Logic Programming Updating
- a guided approach -

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. In this work we review and synthesize, in a selective way, a series of recent developments concerning the dynamics of the evolution of logic programs by means of updates. We do so because this comparatively new and expanding area merits the attention of more researchers and more teachers alike, though there does not exist a single integrative source to induct them to the topic.

1 Introduction

Inasmuch we have accompanied the area of logic program updating from its inception, and contributed assiduously to its growth, we assumed ourselves in a good position to promote the topic and fill-in the absence and lack of a coherent self-contained exposition. The opportunity to do so is afforded by the present chapter-length work in honour of Bob Kowalski, who has done so much to promote logic programming as a whole. Note, however, that this is not a survey. It simply brings together at this juncture, within a uniform notation, continuity of exposition, and under the same 2-valued semantics, the marrow of a series of developments on the topic of logic program updates, which have been co-authored with others. A critical survey would require a much longer work, including the introduction to each of other authors' approaches and notation. Notwithstanding, the original papers we reference contain a number of comparative and critical remarks that the reader can follow up to that effect.

We begin at the beginning, by recapitulating the seminal work of [34] on revision programs (here dubed MT-revision-programs) to specify model updates, and go on to show how they can captured a program transformation, as a result of work by [6]. Next, we present the topic and issues of program updates, a generalization of model updates, show how they can specify the result of sequences of updates known as dynamic logic programs (DLPs) [4], and illustrate their applications. Subsequently, we introduce the language LUPS [8], devised for specifying update commands which produce DLPs, and exhibit its application in a number of domains. Finally, we proffer future perspectives on logic program updating, and mention ongoing work and implementations.

We are indebted to the co-authors of joint papers from which we have extracted or adapted much material, namely João Leite, Halina Przymusinska, Teodor Przymusinski, and Paulo Quaresma.

2 Model Updates

As the world changes so must programs that represent knowledge about it. When dealing with modifications to a knowledge base represented by a propositional theory, two kinds of abstract frameworks have been distinguished both by Keller and Winslett in [28] and by Katsuno and Mendelzon in [27]. One, theory revision, deals with incorporating new knowledge about a static world state. The other deals with changing worlds, and is known as theory update. In this work, we are concerned with theory update only, and, in particular, within the framework of logic programming.

Within the framework of logic programming, simple fact by fact updates have long been addressed [14, 23, 24]. Program updating is distinct from program revision, where a program accommodates, perhaps non-monotonically by revising assumptions, additional information about a world state. Work on logic programs revision (or contradiction removal) has received more attention (e.g. in [3, 5, 22, 40, 41]).

A key insight into the issue of updating theories is due to Winslett [39], who showed that, contrary to theory revision, one must consider the effect of an update in each of the states of the world that are consistent with our current knowledge of its state. The following realistic situation chisels the differences between program update and revision crisply.

Example 1. My secretary has just booked me on a flight from here to London on Wednesday but can't remember to which airport, Gatwick or Heathrow. Clearly, this statement can be represented by:

$$booked\_for\_gatwick \vee booked\_for\_heathrow$$

where propositions $booked\_for\_X$ mean that I have a booking for a Wednesday flight to airport X.

Now someone tells me there never are flights from here to Gatwick on Wednesday i.e., assuming that no one cannot have a booking for a non-existing flight, I'm told $\neg booked\_for\_gatwick$. I conclude that I'll be flying to Heathrow, i.e. *booked_for_heathrow*. This is knowledge revision. The state of the world hasn't changed with respect to the flight information, but on obtaining more information I have revised accordingly my knowledge about that state of the world.

Alternatively, I hear on the radio that all flights to Gatwick on Wednesday have been cancelled and, consequently, all possibly existing bookings for those flights have been withdrawn. In other words, the world changed such that now $\neg booked\_for\_gatwick$ holds. I'm at a loss regarding whether I still have a flight to London on Wednesday. This is knowledge update. The world of flights has changed, and refining my knowledge about its previous state is inadequate: I cannot conclude that I have a booking for a flight to Heathrow (*booked_for_heathrow*). I have obtained knowledge about the new world state but it doesn't help me to disambiguate the knowledge I had about its previous state. What I can do is pick up the phone and book a flight to Heathrow on Wednesday, on any airline. That will change my flight world and at the same time update my knowledge about that change. However, I'm now unsure whether I might not have two flights booked to Heathrow. But if my secretary suddenly remembers he had definitely booked me to Gatwick, then I will no longer believe I have two flights to Heathrow on Wednesday.

Accordingly, theory update is performed "model by model" [27,39], where a set of formulae T_U is a *theory update* of T, following an update request U, iff the models of T_U result from updating each of the models of T by U. Thus a theory update is determined by the update of its models.

The same idea can be applied to knowledge bases represented by logic programs: a program P_U is a *program update* of P, following an update request U, iff the models of P_U (according to some logic program semantics $\mathcal{S}$) are the result of updating each of the models of P (given by semantics $\mathcal{S}$) by U. So, to obtain P_U, first compute all models of P according to a given semantics $\mathcal{S}$; to each of these models apply the update request U to obtain a new set of models $\mathcal{M}$; P_U is then any logic program whose models are exactly $\mathcal{M}$.

In [10,34,35,37], the issue of program change via updating rules has been introduced. There, a new set of models is obtained by means of the update rules, from each of the models of the given program. Any program satisfying the new set of models will count as an update of the original program. However, no procedure, for obtaining such a single program whose model are the ones resulting from the update, is set forth by the cited authors. (Except in the trivial case where the original and final programs are just sets of facts [10,37].) Following [27,39], it is essential to start by specifying precisely how a program's models are to change, before even attempting to specify program change.

In this section, we begin with an overview of the work on model updating, and move on to present a correct transformation on normal programs which, from an initial program, produces another program whose models enact the required change in the initial program's models, as specified by the update rules.

2.1 Marek and Truszczyński's Revision Programs

In [34], the authors introduced a language for specifying updates to knowledge bases, which they called *revision programs*. Given the set of all models of a knowledge base, a revision program specifies exactly how the models are to be changed.[1]

To avoid confusion between the concept of program (or theory) revision, whose difference to updates are reviewed in the beginning of Section 2, and the name "revision programs" choosen by [34] to denote the programs that specify knowledge base updates, in the sequel we will call these *MT-revision-programs*.

The language of MT-revision-programs is quite similar to that of logic programming: MT-revision-programs are collections of *update rules*, which in turn are built of atoms by means of the special operators: $\leftarrow$, *in*, *out*, and ",". The first

[1] For more detailed motivation and background to this section of the present paper the reader is referred to [34, 37].

is an implication symbol, *in* specifies that some atom is added to the models, via an update, *out* that some atom is deleted, and the comma denotes conjunction.

Definition 1 (Update rules for atoms). *Let U be a countable set of atoms. An* update in-rule *or, simply, an* in-rule*, is any expression of the form:*

$$in(p) \leftarrow in(q_1), \ldots, in(q_m), out(s_1), \ldots, out(s_n) \tag{1}$$

where p, q_i, $1 \leq i \leq m$, and s_j, $1 \leq j \leq n$, are all in U, and $m, n \geq 0$.

An update out-rule *or, simply, an* out-rule*, is any expression of the form:*

$$out(p) \leftarrow in(q_1), \ldots, in(q_m), out(s_1), \ldots, out(s_n) \tag{2}$$

where p_i, q_i, $1 \leq i \leq m$, and s_j, $1 \leq j \leq n$, are all in U, and $m, n \geq 0$.

Intuitively, MT-revision-programs can be regarded as operators which, given some initial interpretation I_i, produce its updated version I_u.

Example 2. Consider a knowledge base with two models $I_1 = \{gatwick\}$ and $I_2 = \{heathrow\}$. The information that flights for Gatwick have been cancelled, can be represented by the MT-revision-program UP:

$$out(gatwick)$$

stating that in the resulting knowledge base, *gatwick* is to be deleted. We will see that this MT-revision-program, applied to I_1 and I_2 produces the two models: $\{\}$ and $\{heathrow\}$. In the first, both *gatwick* and *heathrow* are false, and in the latter, *gatwick* is false and *heathrow* is true. As desired, in both of them *gatwick* was deleted, and thus is false.

It is worth noting here some similarities between these update rules and STRIPS operators [18], in that both specify what should be added and what should be deleted from the current knowledge base. However, differently from STRIPS the preconditions of update rules may depend on the models of the resulting knowledge base. With STRIPS they may only depend on the models of the previous knowledge base.

Example 3. Let UP be the MT-revision-program:

$$\begin{array}{c} in(a) \leftarrow out(b) \\ in(b) \leftarrow out(a) \end{array}$$

and an initial knowledge base whose only model is $I_i = \{\}$, where both a and b are false . Intuitively, the first rule of UP, states that if b is not true in the resulting theory (after the update) then a must be added. The second states that if a is not true, then b must be added. Thus, there are two possible update interpretations: $I_u = \{a\}$ and $I_u = \{b\}$.

In this example, differently from STRIPS, the update in a knowledge base can be conditional on the truth or falsity of some atoms in the resulting models. However, the first example shows that there are some changes that are mandatory in a MT-revision-program. In it, removing $gatwick$ is not conditional on anything, and must be done in every resulting model. This notion of mandatory or necessary changes is formalized as follows:

Definition 2 (Necessary change). *Let P be a MT-revision-program with least model M. The* necessary change *determined by P is the pair (In_P, Out_P), where:*

$$In_P = \{a : in(a) \in M\}$$
$$Out_P = \{a : out(a) \in M\}$$

If $In_P \cap Out_P = \{\}$ then P is said coherent.

Intuitively, the necessary change determined by a program P specifies those atoms that must be added and those atoms that must be deleted, whatever the initial interpretation.

Example 4. Take the MT-revision-program $P = \{out(b) \leftarrow out(a);\ in(b);\ out(a)\}$. The necessary changes are irreconcilable (since b must be added, and simultaneously deleted) and P is incoherent.

To build a model of the resulting knowledge base, after the update specified by a MT-revision-program, necessary change must be considered. But, depending on the initial interpretation, other changes are in order. These changes are formalized as follows:

Definition 3 (Justified update). *Let P be a MT-revision-program and I_i and I_u two interpretations. The* reduct $P_{I_u|I_i}$ *with respect to I_i and I_u is obtained by the following operations:*

- *Removing from P all rules whose body contains some $in(a)$ and $a \notin I_u$;*
- *Removing from P all rules whose body contains some $out(a)$ and $a \in I_u$;*
- *Removing from the body of remaining rules of P all $in(a)$ such that $a \in I_i$;*
- *Removing from the body of remaining rules of P all $out(a)$ such that $a \notin I_i$.*

*Whenever P is coherent, I_u is a P-*justified update *of I_i if the following stability condition holds:* $I_u = (I_i - Out_{P_{I_u|I_i}}) \cup In_{P_{I_u|I_i}}$.

The first two operations delete rules which are useless given I_u. Due to stability, the initial interpretation is preserved as much as possible in the final one. The last two rules achieve this since any exceptions to preservation are explicitly dealt with by the union and difference operations in the two stability conditions.

Example 5. In example 2, note e.g. that $\{\}$ is a justified update of $\{gatwick\}$. In fact, the reduct operation does not change the MT-revision-program, and its necessary change is given by $In = \{\}$ and $Out = \{gatwick\}$. Stability is guaranteed because:

$$\{\} = (\{gatwick\} - \{gatwick\}) \cup \{\}$$

Example 6. In example 3, note e.g. that $\{a\}$ is a justified update of $\{\}$. The reduct operation yields the MT-revision-program with the single fact $in(a)$, and thus its necessary change is given by $In = \{a\}$ and $Out = \{\}$. Stability is guaranteed because:

$$\{a\} = (\{\} - \{\}) \cup \{a\}$$

2.2 Model updates as logic programs

With MT-revision-programs, program updating is only implicitly achieved, by recourse to the updating of each of a program's models to obtain a new set of updated models, via the update rules. Following [27], any program satisfying the updated models counts as a program update. Unfortunately, such an approach suffers, in general, from several important drawbacks:

- In order to obtain the update KB' of a knowledge base KB one has to first compute all the models M of DB (typically, a daunting task) and then individually compute their (possibly multiple) updates M_U by U. An update M_U of a given interpretation M is obtained by changing the status of only those literals in M that are *"forced"* to change by the update U, while keeping all the other literals intact by *inertia*.
- The updated knowledge base KB' is not defined directly but, instead, it is indirectly characterized as a knowledge base whose models coincide with the set of all updated models M_U of KB. In general, there is therefore no natural way of computing[2] KB' because the only straightforward candidate for KB' is the typically intractably large knowledge base KB'' consisting of all clauses that are entailed by all the updated models M_U of KB.

To overcome these important drawbacks, in this section we demonstrate a program transformation path to updating, similar to that of [6], that directly obtains, from the original program, an updated program with the required models, which is similar to the first. The updated program's models will be exactly those derivable, one by one, from the original program's models through the update rules. Thus it is possible to sidetrack the model generation path.

We shall see that any MT-revision-program can be transformed into an extended logic program which, by the way, becomes part of the new updated program. This transformation is similar in character to the one in [37], which serves a different purpose though.[3] The normal program which is subjected to the MT-revision-program, has to be transformed too. The final updated program is the union of the two transformations.

Definition 4 (Translation of MT-revision-programs into extended LPs).

Given a MT-revision-program UP, its translation into the updated logic program ULP is defined as follows.

[2] In fact, in general such a database KB' may not exist at all.

[3] In [6] a more complex transformation is considered where the program to be updated can also be an extended logic program.

– *Each update in-rule of the form (1) translates into:*

$$p \leftarrow q_1, \ldots, q_m, \neg s_1, \ldots, \neg s_n$$

– *Each update out-rule of the form (2) translates into:*

$$\neg p \leftarrow q_1, \ldots, q_m, \neg s_1, \ldots, \neg s_n$$

The rationale for this translation can best be understood in conjunction with the next definition, for they go together. Suffice it to say that we can simply equate explicit negation $\neg$ with *out*, since the programs to be updated are normal ones, and thus devoid of explicit negation (so no confusion can arise).

Definition 5 (Update transformation of a normal program). *Given a MT-revision-program UP, consider its corresponding updated logic program ULP. For any normal logic program P, its updated program U with respect to ULP (or to UP) is obtained through the operations:*

- *The rules of ULP belong to U;*
- *The rules of P belong to U, subject to the changes below;*
- *For each atom A figuring in a head of a rule of ULP:*
  - *Replace in every rule of U originated in P all occurences of A by A_i, where A_i is a new atom;*
  - *Include in U the rules $A \leftarrow A_i, not\, \neg A$ and $\neg A \leftarrow not\, A_i, not\, A$.*

The purpose of the first operation is to ensure change according to the MT-revision-program. The second operation guarantees that, for inertia, rules in P remain unchanged unless they can be affected by some update rule. The third operation changes all atoms in rules originating in P which are possibly affected by some update rule, by renaming such atoms. The new name stands for the atom as defined by the P program. The fourth operation introduces inertia rules, stating that any possibly affected atom contributes to the definition of its new version, unless actually affected through being overriden by the contrary conclusion of an update rule; the $not\, \neg A$ and $not\, A$ conditions cater for this test.

Example 7. The translation of the MT-revision-program of example 2 is:

$$\neg gatwick$$

Now suppose that the initial knowledge base is given by the logic program:

$$\begin{array}{r}gatwick \leftarrow not\, heathrow\\ heathrow \leftarrow not\, gatwick\end{array}$$

whose stable models are exactly those initial models mentioned in example 2. The corresponding updated program is (with the obvious abbreviations) :

$$\begin{array}{lrr}\neg g \quad & g_i \leftarrow not\, h \quad & g \leftarrow g_i, not\, \neg g\\ & h \leftarrow not\, g_i \quad & \neg g \leftarrow not\, g_i, not\, g\end{array}$$

Its answer-sets are $\{\neg g, g_i\}$ and $\{\neg g, h\}$.

Note that, if we restrict to the original language of P, and ignore explicit negation, these two models become exactly those that are the justified updates.

Example 8. Suppose that the initial knowledge base of example 3 is given by the empty program (whose only stable model is the empty set). The updated program of example 3 is simply:

$$a \leftarrow not\, b$$
$$b \leftarrow not\, a$$

whose answer-sets are $\{a\}$ and $\{b\}$.

Theorem 1 (Correctness of the update transformation). *Let P be a normal logic program and UP a coherent MT-revision-program. Modulo any new atoms of the form A_i and any explicitly negated elements, the answer-sets of the updated program U of P with respect to UP, are exactly the UP-justified updates of the stable models of P.*

In general, the updated programs of normal programs (without explicit negation) are extended programs (with explicit negation). To update them in turn, and thus making it possible to iterate the update process, the issue of updating extended programs has to be addressed. This was one main motivation of [6] for defining updates for models and programs with explicit negation. For the sake of simplicity of exposition, and as in our opinion this is not a central point of the present work, we refer the interested reader to [6]. There a definition of update transformation of extended program, that allows for the iteration of updates can be found that allows for the iteration of updates.

3 Program updates

As mention above, a key insight into the issue of updating theories is due to Winslett [39], who showed that one must consider the effect of an update in each of the states of the world that are consistent with our current knowledge of its state. MT-revision-programs, described in the previous section, follows this approach. The common intuition behind the update of a model has been based on what is referred to as the commonsense law of inertia, i.e. things do not change unless they are expressly made to, in this case the truth value of each element of the model to be updated should remain the same unless explicitly changed by the update. Suppose, for example, that we have a model in which "sunshine" is true and "rain" is false; if later we receive the information that the sun is no longer shining, we conclude that "sunshine" is false, due to the update, and that "rain" is still false by inertia.

Suppose now that our vision of the world is described instead by a logic program and we want to update it. Is updating each of its models enough? Is all the information borne by a logic program contained within its set of models?

The answer to both these questions is negative. A logic program encodes more than the set of its individual models. It encodes the relationships between the elements of a model, which are lost if we envisage updates simply on a model by model basis, as proposed by MT-revision-programs.

In fact, while the semantics of the resulting knowledge base after an update indeed represents the intended meaning when just the extensional part of the original knowledge base (the set of facts) is being updated, it leads to strongly *counter-intuitive* results when also the intensional part of the database (the set of rules) undergoes change, as the following example shows:

Example 9. Consider the logic program P :

$$\begin{aligned} P : sleep &\leftarrow not\, tv\_on \\ tv\_on &\leftarrow \\ watch\_tv &\leftarrow tv\_on. \end{aligned} \tag{3}$$

Clearly $M = \{tv\_on, watch\_tv\}$ is its only stable model. Suppose now that the update U states that there is a power failure, and if there is a power failure then the TV is no longer on, as represented by the logic program U:

$$\begin{aligned} U : \quad out(tv\_on) &\leftarrow in(power\_failure) \\ in(power\_failure) &\leftarrow \end{aligned} \tag{4}$$

With MT-revision-programs, we would obtain $M_U = \{power\_failure, watch\_tv\}$ as the only update of M by U. This is because *power_failure* needs to be added to the model and its addition forces us to make *tv_on* false. As a result, even though there is a power failure, we are still watching TV. However, by inspecting the initial program and the updating rules, we are likely to conclude that since "*watch_tv*" was true only because "*tv_on*" was true, the removal of "*tv_on*" should implicitly make "*watch_tv*" false by default. Moreover, one would expect "*sleep*" to become true as well.[4] Consequently, the intended model of the update of P by U is the model $M_U^{'} = \{power\_failure, sleep\}$.

Suppose now that another update U_2 follows, described by the logic program:

$$U_2 : out(power\_failure) \leftarrow \tag{5}$$

stating that power is back up again. We should now expect the TV to be on again. Since power was restored, i.e. "*power_failure*" is false, the rule "$out(tv\_on) \leftarrow in(power\_failure)$" of U should have no effect and the truth value of "*tv_on*" should be obtained by inertia from rule "$tv\_on \leftarrow$ " of the original program P.

This example illustrates that, when updating knowledge bases, it is not sufficient to just consider the truth values of literals figuring in the heads of its rules because the truth value of their rule bodies may also be affected by the updates of other literals. In other words, it suggests that the *principle of inertia* should

[4] Note the similarities between this and the "ramification problem", i.e. the problem of proliferation of implicit consequences of actions.

be applied not just to the individual literals in an interpretation but rather to the *entire rules of the knowledge base*.

Newton's first law, also known as the law of inertia, states that: *"every body remains at rest or moves with constant velocity in a straight line, unless it is compelled to change that state by an unbalanced force acting upon it"* (adapted from [36]). One often tends to interpret this law in a commonsensical way, as things keeping as they are unless some kind of force is applied to them. This is true but it doesn't exhaust the meaning of the law. It is the result of all applied forces that governs the outcome. Take a body to which several forces are applied, and which is in a state of equilibrium due to those forces canceling out. Later one of those forces is removed and the body starts to move.

The same kind of behaviour presents itself when updating programs. Before obtaining the truth value, by inertia, of those elements not directly affected by the update program, one should verify whether the truth of such elements is not indirectly affected by the updating of other elements. That is, the body of a rule may act as a force that sustains the truth of its head, but this force may be withdrawn when the body becomes false.

Another way to view program updating, and in particular the rôle of inertia, is to say that the rules of the initial program carry over to the updated program, due to inertia, instead of the truth of literals, just in case they aren't overruled by the update program. Once again this should be so because the rules encode more information than their models.

This approach was first adopted in [31], where the authors present a program transformation which, given an initial program and an update program, produces an updated program obeying the rule of inertia.

The above example also leads us to another important observation, namely, that the notion of an update KB' of one knowledge base KB by another knowledge base U should not just depend on the *semantics* of KB and U, as it is the case with interpretation updates, but that it should also depend on their *syntax*. This is best illustrated by the following, even simpler, example:

Example 10. Consider the logic program P :

$$P : innocent \leftarrow not\, found\_guilty \tag{6}$$

whose only stable model is $M = \{innocent\}$, because $found\_guilty$ is false by default. Suppose now that the update U states that the person has been found guilty:

$$U : in(found\_guilty). \tag{7}$$

Using the interpretation approach of MT-revision-programs, we would obtain $M_U = \{innocent, found\_guilty\}$ as the only update of M by U thus leading us to the counter-intuitive conclusion that the person is both innocent and guilty. This is because $found\_guilty$ must be added to the model M and yet its addition does not force us to make *innocent* false. However, it is intuitively clear that the interpretation $M_U^{'} = \{found\_guilty\}$, stating that the person is guilty but no longer presumed innocent, should be the only model of the updated program.

Mark that, such a desired model could not be obtained simply by imposing an integrity constraint to the effect that *innocent* and $found\_guilty$ cannot be simultaneously true. Such an integrity constraint would remove the undesired model but would not introduce the desired one, leaving us with a resulting update with no models at all.

Observe, however, that the program P is *semantically equivalent* to the following program P':

$$P' : innocent \leftarrow \tag{8}$$

because the programs P and P' have exactly the same set of stable models, namely the model M. Nevertheless, while the model $\{innocent, found\_guilty\}$ is not the intended model of the update of P by U it is in fact the only reasonable model of the update of P' by U.

To overcome these drawbacks of MT-revision-programs, and consider inertia of logic program rules, [4] introduced *dynamic logic programming*. In this setting, sequences of logic 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. Alternatively, the different P_is in the sequence can be viewed as different time points in possible future evolutions of the knowledge, or even as knowledge of ever more specific objects organized in a hierarchy (see [11] for more on this view). In DLP, newer or more specific rules (coming from new, newly acquired, or more specific knowledge) can be added at the end of the sequence, bothering not whether they conflict with the previous or less specific knowledge. The role of dynamic programming is to ensure that these added rules are in force, and that previous or less specific 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.

3.1 Dynamic logic programs

In this section we recapitulate Dynamic Logic Programming (DLP) [4], a framework that can be used to model the evolution of a logic program through sequences of updates.

To represent update rules, instead of using the operators in and *out* of MT-revision-programs, DLP directly uses the syntax of logic programs. In fact, $in(A)$ in the body of an update rule simply stands for A being true in the updated knowledge base, and $out(A)$ for A not being true in the updated knowledge base, i.e. *not* A being true. Accordingly, there is no need for these special operators, and the ins in bodies can be removed whilst *out*s are replaced by default negation.

Operator $in(A)$ in the head of an update rule simply stands for making A true, and thus can also be replaced by the atom itself. To represent negative information in logic programs and their updates, instead of the operator *out* of MT-revision-programs, 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 do we 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 [20], 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.

In other words, a $not\,A$ head means A is deleted if the body holds. Deleting A means that A is no longer true, not necessarily that it is false. When the CWA is adopted as well, then this deletion causes A to be false. In the updates setting, the CWA must be explicitly encoded from the start, by making all $not\,A$ false in the initial program being updated. That is, the two concepts, deletion and CWA, are orthogonal and must be separately incorporated. In the stable models [26, 32] and well-founded semantics [12] of single generalized programs, the CWA is adopted *ab initio*, and default negation in the heads is conflated with non-provability because there is no updating and thus no deletion. Note however that, unlike with single generalized programs (cf. [26]), in updates the head *not* s cannot be moved freely into the body, to obtain simple denials: there is inescapable pragmatic information in specifying exactly which *not* literal figures in the head, namely the one being deleted when the body holds true. It is not indifferent that any other (positive) body literal in the denial would be moved to the head. Example 12 shows just that.

We now recall the semantics of *single generalized logic programs*. The class of generalized logic programs can be viewed as a special case of yet broader classes of programs, introduced earlier in [26] and in [32]. As shown in [4], their semantics coincides with the stable models semantics [19] for the special case of normal programs. Moreover, the semantics also coincides with the one in [32] (and, consequently, with the one in [26]) when the latter is restricted to the language of generalized programs.

Definition 6 (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.

In the sequel, whenever L is of the form $not\,A$*, we use* $not\,L$ *to stand for the atom A.*

The semantics of generalized logic programs is defined as a generalization of the stable models semantics [19]. 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 7 (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 8 (Stable Models of Generalized Programs). *A model M is a stable model of the generalized program P iff $M = least(P \cup Default(P, M))$*

In DLP, sequences of generalized programs $P_1 \oplus \ldots \oplus P_n$ are given. As said before, 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. The role of dynamic programming is to ensure that the newly added rules (from latter programs) are in force, and that previous rules (from previous programs) 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 others 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 [11]). 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 9 (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 10 (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:*

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

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

Definition 11 (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.

Moreover, if the union of all the programs in the sequence is consistent, then the stable models of the union carry over to the update sequence. More precisely, for a sequence of two programs:

Proposition 1. *If M is a stable model of the union $P \cup U$ of programs P and U then it is also a stable model of the update program $P \oplus U$, at state U. Thus, the semantics of the update program $P \oplus U$ is always weaker than or equal to the semantics of the union $P \cup U$ of programs P and U.*

In general, the converse of the above result does not hold. In particular, the union $P \cup U$ may be a contradictory program with no stable models. This is for example the case of the updates discussed in example 9 where $P \cup U$ is contradictory (after the adequate removal of ins and replacement of $outs$ by negation as default).

Special cases where contradiction never appears are when one of the programs, P or U, is empty, or when both of them are normal logic programs (without negation in heads). In both these cases, the semantics of the $P \oplus U$ at U coincides with the semantics of the union of both programs:

Proposition 2. *If either P or U is empty then M is a stable model of $P \cup U$ iff M is a stable model of $P \oplus U$.*

Proposition 3. *If both P and U are normal programs (or if both have only clauses with default atoms $not\, A$ in their heads) then M is a stable model of $P \cup U$ iff M is a stable model of $P \oplus U$.*

It is also shown in [4] that dynamic logic programs generalize the MT-revision-programs of [34]. In fact MT-revision-programs are updates $P \oplus U$ where P is a set of facts describing the initial interpretation and U results from an easy translation of the update program.

For this result, we identify update rules:

$$\begin{aligned} in(A) &\leftarrow in(B), out(C) \\ out(A) &\leftarrow in(B), out(C) \end{aligned} \tag{9}$$

used in MT-revision-programs, with the following generalized logic program clauses:

$$\begin{aligned} A &\leftarrow B, not\, C \\ not\, A &\leftarrow B, not\, C. \end{aligned} \tag{10}$$

Theorem 2 (Program updates generalize interpretation updates). *Let I be any interpretation and U any updating program in the language $\mathcal{L}$. Denote by P_I the generalized logic program in $\mathcal{L}$ defined by*

$$P_I = \{A \ \leftarrow\, : A \in I\} \ \cup \ \{not\, A \ \leftarrow\, : not\, A \in I\}.$$

Then M is a stable model of the program update $P_I \oplus U$ of the program P_I by the program U iff M is an U-justified update of I.

In [4] 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.[5]

3.2 Examples

Example 11. Consider the DLP $P \oplus U_1 \oplus U_2$ introduced in example 9, used to specify the evolution of a knowledge base, and P, U_1 and U_2 are:

$$\begin{array}{lll} P : sleep \leftarrow not\, tv\_on & U_1 : failure & U_2 : not\, failure \\ \quad tv\_on & \quad not\, tv\_on \leftarrow failure & \\ \quad watch\_tv \leftarrow tv\_on & & \end{array}$$

Clearly, the stable models at state P coincide with the stable models of P, i.e. there is only one which is $\{tv\_on, watch\_tv\}$.

At state U_1 there is a single stable model: $M_1 = \{failure, sleep\}$. In fact, $Reject(U_1, M_1) = \{tv\_on\}$, because the rule $not\, tv\_on \leftarrow failure$ of U_1 has a complementary head, of a rule with true body at M_1, and $Default(P \cup U_1, M_1) = \{not\, watch\_tv, not\, tv\_on\}$. The least model of $P \cup U_1$ minus rejected rules plus default literals is exactly M_1, and thus it is a stable model at U_1.

One can easily check the only stable model at U_2 is $M_2 = \{tv\_on, watch\_tv\}$.

Example 12. Consider the DLP $P_1 \oplus P_2$, where P_1 and P_2 are:

$$\begin{array}{ll} P_1 : c \leftarrow & P_2 : not\, a \leftarrow c \\ \quad a \leftarrow not\, b & \end{array}$$

The only stable model at P_2 is $M = \{c, not\, a, not\, b\}$. In fact, $Default(P_1 \cup P_2, M) = \{not\, b\}$, $Reject_2(M) = \{a \leftarrow not\, b\}$, and:

$$M = \{c, not\, a, not\, b\} = least((P_1 \cup P_2 - \{a \leftarrow not\, b\}) \cup \{not\, b\})$$

Note here that, as mentioned in Section 3.1, in DLPs the head *not*'s cannot be moved freely into the body, to obtain denials. The rule in P_2 includes the pragmatic information that a is to be deleted if c is true, information that would be lost with the denial. Intuitively that rule makes a different statement from that of the rule $not\, c \leftarrow a$, which however yields the same denial. And this difference is reflected in the definition of stable models for DLPs. In fact, if the rule in P_2 is replaced by this other one, the only stable model at P_2 would be $\{not\, c, a, not\, b\}$ instead.

The reader can check that if the rule in P_2 is replaced by $u \leftarrow a, c, not\, u$ (which, under the stable models semantics, is equivalent to the denial) the results are also different from the ones above: with this rule instead, there is no stable model at P_2.

Example 13. To illustrate the usage of DLP to represent priority of ever more specific knowledge, consider the well-known problem of flying birds, where we have several rules with different priorities. First, the animals-do-not-fly rule, with the lowest priority; then, the birds-fly rule with a higher priority; next, the penguins-do-not-fly rule with an even higher priority; and, finally, with the

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

highest priority, all the is-a rules describing the actual taxonomy. This can be coded quite naturally in dynamic logic programming (where rules with variables simply stand for all their ground instances):

$$\begin{array}{ll}
P_1 : \mathit{not}\, fly(X) \leftarrow animal(X) & P_4 : animal(X) \leftarrow bird(X) \\
P_2 : fly(X) \leftarrow bird(X) & \quad bird(X) \leftarrow penguin(X) \\
P_3 : \mathit{not}\, fly(X) \leftarrow penguin(X) & \quad animal(pluto) \\
 & \quad bird(duffy) \\
 & \quad penguin(tweety)
\end{array}$$

The reader can check that, as intended, the dynamic logic program $P_1 \oplus P_2 \oplus P_3 \oplus P_4$ at state 4 has a single stable model where $fly(duffy)$ is true, and both $fly(pluto)$ and $fly(tweety)$ are false. Note how the rule $\mathit{not}\, fly(duffy) \leftarrow animal(duffy)$ of P_1 is rejected by the rule $fly(duffy) \leftarrow bird(duffy)$ of P_2, and the rule $fly(tweety) \leftarrow bird(tweety)$ of P_2 is rejected by the rule $\mathit{not}\, fly(tweety) \leftarrow penguin(tweety)$ of P_3.

Example 14. To illustrate the usage of DLP to represent knowledge about hierarchies of objects, consider the following example, adapted from [11].

There are 3 objects in a simple part-of hierarchy, o_1, o_2 and o_3, where object o_2 is part-of object o_1, and object o_3 is also part-of object o_1. Each of the objects has some security information, where users' access authorizations to objects are specified by the predicate $auth(User)$.

Now suppose we want to say, as general rules for o_1 (to be inherited by its part-of objects), that Bob is authorized just in case Ann is not authorized, and that either Ann or Tom is authorized if Alice isn't. Moreover, for object o_2 we want to say that Alice is authorized, and for object o_3 that Bob is not. This information can be represented by the DLP $\bigoplus\{P_{o_1}, P_{o_2}, P_{o_3}\}$, where $o_1 < o_2$ and $o_1 < o_3$, and where:

$$\begin{array}{ll}
P_{o_1} : auth(bob) \leftarrow \mathit{not}\, auth(ann) & P_{o_2} : auth(alice) \\
\quad auth(ann) \leftarrow \mathit{not}\, auth(tom), \mathit{not}\, auth(alice) & \\
\quad auth(tom) \leftarrow \mathit{not}\, auth(ann), \mathit{not}\, auth(alice) & P_{o_3} : \mathit{not}\, auth(bob)
\end{array}$$

The access authorizations for object o_1 are given by the stable models at o_1, which are: $\{auth(ann)\}$ and $\{auth(bob), auth(tom)\}$. For object o_2 there is a single stable model $\{auth(bob), auth(alice)\}$. For object o_3 there are two stable models: $\{auth(ann)\}$ and $\{auth(tom)\}$.

3.3 Other references to program updates

Recently, Dynamic Logic Programs have been studied by Eiter et al. in [17]. There, a syntatic redefinition of DLPs is presented, and semantical properties are investigated. In particular, a study on the DLP-verification of well known postulates of belief revision [1], iterated revision [13], of theory updates [27] is carried out. Further structural properties of DLPs, when viewed as nonmonotonic consequence operators, are also studied in [17]. Structural properties of logic program updates are also studied in [15].

As noted in [17], DLP makes no attempt to minimize the set of rules that are rejected. It is argued there that this could be a natural approach for measuring the change which some program P_1 undergoes when updated by some other program P_2, thus functioning as a good criterion for minimality of change. In case such a minimality criterion is desired, they refine the semantics of DLP, and introduce minimal stable models at states. The following example illustrates the intuition behind minimal stable models. For the formal definition, and further motivation see [17].

Example 15. Consider that the program P_1

$$a \leftarrow not\, b \qquad\qquad not\, b$$

is updated by program $P_2 = \{b \leftarrow not\, a\}$.

There are two stable models of $P_1 \oplus P_2$ at P_2. One is $M_1 = \{a, not\, b\}$. In fact, $Default(P_1 \cup P_2, M_1) = \{not\, b\}$, $Reject_2(M_1) = \{\}$, and:

$$M_1 = \{a, not\, b\} = least(P_1 \cup P_2 \cup \{not\, b\})$$

The other one is $M_2 = \{b, not\, a\}$. Here, $Default(P_1 \cup P_2, M_2) = \{not\, a\}$, $Reject_2(M_2) = \{not\, b\}$, and:

$$M_2 = \{b, not\, a\} = least\left([(P_1 \cup P_2) - \{not\, b\}] \cup \{not\, a\}\right)$$

M_2 is not a minimal stable model of $P_1 \oplus P_2$ because the set of P_1's rules rejected in the case of that model (i.e. $\{not\, b\}$) is a proper superset of the set of P_1's rules rejected in the case of M_1 (i.e. $\{\}$). Accordingly, the only minimal stable model of $P_1 \oplus P_2$ is M_1. This is the only one that guarantees that only a minimal set of rules from the initial P_1 is rejected.

Another important result of [17], is the clarification of the close relationship between DLPs and inheritance programs [11]. Though defined with different goals, inheritance programs share some close similarities with DLP. Inheritance programs [11] aim at extending with inheritance disjunctive logic programming with strong negation. In them, a hierarchy of objects (or knowledge bases) is given, where each object has a logic program. The role of inheritance programs is to establish the semantics at each object. Inheritance is used to inherit, as much as possible, rules from objects higher in the hierarchy into the objects lower down, i.e. as long as they do not conflict with the rules of the more specific objects.

Other approaches to updates of logic programs by logic programs are presented in [25] and in [38]. Based on an abductive framework for (non-monotonic) auto-epistemic theories, that make use of the notion of negative explanation and anti-explanation, in [25] the authors define "autoepistemic updates". Based on this work, in [38] they employ this new abduction framework (in this case rewritten for logic programming instead) to compute minimal programs which result from updating one logic program by another. In their framework, several updates are possible because non-deterministic contradiction removal is used to revise inconsistencies (through abduction) between an initial program and the one updating it, giving preference to the rules of the latter. In their framework, updating and revision take place simultaneously.

Yet another, independently defined, approach to logic programs updates, can be found in [42]. As in DLPs, the semantics of the update of a program by another is obtained by removing rules from the initial program which "somehow" contradict rules from the update program, and retaining all others by inertia. Additionally, at the end, prioritized logic programs are used to give preference to rules from the update program over all retained rules of the initial program. This last step leads to quite different results when compared to DLP. For example, consider a program P with the single rule $a \leftarrow not\, b$, which we want to update with a program $U = \{b \leftarrow not\, a\}$. With DLPs, the rule from P is not rejected, and the semantics of the update equal the semantics of $P \cup U$, i.e. it has two stable models: $\{a\}$ and $\{b\}$. With the approach of [42], priority is given to the rules of U, and the only resulting stable model is $\{b\}$. In our opinion, preferences and updates are different concepts, that have to be considered separately. In DLPs, when there are no conflicts among rules (which is the case in the above example) there is no difference between updating and the union of the programs. If preferences are wanted, then they should be added on top of DLP, to prefer some rules over others. This issue, of preferences and DLPs, is studied in [7]. One important drawback of the approach of [42] is that it cannot be used to consider sequences of updates, and simply captures a single update of one program by another.

4 Languages for updates

Dynamic logic programming does not by itself provide a proper language for specifying (or programming) changes to logic programs. If knowledge is already represented by logic programs, dynamic programs simply represent the evolution of knowledge. But how is that evolving knowledge specified? What makes knowledge evolve? Since logic programs describe knowledge states, it's only fitting that logic programs describe transitions of knowledge states as well. It is natural to associate with each state a set of transition rules to obtain the next state. As a result, an interleaving sequence of states and rules of transition will be obtained. Imperative programming specifies transitions and leaves states implicit. Logic programming, up to now, could not specify state transitions. With the language of dynamic updates LUPS we make both states and their transitions declarative.

Usually updates are viewed as actions or commands that make the knowledge base evolve from one state to another. This is the classical view e.g. in relational databases: the knowledge (data) is expressed declaratively via a set of relations; updates are commands that change the data. In the previous section, updates were viewed declaratively as a given update store consisting of the sequence of programs. They were more in the spirit of state transition rules, rather than commands. Of course, one could say that the update commands were implicit. For instance, in example 9, the sequence $P \oplus U \oplus U_2$ could be viewed as the result of, starting from P, performing first, simultaneously, the update commands **assert** $not\, tv\_on \leftarrow power\_failure$ and **assert** $power\_failure$, and then the update command **assert** $not\, power\_failure$. But, if viewed as a language for (implicitly) specifying update commands, dynamic logic programming is quite poor. For instance, it does not provide any mechanism for saying that some rule (or fact) should be asserted only whenever some conditions are satisfied. This is essential in the domain of actions, to specify direct effects of actions. For example, suppose we want to state that $wake\_up$ should be added to our knowledge base whenever $alarm\_rings$ is true. As a language for specifying updates, dynamic logic programming does not provide a way of specifying such an update command. Note that the command is distinct from **assert** $wake\_up \leftarrow alarm\_rings$. With the latter, if the alarm stops ringing (i.e. if $not\, alarm\_rings$ is later asserted), $wake\_up$ becomes false. In the former, we expect $wake\_up$ to remain true (by inertia) even after the alarm stops ringing. As a matter of fact, in this case, we don't want to add the rule saying that $wake\_up$ is true whenever $alarm\_rings$ is also true. We simply want to add the fact $wake\_up$ as soon as $alarm\_rings$ is true. From there on, no connection between $wake\_up$ and $alarm\_rings$ should persist.

This simple one-rule example also highlights another limitation of dynamic logic programming as a language for specifying update commands: one must explicitly say to which program in the sequence a rule belongs. Sometimes, in particular in the domain of actions, there is no way to know a priori to which state (or program) a rule should belong to. Where should we assert the fact $wake\_up$? This is not known a priori because we don't know when $alarm\_rings$.

In this section we show a language for specifying logic program updates: LUPS – "Language of dynamic updates" [8]. The object language of LUPS is that of generalized logic programs. A sentence U in LUPS is a set of simultaneous update commands (or actions) that, given a pre-existing sequence of logic programs $P_0 \oplus \ldots \oplus P_n$ (i.e. a dynamic logic program), whose semantics corresponds to our knowledge at a given state, produces a sequence with one more program, $P_0 \oplus \ldots \oplus P_n \oplus P_{n+1}$, corresponding to the knowledge that results from the previous sequence after performing all the simultaneous commands. A program in LUPS is a sequence of such sentences.

Given a program in LUPS, its semantics is defined by means of a dynamic logic program generated by the sequence of commands. In [8], a translation of a LUPS program into a generalized logic program is presented, where stable models exactly correspond to the semantics of the original LUPS program. This translation directly provides an implementation of LUPS.

In this update framework, knowledge evolves from one knowledge state to another as a result of update commands stated in the object language. Knowledge states KS_i represent dynamically evolving states of our knowledge. They undergo change due to *update actions*. Without loss of generality (as will become clear below) we assume that the initial knowledge state, KS_0, is empty and that in it all predicates are *false* by default. This is the *default knowledge state*. Given the *current knowledge state* KS, its *successor knowledge state* $KS[U]$ is produced as a result of the occurrence of a non-empty set U of simultaneous *updates*. Each of the updates can be viewed as a set of (parallel) *actions* and consecutive knowledge states are obtained as

$$KS_n = KS_0[U_1][U_2]...[U_n]$$

where U_i's represent consecutive sets of updates. We also denote this state by:

$$KS_n = U_1 \otimes U_2 \otimes \ldots \otimes U_n$$

So defined sequences of updates will be called *update programs*. In other words, an update program is a finite sequence $\mathcal{U} = \{U_s : s \in S\}$ of updates indexed by the set $S = \{1, 2, \ldots, n\}$. Each updates is a set of update commands. Update commands (to be defined below) specify *assertions* or *retractions* to the current knowledge state. By the current knowledge state we mean the one resulting from the last update performed.

Knowledge can be queried at any state $q \leq n$, where n is the index of the current knowledge state. A query will be denoted by:

$$\textbf{holds } B_1, \ldots, B_k, not\, C_1, \ldots, not\, C_m \textbf{ at } q?$$

and is true iff the conjunction of its literals holds at the state KB_q. If $q = n$, we simply skip the state reference "**at** q".

4.1 Update commands

Update commands cause changes to the current knowledge state leading to a new successor state. The simplest command consists of adding a rule to the current state: **assert** $L \leftarrow L_1, \ldots, L_k$. For example, when a law stating that abortion is punished by jail is approved, the knowledge state might be updated via the command: **assert** $jail \leftarrow abortion$.

In general, the addition of a rule to a knowledge state may depend upon some precondition. To allow for that, an assert command in LUPS has the form:

$$\textbf{assert } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{11}$$

The meaning of such an assert rule is that if the precondition $L_{k+1}, \ldots, L_m$ is true in the current knowledge state, then the rule $L \leftarrow L_1, \ldots, L_k$ should belong to the successor knowledge state. Normally, the so added rule persists, or is in force, from then on by inertia, until possibly defeated by some future update or until retracted. This is the case for the assert-command above: the rule $jail \leftarrow abortion$ remains in effect by inertia from the successor state onwards unless later invalidated.

However, there are cases where this persistence by inertia should not be assumed. Take, for instance, the $alarm\_ring$ discussed in the introduction. This fact is a one-time event that should not persist by inertia, i.e. it is not supposed to hold by inertia after the successor state. In general, facts that denote names of events or actions should be *non-inertial.* Both are true in the state they occur, and do not persist by inertia for later states. Accordingly, the rule within the assert command may be preceded with the keyword *event*, indicating that the added rule is non-inertial. Assert commands are thus of the form (11) or of the form:[6]

$$\textbf{assert event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{12}$$

While some update commands, such as **assert** *republican_congress*, represent newly incoming information, and are thus one-time non-persistent update commands (whose effect, i.e. the truth of *republican_congress*, may nevertheless persist by inertia), some other update commands are liable to be *persistent*, i.e., to remain in force until cancelled. For example, an update like:

$$\textbf{assert } jail \leftarrow abortion \textbf{ when } rep\_congress, rep\_president$$

or

$$\textbf{assert } wake\_up \textbf{ when } alarm\_sounds$$

might be always true, or at least true until cancelled. Enabling the possibility of such updates allows our system to dynamically change without any truly new (external) updates being received. For example, the persistent update command:

$$\textbf{assert } set\_hands(T) \textbf{ when } get\_hands(C) \wedge get\_time(T) \wedge (T - C) > \Delta$$

defines a perpetually operating clock whose hands move to the actual time position whenever the difference between the clock time and the actual time is sufficiently large.

In order to specify such persistent updates commands (which we call laws) we introduce the syntax:

$$\textbf{always } [\textbf{event}] \; L \leftarrow L_1, \ldots, L_k \; when \; L_{k+1}, \ldots, L_m \tag{13}$$

For stopping persistent update commands, we use:

$$\textbf{cancel } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{14}$$

The first statement means that, in addition to any new set of arriving update commands, we are also supposed to keep executing this persistent update command. The second statement cancels this persistent update, when the conditions for cancellation are met.

The existence of persistent update commands requires a "trivial" update, which does not specify any truly new updates but simply triggers all the already defined persistent updates to fire, thus resulting in a new modified knowledge state. Such "no-operation" update ensures that the system continues to evolve, even when no truly new updates are specified, and may be represented by **assert** *true*. It stands for the *tick of the clock* that drives the world being modelled.

[6] In both cases, if the precondition is empty we just skip the whole *when* subclause.

To deal with the deletion of rules, we have available the *retraction* command:

$$\textbf{retract } [\textbf{event}]\ L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{15}$$

meaning that, subject to precondition $L_{k+1}, \ldots, L_m$, the rule $L \leftarrow L_1, \ldots, L_k$ is either retracted from now on, or just retracted temporarily in the next state (non-inertial retract, i.e. an event of retraction, triggered by the *event* keyword).

The cancelling of an update command is not equivalent to the retracting of a rule. Cancelling an update just means it will no longer be added as a command to updates, it does not cancel the inertial effects of its previous application(s). However, retracting an update causes any of its inertial effects to be cancelled from now on, as well as cancelling a persistent law. Also, note that "*retract event*..." does not mean the retracting of an event, because events persist only for one state and thus do not require retraction. It represents a temporary removal of a rule from the successor state (a temporary retraction event).

Definition 12 (LUPS). *An update program in LUPS is a finite sequence of updates, where an update is a set of commands of the form (11) to (15).*

Example 16. Consider the following scenario:

- once Republicans take over both Congress and the Presidency they establish a law stating that abortions are punishable by jail;
- once Democrats take over both Congress and the Presidency they abolish such a law;
- in the meantime, there are no changes in the law because always either the President or the Congress vetoes such changes;
- performing an abortion is an event, i.e. a non-inertial update.

Consider the following update history: (1) a Democratic Congress and a Republican President (Reagan); (2) Mary performs abortion; (3) Republican Congress is elected (Republican President remains in office: Bush); (4) Kate performs abortion; (5) Clinton is elected President; (6) Ann performs abortion; (7) G.W. Bush is elected president (8) A democrat is elected President and Democratic Congress is in place (year 2004?); (9) Susan performs abortion.

The specification in LUPS would be:

Persistent update commands:

$$\begin{array}{l} \textbf{always } jail(X) \leftarrow abt(X) \textbf{ when } repC \wedge repP \\ \textbf{always } not\, jail(X) \leftarrow abt(X) \textbf{ when } not\, repC \wedge not\, repP \end{array}$$

Alternatively, instead of the second clause, in this example, we can use a retract statement

$$\textbf{retract } jail(X) \leftarrow abt(X) \textbf{ when } not\, repC \wedge not\, repP$$

Note that, in this example, since there is no other rule implying $jail$, retracting the rule is safely equivalent to retracting its conclusion.

The above rules state that we are always supposed to update the current state with the rule $jail(X) \leftarrow abt(X)$ provided $repC$ and $repP$ hold true and that we are supposed to assert the opposite (or just retract this rule) provided $not\,repC$ and $not\,repP$ hold true. Such persistent update commands should be added to U_1.

Sequence of non-persistent update commands:

U_1 : **assert** $repP$
assert $not\,repC$
U_2 : **assert event** $abt(mary)$
U_3 : **assert** $repC$
U_4 : **assert event** $abt(kate)$
U_5 : **assert** $not\,repP$
U_6 : **assert event** $abt(ann)$
U_7 : **assert** $repP$
U_8 : **assert** $not\,repP$ **assert** $not\,recC$
U_9 : **assert event** $abt(susan)$

Of course, in the meantime we could have a lot of trivial update events representing ticks of the clock, or any other irrelevant updates.

4.2 Semantics of LUPS

In this section we provide update programs with a meaning, by translating them into dynamic logic programs. The semantics of a LUPS program is then determined by the semantics of the so obtained dynamic program.

More precisely, the translation of a LUPS program into a dynamic program is obtainable by induction, starting from the empty program P_0, and for each update U_i, given the already built dynamic program $P_0 \oplus \ldots \oplus P_{i-1}$, determining the resulting program $P_0 \oplus \ldots \oplus P_{i-1} \oplus P_i$. To cope with persistent update commands we will further consider, associated with every dynamic program in the inductive construction, a set containing all currently active persistent commands, i.e. all those that were not cancelled, up to that point in the construction, from the time they were introduced. To be able to retract rules, we need to uniquely identify each such rule. This is achieved by augmenting the language of the resulting dynamic program with a new propositional variable "$rule(L \leftarrow L_1, \ldots, L_n)$" for every rule $L \leftarrow L_1, \ldots, L_n$ appearing in the original LUPS program.[7]

Definition 13 (Translation into dynamic programs). *Let $\mathcal{U} = U_1 \otimes \ldots \otimes U_n$ be an update program. The corresponding dynamic program $\Upsilon(\mathcal{U}) = \mathcal{P} = P_0 \oplus \ldots \oplus P_n$ is obtained by the following inductive construction, using at each step i an auxiliary set of persistent commands PC_i:*

Base step: $P_0 = \{\}$ with $PC_0 = \{\}$.

Inductive step: Let $\mathcal{P}_i = P_0 \oplus \ldots \oplus P_i$ with the set of persistent commands PC_i be the translation of $\mathcal{U}_i = U_1 \otimes \ldots \otimes U_i$. The translation of $\mathcal{U}_{i+1} = U_1 \otimes \ldots \otimes U_{i+1}$ is $\mathcal{P}_{i+1} = P_0 \oplus \ldots \oplus P_{i+1}$ with the set of persistent commands PC_{i+1}, where:

$PC_{i+1} = PC_i \cup$
$\cup \{$***assert*** R ***when*** C : ***always*** R ***when*** $C \in U_{i+1}\}$

[7] Note that, by definition, all such rules are ground and thus the new variable uniquely identifies the rule, where rule/1 is a reserved predicate.

$\cup\{$***assert event*** R ***when*** C : ***always event*** R ***when*** $C \in U_{i+1}\}$
$-\{$***assert [event]*** R ***when*** C : ***cancel*** R ***when*** $D \in U_{i+1} \wedge \bigoplus_i \mathcal{P}_i \models_{sm} D\}$
$-\{$***assert [event]*** R ***when*** C : ***retract*** R ***when*** $D \in U_{i+1} \wedge \bigoplus_i \mathcal{P}_i \models_{sm} D\}$

$$NU_{i+1} = U_{i+1} \cup PC_{i+1}$$

$P_{i+1} = \{R,\ rule(R)$: ***assert [event]*** R ***when*** $C \in NU_{i+1} \wedge \bigoplus_i \mathcal{P}_i \models_{sm} C\}$
$\cup\ \{not\,rule(R)$: ***retract [event]*** R ***when*** $C \in NU_{i+1} \wedge \bigoplus_i \mathcal{P}_i \models_{sm} C\}$
$\cup\ \{not\,rule(R)$: ***assert event*** R ***when*** $C \in NU_i \wedge \bigoplus_{i-1} \mathcal{P}_{i-1} \models_{sm} C\}$
$\cup\ \{rule(R)$: ***retract event*** R ***when*** $C \in NU_i \wedge \bigoplus_{i-1} \mathcal{P}_{i-1} \models_{sm} C, rule(R)\}$

where R denotes a generalized logic program rule, and C and D a conjunction of literals. ***assert [event]*** *R* ***when*** *C and* ***retract [event]*** *R* ***when*** *C are used for notational convenience, and stand for either the assert or the assert-event command (resp. retract and retract-event). So, for example in the first line of the definition of P_{i+1}, R and $rule(R)$ must be added either if there exists a command* ***assert*** *R* ***when*** *C or a command* ***assert event*** *R* ***when*** *C obeying the conditions there.*

In the inductive step, if $i = 0$ the last two lines are omitted. In that case NU_i does not exist.

Definition 14 (LUPS semantics). *Let $\mathcal{U}$ be an update program.*
A query ***holds*** *$L_1, \ldots, L_n$* ***at*** *q is true in $\mathcal{U}$ iff $\bigoplus_q \Upsilon(\mathcal{U}) \models_{sm} L_1, \ldots, L_n$.*

From the results on dynamic programs, it is clear that LUPS generalizes the language of updates of MT-revision-programs presented in section 2.1:

Proposition 4 (LUPS generalizes MT-revision-programs). *Let I be an interpretation and R a MT-revision-program. Let $\mathcal{U} = U_1 \otimes U_2$ be the update program where:*

$$U_1 = \{\textbf{\textit{assert}}\ A : A \in I\}$$

$$U_2 = \{\textbf{\textit{assert}}\ A \leftarrow B_1, \ldots, not\,B_n : in(A) \leftarrow in(B_1), \ldots, out(B_n) \in R\}$$
$$\cup\{\textbf{\textit{assert}}\ not\,A \leftarrow B_1, \ldots, not\,B_n : out(A) \leftarrow in(B_1), \ldots, out(B_n) \in R\}$$

Then, M is a stable model of $\Upsilon(\mathcal{U})$ iff M is an interpretation update of I by R in the sense of [34].

This definition of the LUPS semantics is based on a translation into dynamic logic programs, and is not purely syntactic. Indeed, to obtain the translated dynamic program, one needs to compute, at each step of the inductive process, the consequences of the previous one.

In [8] a translation of update programs and queries, into normal logic programs written in a meta-language is presented. That translation is purely syntactic, and is correct in the sense that a query holds in an update program iff the translation of the query holds in all stable models of the translation of the update program. It also directly provides a mechanism for implementing update programs: with a pre-processor performing the translations, query answering is reduced to that over normal logic programs.[8]

5 Application domains

In this section we discuss and illustrate with examples the applicability of the language LUPS to several broad knowledge representation domains. The selected domains include: *theory of actions, legal reasoning, and software specification.* Additional examples and domains of application of LUPS can be found in [9].

The theory of actions (for a survey see [21]) is very closely related to knowledge updates. An action taking place at a specific moment of time may cause an effect in the form of a change of the status of some fluent. For example, an action of stepping on a sharp nail may result in severe pain. The occurrence of pain can therefore be viewed as a simple (atomic) knowledge update triggered by a given action. Similarly, a set of parallel actions can be viewed as triggering (causing) parallel atomic updates. The following *suitcase* example illustrates how LUPS can be used to handle parallel updates.

Example 17 (Suitcase). There is a suitcase with two latches which opens whenever both latches are up, and there is an action of toggling applicable to each latch [33]. This situation is represented by the three persistent rules:

$$\begin{array}{l} \textbf{always } \mathit{open} \leftarrow \mathit{up}(l1), \mathit{up}(l2) \\ \textbf{always } \mathit{up}(L) \textbf{ when } \mathit{not\ up}(L), \mathit{toggle}(L) \\ \textbf{always } \mathit{not\ up}(L) \textbf{ when } \mathit{up}(L), \mathit{toggle}(L) \end{array}$$

In the initial situation $l1$ is down, $l2$ is up, and the suitcase is closed:

$$U_1 = \{\textbf{assert } \mathit{not\ up}(l1), \textbf{assert } \mathit{up}(l2), \textbf{assert } \mathit{not\ open}\}$$

Suppose there are now two simultaneous toggling actions:

$$U_2 = \{\textbf{assert event } \mathit{toggle}(l1), \textbf{assert event } \mathit{toggle}(l2)\}$$

and afterwards another $l2$ toggling action: $U_3 = \{\textbf{assert event } \mathit{toggle}(l2)\}$. In the knowledge state 2 we will have $\mathit{up}(l1), \mathit{not\ up}(l2)$ and the suitcase is not open. Only after U_3 will latch $l2$ be up and the suitcase open.

Robert Kowalski's team did truly outstanding research work on using logic programming as a *language for legal reasoning* (see e.g. [29]). However logic programming itself lacks any mechanism for expressing dynamic changes in the law due to revisions of the law or due to new legislation. LUPS allows us to handle such changes in a very natural way by augmenting the knowledge base only with the newly added or revised data, and automatically obtaining the updated information as a result. We illustrate this capability on the following simple example.

[8] Such a pre-processor and a meta-interpreter for query answering have been implemented and are available at: `http://centria.di.fct.unl.pt/~jja/updates/`

Example 18 (Conscientious objector). Consider a situation where someone is conscripted if he is draftable and healthy. Moreover a person is draftable when he attains a specific age. However, after some time, the law changes and a person is no longer conscripted if he is indeed a conscientious objector:

$$
\begin{aligned}
U_1 :\ & \textbf{always}\ draftable(X)\ \textbf{when}\ of\_age(X) \\
& \textbf{assert}\ conscripted(X) \leftarrow draftable(X), healthy(X) \\
U_2 :\ & \textbf{assert}\ healthy(a).\ \textbf{assert}\ healthy(b).\ \textbf{assert}\ of\_age(b). \\
& \textbf{assert}\ consc\_objector(a).\ \textbf{assert}\ consc\_objector(b) \\
U_3 :\ & \textbf{assert}\ of\_age(a) \\
U_4 :\ & \textbf{assert}\ not\ conscripted(X) \leftarrow consc\_objector(X)
\end{aligned}
$$

In state 3, b is subject to conscription but after the last assertion his situation changes. On the other hand, a is never conscripted.

One of the most important problems in software engineering is that of choosing a suitable software specification language. The following are among the key desired properties of such a language:

1. Possibility of a concise representation of statements of natural language, commonly used in informal descriptions of various domains.
2. Availability of query answering systems which allow rapid prototyping.
3. Existence of a well developed and mathematically precise semantics of the language.
4. Ability to express conditions that change dynamically.
5. Ability to handle inconsistencies stemming from specification revisions.

It has been argued in the literature that the language of logic programming is a good potential candidate for the language of software specifications. However, it lacks simple and natural ways of expressing conditions that change dynamically and the ability to handle inconsistencies stemming from specification revisions. The following simplified banking example illustrates how LUPS can be used to represent changes in software specifications.

Example 19 (Banking transactions). Consider a software specification for performing banking transactions. Account balances are modeled by the predicate $balance(AccountNo, Balance)$. Predicates $deposit(AccountNo, Amount)$ and $withdrawal(AccountNo, Amount)$ represent the actions of depositing and withdrawing money into and out of an account, respectively. A withdrawal can only be accomplished if the account has a sufficient balance. This simplified description can easily be modeled in LUPS by U_1:

$$
\begin{aligned}
& \textbf{always}\ balance(Ac, OB + Up)\ \textbf{when}\ updateBal(Ac, Up), balance(Ac, OB) \\
& \textbf{always}\ not\ balance(Ac, OB)\ \textbf{when}\ updateBal(Ac, NB), balance(Ac, OB) \\
& \textbf{assert}\ updateBal(Ac, -X) \leftarrow withdrawal(Ac, X), balance(Ac, O), O > X \\
& \textbf{assert}\ updateBal(Ac, X) \leftarrow deposit(Ac, X)
\end{aligned}
$$

The first two rules state how to update the balance of an account, given any event of $updateBal$. By the last two rules, deposits and withdrawals are effected, causing $updateBal$.

An initial situation can be imposed via *assert* commands. Deposits and withdrawals can be stipulated by asserting events of *deposit*/2 and *withdrawal*/2. E.g.:

$$U_2 : \{\textbf{assert } balance(1,0), \textbf{assert } balance(2,50)\}$$
$$U_3 : \{\textbf{assert event } deposit(1,40), \textbf{assert event } withdrawal(2,10)\}$$

causes the balance of both accounts 1 and 2 to be 40, after state 3.

Now consider the following sequence of informal specification revisions. Deposits under 50 are no longer allowed; VIP accounts may have a negative balance up to the limit specified for the account; account #1 is a VIP account with the overdraft limit of 200; deposits under 50 are allowed for accounts with negative balances. These can in turn be modeled by the sequence:

$$U_4 : \textbf{assert } not\ updateBal(Ac, X) \leftarrow deposit(Ac, X), X < 50$$
$$U_5 : \textbf{assert } updateBal(Ac, -X) \leftarrow vip(Ac, L), withdrawal(Ac, X), balance(Ac, B), B + L \geq X$$
$$U_6 : \textbf{assert } vip(1, 200)$$
$$U_7 : \textbf{assert } updateBal(Ac, X) \leftarrow deposit(Ac, X), balance(Ac, B), B < 0$$

6 Future Perspectives

Knowledge updating is not to be simply envisaged as taking place in the time dimension alone. Several updating dimensions may combine simultaneously, with or without the temporal one, such as specificity (as in taxonomies), strength of the updating instance (as in the legislative domain), hierarchical position of knowledge source (as in organizations), credibility of the source (as in uncertain, mined, or learnt knowledge), or opinion precedence (as in a society of agents).

What's more, updating inevitably raises issues about revising and preferring, and some work is emerging on the articulation of these distinct but highly complementary aspects. And learning is usefully seen as successive approximate change, as opposed to exact change, and combining the results of learning by multiple agents, multiple strategies, or multiple data sets, inevitably poses problems within the province of updating. Last but not least, goal directed planning can be fruitfully envisaged as abductive updating.

Thus, not only do the aforementioned topics combine naturally together – and so require precise, formal, means and tools to do so – but their combination results in turn in a nascent complex architectural basis and component for Logic Programming rational agents, which can update one another and common, structured, blackboard agents.

We surmise, consequently, that fostering this meshing of topics within the Logic Programming community is all of opportune, seeding, and fruitful. Indeed, application areas, such as software development, multi-strategy learning, abductive planning, model-based diagnosis, agent architectures, and others, are being successfully pursued employing the above outlook.

In this necessarily selective introductory guided tour, we have not delved into topics, mentioned above, which have already been the subject of research and publication, and that the captive reader may want to pursue, namely: the combination of updates with preferences [7]; the extension to multiple updating dimensions [30]; the coupling of updates and abduction in planning [2]; mutually updating agents [16]; updating postulates, structural properties, and complexity [17, 15].

A well-founded semantics for generalized programs has also been defined [9], which allows carrying over results to 3-valued updates. This semantics is actually the basis for one of our implementations under the XBS system. The other implementation, relies on a preprocessor that produces programs to be run under the DLV-system.

In the body of the references provided below the interested reader may follow up on other works and approaches, and glean in them critical appraisals and comparisons.

Acknowledgements

We thank the co-authors of joint papers from which we have extracted or adapted much material, namely João Leite, Halina Przymusinska, Teodor Przymusinski, and Paulo Quaresma. We acknowledge the support of PRAXIS projects MENTAL and ACROPOLE.

References

1. C. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *J. Symbolic Logic*, 50(2):510–530, 1985.
2. J. Alferes, J. A. Leite, L. M. Pereira, and P. Quaresma. Planning as abductive updating. In D. Kitchin, editor, *AISB'00 Symposium on AI Planning and Intelligent Agents*, pages 1–8. AISB, 2000.
3. J. J. Alferes, C. V. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14:93–147, 1995.
4. 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*, 45(1-3):43–70, 2000. A short version titled *Dynamic Logic Programming* appeared in A. Cohn and L. Schubert (eds.), *KR'98*, Morgan Kaufmann.
5. J. J. Alferes and L. M. Pereira. Contradiction: when avoidance equal removal. In R. Dyckhoff, editor, *4th ELP*, volume 798 of *LNAI*. Springer–Verlag, 1994.
6. J. J. Alferes and L. M. Pereira. Update-programs can update programs. In J. Dix, L. M. Pereira, and T. Przymusinski, editors, *NMELP'96*. Springer, 1996.
7. J. J. Alferes and L. M. Pereira. Updates plus preferences. In M. O. Aciego et al., editor, *JELIA'00*. Springer LNAI 1919, 2000.
8. 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.
9. J. J. Alferes, L. M. Pereira, T. Przymusinski, H. Przymusinska, and P. Quaresma. Dynamic knowledge representation and its applications. In *AIMSA'00*. Springer LNAI, 2000.

10. C. Baral. Rule-based updates on simple knowledge bases. In *AAAI'94*, pages 136–141, 1994.
11. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *ICLP'99*. MIT Press, 1999.
12. C. V. Damásio and L. M. Pereira. Default negation in the heads: why not? In R. Dyckhoff et al., editor, *ELP'96*. Springer, 1996.
13. A. Darwiche and J. Pearl. On the logic of iterated belief revision. *Artificial Intelligence*, 89(1-2):1–29, 1997.
14. H. Decker. Drawing updates from derivations. In *Int. Conf on Database Theory*, volume 460 of *LNCS*, 1990.
15. M. Dekhtyar, A. Dikovsky, S. Dudakov, and N. Spyratos. Monotone expansions of updates in logical databases. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR'99*. Springer, 1999.
16. P. Dell'Acqua and L. M. Pereira. Updating agents. In S. Rochefort, F. Sadri, and F. Toni, editors, *ICLP'99 Workshop on Multi-Agent Systems in Logic*, 1999.
17. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on updates of logic programs. In M. O. Aciego, I. P. de Guzmn, G. Brewka, and L. M. Pereira, editors, *JELIA'00*. Springer LNAI 1919, 2000.
18. R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(2-3):189–208, 1971.
19. 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.
20. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *ICLP'90*. MIT Press, 1990.
21. M. Gelfond and V. Lifschitz. Action languages. *Linkoping Electronic Articles in Computer and Information Science*, 3(16), 1998.
22. L. Giordano and A. Martelli. Generalized stable models, truth maintenance and conflit resolution. In D. Warren and P. Szeredi, editors, *7th ICLP*, pages 427–441. MIT Press, 1990.
23. A. Guessoum and J. W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.
24. A. Guessoum and J. W. Lloyd. Updating knowledge bases II. *New Generation Computing*, 10(1):73–100, 1991.
25. K. Inoue and C. Sakama. Abductive framework for nonmonotonic theory change. In *IJCAI'95*, pages 204–210. Morgan Kaufmann, 1995.
26. K. Inoue and C. Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35:39–78, 1998.
27. 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.
28. A. Keller and M. Winslett Wilkins. On the use of an extended relational model to handle changing incomplete information. *IEEE Trans. on Software Engineering*, 11(7):620–633, 1985.
29. R. Kowalski. Legislation as logic programs. In *Logic Programming in Action*, pages 203–230. Springer-Verlag, 1992.
30. J. A. Leite, J. Alferes, and L. M. Pereira. Multi-dimensional dynamic logic programming. In F. Sadri and K. Satoh, editors, *CL-2000 Workshop on Computational Logic in Multi-Agent Systems (CLIMA'00)*, 2000.
31. J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In *LPKR'97: ILPS'97 workshop on Logic Programming and Knowledge Representation*, 1997.

32. 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.
33. F. Lin. Embracing causality in specifying the indirect effects of actions. In *IJCAI'95*, pages 1985–1991. Morgan Kaufmann, 1995.
34. V. Marek and M. Truszczyński. Revision specifications by means of programs. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *JELIA'94*, volume 838 of *LNAI*, pages 122–136. Springer-Verlag, 1994.
35. V. Marek and M. Truszczyński. Revision programming, database updates and integrity constraints. In *ICDT'95*, pages 368–382. Springer-Verlag, 1995.
36. Isaaco Newtono. *Philosophiæ Naturalis Principia Mathematica*. Editio tertia & aucta emendata. Apud Guil & Joh. Innys, Regiæ Societatis typographos, 1726. Original quotation: *"Corpus omne perseverare in statu suo quiescendi vel movendi uniformiter in directum, nisi quatenus illud a viribus impressis cogitur statum suum mutare."*.
37. T. Przymusinski and H. Turner. Update by means of inference rules. In V. Marek, A. Nerode, and M. Truszczyński, editors, *LPNMR'95*, volume 928 of *LNAI*, pages 156–174. Springer-Verlag, 1995.
38. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR'99*. Springer, 1999.
39. M. Winslett. Reasoning about action using a possible models approach. In *AAAI'88*, pages 89–93, 1988.
40. C. Witteveen and W. Hoek. Revision by communication. In V. Marek, A. Nerode, and M. Truszczyński, editors, *LPNMR'95*, pages 189–202. Springer, 1995.
41. C. Witteveen, W. Hoek, and H. Nivelle. Revision of non-monotonic theories: some postulates and an application to logic programming. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *JELIA'94*, pages 137–151. Springer, 1994.
42. Y. Zhang and N. Foo. Updating logic programs. In H. Prade, editor, *ECAI'98*. Morgan Kaufmann, 1998.