Dynamic Knowledge Representation and its Applications*

José Júlio Alferes[1], Luís Moniz Pereira[1], Halina Przymusinska[3]
Teodor C. Przymusinski[4], and Paulo Quaresma[1,2]

[1] Centro de Inteligência Artificial, Fac. Ciências e Tecnologia, Univ. Nova de Lisboa, P-2825-114 Caparica, Portugal, {jja,lmp}@di.fct.unl.pt
[2] Dept. de Informática, Univ. Évora, Rua Romão Ramalho, 59, P-7000 Évora, Portugal, pq@di.uevora.pt
[3] Department of Computer Science, California State Polytechnic University Pomona, CA 91768, USA, halina@cs.ucr.edu
[4] Department of Computer Science and Engineering, University of California Riverside, CA 92521, USA, teodor@cs.ucr.edu

Abstract. This paper has two main objectives. One is to show that the dynamic knowledge representation paradigm introduced in [ALP+00] and the associated language LUPS, defined in [APPP99], constitute *natural, powerful and expressive tools for representing dynamically changing knowledge*. We do so by demonstrating the applicability of the dynamic knowledge representation paradigm and the language LUPS to several broad knowledge representation domains, for each of which we provide an illustrative example.

Our second objective is to extend our approach to allow proper handling of *conflicting updates*. So far, our research on knowledge updates was restricted to a two-valued semantics, which, in the presence of conflicting updates, leads to an *inconsistent* update, even though the updated knowledge base does not necessarily contain any *truly contradictory* information. By extending our approach to the *three-valued semantics* we gain the added expressiveness allowing us to express *undefined* or *noncommittal* updates.

Keywords: Updates of Knowledge Bases, Dynamic Knowledge Representation, Generalized Logic Programs, Theory of Actions.

1 Introduction

One of the fundamental issues in *artificial intelligence* is the problem of *knowledge representation*. Intelligent machines must be provided with a precise definition of the knowledge that they possess, in a manner, which is independent of procedural considerations, context-free, and easy to manipulate, exchange and reason about.

* This work was partially supported by PRAXIS XXI project MENTAL, and a NATO scholarship while L. M. Pereira was on leave at the Department of Computer Science, University of California, Riverside. We thank João Leite for helpful discussions.

Any comprehensive approach to knowledge representation has to take into account the *inherently dynamic* nature of knowledge. As new information is acquired, new pieces of knowledge need to be dynamically added to or removed from the knowledge base. Such *knowledge updates* often not only significantly modify but outright contradict the information stored in the original knowledge base. We must therefore be able to *dynamically update* the contents of a knowledge base KB and generate a new, *updated knowledge base* KB^* that should possess a *precise meaning* and be *efficiently computable*.

1.1 Dynamic Knowledge Representation

In [ALP$^+$00] we proposed a comprehensive solution to the problem of knowledge base updates. Given the *original* knowledge base KB, and a set of update rules represented by the *updating* knowledge base KB', we defined a new *updated* knowledge base $KB^* = KB \oplus KB'$ that constitutes the *update of the knowledge base* KB *by the knowledge base* KB'. In order to make the meaning of the updated knowledge base $KB \oplus KB'$ declaratively clear and easily verifiable, we provided a *complete semantic characterization* of the updated knowledge base $KB \oplus KB'$. It is defined by means of a simple, *linear-time* transformation of knowledge bases KB and KB' into a *normal logic program* written in a *meta-language*. As a result, not only the update transformation can be accomplished very efficiently, but also query answering in $KB \oplus KB'$ is reduced to query answering about *normal logic programs*. The *implementation* is available at: `http://centria.di.fct.unl.pt/~jja/updates/`.

Forthwith, we extended the notion of a *single* knowledge base update to updates of sequences of knowledge bases, defining *dynamic knowledge base updates*. The idea of dynamic updates is very simple and yet quite fundamental. Suppose we are given a set of knowledge bases KB_s. Each knowledge base KB_s constitutes a knowledge update that occurs at some state s. Different states s may represent different time periods or different sets of priorities or perhaps even different viewpoints. The individual knowledge bases KB_s may therefore contain mutually contradictory as well as overlapping information. The role of the dynamic update KB^* of all the knowledge bases $\{KB_s : s \in S\}$, denoted by $\bigoplus \{KB_s : s \in S\}$, is to use the mutual relationships existing between different knowledge bases (as specified by the ordering relation on $s \in S$) to precisely determine the *declarative* as well as the *procedural* semantics of the combined knowledge base, composed of all the knowledge bases $\{KB_s : s \in S\}$.

Consequently, the notion of a dynamic program update allows us to represent dynamically changing knowledge and thus introduces the important paradigm of *dynamic knowledge representation*.

1.2 Language for Dynamic Representation of Knowledge

Knowledge evolves from one *knowledge state* to another as a result of *knowledge updates*. Without loss of generality we can assume that the initial, *default* knowledge state, KS_0, is empty[1]. Given the *current knowledge state* KS, its *successor knowledge state* $KS' = KS[KB]$ is generated as a result of the occurrence of a non-empty set of simultaneous (parallel) *updates,* represented by the *updating* knowledge base KB. Consecutive knowledge states KS_n can be therefore represented as $KS_0[KB_1][KB_2]...[KB_n]$, where KS_0 is the default state and KB_i's represent consecutive *updating knowledge bases.* Using the previously introduced notation, the n-th knowledge state KS_n is denoted by $KB_1 \oplus KB_2 \oplus ... \oplus KB_n$.

Dynamic knowledge updates, as described above, did not provide any *language* for specifying (or programming) changes of knowledge states. Accordingly, in [APPP99] we introduced a fully declarative, *high-level language for knowledge updates* called *LUPS* (*"Language of UPdateS"*) that describes transitions between consecutive knowledge states KS_n. It consists of *update commands,* which specify what updates should be applied to any given knowledge state KS_n in order to obtain the next knowledge state KS_{n+1}. In this way, update commands allow us to *implicitly* determine the *updating* knowledge base KB_{n+1}. The language LUPS can therefore be viewed as a *language for dynamic knowledge representation.* Below we provide a brief description of LUPS that does not include all of the available update commands and omits some details. The reader is referred to [APPP99] for a detailed description.

The simplest update command consists of adding a rule to the current knowledge state and has the form: *assert* $(L \leftarrow L_1, \ldots, L_k)$. For example, when a law stating that abortion is illegal is adopted, the knowledge state might be updated via the command: *assert* $(illegal \leftarrow abortion)$.

In general, the addition of a rule to a knowledge state may depend upon some preconditions being true in the current state. To allow for that, the assert command in LUPS has a more general form:

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

The meaning of this assert command is that if the preconditions $L_{k+1}, \ldots, L_m$ are true in the current knowledge state, then the rule $L \leftarrow L_1, \ldots, L_k$ should hold true in the successor knowledge state. Normally, the so added rules are *inertial,* i.e., they remain in force from then on by inertia, until possibly defeated by some future update or until retracted.

However, in some cases the persistence of rules by inertia should not be assumed. Take, for instance, the simple fact *alarm_ring.* This is likely to be a *one-time event* that should not persist by inertia after the successor state. Accordingly, the assert command allows for the keyword *event*, indicating that the added *rule* is *non-inertial.* Assert commands thus have the form (1) or[2]:

$$assert\ event\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{2}$$

Update commands themselves (rather than the rules they assert) may either be one-time, non-persistent update commands or they may remain in force until

[1] And thus in KS_0 all predicates are *false* by default.
[2] In both cases, if the precondition is empty we just skip the whole *when* subclause.

cancelled. In order to specify such *persistent update commands* (which we call *update laws*) we introduce the syntax:

$$always\ [event]\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{3}$$

To cancel persistent update commands, we use:

$$cancel\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{4}$$

To deal with rule deletion, we employ the *retraction* update command:

$$retract\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{5}$$

meaning that, subject to precondition $L_{k+1}, \ldots, L_m$, the rule $L \leftarrow L_1, \ldots, L_k$ is retracted. Note that cancellation of a persistent update command is very different from retraction of a rule. Cancelling a persistent update means that the given update command will no longer continue to be applied, but it does not remove any inertial effects of the rules possibly asserted by its previous application(s).

2 Application Domains

In this section we discuss and illustrate by examples the applicability of the *dynamic knowledge representation* paradigm and the language LUPS to several broad knowledge representation domains.

2.1 Reasoning about Actions

An exceptionally successful effort has been made lately in the area of *reasoning about actions.* Beginning with the seminal paper by Gelfond and Lifschitz [GL93], introducing a declarative language for talking about effects of actions *(action language $\mathcal{A}$)*,through the more recent paper of Giunchiglia and Lifschitz [GL98b] setting forth an enhanced version of the language (the so called *language $\mathcal{C}$),* a number of very interesting results have been obtained by several researchers significantly moving forward our understanding of actions, causality and effects of actions (see the survey paper [GL98a] for more details on action languages).

The theory of actions 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.* The *effect* 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 1 (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 [Lin95]. This situation is represented by the three persistent rules:

$$always\ (open \leftarrow up(l1), up(l2))$$
$$always\ (up(L))\ when\ (not\ up(L), toggle(L))$$
$$always\ (not\ up(L))\ when\ (up(L), toggle(L))$$

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

$$KS_1 = \{assert\ (not\ up(l1)), assert\ (up(l2)), assert\ (not\ open)\}$$

Suppose there are now two simultaneous toggling actions:

$$KS_2 = \{assert\ event\ (toggle(l1)), assert\ event\ (toggle(l2))\}$$

and afterwards another $l2$ toggling action: $KS_3 = \{assert\ event\ (toggle(l2))\}$. In the knowledge state 2 we'll have $up(l1), not\ up(l2)$ and the suitcase is not open. Only after KS_3 will latch $l2$ be up and the suitcase open.

However, there are also *major differences* between dynamic updates of knowledge and theories of actions. While in our approach we want to be able to update one knowledge base by an arbitrary *set of rules* that constitutes the *updating knowledge base,* action languages deal only with updates of *propositional knowledge states.* At the *semantic level*, however, the situation is not so simple. The main motivation behind the introduction of the language $\mathcal{C}$ was to be able to express the notion of *causality.* This is a very different motivation from the motivation that we used when defining the semantics of updated knowledge bases.

In spite of these differences, the strong similarities between the two approaches clearly justify a serious effort to *investigate the exact nature of the close relationship between the two research areas* and between the respective families of languages, their syntax and semantics.

2.2 Legal Reasoning

Robert Kowalski and his collaborators did a truly outstanding research work on using logic programming as a *language for legal reasoning* (see e.g. [Kow92]). 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*. Dynamic knowledge representation 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 of LUPS on the following simple example. Another, slightly more ellaborate example, was given in [APPP99].

Example 2 (Conscientious objector). Consider the 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.

$$\begin{array}{ll} KS_1: & always\ (draftable(X))\ when\ (of\_age(X)) \\ & assert\ (conscripted(X) \leftarrow draftable(X), healthy(X)) \\ KS_2: & assert\ (healthy(a)).\ assert\ (healthy(b)).\ assert\ (of\_age(b)). \\ & assert\ (consc\_objector(a)).\ assert\ (consc\_objector(b)) \\ KS_3: & assert\ (not\ conscripted(X) \leftarrow consc\_objector(X)) \\ KS_4: & assert\ (of\_age(a)) \end{array}$$

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

In addition to providing automatic updating, dynamic knowledge representation allows us to keep the entire *history* of past changes and to query the knowledge base *at any given time in the past*. The ability to keep track of all the *past* changes in the law is a feature of crucial importance in the domain of law. We expect, therefore, that by using LUPS as a language for representation and reasoning about legal knowledge we may be able to significantly improve upon the work based on standard logic programming.

2.3 Software Specifications

One of the most important problems in *software engineering* is the problem of choosing a suitable *software specification language*. It has been argued in several papers (see e.g. [LO97,EDD93,FD93]) that the language of *logic programming* is a good potential candidate for the language of software specifications. However logic programming lacks simple and natural ways of expressing conditions that change dynamically and the ability to handle inconsistencies stemming from specification revisions. Another problem is called *elaboration tolerance* and requires that small modifications of informal specifications result in *localized and simple modifications* of their formal counterparts. Dynamic knowledge representation based on generalized logic programs extends logic programming exactly with these two missing dynamic update features. Moreover, small informal specification revisions require equally small modifications of the formal specification, while all the remaining information is *preserved by inertia*. The following *banking example* illustrates the above claims.

Example 3 (Banking transactions). Consider a software specification for performing banking transactions. Account balances are modelled 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 modelled in LUPS by KS_1:

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

The first two rules state how to update the balance of an account, given any event of $updateBal$. Deposits and withdrawals are then 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.:

$$
\begin{array}{l}
KS_2 : \{assert\ (balance(1, 0)), assert\ (balance(2, 50))\} \\
KS_3 : \{assert\ event\ (deposit(1, 40)), assert\ event\ (withdrawal(2, 10))\}
\end{array}
$$

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 modelled by the sequence:

$KS_4 : assert\ (not\ updateBal(Ac, X) \leftarrow deposit(Ac, X), X < 50)$

$KS_5 : assert\ (updateBal(Ac, -X) \leftarrow vip(Ac, L), withdrawal(Ac, X),$
$\qquad balance(Ac, B), B + L \geq X)$

$KS_6 : assert\ (vip(1, 200))$

$KS_7 : assert\ (updateBal(Ac, X) \leftarrow deposit(Ac, X), balance(Ac, B), B < 0)$

This shows dynamic knowledge representation constitutes a powerful tool for software specifications that will prove helpful in the difficult task of building reliable and provably correct software.

3 Representation of Conflicting Knowledge

Let us consider the following *contradictory advice* example, which models a situation where an agent receives *conflicting* advice from two reliable authorities. Since the agent's expected behaviour is *not* to do anything that he was advised not to do by a reliable authority, the agent should neither perform the given action nor refuse to do it. Instead, the agent should remain *non-committal* and the outcome of his decision process should therefore be *undefined.*

Example 4 (Conflicting Advice). An agent receives advice from two reliable sources: his father and his mother. The agent's expected behaviour is to perform an action recommended by a reliable authority unless it is in conflict with the advice received from another authority.

$$
\begin{array}{l}
always\ (do(A) \leftarrow father\_advises(A), not\ dont(A)) \\
always\ (dont(A) \leftarrow mother\_advises(noA), not\ do(A)) \\
always\ (\perp \leftarrow do(A), mother\_advises(noA)) \\
always\ (\perp \leftarrow dont(A), father\_advises(A))
\end{array}
$$

Suppose the father advises buying stocks but the mother advises not to do so:

$KS_1 = \{assert\ event\ (father\_advises(buy)), assert\ event\ (mother\_advises(nobuy))\}$

In this situation, the agent is unable to choose either $do(buy)$ or $dont(buy)$ and, as a result, does not perform any action whatsoever.

The above illustrates the need for a *3-valued semantics* for knowledge updates. So far, in our research on knowledge updates, we were exclusively using a 2-valued semantics, namely, the *stable semantics* [GL88],suitably extended to the class of generalized logic programs [3]. Under the 2-valued semantics, the

[3] The class of generalized logic programs can be viewed as a special case of a yet broader class of programs introduced earlier in [LW92].

above situation results in an *inconsistent* update, because of integrity constraint violation. In this section we extend our approach to the (3-valued) *well-founded semantics* of generalized logic programs. This will enable us to model knowledge updates with *non-committal* or *undefined* outcome, as required.

Recall that both the dynamic updates and the LUPS semantics can be defined by means of linear-time transformations into generalized logic programs. The transformation encodes both the declarative meaning of the update and the inertia rules. To generalize both semantics to a 3-valued setting, one needs to extend the semantics of normal logic programs with default negation in the heads to a 3-valued setting. The resulting update program semantics is based on the well-founded semantics instead of the stable models. Accordingly, below we generalize the well-founded semantics of normal logic programs to generalized logic programs.

We start by presenting the definition of the stable model semantics of generalized logic programs[4].

Definition 1 (Generalized Logic Program). *A* generalized logic program *P in the language $\mathcal{L}$ is a set of rules of the form $L \leftarrow L_1, \ldots, L_n$, where L and L_i are literals. A literal is either an atom A or its default negation not A. Literals of the form not A are called default literals. If none of the literals appearing in heads of rules of P are default literals, then the logic program P is* normal*.*

In order to define the semantics of generalized programs, we start by eliminating all default literals in the heads of rules.

Definition 2. *Let $\overline{\mathcal{L}}$ be the language obtained from the language $\mathcal{L}$ of a generalized logic program P by adding, for each propositional symbol A, the new symbol $\overline{A}$. $\overline{P}$ is the normal program obtained from the generalized program P through replacing every negative head not A by $\overline{A}$.*

The definition of the stable models of generalized programs can now be gotten from the stable models of the program $\overline{P}$. The idea is quite simple: since $\overline{P}$ is a normal program, its stable models can be identified via the usual definition by means of the Gelfond-Lifschitz operator Γ [GL88]; afterwards, all it remains to be done is to interpret the $\overline{A}$ atoms in the stable models of $\overline{P}$ as the default negation of A. Since atoms of the form $\overline{A}$ never appear in the body of rules of $\overline{P}$, this task is trivial: if $\overline{A}$ is true in a stable model then *not* A must also be true in it (i.e. A cannot belong to the stable model); if $\overline{A}$ is false in a stable model, then no rule in P concludes *not* A, and so the valuation of A in the stable model is independent of the existence of rules for $\overline{A}$.

Definition 3 (Stable models of generalized programs). *Let P be a generalized logic program, and let I be a stable model of $\overline{P}$ (i.e. I be such that $I = \Gamma_{\overline{P}} I$) such that for no atom A both A and $\overline{A}$ belong to I. The model M, obtained from I by deleting from it all atoms of the form $\overline{A}$, is a stable model of P.*

[4] The definition below is different from the original one in [ALP$^+$00], but their equivalence can easily be shown given the results in [DP96]

Now, a naïve extension of the well-founded semantics to generalized programs would simply consider the fixpoints of the compound operator Γ^2 for the transformed programs $\overline{P}$, and then remove all fixpoints where, for some atom, both A and $\overline{A}$ held. In fact, for normal programs, the least fixed-point of Γ^2 characterizes the well-founded semantics. However, this naïve definition does not engender intuitive results:

Example 5. Consider the generalized program $P = \{not\, a;\ a \leftarrow not\, b;\ b \leftarrow not\, a\}$. According to the naïve semantics, the well-founded model would be $\{not\, a\}$. In this case, since $not\, a$ is true, one would expect b to be true as well.

In the definition of stable models for generalized programs, whenever an atom $\overline{A}$ is true in some interpretation I (in the extended language $\overline{\mathcal{L}}$), and hence by definition $A \notin I$, it is guaranteed that after applying the Γ-operator once all occurrences of $not\, A$ are removed from rule bodies. In other words, whenever $\overline{A}$ is true, A is assumed false by default in rule bodies. In the well-founded semantics, one must ensure that, whenever $\overline{A}$ belongs to a fixed-point of Γ^2, all literals $not\, A$ in the bodies must be true. In other words, whenever $\overline{A}$ belongs to a fixed-point I of Γ^2, A must not belong to $\Gamma(I)$. This is achieved by resorting to the semi-normal version of the program:

Definition 4 (Semi-normal program). *The semi-normal version $\overline{Ps}$ of a normal program $\overline{P}$ is obtained by adding to the body of each rule in $\overline{P}$ with head A (resp. $\overline{A}$) the literal $not\, \overline{A}$ (resp. $not\, A$).*

Definition 5 (Partial stable models of generalized programs). *Let I be a set of atoms in the language $\overline{\mathcal{L}}$ such that:*

$$(1)\ I = \Gamma_{\overline{P}}(\Gamma_{\overline{Ps}}(I)) \quad \text{and} \quad (2)\ I \subseteq \Gamma_{\overline{Ps}}(I)$$

The 3-valued model $M = T \cup not\, F$ is a partial stable model of the program P, where $not\,\{A_1, \ldots, A_n\}$ stands for $\{not\, A_1, \ldots, not\, A_n\}$, and T is obtained from I by deleting all atoms of the form $\overline{A}$ and F is the set of all atoms A that do not belong to $\Gamma_{\overline{Ps}}(I)$.

With this definition there is no need to explicitly discard interpretations comprising both A and $\overline{A}$ for some atom A. These are already filtered by condition (2). Indeed, if both A and $\overline{A}$ belong to I then, because in $\overline{Ps}$ all rules with head A (respectively, $\overline{A}$) have $not\, \overline{A}$ (respectively, $not\, A$) in the body, neither A nor $\overline{A}$ belong to $\Gamma_{\overline{Ps}}(I)$, and thus condition (2) will fail to hold.

Definition 6 (Well-founded model of generalized programs). *The well founded model of a generalized program P is the set-inclusion least partial stable model of P, and is obtainable by iterating the (compound) operator $\Gamma_{\overline{P}}\Gamma_{\overline{Ps}}$ starting from $\{\}$, and constructing M from the so obtained least fixpoint.*

Example 6. The well-founded model of the program in example 5 is $\{b, not\, a\}$. In fact, $\Gamma_{\overline{P}_s}(\{\}) = \{a, b, \overline{a}\}$, $\Gamma_{\overline{P}}(\{a, b, \overline{a}\}) = \{\overline{a}\}$, $\Gamma_{\overline{P}_s}(\{\overline{a}\}) = \{b, \overline{a}\}$, $\Gamma_{\overline{P}}(\{b, \overline{a}\}) = \{b, \overline{a}\}$. Accordingly, its well-founded model is $\{b, not\, a\}$. Note, in the 3rd application of the operator, how the semi-normality of P is instrumental in guaranteeing truth of b.

4 Concluding Remarks

While LUPS constitutes an important step forward towards defining a powerful and yet intuitive and fully declarative language for dynamic knowledge representation, it is by far not a finished product. There are a number of update features that are not yet covered by its current syntax as well as a number of additional options that should be made available for the existing commands. Further improvement, extension and application of the LUPS language remains therefore one of our near-term objectives.

References

[ALP+00] José J. Alferes, João A. Leite, Luís M. Pereira, Halina Przymusinska, and Teodor C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, page (to appear), 2000. Extended abstract appeared in *KR'98*, pages 98-111. Morgan Kaufmann, 1998.

[APPP99] José J. Alferes, Luís M. Pereira, Halina Przymusinska, and Teodor C. Przymusinski. LUPS - a language for updating logic programs. In *LPNMR'99*, Lecture Notes in AI 1730, pages 162–176. Springer, 1999.

[DP96] C. V. Damásio and Luís M. Pereira. Default negated conclusions: why not ? In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Int. Workshop on Extensions of Logic Programming (ELP'96)*, number 1050 in LNAI, pages 103–117, 1996.

[EDD93] Abdel Ali Ed-Dbali and Pierre Deransart. Software formal specification by logic programming: The example of standard PROLOG. Research report, INRIA, Paris, France, 1993.

[FD93] Gerard Ferrand and Pierre Deransart. Proof method of partial correctness and weak completeness for normal logic programs. Research report, INRIA, Paris, France, 1993.

[GL88] 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.

[GL93] Michael Gelfond and Vladimir Lifschitz. Representing Actions and Change by Logic Programs. *Journal of Logic Programming*, 17:301–322, 1993.

[GL98a] M. Gelfond and V. Lifschitz. Action languages. *Linkoping Electronic Articles in Computer and Information Science*, 3(16), 1998.

[GL98b] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings AAAI-98*, pages 623–630, 1998.

[Kow92] R. Kowalski. Legislation as logic programs. In *Logic Programming in Action*, pages 203–230. Springer–Verlag, 1992.

[Lin95] F. Lin. Embracing causality in specifying the indirect effects of actions. In *IJCAI'95*, pages 1985–1991. Morgan Kaufmann, 1995.

[LO97] K. K. Lau and M. Ornaghi. The relationship between logic programs and specifications. *Journal of Logic Programming*, 30(3):239–257, 1997.

[LW92] Lifschitz and Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *KR'92*. Morgan-Kaufmann, 1992.