Operational semantics for DyLPs*

F. Banti[1], J. J. Alferes[1], and A. Brogi[2]

[1] CENTRIA, Universidade Nova de Lisboa, Portugal
[2] Dipartimento di Informatica, Università di Pisa, Italy

Abstract. Theoretical research has spent some years facing the problem of how to represent and provide semantics to updates of logic programs. This problem is relevant for addressing highly dynamic domains with logic programming techniques. Two of the most recent results are the definition of the refined stable and the well founded semantics for dynamic logic programs that extend stable model and well founded semantic to the dynamic case. We present here alternative, although equivalent, operational characterizations of these semantics by program transformations into normal logic programs. The transformations provide new insights on the computational complexity of these semantics, a way for better understanding the meaning of the update programs, and also a methodology for the implementation of these semantics. In this sense, the equivalence theorems in this paper constitute soundness an completeness results for the implementations of these semantics.

1 Introduction

In recent years considerable effort was devoted to explore the problem of how to update knowledge bases represented by logic programs (LPs) with new rules. This allows, for instance, to better use LPs for representing and reasoning with knowledge that evolves in time, as required in several fields of application. The LP updates framework has been used, for instance, as the base of the MINERVA agent architecture [14] and of the action description language EAPs [4].

Different semantics have been proposed [1, 2, 5, 6, 8, 15, 18, 19, 23] that assign meaning to arbitrary finite sequences $P_1, \ldots, P_m$ of logic programs. Such sequences are called *dynamic logic programs* (DyLPs), each program in them representing a supervenient state of the world. The different states can be seen as representing different time points, in which case P_1 is an initial knowledge base, and the other $P_i s$ are subsequent updates of the knowledge base. The different states can also be seen as knowledge coming from different sources that are (totally) ordered according to some precedence, or as different hierarchical instances where the subsequent programs represent more specific information. The role of the semantics of DyLPs is to employ the mutual relationships among different states to precisely determine the meaning of the combined program comprised of all individual programs at each state. Intuitively, one can add at the end of

* This work was supported by project POSI/40958/SRI/01, FLUX, and by the European Commission within the 6th Framework P. project Rewerse, no. 506779.

the sequence, newer rules or rules with precedence (arising from newly acquired, more specific or preferred knowledge) leaving to the semantics the task of ensuring that these added rules are in force, and that previous or less specific rules are still valid (by inertia) only as far as possible, i.e. that they are kept as long as they are not rejected. A rule is rejected whenever it is in conflict with a newly added one (*causal rejection of rules*). Most of the semantics defined for DyLPs [1, 2, 5, 6, 8, 15] are based on such a concept of causal rejection.

With the exception of the semantics proposed in [5], these semantics are extensions of the stable model semantics [10] to DyLPs and are proved to coincide on large classes of programs [8, 11, 13]. In [1] the authors provide theoretical results which strongly suggest that the refined semantics [1] should be regarded as the proper stable model-like semantics for DyLPs based on causal rejection. In particular, it solves some unintuitive behaviour of the other semantics in what regards updates with cyclic rules.

As discussed in [5], though a stable model-like semantics is the most suitable option for several application domains[1] other domains exist, whose specificities require a different approach. In particular, domains with huge amount of distributed and heterogenous data require an approach to automated reasoning capable of quickly processing knowledge, and of dealing with inconsistent information even at the cost of losing some inference power. Such areas demand a different choice of basic semantics, such as the well founded semantics [9]. In [5] a well founded paraconsistent semantics for DyLPs (WFDy) is defined. The WFDy semantics is shown to be a skeptical approximation of the refined semantic defined in [1]. Moreover, it is always defined, even when the considered program is inconsistent, and its computational complexity is polynomial wrt. the number of rules of the program. For these reasons we believe that the refined and the well founded semantics for DyLPs are useful paradigms in the knowledge representation field and hence implementations for computing both semantics are in order.

The existing definitions of both semantics are purely declarative, a feature that provides several advantages, like the simplicity of such definitions and the related theorems. However, when facing computational problem like establishing computational complexity and programming implementations, a more operational approach would have several advantages. For providing an operational definition for extensions of normal LPs, a widely used technique is that of having a transformation of the original program into a normal logic program and then to prove results of equivalence between the two semantics. In logic programs updates this methodology has been successfully used several times (see, for instance, [2, 8]). Once such program transformations have been established (and implemented), it is then an easy job to implement the corresponding semantics by applying existing software for computing the semantics of normal LPs, like DLV [7] or smodels [20] for the stable model semantics, or XSB-Prolog [22] for the well founded semantics. Following this direction, we provide two transfor-

[1] In particular, the stable model semantics has been shown to be a useful paradigm for modelling NP-complete problems.

mations of DyLPs into normal LPs (namely the *refined* and the *well founded transformation*), one for each semantics and provide equivalence results.

The shape of the transformations proposed for the refined and well founded semantics for DyLPs is quite different from the ones proposed for the other semantics (see for instance [2, 3, 12]). These differences are partially related to the different behaviors of the considered semantics (none of the existing program transformation is sound and complete w.r.t the refined and the well founded semantics) but they are also related to peculiar properties of the presented program transformations. One of such properties is the minimum size of the transformed program. Since the size of the transformed program significantly influences the cost of computing the semantics (especially in case of the stable model semantics), this topic is quite relevant. A drawback of the existing program transformations is that the size of the transformed program linearly depends on the size of the language times the number of performed updates. This means that, when the number of updates grows, the size of the transformed program grows in a way that linearly depends on the size of the language. This happens even when the updates are empty. On the contrary, in our approach the size of the transformed programs has an upper bound that does not depend on the number of updates, but solely (and linearly) on the number of rules and the size of the original language (see Theorems 2 and 4).

Prototypical implementations that use the theoretical background of this paper are available at http://centria.di.fct.unl.pt/∼banti/implementation.htm. These implementations take advantage of DLV, smodels, and XSB-Prolog systems to compute the semantics of the transformed programs.

Due to their simplicity, the proposed transformations are also interesting beyond the scope of implementation. They give new insights on how the rejection mechanism works and how it creates new dependencies among rules. The transformed programs provide an alternative, more immediate description of the behaviour of the updated program.

The rest of the paper is structured as follows. Section 2 establishes notation and provides some background and the formal definition of the refined and well founded semantics for DyLPs. Section 3 illustrates the refined transformation, describes some of its properties and makes comparisons to related transformations for other semantics. The well founded transformation is defined and studied in Section 4 . Finally, Section 5 draws conclusions and mentions some future developments. For lack of space, proofs cannot be presented here, but are available on a technical report from the authors.

2 Background: Concepts and notation

In this section we briefly recall the syntax of DyLPs, and the refined and well founded semantics defined, respectively, in [1] and [5].

To represent negative information in logic programs and their updates, DyLP uses generalized logic programs (GLPs) [16], which allow for default negation *not A* not only in the premises of rules but also in their heads. A language $\mathcal{L}$ is

any set literals of the form A or *not* A such that $A \in \mathcal{L}$ iff *not* $A \in \mathcal{L}$. A GLP defined over a propositional language $\mathcal{L}$ is a (possibly infinite) set of ground rules of the form $L_0 \leftarrow L_1, \ldots, L_n$, where each L_i is a literal in $\mathcal{L}$, i.e., either a propositional atom A in $\mathcal{L}$ or the default negation *not* A of a propositional atom A in $\mathcal{L}$. We say that A is the *default complement* of *not* A and viceversa. With a slight abuse of notation, we denote by *not* L the default complement of L (hence if L is the *not* A, then *not* L is the atom A). Given a rule τ as above, by $hd(\tau)$ we mean L_0 and by $B(\tau)$ we mean $\{L_1, \ldots, L_n\}$.

In the sequel an *interpretation* is simply a set of literals of $\mathcal{L}$. A literal L is *true* (resp. *false*) in I iff $L \in I$ (resp. *not* $L \in I$) and *undefined* in I iff $\{L, not\ L\} \cap I = \{\}$. A conjunction (or set) of literals C is true (resp. false) in I iff $C \subseteq I$ (resp. $\exists\ L \in C$ such that L is false in I). We say that I is *consistent* iff $\forall\ A \in \mathcal{L}$ *at most* one of A and *not* A belongs to I, otherwise we say I is *paraconsistent*. We say that I is *2-valued* iff for each atom $A \in \mathcal{L}$ *exactly* one of A and *not* A belongs to I.

A *dynamic logic program* with length n over a language $\mathcal{L}$ is a finite sequence $P_1, \ldots, P_n$ (also denoted $\mathcal{P}$, where the P_is are GLPs indexed by $1, \ldots, n$), where all the P_is are defined over $\mathcal{L}$. Intuitively, such a sequence may be viewed as the result of, starting with program P_1, updating it with program P_2, ..., and updating it with program P_n. For this reason we call the singles $P_i s$ *updates*. Let P_j and P_i be two updates of $\mathcal{P}$. We say that P_j is *more recent* than P_i iff $i < j$. We use $\rho(\mathcal{P})$ to denote the multiset of all rules appearing in the programs $P_1, ..., P_n$.

The *refined stable model semantics* for DyLPs is defined in [1] by assigning to each DyLP a set of stable models. The basic idea of the semantics is that, if a later rule τ has a true body, then former rules in conflict with τ should be *rejected*. Moreover, any atom A for which there is no rule with true body in any update, is considered false by default. The semantics is then defined by a fixpoint equation that, given an interpretation I, tests whether I has exactly the consequences obtained after removing from the multiset $\rho(\mathcal{P})$ all the rules rejected given I, and imposing all the default assumptions given I. Formally, let:

$$Default(\mathcal{P}, I) = \{not\ A \mid \nexists\ A \leftarrow body \in \rho(\mathcal{P}) \wedge body \subseteq I\}$$
$$Rej^S(\mathcal{P}, I) = \{\tau \mid \tau \in P_i | \ \exists\ \eta \in P_j\ i \leq j,\ \tau \bowtie \eta \wedge\ B(\eta) \subseteq I\}$$

where $\tau \bowtie \eta$ means that τ and η are conflicting rules, i.e. the head of τ is the default complement of the head of η.

Definition 1. *Let $\mathcal{P}$ be any DyLP of length n, $i \leq n$ over language $\mathcal{L}$ and M a two valued interpretation and let $\mathcal{P}^i$ be the prefix of $\mathcal{P}$ with length i. Then M is a refined stable model of $\mathcal{P}$, at state i, iff M is a fixpoint of $\Gamma^S_{\mathcal{P}i}$:*

$$\Gamma^S_{\mathcal{P}i}(M) = least\left(\rho\left(\mathcal{P}^i\right) \setminus Rej^S(\mathcal{P}^i, M) \cup Default(\mathcal{P}^i, M)\right)$$

where $least(P)$ denotes the least Herbrand model of the definite program obtained by considering each negative literal not A in P as a new atom[2].

[2] Whenever clear from the context, we omit the $\mathcal{P}$ in the above defined operators.

The definition of dynamic stable models of DyLPs [2] is as the one above, but where the $i \leq j$ in the rejection operator is replaced by $i < j$. I.e., if we denote this other rejection operator by $Rej(\mathcal{P}, I)$, and define $\Gamma_{\mathcal{P}}(I)$ by replacing in $\Gamma^S_{\mathcal{P}}$ Rej^S by Rej, then the stable models of $\mathcal{P}$ are the interpretations I such that $I = \Gamma_{\mathcal{P}}(I)$.

The *well founded semantics for DyLPs* is defined through the two operators Γ and Γ^S. We use the notation $\Gamma\Gamma^S$ to denote the operator obtained by first applying Γ^S and then Γ. The well founded model of a program is defined as the least fix point of such operator. Formally:

Definition 2. *The well founded model $WFDy(\mathcal{P})$ of a DyLP $\mathcal{P}$ at state i is the (set inclusion) least fixpoint of $\Gamma_{\mathcal{P}^i}\Gamma^S_{\mathcal{P}^i}$ where $\mathcal{P}^i$ is the prefix of $\mathcal{P}$ with length i.*

Since the operators Γ and Γ^S are anti-monotone (see [5]) the composite operator $\Gamma\Gamma^S$ is monotone and, as it follows from the Tarski-Knaster Theorem [21], it always has a least fixpoint. In other words, $WFDy$ is uniquely defined for every DyLP. Moreover, $WFDy(\mathcal{P})$ can be obtained by (transfinitely) iterating $\Gamma\Gamma^S$, starting from the empty interpretation. As already mentioned, the refined and well founded semantics for DyLPs are strongly related. In particular, they share analogous connections to the ones shared by the stable model and the well founded semantics of normal LPs, as we see from the following proposition.

Proposition 1. *Let M be any refined stable model of $\mathcal{P}$. The well founded model $WFDy(\mathcal{P})$ is a subset of M. Moreover, if $WFDy(\mathcal{P})$ is a 2-valued interpretation, it coincides with the unique refined stable model of $\mathcal{P}$.*

This property does not hold if, instead of the refined semantics, we consider any of the other semantics based on causal rejection [2, 6, 8, 15].

Example 1. Let $\mathcal{P}: \ P_1, \ P_2, \ P_3$ be the as follows:

$$P_1 : a \leftarrow b. \qquad P_2 : \ b. \quad c. \qquad P_3 : not\ a \leftarrow c.$$

The well founded model of $\mathcal{P}$ is $M = \{b, c, not\ a\}$. Moreover, M is a two valued interpretation and so, by proposition 1, M is also the unique refined model.

3 A program transformation for the refined semantics.

The refined transformation defined in this section turns a DyLP $\mathcal{P}$ in the language $\mathcal{L}$ into a normal logic program $\mathcal{P}^R$ (called the *refined transformational equivalent of* $\mathcal{P}$) in an extended language. We provide herein a formal procedure to obtain the transformational equivalent of a given DyLP.

Let $\mathcal{L}$ be a language. By $\mathcal{L}^R$ we denote the language whose elements are either atoms of $\mathcal{L}$, or atoms of one of the following forms: u, A^-, $rej(A, i)$, $rej(A^-, i)$, where i is a natural number, A is any atom of $\mathcal{L}$ and no one of the atoms above belongs to $\mathcal{L}$. Intuitively, A^- stands for "A is false", while $rej(A, i)$ (resp. $rej(A^-, i)$), stands for: "all the rules with head A (resp. *not* A) in the update P_i are rejected". For every literal L, if L is an atom A, then $\overline{L}$ denotes

A itself, while if L is a negative literal *not* A then $\overline{L}$ denotes A^-. Finally, u is a new atom not belonging to $\mathcal{L}$ which is used for expressing integrity constraints of the form $u \leftarrow not\ u, L_1, \ldots, L_k$ (which has the effect of removing all stable models containing $L_1, \ldots, L_k$).

Definition 3. *Let $\mathcal{P}$ be a Dynamic Logic Program whose language is $\mathcal{L}$. By the refined transformational equivalent of $\mathcal{P}$, denoted $\mathcal{P}^R$, we mean the normal program $P_1^R \cup \ldots \cup P_n^R$ in the extended language $\mathcal{L}^R$, where each P_i^R exactly consists of the following rules:*

Default assumptions *For each atom A of $\mathcal{L}$ appearing in P_i, and not appearing in any other P_j, $j \leq i$ a rule:*

$$A^- \leftarrow not\ rej(A^-, 0)$$

Rewritten rules *For each rule $L \leftarrow body$ in P_i, a rule:*

$$\overline{L} \leftarrow \overline{body},\ not\ rej(\overline{L}, i)$$

Rejection rules *For each rule $L \leftarrow body$ in P_i, a rule:*

$$rej(\overline{not\ L}, j) \leftarrow \overline{body}$$

where $j \leq i$ is either the largest index such that P_j has a rule with head not L or. If no such P_j exists, and L is a positive literals, then $j = 0$, otherwise this rule is not part of P_i^R. Moreover, for each rule $L \leftarrow body$ in P_i, a rule:

$$rej(\overline{L}, j) \leftarrow rej(\overline{L}, i)$$

where $j < i$ is the largest index such that P_j also contains a rule $L \leftarrow body$. If no such P_j exists, and L is a negative literal, then $j = 0$, otherwise this rule is not part of P_i^R.

Totality constraints *For each pair of conflicting rules in P_i, with head A and not A, the constraint:*

$$u \leftarrow not\ u,\ not\ A,\ not\ A^-$$

Let us briefly explain the intuition and the role for each of these rules. The *default assumptions* specify that a literal of the form A^- is true (i.e. A is false) unless this initial assumption is rejected. The *rewritten rules* are basically the original rules of the sequence of programs with an extra condition in their body that specifies that in order to derive conclusions, the considered rule must not be rejected. Note that, both in the head and in the body of a rule, the negative literals of the form *not* A are replaced by the corresponding atoms of the form A^-. The role of *rejection rules* is to specify whether the rules with a given head in a given state are rejected or not. Such a rule may have two possible forms. Let $L \leftarrow body$ be a rule in P_i. The rule of the form $rej(\overline{L}, j) \leftarrow \overline{body}$ specifies that all the rules with head *not* L in the most recent update P_j with $j \leq i$ must be rejected. The rules of the form $rej(\overline{L}, j) \leftarrow rej(\overline{L}, i)$ "propagate" the rejection to the updates below P_j. Finally, *totality constraints* assure that, for each literal A, at least one of the atoms A, A^- belongs to the model. This is done to guarantee that the models of transformed program are indeed two valued.

The role of the atoms of the extended language $\mathcal{L}^R$ that do not belong to the original language $\mathcal{L}$ is merely auxiliary, as we see from the following theorem. Let $\mathcal{P}$ be any Dynamic Logic Program and P_i and update of $\mathcal{P}$. We use $\rho(\mathcal{P})^{Ri} = P_0^R \cup \ldots \cup P_i^R$.

Theorem 1. *Let $\mathcal{P}$ be any Dynamic Logic Program in the language $\mathcal{L}$, P_i an update of $\mathcal{P}$, and let $\rho(\mathcal{P})^{Ri}$ be as above. Let M be any interpretation over $\mathcal{L}$. Then M is a refined stable model of $\mathcal{P}$ at P_i iff there exits a two valued interpretation M^R such that M^R is a stable model of $\rho(\mathcal{P})^{Ri}$ and $M \equiv_{\mathcal{L}} M^R$. Moreover, M and M^R satisfy the following conditions:*

$$\begin{array}{ll} A \in M \Leftrightarrow A \in M^R & not\ A \in M \Leftrightarrow A^- \in M^R \\ not\ A \in Default(\mathcal{P}^i, M)) \Leftrightarrow & rej(A^-, 0) \notin M^R \\ \tau \in rej^S(M, \mathcal{P}^i) \wedge \tau \in P_i \Leftrightarrow & rej(\overline{hd(\tau)}, i) \in M^R \end{array}$$

For illustration, we present an example of the computation of the refined transformational equivalent of a DyLP.

Example 2. Let $\mathcal{P}: P_1, P_2$ be the as in example 1. The transformational equivalent of $\mathcal{P}$ is the following sequence P_1^R, P_1^R, P_2^R:

$$\begin{array}{lll} P_1^R: & a^- \leftarrow not\ rej(a^-, 0). & b^- \leftarrow not\ rej(b^-, 0). \\ & a \leftarrow b,\ not\ rej(a, 1). & rej(a^-, 0) \leftarrow b. \\ P_2^R & c^- \leftarrow not\ rej(c^-, 0). & \\ & rej(b^-, 0). & rej(c^-, 0). \\ & b \leftarrow not\ rej(b, 2). & c \leftarrow not\ rej(c, 2). \\ P_3^R: & a^- \leftarrow c,\ not\ rej(a, 3). & rej(a, 1) \leftarrow c. \end{array}$$

For computing the refined semantics of $\mathcal{P}$ at P_2 we just have to compute the stable model semantics of the program $P_1^R \cup P_2^R$. This program has a single stable model M^R consisting of the following set[3].

$$M^R = \{a^-,\ b,\ c, rej(a, 1),\ rej(a^-, 0),\ rej(b^-, 0),\ rej(c^-, 0)\}$$

We conclude that, $\mathcal{P}$ has $M = \{b,\ c\}$ as the unique refined model. To compute the refined semantics of $\mathcal{P}$ we have to compute, instead, the stable model semantics of the program $P^R = P_1^R \cup P_2^R \cup P_3^R$. Let us briefly examine the transformed program P^R and see how it clarifies the meaning of the related DyLP. Since there exist no rules with head $rej(b, 2)$ and $rej(c, 2)$, we immediately infer b and c. Then we also infer $rej(a, 0)$, $rej(b, 0)$ and $rej(c, 0)$, and so that all the default assumptions are rejected. The last rule of P_3^R implies $rej(a, 1)$, thus the rule $a \leftarrow b$ in P_1 is rejected and we do not infer a. In fact, we infer a^- by the first rule of P_3^R. Hence, the program has the single stable model M^R which means $\mathcal{P}$as the unique refined model $\{b,\ c\}$.

To compute the refined semantics of a given DyLP $P_1, \ldots, P_n$ at a given state, it is sufficient to compute its refined transformational equivalent $P_1^R, \ldots P_n^R$, then

[3] As usual in the stable model semantics, hereafter we omit the negative literals

to compute the stable model semantics of the normal logic program $\rho(\mathcal{P})^{Ri}$ and, finally, to consider only those literals that belong to the original language of the program. A feature of the transformations in this papers, is that of being incremental i.e., whenever a new update P_{n+1} is received, the transformational equivalent of the obtained DyLP is equal to the union of P^R_{n+1} and the refined transformational equivalent of the original DyLP. The efficiency of the implementation relies on largely on the size of the transformed program compared to the size of the original one. We present here a theoretical result that provides an upper bound for the number of clauses of the refined transformational equivalent of a DyLP.

Theorem 2. *Let $\mathcal{P} = P_1, \ldots P_m$ be any finite ground DyLP in the language $\mathcal{L}$ and let $\rho(\mathcal{P})^{Rn}$ be the set of all the rules appearing in the refined transformational equivalent of $\mathcal{P}$. Moreover, let m be the number of clauses in $\rho(\mathcal{P})$ and l be the cardinality of $\mathcal{L}$[4]. Then, the program $\rho(\mathcal{P})^{Rn}$ consists of at most $2m + l$ rules.*

The problem of satisfiability under the stable model semantics (i.e. to find a stable model of a given program) is known to be NP-Complete, while the inference problem (i.e. to determine if a given proposition is true in all the stable models of a program) is co-NP-Complete [17]. Hence, from the fact that DyLPs extends the class of normal LPs and from theorems 1 and 2 it immediately follows that such problems are still NP-Complete and co-NP-Complete also under the refined semantics for DyLPs. The size of the refined transformational equivalent of a DyLP depends linearly and solely on the size of the program and of the language. It has an upper bound which does not depend on the number of updates performed. Thus, we gain the possibility of performing several updates of our knowledge base without losing too much on efficiency.

The refined transformation presents some similarities with the one presented in [2] and [8]. The three transformations use new atoms to represent rejection of rules. A fundamental difference between these transformations is that they are not semantically equivalent. The transformation in [2] is defined for implementing the *dynamic stable model semantics of DyLPs* of [2], while the one in [8] implements the *Update semantics* [8]. These semantics are not equivalent to the refined one, which was, in fact, introduced for solving some counterintuitive behaviours of the previously existing semantics for DyLPs (cf. [1]). In particular, it is proved in [1] that every refined stable model is also a dynamic stable model and an update stable model but the opposite is not always true. Moreover, the size of the transformation defined in [2] is $2m + l(n + 2)$ where l and m are as in Theorem 2 and n is the number of updates of the considered DyLP. Hence, a single (even empty or single rule) update add at least l rules to the transformed program. A similar result also holds when considering the transformation of [8] (here the size of the transformed program is $2m + nl$). The size of the refined transformational equivalent is instead independent from n. Hence, for DyLPs with many updates, the transformed programs of these transformation become

[4] Since $\mathcal{L}$ contains the positive and the negative literals, l is equal to two times the number of predicates appearing in $\mathcal{P}$.

considerably larger then the ones of the refined transformation, especially in cases where each of these updates has few rules.

Moreover, the transformations of [2] and [8] approach the problem of computing the semantics at different states by introducing an extra index on the body of the transformed program. On the contrary, when using the refined transformation, it is sufficient to ignore the rules of the transformed program that are related to the updates after P_i. Apart from computational aspect, the use of extra indexes and the proliferation of rules make these semantics unsuitable for the purpose of understanding the behaviour of the updated program.

4 Transformational well founded semantics

The well founded transformation turns a given DyLP $\mathcal{P}$ in the language $\mathcal{L}$ into a normal logic program $\mathcal{P}^{TW}$ in an extended language $\mathcal{L}^W$ called the *well founded transformational equivalent of* $\mathcal{P}$.

Let $\mathcal{L}$ be a language. By $\mathcal{L}^W$ we denote the language whose atoms are either atoms of $\mathcal{L}$, or are atoms of one of the following forms: A^S A^{-S}, $rej(A,i)$, $rej(A^S,i)$, $rej(A^-,i)$, and $rej(A^{-S},i)$, where i is a natural number, A is any atom of $\mathcal{L}$ and no one of the atoms above belongs to $\mathcal{L}$. Given a conjunction of literals C, use the notation C^S for the conjunction obtained by replacing any occurrence of an atom A in C with A^S.

Definition 4. *Let $\mathcal{P}$ be a Dynamic Logic Program on the language $\mathcal{L}$. By the well founded transformational equivalent of $\mathcal{P}$, denoted $\mathcal{P}^{TW}$, we mean the normal program $P_1^W \cup \ldots \cup P_n^W$ in the extended language $\mathcal{L}^W$, where each P_i exactly consists of the following rules:*

Default assumptions *For each atom A of $\mathcal{L}$ appearing in P_i, and not appearing in any other P_j, $j \leq i$ the rules:*

$$A^- \leftarrow not\ rej(A^{-S},0) \qquad A^{-S} \leftarrow not\ rej(A^-,0)$$

Rewritten rules *For each rule $L \leftarrow body$ in P_i, the rules:*

$$\overline{L} \leftarrow \overline{body},\ not\ rej(\overline{L}^S,i) \qquad \overline{L}^S \leftarrow \overline{body}^S,\ not\ rej(\overline{L},i)$$

Rejection rules *For each rule $L \leftarrow body$ in P_i, a rule:*

$$rej(\overline{not\ L},j) \leftarrow \overline{body}$$

where $j < i$ is the largest index such that P_j has a rule with head not L. If no such P_j exists, and L is a positive literals, then $j = 0$, otherwise this rule is not part of P_i^W.
Moreover, for each rule $L \leftarrow body$ in P_i, a rule:

$$rej(\overline{not\ L}^S,k) \leftarrow \overline{body}^S.$$

where $k \leq i$ is the largest index such that P_k has a rule with head not L. If no such P_j exists, and L is a positive literals, then $j = 0$, otherwise this rule is not part of P_i^W.

Finally, for each rule $L \leftarrow body$ in P_i, the rules:

$$rej(\overline{L}^S, j) \leftarrow rej(\overline{L}^S, i) \qquad rej(\overline{L}, j) \leftarrow rej(\overline{L}, i)$$

where $j < i$ is the largest index such that P_j also contains a rule $L \leftarrow body$. If no such P_j exists, and L is a negative literal, then $j = 0$, otherwise these rules are not part of P_i^W.

As the reader can see, the program transformation above resembles the one of Definition 3. The main difference is that the transformation of Definition 4 duplicates the language and the rules. This is done for simulating the alternate application of two different operators, Γ and Γ^S used in the definition of $WFDy$. The difference between these two operators is on the rejection strategies: the Γ^S operator allows rejection of rules in the same state, while the Γ operator does not. In the transformation above this difference is captured by the definition of the rejection rules. Let $L \leftarrow body$ be a rule in the update P_i. In the rules of the form $rej(\overline{L}, j) \leftarrow \overline{body}^S$ and $rej(\overline{L}^S, j) \leftarrow \overline{body}^S$, j is *less* than i, in the first case, and *less or equal* than i in the second one. A second difference is the absence of the *totality constraints*. This is not surprising since the well founded model is not necessarily a two valued interpretation. Note however, that the introduction of any rule of the form $u \leftarrow not\ u$, $body$ would not change the semantics. As for the refined transformation, the atoms of the extended language $\mathcal{L}^W$ that do not belong to the original language $\mathcal{L}$ are merely auxiliary. Let $\mathcal{P}$ be any Dynamic Logic Program, P_i and update of $\mathcal{P}$, and let $\rho(\mathcal{P})^{Ri} = P_0^W \cup \ldots \cup P_i^W$.

Theorem 3. *Let $\mathcal{P}$be any Dynamic Logic Program in the language $\mathcal{L}$, P_i and update of $\mathcal{P}$, and let $\rho(\mathcal{P})^{Ri}$ be as above. Moreover, Let W_i be the well founded model of the normal logic program $\rho(\mathcal{P})^{Ri}$ and $WFDy(\mathcal{P}^i)$ be the well found model of $\mathcal{P}$ at P_i. Then $WFDy(\mathcal{P}^i) = \{A \mid \ \in W_i\} \cup \{not\ A \mid A^- \in W_i\}$*

To compute the well founded semantics of a given DyLP $P_1, \ldots, P_n$ at a given state, it is hence sufficient to compute its well founded transformational equivalent $P_1^W, \ldots P_n^W$, then to compute the well founded model of the normal logic program $\rho(\mathcal{P})^{Rn}$ and, finally, to consider only those literals that belong to the original language of the program.

We present here a result analogous that of Theorem 2 that provides an upper bound to the size of the well founded transformational equivalent.

Theorem 4. *Let $\mathcal{P} : P_1, \ldots P_m$ be any finite ground DyLP in the language $\mathcal{L}$ and let $\rho(\mathcal{P})^{Rn}$ be the set of all the rules appearing in the transformational equivalent of $\mathcal{P}$. Moreover, let m be the number of clauses in $\rho(\mathcal{P})$ and l be the cardinality of $\mathcal{L}$. Then, the program $\rho(\mathcal{P})^{Rn}$ consists of at most $5m + l$ rules.*

The problem of computing the well-founded model of a normal LP has a polynomial complexity [9]. Hence, from Theorems 3 and 4, it follows that such a problem is polynomial also under the well founded semantics for DyLPs.

Other program transformations for the computation of a well founded-like semantics for DyLPs (see [2, 3, 12]), do not compute the well founded semantics of DyLPs, as shown in [5]. Moreover, they suffer from the same drawbacks on the size of the transformed program that have been discussed in Section 3.

5 Concluding remarks and future works

Dynamic Logic Programs is a framework for representing knowledge that evolves with time. The purpose of this paper was to illustrate operational characterizations of the refined and well founded semantics for DyLPs defined by program transformations that transform a DyLP into a semantically equivalent normal LP. This directly provides a way to implement these semantics, by relying on software like DLV, smodels (for the refined semantics) and XSB-Prolog (for the well founded one). Moreover, we have shown that the size of the transformed programs is linearly bound by the size of the original program. Moreover, especially in case of the refined semantics, the transformed program is usually readable and may help to better understand the meaning of the considered DyLP.

The close relationships between the two semantics rise the question whether an approach to the implementation based on a *single* program transformation would have been possible instead. Indeed, the answer is positive. The program transformation of Definition 4 for computing the $WFDy$ semantics can be adapted for the refined semantics. This is done by adding proper integrity constraints of the form $u \leftarrow not\ u,\ body$ to the transformed program. Recall that, as noted in section 4, the addition of such constraints does not change the well founded model of the program, and hence the new program transformation would still compute the $WFDy$ semantics.

We opted for presenting two separate transformations for a matter of optimization. Indeed, the size of the transformed program after the transformation of Definition 3 is less than half the size that the unique transformation would require. Moreover, the transformed program can be used also for "reading" a DyLP as a normal logic program, a task for which a program transformation with many more auxiliary rules and predicates would not be suitable.

As mentioned in the introduction there are several works on possible usage of DyLPs. Other possible usages can be found in any application areas where evolution and reactivity are a primary issue. We believe it is the time for the research on DyLPs to realize practical applications of the framework, to provide implementations for such applications and face real world problems. One of the most obvious things to do is to transform the existing prototypical implementation into a real system and test its performance. In this perspective the paper presented here is a, still preliminary, but fundamental step.

References

1. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1), 2005.
2. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, 2000. A preliminary version appeared in KR'98.
3. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002.

4. J.J. Alferes, F. Banti, and A. Brogi. From logic programs updates to action description updates. In *CLIMA V*, 2004.
5. F. Banti, J. J. Alferes, and A. Brogi. The well founded semantics for dynamic logic programs. In Christian Lemaître, editor, *Proceedings of the 9th Ibero-American Conference on Artificial Intelligence (IBERAMIA-9)*, LNAI, 2004.
6. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming (ICLP-99)*, Cambridge, November 1999. MIT Press.
7. DLV. The DLV project - a disjunctive datalog system (and more), 2000. Available at `http://www.dbai.tuwien.ac.at/proj/dlv/`.
8. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2002.
9. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
10. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
11. M. Homola. Dynamic logic programming: Various semantics are equal on acyclic programs. In J. Leite and P. Torroni, editors, *5th Int. Ws. On Computational Logic In Multi-Agent Systems (CLIMA V)*. Pre-Proceedings, 2004. ISBN: 972-9119-37-6.
12. J. A. Leite. Logic program updates. Master's thesis, Dept. de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, November 1997.
13. J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
14. J. A. Leite, J. J. Alferes, and L. M. Pereira. Minerva - a dynamic logic programming agent architecture. In J. J. Meyer and M. Tambe, editors, *Intelligent Agents VIII — Agent Theories, Architectures, and Languages*, volume 2333 of *LNAI*, pages 141–157. Springer-Verlag, 2002.
15. J. A. Leite and L. M. Pereira. Iterated logic program updates. In J. Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP-98)*, pages 265–278, Cambridge, 1998. MIT Press.
16. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the 3th International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*. Morgan-Kaufmann, 1992.
17. W. Marek and M. Truszczynski. Autoepistemic logics. *Journal of the ACM*, 38(3):588–619, 1991.
18. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-99)*, volume 1730 of *LNAI*, pages 147–161, Berlin, 1999. Springer.
19. J. Sefranek. A kripkean semantics for dynamic logic programming. In *Logic for Programming and Automated Reasoning(LPAR'2000)*. Springer Verlag, LNAI, 2000.
20. SMODELS. The SMODELS system, 2000. Available at `http://www.tcs.hut.fi/Software/smodels/`.
21. A. Tarski. A lattice–theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
22. XSB-Prolog. The XSB logic programming system, version 2.6, 2003. xsb.sourceforge.net.
23. Y. Zhang and N. Y. Foo. Updating logic programs. In Henri Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, 1998.