An Exercise with Dynamic Knowledge Representation

José Júlio Alferes
Centro de Inteligência Artificial (CENTRIA)
FCT, Univ. Nova de Lisboa
P-2825-114 Caparica, Portugal
jja@di.fct.unl.pt

Luís Moniz Pereira
CENTRIA
FCT, Univ. Nova de Lisboa
P-2825-114 Caparica, Portugal
lmp@di.fct.unl.pt

Halina Przymusinska
Dept. Computer Science
California State Polytechnic Univ.
Pomona, CA 91768, USA
halina@cs.ucr.edu

Teodor C. Przymusinski
Dept. Computer Science
Univ. of California
Riverside, CA 92521, USA
teodor@cs.ucr.edu

Paulo Quaresma
Univ. Évora, and CENTRIA
R. Romão Ramalho, 59
P-7000 Évora, Portugal
pq@di.uevora.pt

Abstract

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 [APPP99] we introduced a fully declarative, *high-level language for knowledge updates* called *LUPS* that describes *transitions between consecutive knowledge states* and can therefore be viewed as a *language for dynamic knowledge representation.*

The main objective of the present paper 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

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.

paradigm and the language LUPS to several broad knowledge representation domains, for each of which we provide an illustrative example. In [APP+99], we presented a preliminary exploration in the application of this paradigm to examples in the domain of action. Here we show the application to other domains, and also add some more insights in its application to the actions domain. All of the examples have been run and tested under our *implementation* of the LUPS language.

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. A knowledge engineer also needs to have a precise description (specification) of the *knowledge base (KB)* that he/she is expected to design in order to be able to verify that the design meets all the requirements provided by the end user. Due to the great complexity and high cost of knowledge engineering, serious design errors may be difficult or impossible to correct without expensive modifications of the entire knowledge base or of its major parts.

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*.

Most of the work conducted in the past in the field of knowledge representation has focused either on representing *static knowledge*, i.e., knowledge that does not evolve with time, or, on updates of purely *extensional* knowledge bases (i.e., knowledge bases consisting entirely of facts rather than rules). This poses a serious limitation since in a majority of knowledge bases not only the extensional part dynamically changes but so does the *intensional* part (i.e., the set of rules).

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*. Given individual and largely *independent* knowledge bases KB_s describing knowledge updates at different states of the world (for example, new knowledge acquired at different times), the dynamic update

$$\bigoplus\{KB_s \ : \ s \ \in S \ \}$$

specifies the exact meaning of the union of these knowledge bases. Dynamic knowledge representation significantly facilitates *modularization* of knowledge and, thus, modularization of non-monotonic reasoning as a whole.

Whereas traditional knowledge bases concerned themselves mostly with representing static knowledge, we showed how to use knowledge base updates to represent dynamically changing knowledge. In [ALP+00] we restricted our investigation to knowledge bases represented by *generalized logic programs*. Logic programs constitute a relatively simple and yet quite broad and expressive class of non-monotonic knowledge bases, and thus provide an excellent application domain for research on the theory and implementation of knowledge base updates. They not only have a well-defined and precise semantics, but also constitute *reasonably efficient programs*, which can be executed and tested without being first compiled into some other language. Generalized logic programs permit (default) *negation* not only in rule bodies but also in their heads, which allows us to specify that some atoms or rules must become false, i.e., must be deleted or retracted.

It is worth noting why, in the dynamic knowledge representation setting, generalizing the language to allow default negation in rule heads is more adequate than introducing explicit negation in programs (both in heads and bodies). Suppose we are given a rule stating that A is true whenever some condition $Cond$ is met. This is naturally represented by the rule $A \leftarrow Cond$. Now suppose we want to say, as an update, that A should no longer be the case (i.e. should be deleted or retracted), if some condition $Cond'$ is met. How to represent this new knowledge? By using

extended logic programming (with explicit negation) this could be represented by $\neg A \leftarrow Cond'$. But this rule says more than we want to. It states that A is false upon $Cond'$, and we only want to go as far as to say that the truth of A is to be deleted in that case. All is wont to be said is that, if $Cond'$ is true, then $not\, A$ should be the case, i.e. $not\, A \leftarrow Cond'$. As argued in [GL90], the difference between explicit and default negation is fundamental whenever the information about some atom A cannot be assumed to be complete. Under these circumstances, the former means that there is evidence for A being false, while the latter means that there is no evidence for A being true. In the deletion example, we desire the latter case. Note, however, that the adequacy of generalized logic program for this purpose is in facilitating the intuitive writing of updates. Indeed, as proven in [DP96], generalized logic programs and extended logic programs have the same expressive power.

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

$$KB_1 \oplus KB_2 \oplus ... \oplus KB_n.$$

However, in reality, the updating knowledge bases KB_i (i.e., the actual updates) are not given *explicitly* but are instead generated *implicitly* as a result of some *update laws,* which define *transitions* between consecutive knowledge states KS_n. For example, an update law that states *"wake up when alarm rings"* results in *"wake-up"* being true in the next knowledge state KS_{n+1} if *"alarm rings"* is true in the current state KS_n. Note that such an update command cannot be simply represented by adding to the knowledge state KS_{n+1} a new rule $wake\_up \leftarrow alarm\_rings$. 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 (until retracted) even after the alarm stops ringing, i.e., from then on, no direct connection between $wake\_up$ and $alarm\_rings$ should persist.

Dynamic knowledge updates, as described above, did not provide any *language* for specifying (or programming) changes of knowledge states. In other words, they did not provide any means of describing in a formal and declarative way *what* makes knowledge evolve and *how* knowledge evolves.

[1] And thus in KS_0 all predicates are *false* by default.

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*[2] 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.* A program written in LUPS will be called an *update program.* 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:

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

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 $L \leftarrow L_1, \ldots, L_k$ are *inertial,* i.e., they remain in force from then on by inertia, until possibly defeated by some future update or until retracted. This is for example the case for the rule $illegal \leftarrow abortion$, which remains in effect by inertia beginning from the successor state onwards, unless later invalidated (i.e. until the rule is retracted or a rule with true body and *not illegal* in the head is later asserted).

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[3]:

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

[2]It is perhaps useful to remark at this point that in *imperative programming* the programmer specifies only transitions between different knowledge states while leaving the actual (resulting) knowledge states implicit and thus highly imprecise and difficult to reason about. On the other hand, dynamic knowledge updates, as described above, enabled us to give a precise and fully declarative description of actual knowledge states but did not offer any mechanism for specifying state transitions. With the high-level language of dynamic updates, we are able to make *both* the knowledge states and their transitions fully declarative and precise.

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

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 cancelled, and we must distinguish the two cases syntactically. For example, an update command like:

$$\textbf{assert } \mathit{illegal} \leftarrow \mathit{abortion} \textbf{ when } \mathit{republican\_congress}, \mathit{republican\_president}$$

or

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

could be intended as a one ocasion command triggered once by the when clause, or it could be intended as a command to be triggered everytime the when clause is true, or at least remain true until cancelled. This means, in the latter case, that the update command applies to all the *subsequent* knowledge states (or until cancelled) rather than just to the current one. Such persistent update commands enable knowledge states to *dynamically evolve* without the occurrence of any truly new external updates, though that is not the case in this example. Knowledge bases subject to persistent updates become in effect *active knowledge bases*, because a command may produce an update that triggers another command, or even itself. For example, an intendedly persistent update command:

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

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

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

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

To disable persistent update commands, we use:

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

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, which may be formally represented by *assert* (*true*), ensures that the system continues to evolve, even when no truly new updates are specified. It represents a *tick of the clock* or a *virtual timer* operating within the modelled world.

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

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

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).

Knowledge can be queried at any knowledge state KS_q with:

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

and is true if and only if the conjunction of its literals holds at the state KS_q. If the state is implicitly known, we just skip the phrase "**at state** q".

In [APPP99] we provided a precise *declarative and procedural semantics* for the language LUPS so that the update commands not only admit a definite declarative meaning but also can be readily implemented. The declarative semantics is obtained by translating a LUPS program into a dynamic update $KB_0 \oplus \cdots \oplus KB_n$ discussed above. The procedural semantics for LUPS is obtained by the translation of the LUPS program into a *normal logic program,* written in a *meta-language*. The translation is *linear-time* and thus it offers an efficient mechanism for implementing LUPS programs: after a pre-processor performs the necessary linear-time translation, query answering in LUPS is reduced to query answering in normal logic programs. The complete implementation of LUPS is available at:

`http://centria.di.fct.unl.pt/~jja/updates/lups.p`.

1.3 Main Objective of the Paper

The main objective of this paper is to show that the new dynamic knowledge representation paradigm and the language LUPS represent *natural, powerful and expressive tools for representing dynamically changing knowledge*. We do so by demonstrating the applicability of the dynamic knowledge representation paradigm and LUPS to several broad knowledge representation domains. In [APP+99] we have explored the application of LUPS to the domain of actions. Here we enlarge to scope of application domains by including: *active databases, legal reasoning, intelligent agents, and software specifications.* They are not intended by any means to constitute an exhaustive list of potential application domains but just to serve as their *sample representatives.*

We argue that all of these domains (and many others) can significantly benefit from the dynamic knowledge representation paradigm introduced in [ALP+00] and from the expressive power of the language LUPS defined in [APPP99]. For each one of the selected application domains we present an illustrative example, each of which having been run and tested under our *implementation* of the LUPS.

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 Active Knowledge Bases

Persistent update laws allow us to handle not only knowledge states that dynamically change due to *newly received updates*, but they also enable us to model the much more involved case of *self-updating* or *active knowledge bases,* which undergo change even though no truly new updates occurred. For example, the watch's hands moves whether new updates are received or not. Since the high-level language of updates LUPS outlined above has a built-in capability to define persistent updates, it permits us to *model active knowledge bases.* The following example illustrates the language's ability to handle persistent updates.

Example 1 (Heating) *When a certain set minimum temperature is reached, the heater is switched on. This triggers a gradual increase in the temperature of the room, for a while. The heater is switched off when the estimated temperature reaches a maximum set value. The temperature will the gradually decrease, until an estimated minimum value is reached. We assume that from state to state the estimated increase or decrease of the temperature is fixed by an absolute amount* Δ.

This situation can be modelled in LUPS, by the following persistent rules:

$$\textbf{\textit{always event}}\ temperature(V-\Delta)\ \textbf{\textit{when}}\ not\ on,\ temperature(V)$$
$$\textbf{\textit{always event}}\ temperature(V+\Delta)\ \textbf{\textit{when}}\ on,\ temperature(V)$$
$$\textbf{\textit{always}}\ not\ on\ \textbf{\textit{when}}\ temperature(V),\ setMax(M),\ V > M$$
$$\textbf{\textit{always}}\ on\ \textbf{\textit{when}}\ temperature(V),\ setMin(M),\ V < M$$

Note that the current temperature is represented as an event. This is so because, in this example, its value does not persist by inertia. Given an initial temperature by a command ***assert event*** $temperature(aValue)$*, and the set minimum and maximum values by* ***assert*** $setMin(min)$ *and* ***assert*** $setMax(max)$*, the system starts and then evolves by itself. For example, if the initial value of the temperature is less than the minimum, the predicate on is asserted, and the temperature will increase until the maximum is surpassed. Conversely, when this occurs, on is retracted, and the temperature begins to decrease ...*

The problem of building, querying and modifying *active databases* is studied by many researchers in the database community, and is considered to be an important research topic. However, to the best of our knowledge, none of the approaches proposed so far in the literature resulted in a solution based on *clear and precise declarative semantics* while retaining the immediate availability of *procedural execution.* We believe therefore that our approach is likely to have a major impact on the ongoing research in this area by helping to precisely define both the declarative and the procedural meaning of the notion of active database.

2.2 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). These recent papers also significantly influenced our own work on dynamic knowledge updates.

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.* 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.*

Example 2 (Yale Shooting) *This well know example from the theory of actions can be represented in LUPS by adding the following persistent rules:*

$$\begin{array}{l} \textbf{\textit{always}}\ loaded\ \textbf{\textit{when}}\ load \\ \textbf{\textit{always}}\ dead\ \textbf{\textit{when}}\ shoot, loaded \\ \textbf{\textit{always}}\ not\ loaded\ \textbf{\textit{when}}\ shoot \\ \textbf{\textit{always}}\ not\ alive \leftarrow dead \end{array}$$

The initial situation, where the turkey is alive, is represented by adding ***assert*** *alive to KS_1.*

After some time, there is an action of loading the gun. This is easily represented by adding to the corresponding KS_i (where $i > 1$) the command ***assert event*** *load. After waiting a while, there is an action of shooting, which is represented by adding to the corresponding KS_j (where $j > i + 1$) the command* ***assert event*** *shoot.*

It is easy to check that from KS_j onwards dead and not alive hold in all stable models.

The following *Mary's soup* example, adapted from [GLR91], illustrates how LUPS can be used to handle parallel updates.

Example 3 (Mary's soup) *There is a dish of soup. To lift it in a stable manner, and avoid spilling, it is required to hold it with both hands. Actions of grabbing and releasing are available for, respectively, initiating and terminating the holding, with left or the right hands. This situation is represented by the commands:*

$$\begin{array}{l} \textbf{\textit{always}}\ stable \leftarrow holding(left), holding(right) \\ \textbf{\textit{always}}\ spilling \leftarrow not\ stable \\ \textbf{\textit{always}}\ holding(H)\ \textbf{\textit{when}}\ grab(H) \\ \textbf{\textit{always}}\ not\ holding(H)\ \textbf{\textit{when}}\ release(H) \end{array}$$

In the initial situation the dish of soup is being held by the left hand alone:

$$KS_1 = \{\textbf{\textit{assert}}\ not\ holding(right), \textbf{\textit{assert}}\ holding(left)\}$$

Suppose there are now two simultaneous actions, of grabbing with the right hand and releasing the left:

$$KS_2 = \{\textbf{\textit{assert event}}\ grab(right), \textbf{\textit{assert event}}\ release(left)\}$$

and afterwards an action of grabbing with the left hand:

$$KS_3 = \{\textbf{\textit{assert event}}\ grab(left)\}$$

In the knowledge state 2 *we will have* $holding(right)$*, not* $holding(left)$*, the dish is not stable and spilling. Only after* KS_3 *will the dish be stable and not spilling.*

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.* In other words, action languages are limited to purely *atomic* assertions and retractions, and thus are dealing exclusively with purely *extensional* (or relational) knowledge bases. Our approach also allows us to model *self-updating* or *active* knowledge bases that are capable of undergoing change *without any outside triggers* at all. As a result, from a purely syntactic point of view, our language for knowledge updates, LUPS, is strictly more expressive than action languages $\mathcal{A}$ or $\mathcal{C}$.

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. Here the main principle was to inherit as much information as possible from the previous knowledge state while changing only those rules that truly have to be affected by the given update(s). As a result, one can easily see that, even in simple cases, the semantics of knowledge updates and that of action languages often differ.

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. Hopefully, we will be able to *bridge the gap* between these two intriguing and closely related research areas.

2.3 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.

Example 4 (Abortion Law) *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) Gore is elected President and Democratic Congress is in place (year 2000?); (8) Susan performs abortion.

The specification in LUPS would be:

$$\begin{array}{l} \%\ \textit{Persistent update commands} \\ \textbf{\textit{always}}\ jail(X) \leftarrow abortion(X)\ \textbf{\textit{when}}\ repC \wedge repP \\ \textbf{\textit{always}}\ not\, jail(X) \leftarrow abortion(X)\ \textbf{\textit{when}}\ not\, repC \wedge not\, repP \end{array}$$

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

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

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 $jail(X) \leftarrow abortion(X)$ 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.

$$\begin{array}{ll} \%\ \textit{Sequence of non-persistent update commands} & \\ U_1: & \textbf{\textit{assert}}\ repP \\ & \textbf{\textit{assert}}\ not\, repC \\ U_2: & \textbf{\textit{assert event}}\ abortion(mary) \\ U_3: & \textbf{\textit{assert}}\ repC \\ U_4: & \textbf{\textit{assert event}}\ abortion(kate) \\ U_5: & \textbf{\textit{assert}}\ not\, repP \\ U_6: & \textbf{\textit{assert event}}\ abortion(ann) \\ U_7: & \textbf{\textit{assert}}\ not\, repC \\ U_8: & \textbf{\textit{assert event}}\ abortion(susan) \end{array}$$

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

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.

2.4 Intelligent Agents

Similar comments apply to the area of *intelligent agent* and *multi-agent reasoning.* The ability of dynamic knowledge representation to naturally express continuously changing state of knowledge seems to be particularly valuable for modelling behaviour of intelligent agents whose perception of the world continuously changes and who operate in a perpetually changing environment. It is also invaluable for representing *multi-agent communication* and reasoning which is also characterized by continuous changes of beliefs about beliefs of other agents as well as about agents' own beliefs. The following simple example illustrates the use of the language LUPS to model such simple exchange of information between *intelligent and cooperative agents.*

Example 5 (Cooperative agents) *Here we model a simple dialogue where agent A informs another agent B about the truth of a property P that contradicts B's previous beliefs. We assume that agents are* cooperative. *We begin by describing how the information is communicated between the agents [PQ98]. First, we specify the effect of the inform action:*

$$\textbf{\textit{always}}\ bel(Listener, bel(Speaker, P))\ \textbf{\textit{when}}\ inform(Speaker, Listener, P)$$

i.e., after the speaker informs the listener that P is true, the listener believes the speaker believes in P. Next, we describe how new beliefs are formed:

$$\textbf{\textit{always}}\ bel(A, P)\ \textbf{\textit{when}}\ bel(A, bel(B, P))$$
$$\textbf{\textit{always}}\ not\ bel(A, Q) \leftarrow incompatible(P, Q)\ \textbf{\textit{when}}\ bel(A, Q), bel(A, bel(B, P))$$

These rules mean that a cooperative agent always accepts known beliefs of the other agent and eliminates his previous beliefs if they are incompatible with the new ones. For instance, it may be incompatible for an agent to be at two different places simultaneously:

$$\textbf{\textit{always}}\ incompatible(at(P1, A), at(P2, A)) \leftarrow P1 \neq P2$$

Suppose that in state 1 *agent a believes agent b is at the hospital:*

$$KS_1 : \textbf{\textit{assert}}\ bel(a, at(hospital, b))$$

But agent c informs a that b is at home:

$$KS_2 : \textbf{\textit{assert event}}\ inform(c, a, at(home, b))$$

In state 2, $bel(a, at(home, b))$ *will hold as a consequence of the first update command and the belief forming commands rule.* $bel(a, at(hospital, b))$, *on the other hand, will no longer hold.*

Depending on the definition of incompatible contradiction may arise of course. Contradiction avoidance, or its detection and removal, are different mechanisms which could be added, but we do not deal with them in this paper.

2.5 Software Specifications

One of the most important problems in *software engineering* is the problem 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 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. While logic programming clearly possesses the first three properties, it lacks simple and natural ways of expressing conditions that change dynamically and the ability to handle inconsistencies stemming from specification revisions. The last 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 6 (Banking transactions) *Consider a software specification for performing banking transactions. Predicate* $balance(AccountNo, Balance)$ *is used to model account balances. To represent the actions of depositing and withdrawing money into and out of an account we use, respectively,* $deposit(AccountNo, Amount)$ *and* $withdrawal(AccountNo, Amount)$*. 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*:*

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

The first two rules state how to update the balance of an account (i.e. remove its old value, and replace it by the new one) , 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., the following causes the balance of both accounts 1 and 2 to be 40, after state 3.:*

KS_2 : $\{\textbf{\textit{assert}}\ balance(1,0), \textbf{\textit{assert}}\ balance(2,50)\}$
KS_3 : $\{\textbf{\textit{assert event}}\ deposit(1,40), \textbf{\textit{assert event}}\ withdrawal(2,10)\}$

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 : $\textbf{\textit{assert}}\ not\ updateBal(Ac,X) \leftarrow deposit(Ac,X), X < 50$
KS_5 : $\textbf{\textit{assert}}\ updateBal(Ac,-X) \leftarrow vip(Ac,L), withdrawal(Ac,X), balance(Ac,B), B + L \geq X$
KS_6 : $\textbf{\textit{assert}}\ vip(1,200)$
KS_7 : $\textbf{\textit{assert}}\ updateBal(Ac,X) \leftarrow deposit(Ac,X), balance(Ac,B), B < 0$

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 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*, 45(1-3), pages 43-70, 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.

[APP+99] José J. Alferes, Luís M. Pereira, Halina Przymusinska, Teodor C. Przymusinski, and Paulo Quaresma. Preliminary exploration on actions as updtaes. In M. C. Meo, editor, *AGP'99*, 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.

[GL90] M. Gelfond and V. Lifschitz. Logic Programs with classical negation. In Warren and Szeredi, editors, *ICLP'90*. MIT Press, 1990.

[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.

[GLR91] M. Gelfond, V. Lifschitz and A. Rabinov What are the limitation of the situation calculus. In R. Moore, editor, *Automated Reasoning: essays in honour of Woody Bledsoe*, pages 167–179. 1991.

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

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

[PQ98] L. M. Pereira and P. Quaresma. Modelling agent interaction in logic programming. In Osamu Yoshie, editor, *INAP'98 - The 11th International Conference on Applications of Prolog*, pages 150–156, Tokyo, Japan, September 1998. Science University of Tokyo.