Agent Deliberation via Forward and Backward chaining in Linear Logic

(Extended Abstract)

Luke Trodd, James Harland, John Thangarajah
School of CS & IT, RMIT University, Melbourne, Australia
{luke.trodd, james.harland, john.thangarajah}@rmit.edu.au

Categories and Subject Descriptors

I.2.11 [**Computing Methodologies**]: Distributed Artificial Intelligence—*intelligent agents, languages and structures*

General Terms

Languages, Design

Keywords

BDI Deliberation cycle, linear logic, forward chaining, Lygon

Agent solutions to programming problems are often based on the *Belief-Desire-Intention (BDI)* paradigm [12]. *Beliefs* represent what the agent believes to be the current state of the world. *Desires* specify the proactive behaviour of the agent, in that the agent works to make these true. Often desires can be mutually exclusive or contradictory, requiring the agent to select from among them, and so BDI implementations often use *goals*, which can be thought of as desires with some restrictions on them (such as requiring goals to be consistent, feasible and not yet achieved). There can be several types of goals, including *achievement goals*, whcih are dropped once they have been achieved, and *maintenance goals*, which are continually monitored, even when currenlty true. *Intentions* are plans of action that the agent has committed to to achieve its current goals. Often there are many ways to achieve a set of goals that the agent is working on, implying the need for a mechanism to choose between them.

Implementations of BDI systems are usually based around an *observe-think-act* cycle, in which an agent will observe the current environment, which may have changed since the last observation, determine which goals it should be pursuing and what plans should be used to achieve them, and choose a particular action to perform. Note that while the number of actions performed in the *act* phase is not specified, it is intended to be relatively small, so that the agent will be able to detect changes in the environment (which is only done in the *observe* phase) and respond to them within an appropriate amount of time. Hence a fundamental feature of BDI systems is the manner in which they provide both proactive (or goal-directed) and reactive behaviour.

In this paper, we consider how we may adapt existing logical inference techniques to implement a BDI architecture. Using logic as a basis for the architecture will mean that we can develop methods for formal analysis of agent systems via logical inference, as well as being able to exploit existing automated reasoning technologies to develop applications. In particular, we will investigate the use of *linear logic* [4] for such systems. Linear logic has the potential to offer many advantages in the agent context over other logics due to its resource-oriented nature. Linear logic is able to specify actions cleanly and intuitively [9], can effectively express resource oriented problems and has a native notion of concurrency appropriate for agent architectures. Linear logic has also been recently applied to agent negotiation [11], and adaptive narratives [1]. This suggests that there is significant potential for the development of BDI agents based on linear logic, for which there are existing logic programming languages such as *Lygon* [5].

Our BDI agent architecture is based on Lygon technology. This means that we proceed in a bottom-up manner, commencing with what can be readily implemented in Lygon, identifying where extensions are needed, adding these to Lygon and eventually developing a BDI deliberation cycle. This has been implemented and applied to various problems (including the gold mining problem used in the CLIMA agent programming contest[1]). Our focus is hence not so much on the design of (yet another) agent programming language, nor on the formal analysis of such a language, but on the similarities and differences between what is provided in linear logic programming languages such as Lygon and what is required by a BDI agent architecture. Once this is done, we intend to use our implementation experience to develop both appropriate language features and a formal analysis of their properties.

We have developed and implemented a forward-chaining inference mechanism to complement Lygon's existing backward-chaining, in order to provide a natural mechanism for reactive behaviour. This combination turns out to be a simple but effective technique for proactive checking of maintenance goals [3] in a generic manner. We have implemented these techniques in Lygon and have developed and tested a number of applications.

The requirement for an agent to use a combination of both proactive and reactive behaviour corresponds in automated reasoning to a combination of both *backward-chaining* and *forward-chaining* inference [6, 2, 8]. Backward-chaining involves reasoning backwards from a goal towards known truths, whereas forward-chaining involves using what is known to be true to infer new results. Harland and Winikoff [7] have proposed a BDI system based on linear logic, in which the proactive behaviour of the agent is provided by backward-chaining methods and the reactive behaviour of the agent is provided by forward-chaining methods. In terms of the BDI cycle, this means that the *think* phase would be implemented by backward-chaining techniques and the *act* and *observe* phases by forward-chaining ones. Backward-chaining methods have been the basis of logic programming languages based on linear logic, such

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), 4-8 June 2012, Valencia, Spain.

[1]http://centria.di.fct.unl.pt/~clima

as Lygon [5]. Forward-chaining methods have also been used[6], and techniques to combine both methods into one system have also been studied [2, 8]. However, there has been comparatively little work on applying such methods in linear logic to agent systems. In addition, the work of Harland and Winikoff was purely a design; no precise execution method was given and no implementation was developed.

Our first step is to extend Lygon with abductive capabilities, so that the result of a computation is not just an answer, but is a set of actions to be performed (possibly empty, corresponding to a "yes") in order to make the goal true. The *abducibles*, i.e. the results of the abduction process, are constrained here to be actions.

We write actions and plans as rules in Lygon, and use backward-chaining together with abduction to determine a set of actions that will achieve the goal. A subtlety here that may not be immediately apparent is that *there is a need to specify sequences of goals*, i.e. goals and actions that must be performed in a particular order. For example, a robot vacuum cleaner that needs to move to a particular room before cleaning it will require the move action to be done before the cleaning one, as a post-condition of the move action is a pre-condition of the next action. Moreover, it is common for plans to require that a particular set of actions be performed in a specific order, sometimes intermixed with subgoals. This means that to implement a BDI-style system, we need to be able to specify a sequential order in which actions, plans and goals are to be executed or achieved. This is nothing more or less than a reflection of the fact that the actions required to achieve a particular goal are usually constrained to work in a particular sequence.

Unfortunately there is no (simple) way to use existing Lygon connectives to do this. One possibility is to use *, which does something related, but as discussed by Winikoff [13], this does not work, as * can only distribute existing resources. Given a goal $G_1 * G_2$ any new information generated in the solution of G_1 (and in particular the postconditions of an executed action) cannot be passed onto G_2. Using $G_1 \# G_2$ does allow this, but does not restrict the computation of G_1 to be performed before G_2 (and in fact allows both goals to be pursued concurrently). Another possibility is to use the "continuation-passing style" mechanism proposed by Winikoff, which adds a continuation argument to each rule, and splits each rule into a number of rules. However, this is unwieldy, and the number of rules can potentially grow very large and hence difficult to maintain, especially due to the recursive nesting of rules that is required.

Hence we introduce a new connective $>>$ (read 'then'), in order to succinctly state what is required. Intuitively, an agent wanting to sequentially achieve goals G_1 and G_2 will first perform actions to achieve G_1, and, having noted the updates to the world that these actions have made, make plans for achieving G_2 from that updated world. Hence a program and goal $P, G_1 >> G_2$ results in the program and goal P_1, G_2 where P_1 is the result of actions A_1 which convert P to P_1 and for which $P_1 \vdash G_1$.

The $>>$ mechanism makes it straightforward to specify agent behaviours. It also seems intuitively simple, although it in some ways combines both forward- and backward-chaining. Consider a program P_0 and the goal $G_1 >> G_2 >> \ldots >> G_n$. This asks the agent system to find, if possible, actions $A_1, A_2 \ldots A_n$ such that $P_{i-1} \xmapsto{A_i} P_i$ (i.e. the actions A_i will convert P_{i-1} to P_i) and $P_i \vdash G_i$. If at any point, such an A_i cannot be found, backtracking occurs to see if some alternatives can be found for earlier goals (meaning that there can be many such A_i for each G_i). In other words, solving for each goal G_i results in a backward-chaining computation to find A_i, and the results of each action are propagated forwards to the next goal.

The mechanisms that have been discussed in this paper have been implemented in an extended version of Lygon. Our extensions to Lygon have added around 1100 lines (of sparsely arranged and duly commented code) to the original Lygon interpreter of 720 lines.

We believe that our experiments show that this approach has been an effective way to develop BDI agents. One of the more pleasing artefacts of the implemented agent extensions was the relatively straightforward means by which proactive constraints could be implemented. Proactive constraints provide an extremely powerful mechanism for arbitrarily restricting agent behaviours in an intuitive way. The constraint mechanism effectively implements many of the ideals proposed by Duff at al. [3] for proactive maintenance goals in an agent context.

For future work, the precise relationship between the $>>$ operator and the increasingly sophisticated proof-theoretic combinations of backward- and forward-chaining [2, 8] requires further investigation. The definition of the $>>$ operator itself is in some sense orthogonal to the issues of backward- and forward-chaining, but the way in which it is used in agent programs seems to imply that further analysis will be rewarding. Given that $G_1 >> G_2$ specifies a particular order in which G_1 and G_2 must be used, non-commutative versions of linear logic may be an appropriate starting point [10]. The key technical issue is finding an appropriate interaction between the non-commutative connective $>>$ and the other commutative connectives, as distinct from having only commutative or non-commutative properties alone.

Another aspect of future work is to incorporate maintenance goals into the planning mechanism. This would mean that the generation of actions would also include the possibility to generate actions designed to restore maintenance goals after a predicated violation. Hence rather than just avoid situations where violations occur, the agent can take actions to recover from violations.

1. REFERENCES

[1] A.-G. Bosser, M. Cavazza, and R. Champagnat. Linear logic for non-linear storytelling. In *ECAI*, Lisbon, August 2010.

[2] K. Chaudhuri, F. Pfenning, and G. Price. A logical characterization of forward and backward chaining in the inverse method. *Journal of Automated Reasoning*, 40:133–177, March 2008.

[3] S. Duff, J. Harland, and J. Thangarajah. On proactivity and maintenance goals. In *AAMAS*, pages 1033–1040, Hakodate, 2006.

[4] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[5] J. Harland, D. Pym, and M. Winikoff. Programming in lygon: An overview. In *AMAST*, pages 391–405, Munich, July 1996.

[6] J. Harland, D. Pym, and M. Winikoff. Forward and backward chaining in linear logic. In *CADE-17 Workshop on Proof-Search in Type-Theoretic Systems*, Pittsburgh, June 2000.

[7] J. Harland and M. Winikoff. Agents via mixed-mode computation in linear logic. *Annals of Mathematics and Artificial Intelligence*, 42(1):167–196, 2004.

[8] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logic. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[9] M. Masseron, C. Tollu, and J. Vauzeilles. Generating plans in linear logic i: Actions as proofs. *Theoretical Computer Science*, 113(2):349–370, 1993.

[10] J. Polakow. Linear logic programming with an ordered context. In *Principles and Practice of Declarative Programming*, Montreal, September 2000.

[11] D. Porello and U. Endriss. Modelling multilateral negotiation in linear logic. In *ECAI*, page 939, Lisbon, August 2010.

[12] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *KR*, pages 439–449, 1992.

[13] M. Winikoff. *Logic Programming With Linear Logic*. PhD Thesis, University of Melbourne, 1997.