Towards modelling intelligent agents with LUPS*

Mauricio Javier Osorio Galindo[1] Juan Carlos Acosta Guadarrama[1],

Universidad de las Américas, CENTIA
Sta. Catarina Mártir, Cholula, Puebla
72820 México
sp081829@mail.udlap.mx

Abstract. We present two main issues towards an application of dynamic knowledge representation programmed in LUPS (a language for updates). This is a study case to model intelligent agents in logic and we use a calendar example in order to explain their features, such as non-monotonic reasoning in fixing appointments, making proposals and belief revision. We argue that a calendar is a good and practical example of dynamic knowledge representation to be programmed in LUPS. Other several challenges are suggested as well, so as to achieve a more powerful and useful system.

1 Introduction and Motivation

There is a new logic language called LUPS, a language for updates

Nowadays we have powerful calendar systems to manage our dates and to do tasks. You can classify your appointments, work in groups, check them everywhere in the world, and even they can remind your important events via e-mail, or mobile devices at local time. Namely Yahoo Calendar^{TM}, Netscape Calendar^{TM}, or even desktop computer local calendars. However, it seams that current calendar systems are not paying attention on what to do when you want (by accident or intentionally) to fix two dates at the same time! It seems that there are no calendar systems with an autonomous agent that help you fix your dates automatically, without contravening previous ones. Nor even they negotiate group dates in order to gather people for an event in a convenient time for everybody.

On the other hand, we have LUPS, a language for dynamic knowledge updates (see [8] for details). In this context, we have not found any agent system programmed in this language yet, but some pieces of coded rules that show how LUPS can be appropriate for the agent paradigm —see also [7,9] for details.

Then, we present an application of dynamic knowledge representation programmed in LUPS (a language for updates). This is a study case towards modelling intelligent agents in logic and we use a calendar example in order to explain

* This work is supported by Consejo Nacional de Ciencia y Tecnología: CONACYT project 35804-A and Universidad de las Américas, Puebla: Fondo de Apoyo al Posgrado

their non-monotonic features. We argue that a calendar is a good and practical example of dynamic knowledge representation to be programmed in LUPS. In addition, we argue several challenges, in order to achieve a more powerful and useful system.

We have split this paper in several sections starting with a background where we found our research. Second, we describe our calendar agent as an implementation in LUPS and how to deal with belief revision. Third, we propose some future challenges for the reader. Then we finish giving our concluding remarks.

2 Background

Following we start describing some basic generalities about intelligent agents, LUPS and belief revisio, where we found our research.

2.1 Intelligent Agents

There is special interest on developing 'intelligent' agents recently, in order to build new applications such as Space exploration, intelligent communication through Internet, and control systems development, among others [4]. However, we are "far from having clear understanding about the basic principles and techniques needed for their design" [4].

This is a complex problem since intelligent agents are different from traditional software systems in several aspects, according to [4]

- Knowledge amount about the world and its own capabilities could be large.
- Intelligent agents should be able to expand this knowledge.
- They must be aware of its own actions and of the other agents.
- Ability to extract implicit knowledge (reason) stored in its memory.
- Finally they should use their knowledge to plan and execute their actions rationally.

All this implies a solid theoretical basis on knowledge representation for agents design: That is to say, logic programming and non-monotonic reasoning theory, as well as theory of actions and change, in order to model agent domain. [4]

Finally, let us talk about a general approach of what our intelligent agents are in order to locate our study case. They can fit one of the following architectures suggested by [1,3]:

- logic (deliberative) agents
- Beliefs-desires-Intentions agents —BDI agents.

2.2 Belief Revision

Whenever someone designs a program, he/she must consider all possible cases of the problem that the program could run into. However, many of them cannot be seen at the design stage. Especially, when the program is modelling something which conditions change in time. Traditional software engineering, including O-O development, solves this problem at the maintenance stage. However, it lacks of an automatic process to realize it. Moreover, human aid is necessary. Even if we had all those resources, the agent paradigm is more dynamic than that and may not depend on a software enterprise organization. Therefore, if we are writing a calendar agent, we need incorporate autonomy.

Such autonomy means the agent ability to meet its goals on behalf of the user, by operating on their own without the need of for human guidance — 'taking the initiative' [2]. Thus, an agent must have the ability to rearrange its goals as well, by revising or rejecting (making *belief revision*) not to get inconsistencies, namely, "the problem of reconciling beliefs with conflicting facts by an appropriate revision of beliefs" [10].

2.3 LUPS —A Language for Dynamic Knowledge Representation

When we say logic programming, we still mean static knowledge[1]. However, the agent paradigm is an environment typically more open and dynamic [9]. Thus, we need "ways of representing and integrating knowledge from different sources which may evolve in time." [9] In [7,9,8], a group of people are working on overcoming this static limitation and have introduced Dynamic Logic Programming (DLP) and Multi Dynamic Logic Programming (MDLP). Those techniques are combined in an agent architecture called MINERVA (see [7,9]) where they use a language to maintain an initial knowledge base. They named it LUPS: "a language for dynamic updates" [9,8]. This language is designed for specifying declaratively changes to logic programs, where each one in the sequence contains knowledge about a given state [9]. They provide a way to use both stable models (where we base our research) and well-founded semantics. On the other hand, when we talk about non-monotonic reasoning, we mean adding information that invalidates previous conclusions [5]. As you may know, both kinds of semantics are non-monotonic. This means the need to implement a mechanism to avoid inconsistencies, which we will call belief revision.

This powerful declarative language provides two sets of rules called in [8] *persistent and non-persistent update commands*. Persistent commands are those rules that have to do with the agent functionality. They remain idle until there are certain conditions that make one or more of them trigger. The other kind of rules has to do with events. They describe those conditions that make knowledge evolve in different time points. The scope of the first set of rules is at every state. The other kind has 'local effect' at each state.

Finally, since the purpose of LUPS is maintaining a knowledge base, we think it is a good tool also for belief revision.

[1] Although we have assert in Prolog, it implies side effects.

3 A Calendar in LUPS

The meaning of modelling a calendar in LUPS is updating knowledge. We think it is a practical example because it has to do with non-monotonic reasoning and agents. Following we describe its main features.

In our calendar, we will have two top modules with a specific propose. Mainly the constituent parts are a graphic user interface (GUI) module and an inference machine. Frequently, software designers use an object-oriented programming language for the former, and prototype in any logic language for the latter. However, owing to its evolution in time and autonomy, we believe logic programming should be a language not only for prototyping, but also for a final product. This paper will focus on the inference machine, nevertheless.

We use LUPS, XSB and DLV for the inference machine because we can model knowledge update in LUPS and use XSB-Prolog to perform arithmetic and other lower level static-knowledge functions. The role of Java will become a meta-language for LUPS and for XSB in order to realize many useful tasks at this level, namely belief revision.

3.1 Basic Knowledge in XSB-Prolog

This module is made for keeping lower level libraries of a calendar. i.e. In this file you can find Prolog rules such as arithmetic operations; outside information like the agent compromises and beliefs; appointment types; other agents awareness; and last, time management rules, such as work days, office hours and holidays, mostly.

Some of the most interesting rules are those that deal with time —For instance, office hours and holidays. However, we will not examine them in depth, because it is out of the scope of this paper.

3.2 calendar.P

This is the main program for this study case, the highest abstraction level. It is coded in LUPS, in order to allow dynamic knowledge representation. It is responsible of managing beliefs, desires, intentions and actions about the calendar and the outside world. There are three main persistent rules to manage the calendar: fix, appointment and proposal, which are desires, actions and intentions, respectively. The specification in LUPS for appointment is in the first persistent rule we present following:

$$\begin{aligned} always\ appointment(X)\ when\ fix(X)\ \wedge \\ not\ holiday(X)\ \wedge \\ officeHours(X)\ \wedge \\ not\ family(X) \end{aligned} \tag{1}$$

which means: *Whenever we find a fix desire and it is not a holiday and it is at office hours and it does not come from my family, make it appointment.* i.e. It will be triggered as soon as a *fix* event[2] appears and fits the conditions. Then we will get an appointment fact that would look like

$$appointment(X) \tag{2}$$

Fixing Appointments The first step in the process on fixing an appointment on the calendar is *having the desire*. Let us look at the first week of December 2002. Now there comes a 5-priority event on 4 of December (Wednesday) at 12:00 PM, asked by me. As there are *intentions*[3] (compromises) to fix dates that come from me, then it becomes a *desire* to fix that appointment —**Update 1**

$$assert\ event\ fix(dec4, 12:00, 5, me) \tag{3}$$

After a new update command, the persistent appointment rule is triggered and we get an appointment fact

$$appointment(dec4, 12:00, 5, me) \tag{4}$$

Next, let us *desire* another 9-priority appointment[4] on Thursday at 10:00 AM, requested by me (**Update 2**):

$$assert\ event\ fix(dec5, 10:00, 9, me) \tag{5}$$

So far, there is no problem with either appointment, since there were no previous dates at that time, and none of them were outside office hours. Thus, they become appointments (*actions*), as explained above.

Making Proposals Now suppose I try to arrange an appointment at 3:00 PM with priority 5, but such hour is outside office hours! —**Update 3**

$$assert\ event\ fix(dec4, 15:00, 5, me) \tag{6}$$

Next, the calendar agent is going to change that *fix* into another time proposal. It will be within office hours because of a persistent rule similar to the one of the appointment (1).Then, we get a proposal fact from that *fix desire* at another time:

$$proposal(dec4, 16:00, 5, me) \tag{7}$$

This proposed time depends on other appointments and current date and time —It *prefers* proposing a date close to the original one. Proposals are agent's capabilities to *negotiate* dates with the user or other agents.

[2] An event is a non-persistent rule in LUPS.
[3] Recall intentions are one kind of the agent priorities.
[4] Remember high priority is 10, while 1 is low.

Now suppose another agent desires to make a common appointment at 12:00 PM on Wednesday, **update 4**, and I try to arrange another one at the same time with the same priority:

$$assert\ event\ fix(dec4, 12:00, 5, me) \tag{8}$$

As both dates have the same priority, the calendar agent is going to take the last *fix* event and propose another time. Again, it depends on other appointments and current date and time. It would look like the following proposal fact:

$$proposal(dec4, 13:00, 5, me) \tag{9}$$

Finally, let us suppose there comes a new event for my boss (priority 9) on Wednesday at 12:00 PM —**update 5**. It is at the same time as the appointment at state 1. Normally, my boss's dates have higher priority than other activities. Thus, the first appointment will be moved to a proposal and my boss's takes its place.

Searching for an ideal free space We propose three persistent rules to achieve this ideal space and a current time fact: Firstly, one to find a free space. Secondly, a rule to find a space better than the one found. Lastly, another rule to search for the best free space. We defined an ideal free space is the best one found, close to the original fixing date. Users may change this preference.

By now, we have a 'manual' fact to manage current time. You can find it in the XSB system (libs.P file) like this:

$$today(26/05/2002, 10:00) \tag{10}$$

meaning: "Today is the 1st of February, 2002. It is 10:00 hr". However, we designed it to be given automatically from the highest level of the system (in Java), by using the system's clock.

3.3 Making Basic Revision

There is an example in LUPS on cooperative agents in [11]. In this paper, the authors introduce a belief revision technique when contradictions arise owing to coming information from another agent. They realize it by means of a persistence rule. However, the example just eliminates previous beliefs when they are *incompatible* with the new ones [11]. We try to go beyond not only eliminating the rules, but also modifying them alternatively.

First, let us think of a non-common, but possible situation. Several circumstances are not conceived in current schedules perhaps because we have incomplete information:

Consider the following simple-minded common-sense rules to model non-long future appointments.

– An appointment which date is two or more years far from today is too far.

– Today is 26 of May, 2002.
– An appointment too far from today is not possible.

We believe that making an appointment after two years from now is impossible. Now let us suppose I fix an appointment to take a qualification course on 2 of January 2003. We can fix appointments in this year and the following, and the program will work fine (we obtain stable models). Now let us suppose my boss tells me that he will promote me after my course finishes, provided, I hand the certificate in. Nevertheless, it is one year later! Namely, the course lasts one year, but I start it the next year. So I must remember the event by January 2004, which is out of the bounds of the original calendar, a contradiction at the restriction —note that this restriction does not have to be known by the user nor the boss. Then we must find a way to change the agent beliefs so that contradictory information can coexist.

We can represent this knowledge by the following general program.

$$\begin{aligned} tooFar(Date, Today) \leftarrow\ & Difference\ is\ Date - Today, \\ & Difference \geq 2. \qquad (11) \\ \leftarrow\ & appointment(Date), \\ & tooFar(Date, Today), \\ & today(Today). \qquad (12) \end{aligned}$$

Next let us add the fact today (26/5/2002) as well as the new appointment, 2 of January 2003, by the following set of rules:[5]

$$today(26/05/2002). \qquad (13)$$
$$appointment(2/01/2003). \qquad (14)$$

Then the stable semantics produces the desired results by these two last facts: *Today is 26 May 2002* and *I have an appointment on 2 of January 2003.*

On the other hand, after introducing a 3/January/2004 appointment, our program infers that today is 26 of January 2002; I have an appointment on 2/Jan/2003; I have another appointment on 3 of January 2004; and finally, this appointment is too far from today:

$$today(26/05/2002). \qquad (15)$$
$$appointment(2/01/2003). \qquad (16)$$
$$appointment(3/01/2004). \qquad (17)$$
$$tooFar(3/01/2004, 26/05/2002). \qquad (18)$$

[5] In order to ease reading, we use a common date format dd/mm/yyyy. However, we actually use a yyyymmdd format in our code to perform arithmetic operations with dates represented by integers.

and the instantiated restriction rule becomes

$$\begin{aligned} \leftarrow\ & appointment(3/01/2004, 26/05/2002), \\ & tooFar(3/01/2004, 26/05/2002), \\ & today(26/05/2002). \end{aligned} \tag{19}$$

meaning: *I have an appointment on 3 of January 2004 and it is too far from today.* This is a contradiction to our belief that we could not have an appointment too far from today. Thus, we have no stable models.

Our proposal to solve this problem consists in reducing the program by a meta-program in LUPS until we find the inconsistency. We have a set of rules (see[12,13,14] for details) that preserve equivalence under stable semantics. There is also another system to revise contradictory logic programs. Its name is REVISE [15] and it is based on the well-founded semantics with explicit negation, WFSX. However, it is out of the bounds of our work.

Before applying our rules, we must modify the program restrictions so that it is easy to trace at the final stage. Then, we introduce a positive and negative atom, at each side of the implication. The atom name should correspond to the restriction meaning and its variables should be the same used in the rule. We will see later that this technique will tell us where the contradiction is.

Let us name such atom as Far Appointment Restriction (farAppRes in short) and introduce it in our restriction, as follows:

$$\begin{aligned} farAppRes(Date, Today) \leftarrow\ & appointment(Date), \\ & tooFar(Date, Today), \\ & today(Today), \\ & \neg farAppRes(Date, Today). \end{aligned} \tag{20}$$

The next step is applying a reduction rule to the ground program. We choose **Dsuc** from a set of transformation rules in [13,14]. The rule says:

Definition 1 (Dsuc). *If P contains the clause $a \leftarrow \top$ and there is also a clause $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ such that $a \in \mathcal{B}^+$, then replace it by $\mathcal{A} \leftarrow (\mathcal{B}^+ \setminus \{a\}), \neg\mathcal{B}^-$.*

If we apply it to our ground program, we 'remove' the positive atoms from the restriction body, which are facts of the program, and get the following reduced set of atoms

$$today(26/05/2002). \tag{21}$$

$$appointment(2/01/2003). \tag{22}$$

$$appointment(3/01/2004). \tag{23}$$

$$tooFar(3/01/2004, 26/05/2002). \tag{24}$$

$$\begin{aligned} farAppRes(3/01/2004, 26/05/2002) \leftarrow & \\ \neg farAppRes(3/01/2004, 26/05/2002). & \end{aligned} \tag{25}$$

Obviously, the last conclusion is a contradiction. However, it gives enough information so as to find the source, because it says that there is an inconsistency at the Far Appointment Restriction with the date 3/01/2004 and today —26/05/2002. It is matched with the last appointment, which date is 3 of January 2004.

The next step is asking the user what to do with that appointment and with today. We could reject either. Why rejecting today? Although it is supposed to come from the system time, there is no warranty it is always right (perhaps the computer battery is exhausted or somebody changed the time by accident). The other alternative is making it an exception. In either case, a meta-program should modify the program. Let us suppose the user wants it to be an exception. Hence, as well as adding the exception fact $exception(3/01/2004)$.
a meta-program changes the rule on too far appointments as follows:

$$\begin{aligned} tooFar(Date, Today) \leftarrow\ & Difference\ is\ Date - Today, \\ & Difference \geq 2. \\ & \neg exception(Date) \end{aligned} \tag{26}$$

Now our knowledge base is correctly updated. We have more transformation rules in [13,14] that can be used in situations more complex than this.

4 Some Future Challenges

According to Nwana [2], you may not say a software system is an agent unless it is autonomous, cooperative and capable of *learning*. There is a chance and a challenge to make this calendar agent learn. For example, it could really determine the user's preferences, which is something we hardly ever take care of. We are not aiming at basic preferences, such as customizing the system interface. We are talking about determining something more diffuse: *Learning the way a user likes or dislikes dates.* Consider for instance, meetings on Thursday afternoon, free evenings on Friday and no meeting on Monday morning.

To reach this proposal we suggest using Inductive Logic Programming (ILP). ILP consists of giving a set of examples of background knowledge and a 'rule sketch' that we want to get. Then, ILP builds a hypothesis which explains beliefs of our agent in terms of such knowledge [6].

Talking about what an agent is, the last third component of the knowledge base is cooperation. We present a primitive base towards a social construction of knowledge.

5 Conclusions

We presented a practical example prototype on how easy, simple and powerful an agent can be if it is modelled in LUPS. It is more evidence that this declarative

language is adequate when representing dynamic knowledge updates. We also give some ideas on what possible extensions can be incorporated in LUPS, such as belief revision mechanism, restrictions and preferences rules. Finally, we try to encourage companies so that they incorporate an agent feature to their calendar systems.

References

1. Gerhard Weiss. *Multiagent Systems.* MIT Press, 1999, Cambridge, Mass. 02142 USA, 1999.
2. H. S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(2):205–244, 1995.
3. Rodney A. Brooks. Intelligence without representation. Number 47 in Artificial Intelligence, pages 139–159. 1991.
4. Chitta Baral and Michael Gelfond. Reasoning agents in dynamic domains. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.
5. Gerhard Brewka. *Nonmonotonic Reasoning: Logical Foundations of Commonsense*, volume 12 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1991.
6. S. Muggleton. Inductive logic programming: issues, results and the LLL challenge. *Artificial Intelligence*, 114(1–2):283–296, December 1999.
7. J. J. Alferes J. A. Leite and L. M. Pereira. Minerva - combining societal agents knowledge. International workshop on agent theories, architectures, and languages (atal'01), Dept. de Informatica, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, August 2001.
8. José Júlio Alferes, Luís Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. LUPS - a language for updating logic programs. In *Logic Programming and Non-monotonic Reasoning*, pages 162–176, 1999.
9. J. Leite, J. Alferes, and L. Pereira. Minerva - a dynamic logic programming agent architecture, 2001.
10. José Júlio Alferes, Luís Moniz Pereira, and Teodor C. Przymusinski. Belief revision in non-monotonic reasoning and logic programming. *Fundamenta Informaticae*, 1(1):1–6, 1996.
11. José Júlio Alferes, Luís Moniz Pereira, and et al. An exercise with dynamic knowledge representation.
12. Jürgen Dix, Mauricio Osorio, and Claudia Zepeda. A General Theory of Confluent Rewriting Systems for Logic Programming and its Applications. *Annals of Pure and Applied Logic*, 108/1-3:153–188, 2001.
13. Mauricio Osorio, J. C. Nieves, and Chris Giannella. Useful transformation in answer set programming. In *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 146–152. AAAI Press, Stanford, USA, 2001.
14. Mauricio Osorio, Juan Antonio Navarro, and Jose Arrazola. Equivalence in answer set programming. In *Proceedings of LOPSTR*, pages 18–28, 2001.
15. Carlos Viegas Damásio, Luis Moniz Pereira, and Michael Schroeder. Revise: Logic programming and diagnosis. In *LNAI 1265.* Springer-Verlag, 1997.