Updates plus Preferences*

José Júlio Alferes and Luís Moniz Pereira

Centro de Inteligência Artificial, Fac. Ciências e Tecnologia, Univ. Nova de Lisboa, P-2825-114 Caparica, Portugal, jja,lmp@di.fct.unl.pt
Voice:+351 21 294 8533, Fax: +351 21 294 8541

Abstract. The aim of this paper is to combine, into a single logic programming framework, the hitherto separate forms of reasoning of preferences and updating. More precisely, we define a language capable of considering sequences of logic programs that result from the consecutive updates of an initial program, where it is possible to define a priority relation among the rules of all successive programs. Moreover, within the framework, the priority relation can itself be updated.
In order to define a declarative semantics for the language, we start by reviewing the declarative semantics of updates of [1], and by presenting a definition of a semantics for preferences, shown equivalent to the one in [5], in a form suitable for its integration with the updates one.
Before the conclusions and mention of future work, we present two illustrative examples of application of the framework.

1 Introduction

In recent times, there has been a spate of work on reasoning with preferences and also, but separately, another spate of work on knowledge updating, both of which in the logic programming context. This interest has followed in the wake of a more general examination of flexible and dynamic forms of non-monotonic reasoning within artificial intelligence (AI). The present writing aims at combining these two heretofore separate forms of reasoning, preferring and updating, again in the purview of logic programming. We shall show how they complement each other, in that preferences select among pre-existing models, and updates actually create new models. Moreover, preferences may be enacted on the results of updates, and updates may be pressed into service for the purpose of changing preferences.

Forms of preference which have been intensely studied include specificity in taxonomic defaults, authority as well as temporal overriding in legal reasoning, priority of effect rules over inertia rules in causal reasoning, more likely faults in model-based diagnosis, preferred configurations in system synthesis, and scenario considerations in decision making. Many prioritized versions of existing non-monotonic formalisms have, already for some time, been developed, namely for circumscription, for hierarchical auto-epistemic logic, for default logic, for belief revision, and for abduction. In the case of logic programming (LP), research on the topic of preferences is much more recent. Cf. [4,5] for additional motivation, comparisons, applications, and references. Here, we expressly adopt the stable models

* Work partly supported by PRAXIS XXI project #2/2.1/TIT/1593/95 MENTAL.

based semantic framework of Brewka and Eiter [5], though replacing it with an equivalent formulation to bring it in line with our own stable models based update framework, with which we enmesh it. Another paramount reason for this choice of preference semantics are the two desirable principles (cf. Section 3) and the properties that semantics obeys, as spelled out by their authors.

In what concerns updates, its significance for AI has long been the object of much study [16,11,8]. In the LP setting, the accomplishments in this topic have likewise been garnered at a much later date [1,6,13,15,14,17]. Herein we adopt the stable models based update framework of [1] for the purpose of expanding it with the aforesaid preferences one. Sample prototypical applications of LP updates have included legal knowledge evolution [2], modelling of actions [3], taxonomic inheritance [6], and software development.

Preferences and updates are different forms of reasoning and serve different goals and applications. Preferences are used along with incomplete knowledge, when this is modeled with default rules. In such a setting, due to the incompleteness of the knowledge, several models may be possible. Preferences act by choosing among those possible models. A classical example is the birds-fly problem, where the incomplete knowledge contains the rules that birds normally fly and penguins normally don't. Given an individual which is both a penguin and a bird, two models are possible: one, using the one rule, where the individual flies; another, using the other more specific rule, where it doesn't. Preferences among rules can then be used to choose which one.

Updates are used to model dynamically evolving worlds. The problem arising here being, given a piece of knowledge describing the world, and given a change in the world (be it a rule or fact), how to modify the knowledge to cope with that change. The knowledge may itself be complete or incomplete: that's not the key issue in updates; rather, the key issue is about the process of accomodating, in the represented knowledge, any changes in the world. In this setting it may well happen that change in the world contradicts previous knowledge, i.e. the union of the previous knowledge with the representation of the new knowledge has no model. It is up to updates to remove from the prior knowledge representation a piece that changed, and to replace it by the new one. In this respect, mark well the distinction between update and revision of the knowledge, well broughtout e.g. in [16]. Whereas in the former knowledge changes due to changes in the world, in the latter incomplete knowledge is changed due to additional information (further completing the knowledge) about a static world view. These processes are different, and lead to different results. For example, suppose that your knowledge consists of a single rule stating that you have a flight booked for London, that is either for Heathrow or for Gatwick. If new information, stating that it is not for Heathrow, arrives thereby completing this knowledge (e.g. a call from your travel agency, clarifying this issue), then you should conclude that the flight is booked for Gatwick. If, the same information ($\neg Heathrow$) arrives due to a change in the world (e.g. you heard on the radio that all flights for Heathrow have been cancelled), then you should not conclude now that your flight is booked for Gatwick.

One way to look at revision is to consider any prior rules as defeasible, add the new knowledge to the previous one, and assign preference to this new knowledge over the old one (revision as chronological preference). This stance is justifiable in revision: our knowledge is incomplete; to make it less incomplete, when some new information arrives it should be given preference over the previous. But a similar rationale makes little sense in updates. Suppose that at some point we know that normally quakers are pacifists, and that the republican Nixon is a quaker. Forthwith we can conclude that Nixon is a pacifist. Now something happens in the world so that republicans tend to be belicists. A new rule, stating that normally republicans are belicists, is added as an update. What should we conclude about Nixon? In our opinion, nothing different from a situation where both rules are given at the same time: for Nixon, there is a conflict, and two models exist - one where he is considered pacifist, and the other where he isn't. It may well happen that, given the conflict among such defeasible rules in our incomplete knowledge, one may want to give preference to the quakers-pacisfist rule over the other rule.

In many real applications one is bound to have just incomplete knowledge about the world, default rules, and may want to be able to deal with a dynamically evolving world, where these rules may change in time. In such a situation preferences may be needed to choose among various possible models of the world, whereas updates are needed to deal with the knowledge on the evolution of the world. In this evolution, preferences themselves may change in time. To cope with this, a combination of both reasoning forms into a single framework is needed.

Consider the following example, where default rules as well as preferences change over time, which requires a combination of preferences and updates, including the updating of preferences themselves.

Example 1 (A sad story). (1) In the initial situation I am living and working everyday in the city. (2) Next, as I have received some monies, I conjure up other, alternative but more costly, living scenarios, namely travelling, settling up on a mountain, or living by the beach. And, to go with them, also the attending preferences, but still in keeping with the work context, namely that the city is better for that purpose than any of the new scenarios, which are otherwise incomparable amongst themselves. (3) Consequently, I decide to quit working and go on vacation, supported by my increased wealth, and hence to define my vacation priorities. To wit, the mountain and the beach are each preferable to travel, which in turn gainsays the city. (4) Next, I realize my preferences keep me all the while undecided between the mountain and the beach, and opt for the former. (5) Forthwith, I venture up the mountain, only to become ill on account of the height, and a physician advises me against too much sun exposure, be it at the mountain or the beach level. (6) So, I update my knowledge regarding health, and my concomitant priorities, and thus travel becomes *the* choice par excellence. (7) I finally run out of money for travel and return, still ill, to the city, cannot work, and continue my sad vacation there.

Despite their differences, the preference and the update LP approaches we adopt are also similar, in that both can be envisaged as wiping out rules. In the preference setting, one wipes out less preferred rules in order to select only some among the available stable models. In the update setting, one wipes out rules that are overruled by new rules, thereby engendering new models, including cases when there were none before the update took place. Looking at both in a similar way facilitates their coming together under one same framework. For preferences it makes all the sense to employ some (strict) but partial order on rules, for there are cases where one wishes to allow incomparable rules to defeat but not wipe out one another. For updates, a linear temporal order is employed, and alternative results may be obtained via distinct but nevertheless linear updating sequences, to produce a tree. A root node always exists, if need be the initial empty program.

The sequel is organized as follows. First, we recap the fixpoint semantics of updates, which relies on erasing rules rejected by an update. Second, we define a fixpoint semantics for preferences which resorts to erasing unpreferred rules. Third, on the basis of these, we proffer a joint fixpoint semantics for both updates and preferences. Finally, conclusions and future work are brought out.

2 Dynamic Logic Programs

In this section we recall the framework of Dynamic Logic Programming (DLP) [1] that, as motivated above, can be used to model the evolution of logic program through sequences of updates.

To represent negative information in logic programs and their updates, DLP allows for the presence of default negation in rule heads. It is worth noting why, in the update setting, generalizing the language to allow default negation in rule heads (thus defining "generalized logic programs") is more adequate than introducing explicit negation in programs (both in heads and bodies). Suppose we are given a rule stating that A is true whenever some condition $Cond$ is met. This is naturally represented by the rule $A \leftarrow Cond$. Now suppose we want to say, as an update, that A should no longer be the case (i.e. should be deleted or retracted), if some condition $Cond'$ is met. How to represent this new knowledge? By using extended logic programming (with explicit negation) this could be represented by $\neg A \leftarrow Cond'$. But this rule says more than we want to. It states that A is false upon $Cond'$, and we only want to go as far as to say that the truth of A is to be deleted in that case. All is wont to be said is that, if $Cond'$ is true, then $not\, A$ should be the case, i.e. $not\, A \leftarrow Cond'$. As argued in [10], the difference between explicit and default negation is fundamental whenever the information about some atom A cannot be assumed to be complete. Under these circumstances, the former means that there is evidence for A being false, while the latter means that there is no evidence for A being true. In the deletion example, we desire the latter case.

Note, however, that the adequacy of generalized logic program for this purpose is in facilitating the intuitive writing of updates. Indeed, as proven in [7], generalized logic programs and extended logic programs have the same expressive power.

Definition 1 (Generalized logic program). *A generalized logic program in the language $\mathcal{L}$ is a finite or infinite set of ground rules r of the form:*

$$L_0 \leftarrow L_1, \ldots, L_n. \qquad n \geq 0$$

where each L_i is a literal in $\mathcal{L}$ (i.e. an atom or a default literal $not\, A$ where A is an atom). By $head(r)$ we mean L_0, by $body(r)$ the set of literals $\{L_1, \ldots, L_n\}$, by $body_{pos}(r)$ the set of all atoms in $body(r)$, and by $body_{neg}(r)$ the set of all default literals in $body(r)$. We refer to $body_{pos}(r)$ as the prerequisites of r. Whenever L is of the form $not\, A$, $not\, L$ stands for the atom A.

The semantics of generalized logic programs is then defined as a generalization of the stable models semantics [9]. Before advancing the generalized definition, let us sketch, as a first step, a definition equivalent to the stable models semantics for the case of normal logic programs.

In the fixpoint operator $\Gamma(M)$ of the stable models semantics, first one deletes every program rule whose body contains some $not\,A$ where $A \in M$, and deletes too, from rule bodies every literal $not\,A$ such that $A \notin M$. The least model of the so obtained program is then computed. In its stead, one may take default literals in rule bodies as new propositional variables, add a fact $not\,A$ for every $A \notin M$, and then compute the least model of the resulting definite program. It is easy to check that the resulting set of atoms, not of the form $not\,A$, will be exactly the same as in $\Gamma(M)$. Moreover, for every fixpoint of $\Gamma(M)$, $A \notin M$ iff all rules of the program with head A have a false body in M. Thus, if one is only interested in fixpoints, instead of adding facts $not\,A$ for every $A \notin M$, one may add $not\,A$ for just every A having no rule with a true body in M. This approach views stable models as deriving $not\,A$ for every atom A which is not "supported" in the program by the model.

Now, since one can have default literals in rule heads, there are more ways of deriving them. But the previous one remains, i.e. if for some A there is no rule for A whose body is true, then $not\,A$ should be the case. This is the basic intuition behind the definition of stable models for generalized programs: given a model M, first add facts $not\,A$ for every A with no rule with true body in M; M is a stable model if the least model obtained after such additions coincides with M, where M has been enlarged with new propositional variables $not\,A$ for every $A \notin M$.

Definition 2 (Default assumptions). *Let M be a model of P. Then:*

$$Default(P, M) = \{not\,A \mid \nexists r \in P : head(r) = A \wedge M \models body(r)\}$$

Definition 3 (Stable Models of Generalized Programs). *A model M is a stable model of the generalized program P iff $M = least(P \cup Default(P, M))$*

It is easy to check that, for normal programs, this definition is equivalent to the original definition of stable models [9]. As shown in [1], it also coincides with the semantics presented in [12] when the latter is restricted to the language of generalized programs.

In DLP, sequences of generalized programs $P_1 \oplus \ldots \oplus P_n$ are given. Intuitively 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. In such a view, dynamic logic programs are to be used in knowledge bases that evolve. New rules (coming from new, or newly acquired, knowledge) can be added at the end of the sequence, bothering not whether they conflict with previous knowledge. The rôle of dynamic programming is to ensure that these newly added rules are in force, and that previous rules are still valid (by inertia) as far as possible, i.e. they are kept for as long as they do not conflict with newly added ones.

The semantics of dynamic logic programs is defined according to the rationale above. Given a model M of the last program P_n, start by removing all the rules from previous programs whose head is the complement of some later rule with true body in M (i.e. by removing all rules which conflict with more recent ones). All other persist through by inertia. Then, as for the stable models of a single generalized program, add facts $not\,A$ for all atoms A which have no rule at all with true body in M, and compute the least model. If M is a fixpoint of this construction, M is a stable model of the sequence up to P_n.

Other possible views on and usage of DLP, justify slight generalizations of the above informally described language and semantics. In general, the distinguished programs represent knowledge true at some state s, where different states may stand for different stages of knowledge in the linear evolution of the knowledge base (as above), but also for different time points in possible future evolutions of the knowledge, or even for knowledge of ever more specific objects organized in a hierarchy. In the latter case, each program contains the rules that are specific to the object under consideration, and rules from programs above in the hierarchy are inherited just as long as they do not conflict with the more specific information (for more on this stance see [6]). These other views justify a tree-like structure of programs (rather than a sequence), and also that dynamic programs can be queried at any state, rather than only at the last one.

Definition 4 (Dynamic Logic Program). *Let S be an ordered set with a smallest element s_0 and with the property that every $s \in S$ other than s_0 has an immediate predecessor $s-1$ and that $s_0 = s - n$ for some finite n. Then $\bigoplus\{P_i : i \in S\}$ is a Dynamic Logic Program, where each of the P_is is a generalized logic program.*

Definition 5 (Rejected rules). *Let $\bigoplus\{P_i : i \in S\}$ be a Dynamic Logic Program, let $s \in S$, and let M be a model of P_s. Then:*

$$Reject(s, M) = \{r \in P_i \mid \exists r' \in P_j,\ \ head(r) = not\, head(r') \wedge\ i < j \leq s \wedge M \models body(r')\}$$

To allow for querying a dynamic program at any state s, the definition of stable model is parameterized by the state:

Definition 6 (Stable Models of a DLP at state s). *Let $\bigoplus\{P_i : i \in S\}$ be a Dynamic Logic Program, let $s \in S$, and let $\mathcal{P} = \bigcup_{i \leq s} P_i$. A model M of P_s is a stable model of $\bigoplus\{P_i : i \in S\}$ at state s iff:*

$$M = least([\mathcal{P} - Reject(s, M)] \cup Default(\mathcal{P}, M))$$

It is clear from the definitions that stable models of dynamic programs are a generalization of stable models of generalized and normal programs, i.e. if the dynamic program consists of a single generalized (resp. normal) program then its semantics is the same as that of the stable models of generalized (resp. normal) programs. It is also shown in [1] that dynamic logic programs generalize the interpretation updates of [13].

In [1] a transformational semantics for dynamic programs is also presented. According to this equivalent definition, a sequence of programs is translated into a single generalized program (with one new argument added to all predicates) whose stable models are in one-to-one correspondence with the stable models of the dynamic program. This transformational semantics is the basis of an existing implementation of dynamic logic programming[1].

[1] Publicly available from: `http://centria.di.fct.unl.pt/~jja/updates/`

3 Preferred Stable Models

In this section we recall the preferences approach of [5], and set forth a definition of preferred stable models for generalized logic programs (rather than for extended logic programs as in [5]) in a form suitable for integration with the above described updates. In [5], logic programs are supplied with priority information, given in the form of a strict partial ordering on program rules[2].

Definition 7 (Prioritized generalized logic program). *Let P be a generalized logic program and let $<$ be a strict partial order over the rules of P, where $r_1 < r_2$ means r_1 is preferred to r_2. Then $(P, <)$ is a prioritized generalized logic program*[3].

Intuitively, the priority information is used to prefer among the various stable models of the program. The question here is what stable models to prefer in the face of a given priority relation among rules. To respond to this question, the authors in [5] start by formulating two principles all preference system should satisfy. The first (*Principle I*), is envisaged as a minimal requirement for preference handling, and states that if a stable model M_1 is generated by a set of rules[4] $R \cup \{r_1\}$, and another stable model M_2 is generated by $R \cup \{r_2\}$, where $r_1, r_2 \notin R$, then, if $r_1 < r_2$, M_2 cannot be preferred. The second (*Principle II*), captures a notion of relevance. It affirms that adding a rule which is not applicable in a preferred model can never render this model unpreferred.

With these two principles in mind, [5] defines a criterion for preferring among stable models, given a priority relation on rules. Their basic idea is that a stable model M can only be preferred if, for each rule in the program, whenever its (positive) prerequisites are true in M and its head is false in M, then there must be some *not* A in its body which is false in M, and there is a more preferred rule generating A. I.e. for a rule with true prerequisites not to be applied, there must be a more prioritary rule preventing its application.

Before presenting our equivalent definition of preferred stable models, let us first briefly review the formal definition of preferred answer sets of [5] specialized for the particular case where the program is ground. A preferred answer-set is a model of the program simultaneously satisfying two conditions: it must be a stable model (i.e. be a fixpoint of the Γ Gelfond-Lifschitz operator); it must satisfy a fixpoint equation which, intuitively, guarantees that the rules are being applied in observance of the partial order, i.e. that the criterion described above is met.

Adopting the view that rules are applied one at a time, a partial ordering on rules should be viewed as a representative of all its possible refinements into total orderings. These, defined in [5], are dubbed *full prioritizations* of prioritized programs. A program is said *fully prioritized* if it coincides with its single full prioritization. The fixpoint construction guaranteeing that rules of a fully prioritized

---

2 For a comparison with approaches ordering atoms rather than rules see [5].

3 Note that, in contradistinction to [5], our priority relation is defined for ground programs. To define the relation directly on non-ground programs, the methodology given in [5], using well-orderings, could just as well be applied to our case. However, for simplicity, we will not consider it in this paper.

4 The set of rules that generate a stable model is made up of all the rules in the program whose body is true in the stable model.

program are applied in the correct order is carried out in two steps. First, all (positive) atoms in the body are preprocessed away on the basis of their truth value in the model. More precisely, this so called dual Gelfond-Lifschitz reduction ${}^{M}\mathcal{R}$ is obtained from $\mathcal{R}$ by first deleting every rule having a prerequisite A such that $A \notin M$, and then removing from the remaining rules all prerequisites. All bodies of rules now exhibit only default literals.

The correct order of applying rules is then checked in the thus obtained prerequisite-free program. Informally, this is achieved by, following rule order, adding the heads of those rules that are not defeated by a rule having higher priority (whose head has been added). Formally:

Definition 8 (Defeating of rules). *A rule r is defeated by a set of literals S iff $\exists\ not\, A \in body(r) : A \in S$.*

Definition 9 ($C_{\mathcal{R}}$ operator [5]). *Let $\mathcal{R} = (P, <)$ be a prerequisite-free fully prioritized logic program, and let M be a set of ground literals. $C_{\mathcal{R}}(M)$ is the least fixpoint of the sequence S_α (where α ranges over the rules of the fully prioritized P, according to their (total) ordering):*

$$S_\alpha = \begin{cases} \bigcup_{\beta<\alpha} S_\beta & \text{if } r_\alpha \text{ is defeated by } \bigcup_{\beta<\alpha} S_\beta \text{ or} \\ & r_\alpha \text{ is defeated by } M \text{ and } head(r_\alpha) \in M; \\ \bigcup_{\beta<\alpha} S_\beta \cup \{head(r_\alpha)\} & \text{otherwise} \end{cases}$$

Definition 10 (Preferred Answer Set). *Let $\mathcal{R} = (P, <)$ be a prioritized logic program and let $\mathcal{R}_f = (P, <_f)$ be a full prioritization of $\mathcal{R}$. A model M of P is a preferred answer set of $\mathcal{R}$ iff $M = \Gamma_P(M)$ and $M = C_{{}^{M}\mathcal{R}_f}(M)$.*

As motivated in the Introduction, and in order to facilitate the capture of both preferences and updates in one single framework, it is our goal in this section to devise a declarative semantics for prioritized generalized programs based on the removal of (less preferred) rules, inasmuch our update framework hinges likewise on the removal of rules; this maneuver is crucial for fusing the two. Moreover, we require this semantics to coincide with the one in [5] on normal programs. The main issue in so doing rests in determining criteria for which rules to remove, in order to obtain exactly the same semantics. Before presenting its definition, we begin by reporting, with small but illustrative examples, on the problems involved in finding them[5]. Like in [5], we start with the case of prerequisite-free programs.

Example 2. Consider the program: (1) $a \leftarrow not\, b$ (2) $b \leftarrow not\, a$, where rule (1) is preferred over rule (2). Its stable models are $M_1 = \{a\}$ and $M_2 = \{b\}$, the preferred one being M_1. Intuitively, since (1) $<$ (2) and the head of rule (1) defeats (2), in order to obtain the preferred stable model, one should remove rule (2). Indeed, M_1 is the single stable model of the program after the excision.

Mark that the reasoning brought out in this example concurs with the definition of the $C_{\mathcal{R}}$ operator. According to it, the head of a rule is not added if the rule

[5] This account is important here because for lack of space, the proof of equivalence with [5] does not fit. The problems depicted below form the core issues dealt with by the proof.

is defeated by the previously constructed set. But this set is formed precisely by the heads of the more preferred rules. Instead of not adding the head to the set, the same effect can be achieved by removing the rule, i.e. by removing all rules defeated by the head of a more preferred rule, which has not itself in turn been removed.

Example 3. Consider now the program:

(1) $b \leftarrow not\, c$ (2) $c \leftarrow not\, d$ (3) $a \leftarrow not\, b$ (4) $b \leftarrow not\, a$

where a rule (i) is preferred over rule (j) iff $i < j$. Its stable models are $M_1 = \{a, c\}$ and $M_2 = \{b, c\}$. According to Principle I above, since M_1 is generated by rules (2) and (3), and M_2 by rules (2) and (4), M_2 should not be preferred. But, resorting to the reasoning explained above, rule (3) is removed (as it is defeated by the head of rule (1)), and the only stable model of the resulting program becomes M_2. Why shouldn't rule (1) remove (3)? Because rule (1) is defeated in whichever model. This is in line with Definition 9 (2nd line of S_α) where heads of rules true in the model, whose body is defeated by the model, are not added to the set. Accordingly, given some model, all such rules are removed. Hereafter, we refer to them as "unsupported rules".

Consequently, in model M_2 rule (1) is removed, as well as rule (4) (the latter is defeated by the head of the more preferred and non-removed rule (3)). And M_2 is not a stable model of the program after those rules are withdrawn.

The two above criteria for deleting rules (viz. deleting less preferred rules defeated by the head of some more preferred rule, and deleting "unsupported rules") concur with the definition of the $C_\mathcal{R}$ operator. However, as evidenced by the example below, they are not enough.

Example 4. Consider now: (1) $a \leftarrow not\, b$ (2) $b \leftarrow not\, c$ $(1) < (2)$, whose only stable model is $M = \{b\}$, which according to [5] is not preferred. This is so because rule (1) is neither unsupported (a is not true in M) nor defeated by a more preferred rule, so a is added in the construction of $C_\mathcal{R}(M)$, and M cannot thereafter be a fixpoint of the operator. However, using only the two above criteria none of these two rules is eliminated, and M would be preferred.

To obtain the effect achieved by [5], one must guarantee that, in spite of rule removal, a is enforced in the preferred models of the reduced program. This is accomplished by removing any rules less preferred than the one for a, which, if otherwise were not removed, would cause a not to be in the preferred models. In other words, one is required to remove all rules having true body in the model, whose heads defeat a more preferred rule. Mark well that if the body of the less preferred rule is not actually true in the model, then the defeating is only a potential but not effective one, and the rule must not be eliminated. Indeed, its preservation will permit it to defeat, and cause to remove, rules less preferred than itself even if they attack it. When considering programs with prerequisites, one must further insist that the more preferred rule is not deleted by the dual reduct transformation. This is ensured by verifying that the positive part of the body of the more preferred rule is actually true in the model. A similar reasoning applies to the other two criteria explained above.

These three criteria suffice for formalizing, in Definition 12, the set of unpreferred rules. To ease in the definition, we first define:

Definition 11 (Unsupported rules). *Let M be a model of P. Then:*

$$Unsup(P, M) = \{r \in P : M \models \{head(r)\} \cup body_{pos}(r) \wedge M \not\models body_{neg}(r)\}$$

Definition 12 (Unpreferred rules). *Let M be a model of P. The set of unpreferred rules, $Unpref(P, M)$, is the least set of rules that includes $Unsup(P, M)$, and every r in P such that:*

$$\exists r' \in P - Unpref(P, M) : r' < r \wedge M \models body_{pos}(r') \wedge \\ [not\, head(r') \in body_{neg}(r) \quad \vee \quad (not\, head(r) \in body_{neg}(r') \wedge M \models body(r))\ ]$$

Lack of space prevents us from showing that such a least set always exists. Indeed, it can be contructed by iterating the definition of unpreferred rule according to rules' ordering, starting from the set of unsupported rules.

For programs with positive atoms in rule bodies, the effect of the dual reduction operation of [5] is obtained by adding to the program facts $not\, A$ for every A with no rule in the original program with true body in the model, and thereafter computing the least model.

Definition 13 (Preferred Stable Models). *A model M of program P is a preferred stable model of the prioritized generalized program $(P, <)$ iff:*

$$M = least([P - Unpref(P, M)] \cup Default(P, M))$$

This guarantees that the preferred models obtained after removing all unpreferred rules are also stable models of P, and so only one fixpoint equation is needed in this definition, as desired. Verily:

Proposition 1. *Let M be a preferred stable model of $(P, <)$. Then M is also a stable model of P, i.e. $M = least(P \cup Default(P, M))$.*

Now, as expected, as this was one of our primary goals for the definition of preferred stable models, in programs where both the preferred answer sets of [5] and our preferred stable models can be applied (i.e. in normal programs), their results coincide. For an extensive study of the properties of preferred answer-sets, its intuitions, examples, and comparisons with related approaches see [5].

Theorem 1. *Let P be a ground normal logic program, and let $<$ be a strict partial order over the rules of P. M is a preferred stable model of $(P, <)$ iff M is a preferred answer-set of $(P, <)$ in the sense of Brewka and Eiter [5].*

4 Updating logic programs with preferences

Having separately defined both updates and preferences in an analogous way, based on the removal (or rejection) of rules, in this section we combine both concepts into an unified framework. Moreover, as motivated in the Introduction, the combined framework must also allow for the updating of the priority relation itself.

Leaving, for now, the issue of updating the priority relation, we must consider sequences of generalized programs $P_1 \oplus \ldots \oplus P_n$, viewed as sequences of updates of an original program, plus some priority relation among rules. One first basic question is in order: where to define the priority relation? Among the rules for the same program? Or among rules in the union of all programs in the sequence? More formally, should there be a strict partial order $<_i$ for each of the P_i in the sequence, or should there be a single strict partial order $<$ defined over the rules of $\bigcup_{i \in S} P_i$? Clearly, the latter approach is more general than the former: it does not prevent limiting the priority relation to rules in the same P_i, while the former does prevent priority relations between rules from different P_is. Furthermore, the extra generality is useful. For instance, in the situation of Example 1, one may want to say at a given state that I go to the beach unless I go to the mountain, and later say that I go to the mountain unless I go to the beach, and establish a priority over these rules. Note that the rules were introduced at different update stages, and so the priority relation is to be established between rules of different P_is. Accordingly, in our framework we consider a single priority relation defined on the rules of $\bigcup_{i \in S} P_i$, which can evolve as new rules are introduced.

To cope with the possibility of updating the priority relation, it cannot be fixed. Rather it must be described in some language that allows for the possibility of its evolution, via updates. One such language is precisely DLP and, for uniformity, that is what is used in our framework. Thus, instead of a sequence of programs representing knowledge, we have a sequence of pairs: of programs representing knowledge, and of programs describing the priority relation among rules of the knowledge representation. In general, an update of the priority relation may depend on some other predicate (e.g. in Example 1, I may want to say that, if I have to work, then I prefer the rule advising me to stay in the city). To permit this generality, we allow rules in programs describing the priority relation to refer to predicates defined in the programs that represent knowledge.

Definition 14 (Dynamic Prioritized Programs). *Let $\mathcal{P} = \{P_s : s \in S\}$ be a dynamic logic program whose alphabet does not contain the strict partial order arity 2 predicate symbol $<$, and let $\mathcal{R} = \{R_s : s \in S\}$ be another dynamic logic program whose alphabet contains at least the predicate symbol $<$, and whose sets of constants includes all the rules in the union of all Ps in $\mathcal{P}$. Then $\bigoplus\{(P_s, R_s) : s \in S\}$ is a Dynamic Prioritized Program.*

Given the very deliberate definition forms of the semantics of preferences and of updates, it is not difficult to combine both in a single one, as per the above delineated framework. Given a model M of the last program in the sequence (or, in the general setting, of the program state we want to query), for testing stability we have first to remove all the rejected rules according to updates, and then all the unpreferred rules according to preferences. Note, however, that if both sets of rules (rejected and unpreferred) were removed simultaneosly, then a rule which is rejected by an update, might serve for unprefering some other rule. This would lead to counterintuitive results. In fact, updates have precedence over preferences. If a previous rule is invalidated by a subsequently introduced rule, then the former should no longer be available in the preferences setting. Accordingly, the set of unpreferred rules must be determined on the basis of the program obtained after removing those rules rejected by any updates. In general, the union of all programs in the sequence may be inconsistent, and it would make no sense to apply preferences to this inconsistent set of rules; updates are applied first (by rejecting rules) and allow you to come up with a consistent set of rules; preferences then intervene to choose among the various models of that consistent set of rules.

Since the priority relation is itself defined by the dynamic prioritized program, models must also take into account the $<$ predicate, i.e. one has to entertain models of the union of P_n with R_n. Moreover, in the definition of unpreferred rules, the priority relation must be checked in regard to the model under consideration:

Definition 15 (Unpreferred rules). *$Unpref(P, M)$ is the least set of rules including $Unsup(P, M)$ and rules r in P such that:*
$\exists r' \in P - Unpref(P,M) : M \models r' < r \wedge M \models body_{pos}(r') \wedge$
$[not\, head(r') \in body_{neg}(r) \quad \vee \quad (not\, head(r) \in body_{neg}(r') \wedge M \models body(r))\ ]$

In the definition of preferred stable model, it is crucial that the priority relation be a strict partial order (i.e. irreflexive and transitive). In our framework, since the user can write any rules for describing predicate $<$, it may well happen that its extention be a relation not complying with those properties. The definition of the semantics must prevent this being the case, i.e. must only consider models where the extension of predicate $<$ is indeed a strict partial order. Thus:

Definition 16 (Preferred Stable Models at state s). *Let $\bigoplus\{(P_i, R_i) : i \in S\}$ be a Dynamic Prioritized Logic Program, let $s \in S$, and let $\mathcal{PR} = \bigcup_{i \leq s}(P_i \cup R_i)$. A model M of $P_s \cup R_s$ is a preferred stable model at state s iff:*
- $\forall r : (r < r) \notin M$
- $\forall r_1, r_2, r_3 : (r_1 < r_2) \in M \wedge (r_2 < r_3) \in M \Rightarrow (r_1 < r_3) \in M$
- $M = least(\ [\mathcal{PR} - Reject(s, M) - Unpref(\mathcal{PR} - Reject(s,M), M)] \cup \ Default(\mathcal{PR}, M)\ )$

This definition makes it clear that dynamic prioritized programs generalize both dynamic logic programs and prioritized logic programs. In fact, if all the R_is are empty, then Definition 16 is clearly equivalent to Definition 6. And if there is a single pair (P, R) in the sequence, then Definition 16 is equivalent to Definition 13, the priority relation of the prioritized program being the least model of R.

We now illustrate the overall framework with two examples:

Example 5. The first 4 stages in the "sad story" of Example 1 can be modelled by the dynamic prioritized program $(P_1, R_1) \oplus \ldots \oplus (P_4, R_4)$ (where, for simplicity, we adopt unique numbers for rules, instead of the rules themselves in the priority relation, and where c stands for "living in the city", mt for "settling on a mountain", b for "living by the beach", t for "travelling", wk for "work", vac for "vacations", and mo for "possessing money"):

$$
\begin{array}{llll}
P_1 : & (1)\ c \leftarrow not\, mt, not\, b, not\, t & R_1 : & X < Y \leftarrow X < Z, Z < Y \\
 & (2)\ wk \leftarrow & & \\
 & (3)\ vac \leftarrow not\, wk & & \\
 & & & \\
P_2 : & (4)\ mt \leftarrow not\, c, not\, b, not\, t, mo & R_2 : & (1) < (4) \leftarrow wk \\
 & (5)\ b \leftarrow not\, mt, not\, c, not\, t, mo & & (1) < (5) \leftarrow wk \\
 & (6)\ t \leftarrow not\, mt, not\, b, not\, c, mo & & (1) < (6) \leftarrow wk \\
 & (7)\ mo \leftarrow & &
\end{array}
$$

$$
\begin{array}{lrl}
P_3 : (8)\ not\, wk \leftarrow & R_3 : & (4) < (6) \leftarrow vac \\
 & & (5) < (6) \leftarrow vac \\
 & & (6) < (1) \leftarrow vac
\end{array}
$$

$$
P_4 : \quad \{\} \qquad\qquad R_3 : (4) < (5)
$$

For example, the only preferred stable model at state 4 is:

$$\{mt, vac, mo, (4) < (5), (4) < (6), (4) < (1), (5) < (6), (5) < (1), (6) < (1)\}$$

and the preferred stable models at state 3 are two:

$$\{mt, vac, mo, (4) < (6), (4) < (1), (5) < (6), (5) < (1), (6) < (1)\}$$
$$\{b, vac, mo, (4) < (6), (4) < (1), (5) < (6), (5) < (1), (6) < (1)\}$$

Note in this example how the inertia of the transitivity rule (added in R_1) enforces transitivity on the priority relation in all the subsequent states.

Example 6. Consider the following situation (adapted from an example of qualitative decision making in [5]). You want to buy a car and, for that purpose, you have collected the following information about different types of cars: $fast(chevrolet)$, $expensive(chevrolet)$, $safe(chevrolet)$, $safe(volvo)$, and $fast(porsche)$. Let's assume you like fast cars, and your budget does not allow you to purchase an expensive one. Moreover, you cannot afford more than one car.

This situation can be modelled by P_1 which, besides the facts above, contains[6]:

$$
\begin{array}{lrl}
(1) & not\, buy(X) & \leftarrow avoid(X) \\
(2) & avoid(X) & \leftarrow not\, buy(X), expensive(X) \\
(3) & buy(X) & \leftarrow not\, avoid(X), fast(X) \\
(4) & avoid(Y) & \leftarrow fast(X), buy(X), Y \neq X
\end{array}
$$

See [5] for an explanation on how to come up with this program given the described situation, in particular the need for rule (4) in modelling the fact that you may not buy two cars[7].

Since there is not much you can do with your restricted budget, rule (2) has priority over rules (3) and (4). So $R_1 = \{(2) < (3), (2) < (4)\}$.

The reader can check that the only preferred stable model of (P_1, R_1) includes $\{buy(porsche), avoid(volvo), avoid(chevrolet)\}$, besides the facts and the priority relation, and you should buy the Porsche.

Now your "significant other" insists that you should consider buying a safe car. Moreover, as a gentleperson, you ascribe priority to your partner's suggestion. To assimilate this new information you update your knowledge with:

$$
\begin{array}{lrl}
P_2 : (5) & buy(X) & \leftarrow not\, avoid(X), safe(X) \\
(6) & avoid(Y) & \leftarrow safe(X), buy(X), Y \neq X
\end{array}
$$

and $R_2 = \{(5) < (3), (5) < (4), (6) < (3), (6) < (4), (2) < (5), (2) < (6)\}$. Now the only preferred stable model (at state 2) includes $buy(volvo)$, $avoid(porsche)$ and $avoid(chevrolet)$, and you should buy the Volvo instead.

[6] Rules with variables simply stand for their (finite) ground instances.

[7] In fact, the coding of this piece of knowledge by itself is not related to updates, and the rules above are just those present in [5] where $\neg buy(X)$ is here replaced by $avoid(X)$, and (1) encodes the relation between these two predicates.

Now suppose you discover Volvos are out of stock, and so you cannot buy one so soon. For that you add $P_3 = \{not\, buy(volvo)\}$, plus an empty R_3. With this new update, rule (5) is now rejected, and the only stable model at state 3 this time includes $\{buy(porsche), avoid(volvo), avoid(chevrolet)\}$.

5 Conclusions and future work

We have motivated the need for coupling preferences with updates, and shown how to accomplish it within the logic programming paradigm. We did so by devising a unified framework that combines the hitherto separate approaches to each aspect, and allows for preferences themselves to be updated. The framework coincides with [5] when a single program is given in the sequence, and with [1] when the preference relation is empty. Thus, for comparisons of this framework with others with preferences alone see [5], and for that with others with updates alone see [1].

To the best of our knowledge, [17] is the only work considering some combination of preferences and updates. However, the generality of the combination of both reasoning mechanisms in [17] is far from that of the present paper. In fact, [17]'s concern is with updates alone, and mainly considers the process of updating one program by another program, with mechanisms similar to those of [1] (i.e. removing rules from the initial program which "somehow" contradict rules from the update program, and retaining all others by inertia). Additionally, at the end, all rules from the update program are given preference over all retained rules of the initial program. No other preference ordering is considered there. And, as argued in the Introduction, updates alone do not necessarily force such preferences. In our framework, the user can state that more recent rules are preferred over older ones, but is also free to state differently. Moreover, in our framework the preference relation itself can be updated. The greater generality of our approach stems as well from our usage of [1] as the basis for updates. In fact, note that [1] considers arbitrary sequences of updates whereas [17] simply considers the update of one program by another. In [17] some semantical properties of their system are investigated. However, all such properties address only updating, and not some combination of it with preferences. It remains to be studied what generic principles any system combining preferences and updates (not necessarily in logic programming) should comply with. Those principles would help the comparison with [17] and possible other systems. Such generic study, and the verification of the principles in our framework, is work we are now developing.

Several other topics cry out for subsequent development. First, we are working on a transformational semantics of preferences into logic programs, to be coupled with the extant aforementioned one for updates. This will readily propitiate an implementation of the overall framework, as well as serve as a basis for the study of its computational properties.

An outstanding issue, on which some effort needs deploying, concerns how to automatically ensure irreflexivity and transitivity of the partial order, as it is being updated. For the moment this responsibility is wholly relegated to the updater. As it stands, in case of infringement there will simply be no model, as per Definition 16. It is in our plans to study the adequacy of the update mechanism on rules for predicate < so as to automatically guarantee irreflexivity and transitivity. In this respect, note in Example 5, how transitivity is always guaranteed by adding one rule to the initial program.

Finally, we also intend to explore application areas such as e-commerce, legal reasoning, and rational agents. They will certainly provide valuable opportunities and hints for the evolution of the topics broached in this paper.

References

1. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 2000. To appear. A short version titled *Dynamic Logic Programming* appeared in A. Cohn and L. Schubert (eds.), *KR'98*, Morgan Kaufmann.
2. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS – a language for updating logic programs. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR'99*. Springer, 1999.
3. J. J. Alferes, L. M. Pereira, T. Przymusinski, H. Przymusinska, and P. Quaresma. Preliminary exploration on actions as updates. In *AGP'99*, 1999.
4. G. Brewka. Well-founded semantics for extended logic programs with dynamic preferences. *Journal of Artificial Intelligence Research*, 4, 1996.
5. G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109, 1999. A short version appeared in A. Cohn and L. Schubert (eds.), *KR'98*, Morgan Kaufmann.
6. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *ICLP'99*. MIT Press, 1999.
7. C. V. Damásio and L. M. Pereira. Default negation in the heads: why not? In R. Dyckhoff et al., editor, *ELP'96*. Springer, 1996.
8. M. Dekhtyar, A. Dikovsky, S. Dudakov, and N. Spyratos. Monotone expansion of updates in logical databases. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR'99*. Springer, 1999.
9. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *ICLP'88*. MIT Press, 1988.
10. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *ICLP'90*. MIT Press, 1990.
11. H. Katsuno and A. Mendelzon. On the difference between updating a knowledge base and revising it. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR'91*. Morgan Kaufmann, 1991.
12. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *KR'92*. Morgan-Kaufmann, 1992.
13. V. Marek and M. Truszczynski. Revision specifications by means of programs. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *JELIA'94*. Springer, 1994.
14. T. Przymusinski and H. Turner. Update by means of inference rules. In V. Marek, A. Nerode, and M. Truszczynski, editors, *LPNMR'95*. Springer, 1995.
15. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR'99*. Springer, 1999.
16. M. Winslett. Reasoning about action using a possible models approach. In *AAAI'88*, 1988.
17. Y. Zhang and N. Foo. Updating logic programs. In H. Prade, editor, *ECAI'98*. Morgan Kaufmann, 1998.