Planning as Abductive Updating

José Júlio Alferes[⋆†]; João Alexandre Leite[†]
Luís Moniz Pereira[†]; Paulo Quaresma[⋆†]
[⋆]Universidade de Évora, R. Romão Ramalho, 59, 7000 Évora, Portugal
[†]Centro de Inteligência Artificial (CENTRIA), Departamento de Informática
Universidade Nova de Lisboa, 2825-114 Caparica, Portugal
jja@dmat.uevora.pt; jleite@di.fct.unl.pt
lmp@di.fct.unl.pt; pq@dmat.uevora.pt

Abstract

In this paper we show how planning can be achieved by means of abduction, a form of non-monotonic reasoning, in the *LUPS* language. *LUPS* employs the recently introduced notion of *Dynamic Logic Programming*, whereby the knowledge representation rules, namely those representing actions, can dynamically change, crucial when agents are to be situated in evolving environments. By integrating into a single framework several recent developments in the logic programming and non-monotonic reasoning field of research, this work contributes to a better modeling and understanding of rational agents. At the same time, it enjoys the advantages of a declarative and implementable specification, shortening the usual gap between theory and practice often found in logical based approaches to agents. The system integrating *Dynamic Logic Programming*, *LUPS* and *Abduction*, in order to achieve this form of planning, has been implemented.

1 Introduction and Motivation

In the last few years *agent-based computing* has been one of the most debated concepts. Being a paradigm that virtually invaded every sub-field of computer science, it found in imperative languages the most common adopted vehicle to its implementation, mainly for reasons of efficiency. However, since efficiency is not always a real issue, but clear specification and correctness is, *Logic Programming* and *Non-monotonic Reasoning* have been brought (back) to the spot-light. Add to this significant recent improvements in the efficiency of *Logic Programming* implementations (Niemelä and Simons, 1997; XSB System, 1999). Besides allowing for a unified declarative and procedural semantics, eliminating the traditional high gap between theory and practice, the use of several and quite powerful results in the field of non-monotonic extensions to *Logic Programming (LP)* can represent an important added value to the design of rational agents. For a better understanding a thorough exposition of how *Logic Programming* can contribute to agent-based computing, the reader is referred to Sadri and Toni (1999) and Bozzano et al. (1999). Embedding agent rationality in the LP paradigm affords us with a number of tools and formalisms captured in that paradigm, such as belief revision, inductive learning, argumentation, preferences, etc. (Kowalski and Sadri, 1996; Rochefort et al., 1999)

Traditionally, the work on logic programming was mainly focused on representing static knowledge, i.e. knowledge that does not evolve with time. Some work had been done on updating knowledge bases but limited to factual updates. The problem of updating the knowledge rules, as opposed to updating the models generated by them remained an open issue. Recently, in Leite and Pereira (1997), the authors argued that the *principle of inertia* could be successfully applied to the rules of a knowledge base, instead of to the literals in its models, thereby yielding the desired result. This lead to the introduction of the paradigm of *Dynamic Logic Programming* (Alferes et al., 1998, 2000).

Dynamic Logic Programming, supported by the notion of *Logic Program Updates*, is simple and quite fundamental. Suppose that we are given a set of theories (encoded as generalized logic programs) representing different states of the world. Different states may represent different time periods or different sets of priorities. Consequently, the individual theories contain mutually contradictory as well as overlapping information. The rôle of *Dynamic Logic Programming* is to use the mutual relationships existing between different states to precisely determine the declarative as well as the procedural semantics of the combined theory composed of all individual theories.

Although solving the problem of dynamically evolving logic programs, *Dynamic Logic Programming* does not by itself provide a proper language for its specification. To achieve this goal, and in particular to allow logic programs to describe transitions of knowledge states in addition to the knowledge states themselves, Alferes et al. (1999) introduced the language *LUPS*. *LUPS* allows the association, with each state, of a set of transition rules, providing an interleaving sequence of states and transition rules in an integrated declarative framework. It is worth pointing out that, the most notable difference between *LUPS* and Action Languages (Gelfond and Lifschitz, 1998) is that the latter deal only with updates of propositional knowledge states while *LUPS* updates knowledge states that consist of knowledge rules i.e. the outcome of a *LUPS* update is not a simple set of propositional literals but rather a set of rules. *LUPS* also makes it easier to specify so-called *"static laws"*, to deal with indirect effects of actions and to represent, and reason about, simultaneous actions.

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.

Rational agents must be able to reason about the knowledge they possess and, among other things, plan to achieve their goals. In general, planning consists of the deliberative process by which a set of (partially or totally) ordered actions is generated to achieve one or more goals. Most approaches to agents based on computational logic achieve planning but none of them allowing for dynamically changing rules since they are limited to static intensional knowledge. For a survey and roadmap of computational logic based agents the reader is referred to Sadri and Toni (1999) and Bozzano et al. (1999).

Examples where this dynamic behaviour of rules is essential, where new rules come into play while, at the same time, some rules cease to be valid, can be found in *Legal Reasoning*, namely in what concerns the application of law over time. In countries with legal systems where laws are often changed, jurisprudence makes heavy use of the articles governing the application of law in time. The representation of and reasoning about such articles is not trivial, being most important the part dealing with transient situations, for example when an event occurs after some new law has been approved but hasn't taken effect yet. Different outcomes can be obtained depending on the date of the trial. In such situations an agent acting as a lawyer for example, would have to plan its course of action in quite complex situations due to the changing rules. In Sect. 5 we present an example of such a situation.

Other examples can be found in computer games (often not taken very seriously by the computer science community but extremely challenging and lucrative (Jennings et al., 1998)) where an agent has to deal with partially different rules from level to level, and cross level planning is needed. In general, this framework is quite useful, in all its power, in situations where agents with learning capabilities are moved into slightly different domains (or the domain descriptions change) and it is useful to use the knowledge from the previous domain.

In this paper we show how planning can be achieved by means of abduction, a form of non-monotonic reasoning, in the *LUPS* language, where actions to be performed can be envisaged as abduced update rules. By integrating into a single framework several present developments in the logic programming and non-monotonic reasoning field of research, this work contributes to a better modeling and understanding of rational agents while, at the same time, it enjoys the advantages of a declarative and implementable specification, shortening the usual gap between theory and practice often found in logical based approaches to agents (Sadri and Toni, 1999).

The system integrating *Dynamic Logic Programming*, *LUPS* and *Abduction*, to achieve this form of planning, has been implemented and tested on top of the XSB System (1999). This overall system allows for several forms of reasoning having many applications currently being explored.

The paper is structured thus: After an introductory section to briefly recap *Dynamic Logic Programming* and *LUPS*, the planning problem in *LUPS* is formalized and its solutions characterized. This is followed by the presentation of an implemented solution based on abduction, proven correct according to the formal characterization. Finally we illustrate with an example and conclude.

2 Dynamic Logic Programming and LUPS

In the LUPS framework (Alferes et al., 1999), knowledge evolves from one state to another as a result of sets of (simultaneous) update commands. In order to represent *negative* information in logic programs and in their updates, the framework resorts to more general logic programs, those allowing default negation $not\ A$ not just in the premises of their rules but in their heads as well. In Alferes et al. (1998), such programs are dubbed *generalized logic programs*:

Definition 1 (Generalized Logic Program) *A generalized logic program P in the language $\mathcal{L}$ is a (possibly infinite) set of propositional rules of the form:*

$$L \leftarrow L_1, \ldots, L_n \tag{1}$$

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. ◇

The semantics of generalized logic programs is defined as a generalization of the stable models semantics (Gelfond and Lifschitz, 1988).

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

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

Definition 3 (Stable Models of Generalized Programs) *A model M is a stable model of the generalized program P iff:*

$$M = least(P \cup Default(P,M)) \quad \diamond$$

As proven in Alferes et al. (1998), the class of stable models of generalized logic programs extends the class of stable models of normal programs Gelfond and Lifschitz (1988) in the sense that, for the special case of normal programs both semantics coincide.

In LUPS, knowledge states, each represented by a generalized logic program, evolve due to sets of update commands. By definition, and without loss of generality Alferes et al. (1999), the initial knowledge state KS_0 is empty and, in it, all predicates are assumed false by default. Given a current knowledge state KS_i, its successor state is produced as a result of the occurrence of a non-empty set U of simultaneous update commands. Thus, any knowledge state is solely determined by the sequence of sets of updates commands performed from the initial state onwards. Accordingly, each non-initial state can be denoted by:

$$KS_n = U_1 \otimes \cdots \otimes U_n \qquad n > 0$$

where each U_i is a set of update commands.

Update commands specify assertions or retractions to the current knowledge state (i.e. the one resulting from the last update performed). In LUPS a simple assertion is represented as the command:

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

Its meaning is that if $L_{k+1}, \ldots, L_m$ is true in the current state, then the rule $L \leftarrow L_1, \ldots, L_k$ is added to its successor state, and persists by inertia, until possibly retracted or overridden by some future update command.

In order to represent rules and facts that do not persist by inertia, i.e. that are one-state events, LUPS includes the modified form of assertion:

$$assert\ event\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \quad (3)$$

The retraction of rules is performed with the update command:

$$retract\ [event]\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \quad (4)$$

Its meaning is that, subject to precondition $L_{k+1}, \ldots, L_m$ (verified at the current state) rule $L \leftarrow L_1, \ldots, L_k$ is either retracted from its successor state onwards, or just temporarily retracted in the successor state (if governed by $event$).

Normally assertions represent newly incoming information. Although its effects remain by inertia (until countervened or retracted), the assert command itself does not persist. However, some update commands may desirably persist in the successive consecutive updates. This is especially the case of laws which, subject to some preconditions, are always valid, or of rules describing the effects of an action. In the former case, the update command must be added to all sets of updates, to guarantee that the rule remains indeed valid. In the latter case, the specification of the effects must be added to all sets of updates, to guarantee that, when the action takes place, its effects are enforced.

To specify such persistent update commands, LUPS introduces:

$$always\ [event]\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \quad (5)$$

$$cancel\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \quad (6)$$

The first states that, from the current state onwards, in addition to any newly arriving set of commands, whenever the preconditions are verified, the persistent rule is added too. The second command cancels this persistent update.

Definition 4 (LUPS language) *An update program in LUPS is a finite sequence of updates $U_1 \otimes \cdots \otimes U_n$ where each U_i is a non-empty set of (simultaneous) commands of the forms (2)-(6).* $\diamond$

Any knowledge state KS_q ($0 \leq q \leq n$) resulting from an update program $U_1 \otimes \cdots \otimes U_n$ can be queried via "$holds(L_1, \ldots, L_m)\ at\ q?$". The query is true iff the conjunction of its literals holds at KS_q.

The semantics of LUPS (Alferes et al., 1999) is defined by incrementally translating update programs into sequences of generalized logic programs. The meaning of such sequences of programs is determined by the semantics defined in Alferes et al. (1998). Given a sequence of generalized programs $P_1 \oplus \cdots \oplus P_n$ the semantics has to ensure that the newly added rules (in the later programs of the sequence) are in force, and that previous rules are still valid (by inertia) as far as possible, i.e. they are kept for as long as they do not conflict with newly added ones. Accordingly, given a model M of the last program P_n, start by removing all the rules from previous programs whose head is the complement of some later rule with true body in M (i.e. by removing all rules which conflict with more recent ones). All other persist through by inertia. Then, as for the stable models of a single generalized program, add facts $not\ A$ for all atoms A which have no rule at all with true body in M, and compute the least model. If M is a fixpoint of this construction, M is a stable model of the sequence up to P_n.

Definition 5 (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.* $\diamond$

Definition 6 (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*

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

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

Definition 7 (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:*

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

If some literal or conjunction of literals ϕ holds in all stable models at state s of the Dynamic Program, we write $\bigoplus_s\{P_i : i \in S\} \vDash_{sm} \phi$. $\diamond$

The translation of a LUPS program into a dynamic program is made by induction, starting from the empty program P_0, and for each update U_i, given the already built dynamic program $P_0 \oplus \cdots \oplus P_{i-1}$, determining the resulting program $P_0 \oplus \cdots \oplus P_{i-1} \oplus P_i$. 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 until 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.

Definition 8 (Translation into dynamic programs) *Let $\mathcal{U} = U_1 \otimes \cdots \otimes U_n$ be an update program. The corresponding dynamic program $\Upsilon(\mathcal{U}) = \mathcal{P} = P_0 \oplus \cdots \oplus P_n$ 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 $\mathcal{P}_i = P_0 \oplus \cdots \oplus P_i$ with set of persistent commands PC_i be the translation of $\mathcal{U}_i = U_1 \otimes \cdots \otimes U_i$. The translation of $\mathcal{U}_{i+1} = U_1 \otimes \cdots \otimes U_{i+1}$ is $\mathcal{P}_{i+1} = P_0 \oplus \cdots \oplus P_{i+1}$ with set of persistent commands PC_{i+1}, where PC_{i+1} is:*

$$\begin{aligned} & PC_i \cup \{assert\ [event]\ (R)\ when\ (C) : \\ & \quad always\ [event]\ (R)\ when\ (C) \in U_{i+1}\} \\ & \quad - \{assert\ [event]\ (R)\ when\ (C) : \\ & \quad cancel\ (R)\ when\ (D) \in U_{i+1}, \textstyle\bigoplus_i \mathcal{P}_i \vDash_{sm} D\Big\} \\ & \quad - \{assert\ [event]\ (R)\ when\ (C) : \\ & \quad retract\ (R)\ when\ (D) \in U_{i+1}, \textstyle\bigoplus_i \mathcal{P}_i \vDash_{sm} D\Big\} \end{aligned}$$

$NU_{i+1} = U_{i+1} \cup PC_{i+1}$, and P_{i+1} is:

$$\begin{aligned} & \left\{ \begin{array}{c} R,\ rule(R) : assert\ [event]\ (R)\ when\ (C) \in \\ NU_{i+1}, \bigoplus_i \mathcal{P}_i \vDash_{sm} C \end{array} \right\} \\ \cup & \left\{ \begin{array}{c} not\,rule(R) : retract\ [event]\ (R)\ when\ (C) \in \\ NU_{i+1}, \bigoplus_i \mathcal{P}_i \vDash_{sm} C \end{array} \right\} \\ \cup & \left\{ \begin{array}{c} not\,rule(R) : assert\ event\ (R)\ when\ (C) \in \\ NU_i, \bigoplus_{i-1} \mathcal{P}_{i-1} \vDash_{sm} C \end{array} \right\} \\ \cup & \left\{ \begin{array}{c} rule(R) : retract\ event\ (R)\ when\ (C) \in \\ NU_i, \bigoplus_{i-1} \mathcal{P}_{i-1} \vDash_{sm} C, rule(R) \end{array} \right\} \end{aligned}$$

where R denotes a generalized logic program rule, and C and D a conjunction of literals. In the inductive step, if $i = 0$ the last two lines are omitted. In that case NU_i does not exist. $\diamond$

Definition 9 (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 $\bigoplus_q \Upsilon(\mathcal{U}) \vDash_{sm} L_1, \ldots, L_n$. $\diamond$

3 LUPS and Plans

LUPS, by allowing to declaratively specify both knowledge states and their transitions, can be used as a powerful representation language in planning scenarios. Its variety of update commands can serve to model from the simplest condition-effect action to parallel actions and their indirect effects.

In this section we formalize the planning problem in LUPS, and characterize its solutions. Throughout we consider $\mathcal{U} = U_1 \otimes \cdots \otimes U_n$ to be an update program in the language $\mathcal{L}$. We begin by considering a set of actions from $\mathcal{L}$, whose specification is defined by update commands.

Definition 10 (Action) *Let $\mathcal{L}_\alpha = \{\alpha_1, \ldots, \alpha_m\}$ be a set of atoms from $\mathcal{L}$ where each $\alpha \in \mathcal{L}_\alpha$ represents an action. We call the elements of $\mathcal{L}_\alpha$* actions. *Typically for every* action *there will be one (or more) update commands of the forms (2)-(6), where the action α appears in the 'when' clause.* $\diamond$

Fo example, in

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

we have, intuitively, that α is an action whose preconditions are $L_{k+1}, \ldots, L_m$ and whose effect is an update that, according to its type, can model different kinds of actions, all in one unified framework. Examples of kinds of actions are:

- actions of the form "α causes $F\ if\ F_1, \ldots, F_k$", where $F, F_1, \ldots, F_k$ are fluents (such as in the language $\mathcal{A}$ of Gelfond and Lifschitz (1998)) translates into the update command "$always\ F\ when\ (F_1, \ldots, F_k, \alpha)$".
- actions whose epistemic effect is a rule update of the form "α updates with $L \leftarrow L_1, \ldots, L_k\ if\ L_{k+1}, \ldots, L_m$" translates into the update command "$always\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m, \alpha)$".
- actions that, when performed in parallel, have different outcomes, of the form "α_a or α_b cause $L_1\ if\ L_{k+1}, \ldots, L_m$" and "$\alpha_a$ and α_b in parallel cause $L_2\ if\ L_{k+1}, \ldots, L_m$" translates into the three update commands:

$$always\ L_1\ when\ (L_{k+1}, \ldots, L_m, \alpha_a, not\,\alpha_b)$$
$$always\ L_1\ when\ (L_{k+1}, \ldots, L_m, not\,\alpha_a, \alpha_b)$$
$$always\ L_2\ when\ (L_{k+1}, \ldots, L_m, \alpha_a, \alpha_b)$$

- actions with non-deterministic effects of the form "α causes $(L_1$ or $L_2)\ if\ L_{k+1}, \ldots, L_m$" translates into the update commands (where β_a, β_b are new auxiliary predicates and I is a unique identifier of an occurrence of α):

$$always\ (L_1 \leftarrow \beta_a(I))$$
$$when\ (L_{k+1}, \ldots, L_m, \alpha(I))$$
$$always\ (L_2 \leftarrow \beta_b(I))$$
$$when\ (L_{k+1}, \ldots, L_m, \alpha(I))$$
$$always\ (\beta_a(I) \leftarrow not\,\beta_b(I))$$
$$when\ (L_{k+1}, \ldots, L_m, \alpha(I))$$
$$always\ (\beta_b(I) \leftarrow not\,\beta_a(I))$$
$$when\ (L_{k+1}, \ldots, L_m, \alpha(I))$$

  In this representation of the non-deterministic effects of an action α, we create two auxiliary actions (β_a, β_b) with deterministic effects, and make the effect of α be the non-deterministic choice between actions β_a or β_b.

Next, we formalise *action updates* and *plans*:

Definition 11 (Action Update) *An* action update, U^α, *is a set of update commands of the form:*

$$U^\alpha = \{assert\ event\ \alpha : \alpha \in \mathcal{L}_\alpha\} \quad \diamond$$

Intuitively, each command of the form "$assert\ event\ \alpha$" will represent the performing of action α. Note that performing an action is something that does not persist by inertia. Thus, according to the description of LUPS in Sect. 2 the assertion must be of an event. By asserting $event\ \alpha$, the effect of the action will be enforced if the preconditions are met. Each *action update* represents a set of simultaneous actions. For simplicity, we represent the update commands in an action update just by their corresponding $\alpha's$. Instead of $U^\alpha = \{assert\,event\,\alpha_1; assert\,event\,\alpha_2\}$ we write $U^\alpha = \{\alpha_1; \alpha_2\}$.

Definition 12 (Plan) *A* plan *is a finite sequence of action updates.* $\diamond$

In order to relate the goals to be achieved with the plans, we need to know the effect of executing the plan. This is given by the following function:

Definition 13 (Result) *The* result *of applying a plan $\mathcal{U}^\alpha = U_1^\alpha, \ldots, U_m^\alpha$ to an update program $\mathcal{U} = U_1 \otimes \cdots \otimes U_n$ is given by the update program:*

$$Result(\mathcal{U}^\alpha, \mathcal{U}) = U_1 \otimes \cdots \otimes U_n \otimes U_1^\alpha \otimes \cdots \otimes U_m^\alpha \ \diamond$$

Finally, the planning problem is about finding a plan such that a goal, given in the form of a LUPS query, is achieved as the result of applying the plan to the initial state update program.

Definition 14 (Planning Solution) *Given an update program $\mathcal{U} = U_1 \otimes \cdots \otimes U_n$ and a query (goal) "$holds(L_1, \ldots, L_k)$ at q?", the plan $\mathcal{U}^\alpha = U_{n+1}^\alpha, ..., U_q^\alpha$ is a* planning solution *if the query is true in the result of applying $\mathcal{U}^\alpha$ to $\mathcal{U}$, i.e. such that*

$$\bigoplus_q \Upsilon(Result(\mathcal{U}^\alpha, \mathcal{U})) \vDash_{sm} L_1, \ldots, L_n. \quad \diamond$$

This definition suggests that planning solutions can be seen as abductive update solutions in the LUPS framework. This will be explored in the next section.

4 On the implementation of LUPS and planning

In Alferes et al. (1999), a translation is presented of update programs and queries into single normal logic programs which are written in a meta-language. The translation is purely syntactic, and has been proven correct there: a query holds in an update program iff its translation belongs to all stable models of the update program translation. The latter directly supports a mechanism for implementing update programs: after a pre-processor performs the translations, query answering is reduced to that over normal logic programs by means of a meta-interpreter.

The pre-processor and a meta-interpreter for answering queries have been implemented[1]

The translation uses a meta-language generated by the language of the update programs. For each objective atom A in the language of the update program, and each special propositional symbol $rule_{L \leftarrow Body}$ or $cancel_{L \leftarrow Body}$ (where these symbols are added to the language for each rule $L \leftarrow Body$ in the update program), the meta-language includes the following symbols: $A(s,t)$, $A^u(s,t)$, $\overline{A(s,t)}$, and $\overline{A^u(s,t)}$, where s and t range over the indexes of the update program. Intuitively, these new symbols mean, respectively: A is true at state s considering available all states up to t; A is true due to the update program at state s, considering all states up to t; A is false at state s considering all states up to t; *not* A is true due to the update program at state s, considering all states up to t.

Intuitively, the first indexical argument added to atoms stands from the update state at which the atom was been introduced via a rule. So, according to the transformation in Alferes et al. (1999), in non-persistent asserts, the first argument of atoms in the head of rules is instantiated with the index of the update state where the rule was asserted. In persistent asserts, the argument ranges over the indexes where the rule should be asserted (i.e. all those greater than the state where the corresponding $always$ command is). The second indexical argument stands for the query state. Accordingly, when translating (non-event) asserts, the second argument of atoms in the head of rules ranges over all states greater than the one where the rule was asserted. For event asserts, the second argument is instantiated with the index of the update state where the event was asserted. This is so in order to guarantee that the event is only true when queried about in that state.

Inertia rules are added to allow for access to rules asserted in states before the state the query is posed. Such rules say that one way to prove L at state s with query state t, is by proving L at state $s-1$ with the same query state (unless its complement is proven at state s, thus *blocking* the inertia of L).

Literals in the body of asserted rules are translated such that both arguments are instantiated with the query state. This guarantees that body literals are always evaluated with respect to the query's state. Literals in the *when* clause have both arguments instantiated with the state immediately prior to that in which the rule was asserted. This guarantees that those literals are always evaluated considering that prior state as their query state. The complete formalization of the translation $Tr(\mathcal{U})$ can be found in Alferes et al. (1999).

Planning solutions, as defined in the previous section, are clearly similar to abductive solutions in the LUPS framework: when finding a plan, one is looking for sets of ”assert event´´commands, standing for actions performed, that when added to the sequence derive the top query and are consistent (in the sense that a stable model exists). In this abductive setting, the abducibles are commands of the form $assert\ event\ (\alpha)$ where $\alpha \in \mathcal{L}_\alpha$. Given the above described translation of a sequence of update commands into a single normal logic program with inertia rules, this abduction problem in the LUPS setting, can easily be transformed into an abduction problem in normal logic programs. Simply translate the sequence of update commands into a single program, and define as abducibles those translation predicates arising from translating commands $assert\ event\ (\alpha)$ for every α, where α is an action. It is easily seen that finding abductive solutions for queries over the translated abductive logic program is tantamount to finding planning solutions in the original update program.

Theorem 1 (Planning as abduction) *Let $Tr(\mathcal{U})$ be the logic program obtained from the translation of an update program $\mathcal{U} = U_1 \otimes \dots U_n$, and let $Ab = \{\alpha(i,i) : n < i \leq m+n, \alpha \in \mathcal{L}_\alpha\}$ be the set of abducibles. Given a subset A of Ab, let $\mathcal{U}^\alpha(A) = U_1^\alpha \otimes \dots U_m^\alpha$ be such that $assert\ event\ (\alpha) \in U_i^\alpha$ iff $\alpha(i,i) \in A$.*

Then A is an abductive solution for $Tr(holds\ L_1, \dots L_k\ at\ q)$ in the logic program $Tr(\mathcal{U})$ iff $\mathcal{U}^\alpha(A)$ is a planning solution for “$holds\ L_1, \dots L_k\ at\ q$” in $\mathcal{U}$. ⋄

According to this theorem, to implement a plan generator in LUPS, all that needs doing is to implement the translation from LUPS programs to logic programs, and then to use an interpreter for abduction on top of the translated program. This is the basis of our implementation which, aside from the aforementioned preprocessor for the translation, employs an interpreter for the abduction procedure ABDUAL (Alferes et al., 1999). Note that multiple abductions at one state, ie parallel actions, can be generated.

5 Illustrative Example

In this section we present an example illustrating the ability of this framework to deal with dynamically changing rules. One domain where such dynamic behaviour of rules is essential is *Legal Reasoning*. In this domain new rules come into play while, at the same time, other rules cease to be valid. In countries with legal systems where laws are often changed, jurisprudence makes heavy use of the articles governing the application of law over time. The representation of, and reasoning about, such articles is non trivial, the one most important being the part dealing with transient situations. For example, when an event occurs after some new law has been approved, but hasn't yet taken effect. Different outcomes could be obtained

[1] The system, running under XSB-Prolog, a system with tabling, is available from:
`http://centria.di.fct.unl.pt/˜jja/updates/`
Rather than stable models semantics, the well-founded semantics is used instead.

depending on the date of the trial. In such situations an agent, acting as a lawyer, would have to plan its course of action in complex situations due to the changing rules.

Consider a fictional situation where someone is conscripted if he is draftable and healthy. Moreover a person is draftable when he reaches a specific age. In this situation, if someone is conscripted and not incorporated (for example because he hides), he is cited for a crime and, if tried, goes to jail. Of course one cannot be tried while in hiding. Consider that a person is electable for office if electable previously and not in hiding. Moreover, the person ceases to be electable if ever been to jail. After some time, a new law is approved that renders one not conscripted if a conscientious objector. However, this law will only take effect after 20 days. How could John, electable, healthy, conscientious objector, that became of age 10 days after this new law has been enacted, avoid being incorporated, and remain electable for office in the future? This is an illustrative scenario easily expressible in LUPS. It translates into the update commands[2]:

$\mathbf{U}_1$:

$always\ draftable(X)\ when\ of\_age(X)$
$assert\ (conscripted(X) \leftarrow draftable(X),$
$\quad healthy(X))$
$always\ hiding(X)\ when\ hide(X)$
$always\ not\ hiding(X)\ \ when\ \ unhide(X)$
$assert\ (jail(X) \leftarrow trialed(X), cited(X))$
$always\ incorporated(X)$
$\quad when\ (conscripted(X), not\ hiding(X))$
$always\ cited(X)\ when\ (conscripted(X),$
$\quad not\ incorporated(X))$
$assert\ (trialed(X) \leftarrow cited(X), not\ hiding(X))$
$always\ electable(X)$
$\quad when\ (electable(X), not\ hiding(X))$
$always\ not\ electable(X)\ \ when\ jail(X)$
$assert\ objector(John)$
$assert\ healthy(John)$

$\mathbf{U}_{10}$:

$always\ (not\ conscripted(X) \leftarrow objector(X))$
$\quad when\ current(30)$

$\mathbf{U}_{20}$:

$assert\ of\_age(John)$

where $\mathcal{L}_\alpha = \{hide(X), unhide(X)\}$ is the set of possible actions. If we have the goal of never being incorporated and being electable after 30 , the desirable solution would be to perform the action $hide(John)$ at 20, and perform the action $unhide(John)$ after 30. Note that if John does not hide at 20, he will be cited for a crime, be trialed, sent to jail and thus would never be electable for office again. The goal is:

$$holds\ electable(John)\ at\ 31$$

would return an abductive solution that would correspond to the plan $\mathcal{U}^\alpha = U_{20}^\alpha, U_{30}^\alpha$, where:

$$U_{20}^\alpha = \{assert\ event(hide(John))\}$$
$$U_{30}^\alpha = \{assert\ event(unhide(John))\}$$

representing the desired solution.

[2]Where $current(S)$ is a built in predicate such that it is true iff the current time is S, which could be implemented in LUPS itself.

6 Conclusions and Future Work

The ability to deal with dynamic situations is one of the major features of our proposal, as it allows one to handle a new class of planning problems. In fact, extant planning frameworks do not easily encompass the description of worlds with dynamically changing rules. For instance, neither PDDL (McDermott et al., 1998) nor OCL (Liu and McCluskey, 2000) are capable of describing dynamic situations. Dynamic Logic Programming permits as well the representing of actions in the STRIPS- or ADL-style, utilized in these planners, with pre-conditions, effects, and conditional and logical operators. Additionally, it caters for simultaneous actions and, due to its expressiveness, it can model complex effects of actions. By using an explicit representation of the world, and of the actions available at each state, its history and attending change can itself be queried and reconstructed.

Above all, embedding planning into a logic programming framework with a precise declarative semantics, makes it amenable to integration with other, already developed, monotonic and non-monotonic knowledge representation and reasoning functionalities. Among these:

- Extensive declarative knowledge representation, comprising default and explicit negation
- Semantics (and implementation) for non-stratified knowledge
- Observance and updating of integrity constraints
- Knowledge rules updating, besides that of action rules updating
- Abductive reasoning, over and above the abduction of actions
- Inductive learning of knowledge and action rules
- Belief revision and contradiction removal
- Argumentation for collaboration and competition
- Preference semantics, combinable with updates
- Meta- and object-language combination through meta-interpreters, facilitating language extensions and execution control
- Model-based diagnosis of artifacts, via observations and actions on them
- Declarative debugging of logic programs representing knowledge bases

- Explanation generation
- Distribution with communication
- Agent architectures
- Test and tried implemented logic programming systems, tabled execution

Our ongoing research has promoted and achieved some of these integrative *desiderata*, and currently pursues a number of them. Such integrateable facilities pave the way for the building of complex rational agents employing sophisticated planning amongst themselves.

Acknowledgements

All authors were partially supported by PRAXIS XXI project MENTAL. J. A. Leite was partially supported by PRAXIS XXI scholarship no. BD/13514/97.

References

J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinski and T. C. Przymusinski. *Dynamic Logic Programming*. In A. Cohn, L. Schubert and S. Shapiro (eds.), Procs. of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy, pages 98-109. Morgan Kaufmann, June 1998.

J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinski and T. C. Przymusinski. *Dynamic Updates of Non-Monotonic Knowledge Bases*. To appear in The Journal of Logic Programming, 2000.

J. J. Alferes, L. M. Pereira, H. Przymusinska and T. C. Przymusinski, *LUPS - a language for updating logic programs*. In M. Gelfond, N. Leone and G. Pfeifer (eds.), Procs. of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99), El Paso, Texas USA, pages 162-176, Springer-Verlag, LNAI 1730, 1999.

J. J. Alferes, L. M. Pereira, H. Przymusinska, T. C. Przymusinski and P. Quaresma, *Preliminary exploration on actions as updates*.In M. C. Meo and M. Vialres-Ferro (eds.), Procs. of the 1999 Joint Conference on Declarative Programming (AGP'98), L'Aquila, Italy, pages 259-271, September 1999.

J. J. Alferes, L. M. Pereira and T. Swift, *Well-founded Abduction via Tabled Dual Programs*. In Procs. of the 16th International Conference on Logic Programming, Las Cruces, New Mexico, Nov. 29 - Dec. 4, 1999.

M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi and F. Zini, *Logic Programming and Multi-Agent System: A Synergic Combination for Applications and Semantics*. In K. Apt, V. Marek, M. Truszczynski and D. S. Warren (eds.), The Logic Programming Paradigm - A 25-Year Perspective, pages 5-32, Springer 1999.

M. Gelfond and V. Lifschitz. *The stable model semantics for logic programming*. In R. Kowalski and K. A. Bowen. editors. 5th International Logic Programming Conference, pages 1070-1080. MIT Press, 1988.

M. Gelfond and V. Lifschitz. *Classsical negation in logic programs and disjunctive databases*. New Generation Computing, 9:365-385, 1991.

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

N. Jennings, K. Sycara and M. Wooldridge. *A Roadmap of Agent Research and Development*. In Autonomous Agents and Multi-Agent Systems, 1, 275-306, Kluwer, 1998.

R. Kowalski and F. Sadri. *Towards a unified agent architecture that combines rationality with reactivity*. In D. Pedreschi and C. Zaniolo (eds), Logic in Databases, Intl. Workshop LID'96, pages 137-149, Springer-Verlag, LNAI 1154, 1996.

J. A. Leite and L. M. Pereira. *Generalizing updates: from models to programs*. In J.Dix, L.M. Pereira and T.C.Przymusinski (eds), Selected extended papers from the LPKR'97: ILPS'97 workshop on Logic Programming and Knowledge Representation, pages 224-246, Springer-Verlag, LNAI 1471, 1998.

D. Liu and L. McCluskey. *The Object Centered Language Manual-OCL$_2$*. University of Huddersfield. 2000.

D. McDermott et al. *PDDL - The Planning Domain Definition Language*. Yale University, 1998.

I. Niemelä and P. Simons. *Smodels - an implementation of the stable model and well-founded semantics for normal logic programs*. In Procs. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), pages 420-429, Springer, July 1997.

S. Rochefort, F. Sadri and F. Toni, editors, *Proceedings of the International Workshop on Multi-Agent Systems in Logic Programming*, Las Cruces, New Mexico, USA,1999. Available from `http://www.cs.sfu.ca/conf/MAS99`.

F. Sadri and F. Toni. *Computational Logic and Multiagent Systems: a Roadmap*, 1999. Available from `http://www.compulog.org`.

The XSB Group. *The XSB logic programming system, version 2.0*, 1999. Available from `http://www.cs.sunysb.edu/~sbprolog`.