Agents via Mixed-mode Computation in Linear Logic

James Harland (jah@cs.rmit.edu.au) and Michael Winikoff (winikoff@cs.rmit.edu.au)
RMIT University, Melbourne AUSTRALIA

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. In this paper we address these issues by re-casting the foundations of such systems into a logic programming framework. In particular we show how the integration of backward- and forward-chaining techniques for linear logic provides a natural starting point for this investigation. We discuss how the integrated system provides for the interaction between the proactive and reactive parts of the system, and we discuss several aspects of this interaction. In particular, one perhaps surprising outcome is that goals and plans may be thought of as *declarative* and *procedural* aspects of the same concept. We also 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.

Keywords: Linear Logic, Mixed-mode computation, Intelligent Agents, Belief Desire Intention (BDI)

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 [23], 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 [24].

There are many possible conceptions of agent-oriented programming. However a common theme [12, 49, 48] is that agent systems should include properties such as

© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

- *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. consisting of a number of agents that interact with each other), *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 [39], 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 states 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 [39] (a version of temporal logic extended to include the appropriate notions of belief, desire and intension) and an architecture for the execution of programs following the BDI paradigm [40].

Whilst BDI-based systems have been successfully applied in a number of areas, there are still foundational and design issues to be addressed. One such issue is the "gap" between the BDI theory, which is a particular multi-modal branching-time temporal logic (in which there is only one past but many possible futures), and the BDI architectures, which are the basis for systems such as dMARS[9], JACK[1] and JAM[22]. The BDI theory has an elegant description of the relationship between beliefs, desires and intentions via possible worlds, but the

architectures tend 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 [41] 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 have 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 [46].

A contrast with Prolog is instructive here. It is hardly a secret that Prolog can be taught to final-year undergraduates; however, whilst it is possible to do so by teaching them the fundamentals of logic (such as well-formed formulae, model satisfaction, logical equivalence, inference rules, etc.) followed by Horn clauses, refutations and so forth, it is by no means necessary to do so. In fact, it is possible to give an intuitive "programming" feel for the language, often by means of some well-chosen examples such as `ancestor`, without a lot of explicit reference to the logical concepts underpinning it all. This informal computational model is generally sufficient to get programmers programming in Prolog, after which the more formal model can be introduced as required.

In order to address these issues, we re-visit the foundations of the BDI framework in order to develop an alternative (and hopefully simpler) view of the semantics of an agent system. It should be noted that this is not intended as a replacement for the BDI semantics, but rather as an alternative, complementary view of the activities of the agent system. In particular, we hope to minimise the number of concepts necessary to the understanding of a simple agent system, along the lines of [46].

Our method for doing this is to re-cast the basic agent model into a logic programming framework. As noted by Kowalski and Sadri [27] 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). Indeed, there has been a noticeable movement within logic programming to include

abduction techniques into logic programming languages [25], which can be viewed as a means of determining how to make goals which fail into goals which succeed, i.e. a means of determining the actions necessary to achieve the goal.

Abductive methods have generally been studied in the context of classical logic, which may be thought of as determining what missing information would make the goal true. In agent systems, the requirement is somewhat stronger; not only does the agent need to know what needs to be achieved, but also it needs to be able to take appropriate action in order to ensure that the desired state of the world is achieved. Thus an agent system requires not just abductive reasoning, but also a suitable notion of action.

One way to provide such a notion is to use *linear logic* [15], 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 [16, 20, 17]. It has also been shown that this logic has a natural notion of *actions*, such as those required by the classic blocks world scenario [29, 30]. Given also the existence of a number of logic programming languages based on linear logic (such as LO [6], Lolli [20], Forum [31], LLP [21] and Lygon [17]), it seems natural to explore the use of linear logic as a basis for BDI-style agent systems.

Some initial work in this direction was done by Abdullah-Al Amin [3]; however, that was limited to the use of a particular language (Lygon) and did not take reactive behaviour into account. In [18] 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. Hence such *mixed-mode* inference is of patricular interest as a basis for agent systems.

This paper thus attempts to re-cast the framework of BDI systems into a logic programming paradigm based on linear logic. Whilst this may be somewhat cynically viewed as yet another exercise in making things look like nails when one only has a hammer, it seems that there is not only a potentially very natural fit between the requirements of a BDI-style system and the properties of linear logic, but also that this exercise in itself will help shed light on the essential properties of BDI systems. This, in turn, will hopefully lead to the simplification of the programming model for such systems.

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 issues in the design of BDI agent systems. In §4 we show how these issues are addressed by the combination of backward- and

forward-chaining in linear logic and in §5 we discuss scheduling issues. Finally in §6 we discuss our conclusions and possibilities for further work.

2. Background

2.1. BDI Agents

The BDI model (Belief Desire Intention) [39, 40] is a popular foundation for intelligent agents. The model has its basis in philosophical investigations by Bratman [8] that discuss the role of *intentions* in (human) planning agents. This philosophical work views beliefs as information that the agent has about the world. Unlike knowledge, this information is not required to be correct. Desires are states of the world that the agent would like to reach. Intentions are future courses of action[1] that the agent has chosen to pursue. In addition to the three concepts of Belief, Desire, and Intention a fourth concept – Plans – is crucial to the framework. Bratman's analysis assumes planning agents.

Intentions are used to reduce deliberation by constraining what courses of action the agent considers. Once an agent has decided on a particular means of achieving a goal (an intention) that agent no longer considers courses of actions that clash with the chosen intention. For example, if a (human) agent intends to leave work early to drive their spouse to the airport then that human should not consider intentions that require it to stay late at work.

Intentions also influence future planning: once an intention has been adopted, the agent can continue planning assuming that the intention will be pursued and (normally) eventually achieved. For example, if an agent intends to drive to the airport and she also has the desire to collect a book from the library then the agent can formulate plans to reach the library based on the route back from the airport going past the library.

Based on these ideas, Rao and Georgeff developed a family of BDI logics, which are multi-modal logics that capture formally the concepts of belief, desire and intention. Using these logics they explored the properties of, and relationship between, the concepts.

However, what makes the BDI model significant from a practical perspective is the development of agent architectures based on the concepts, the implementation of these architectures (e.g. PRS, JAM, dMars, and JACK) and the use of these systems in various applica-

[1] In a general sense encompassing plans and goals, not just single actions.

tions including fault diagnosis on the space shuttle, scheduling aircraft landings and robotic soccer [24].

From the pragmatic perspective of the implemented systems the concepts are slightly different. Whereas the philosophy talks about desires, and the multi-modal logics address goals[2], the implemented systems deal with *events*. Events are "significant occurrences" and are used to model the addition of a new goal or sub-goal. The event triggers plans that attempt to achieve the goal.

Like the philosophical viewpoint, but unlike the logical one, plans play a crucial role. Plans in implemented BDI systems consist of

- a body describing the primitive actions or sub-goals that have to be achieved for plan execution to be successful;
- an invocation condition which specifies the triggering event, and
- a context condition which specifies the situation in which the plan is applicable.

The execution cycle of BDI systems such as dMars and JACK includes the following steps:

1. The event being handled is matched against the plan library to obtain a set of plans that are *relevant* to the event.
2. The context conditions of the relevant plans are evaluated[3]. The subset of the relevant plans with a true context condition is the *applicable* set.
3. An applicable plan is selected and added to the intention stack.
4. An intended plan is selected and its first step is executed. This first step can be an action (that is performed), or a sub-goal that posts an event.

We now turn to intentions. Whereas intentions are arguably the centrepiece of Bratman's philosophical work, in BDI implementations intentions are merely a call-stack of plans. Implemented systems such as dMars and JACK do not perform the sort of reasoning that Bratman describes, where the existence of an intention constrains the choice of plans that an agent will consider.

[2] Goals differ from desires in that they must be consistent

[3] The systems differ in exactly how this is done. For example, whereas JAM evaluates the context conditions of all relevant plans at once, JACK evaluates the context conditions one at a time, stopping when an applicable plan is found.

Table I. BDI: Philosophy vs. Theory vs. Implementation

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

The BDI model has developed over about 15 years and while there are certainly strong relationships between the theoretical work and implemented systems, the different viewpoints are not as closely integrated as one might wish. The concepts that have been found to be useful for development do not necessarily match those concepts most developed in the theoretical work. Neither are they necessarily exactly the concepts which have arisen within particular implemented systems such as JACK. For example, in the theoretical model plans are simply beliefs or intentions. However in the implementations plans are a central concept. Events are ignored in the theoretical framework but play a key role in implementations, although they are not well distinguished from goals.

Further, the overloading of terms has lead to confusion; this is particularly evident around intentions. Some key differences between the philosophy, theory, and implementation viewpoints of BDI are shown in table 2.1.

2.2. Linear Logic

Linear logic [15] 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 [15, 16], database updates [20] and planning problems [29, 30].

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 (see below) to be made between the resources used), as, by default, each formula in linear logic represents a resource which must be used exactly once.

A classic example is the following menu from a restaurant:

fruit or seafood (in season)
main course
all the chips you can eat
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 ! behave 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, while ! `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:

$$\begin{aligned}(F_1 \otimes F_2)^{\perp} &\equiv (F_1)^{\perp} \parr (F_2)^{\perp} \\ (F_1 \parr F_2)^{\perp} &\equiv (F_1)^{\perp} \otimes (F_2)^{\perp} \\ (F_1 \oplus F_2)^{\perp} &\equiv (F_1)^{\perp} \,\&\, (F_2)^{\perp} \\ (F_1 \,\&\, F_2)^{\perp} &\equiv (F_1)^{\perp} \oplus (F_2)^{\perp}\end{aligned}$$

Each of these four connectives also has a unit, which, for $\otimes$ and $\&$ are written as $\mathbf{1}$ 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 $\mathbf{0}$, 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 [15, 16, 42, 2], among others. The sequent calculus for linear logic is given in Appendix A.

2.3. Logic Programming in Linear Logic

There has been a significant amount of work on logic programming languages based on linear logic, including LO [6], Lolli [20], LinLog [4], Forum [31], and Lygon [17]. Such languages have been successfully applied to a number of application areas, including database updates, natural language processing, concurrency, and knowledge representation.

A common way in which such languages are identified from the inference rules of the logic is via the properties of a certain class of proofs known as *goal-directed* [32] proofs. Roughly speaking, given a sequent[4] $\Gamma \vdash \Delta$ the goal-directed property requires that Δ be reduced before Γ, and so inference rules which reduce Δ are preferred over those which reduce Γ. In this way the analysis is one based on *backward-chaining*, in that a program and goal are given, and the rules of the linear sequent calculus are used, in this goal-directed manner, to determine whether or not the sequent is provable.

The completeness of properties of goal-directed proofs can then be used as a criterion for the identification of logic programming languages [32]. In particular, a particular fragment of the logic (such as Horn clauses or hereditary Harrop formulae [32] is considered a logic

[4] "$\Gamma \vdash \Delta$" is a sequent, where Δ and Γ are both multisets of formulae. The formulae on the left of the turnstile (Γ) are the *antecedent*, and those on the right (Δ) the *succedent*. The sequent is (roughly) read as "some formula in Δ follows from Γ".

programming language iff goal-directed proofs are complete for that fragment (i.e. there is a proof iff there is a goal-directed proof).

The use of linear logic (rather than classical or intuitionistic logic) means that whilst the query-and-answer style of Prolog is maintained, the process of searching for a proof (i.e. computation) takes place in a more dynamic environment. Specifically, as some formula in the program must be used exactly once, it is necessary to keep track of which formulae have been used earlier in the search and which are still available to be used, and so the execution of a program in such a language requires the maintainance of the current state of resources available. It is this property that makes linear logic programming languages appropriate for applications involving state changes, such as database transactions, bin packing algorithms and planning [29, 30].

For example, it is straightforward to implement planning applications such as the blocks world. Below we describe the Lygon code for this example from [17] (although this example will be similar in any linear logic programming language). A number of blocks are sitting either on a table or on another block and there is a robotic arm capable of picking up and moving a single block at a time.

The predicates used in the Lygon program below are the following:

- *empty:* the robotic arm is empty;
- *hold(x):* the robotic arm is holding block x;
- *clear(x):* block x does not support another block;
- *ontable(x):* block x is supported by the table;
- *on(x,y):* block x is supported by block y.

There are a number of operations that change the state of the world. The rules for each of these are simple enough to write and are given below. Note that this is little more than writing down the pre- and post-conditions for each action. For example, *taking* a block requires that the arm is empty, and that the chosen block is clear, and results in the block being held by the robot. We can write similar rules for other actions, such as *removing* a block from the block beneath it, which must be done before picking up the bottom block, *putting* a block down on the table or *stacking* it on another block.

$$!\forall x\ (empty \otimes clear(x) \otimes ontable(x)) \otimes hold(x)^{\perp} \multimap take(x)$$
$$!\forall x, y\ (empty \otimes clear(x) \otimes on(x,y)) \otimes (hold(x)^{\perp} \parr clear(y)^{\perp}) \multimap remove(x,y)$$
$$!\forall x\ hold(x) \otimes (empty^{\perp} \parr clear(x)^{\perp} \parr ontable(x)^{\perp}) \multimap put(x)$$

$$!\forall x, y\ (hold(x) \otimes clear(y)) \otimes (empty^{\perp} \parr clear(x)^{\perp} on(x, y)^{\perp}) \multimap stack(x, y)$$

Given a query such as $take(a)$, these rules can be used to reduce the query to the goal

$$(empty \otimes clear(a) \otimes ontable(a)) \otimes hold(a)^{\perp}$$

and if the first three conditions succeed, these are deleted, and replaced with $hold(a)$.

Note that we could so also write the rule for remove as below.

$$!\forall x, y\ (empty \otimes clear(x) \otimes on(x, y)) \otimes (hold(x) \otimes clear(y))^{\perp} \multimap remove(x, y)$$

and similarly for *stack* and *put*.

Finally, to initialise the state of the blocks, we can use a rule such as

$$ontable(a)^{\perp} \parr ontable(b)^{\perp} \parr on(c, a)^{\perp} \parr clear(b)^{\perp} \parr clear(c)^{\perp} \parr empty^{\perp} \multimap initial$$

to specify the initial state of the blocks (in this case blocks a and b ont he table with c on top of a) together with the goal below, specifying the actions to be performed.

$$initial \parr remove(c, a) \parr put(c) \parr take(a) \parr stack(a, b)$$

This computation will then initialise the blocks and attempt to perform the actions specified. The order of the instructions *take*, *put* etc., is not significant; there are some actions, as specified by the rules, such as *put(c)*, which cannot take place from the initial state, and others, such as *take(b)* which can. It is the problem of the implementation to find an appropriate order in which to execute the instructions, so giving the final state.

In general, we can write an action Act with pre-conditions Pre and post-conditions $Post$ as

$$Pre \otimes Post^{\perp} \multimap Act$$

This rule has the effect that when the query Act is asked, then if the conditions Pre are satisfied, these are deleted and the conditions $Post$ are asserted in their place.

Hence given a current state $State$ and a number of actions to perform, of which Act is one, we can proceed from

$$State_1, State_2 \vdash Act, Rest$$

by using the rule

$$Pre \otimes Post^{\perp} \multimap Act$$

to

$$State_1, State_2 \vdash Pre \otimes Post^{\perp}, Rest$$

and then to

$$State_1 \vdash Pre \qquad State_2, Post \vdash Rest$$

provided that $State_1 \vdash Pre$ is provable.

Other applications discussed in [17, 45] include cycle-tolerant graph searches (such as finding Hamiltonian circuits), the Yale shooting problem, counting programs and topological sorts.

2.4. Bottom-up Execution

The execution models on which linear logic programming languages are based have generally been designed to use *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.

In addition, this query-and-answer paradigm is perhaps a little unnatural for representing actions and transitions. In particular, when the final state is reached (i.e. all the actions have been performed), it is necessary to find a way to "close" the proof and report the final state found. Moreover, given some simple pre- and post-conditions such as those found in the blocks world, it seems a little unusual to write a transition as a goal of the form

$$Pre \otimes Post^{\perp}$$

rather than as a rule of the form

$$Pre \multimap Post$$

The latter form also allows extra conditions to be added to the premise, if desired,

$$Act \otimes Pre \multimap Post$$

to allow some finer control over the action (i.e. so that it just doesn't trigger whenever the pre-conditions are satisfied, but also only when it is determined that this action is the right choice to make).

The above rules for the blocks world would then become

$$!\forall x\ take(x) \otimes (empty \otimes clear(x) \otimes ontable(x)) \multimap hold(x)$$
$$!\forall x, y\ remove(x, y) \otimes (empty \otimes clear(x) \otimes on(x, y))) \multimap hold(x) \otimes clear(y)$$
$$!\forall x\ put(x) \otimes hold(x) \multimap (empty \otimes clear(x) \otimes ontable(x))$$
$$!\forall x, y\ stack(x, y) \otimes (hold(x) \otimes clear(y)) \multimap (empty \otimes clear(x) \otimes on(x, y))$$

where the initial state includes

$$ontable(a),\ ontable(b),\ on(c, a),\ clear(b),\ clear(c),\ empty,\ remove(c, a)\ ,\ put(c)\ ,\ take(a),\ stack(a, b)$$

which can then be transformed into the state resulting after the application of all the actions (in an appropriate order), i.e.

$$ontable(c),\ ontable(b),\ on(a, b),\ clear(c),\ clear(a),\ empty$$

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 [26, 32, 38, 20]; the integration of forward-chaining techniques into such a system was investigated in [18]. 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 [43, 5], amongst others, the process view of computation is one that is becoming increasingly important, and in which safety considerations are vital.

Kowalski and Sadri [27] 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.

Hence our approach is to develop inference rules which incorporate forward-chaining techniques into the sequent calculus (which is the standard framework for backward-chaining systems). The integrated set of rules is taken from [18] and can be found in Appendix B.

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). As discussed in [18], 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 $\mathcal{P} \vdash \mathcal{G}$ (or, for that matter, that $\mathcal{P}, \mathcal{G} \multimap D \rightsquigarrow D$, as the two premises are sufficient to establish that $\mathcal{P} \vdash \mathcal{G}$), 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. Further discussion on this inference system may be found in [18].

3. Designing Agent Systems

In order to examine the complexities of the BDI model, it is instructive to ask what concepts comprise the *minimal necessities* for an agent system. In other words, what, intuitively, is the most fundamental difference between an agent-oriented programming paradigm and other programming paradigms? Naturally there are many possible answers to this question, but one reasonable answer would be that an agent system needs to have a *goal-directed semantics*, i.e. a means of acting to achieve its objectives in a dynamic and unpredictable environment. For example, consider a drink-waiter agent, who is instructed to produce the finest wine possible. This instruction (i.e. goal) may produce different results when given at three different times; the wine that was produced the first time may be out of stock by the second request, or a better wine may have come into stock between the requests, or by the third request, a still better wine may be in stock, but cannot be served in time as it requires a certain amount of chilling or airing before being served.

Whilst one can certainly generate a seemingly arbitrarily long list of features necessary to the function of an agent system, in order to simplify the conceptual model of agents, it seems reasonable to concentrate on this central notion before developing others. Hence we will focus on goals, rather than take as our point of conceptual deparure the threefold interaction of beliefs, desires and intentions.

In general, an agent may have several goals; indeed, one may think of such a system having a number of desires (which are unconstrained) and from these determining a number of goals, which are required to have a number of properties (such as being consistent, possible, not yet achieved, etc. [47]. In addition, as discussed in [33], it seems reasonable to consider only goals for which a feasible means of achievement is

available. For example, there is little point in having a goal such as making it rain tomorrow (which may well be consistent, possible, not yet achieved and so forth) but which the agent has no feasible means to achieve. As we shall see, this property will have an important effect in the design of such systems.

One of the fundamental technical issues in the design of BDI systems is then to determine how best to balance the competing needs of being proactive (i.e. pursuing one's goals *despite* the environment) and being reactive (i.e. adapting one's behaviour *in response to* the environment). At one extreme are classic planning systems such as STRIPS [11], in which plans are generated in great detail in advance, and only executed once all steps are known (which often comes at great computational expense). A classic criticism of such systems is that by the time that their plans are completed, they may have long since been rendered obsolete by changes in the environment. At the other extreme are purely reactive systems, in which there is no time to do anything but use pre-determined actions corresponding to the given inputs.

The architectural realisation of this tension is the need to both execute a given course of action, as well as to periodically evaluate whether this is indeed the most appropriate action to take, and, if necessary, suspend (or abort) the current plan in favour of another.

Hence it seems necessary for an agent system to have at least the specific functionalities 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

For example, consider the problem of providing a drink to a thirsty person (as discussed in [40] and [27]). The main goal is to quench the drinker's thirst[5]. There are two ways to do this: one is to drink lemonade, the other to drink water. To drink lemonade, one needs to be holding a bottle of lemonade, a glass (presumably clean and empty) and then perform a pour action followed by a drink action. To be holding a bottle of lemonade, one has to open the fridge and take a bottle. To drink water, one needs to have a glass, fill it with water from the tap and then perform a drink action.

[5] Here, for simplicity, we have ignored time constraints, such as achieving this goal within 10 units of time. As in [27] one could simulate such constraints by the judicious use of an extra argument and some equalites and inequalities.

As discussed in [27] it is possible to represent the state information via clausal rules similar to those of Prolog, such as those below (taken from [27] with some minor variations):

holds(quench-thirst)	iff	*holds(drink-lemonade)* or *holds(drink-water)*
holds(drink-lemonade)	iff	*holds(have-glass)* and *holds(have-lemonade)* and *do(pour)* and *do(drink)*
holds(have-lemonade)	iff	*do(open-fridge)* and *do(get-lemonade)*
holds(drink-water)	iff	*holds(have-glass)* and *do(open-tap)* and *do(drink)*

The key point to note is that as the above rules define the conditions under which various pieces of state information hold *(holds(drink-lemonade), holds(have-lemonade)*, etc.), the bodies of the rules contain a mixture of states assumed to have been achieved (e.g. *holds(have-lemonade)*) together with actions needed to achieve some new state (e.g. *do(open-fridge)*). Hence one can think of the above rules as defining sets of actions[6] which, when performed, will achieve the desired state. In particular, the ability to use "intermediate" states, such as *holds(have-lemonade)*, means that alternatives can be easily integrated. For example, if two new methods of getting a bottle of lemonade become available (say purchasing one from a nearby vending machine and ordering one over the Internet), then these can be simply added as extra cases to the rule for *holds(have-lemonade)* without changing any other parts of the code. This flexibility is one of the key features of agent systems, as it shows clearly the way in which alternative methods of achieving the same goal can be interleaved as required in response to environmental changes.

An alternative presentation of the above example is

to *achieve(quench-thirst)*	try	*achieve(drink-lemonade)* or *achieve(drink-water)*
to *achieve(drink-lemonade)*	try	*(achieve(have-glass)* and *achieve(have-lemonade))* then *do(pour)* then *do(drink)*
to *achieve(have-lemonade)*	try	*do(open-fridge)* then *do(get-lemonade)*

[6] In fact, in [40] and [27] these are *sequences* of actions, with the sequencing obtained in the former case by the use of procedural programming techniques, and in the latter via an extra argument for time for each predicate.

to *achieve(drink-water)* try *achieve(have-glass)* then *do(open-tap)* then *do(drink)*

where we distinguish between "and" which does not imply any sequencing, and "A then B" which implies that A must be done before B. This presentation is more "honest" in that it emphasises that the link between a plan body and the head is partial: there can be no guarantee that executing a plan will actually achieve the goal which is the head of the clause (due to environmental changes possibly altering the correctness of the reasoning process during computation).[7] All that can be said is that the body is a reasonable way to try and and achieve the goal. More discussion on this point can be found in [47].

As the above thirst-quenching example makes clear, in order to achieve the top-level goal, it is generally necessary to consider a mixture of sub-goals, plans, and actions. In fact, as there is an understood nexus between goals and plans (such as in the JACK system, in which goals are represented by events, and every event must be handled by some plan, and every plan must have some event that it handles), it seems reasonable to consider them not as separate concepts which need to be somehow linked up, but as different aspects of the same underlying concept. As discussed above, a goal can reasonably be expected to require that there is some plan which will achieve it (perhaps as yet unknown, but able to be determined effectively and efficiently); conversely, plans come with pre-conditions and intended outcomes, which indicate information about the state of the world both before and after execution. In logic programming terms, both goals and plans have *declarative* and *procedural* content — the goal specifies *what* needs to be achieved, and the plan specifies *how* it is to be achieved. *Hence as goals need to be associated with some procedure to achieve them, and plans require a goal which they achieve, it seems sensible to think of the combination of goal and plan as a unified concept.* The unified goal/plan concept is the *clause*: a clause of the form $G \leftarrow P$ has a goal (the head) and a plan (the body). As discussed earlier, the link between them is of the form "this is a reasonable decomposition to try". The plan can contain other goals which are in turn decomposed. Hence the unification is achieved by the ability to interpret clauses both declaratively and procedurally, although in this case the procedural interpretation may include actions and state changes as well as proof-search.

As discussed in [32], logic programming can be thought of as a particular implementation of *goal-directed proof search*, in that computation is specified by a given goal G, and the way the computation proceeds

[7] For that matter, there is also no guarantee that the failure of all plans to achieve a given goal implies the failure of the goal.

is dependent primarily on G rather than the program itself. In order to achieve the above integration, it is clearly necessary to extend the notion of a goal to include actions (and hence plans) in some way. As mentioned above, this has happened to some extent via abduction methods (as well as in the work of Gabbay [13]) in order to specify what needs to be achieved in order to make the top-level goal succeed; below we will discuss how this can be done in linear logic by means of actions as well.

Now in order to achieve the three functionalities mentioned above, backchaining can be used to decompose a goal into subgoals (see point 1 above). For example, in the quench-thirst example above, it is simple enough to see how the goal *holds(quench-thirst)* can be reduced to

holds(have-glass) and do(open-fridge) and do(get-lemonade) and do(pour) and do(drink)

or alternatively to

holds(have-glass) and do(open-tap) and do(drink).

Clearly the standard logic programming techniques of resolution and backtracking can be used to perform this reduction. In general a goal G is reduced by this process to a set of sufficient sub-goals and actions $G_1, G_2, \ldots G_n$. If any one of these sub-goals is defined by a rule in the above manner, then further such reduction can be performed. If not, then we need to be able to find a set of actions (i.e. a plan) which will achieve this goal.

From the point of view of planning, the relationship between the subgoals can vary — the G_i can be viewed disjunctively (i.e. the success of one sub-goal is sufficient for the success of the overall goal), conjunctively (i.e. the success of all sub-goals is required for the success of the overall goal) or as a sequence. Naturally the first two possibilities fit naturally into a logic programming context; the third would require a logical notion of sequence. This is generally a difficult problem for logic programming languages to solve cleanly. However, some recent work by Polakow [36, 35] on ordered linear logic (i.e. non-commutative linear logic) may be a useful solution to this problem.

Another point to note is that as the environment can change during the backchaining process, it may be necessary to verify that the achievement of the sub-goals, once attained, does actually achieve the overall goal. In doing so it would presumably be useful to keep track of which chains of reasoning will remain true despite changes in the environment (such as the above decomposition of quenching thirst) as distinct from

those which depend either on the current state of the environment (which may change) or on the success of some action (which may fail).

Hence an agent arrives at a list of sub-goals it wishes to pursue, and generates a list of plans which would achieve them. It then selects a given plan to execute. In order to incorporate reactive behaviour as well, at some point during execution the environment is checked to determine what has changed. At this point it is possible that another one of the applicable plans is selected, and work on the original is suspended. If a plan fails, then it is discarded. If all plans fail, then the parent plan fails. Hence the proactive behaviour extends only to the level of plans; one doesn't investigate the cause of failure, but instead looks for an alternative.

In order to provide this functionality within the system, what is required is a means of re-evaluating the previous planning process. One way to do this is to simply re-try the original goal, and see what results. Either the same conclusion will be reached (i.e. that no changes (or at least none of significance) have occurred), or a new conclusion will be reached, presumably due to environmental changes.

Hence reactivity is not so much about firing off triggers and the like in response to a given signal, but more about re-evaluating whether the current plan is still the best option.

4. Agents via Mixed-mode Computation in Linear Logic

4.1. Representation

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[8]. We also allow events ($\mathcal{E}$) and actions ($\mathcal{A}$) to appear.

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

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 (do(A)) to the context triggers forward chaining using a directed cut (labelled $Cut\vdash$ below):

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

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

$$!(\ \mathit{get_lemonade} \otimes \mathit{fridge(open)} \multimap \mathit{fridge(open)} \otimes \mathit{have_lemonade}\ )$$
$$!(\ \mathit{open_fridge} \otimes \mathit{fridge(closed)} \multimap \mathit{fridge(open)}\ )$$

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. $\Delta, A \vdash \Gamma$ or $\Delta, \mathit{Evdash}\Gamma$) then forward chaining is performed. Permutabilities of the inference rules can be used to precisely characterise where rule selection policies affect completeness; note that completeness will need to be compromised in order to rule out backtracking over performed actions.

Another issue is that of actions failing. For example, in the above *do(get-lemonade)* action, a pre-condition is that the fridge door is open. If it is closed when this action is executed, then the agent could either attempt to open the door, or abandon the current action (and hence current plan) in favour of some other option. To represent the second scenario would require an extra rule for the *do(get-lemonade)* action along the lines of the one below.

$$!(\ \mathit{get_lemonade} \otimes \mathit{fridge(closed)} \multimap \mathit{fails(get_lemonade)}\ )$$

In general when an action's pre-conditions are not satisfied, we can either fail, attempt to make the pre-conditions true, or delay (i.e. in order to try the action again at a later time). Current BDI systems make the action (and hence the current plan) fail, and is not re-tried. How to deal with such issues remains an interesting question.

4.2. An Example

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,y)	put block x onto y

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 $!\forall x\ p \otimes q \multimap r \otimes s$ is parsed as $!(\forall x\ ((p \otimes q) \multimap (r \otimes s)))$.

$$! \forall x, y\ \ red(x) \otimes blue(y) \otimes clear(x) \otimes on(x,y) \multimap redtop$$

$$! \forall x, y\ \ move^{\perp} \multimap red(x) \otimes blue(y) \otimes clear(x)$$

$$! \forall x, y, z\ \ move \otimes on(x,y) \otimes red(x) \otimes red(y) \otimes clear(x) \otimes empty \multimap clear(y) \otimes hold(x) \otimes put(x,y) \otimes blue(z)$$

$$! \forall x, y\ \ move \otimes ontable(x) \otimes red(x) \otimes clear(x) \otimes empty \multimap hold(x) \otimes put(x,y)$$

$$! \forall x, y\ \ put(x,blue) \otimes hold(x) \otimes clear(y) \otimes blue(y) \multimap on(x,y) \otimes clear(x) \otimes empty$$

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

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

then the first two rules can be used in a backward-chaining manner to reduce the goal to $move^{\perp}$, which adds the fact *move* to the program. This makes the third rule above applicable, as the premise is satisfied, and applying this rule results in the replacement of

move, on(a,b), clear(a), empty

with

clear(b), hold(a), put(a,c), blue(c)

This makes the last rule above applicable, which replaces

hold(a), put(a,c), clear(c)

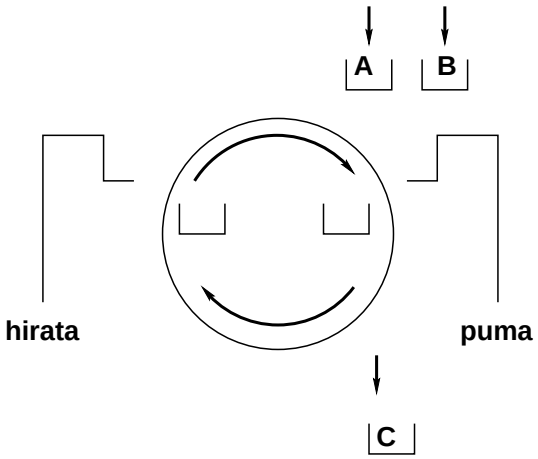
with

clear(a), on(a,c), empty

As there are no actions here, no further changes take place (note that block a, which is red, is now on top of the blue block c).

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 [47].

A second example, similar in spirit to the blocks world, comes from holonic manufacturing. Here there are two robots: one that moves parts around (puma) and one that screws parts together (hirata). There is a rotating table with one jig situated between the two robots, as in the diagram below.



The steps required to produce a finished part are for puma to place an A in the jig and a B in the jig, the table to move the jig to hirata, hirata to join A to B, the table to move AB to puma and puma to unload AB. The parts A and B come from separate buffers (possibly loaded by another agent, but not dealt with in this example) and the completed part is to be placed in the C buffer.

We can represent this situation as follows:

$$! \forall x\ loadA \otimes a_buffer(x) \multimap in_jig(a(x))$$
$$! \forall x\ loadB \otimes b_buffer(x) \multimap in_jig(b(x))$$
$$! \forall x\ move_1 \otimes at_jig(puma) \otimes in_jig(a(x)) \otimes in_jig(b(x)) \multimap at_jig(hirata) \otimes in_jig(a(x)) \otimes in_jig(b(x))$$
$$! \forall x, y, z\ join \otimes at_jig(hirata) \otimes in_jig(a(x)) \otimes in_jig(b(y)) \otimes join(x, y, z) \multimap in_jig(c(z)) \otimes at_jig(hirata)$$
$$! \forall x\ move_2 \otimes at_jig(hirata) \otimes in_jig(c(x)) \multimap at_jig(puma) \otimes in_jig(c(x)$$
$$! \forall x, y, z\ unload \otimes at_jig(puma) \otimes in_jig(c(x)) \multimap c_buffer(c(x)) \otimes at_jig(puma)$$

$$! loadA^{\perp} \parr loadB^{\perp} \parr move_1^{\perp} \parr join^{\perp} \parr move_2^{\perp} \parr unload^{\perp} \multimap c_buffer(c(x))$$

Denoting the above rules as R, from the overall goal $c_buffer(c(3)) \otimes \top$ and the initial state $at_jig(puma)$ we get the following sequence:

$$R, at_jig(puma) \vdash c_buffer(c(3))$$
$$R, at_jig(puma), unload, move_2, join, move_1, loadA, loadB \vdash \perp$$

and

$$R, at_jig(puma), a_buffer(1), b_buffer(2) munload, move_2, join, move_1, loadA, loadB \rightsquigarrow$$
$$R, at_jig(puma), b_buffer(2), unload, move_2, join, move_1, in_jig(a(1)), loadB \rightsquigarrow$$
$$R, at_jig(puma), unload, move_2, join, move_1, in_jig(a(1)), in_jig(b(2)) \rightsquigarrow$$
$$R, at_jig(hirata), unload, move_2, join, in_jig(a(1)), in_jig(b(2)) \rightsquigarrow$$
$$R, at_jig(puma), unload, in_jig(c(3)) \rightsquigarrow$$
$$R, at_jig(puma), c_buffer(c(3))$$

where $join(1, 2, 3)$ holds.

Naturally there are various ways to make this example more realistic, such as considering timing information or adding a second jig to the tray. The main point is to note that the interaction between actions and goals can be simply expressed by rules of this form.

4.3. Systems of Agents

Most of our discussion (and presented formalism) has focussed on a single agent. There is a natural extension to multiple agents which simply uses separate sequents to mark the context boundary between agents. In particular, an agent has access to its own beliefs only. We introduce the ability to have multiple agents by simply maintaining parallel sequents

$$\mathcal{B}_1, !\mathcal{P}_1 \vdash_1 \mathcal{G}_1 \qquad \ldots \qquad \mathcal{B}_n, !\mathcal{P}_n \vdash_n \mathcal{G}_n$$

and allow for communication by sending events to other agents. Sending an event is denoted by $E \overset{X}{\rightarrowtail} G'$ which sends the event E to agent X and then continues with G'. The rule below shows how agent 1 sends a message to agent 2.

$$\frac{\mathcal{B}_1, !\mathcal{P}_1 \vdash_1 G', \Gamma \qquad E, \mathcal{B}_2, !\mathcal{P}_2 \vdash_2 \Delta}{\mathcal{B}_1, !\mathcal{P}_1 \vdash_1 E \overset{2}{\rightarrowtail} G', \Gamma \qquad \mathcal{B}_2, !\mathcal{P}_2 \vdash_2 \Delta} \; Send$$

The messaging connective $\rightarrowtail$ can be seen as a form of distributed implication. Instead of adding a formula to the premises of its sequent, like normal implication, it adds a formula to another sequent's premises. The use of multiple, parallel, sequents is reminiscent of Avron's hypersequents [7].

The effects of actions can be captured by making the environment into a distinguished agent. This agent (the Environment Agent, EA) has no goals, it simply performs a cycle of receiving actions from agents, performing them, and sending percepts (based on the new state of the world) to agents.

5. 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. [17, 20]). 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.

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. If rules are not independent, then we need to use $\&$ as above. For example, if Δ denotes the two rules above, then

$$p, q, r, \Delta \rightsquigarrow (r \otimes r) \,\&\, (q \otimes s), \Delta$$

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, deductive database systems such as Aditi [44], which have used a combination of backward-and forward-chaining in the classical case, 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 is the maximum of n_1, n_2, $\ldots n_m$, which is a breadth-first application of the rule instances.

6. Conclusions and Further Work

This paper presents a proposal for a framework for agents based on mixed mode computation in linear logic. As we have seen, linear logic can be used to provide a natural framework for state changes and actions, and the combination of backward- and forward-chaining enables us to separate reactive computation (percepts and events) from proactive computation (goals and plans). Reactive computation is captured by forward-chaining (which, due to the basis in linear logic, naturally incorporates state changes), and goal decomposition (or planning) takes

place via backward-chaining. The interaction between the two systems then becomes a matter of judiciously chosen directed cuts[9]. 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. This will require some potentially 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.

We have also discussed how the logic programming framework allows the use of program clauses such as $G \leftarrow P$ to unify the notions of goal and plan into a single concept. Simplifications such as this one are an important pragmatic consideration for the designers of agent systems.

In this way this paper describes a broad agenda for future work aimed at providing a clean, elegant, natural, and powerful foundation for situated agents which are reactive, proactive, and autonomous. Whilst this is somewhat ambitious, our previous experience in implementation of linear logic programming languages [18, 17], together with the technical evidence, strongly suggest that this agenda is feasible.

Naturally there remains a considerable amount of further work to be done. One such item is the further development of the mixed mode system. Whilst the inference rules appear satisfactory, there remains a considerable amount of non-determinism, and hence any computational model based on these rules will require an appropriate means for dealing with this. For example, given a formula A and two rules $A \multimap B$ and $A \multimap C$, we need some method for choosing which rule to apply. In addition, as the application of one forward-chaining rule may trigger other rules, we need to determine whether to execute such rules in a depth-first or breadth-first (or iterative deepening) manner. In short, development of an appropriate operational semantics is required. The extension of the mixed mode system to other logics (such as affine logic, light linear logic, **BI** [37], etc.) is also a topic of interest.

Another item is the further pursuit of this particular notion of agent-oriented programming. For example, it is not clear how the notion of intention fits into our framework (although it seems reasonable that the search strategy used to solve the goals/execute the plans may be the manifestation of this concept). Naturally, beliefs should be considered as well, although these appear to be less problematic. In addition, actions, even when the pre-conditions are satisfied, may still fail, and hence we need to be able to model this in our framework. One possibility is to add a rule such as $B \multimap C$ to the environment when outcome B

[9] In the proof-theoretic sense, not in the sense of a pruning operator.

is expected; when it arrives, we can easily perform the computation from $B, B \multimap C$ to C. If it does not arrive (or at least not within some reasonable period), the presence of $B \multimap C$ can be used to indicate the absence of the expected outcome.

Development of the properties of the mixed mode inference system and the agent framework will presumably be symbiotic in nature, in that developments in one will suggest possible developments in the other. In addition, implementation of the system and some sample applications will greatly assist this process.

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.

This broad agenda may be characterised as an attempt to delineate a programming paradigm of *goal-directed computation*; in other words, the key aspect is to determine how to satisfy the given goal, whether it is a Prolog-style goal (i.e. test for truth, possibly determining some extra information when successful) or a planning-style goal (i.e. change the world to achieve this state), some other style (such as *maintaining* a given goal in the manner of integrity constraints, or endlessly processing requests as they are made to a server). In all cases, the ability to balance proactive and reactive behaviour is critical to the design of the system.

Appendix

A. Sequent Calculus for Linear Logic

$$\frac{}{\phi \vdash \phi}\ \text{axiom} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma', \phi \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \text{cut}$$

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

$$\frac{\Gamma \vdash \Delta}{\Gamma, \mathbf{1} \vdash \Delta}\ \mathbf{1}\text{-L} \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1}\text{-R}$$

$$\frac{}{\bot \vdash}\ \text{L}^{\bot} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot, \Delta}\ \text{R}^{\bot}$$

$$\frac{}{\Gamma, \mathbf{0} \vdash \Delta}\ \mathbf{0}\text{-L} \qquad \frac{}{\Gamma \vdash \top, \Delta}\ \top\text{-R}$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \phi^{\perp} \vdash \Delta}\ \perp\text{-L} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \phi^{\perp}, \Delta}\ \perp\text{-R}$$

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

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, \phi \,\&\, \psi \vdash \Delta} \qquad \frac{\Gamma, \psi \vdash \Delta}{\Gamma, \phi \,\&\, \psi \vdash \Delta}\ \&\text{-L} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \,\&\, \psi, \Delta}\ \&\text{-R}$$

$$\frac{\Gamma, \phi \vdash \Delta \qquad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \oplus \psi \vdash \Delta}\ \oplus\text{-L} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi \oplus \psi, \Delta} \qquad \frac{\Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \oplus \psi, \Delta}\ \oplus\text{-R}$$

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

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

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

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

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

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

$$\frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, \forall x\,.\,\phi \vdash \Delta}\ \forall\text{-L} \qquad \frac{\Gamma \vdash \phi[y/x], \Delta}{\Gamma \vdash \forall x\,.\,\phi, \Delta}\ \forall\text{-R}$$

$$\frac{\Gamma, \phi[y/x] \vdash \Delta}{\Gamma, \exists x\,.\,\phi \vdash \Delta}\ \exists\text{-L} \qquad \frac{\Gamma \vdash \phi[t/x], \Delta}{\Gamma \vdash \exists x\,.\,\phi, \Delta}\ \exists\text{-R}$$

where y is not free in Γ, Δ.

B. Mixed-mode Inference Rules

From [18]

DEFINITION 1. *Let $\mathcal{P}$ be a multiset of definite formulae and let $x, y \in$* free$(\mathcal{P})$*. The variables x and y are* connected *in $\mathcal{P}$ if $\exists F \in \mathcal{P}$ such that $x, y \in$* free(F)*, or $\exists z, F$ such that $x, z \in$* free(F) *and z and y are connected in $\mathcal{P}$.*

A free variable x in free$(\mathcal{P})$ is connected to a formula $F \in \mathcal{P}$ if x is connected to a free variable in F.

A formula $\forall x_1 \ldots \forall x_n F_1 \otimes F_k$ is tightly quantified *if x_i is connected to each F_j, $1 \leq j \leq k$ and each F_j is tightly quantified.*

DEFINITION 2. *Let* $\{\mathcal{P}_1, \ldots \mathcal{P}_n\}$ *be a maximal partition of a multiset of definite formulae* $\mathcal{P}$ *such that* $\text{free}(P_i) \cap \text{free}(\mathcal{P}_j) = \emptyset$ *for* $i \neq j$.

We define $\Diamond\mathcal{P}$ *as* $\bigcup_{i=1}^{n} \forall(\otimes\mathcal{P}_i)$. *We define* $\spadesuit\mathcal{P}$ *as* $\bigotimes_{i=1}^{n} \forall(\otimes\mathcal{P}_i)$.

$$\frac{}{\mathcal{P} \rightsquigarrow \mathcal{P}}\ \text{Axiom} \rightsquigarrow \qquad \frac{}{A \vdash A}\ \text{Axiom} \vdash$$

$$\frac{\mathcal{P} \rightsquigarrow \mathcal{P}'}{\mathcal{P}, !D \rightsquigarrow \mathcal{P}'}\ W \rightsquigarrow \qquad \frac{\mathcal{P}, !D, !D \rightsquigarrow \mathcal{P}'}{\mathcal{P}, !D \rightsquigarrow \mathcal{P}'}\ C \rightsquigarrow$$

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

$$\frac{\mathcal{P} \rightsquigarrow \mathcal{P}'}{\mathcal{P}, \mathcal{P}'' \rightsquigarrow \mathcal{P}', \mathcal{P}''}\ Weak \qquad \frac{\mathcal{P}_1 \rightsquigarrow \mathcal{P}' \quad \Diamond\mathcal{P}' \vdash G}{\mathcal{P}_1, \mathcal{P}_2, G \multimap D \rightsquigarrow \mathcal{P}_2, D}\ \multimap\!\rightsquigarrow$$

$$\frac{\mathcal{P}, D_i \rightsquigarrow \mathcal{P}'}{\mathcal{P}, D_1 \,\&\, D_2 \rightsquigarrow \mathcal{P}'}\ \& \rightsquigarrow \qquad \frac{\mathcal{P}, D_1, D_2 \rightsquigarrow \mathcal{P}'}{\mathcal{P}, D_1 \otimes D_2 \rightsquigarrow \mathcal{P}'}\ \otimes \rightsquigarrow$$

$$\frac{\mathcal{P}, D \rightsquigarrow \mathcal{P}'}{\mathcal{P}, !D \rightsquigarrow \mathcal{P}'}\ ! \rightsquigarrow \qquad \frac{\mathcal{P} \rightsquigarrow \mathcal{P}'}{\mathcal{P}, \mathbf{1} \rightsquigarrow \mathcal{P}'}\ \mathbf{1} \rightsquigarrow \qquad \frac{\mathcal{P}, D[t/x] \rightsquigarrow \mathcal{P}'}{\mathcal{P}, \forall x D \rightsquigarrow \mathcal{P}'}\ \forall \rightsquigarrow$$

$$\frac{\mathcal{P} \rightsquigarrow \mathcal{P}_1 \quad \ldots \quad \mathcal{P} \rightsquigarrow \mathcal{P}_n}{\mathcal{P} \rightsquigarrow (\spadesuit\mathcal{P}_1) \,\&\, \ldots \,\&\, (\spadesuit\mathcal{P}_n)}\ Collect \qquad \frac{!\mathcal{P} \rightsquigarrow \mathcal{P}'}{!\mathcal{P} \rightsquigarrow !\Diamond\mathcal{P}'}\ !M$$

$$\frac{}{\vdash \mathbf{1}}\ 1 \vdash$$

$$\frac{\mathcal{P} \vdash G_1 \quad \mathcal{P}' \vdash G_2}{\mathcal{P}, \mathcal{P}' \vdash G_1 \otimes G_2}\ \otimes \vdash \qquad \frac{\mathcal{P} \vdash G_1 \quad \mathcal{P} \vdash G_2}{\mathcal{P} \vdash G_1 \,\&\, G_2}\ \& \vdash \qquad \frac{\mathcal{P} \vdash G_i}{\mathcal{P} \vdash G_1 \oplus G_2}\ \oplus \vdash$$

$$\frac{!\mathcal{P} \vdash G}{!\mathcal{P} \vdash !G}\ ! \vdash \qquad \frac{\mathcal{P} \vdash G[y/x]}{\mathcal{P} \vdash \forall x.G}\ \forall \vdash \qquad \frac{\mathcal{P} \vdash G[t/x]}{\mathcal{P} \vdash \exists x.G}\ \exists \vdash$$

The rule $\forall \vdash$ has the usual restriction that y is not free in $\mathcal{P}$ or G.

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. Abdullah-Al Amin, *Agent-Oriented Programming in Linear Logic*, Honours Thesis, Department of Computer Science, RMIT, November, 1999.
4. J.-M. Andreoli, Logic Programming with Focusing Proofs in Linear Logic, *Journal of Logic and Computation* 2:3, 1992.
5. J.-M. Andreoli, Focussing and Proof Construction, *Annals of Pure and Applied Logic* 107:1:131-153, 2001.
6. 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.
7. A. Avron. *The Method of Hypersequents in the Proof Theory of Propositional Non-Classical Logics* In "Logic: Foundations to Applications" (edited by W. Hodges, M. Hyland, C. Steinhorn and J Truss), Oxford Science Publications, 1-32, 1996
8. Michael E. Bratman, *Intentions, Plans, and Practical Reason*, Harvard University Press, Cambridge, MA, 1987.
9. *The dMARS V1.6.11 System Overview*, Technical Report, Australian Artificial Intelligence Institute (AAII), 1996.
10. 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.
11. R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to theorem proving in problem solving. *Artificial Intelligence*, 2:189–208, 1971.
12. 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.
13. Dov Gabbay, *Dynamics of Practical Reasoning: A Position Paper*, in K. Segerberg, M. Zakharyaschev, M. de Rijke, and H. Wansing, editors, *Advances in Modal Logic*, 2:197-242, CSLI Publications, 2000.
14. 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.
15. J-Y. Girard, Linear Logic, *Theoretical Computer Science* 50, 1-102, 1987.
16. J-Y. Girard, Y. L and L. Regnier, *Advances in Linear Logic*, London Mathematical Society Lecture Note Series 222, Cambridge University Press, 1995.

17. 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.
18. 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.
19. 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.
20. J. Hodas and D. Miller, Logic Programming in a Fragment of Intuitionistic Linear Logic, *Information and Computation* 110:2:327-365, 1994.
21. 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.
22. 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.
23. Nick Jennings, An agent-based approach for building complex software systems, *Communications of the ACM* 44:4:35-41, 2001.
24. 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.
25. A. Kakas, R. Kowalsk and F. Toni, Abductive Logic Programming, *Journal of Logic and Computation* 2:719-770, 1992.
26. R. Kowalski, Predicate Logic as a Programming Language, *Information Processing 74*, North-Holland, Amsterdam, 1974.
27. R. Kowalski and F. Sadri, From Logic Programming towards Multi-agent Systems, *Annals of Mathematics and Artificial Intelligence*, 1999.
28. Patrick D. Lincoln. *Computational Aspects of Linear Logic.* PhD thesis, Stanford University, August 1992.
29. M. Masseron and C. Tollu and J. Vauzeilles, Generating Plans in Linear Logic I: Actions as proofs, *Theoretical Computer Science 113:349-371*, 1993.
30. M. Masseron, Generating Plans in Linear Logic II: A geometry of conjunctive actions, *Theoretical Computer Science 113:371-375*, 1993.
31. D.A. Miller, A Multiple-Conclusioned Meta-Logic, Proceedings of the Symposium on Logic in Computer Science 272-281, Paris, June, 1994.
32. 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.
33. L. Padgham and P. Lambrix, Agent Capabilities: Extending BDI Theory, in *Proceedings of Seventeenth National Conference on Artificial Intelligence - AAAI 2000* 68-73, August, 2000.
34. G. Plotkin. Structural Operational Semantics (lecture notes). Technical Report DAIMI FN-19, Aarhus University, 1981 (reprinted 1991).
35. Jeff Polakow, *Ordered Linear Logic and Applications*, Ph.D. thesis, Department of Computer Science, Carnegie Mellon University, August 2001. Available as Technical Report CMU-CS-01-152 and from `http://www.cs.cmu.edu/~jpolakow`.

36. Jeff Polakow, Linear Logic Programming with an Ordered Context, *2nd International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, Montreal, September 2000.
37. D. Pym, On Bunched Predicate Logic, *Proceedings of the IEEE Symposium on Logic in Computer Science*, Trento, July, 1999.
38. D. Pym and J. Harland, A Uniform Proof-Theoretic Investigation of Linear Logic Programming, *Journal of Logic and Computation* 4:2, April, 1994.
39. 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.
40. 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.
41. Anand Rao and Michael Georgeff, Decision Procedures for BDI Logics, Journal of Logic and Computation, 8(3):292-342, 1998.
42. A. Scedrov, A Brief Guide to Linear Logic, in *Current Trends in Theoretical Computer Science*, G. Rozenberg and A. Salolmaa (eds.), World Scientific, 1993.
43. 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/`.
44. Vaghani, J., K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask and J. Harland. *The Aditi Deductive Database System*, VLDB Journal 3:2:245-288, April, 1994.
45. Michael Winikoff and James Harland, Some Applications of the Linear Logic Programming Language Lygon, *Proceedings of the Australasian Computer Science Conference* 262-271, Melbourne, February, 1996.
46. 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.
47. 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.
48. Michael Wooldridge, *Reasoning about Rational Agents*, MIT Press, 2000.
49. Michael Wooldridge and Nick Jennings, *Agent Theories, Architectures and Languages: A Survey*, in Michael Wooldridge and Nick Jennings (eds.), *Intelligent Agents* 1-22, Springer, Berlin, 1995.