Language Design Issues for Agents based on Linear Logic (Extended Abstract)

James Harland and Michael Winikoff

School of Computer Science and Information Technology
RMIT University
GPO Box 2476V
Melbourne 3001, Australia
Email: {jah,winikoff}@cs.rmit.edu.au
WWW: www.cs.rmit.edu.au/∼{jah,winikoff}

Abstract. Agent systems based on the *Belief*, *Desire* and *Intention* model of Rao and Georgeff have been used for a number of successful applications. However, it is often difficult to learn how to apply such systems, due to the complexity of both the semantics of the system and the computational model. In addition, there is a gap between the semantics and the concepts that are presented to the programmer. One way to bridge this gap is to re-cast the foundations of such systems into a logic programming framework. In particular, the integration of backward- and forward-chaining techniques for linear logic provides a natural starting point for this investigation. In this paper we discuss the language design issues for such a system, and particularly the way in which the potential choices for rule evaluation in a forward-chaining manner is crucial to the behaviour of the system.

1 Introduction

An increasingly popular programming paradigm is that of *agent-oriented programming*. This paradigm, often described as a natural successor to object-oriented programming [18], is highly suited for applications which are embedded in complex dynamic environments, and is based on human concepts, such as beliefs, goals and plans. This allows a natural specification of sophisticated software systems in terms that are similar to human understanding, thus permitting programmers to concentrate on the critical properties of the application rather than getting absorbed in the intricate detail of a complicated environment. Agent technology has been used in areas for applications such as air traffic control, automated manufacturing, and maintenance tasks on the space shuttle [19].

There are many possible conceptions of agent-oriented programming. However a common theme [8, 35] is that agent systems should include properties such as

- *pro-activeness:* the agent has an agenda to pursue and will persist in trying to achieve its aims
- *reactiveness:* the agent will notice and respond to changes in the environment
- *autonomy:* the agent will act without necessarily being instructed to take particular steps

– *situated:* the agent both influences and is influenced by the environment around it

Other possible attributes of agent systems include being *social*, (i.e. teaming up with other agents in order to achieve common goals), *learning* (i.e. taking note of previous actions and adjusting future actions accordingly), and *rationality*, (i.e. working to achieve its aims, and not working against them).

One of the most popular and successful technical realisations of such conceptions is the framework of Rao and Georgeff [28], in which the notions of *Belief*, *Desire* and *Intention* are central, and hence are often referred to as *BDI* agents. Roughly speaking, beliefs represent the agent's current knowledge about the world, including information about the current state of the environment inferred from perception devices (such as cameras or microphones) and messages from other agents, as well as internal information. Desires represent a state which the agent is trying to achieve, such as the safe landing of all planes currently circling the airport, or the achievement of a certain financial return on the stock market. Intentions are the chosen means to achieve the agent's desires, and are generally implemented as plans (which may be thought of as procedures which come with pre-conditions (to determine when a plan is applicable) and intended outcomes (to state what is achieved upon the successful completion of the plan)).

Rao and Georgeff gave both a logical system incorporating these concepts [28] (a version of temporal logic extended to include the appropriate notions of belief, desire and intention) and an architecture for the execution of programs following the BDI paradigm [29].

Whilst BDI-based systems have been successfully applied in a number of areas, there remain some foundational and design issues to be solved. One such issue is the "gap" between the BDI theory, which is based on branching-time temporal logic (in which there is only one past but many possible futures) and the BDI architectures on which systems such as dMARS[6], JACK[1] and JAM[17] are based. The BDI theory has an elegant description of the relationship between beliefs, desires and intentions via possible worlds, but the architectures tends to deal in beliefs, events and plans, and it is not altogether obvious how these relate to the given semantics (although the latter can clearly be seen as an inspiration and specification of the ideal behaviour). The closure of this gap is not helped by the development of purely model-theoretic approaches to the semantics of such systems, with a corresponding lack of emphasis on the proof-theoretical aspects (although [30] is a notable step in this direction).

A second difficulty, particularly when it comes to making agent-oriented programming accessible to a wide audience, is the complexity of the BDI semantics. Whilst there is no magical way to simplify an inherently complex system, it has generally been the case that successful applications of this technology has come from either the developers of the agent system or from groups who have relied on a significant amount of input from a BDI expert (who are generally in very short supply). Hence in order for BDI agents to become significantly more widespread, a simpler means of understanding the system is required [33].

One way to address this is to re-cast the basic agent model into a logic programming framework. As noted by Kowalski and Sadri [21] the main technical question here is to determine how to incorporate actions and reactive behaviour into a logic programming

environment (which has traditionally been very strong on backward-chaining methods, and thus is closely related to traditional planning techniques).

Our approach is based on the use of *linear logic* [10], a logic designed with bounded resources in mind. In particular, linear logic is not only a conservative extension of classical logic (so that classical reasoning, where appropriate, may be used), but also has been shown to be a natural way to model concurrency, database updates and state-based transitions [11, 12, 15]. In particular, it has been shown that actions, such as those required by the classic blocks world scenario, can be modelled simply and naturally in linear logic [22, 23]. Given also the existence of a number of logic programming languages based on linear logic (such as LO [4], Lolli [15], Forum [24], LLP [16] and Lygon [12]), it seems natural to explore the use of linear logic as a basis for BDI-style agent systems.

In [13] it was shown how a notion of *forward-chaining* could be introduced into the standard sequent calculus for linear logic in order to provide such behaviour. In [14] the use of this framework as a basis for agent systems was discussed. In this paper we develop this direction further by examining the language design issues for such a system.

This paper is organised as follows: in §2 we give an overview of BDI systems, linear logic and linear logic programming, and in §3 we discuss the design issues for an agent programming system based on linear logic. In §4 we focus on scheduling issues and in §5 we discuss our conclusions and possibilities for further work.

2 Background

2.1 BDI Agents

The BDI model (Belief Desire Intention) of Rao and Georgeff [28, 29] is a popular model for intelligent agents which has its basis in philosophy [5] and offers a *logical theory* which defines the mental attitudes of Belief, Desire, and Intention using a modal logic; a *system architecture*; a *number of implementations of this architecture* (e.g. PRS, JAM, dMars, JACK); and *applications* demonstrating the viability of the model. The central concepts in the BDI model are [9, page 144]:

Beliefs: The agent's information about the environment;

Desires: Objectives to be accomplished, possibly with each objective's associated priority/payoff;

Intentions: The currently chosen course of action; and

Plans: Means of achieving certain future world states. Intuitively, plans are an abstract specification of both the means for achieving certain desires and the options available to the agent. Each plan has (i) a body describing the primitive actions or subgoals that have to be achieved for plan execution to be successful; (ii) an invocation condition which specifies the triggering event, and (iii) a context condition which specifies the situation in which the plan is applicable.

The BDI model has developed over about 15 years and there are certainly strong relationships between the theoretical work and implemented systems. The paper [29]

describes an abstract architecture which is instantiated in systems such as dMars and JACK and shows how that is related to the BDI logic. However, the concepts that have been found to be useful for development within these systems do not necessarily match the concepts most developed in the theoretical work. Neither are they necessarily exactly the concepts which have arisen within particular implemented systems such as JACK. An additional complication is confusion and small differences between similar concepts, such as Desires and Goals, which receive differing emphasis in different work at different times. Some key differences between the philosophy, theory, and implementation viewpoints of BDI are shown below.

Philosophy:	Belief	Desire	Intention
Theory:	Belief	Goal	Intention
Implementation:	Relational DB (or arbitrary object)	Event	Running Plan

2.2 Linear Logic

Linear logic [10] is sometimes described as *resource-sensitive*, in that the notion of resource is a natural one in this logic. The traditional techniques of logic treat two copies of a formula as being equivalent to one copy (as mathematical truth is not dependent on the number of times a property is stated), and hence formulae can be arbitrarily copied. However, this does not fit well with some application areas, in which there is a finite amount of resources, such as money, computer memory, floor space or execution time. Resource-sensitive logics such as linear logic do not allow arbitrary copying; in linear logic, by default, each formula has to be used exactly once. This property means that linear logic is a natural way to study state changes, and so provides a more direct way to model resource-bounded applications than the traditional techniques. In particular, linear logic has been applied to concurrency problems [10, 11], database updates [15] and planning problems [22, 23].

Linear logic contains two forms of conjunction: one which is "cumulative", i.e. for which $p \otimes p \not\equiv p$, and one which is not, i.e. $p \,\&\, p \equiv p$. Roughly speaking, the former is what allows linear logic to deal with resource issues, whilst the latter allows for these issues to be overlooked (or, more precisely, for an "internal" choice to be made between the resources used), as, by default, each formula in linear logic represents a resource which must be used exactly once.

Consider the following menu from a restaurant: fruit or seafood (in season), main course, all the chips you can eat, and tea or coffee.

Note that the first choice, between fruit and seafood, is a classical disjunction; we know that one or the other of these will be served, but we cannot predict which one, which may be thought of as an "external" choice, in that someone else makes the decision. On the other hand, the choice between tea and coffee is an "internal" choice — the customer is free to choose which one shall be served. Note the internal choice is a conjunction; in order to satisfy this, the restauranteur has to be able to supply *both* tea and coffee, and not just one of them. The chips course clearly involves a potentially infinite amount of resources, in that there is no limit on the amount of chips that the customer may order. We represent this situation by prefixing such formulae with a !. Note also

that the meal consists of four components, and hence we connect the components with $\otimes$. Hence we have the following representation of the menu:

$$(\texttt{fruit} \oplus \texttt{seafood}) \otimes \texttt{main} \otimes \,!\,\texttt{chips} \otimes (\texttt{tea} \,\&\, \texttt{coffee})$$

where we write $\oplus$ for the classical disjunction. Note that the use of ! makes it possible to recover classical reasoning, as formulae beginning with ! with ? in a succedent behaves classically, in that such formulae may used arbitrarily many times, including 0, rather than exactly once. Hence `chips` corresponds to *exactly* one serving of chips, ! `chips` corresponds to an arbitrary number of servings (including 0). In this way we may think of a formula $!F$ in linear logic as representing an unbounded resource, i.e. one that may be used as many times as we like. Thus classical logic may be seen as a particular fragment of linear logic, in that there is a class of linear formulae which precisely matches classical formulae.

Linear logic also contains a negation, which behaves in a manner reminiscent of classical negation. The negation of a formula F is written as $F^{\perp}$. As there are two conjunctions, there are two corresponding disjunctions, as well as a dual to ! denoted as ?. The following laws, reminiscent of the de Morgan laws, all hold:

$$(F_1 \otimes F_2)^{\perp} \equiv (F_1)^{\perp} \parr (F_2)^{\perp} \qquad (F_1 \parr F_2)^{\perp} \equiv (F_1)^{\perp} \otimes (F_2)^{\perp}$$

$$(F_1 \oplus F_2)^{\perp} \equiv (F_1)^{\perp} \,\&\, (F_2)^{\perp} \qquad (F_1 \,\&\, F_2)^{\perp} \equiv (F_1)^{\perp} \oplus (F_2)^{\perp}$$

Each of these four connectives also has a unit, which, for $\otimes$ and $\&$ are written as **1** and $\top$, and which may be thought of as generalisations of the boolean value true, and for $\parr$ and $\oplus$ are written as $\perp$ and **0**, and which may be thought of as generalisations of the boolean value false.

There is far more to linear logic than can be discussed in this paper; for a more complete introduction see the papers [2, 10, 11, 31], among others. A part of the sequent calculus for linear logic is given in Appendix A.

2.3 Logic Programming in Linear Logic

The execution models on which these languages are based have generally been based on *backward-chaining*, i.e. given a number of statements (or *formulae*) which make up the program, the user then requests the system to determine whether or not a given formula (the *goal*) follows from the information in the program. The way that this is achieved is generally by working backwards, i.e. establishing premises which, if true, would establish the truth of the goal. This process is repeated until either an unconditional statement of truth is found (an *axiom*), in which case the original goal succeeds, or no such premises can be found, in which case the original goal fails. Hence backward-chaining consists of starting with a given conclusion and working our way back (hopefully) to the axioms.

Whilst this paradigm appears to be intuitive and natural for various applications, such as querying a database, or solving a particular set of constraints, it does not allow programs to react to an environment, as they must wait for a specific goal to be given. In applications such as a stock market monitor, it is generally desirable to have the

program "watch" the environment, which may include large amounts of data, until a given set of circumstances is observed, such as a sharp fall in the price of a blue-chip stock. Then it would be expected to take the appropriate action, such as buying such stock, and selling it again once the price has recovered. Thus the key element is for the program to evaluate the current environment until certain trigger conditions are met.

Such *reactive* behaviour is more akin to *forward-chaining*, which is a method of reasoning which begins with the axioms, and applies any known rules to generate new results. In the context of linear logic, which is well-known to be a useful way to model and reason about state changes, a forward-chaining approach seems particularly suitable for reactive systems, as this provides a simple and natural way to express conditions which are dependent on the dynamics of the environment.

The techniques for backward-chaining (both classically and for linear logic) are well-known [15, 20, 25, 27]; the integration of forward-chaining techniques into such a system was investigated in [13]. In particular, this allows a combination of *don't know* nondeterminism (common in logic programming) via backward-chaining with *don't care* nondeterminism via forward-chaining.

Such an integrated system is thus able to both follow a planned sequence of instructions (backward-chaining) and react to the environment and make appropriate changes (forward-chaining). In particular, this may be thought of as an increased emphasis on *process-oriented* computation (such as the safe running of a power plant or operating system) rather than *result-oriented* computation (such as calculating a pay cheque or checking whether a given credit card number is valid). As argued in [3, 32], amongst others, the process view of computation is one that is becoming increasing important, and in which safety considerations are vital.

Kowalski and Sadri [21] developed extensions to the traditional logic programming paradigm to incorporate agent features. A key difference in our approach is the use of linear logic, which provides a better representation of dynamic information (such as actions and environmental changes) than classical logic.

The key technical point is to determine the appropriate inference rules for the forward-chaining part of the system. At first, this may seem rather trivial, in that we simply take the well-known rule of *modus ponens*, and use it to determine that B follows from A and $A \supset B$. However, there are some subtleties to this, particularly for resource-sensitive logics.

In the classical case, the formulae $A \wedge A \supset B$ and $A \wedge B$ are equivalent, which means that a forward-chaining system based on this rule has several strong properties. One of these is *monotonicity*, in that the set of conclusions reached can only increase. This property is exploited not only in the well-known T_P semantics for logic programs [7], but also by deductive database systems such as Aditi, for which the monotonicity property is the underlying reason behind the differential optimisation.

The corresponding analysis in linear logic is not as straightforward, though, as the above equivalence does not hold. In particular, given p and $p \multimap q$, it is possible to derive q, but p is "consumed" in this process. Hence the use of modus ponens in linear logic is more like a committed choice, in that once the inference rule is applied, p is no longer available, but q is, and so our analysis needs to proceed in a more subtle way than in the classical case.

Our general procedure is to integrate elements of forward-chaining into the standard sequent calculus for linear logic (which is given in an Appendix). The sequent calculus is well-known as an inference system which permits an appropriate analysis of backward-chaining, and whilst there are systems similarly suitable for forward-chaining (such as natural deduction and Hilbert-type systems), it is not clear how backward-chaining can be introduced into such systems.

It should be noted that backward-chaining techniques are generally applied to a program and a goal: given a program $\mathcal{P}$ and a goal $\mathcal{G}$, we proceed to search for a proof of the sequent $\mathcal{P} \vdash \mathcal{G}$ via some appropriate search strategy. Such proofs are generally *cut-free*, in that the cut rule is not used in the search, as it may introduce formulae with no known relationship to the original sequent, and thus result in a hopelessly infeasible search.

By contrast, forward-chaining techniques are applied to a program, and produce another program. Hence, the natural approach is to define a relation $\rightsquigarrow$ between programs, so that $\mathcal{P} \rightsquigarrow \mathcal{P}'$ denotes that $\mathcal{P}'$ can be derived from $\mathcal{P}$ (via forward-chaining techniques).

We then need to determine not only the rules for $\rightsquigarrow$, but also how these inference rules interact with the standard rules of the linear sequent calculus (and hence with backward-chaining methods). Our approach is to model the interaction between the two types of inference by a particular type of occurrence of the cut rule, known as *direct* or *analytic* cuts. In particular, given a forward-chaining inference $\mathcal{P} \rightsquigarrow \mathcal{P}'$ and a backward-chaining one $\mathcal{P}'' \vdash \mathcal{G}$, then these two inferences can synchronise when $\mathcal{P}' = \mathcal{P}''$. Thus we have that from $\mathcal{P} \rightsquigarrow \mathcal{P}'$ and $\mathcal{P}' \vdash \mathcal{G}$ we can deduce $\mathcal{P} \vdash \mathcal{G}$, which is just an instance of the cut rule. The key point to note is that not only are $\mathcal{P}$ and $\mathcal{G}$ known at the outset, but also that we expect the inference rules for a conclusion such as $\mathcal{P} \rightsquigarrow \mathcal{P}'$ to be such that given $\mathcal{P}$, we can readily derive $\mathcal{P}'$ from an appropriate number of applications of the rule.

In this sense the rules for $\rightsquigarrow$ are reminiscent of *conditional rewriting rules*, or of Plotkin's *Structured Operational Semantics* [26], in that given a unary rule for $\rightsquigarrow$

$$\frac{\mathcal{P} \rightsquigarrow \mathcal{P}''}{\mathcal{P} \rightsquigarrow \mathcal{P}'} R$$

it will generally be the case that $\mathcal{P}$ is known but $\mathcal{P}'$ is not, and so we will use the premise (and any appropriate sub-proofs) to evaluate $\mathcal{P}$ to $\mathcal{P}''$, and then using the rule R we will then determine that $\mathcal{P} \rightsquigarrow \mathcal{P}'$.

Hence we proceed by inserting cuts into the inference rules of the linear sequent calculus, and study the properties of the resulting rules. We have some preliminary results along these lines, which derive an appropriate set of inference rules for $\rightsquigarrow$ and examines their integration into the linear sequent calculus. These results were published in [13] and include permutation properties (which are fundamental to the issue of proof normalizations and hence proof-search strategies) and cut-elimination results. Some of the rules for mixed-mode inference in linear logic can be found in Appendix B. For the full set of rules (and additional details relating to quantification) see [13].

3 Designing Agent Programming Systems

In [14] it was discussed how an agent system requires (at least) the properties below:

1. A means of decomposing a given goal G into subgoals
2. A means of determining a set of possible plans to achieve the subgoals
3. A means of monitoring environmental changes and accordingly evaluating the most appropriate plan to execute

We have discussed how backward and forward chaining can be integrated within the one inference system. This section discusses how these two aspects of the unified inference system can naturally model agents. In particular backward chaining is used to model the *proactive* aspect of the agent (which involves finding ways to achieve goals), and forward chaining is used to model the *reactive* aspect of the agent (which involves integrating events/percepts). We also show below how actions can be performed via forward chaining.

An agent can be represented by the sequent

$$\mathcal{E}, \mathcal{A}, \mathcal{B}, !\mathcal{P} \vdash \mathcal{G}$$

where $\mathcal{B}$ is the beliefs of the agent (which are linear since they change), $\mathcal{P}$ is the program clauses (i.e. goal-plan decompositions), and $\mathcal{G}$ is the agent's goals[1]. We also allow events ($\mathcal{E}$) and actions ($\mathcal{A}$) to appear.

The following proof fragment illustrates the interaction of backwards chaining over goals (proactive) with actions. Events are also handled using forward chaining ($\rightsquigarrow$). Adding an action (A) to the context triggers forward chaining using a directed cut (labelled $Cut \vdash$ below):

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots \qquad \vdots}{\mathcal{P}, A \rightsquigarrow \mathcal{P}' \quad \mathcal{P}' \vdash G', \ldots, G_n}\, Cut \vdash}{\mathcal{P}, A \vdash G', \ldots, G_n}}{\mathcal{P} \vdash A \multimap G', \ldots, G_n}\, \multimap\text{-R}}{\vdots}}{\mathcal{P} \vdash G_1, \ldots, G_n}$$

where $\mathcal{P}$ includes both beliefs (linear) and program clauses (non-linear), which may include action descriptions.

One issue concerns the choice of rules: the inference rules do not constrain which rule is to be applied at any given point. However, in order for agents to respond to events in a timely fashion, and in order for actions to be executed when they are scheduled we would like to constrain the selection of rules. In particular, whenever there are events or actions in the left side (antecedent) of the sequent (i.e. $\Delta, \text{do(A)} \vdash \Gamma$ or $\Delta, \text{event(E)} \vdash \Gamma$) it seems reasonable to expect that forward-chaining will be performed in preference to backward-chaining. Hence as planning is implemented via backward-chaining, actions and events tend to take precedence over planning computations.

[1] Actually since $\mathcal{G}$ includes executing plans it is closer to the intention structure

To make things concrete, consider an "embedded" variety of the blocks world, in which blocks can be moved around or added to the system without the agent doing so (and hence the environment can alter the position and number of blocks). There are red and blue blocks, and the agent's goal is to finish with a pile of blocks that has a red block on top and a blue block under it.

We use the following predicates in the rules below:

blue(X): block X is blue;
red(X): block X is red;
ontable(X): block X is sitting on the table;
on(X,Y): block X is on block Y;
clear(X): block X is clear, i.e. no block is on top of it;
empty: the robot arm is empty;
holds(X): block X is in the robot arm;
move: move a red block onto a blue one;
put(X,blue): put block X onto a blue block;

This leads to rules such as those below. We assume that *red* and *blue* are classical (i.e. blocks do not change colour). We also assume that $\otimes$ binds tighter than $\multimap$, that $\multimap$ binds tighter than $\forall$, and that $\forall$ binds tighter than !. Thus $!\forall xp \otimes q \multimap r \otimes s$ is parsed as $!(\forall x((p \otimes q) \multimap (r \otimes s)))$.

$$! \forall x, y \;\; \textit{red(x)} \otimes \textit{blue(y)} \otimes \textit{clear(x)} \otimes \textit{on(x,y)} \;\multimap\; \textit{redtop}$$
$$! \forall x, y \;\; \textit{move}^{\perp} \;\multimap\; \textit{red(x)} \otimes \textit{blue(y)} \otimes \textit{clear(x)}$$
$$! \forall x, y \;\; \textit{move} \otimes \textit{on(x,y)} \otimes \textit{red(x)} \otimes \textit{red(y)} \otimes \textit{clear(x)} \otimes \textit{empty} \multimap \textit{clear(y)} \otimes \textit{hold(x)} \otimes \textit{put(x,blue)}$$
$$! \forall x, y \;\; \textit{move} \otimes \textit{ontable(x)} \otimes \textit{red(x)} \otimes \textit{clear(x)} \otimes \textit{empty} \;\multimap\; \textit{hold(x)} \otimes \textit{put(x,blue)}$$
$$! \forall x, y \;\; \textit{put(x,blue)} \otimes \textit{hold(x)} \otimes \textit{clear(y)} \otimes \textit{blue(y)} \;\multimap\; \textit{on(x,y)} \otimes \textit{clear(x)} \otimes \textit{empty}$$

Given these rules (which we denote R) and an initial state of the blocks, say F, we then wish to determine whether $R, F \rightsquigarrow R, F'$ such that *redtop* is true in F', so that our goal is *redtop* $\otimes \top$.

Clearly if for some blocks t and v we have $\{on(t,v), clear(t), !red(t), !blue(v)\} \subseteq F$, then no actions are taken. Otherwise, for example if we have the facts

$$!red(a), !red(b), on(a,b), ontable(b), clear(a), !blue(c), ontable(c), clear(c)$$

then the first two rules can be used in a backward-chaining manner to reduce the goal to $\textit{move}^{\perp}$, which adds the fact *move* to the program. Forward-chaining using the last two rules then takes place to get firstly the replacement of *move* $\otimes$ *on(a,b)* $\otimes$ *clear(a)* $\otimes$ *empty* by *clear(b)* $\otimes$ *hold(a)* $\otimes$ *put(a,blue)* and then to replace *hold(a)* $\otimes$ *put(a,blue)* $\otimes$ *clear(c)* with *clear(a)* $\otimes$ *on(a,c)* $\otimes$ *empty*. As there are no actions here, no further changes take place.

In the event that the blocks arrangement changes during computation, then the pre-condition of the *move* action may fail (as a may no longer be on top of b, or a may no longer be clear) or there may not be a clear blue block to place the red one on. In such cases backtracking (including backtracking over the unsuccessful action *move*) will lead

to re-evaluation of the overall goal. Hence when backtracking in the presence of actions, we may need to re-try goals which in the standard logic programming paradigm would fail; this is simply a reflection of the persistence of goals in a situated environment [34].

4 Scheduling Issues

It is one thing to derive a set of inference rules; it is another to design a programming language based on them. In particular, we need to determine an appropriate computational interpretation of forward-chaining and its integration with backward-chaining rules (whose operational behaviour is well understood). In order to do so, we will make two simplifying assumptions, in order to illustrate the principles involved.

The first assumption is based on the observation that the vast majority of programs written in linear logic programming languages do not use linear rules (e.g. [12, 15]). In other words, most applications of linear logic in logic programming use rules which can be used any number of times (including 0) together with a mixture of classical and linear facts (i.e. some of which can be used an arbitrary number of times, and some of which must be used exactly once). For example, a set of rules describing actions that may be taken together with the current state of the world fits this scenario, such as the blocks world or a bin-packing problem. Hence, as far as the forward-chaining aspect is concerned, we can consider a program as a set of classical rules with a mixture of classical and linear facts to be used as input.

Definition 1. *Definite and goal formulæ are defined as follows:*

$$\begin{array}{ll} \textit{Definite formulæ:} & D ::= A \mid \mathbf{1} \mid D \otimes D \mid D \,\&\, D \mid \forall x.D \mid !D \mid G \multimap D \\ \textit{Goal formulæ:} & G ::= A \mid \mathbf{1} \mid G \otimes G \mid G \,\&\, G \mid G \oplus G \mid \forall x.G \mid \exists x.G \mid !G \end{array}$$

A program *is a multiset of closed definite formulae in which every occurrence of $\multimap$ is within the scope of a* !.

Thus we can consider a program to consist of two parts: a multiset of rules R, which must be classical, and a multiset of facts F, which may contain either linear or classical formulae.

The second assumption is also to do with the pragmatics of execution, and in particular the extra choices available in the linear case when compared to the classical one. Consider the program $p, !(p \multimap q), !(p \multimap r)$. Here, as the fact p can be applied to either (but only one of) the two rules, there is a choice to be made as to whether this program should "evolve" to $q, !(p \multimap q), !(p \multimap r)$ or to $r, !(p \multimap q), !(p \multimap r)$.

In the classical case, this choice does not have to be made, as due to the ability to make arbitrary copies of formulae, we can duplicate p and hence apply both rules in parallel. In the linear case, though, we cannot duplicate and hence must make a choice.

In terms of inference rules, it is not difficult to show that we can derive $q \,\&\, r, !(p \multimap q), !(p \multimap r)$ from this program, which essentially delays any choice between q and r to a later point in the computation. However, this introduces the possibility of having to keep track of a large number of possible branches, and so, at least initially, it will be simpler to avoid this behaviour (which may be thought of as an analogy of breadth-first

search in the classical case), if possible. One way to do this is to insist that the rules of the program be *independent* (i.e. they operate on different parts of the facts). For example, given the facts $\{p, q, r\}$ the two rules $!p \multimap r$ and $!q \multimap r$ are independent, but the two rules $\Delta = \{!(p \otimes q) \multimap r, !(p \otimes r) \multimap s\}$ are not independent. If rules are not independent, then we need to use $\&$ as above. For example, if Δ denotes the two rules above, then $p, q, r, \Delta \rightsquigarrow (r \otimes r) \,\&\, (q \otimes s), \Delta$.

This property also makes it possible to think of the rules as independent, in that two rules R_1 and R_2 are either not both applicable or they operate on separate pieces of the program. Hence a program will consist of a number of facts (linear or classical) together with a collection of independent rules $R_1, R_2, \ldots R_n$ and a collection of inter-dependent rules R. The operational semantics of the program will be determined by the way in which these rules are applied to the facts. One reasonable heuristic is to apply the independent rules before the inter-dependent ones, on the grounds that the changes made by the independent rules may break some of the inter-dependencies. Otherwise, if all such inter-dependencies remain, then the only possibility seems to be the use of $\&$ as mentioned above.

This is a finer-grained notion than in the classical case. There, as facts may be arbitrarily copied, all rules are independent, as it is possible to make as many copies as is needed to satisfy each rule. Moreover, systems such as Aditi generally compute "to the fixpoint", i.e. the set of new facts accumulates until no more can be generated. Hence we can think of this as the rules being fired in parallel as many times as needed in order to generate the fixpoint.

We now define *rule scheduling expressions* that can be used to describe strategies for applying rules. Given rules R_1 and R_2 we can apply the rules *sequentially* (denoted by $R_1 R_2$) or in parallel (denoted by $R_1 \cup R_2$). Also, given a rule R we can apply it until it can no longer be applied or a fixpoint is reached (denoted R^*). Note that this notation is similar to regular expressions.

Using this notation we can describe the scheduling applied by systems such as Aditi with the expression $(R_1 \cup R_2 \cup \ldots \cup R_n)^*$, in that a (ground) fact A will be generated by this process if there is some sequence of application of the rules $R_1, \ldots R_n$ to the facts which results in A.

In the linear case, we do not necessary want to compute fixpoints; in particular, unlike the classical case, fixpoints do not always exist. For example, consider the program $p, !(p \multimap q), !(q \multimap p)$. These rules are independent, but repeated application of the rules will see the program oscillate between the above and $q, !(p \multimap q), !(q \multimap p)$. Thus we require that forward-chaining terminates not at fixpoints, but when a given goal G is satisfied.

Now for independent rules R_1 and R_2, it is clear that $(R_1 \cup R_2)(F) = (R_1 R_2)(F) = (R_2 R_1)(F)$, and hence we can choose to execute such rules either in parallel $(R_1 \cup R_2)$ or in a particular sequence. However, we do not know in advance whether or not an application of R_1 or R_2 alone will suffice to prove the overall goal. Furthermore, as a given rule, say R_1 may be applicable to a number of facts in the program, we also need to consider whether we should check for termination after *each* application of R_1, or after *all* applications of R_1.

Hence we need to make two strategic decisions: whether to pursue the independent rules parallel or in some sequence; and whether to pursue the application of each independent rule in parallel or in sequence on all appropriate facts.

In the absence of any other information, a reasonable default would be to do both in sequence, in order to maximise the ability to flexibly react to environmental changes. Hence if each independent rule R_i can be applied n_i times to the facts, this is just the sequence of rule applications of the form $(R_1)^{n_1}(R_2)^{n_2} \ldots (R_m)^{n_m}$ which is essentially a depth-first application of the rule instances.

A sequence of similar granularity is $(R_1 R_2 \ldots R_n)^k$ where $k = max(n_1, n_2, \ldots n_m)$, which is a breadth-first application of the rule instances.

5 Conclusions and Further Work

This paper has discussed various implementation issues for a framework for agents based on mixed mode computation in linear logic. In a sense, this is the first tentative step of the ambitious programme outlined in [14]. In particular, in the integration of the forward- and backward-chaining techniques it seems reasonable to give preference to forward-chaining (i.e. actions and reactions to environmental changes) over backward-chaining. However, this will require some counterintuitive features, such as backtracking over actions and re-trying goals which have already failed. However, both seem reasonable in the context of a dynamic environment.

One clear issue that is raised by the independence property of rules is that the use of aggregate constructs (such as Negation as Failure, or findall) will be crucial to the development of applications. A linear rule such as $A \multimap B$ may be thought of as having an implicit existential quantifier: "if there is a resource A, then change it to B". A complementary rule would then be of the form "if there is no such resource, then ...". A logical method of specifying such rules is clearly of fundamental importance.

Acknowledgements. We would like to acknowledge the support of Agent Oriented Software Pty. Ltd. and of the Australian Research Council (ARC) under grant CO0106934. We would also like to thank Lin Padgham, Omar Rana, David Pym and the Agents group at RMIT for discussions related to this work.

References

1. *JACK Intelligent Agents User Guide*, Agent Oriented Software (AOS), Carlton, 2000.
2. V. Alexiev, Applications of Linear Logic to Computation: An Overview, *Bulletin of the IGPL* 2:1:77-107, 1994.
3. J.-M. Andreoli, Focussing and Proof Construction, *Annals of Pure and Applied Logic* 107:1:131-153, 2001.
4. J.-M. Andreoli and R. Pareschi, Linear Objects: Logical Processes with Built-in Inheritance, *Proceedings of the International Conference on Logic Programming* 496-510, Jerusalem, June, 1990.
5. Michael E. Bratman, *Intentions, Plans, and Practical Reason*, Harvard University Press, Cambridge, MA, 1987.
6. *The dMARS V1.6.11 System Overview*, Technical Report, Australian Artificial Intelligence Institute (AAII), 1996.

7. M.H. van Emden and R.A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *Journal of the Association for Computing Machinery* 23:4:733-742, October, 1976.
8. Stan Franklin and Art Graesser, *Is it an Agent or just a Program?: A Taxonomy for Intelligent Agents*, Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Springer, 1996.
9. M. Georgeff and A. Rao. Rational Software Agents: From Theory to Practice. In *Agent Technology: Foundations, Applications, and Markets*, 139-160, Nick Jennings and Michael Wooldridge (eds.) Springer, 1998.
10. J-Y. Girard, Linear Logic, *Theoretical Computer Science* 50, 1-102, 1987.
11. J-Y. Girard, Y. L and L. Regnier, *Advances in Linear Logic*, London Mathematical Society Lecture Note Series 222, Cambridge University Press, 1995.
12. J. Harland, D. Pym and M. Winikoff, *Programming in Lygon: An Overview*, Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology 391-405, Munich, July, 1996.
13. J. Harland, D. Pym and M. Winikoff, *Forward and Backward Chaining in Linear Logic*, Proceedings of the CADE-17 Workshop on Proof-Search in Type-Theoretic Systems, Pittsburgh, June, 200.
14. J. Harland and M. Winikoff, Agents via Mixed-mode Computation in Linear Logic: A Proposal, *Proceedings of the ICLP'01 Workshop on Computational Logic in Multi-Agent Systems (CLIMA-01)*, Paphos, December, 2001.
15. J. Hodas and D. Miller, Logic Programming in a Fragment of Intuitionistic Linear Logic, *Information and Computation* 110:2:327-365, 1994.
16. J. Hodas, K. Watkins, N. Tamura and K-S. Kang, Efficient Implementation of a Linear Logic Programming Language, *Proceedings of the Joint International Conference and Symposium on Logic Programming* 145-159, June, Manchester, 1998.
17. Marcus Huber, *JAM: A BDI-theoretic Mobile Agent Architecture*, Proceedings of the Third International Conference on Autonomous Agents (Agents'99) 236-243, Seattle, May, 1999.
18. Nick Jennings, An agent-based approach for building complex software systems, *Communications of the ACM* 44:4:35-41, 2001.
19. Nick Jennings and Michael Wooldridge, Applications of Intelligent Agents, in *Agent Technology: Foundations, Applications, and Markets* 3-28, Nick Jennings and Michael Wooldridge (eds.) Springer, 1998.
20. R. Kowalski, Predicate Logic as a Programming Language, *Information Processing 74*, North-Holland, Amsterdam, 1974.
21. R. Kowalski and F. Sadri, From Logic Programming towards Multi-agent Systems, *Annals of Mathematics and Artificial Intelligence*, 1999.
22. M. Masseron and C. Tollu and J. Vauzeilles, Generating Plans in Linear Logic I: Actions as proofs, *Theoretical Computer Science 113:349-371*, 1993.
23. M. Masseron, Generating Plans in Linear Logic II: A geometry of conjunctive actions, *Theoretical Computer Science 113:371-375*, 1993.
24. D.A. Miller, A Multiple-Conclusioned Meta-Logic, Proceedings of the Symposium on Logic in Computer Science 272-281, Paris, June, 1994.
25. D.A. Miller, G. Nadathur, F. Pfenning and A. Scedrov, Uniform Proofs as a Foundation for Logic Programming, *Annals of Pure and Applied Logic* 51:125-157, 1991.
26. G. Plotkin. Structural Operational Semantics (lecture notes). Technical Report DAIMI FN-19, Aarhus University, 1981 (reprinted 1991).
27. D. Pym and J. Harland, A Uniform Proof-Theoretic Investigation of Linear Logic Programming, *Journal of Logic and Computation* 4:2, April, 1994.

28. Anand Rao and Michael Georgeff, Modelling Rational Agents within a BDI-Architecture, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning* 473-484, J. Allen, R. Fikes and E. Sandewall (eds.), Cambridge (USA), 1991.
29. Anand Rao and Michael Georgeff, An Abstract architecture for Rational Agents, *Proceedings of the third International Conference on Principles of Knowledge Representation and Reasoning* 439-449, C. Rich, W. Swartout and B. Nebel (eds.), Boston, 1992.
30. Anand Rao and Michael Georgeff, Decision Procedures for BDI Logics, Journal of Logic and Computation, 8(3):292-342, 1998.
31. A. Scedrov, A Brief Guide to Linear Logic, in *Current Trends in Theoretical Computer Science*, G. Rozenberg and A. Salolmaa (eds.), World Scientific, 1993.
32. Stein, L. A., Challenging the Computational Metaphor: Implications for How We Think, *Cybernetics and Systems*, 30(6), September 1999. Available from `http://www.ai.mit.edu/people/las/papers/`.
33. M. Winikoff, L. Padgham, and J. Harland. Simplifying the development of intelligent agents. In Stumptner, M., Corbett, D., and Brooks, M., editors, *AI2001: Advances in Artificial Intelligence. 14th Australian Joint Conference on Artificial Intelligence*, pages 555–568. Springer, LNAI 2256.
34. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah, "Declarative & procedural goals in intelligent agent systems", in *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, Toulouse, France, April, 2002.
35. Michael Wooldridge, *Reasoning about Rational Agents*, MIT Press, 2000.

A Sequent Calculus for Linear Logic (excerpt)

$$\frac{}{\phi \vdash \phi}\ \text{axiom} \qquad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma', \phi \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \text{cut} \qquad \frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, \forall x\,.\,\phi \vdash \Delta}\ \forall\text{-L} \qquad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \multimap \psi, \Delta}\ \multimap\text{-R}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \otimes \psi \vdash \Delta}\ \otimes\text{-L} \qquad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma' \vdash \psi, \Delta'}{\Gamma, \Gamma' \vdash \phi \otimes \psi, \Delta, \Delta'}\ \otimes\text{-R} \qquad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma', \psi \vdash \Delta'}{\Gamma, \Gamma', \phi \multimap \psi \vdash \Delta, \Delta'}\ \multimap\text{-L}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, !\phi \vdash \Delta}\ !\text{-L} \qquad \frac{!\Gamma \vdash \phi, ?\Delta}{!\Gamma \vdash !\phi, ?\Delta}\ !\text{-R} \qquad \frac{\Gamma \vdash \Delta}{\Gamma, !\phi \vdash \Delta}\ W!\text{-L} \qquad \frac{\Gamma, !\phi, !\phi \vdash \Delta}{\Gamma, !\phi \vdash \Delta}\ C!\text{-L}$$

where y is not free in Γ, Δ.

B Mixed-mode Inference Rules from [13] (excerpt)

$$\frac{}{\mathcal{P} \rightsquigarrow \mathcal{P}}\ \text{Axiom} \rightsquigarrow \qquad \frac{}{A \vdash A}\ \text{Axiom} \vdash \qquad \frac{\mathcal{P} \rightsquigarrow \mathcal{P}'}{\mathcal{P}, !D \rightsquigarrow \mathcal{P}'}\ W \rightsquigarrow$$

$$\frac{\mathcal{P} \rightsquigarrow \mathcal{P}'' \quad \mathcal{P}'' \rightsquigarrow \mathcal{P}'}{\mathcal{P} \rightsquigarrow \mathcal{P}'}\ Cut \rightsquigarrow \qquad \frac{\mathcal{P} \rightsquigarrow \mathcal{P}' \quad \diamondsuit\mathcal{P}' \vdash G}{\mathcal{P} \vdash G}\ Cut \vdash$$

$$\frac{\mathcal{P}, D_1, D_2 \rightsquigarrow \mathcal{P}'}{\mathcal{P}, D_1 \otimes D_2 \rightsquigarrow \mathcal{P}'}\ \otimes \rightsquigarrow \qquad \frac{\mathcal{P}, D \rightsquigarrow \mathcal{P}'}{\mathcal{P}, !D \rightsquigarrow \mathcal{P}'}\ ! \rightsquigarrow \qquad \frac{\mathcal{P}, D[t/x] \rightsquigarrow \mathcal{P}'}{\mathcal{P}, \forall x D \rightsquigarrow \mathcal{P}'}\ \forall \rightsquigarrow$$

$$\frac{\mathcal{P}, !D, !D \rightsquigarrow \mathcal{P}'}{\mathcal{P}, !D \rightsquigarrow \mathcal{P}'}\ C \rightsquigarrow \qquad \frac{\mathcal{P}_1 \rightsquigarrow \mathcal{P}' \quad \diamondsuit\mathcal{P}' \vdash G}{\mathcal{P}_1, \mathcal{P}_2, G \multimap D \rightsquigarrow \mathcal{P}_2, D}\ \multimap\!\rightsquigarrow \qquad \frac{!\mathcal{P} \rightsquigarrow \mathcal{P}'}{!\mathcal{P} \rightsquigarrow !\diamondsuit\mathcal{P}'}\ !M$$

We define $\diamondsuit P$ as $\bigcup_{i=1}^{n} \forall(\otimes P_i)$. We define $\spadesuit P$ as $\bigotimes_{i=1}^{n} \forall(\otimes P_i)$.