = R \ .$$

We will prove this equality in three steps:

1. The first step is also the most difficult. For every literal L over $\mathcal{L}_{\mathcal{A}}$ and every $j \in \{1, 2, \dots, n\}$ we will prove

$$L^j \in N \iff L^j \in R$$

by complete induction on j. The inductive hypothesis will be used in many places, so it is formulated here once and for all: Let $j \in \{1, 2, \dots, n\}$. Then for every literal L over $\mathcal{L}_{\mathcal{A}}$ the following holds:

$$(\forall i \in \{1, 2, \dots, j-1\})(L^i \in N \iff L^i \in R) \ . \tag{3.6}$$

2. Next we will prove that for every literal L over $\mathcal{L}_{\mathcal{A}}$, every $j \in \{1, 2, \dots, n\}$ and every $i \in \{0, 1, \dots, j\}$ it holds that

$$\text{rej}(L^j, i) \in N \iff \text{rej}(L^j, i) \in R \ .$$

3. The last thing we have to take care of is that none of the totality constraints are broken, i.e. we have to prove

$$u \notin R \ .$$

The preparation phase is now over, we can step forward to the proofs. The first 5 lemmas are a prelude to the proof of the first step.

Lemma 3.11. Assume (3.6) holds. $P_{\mathcal{E}}$ contains a rewritten rule of level i with L^j in its head and a guardless body $body^j$ iff

$$(L \leftarrow body.) \in \mathcal{P}_j^i \enspace .$$

Proof. Let $P_{\mathcal{E}}$ contain a rewritten rule r^* of level i with L^j in its head and a guardless body $body^j$. Let $r = (L \leftarrow body.)$. We will consider three cases:

1. If r^* is a rewritten program rule, then by Def. 3.1 we have $r \in \mathcal{P}_j^1$.
2. If r^* is a rewritten event rule, then by Def. 3.1 we have $r \in \mathcal{P}_j^j$.
3. If r^* is an assertable rule, then $i \in \{2, 3, \ldots, j\}$ and its assertion guard $(\mathrm{assert}(r))^{i-1}$ is true in R. Then by (3.6) we have $(\mathrm{assert}(r))^{i-1} \in N$ and by (3.5) we have $\mathrm{assert}(r) \in M_{i-1}$. Hence $r \in \mathcal{P}_j^i$.

For the converse implication let $r = (L \leftarrow body.) \in \mathcal{P}_j^i$ for some $i \in \{1, 2, \ldots, j\}$. We will consider three cases:

1. If $r \in P$, then $P_{\mathcal{E}}$ contains a rewritten program rule of level i with L^j in its head and the guardless body $body^j$ by the definition.
2. If $r' \in E_j$, then $P_{\mathcal{E}}$ contains a rewritten program rule of level i with L^j in its head and the guardless body $body^j$ by the definition.
3. If $r \in P_i$ and $i > 1$, then $\mathrm{assert}(r) \in M_{i-1}$ and by (3.5) we have $(\mathrm{assert}(r))^{i-1} \in N$. By (3.6) we get $(\mathrm{assert}(r))^{i-1} \in R$. Hence $P_{\mathcal{E}}$ must contain a rule with $(\mathrm{assert}(r))^{i-1}$ in its head and by Def. 3.1 it must also contain the assertable rule

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

Lemma 3.12. Assume (3.6) holds. $P_{\mathcal{E}}$ contains an unrejected rewritten rule with L^j in its head and a guardless body body^j iff

$$(L \leftarrow \mathit{body}.) \in \rho(\mathcal{P}_j) \setminus \mathrm{Rej}(\mathcal{P}_j, M_j) \enspace .$$

Proof. Let $P_{\mathcal{E}}$ contain an unrejected rewritten rule r^* of level i with L^j in its head and a guardless body body^j. Then by Lemma 3.11 we have

$$r = (L \leftarrow \mathit{body}.) \in \mathcal{P}_j^i \enspace .$$

It is left to prove that r is not a member of the set $\mathrm{Rej}^i(\mathcal{P}_j, M_j)$. As r^* is unrejected, we know that $\mathbf{not}\, \mathrm{rej}(L^j, i) \in R$. The only source of default literals in R is N^-, so $\mathbf{not}\, \mathrm{rej}(L^j, i) \in N^-$. Therefore $\mathrm{rej}(L^j, i) \notin N$ and by (3.5) we have that no rule of $\mathrm{Rej}^i(\mathcal{P}_j, M_j)$ has L in its head. Thus $r \notin \mathrm{Rej}^i(\mathcal{P}_j, M_j)$.

For the converse implication let $r = (L \leftarrow \mathit{body}.) \in \mathcal{P}_j^i \setminus \mathrm{Rej}^i(\mathcal{P}_j, M_j)$ for some $i \in \{1, 2, \ldots, j\}$. By Lemma 3.11 we have that $P_{\mathcal{E}}$ contains a rewritten rule r^* with L^j in its head and the guardless body body^j. Now we need to prove that r is unrejected. Assume it is rejected, i.e. $\mathbf{not}\, \mathrm{rej}(L^j, i) \notin R$. Then $\mathbf{not}\, \mathrm{rej}(L^j, i) \notin N^-$ and thus $\mathrm{rej}(L^j, i) \in N$. Then by (3.5) we have that $\mathrm{Rej}^i(\mathcal{P}_j, M_j)$ contains some rule with L in its head. Moreover, according to the definition of $\mathrm{Rej}^i(\mathcal{P}_j, M_j)$, some $k \in \{i, i+1, \ldots, j\}$ must exist such that $\mathcal{P}_j^k$ contains a rule with $\mathbf{not}\, L$ in its head and its body satisfied in M_j. But according to the very same definition, this implies $r \in \mathrm{Rej}^i(\mathcal{P}_j, M_j)$, a contradiction with the assumption. $\square$

Lemma 3.13. Let A be an atom of $\mathcal{L}_{\mathcal{A}}$ that trans-appears in $P_{\mathcal{E}}$. Then

$$\mathbf{not}\, A \in \mathrm{Def}(\mathcal{P}_j, M_j) \iff \mathbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, 0) \in R_1 \enspace .$$

Proof. Assume $\mathbf{not}\, A \in \mathrm{Def}(\mathcal{P}_j, M_j)$. Then according to (3.5) we have $\mathrm{rej}(A^j_{\mathrm{neg}}, 0) \notin N$. This implies $\mathbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, 0) \in N^- \subseteq R_1$.

On the other hand, if $\mathbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, 0) \in R_1$, then it must be the case that $\mathbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, 0) \in N^-$. But this implies $\mathrm{rej}(A^j_{\mathrm{neg}}, 0) \notin N$ and by (3.5) we get $\mathbf{not}\, A \in \mathrm{Def}(\mathcal{P}_j, M_j)$. $\square$

Lemma 3.14. Let L be a literal over $\mathcal{L}_{\mathcal{A}}$. If (3.6) holds, then

$$(\forall i \in \mathbb{N})(L \in M_{j,i} \wedge L \text{ trans-appears in } P_{\mathcal{E}} \Longrightarrow (\exists l \in \mathbb{N})(L^j \in R_l)) \ .$$

Proof. We will prove by induction on i:

1° For $i = 0$ the claim trivially follows from $M_{j,0} = \emptyset$.

2° We assume the claim holds for i, i.e. if L is a literal over $\mathcal{L}_{\mathcal{A}}$, then

$$L \in M_{j,i} \wedge L \text{ trans-appears in } P_{\mathcal{E}} \Longrightarrow (\exists l \in \mathbb{N})(L^j \in R_l) \ .$$

We will prove the claim for $i + 1$. So let's assume $L \in M_{j,i+1}$ and L trans-appears in $P_{\mathcal{E}}$. Then there is some rule $r = (L \leftarrow \mathit{body}.) \in [\rho(\mathcal{P}_j) \setminus \mathrm{Rej}(\mathcal{P}_j, M_j)] \cup \mathrm{Def}(\mathcal{P}_j, M_j)$ such that $\mathit{body} \subseteq M_{j,i}$. Let's consider two cases:

(a) If r is a default assumption, then *body* is empty and L is a default literal **not** A. $P_{\mathcal{E}}$ must also contain a default assumption of the form

$$A^j_{\mathrm{neg}} \leftarrow \textbf{not}\, \mathrm{rej}(A^j_{\mathrm{neg}}, 0).$$

because L trans-appears in $P_{\mathcal{E}}$. Moreover, by Lemma 3.13 we have **not** $\mathrm{rej}(A^j_{\mathrm{neg}}, 0) \in R_1$. Hence $A^j_{\mathrm{neg}} \in R_2$.

(b) If $r \in \rho(\mathcal{P}_j) \setminus \mathrm{Rej}(\mathcal{P}_j, M_j)$, then by Lemma 3.12 there must be some unrejected rewritten rule r^* of $P_{\mathcal{E}}$ with L^j in its head and the guardless body body^j. We can apply the inductive assumption for each literal in *body* and take the maximum p of all the indices we get. From the monotonicity of the immediate consequence operator (Prop. 2.11) we have $\mathit{body}^j \subseteq R_p$.

If r^* has an assertion guard, then let $q \in \mathbb{N}$ be such that the assertion guard belongs to R_q. Otherwise let $q = 1$.

Now let $s = \max\{p, q\}$. As r^* is unrejected, its rejection guard is also true in $R_1 \subseteq R_s$. We can use Prop. 2.11 again and infer $L^j \in R_{s+1}$. □

Lemma 3.15. Let L be a literal over $\mathcal{L}_{\mathcal{A}}$. If (3.6) holds, then

$$(\forall i \in \mathbb{N})(L^j \in R_i \Longrightarrow L \text{ trans-appears in } P_{\mathcal{E}} \wedge (\exists l \in \mathbb{N})(L \in M_{j,l})) \ .$$

Proof. We will prove by induction on i:

1° For $i = 0$ the claim trivially follows from $R_0 = \emptyset$.

2° We assume the claim holds for i, i.e. if L is a literal over $\mathcal{L}_{\mathcal{A}}$, then

$$L^j \in R_i \Longrightarrow L \text{ trans-appears in } P_{\mathcal{E}} \wedge (\exists l \in \mathbb{N})(L \in M_{j,l}) \ .$$

We will prove the claim for $i + 1$. So let's assume $L^j \in R_{i+1}$. Then there is some rule $r^* \in P_{\mathcal{E}}$ with L^j in its head and its body satisfied in R_i. Hence the first part of the proof is done – L trans-appears in $P_{\mathcal{E}}$ because $P_{\mathcal{E}}$ contains a rule with L^j in its head. To prove the second proposition, let's consider two cases:

(a) If r^* is a default assumption, then L is a default literal **not** A and r^* is of the form

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

and $\textbf{not}\, \text{rej}(A^j_{\text{neg}}, 0) \in R_1$. So by Lemma 3.13 we have $\textbf{not}\, A \in \text{Def}(\mathcal{P}_j, M_j)$, and thus $\textbf{not}\, A \in M_{j,1}$.

(b) If r^* is not a default assumption, then it must be an unrejected rewritten rule of $P_{\mathcal{E}}$. Let its guardless body be $body^j$. Then by Lemma 3.12 we have

$$(L \leftarrow body.) \in \rho(\mathcal{P}_j) \setminus \text{Rej}(\mathcal{P}_j, M_j) \ .$$

We can apply the inductive assumption for each literal in $body^j$ and take the maximum p of all the indices we get. From the monotonicity of the immediate consequence operator (Prop. 2.11) we have $body \subseteq M_{j,p}$. Hence $L \in M_{j,p+1}$. □

Lemma 3.16. Let L be a literal over $\mathcal{L}_{\mathcal{A}}$ and $j \in \{1, 2, \ldots, n\}$. Then

$$L^j \in N \iff L^j \in R \ .$$

Proof. We will prove by complete induction on j.

1° The basis can be inferred from the inductive step with $j = 1$.

2° We assume (3.6) holds and prove

$$L^j \in N \iff L^j \in R \ .$$

First let $L^j \in N$. Then $M_j \models L$ and L trans-appears in $P_{\mathcal{E}}$. So some $i \in \mathbb{N}$ exists such that $L \in M_{j,i}$ and by Lemma 3.14 we have $L^j \in R_l$ for some $l \in \mathbb{N}$. Hence $L^j \in R$.

On the other hand, if $L^j \in R$, then some $i \in \mathbb{N}$ exists such that $L^j \in R_i$. Thus by Lemma 3.15 L trans-appears in $P_{\mathcal{E}}$ and $L \in M_{j,l}$ for some $l \in \mathbb{N}$. This implies $L^j \in N$. □

Lemma 3.17. Let L be a literal over $\mathcal{L}_{\mathcal{A}}$, $j \in \{1, 2, \ldots, n\}$ and $i \in \{0, 1, \ldots, j\}$. Then

$$\mathrm{rej}(L^j, i) \in N \iff \mathrm{rej}(L^j, i) \in R \ .$$

Proof. We know that $\mathrm{rej}(L^j, i) \in N$ holds iff $\mathrm{Rej}^i(\mathcal{P}_j, M_j)$ contains a rule r_1 with L in its head. This in turn holds iff some $k \in \{i, i+1, \ldots, j\}$ exists such that $\mathcal{P}_j^k$ contains a rule $r_2 = (\textbf{not}\, L \leftarrow \mathit{body}.)$ such that $M_j \models \mathit{body}$. Furthermore, by Lemma 3.11 and Lemma 3.16 this holds iff

$$\begin{array}{c}P_{\mathcal{E}} \text{ contains a rewritten rule } r_2^* \text{ with } L^j_{\mathrm{neg}} \text{ in its head} \\ \text{and the guardless body } \mathit{body}^j \text{ and } \mathit{body}^j \subseteq R.\end{array} \tag{3.7}$$

Now let's assume $\mathrm{rej}(L^j, i) \in N$. Then (3.7) holds and according to the definition of rejection rules, $P_{\mathcal{E}}$ must also contain the rules

$$\begin{aligned}
&\mathrm{rej}(L^j, i_1) \leftarrow \mathit{body}^*.\\
&\mathrm{rej}(L^j, i_2) \leftarrow \mathrm{rej}(L^j, i_1).\\
&\mathrm{rej}(L^j, i_3) \leftarrow \mathrm{rej}(L^j, i_2).\\
&\qquad\qquad\vdots\\
&\mathrm{rej}(L^j, i_s) \leftarrow \mathrm{rej}(L^j, i_{s-1}).
\end{aligned}$$

where $k \geq i_1 > i_2 > \ldots > i_s = i$ and body^* contains all literals in body^j and the assertion guard of r_2^* if it has one. As $\mathit{body}^j \subseteq R$ and r_2^* is a rewritten rule, there must be some $p \in \mathbb{N}$ such that $\mathit{body}^* \subseteq R_p$. Hence we have $\mathrm{rej}(L^j, i_1) \in R_{p+1}$, $\mathrm{rej}(L^j, i_2) \in R_{p+2}$, $\ldots$, $\mathrm{rej}(L^j, i) \in R_{p+s} \subseteq R$.

For the converse implication let's assume $\mathrm{rej}(L^j, i) \in R$. Then $\mathrm{rej}(L^j, i) \in R_p$ for some $p \in \mathbb{N}$, so $P_{\mathcal{E}}$ contains some rejection rule r_3^* with $\mathrm{rej}(L^j, i)$ in its head and its body satisfied in R_{p-1}. We will consider two cases:

1. If r_3^* is of the form (3.2), then a look at the definition of rejection rules tells us that (3.7) is satisfied. Thus $\mathrm{rej}(L^j, i) \in N$.

2. If r_3^* is of the form (3.3), then its body is of the form $\mathrm{rej}(L^j, i_1)$ where $i < i_1 \leq j$. As $\mathrm{rej}(L^j, i_1) \in R_{p-1}$, $P_{\mathcal{E}}$ must contain a rejection rule with $\mathrm{rej}(L^j, i_1)$ in its head and its body satisfied in R_{p-2}. Two cases are possible again. In the first the proof ends and in the second we get an index i_2 such that $i_1 < i_2 \leq j$ and $\mathrm{rej}(L^j, i_2) \in R_{p-2}$. If the second case would occur forever, we would get an infinite increasing bounded sequence of natural numbers $i < i_1 < i_2 < \ldots \leq j$, which is not possible. Hence after a finite number of iterations the first case must occur. □

Lemma 3.18. It holds that

$$u \notin R\ .$$

Proof. We will prove by contradiction. Assume $u \in R$. Then for some atom

A of $\mathcal{L}_{\mathcal{A}}$ that trans-appears in $P_{\mathcal{E}}$ we have

$$\textbf{not}\, A^j, \textbf{not}\, A^j_{\text{neg}} \in R \ .$$

This implies

$$\textbf{not}\, A^j, \textbf{not}\, A^j_{\text{neg}} \in N^-$$

so we have

$$A^j, A^j_{\text{neg}} \notin N \ .$$

But then by the definition of N we have both $M_j \not\models A$ and $M_j \not\models \textbf{not}\, A$, which is not possible. □

Theorem 3.19 (Completeness)**.** N is a stable model of $P_{\mathcal{E}}$.

Proof. Follows from Lemmas 3.16, 3.17 and 3.18. □

3.4 Size of the Transformed Program

We now know the transformation defined in Def. 3.1 is sound and complete. But we also want to know something about its computational complexity because we want to use it to write an implementation 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 number of rules of the transformed program. The more rules there will be, the longer it will take to generate them and perform any further processing.

In this Section we will derive both a lower and an upper bound for the number of rules of 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. First let's take a look at the lower bound.

3.4.1 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:

$$|P_{\mathcal{E}}| \geq n|P| + \sum_{j=1}^{n} |E_j| \ . \tag{3.8}$$

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

3.4.2 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 $P_{\mathcal{E}}$. We will define the sets $\overline{A_1}, \overline{A_2}, \ldots, \overline{A_{n-1}}$ and prove that each $\overline{A_j}$ contains a rule r iff $P_{\mathcal{E}}^j$ contains some rule with $(\mathrm{assert}(r))^j$ in its head. The motivation for this is that in case it is true, then

$$|A| = \sum_{j=1}^{n} (n-j) \left|\overline{A_j}\right| \tag{3.9}$$

because each rule $r \in \overline{A_j}$ will generate $n - j$ assertable rules, one in each of $P_{\mathcal{E}}^{j+1}, P_{\mathcal{E}}^{j+2}, \ldots, P_{\mathcal{E}}^{n}$. The mentioned definition and proof follow:

Definition 3.20. Let $E_0 = \emptyset$. Then

$$A_1 \overset{\text{def}}{=} \{r \mid (\exists r_1 \in P)(H(r_1) = \mathrm{assert}(r))\} \ , \tag{3.10}$$

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

$$\begin{aligned} A_i \overset{\text{def}}{=} & \{r \mid (\exists r_1 \in A_{i-1})(H(r_1) = \mathrm{assert}(r))\} \\ & \cup \{r \mid (\exists r_2 \in E_{i-1})(H(r_2) = \mathrm{assert}(r_1) \wedge H(r_1) = \mathrm{assert}(r))\} \end{aligned} \tag{3.11}$$

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

$$\overline{A_j} \overset{\text{def}}{=} \bigcup_{i=1}^{j} A_i \cup \{r \mid (\exists r_1 \in E_j)(H(r_1) = \mathrm{assert}(r))\} \ . \tag{3.12}$$

Remark. Let $j \in \{1, 2, \ldots, n\}$. Each assertable rule in $P_{\mathcal{E}}^j$ is fully determined by its assertion guard, i.e. if we know that it has the assertion guard

$(\mathrm{assert}(r))^{i-1}$ and $r = (L \leftarrow \mathit{body}.)$, then the assertable rule must be:

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

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

Lemma 3.21. Let $i \in \{1, 2, \ldots, n-1\}$ and $r \in A_i$. Then for all $j \in \mathbb{N}$ such that $i < j \leq n$ the set $P^j_{\mathcal{E}}$ contains an assertable rule with the assertion guard $(\mathrm{assert}(r))^{j-1}$.

Proof. We will prove by induction on i.

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

2° We assume the claim holds for i and prove it for $i + 1$. Let $r \in A_{i+1}$ and let $j \in \mathbb{N}$ be such that $i + 1 < j \leq n$. Two cases are possible:

(a) Some rule $r_1 \in A_i$ exists such that $H(r_1) = \mathrm{assert}(r)$. By the induction hypothesis we have that $P^{j-1}_{\mathcal{E}}$ contains an assertable rule with the assertion guard $(\mathrm{assert}(r_1))^{j-2}$. This rule has $(\mathrm{assert}(r))^{j-1}$ in its head. Hence $P^j_{\mathcal{E}}$ contains an assertable rule with the assertion guard $(\mathrm{assert}(r))^{j-1}$.

(b) Some rule $r_2 \in E_i$ exists such that $H(r_2) = \mathrm{assert}(r_1)$ and $H(r_1) = \mathrm{assert}(r)$. Then $P^i_{\mathcal{E}}$ contains a rewritten event rule with $(\mathrm{assert}(r_1))^i$ in its head. Hence $P^{j-1}_{\mathcal{E}}$ will contain an assertable rule with the assertion guard $(\mathrm{assert}(r_1))^i$ and $(\mathrm{assert}(r))^{j-1}$ in its head. Therefore $P^j_{\mathcal{E}}$ must contain an assertable rule with the assertion guard $(\mathrm{assert}(r))^{j-1}$. □

Lemma 3.22. Let $j \in \{1, 2, \ldots, n-1\}$ and $r \in \overline{A_j}$. Then $P^j_{\mathcal{E}}$ contains a rule with $(\mathrm{assert}(r))^j$ in its head.

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

a) $r \in A_i$ for some $i \in \{1, 2, \dots, j\}$. Then by Lemma 3.21 we have that $P_{\mathcal{E}}^{j+1}$ contains an assertable rule with the assertion guard $(\mathrm{assert}(r))^j$. Hence $P_{\mathcal{E}}^j$ must contain a rule with $(\mathrm{assert}(r))^j$ in its head.

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

Lemma 3.23. Let $j \in \{1, 2, \dots, n-1\}$ and let r be a rule over $\mathcal{L}_{\mathcal{A}}$. If $P_{\mathcal{E}}^j$ contains a rule with $(\mathrm{assert}(r))^j$ in its head, then $r \in \overline{A_j}$.

Proof. We will prove by complete induction on j.

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

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

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

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

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

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

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

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

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

Theorem 3.24. Let $j \in \{1, 2, \ldots, n-1\}$ and let r be a rule over $\mathcal{L}_{\mathcal{A}}$. $P^j_{\mathcal{E}}$ contains a rule with $(\mathrm{assert}(r))^j$ in its head iff $r \in \overline{A_j}$.

Proof. Follows directly from Lemmas 3.22 and 3.23. □

Now we can make an approximation of $|A|$. According to (3.10), (3.11) and (3.12) we have for all $j \in \{1, 2, \ldots, n-1\}$

$$|A_j| \leq |P| + \sum_{i=1}^{j-1} |E_i| \ ,$$

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

Furthermore, by (3.9) we have

$$\begin{aligned} |A| = \sum_{j=1}^{n} (n-j)\left|\overline{A_j}\right| &\leq \sum_{j=1}^{n} (n-j)\left(j|P| + |E_j| + \sum_{i=1}^{j}(j-i)|E_i|\right) \\ &= |P|\sum_{j=1}^{n} j(n-j) + \sum_{j=1}^{n}(n-j)|E_j| + \sum_{j=1}^{n}(n-j)\sum_{i=1}^{j}(j-i)|E_i| \ . \end{aligned} \tag{3.13}$$

First let's solve the first sum:

$$\begin{aligned} \sum_{j=1}^{n} j(n-j) = n\sum_{j=1}^{n} j - \sum_{j=1}^{n} j^2 &= \frac{n^2(n+1)}{2} - \frac{n(n+1)(2n+1)}{6} \\ &= \frac{n(n+1)(3n-2n-1)}{6} = \frac{n^3-n}{6} \ . \end{aligned} \tag{3.14}$$

The third sum can be simplified as follows:

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

The inner sum is the same as the one in (3.14), just with $n-i$ instead of n.

Hence this holds, too:

$$\sum_{j=1}^{n}(n-j)\sum_{i=1}^{j}(j-i)|E_i| = \sum_{i=1}^{n}|E_i|\frac{(n-i)^3-(n-i)}{6} \quad . \tag{3.15}$$

So by (3.13), (3.14) and (3.15) we have

$$\begin{aligned} |A| &\leq |P|\frac{n^3-n}{6} + \sum_{j=1}^{n}(n-j)|E_j| + \sum_{i=1}^{n}|E_i|\frac{(n-i)^3-(n-i)}{6} \\ &= |P|\frac{n^3-n}{6} + \sum_{j=1}^{n}|E_j|\frac{(n-j)^3+5(n-j)}{6} \quad . \end{aligned} \tag{3.16}$$

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 assert atoms), 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

$$\begin{aligned} |A| &\leq \sum_{j=1}^{n}(n-j)(|P| + |E_j|) \\ &= |P|\frac{n^2-n}{2} + \sum_{j=1}^{n}(n-j)|E_j| \quad . \end{aligned} \tag{3.17}$$

3.4.3 Upper Bound

We already know the number of rewritten program rules and rewritten event rules in the transformed program. We also have 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 $\mathcal{L}_{\mathcal{A}}$ 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

$$|P_{\mathcal{E}}| \leq \frac{7}{2}\left(n|P| + \sum_{j=1}^{n}|E_j| + |A|\right) + n|\mathcal{L}_{P,\mathcal{E}}| \ . \tag{3.18}$$

If we use the approximation of $|A|$ (3.16) we get the following inequality:

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

which can be further simplified to

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

and by using the big-oh notation we get

$$|P_{\mathcal{E}}| = \mathcal{O}\big(n^3|P|\big) + \sum_{j=1}^{n}\mathcal{O}\big((n-j+1)^3|E_j|\big) + n|\mathcal{L}_{P,\mathcal{E}}| \ . \tag{3.19}$$

In case of programs without nested asserts we can use (3.17) 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, equivalently,

$$|P_{\mathcal{E}}| = \mathcal{O}\big(n^2|P|\big) + \sum_{j=1}^{n}\mathcal{O}((n-j+1)|E_j|) + n|\mathcal{L}_{P,\mathcal{E}}| \ . \tag{3.20}$$

3.4.4 Conclusion

The lower bound (3.8) for $|P_{\mathcal{E}}|$ implies that an implementation of **EVOLP** based on this transformation is not feasible usable for large values of n. For example, an agent running for a long time and frequently receiving events from its environment would soon reach a point at which its memory would be too small for the transformed program. In such situations, a different approach would be needed.

One possibility is to make an incremental implementation that always computes 1 dynamic stable model and constructs the next program in the sequence, just as it is drawn in Fig. 2.1. This approach was also used in the implementation mentioned in [15, 16]. Such an implementation has to deal with more details which can be both good and bad, depending on what we want to use it for. It brings more control over what is being computed and solves the problem with huge transformed programs when n is large. On the other hand, more control is also a source of more problems.

A typical situation that arises after each step is this one: We already have the program sequence $(P_1, P_2, \ldots, P_j)$ and we compute the dynamic stable models $M_{j,1}, M_{j,2}, \ldots, M_{j,n_j}$ of the **DLP** $\mathcal{P} = (P_1, P_2, \ldots, P_j \cup E_j)$. Now we need to choose the model according to which we will construct the program P_{j+1}. The easiest situation is when the sets $\{r \mid \mathrm{assert}(r) \in M_{j,i}\}$ are equal for all $i \in \{1, 2, \ldots, n_j\}$ because we only have one choice for P_{j+1}. But in general each of these sets can be different, so we can have exponentially many candidates. Moreover, if $n_j = 0$, then we have to backtrack and try a different candidate for one of $P_2, P_3, \ldots, P_j$. If many of the possible program sequences lead to such a dead end, we may have to try all the bad choices before we find some evolution stable model. This is illustrated in the following example:

Example 3.25. Consider this evolving logic program:

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

and the sequence of events $\mathcal{E}_n = (E_1, E_2, \ldots, E_n)$, $E_1 = E_2 = \ldots = E_n = \emptyset$. P given $\mathcal{E}_n$ has 2^n evolution stable models with 2^{n-1} different evolution traces. Now let $\mathcal{E}_{n+1} = (E_1, E_2, \ldots, E_{n+1})$ where

$$\begin{aligned} E_{n+1}: \quad & \mathrm{c} \leftarrow \mathrm{a}. \\ & \mathbf{not}\, \mathrm{c} \leftarrow \mathrm{a}. \end{aligned}$$

P given $\mathcal{E}_{n+1}$ has only two evolution stable models that share a common evolution trace. The incremental implementation may have to generate all 2^{n-1} evolution traces in order to find the correct one whereas the transformation-based implementation could find the two models much more quickly.

The good news regarding the transformation-based implementation is that, according to (3.19), 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, (3.20) implies that we can lower the power of n that $|P_{\mathcal{E}}|$ grows with. So for experimental use and especially for cases when n is not too large and we want to compute all evolution stable models or any evolution stable model (i.e. if we don't want to influence the order in which the models are computed), the transformation-based implementation is a good choice and, once we have the definition of the transformation, it is also easier to write and test.

Chapter 4

Implementation of EVOLP

The transformational semantics for **EVOLP** together with an ASP solver can be used to implement **EVOLP**. Figure 4.1 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.

4.1 Propositional Evolving Logic Programs

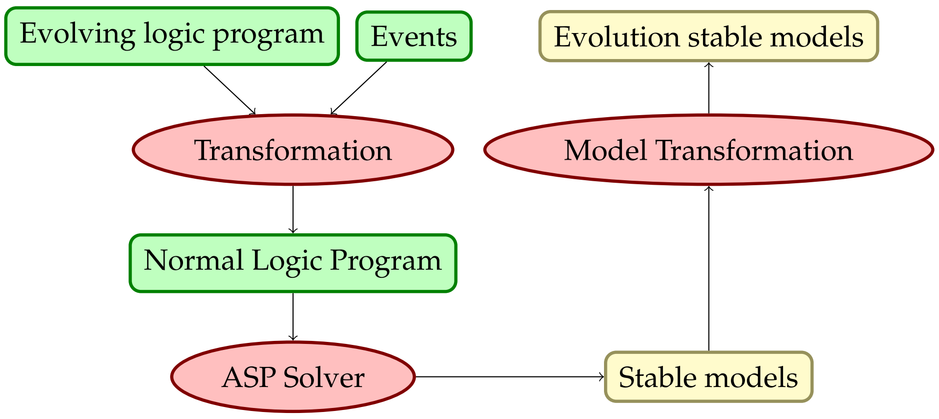
We decided to write the implementation in Java[1]. First we wrote a prototype to see if everything works as expected. It supports 2 ASP solvers: Smodels[2] and DLV[3]. It has two frontends, a web form[4] and a command line interface[5]. Both of these interfaces can be used to enter an evolving logic program and compute some or all its evolution stable models.

Parts of the prototype were later used to write a more extensible and modular implementation. It currently contains:

- classes for parsing and creating object models of logic programs, dynamic logic programs and evolving logic programs,

[1]`http://java.sun.com/`
[2]`http://www.tcs.hut.fi/Software/smodels/`
[3]`http://www.dbai.tuwien.ac.at/proj/dlv/`
[4]runs at `http://www.ii.fmph.uniba.sk/~slota/evolp-prop-prototype/`
[5]downloadable from `http://slotik.medovnicek.sk/2006/thesis/results/`

Figure 4.1: Implementation of **EVOLP** using the transformation



- classes for computing their stable models,
- a set of developer-friendly classes.

Sources are licensed under GPL[6]. They are provided with standard Java documentation (javadoc) and a set of tests. Although it is far from a complete test set, it tests the main functionality of the core classes. Support for DLV has been dropped for now and the implementation has only one frontend – a web form[7].

The current implementation of parser has extended capability. Parsing of variables and function symbols is supported. Computing (dynamic) stable models of (dynamic) logic programs with variables and function symbols is supported in case they can be grounded using lparse, i.e. variables and symbols satisfy the condition that each variable is (also) bound by some domain predicate. The implementation of **DLP**s is available through another web form[8].

As for **EVOLP**, variable support is more difficult to implement and the implementation is not finished yet. In Sect. 4.3 we'll show what problems there are and propose some solutions. But first we need to take a look at an alternative transformation for **EVOLP**.

[6] `http://www.gnu.org/licenses/licenses.html#GPL`
[7] runs at `http://www.ii.fmph.uniba.sk/~slota/evolp-prop/`
[8] runs at `http://www.ii.fmph.uniba.sk/~slota/dlp/`

4.2 Transformation into an Equivalent DLP

As we were facing the problems with variables we discovered the following: First we can transform the evolving logic program into an equivalent *dynamic* logic program. Then we can continue with the transformation from [17] to obtain an equivalent normal logic program. This makes the whole transformation easier to imagine, implement and debug. The definition of the transformation into an equivalent **DLP** follows:

Definition 4.1. Let P be an evolving logic program and

$$\mathcal{E} = (E_1, E_2, \dots, E_n)$$

an event sequence. Furthermore, let $\overline{A_j}$ be for all $j \in \{1, 2, \dots, n-1\}$ defined as in (3.12) and $\mathcal{L}_{\mathcal{T}}^{D} = \{A^j \mid A \in \mathcal{L}_{\mathcal{A}} \wedge 1 \le j \le n\}$. By a dynamic transformational equivalent of P given $\mathcal{E}$ we mean the dynamic logic program $\mathcal{P}_{P,\mathcal{E}} = (P_1, P_2, \dots, P_n)$ over $\mathcal{L}_{\mathcal{T}}^{D}$ consisting of exactly these rules:

1. **Rewritten program rules.** For every rule

   $$(L \leftarrow body.) \in P$$

   P_1 contains the rules

   $$\begin{gathered} L^1 \leftarrow body^1. \\ L^2 \leftarrow body^2. \\ \vdots \\ L^n \leftarrow body^n. \end{gathered}$$

2. **Rewritten event rules.** For all $j \in \{1, 2, \dots, n\}$ and every rule

   $$(L \leftarrow body.) \in E_j$$

   P_j contains the rule

   $$L^j \leftarrow body^j.$$

3. **Assertable rules.** For all $j \in \{1, 2, \dots, n-1\}$ and every rule

$$r = (L \leftarrow \mathit{body}.) \in \overline{A_j}$$

P^{j+1} contains the rules

$$\begin{aligned} L^{j+1} &\leftarrow \mathit{body}^{j+1}, (\mathrm{assert}(r))^j. \\ L^{j+2} &\leftarrow \mathit{body}^{j+2}, (\mathrm{assert}(r))^j. \\ &\vdots \\ L^n &\leftarrow \mathit{body}^n, (\mathrm{assert}(r))^j. \end{aligned}$$

The proofs of soundness and completeness of this transformation haven't been written yet, but they should be just simplified versions of the proofs for the original transformation. The new transformation essentially postpones the addition of default assumptions, rejection rules and totality constraints. It can be performed as soon as we know the contents of the sets $\overline{A_1}, \overline{A_2}, \dots, \overline{A_{n-1}}$. With a propositional language it is easy to construct them. But when the input program contains variables, we need to do something more.

4.3 Grounding of an Evolving Logic Program

The next example shows that sometimes we also want to assert a rule that is partially grounded, i.e. some of the variables appearing in the rule are instantiated and some are not:

Example 4.2. Consider the following program:

$$\begin{aligned} \mathrm{c}(1) &\leftarrow . \\ \mathrm{assert}(\mathrm{a}(X, Y) \leftarrow \mathrm{b}(Y)) &\leftarrow \mathrm{c}(X). \end{aligned}$$

What rule do we expect to be asserted in the second program? We probably want the variable X to get instantiated with 1 and Y should stay a variable and get grounded later when its time comes. So the desired rule to be asserted is:

$$\mathrm{a}(1, Y) \leftarrow \mathrm{b}(Y).$$

Alternatively, we could instantiate Y with all possible terms appearing in the program. But then we need to find the list of all terms before grounding the program. With function symbols this can get complicated. Either we write a separate grounder for this or we disallow the use of variables inside function symbols. Moreover, we should also include terms that could appear later in the evolution of the program because the rule will also be asserted into further **DLP**s. Most probably we would generate a lot of unnecessary grounded atoms in the models, and hence also unreachable rules in the further programs.

Another possibility is to say that such rules are not allowed, i.e. force the programmer to always bind all variables with a domain predicate. However, this removes some expressivity of the language. The first approach of asserting a partially grounded rule looks like the best alternative.

The proposed solution can be implemented as follows:

1. Ground those variables inside an $\mathrm{assert}(\cdot)$ in a head of a rule that also appear in that rule's body.
2. Protect the other variables by transforming them into constants.

However, this brings another issue worth considering:

Example 4.3. Let's take following program:

$$\begin{aligned} &\mathrm{assert}(\mathrm{a}(X) \leftarrow \mathrm{b}(X)) \leftarrow . \\ &\qquad\qquad\qquad \mathrm{b} \leftarrow \mathrm{assert}(\mathrm{a}(1) \leftarrow \mathrm{b}(1)). \end{aligned}$$

In case we encode X as a constant $\mathrm{enc\_X}$, b will not be true in the model. If we would like it to be there we need to add the rule

$$\mathrm{assert}(\mathrm{a}(1) \leftarrow \mathrm{b}(1)) \leftarrow \mathrm{assert}(\mathrm{a}(\mathrm{enc\_X}) \leftarrow \mathrm{b}(\mathrm{enc\_X})). \tag{4.1}$$

to the transformed program. The question whether b should really be in the model is left open.

So if we transform the variables into constants, they will not match the ordinary constants in asserts appearing in rule bodies. This can be solved by using the unification algorithm to find out when an atom with an encoded variable is more general than some other atom used in the body of some rule. In case such a pair is found, a rule like (4.1) is added to the transformed dynamic logic program.

Now to a different problem:

Example 4.4. Consider the rule:

$$\mathrm{assert}(X) \leftarrow \mathrm{says}(\mathrm{joe}, X). \tag{4.2}$$

In order for it to be of some use to us, the second argument of says/2 should always be a rule. Or at least there should be the possibility to put a rule there. We believe it is not a good idea to mix rules with ordinary terms because it can make things rather messy and we don't see any good use for a predicate that can have both an ordinary term and a rule as argument on the same position.

Therefore, we need to divide the variables into two groups – rule variables and ordinary variables. We also see that the rule variable X must be grounded before we know what set of rules can be generated into the next evolution steps by the rule (4.2).

One way of grounding the rule variables is to perform a step-by-step grounding where each single step looks as described below:

1. We already have a partially constructed $\mathcal{P}_{P,\mathcal{E}}$ that contains all rules with heads labeled by some $i \in \{1, 2, \ldots, j-1\}$ and also the sets $\overline{A_1}, \overline{A_2}, \ldots, \overline{A_{j-1}}$.

2. Hence we know the rules of $\mathcal{P}_{P,\mathcal{E}}$ with their heads labeled by j. We can add them to our partially constructed $\mathcal{P}_{P,\mathcal{E}}$ and ground it using lparse.

3. $\overline{A_{j+1}}$ now consists of those rules r for which $\mathcal{P}_{P,\mathcal{E}}$ contains a rule with $(\mathrm{assert}(r))^j$ in its head.

The current implementation already does this sort of grounding. But it doesn't handle variables inside heads of asserts correctly – it doesn't bind any of them with the variables in the rule's body. It also doesn't sort out the problem from Ex. 4.3 and doesn't check whether the programmer consistently used the predicates, i.e. it allows an input program like

$$\begin{gathered}\mathrm{p}(\mathrm{const}) \leftarrow . \\ \mathrm{assert}(X) \leftarrow \mathrm{p}(X).\end{gathered}$$

that would result in asserting a constant const which doesn't make much sense. The incomplete implementation is also available through a web form[9].

4.4 Optimizations in the Implementations

All implementations mentioned in this Chapter include an optimization that prevents the generation of unnecessary default assumptions and rejection rules. The formal definitions could also be simplified in this manner, but then they would get more complicated and it would be more difficult to work with them in the proofs. And although we haven't proved that these optimizations are safe to perform, the proofs should be easy to write given that we already proved the original transformation is sound and complete.

As an example let's take the transformed program from Ex. 2.23. The optimized implementation would not generate the default assumptions (2.11) and (2.12). Consequently, the rejection rules (2.18) and (2.19) would also be dropped. Moreover, the rejection rule (2.21) would be removed because $\mathrm{rej}(2, \mathrm{tired}^-)$ doesn't appear in the head of any rule.

[9]runs at http://www.ii.fmph.uniba.sk/~slota/evolp-var/

Chapter 5

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 effectiveness of the transformation and identified situations in which it is practically applicable. Properties of transformation-based implementations of **EVOLP** were compared with an incremental approach to the implementation. We also presented an implementation of evolving logic programs that relies on the defined transformation.

Future work can be devoted to improvements and extensions of the existing implementation. In particular, variable support needs to be finished, including the proof that the transformation from Def. 4.1 is sound and complete. In order to be practically usable, the implementation should also support weight constraints, arithmetic predicates and strong negation. The number of rules of the transformed program could also be optimized in certain situations.

References

[1] José Júlio Alferes, João Alexandre Leite, Luís Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. Dynamic logic programming. In A. Cohn, L. Schubert, and S. Shapiro, editors, *Procs. of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 98–111. Morgan-Kaufmann, June 1998.

[2] José Júlio Alferes, João Alexandre Leite, Luís Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. Updates of logic programs by logic programs. In *IIS'98: Seventh International Symposium on Intelligent Information Systems (Former WIS Series)*, pages 98–111, June 1998.

[3] José Júlio Alferes, João Alexandre Leite, Luís Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. Dynamic logic programming. In M. Falaschi J. L. Freire and M. Vialres-Ferro, editors, *Procs. of the 1998 Joint Conference on Declarative Programming (AGP'98)*, pages 393–408, July 1998.

[4] José Júlio Alferes, João Alexandre Leite, Luís Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1-3):43–70, September/October 2000.

[5] José Júlio Alferes, Federico Banti, and Antonio Brogi. A principled semantics for logic program updates. In Brewka and Peppas, editors, *Nonmonotonic Reasoning, Action, and Change (NRAC'03)*, 2003.

[6] José Júlio Alferes, Federico Banti, Antonio Brogi, and João Alexandre Leite. Semantics for dynamic logic programming: a principle-based approach. In V. Lifschitz and I. Niemelä, editors, *Procs. of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, pages 8–20. Springer-Verlag, 2004.

[7] José Júlio Alferes, Federico Banti, Antonio Brogi, and João Alexandre Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1):7–32, 2005.

[8] João Alexandre Leite. *Evolving Knowledge Bases – Specification and Semantics*. PhD thesis, Universidade Nova de Lisboa, July 2002.

[9] Martin 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)*, pages 78–95. Springer, 2004.

[10] José Júlio Alferes, Luís Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. LUPS – a language for updating logic programs. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Procs. of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 162–176. Springer, 1999.

[11] José Júlio Alferes, Luís Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. LUPS – a language for updating logic programs. *Artificial Intelligence*, 138(1&2), June 2002.

[12] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. A framework for declarative update specifications in logic programs. In *IJCAI'01*, pages 649–654. Morgan-Kaufmann, 2001.

[13] José Júlio Alferes, Antonio Brogi, João Alexandre Leite, and Luís Moniz 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)*, pages 50–61. Springer-Verlag, 2002.

[14] Implementations of logic programs updates,
http://centria.di.fct.unl.pt/~jja/updates/.

[15] João Alexandre Leite and Luís Soares. Enhancing a multi-agent system with evolving logic programs. In K. Inoue, K. Satoh, and F. Toni, editors, *Pre-Procs. of the 7th International Workshop on Computational Logic in Multi-Agent Systems, (CLIMA VII)*, pages 207–222, 2006.

[16] João Alexandre Leite and Luís 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)*. Springer-Verlag, 2007. To appear.

[17] Federico Banti, José Júlio Alferes, and Antonio 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.

[18] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003.

[19] José Júlio Alferes, Antonio Brogi, João Alexandre Leite, and Luís Moniz Pereira. An evolving agent with evolp. In P. Rullo N. Leone, editor, *Procs. of APPIA-GULP-PRODE'03 Joint Conf. on Declarative Programming (AGP'03)*, pages 205–216, September 2003.

[20] José Júlio Alferes, Antonio Brogi, João Alexandre Leite, and Luís Moniz Pereira. An evolvable rule-based e-mail agent. In S. Abreu, editor, *Progress in Artificial Intelligence, Procs. 11th Portuguese Int. Conf. on Artificial Intelligence (EPIA'03)*. Springer, December 2003.

[21] José Júlio Alferes, Antonio Brogi, João Alexandre Leite, and Luís Moniz Pereira. An evolving agent with evolp. In M. Klusch, S. Ossowski, A. Omicini, and H. Laamanen, editors, *Procs. of the 7th International Workshop on Cooperative Information Agents (CIA'03)*, pages 281–297. Springer-Verlag, 2003.

Definition Index

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which

made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this

ADDENDUM: How to use this License for your documents

Abstrakt

V priebehu svojho vývoja sa logické programovanie ukázalo byť dobrým a prirodzeným nástrojom na formuláciu, dotazovanie a manipulovanie symbolovo vyjadrených znalostí v mnohých aplikačných oblastiach informatiky. Avšak použiteľnosť týchto prostriedkov je v dynamických prostrediach podstatne obmedzená. Evolučné logické programy (**EVOLP**) sú elegantným rozšírením logického programovania, ktoré je vhodné pre multiagentové systémy, plánovanie, či iné použitie, v ktorom sa informácie dynamicky menia.

Táto práca sa zaoberá transformáciou, ktorá ľubovoľný evolučný logický program transformuje na ekvivalentný normálny logický program. Navrhnutú transformáciu ďalej využíva na naprogramovanie prvej voľne dostupnej, rozšíriteľnej a znova použiteľnej implementácie sémantiky evolutívnych stabilných modelov pre **EVOLP**.

Kľúčové slová: logické programovanie, sémantika stabilných modelov, evolučné logické programy, transformačná sémantika, implementácia