Using dynamic logic programming to model legal reasoning

Nuno Graça and Paulo Quaresma
ngraca|pq@di.uevora.pt

Departamento de Informática
Universidade de Évora
7000 Évora, Portugal

Abstract. Dynamic logic programming allows the representation and the inference of evolving knowledge.
Legal knowledge reasoning needs the capability to model laws that change over time and to model laws produced by distinct entities with different priorities at different time points.
In this paper we propose the use of dynamic logic programming to model these legal dynamic situations. Some examples are discussed and the implementation of a legal oracle server is described.

1 Introduction

It is well known that knowledge representation needs to take into account the dynamic nature of knowledge. As new information is acquired, new pieces of knowledge need to be dynamically added to or removed from the knowledge bases. Moreover, information may be produced from different sources having different degrees of reliability and, as a consequence, having different priorities.

In [2] dynamic logic programming was proposed as a possible solution to the problem of knowledge base updates and in [3] a new language, *LUPS – Language of UPdateS*, was described and applied to the representation of actions.

On the other hand, legal knowledge reasoning needs the capability to model laws that change over time and to model laws produced by distinct entities with different priorities at different time points. The representation problem of time and law was studied by several researchers, [8, 7, 9], basically using Event Calculus [5] as the base formalism. The proposed approaches had some limitations, namely some lack of generality and the necessity to explicitly define time and inertia rules.

In this paper we propose the use of dynamic logic programming to model these legal dynamic situations. Specifically, we will deal with the representation of evolving rules, i.e., laws where there were changes in their pre-requisites over time, and with the problem of having several sources of laws with possible contradictions.

In order to fully explore these situations, a legal server was implemented in Prolog and it is able to receive logic programming descriptions of laws and

events and to answer queries about what is valid in specific states. Some simple examples are described and discussed.

In section 2 the logic programming framework used to model the legal server is presented. Then, in section 3, the legal server implementation is briefly described and in section 4, examples of hypothetical server interactions are presented. Finally, in section 5 some conclusions and future work are pointed out.

2 Dynamic Logic Programming

Before describing the legal server it is necessary to present the formalism used to implement it. As referred in the previous section, the basic formalism is the dynamic logic programming paradigm and the related language used to represent actions: LUPS[1].

2.1 Dynamic Knowledge Representation

One of the main requirements of the formalism used to represent legal knowledge is to be able to handle evolving knowledge. In fact, legal knowledge may be represented by specific knowledge states but, after each event, such as a law change, knowledge evolves to another state. The formalism should be able to handle these situations, allowing the inference of what properties are valid in each knowledge state.

Dynamic logic programming (DLP) was proposed [2] as a possible solution to this evolution requirement. In fact, DLP defines how a knowledge base can be updated by another knowledge base, obtaining a new knowledge base.

Specifically, given an original knowledge base KB, and an updating knowledge base KB', it is possible to obtain 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, in [2] there is a complete semantic characterisation 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.

2.2 LUPS – Language of UPdateS

In DLP knowledge evolves from one knowledge state to another as a result of knowledge updates. Given the current knowledge state KS, its successor knowledge state $KS' = KS[KB]$ is obtained as a result of the occurrence of a non-empty set of simultaneous updates, represented by the updating knowledge base KB.

[1] This section is based on a previous work describing DLP and LUPS [4]

However, dynamic knowledge updates, do not provide any language for specifying changes of knowledge states. Accordingly, in [3] it was described 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.

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 action *request help*. 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*.

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

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 cancelled. In order to specify such *persistent update commands* (which are called *update laws*) there is 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, there is 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.

Knowledge can be queried at any knowledge state KS_q with:

$$\textbf{holds } B_1, \ldots, B_k, not\, C_1, \ldots, not\, C_m \textbf{ at state } q\textbf{?} \tag{6}$$

and is true if and only if the conjunction of its literals holds at the state KS_q.

The language LUPS has a *declarative and procedural semantics* [2] so that the update commands not only have a definite declarative meaning but also can be readily implemented. 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 of LUPS into XSB-Prolog is available at `http://centria.di.fct.unl.pt/~jja/updates/lups.p`.

3 Legal Server

The legal server is a XSB-Prolog process able to deal with connections via a user defined TCP/IP port.

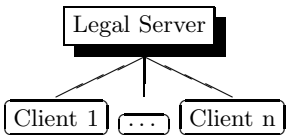

Fig. 1. Architecture

There are two possible commands that clients may send to the legal server:

- Update
- Query

The clients send sequences of update rules and/or queries about properties and they receive the answers to their queries.

These commands will be described in detail in the next two sub-sections:

3.1 Update command

The update rules have the following syntax:

$$update(KB, Agent, LUPSList) \tag{7}$$

where KB means knowledge base and allowing the definition and the update of distinct knowledge bases; $Agent$ defines the name of the agent performing the action; and $LUPSList$ is a Prolog list of LUPS commands.

As an example, we may have:

$$update(law, crimeLaw, [(always\ (illegal \leftarrow abortion))]) \tag{8}$$

It is also possible to use predicate *update* with only two arguments, KB and $LUPSList$, meaning we want to update or to query the KB as a top priority independent agent: "God's view of the KB". This special top agent has also the possibility to define priorities among the different agents:

$$higherPriority(Agent1, Agent2). \tag{9}$$

The priority relations will have important consequences in the processing of the agents updates.

After receiving an update predicate, the legal server will create correspondent LUPS updates, evolving the KB received as a parameter to a new state. The translation between the client requests (rule 7) and the LUPS updates follows these rules:

1. For every request received from agent $Agent1$
   (a) For every LUPS command in LUPSList having a rule with head H create a correspondent command, substituting H by H_{Agent1}
   (b) For each distinct predicate H (head of some LUPS command) introduced by the request, create the new rule:

$$always\ (H \leftarrow H_{Agent1}, not - H_{A1}, \ldots, not - H_{An}) \tag{10}$$

   where $A1$, ..., An are more priority agents than $Agent1$ and not and $-$ stand for the default and explicit negation, respectively.

The general idea of the first part of the translation is to index every conclusion (head of the rules) to the correspondent agent. The second part of the translation relates the agents beliefs with its more priority agents: an agent belief is only accepted if it is not contradictory with a belief of another more priority agent.

As a simple example, rule 8 would produce the following LUPS rules:

$$always\ (illegal\_crimeLaw \leftarrow abortion) \tag{11}$$

$$always\ (illegal \leftarrow illegal\_crimeLaw) \tag{12}$$

If, for instance, Constitutional Law is an agent of a higher priority, then we would have the following rule (instead of rule 12):

$$always\ (illegal \leftarrow illegal\_crimeLaw, not - illegal\_constitutional\_law) \tag{13}$$

3.2 Query command

The query commands have the following syntax:

$$query(KB, Agent, \textbf{holds}\ B_1, \ldots, B_k, not\, C_1, \ldots, not\, C_m\ \textbf{at state}\ q\textbf{?}) \tag{14}$$

where KB means knowledge base and allowing the query of distinct knowledge bases; $Agent$ defines the name of the agent performing the action.

As in the previous section, it is possible to omit the second parameter and to query the KB from the special top agent. As an example, we may have:

$$query(law, \textbf{holds}\ illegal\ \textbf{at state}\ now\textbf{?}) \tag{15}$$

4 Examples

In the next two sub-sections two kind of examples will be presented:

- Evolution of laws
- Laws produced by distinct agents/entities

4.1 Evolution of laws

Suppose there is a law stating that in order to have a specific degree it is necessary to be enrolled in that degree and to verify certain conditions:

$$update(law1, [assert\ (degree(X) \leftarrow enrolled(X), cond1(X))]). \tag{16}$$

Now, suppose John enrolls himself in that degree:

$$update(law1, [assert\ (enrolled(john))]). \tag{17}$$

But, after that, there is an update in the law and the needed conditions change:

$$\begin{aligned} update(law1, [&retract\ (degree(X) \leftarrow enrolled(X), cond1(X)), \\ &assert\ (degree(X) \leftarrow enrolled(X), cond2(X))]). \end{aligned} \tag{18}$$

After this change John verifies $cond1$ and Mary enrolls herself in the degree:

$$update(law1, [assert\ (cond1(john)), assert\ (enrolled(mary))]). \tag{19}$$

Finally, Mary satisfies $cond2$:

$$update(law1, [assert\ (cond2(mary))]). \tag{20}$$

If we query the KB about who has a degree and in which states:

$$query(law1, \textbf{holds}\ degree(X)\ \textbf{at state}\ S\textbf{?}]). \tag{21}$$

We will obtain the expected behavior:

$$X = mary, S = 5.$$

Only Mary has a degree and only at state 5 (we are assuming initial state to be equal to 0).

Note that, if we change update 17 to:

$$update(law1, [assert\ (enrolled(john)), assert\ (cond1(john))]). \tag{22}$$

We would get:

$$\begin{aligned} X = john, S = 2; \\ X = mary, S = 5. \end{aligned}$$

This answer captures the fact that rule defining how to obtain a degree has changed. However, law updates are usually not retroactive: if John had a degree at state 2, then he shouldn't have lost his degree afterwards!

This requisite leads to a new schema for the representation of law changes[2]:

$$update(law2, [assert\ (degree(X) \leftarrow en(X), cond1(X))]). \tag{23}$$

$$update(law2, [assert\ (en(john))]). \tag{24}$$

$$update(law2, [retract\ (degree(X) \leftarrow en(X), cond1(X))\ when\ (not\ en(X)),$$
$$assert\ (degree(X) \leftarrow en(X), cond2(X))\ when\ (not\ en(X))]). \tag{25}$$

$$update(law2, [assert\ (cond1(john)), assert\ (en(mary))]). \tag{26}$$

$$update(law2, [assert\ (cond2(mary))]). \tag{27}$$

Rule 25 captures the notion that the law changes only to those that are not yet enrolled in the degree.

As intended, we'll have:

$$X = john, S = 4;$$
$$X = john, S = 5;$$
$$X = mary, S = 5.$$

4.2 Laws from distinct entities

In this example suppose there are three sources of knowledge defining what is needed to obtain a master's degree (conditions necessary and sufficient):

- S1: To write a thesis;
- S2: To obtain a certain amount of course credits;
- S3: To write a thesis and to obtain a certain amount of course credits (100 in this example).

We'll have the following correspondent commands:

$$update(law3, s1, [assert\ (master(X) \leftarrow thesis(X)),$$
$$assert\ (-master(X) \leftarrow -thesis(X))]). \tag{28}$$

$$update(law3, s2, [assert\ (master(X) \leftarrow credits(X, N), N > 100),$$
$$assert\ (-master(X) \leftarrow credits(X, N), N <= 100)]). \tag{29}$$

$$update(law3, s3, [assert\ (master(X) \leftarrow credits(X, N), N > 100,$$
$$thesis(X)),$$
$$assert\ (-master(X) \leftarrow (credits(X, N), N <= 100;$$
$$-thesis(X))]). \tag{30}$$

Suppose agents are related by the following relation $s1 < s2 < s3$:

$$higherPriority(s3, s2). \tag{31}$$

$$higherPriority(s2, s1). \tag{32}$$

[2] Due to formatting problems we will use *en* instead of the predicate *enrolled*

With these constraints, the following update facts:

$$update(law3, [assert\ (thesis(john))]). \tag{33}$$
$$update(law3, [assert\ (credits(mary, 120))]). \tag{34}$$
$$update(law3, [assert\ (credits(john, 110))]). \tag{35}$$

And the following queries:

$$query(law3, \textbf{holds}\ master_{s1}(X)\ \textbf{at state}\ S\textbf{?}]). \tag{36}$$
$$query(law3, \textbf{holds}\ master_{s2}(X)\ \textbf{at state}\ S\textbf{?}]). \tag{37}$$
$$query(law3, \textbf{holds}\ master_{s3}(X)\ \textbf{at state}\ S\textbf{?}]). \tag{38}$$
$$query(law3, \textbf{holds}\ master(X)\ \textbf{at state}\ S\textbf{?}]). \tag{39}$$

We'll have:

$$master_{s1}(john), S = 4;$$
$$master_{s1}(john), S = 5;$$
$$master_{s1}(john), S = 6;$$
$$master_{s2}(mary), S = 5;$$
$$master_{s2}(mary), S = 6;$$
$$master_{s3}(john), S = 6;$$
$$master(john), S = 6;$$

These results, as intended, state that accordingly with $S1$ John has a master's degree after terminating his thesis (state 4); accordingly with $S2$ this is accomplished only at state 6 for John (and 5 for Mary); and accordingly with $S3$ only John has a master's degree and only at state 6. As $S3$ is the most priority agent, its "opinion" is the accepted by the legal server.

5 Conclusions and Future Work

The use of dynamic logic programming and its associated language, LUPS, to model some characteristics of legal reasoning was proposed. Specifically, the problem of laws that change over time and the problem of laws produced by different sources with different reliabilities/priorities was dealt with.

Dynamic logic programming revealed to be a powerful methodology to handle these kind of requisites and the obtained solutions were quite satisfactory and easy to model.

However, the proposed solution to integrate time and priority constraints is not easily generalized to additional constraints/dimensions. As future work, we intend to use an extension of DLP, MDLP – Multidimensional Dynamic Logic Programming, from J. Leite et al. [6] to allow a more generic approach and the integration of several dimensions, such as time and priorities. Another possible

direction is to use the new language, EVOLP, proposed by Alferes et. al [1], to simplify LUPS.

A legal server able to receive law updates and requests was also implemented. These server is able to handle requests from different agents and about different knowledge bases. As future work, we intend to change the communication protocol to a standard agent communication protocol, such as FIPA ACL [10], able to deal with actions like $requests$ and $informs$. We also plan to develop a web interface able to handle user actions and to interact with the legal server via the FIPA ACL protocols.

References

1. J. Alferes, A. Brogi, J. Leite, and L. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *JELIA'02 – Proceedings of the 8th European Conference on Logics and Artificial Intelligence*, pages 50–61. Springer-Verlag LNCS 2424, 2002.
2. J. J. Alferes, J. Leite, L. M. Pereira, H. Przymusinska, and T. Przymuzinski. Dynamic logic programming. In *Proc. of KR'98*, 1998.
3. 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. Vilares-Ferro, editors, *Procs. of the 1999 Joint Conference on Declarative Programming (AGP'99)*, pages 259–271, L'Aquila, Italy, September 1999.
4. J. J. Alferes, L. M. Pereira, H. Przymusinska, T. C. Przymusinski, and P. Quaresma. An exercise with dynamic logic programming. In L. Garcia and M. C. Meo, editors, *Procs. of the 2000 APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP'2000)*, La Habana, Cuba, December 2000.
5. R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 24:67–95, 1986.
6. J. Leite, J. Alferes, L. Pereira, H. Przymusinska, and T. Przymusinski. A language for multi-dimensional updates. In J. Dix, J. Leite, and K. Satoh, editors, *Computational logic in multiagent systems: Proceedings of the 3rd international workshop CLIMA'02*, number 93, pages 19–34. Roskilde University, Denmark, 2002.
7. E. Mackaay, D. Poulin, J. Fremont, C. Deniger, and P. Bratley. The logic of time in law and legal expert systems. *RatioJuris*, 3:254–271, 1990.
8. Rafael Marin and Giovanni Sartor. Time and norms: a formalisation in the event calculus. In *ICAIL'99 – Internatinal Conference on Artificial Intelligence and Law*. ACM, 1999.
9. A. Provetti. The law of contracts in the event calculus. In *GULP'92 – 9th Italian Conference on Logic Programming*, 1992.
10. fipa.org www. *FIPA ACL - Agent Communication Language*. FIPA - Foundation for Intelligent Physical Agents, 2001.