Forward-Chaining Rules in Agent Systems

Luke Trodd

Supervisor: James Harland

Honours Thesis

School of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA

October, 2010

Abstract

Agent systems can be divided into two broad categories of deliberative and reactive architectures. Deliberative architectures tend to excel within closed world, non real-time scenarios, offering proactive agent planning capabilities for achieving goal oriented behaviours. Reactive architectures offer timely, situation based behaviours that are suitable for application to real-time scenarios. An alternative approach seeks to combine the benefits of both architectures into a hybrid agent architecture, offering both goal oriented proactive behaviours, as well as situation based reactive behaviours. In this paper we demonstrate the implementation of such a hybrid agent architecture within a framework of the linear logic programming language Lygon. We demonstrate how backward and forward chaining techniques can be practically integrated within this framework to provide proactive and reactive behaviours within a BDI agent paradigm. Various Lygon syntactic extensions are discussed, enabling the convenient specification of agent structures such as actions, plans and events. Motivations and methods for representing sequentiality within a linear logic are examined, and appropriate logic programming connectives are introduced to the Lygon syntax. We demonstrate a practical conjunctive action system that enables intuitive specification of agent actions and their effects on state within the environment. We demonstrate how agent plans can be constructed both explicitly and implicitly with heuristics search specifications. Event based reactive architectural features are specified and their significance within a novel, hybrid BDI agent cycle are demonstrated. Maintenance goals and their syntactic specification are discussed in reactive and proactive forms, and the implementation of a novel proactive constraint enforcement mechanism are presented. Finally a number of applications of the implemented platform are presented, and various software design issues are discussed.

Acknowledgements

I would like to offer a huge thanks to my supervisor James Harland for all his support, understanding and input into my research and wider studies. He has been a great encouragement throughout the year, especially in the midst of difficult circumstances in my life. I would also like to thank my beautiful wife Wendy, whose love and support have kept me going. Without her this work would not have been possible. Finally I would like to thank all of my family and friends for their encouragement and guidance in life and career. Thank you all!
## Contents

1 Introduction 3

2 Background 4
   2.1 Agent Oriented Programming 4
   2.2 Linear Logic 7
   2.3 Lygon 9
   2.4 Forward and Backward Chaining 11
   2.5 Abduction and mixed mode computation 12

3 Agent Programming in Lygon 13
   3.1 Sequentiality in Agent planning 13
   3.2 Programming for Agents 16
      3.2.1 Dealing with Facts 16
      3.2.2 Representing Agent Beliefs 17
      3.2.3 Actions 18
      3.2.4 Plans 19
   3.3 From plans to a plan of action 22
   3.4 Implementing reactive behaviour 24
   3.5 The BDI cycle 25
   3.6 Maintenance goals 27
   3.7 Implementation 30

4 Experiments and Evaluation 30
   4.1 Robotic cleaner program 30
   4.2 Reactive Behaviour 34
   4.3 Gold mining program 36
   4.4 Reflection 38

5 Conclusions and Future Work 38
1 Introduction

Agent oriented programming has traditionally been approached using deliberative, symbolic representations of problem domains. In this paradigm, software reasons or 'thinks' about a situation in order to make appropriate decisions that achieve desired outcomes. The deliberative method has proven useful in a variety of closed world applications, and where thoughtful considered action is valued above responsiveness. Over time, the deliberative paradigm has proven to be deficient for many applications. One limitation that presents in many real-time applications is the difficulty in providing guarantees of timeliness for agent interactions. In practical applications the intractable nature of first order logic ensures that strong guarantees cannot be given regarding the amount of resources used in the reasoning process (35). Additionally it becomes apparent that modelling useful and complete symbolic representations of real world problems tends to be non-trivial (21; 35), complicated further by the open world nature of many problems. In response to these deficiencies, much research has focussed on reactive agent architectures. A reactive architecture models the world as a series of responses to environmental conditions, viewing intelligence as a product of many simpler interactions within that environment (21). Although the reactive architecture solves some of the problems associated with deliberative architectures, it is lacking in a number of key aspects. In reactive architectures it is difficult to see how a system can take long term view of the world, it is non-trivial to engineer systems that complete specific tasks and there is little in the way of software development methodologies for such systems. An alternative approach that attempts to harness that advantage of both paradigms is with a hybrid architecture (8; 21). Hybrid agent architectures combine deliberative and reactive components to offer both a long term proactive approach to problem solving, as well as reactive behaviours useful in real-time contexts.

In (8) Harland and Winikoff propose a hybrid agent architecture built upon a linear logic framework. Linear logic has the potential to offer many advantages in the agent context over classical and intuitive logics due to its resource oriented nature. Linear logic is able to specify actions cleanly and intuitively, can effectively express resource oriented problems and has a native notion of concurrency appropriate for agent architectures. In this paper we describe a practical implementation of a hybrid agent architecture, constructed upon a logic programming language called Lygon. Lygon is a declarative logic programming language designed by Michael Winikoff and is based upon a fragment of linear logic (23). Lygon operates on the principal of backward chaining — a method for determining proofs of a logical formulae by recursively decomposing them into atomic formulae. We show that a hybrid agent architecture can be constructed practically by the introduction of a forward-chaining as a complement to the backward-chaining approach. Forward-chaining maps from states in the world to actions or new states in the world. It has a natural resemblance to the reactive behaviour specifications of a reactive agent architecture, complementing the deliberative behaviours associated with backward-chaining. We demonstrate a variety of enhancements that extend upon the existing Lygon architecture, facilitating agent oriented programming in a hybrid architecture. We introduce the concept of sequential operators in Lygon and its applications to planning. We describe the implementation of a conjunctive action system that intuitively expresses agent interactions with its environment. We specify methods for constructing agent plans in explicit and implicit syntax, and demonstrate applications within practical agent problems. We define event based syntax for specifying reactive behaviours in a practical agent architecture. We describe a novel BDI agent cycle that accommodates reactive and deliberative behaviours within a linear logic framework. Finally we describe methods for defining maintenance goals, demonstrating means for specifying reactive constraints, and introduce a novel proactive maintenance goal mechanism that enforces arbitrary constraints during planning.

This paper is organised as follows: In section 2 we introduce various background topics including Agent Oriented Programming and BDI, Linear Logic, Lygon, forward and backward chaining, abduction and mixed mode computation. In section 3 we specify implementation work that we have carried out, covering the specification of sequential operators in Lygon and linear logic, mechanisms for programming agents and specifying plans, using abduction to generate action sequences, the implementation of reactive behaviours in Lygon, specifications of a BDI cycle incorporating reactive and deliberative components and maintenance goals in both reactive and proactive forms. In section 4 we specify a number of applied problems using the implemented agent architecture and evaluate various issues and concerns relating to their specification, identify limitations in the implemented architecture and demonstrate the expressive and intuitive nature of the implemented extensions to Lygon. Finally, in section 5 we conclude with final
2 Background

2.1 Agent Oriented Programming

Agent oriented programming was proposed by Yoav Shoham to be a 'new programming paradigm based on a societal view of computation' (32). The agent paradigm takes inspiration from practical reasoning and human psychology, adapting concepts such as mental states as abstractions for computer systems. Although there is no agreed consensus on the definition of an agent, they tend to have a number of identifiable characteristics (16; 35; 8). Consider the example of a cleaning robot implemented within the agent paradigm. The agent is likely to have the following characteristics:

- **Situated** - an agent is embedded within its environment, being influenced by that environment and having some control over it. For example, our cleaning robot is situated within the environment of the house it is designated to clean. Its behaviour is influenced by the detection of dirty and clean rooms and its interactions with the environment cause room in the house to change from dirty to clean.

- **Autonomous** - an agent controls its state and behaviour in such a way as to achieve certain design objectives, taking an active role in the environment. A cleaning agent does not need to be told how to achieve cleanliness of a house by specifying a series of servo movements, but can automatically determine how to achieve this goal by performing the appropriate actions.

- **Flexible** - an agent can adapt to varying situations, demonstrating problem solving behaviour. A cleaning robot might be able to determine which rooms require cleaning at a point in time and react appropriately to clean them.

- **Proactive** - an agent seeks to achieve certain aims and will persist to achieve them. Our cleaning agent would be able to determine a series of actions that it will carry out to clean the house.

- **Reactive** - an agent responds to the environment in which it is situated. A cleaning robot might be able to react to emergency events such as the detection of a fire in a timely fashion.

From a design perspective, Agent Programming may be seen as a specialisation of the Object Oriented Paradigm (32), indeed it is sometimes described as a successor to OOP (16). A number of similarities between the two paradigms are apparent: both an object and an agent encapsulate state and behaviour (variables and methods), and both communicate using message passing. Indeed, agent systems can be implemented as extensions to OOP languages (for example JACK) (25). However, object oriented and agent oriented programming represent two different paradigms suited to different problem domains. An agent cannot be viewed purely as an object. Objects do not possess many of the characteristics we have identified for example they are not autonomous, have no notion of proactive behaviour and do not react autonomously to their environment. Additionally, unlike objects, agents are inherently multi-threaded. This places the two approaches in fundamentally different programming paradigms despite their similarities.

Agent architectures can be identified by two broad classifications - deliberative architectures which make decisions through reasoning and symbolic representation, and reactive architectures which view intelligence as a product of interactions with the environment (3; 35). Additionally, hybrid architectures attempt to combine the properties and benefits of both approaches.

Traditionally a deliberative, or 'rational' agent architecture attempts to explicitly represent a model of the world in which it operates. Such an agent focusses on 'thinking' about its environment rather than acting on stimuli from that environment. Decision making may occur by interacting with this model through pattern matching, symbolic manipulation and logical reasoning. At its extreme, a deliberative agent architecture may be implemented as a theorem prover, a notion which experience has proven to be impractical in real world systems, for reasons such as the intractable nature of first order logic (35; 21). The internal state of a deliberative system will often be in the form of a database of formula, from which
deductions can be derived (35). Such an approach is effective for making decisions where all necessary facts are known at the time - effective in closed world contexts, but one that does not scale to many real world problems (which tend to open in nature). An obstacle often encountered in the design of such systems is how to translate the world into an accurate and adequate symbolic description that represents all the complexities of the entities and processes they model (21). Additionally, it tends to be difficult to provide any guarantees of timeliness for decision-making, since deductions may be intractable. Such shortfalls make the architecture, in its purist form, an approach only suited to a limited domain of problems which tend to be closed, and where rapid timely response is not a requirement.

A reactive architecture falls at the opposite end of the scale. It does not explicitly represent the world, nor does it perform any complex symbolic reasoning about the world. In this paradigm, the intelligence of an agent is considered to be a by-product of many simpler behaviours and the agents interaction with the world as it responds to perceptions (21; 35; 13). Brook’s (3; 4) notes that intelligent behaviour can be generated without any explicit symbolic representation of the world, and that intelligence can be viewed as ‘emergent’ in some complex systems. In a subsumption architecture, reactive behaviour is modelling through direct mappings from situations to actions or plans dealing with that particular context (35). As a logical implication this can be characterised as:

\[
\text{situation} \rightarrow \text{action}
\]

The situation to which we refer typically takes the form of sensor input and tends to have minimal processing applied. For example, we may have a fire alarm which detects the presence of heat and smoke, then takes some action such as activating an alarm. Such a behaviour could be represented like so:

\[
\text{smoke AND heat} \rightarrow \text{activate-alarm}
\]

It is an important consideration to realise that more than one behavioural rule may be activated at any one time. It will often be the case that we may wish to model a variety of behaviours that apply to a situation, but where only one can reasonably be applied in any single circumstance. For this reason there has to be some mechanism for deciding between multiple rules. One approach may be to prioritise the various rules and take the highest priority rule that applies to the situation.

The reactive architectural approach proves to have a number of advantages over deliberative architectures (35; 21). It is simple to understand and to implement, it is robust, elegant and importantly it enables the construction of systems that can reliably react in a timely fashion to external stimuli. Its shortfalls however are notable. It is apparent that in a reactive system, decisions must be based on local information only (meaning that we must have enough information available locally to make good decisions). Due to its direct situation \(\rightarrow\) action mapping, a reactive system tends to take a ‘short term view’ of the world (35). For example, it will likely prove difficult to define an action based on something that is likely to happen in the future. In addition, it is difficult to see how we would define long term behaviours in a practical and understandable way, how one would effectively engineer a reactive system that is able to perform specific tasks, nor how one could apply software engineering methodologies for complex reactive systems.

A third approach to agent architectural design is to take a hybrid approach - attempting to take the best of both the deliberative and reactive architectures and combining into a unified architecture (21; 8). It is clear that there are many situations to which the pure forms of deliberative and reactive architectures are not enough. A real world system may have requirements that include complex planning for future events, as well as timely, reactive behaviour in certain situations. A hybrid architecture may integrate symbolic representation of the world and logical reasoning capabilities to form plans and make decisions, whilst applying a reactive approach that observes the environment and provides rapid response to situations with timeliness requirements. Practical implementations of such systems include the Procedural Reasoning System (PRS), TOURINGMACHINES, COSY and INTERRAP (36). The hybrid approach to intelligence is one that is more in line with the characteristics we have identified for agents. It offers to provide both the proactive (deliberative) and the reactive behaviours often identified with intelligent agents.

The Agent Programming paradigm offers a promising framework for defining computer systems in terms that humans can relate to, but to be of practical use it must offer tangible benefits to real-world systems. Since its conception, the agent paradigm has been applied to Space Probes and Space Shuttle maintenance by NASA (29; 35), search, air traffic control, business processes (17) and numerous other applications. One of the great strengths of the paradigm is its ability to manage complexity inherent in large concurrent systems, by decomposing them into manageable pieces and providing understandable abstractions (16). As a rule, complex systems tend to be difficult and expensive to build, verify and maintain.
Concurrency, hierarchy and interaction can be highly difficult to define in more traditional paradigms such as functional and OOP. AOP has much to offer through decomposition, abstraction and organisation. Decompositions and explicit relation modelling offer effective methods for partitioning of problems into manageable chunks. The abstractions of AOP are familiar and intuitive for humans, as they resemble abstractions that we ourselves use in day to day practical reasoning. Further, AOP provides organisational properties that allow us to group components together by a variety of properties such as role, behaviour, locality and control.

One incarnation of the Agent Programming paradigm is the BDI model of agents. A BDI system consists of three important abstractions:

- **Beliefs** – These describe an agent’s current knowledge about the world.
- **Desires** – States of the world that the agent would like to achieve.
- **Intentions** – Means that an agent has chosen to achieve its current desires.

BDI theory has its roots in practical reasoning that humans use from day to day. It involves aspects of deliberation - deciding what goals the agent would like to achieve, as well as ‘means-end’ reasoning - deciding how the agent is going to achieve those goals. BDI is a popular foundation for designing intelligent agents due to the intuitive nature of its human-like abstractions, as well as its clear decomposition into practical systems.

Beliefs in a BDI system consist of observations that have been made by the agent and deductions it has made about the environment. In this respect it is important to note the difference between an agent’s beliefs about the world, and the facts in that world. It is possible for an agent to believe something that is not true in the environment it is operating within. This is characteristic of its human-like ‘mental state’ and suited to the open-world semantics of real systems where not all information is readily available to an agent, and is potentially corrupted. Beliefs are also incomplete, in the sense that facts can be unknown. For example, an agent might not know whether a door is open or closed (unknown), or it may have outdated beliefs that the door is currently closed based on an old observation. Due to these properties and since a BDI agent interacts with the world based on its current (incorrect or incomplete) beliefs, plans may therefore fail.

Desires are often confused with goals in BDI systems. A desire represents things that the agent wants to achieve - often desires can be mutually exclusive or contradictory. For example a youth who has just finished school may desire to both become an academic, as well as work full time in industry. Clearly it is not likely that both can be achieved simultaneously so the agent (our youth) must make a decision between the two. Such decisions form goals - a consistent set of desires that the agent can work towards.

Intentions play a significant role in practical reasoning as used by humans. Intentions tend to constrain future reasoning by limiting options available to the agent through cause and effect - the agent cannot entertain inconsistent options. Thus intentions must be a consistent set of goals. Intentions tend to persist, in the sense that an agent will commit to them and continue to try to achieve them over time. Such commitment must be limited, as there can be circumstances where an agent’s intentions are no longer achievable due to some change in the environment or failure of some plan.

A BDI system tends to consist of a common set of components in one form or another as defined by:

- **Set of beliefs** – what the agent believes.
- **Belief revision function** – updates the current beliefs based on new knowledge.
- **Option generation function** – constructs the agents desires based on current beliefs and intentions.
- **Set of options** – a set of possible courses of action that can be selected to achieve desires.
- **Filter function** – constructs the agents intentions based on current beliefs, desires and intentions.
- **Action selection function** – determines an action to perform from the agents current intentions.
2.2 Linear Logic

Linear Logic was devised by Girard (15) in 1987. In contrast to classical logic, linear logic is often described as being 'resource sensitive', in its ability to control the duplication of clauses and facts. A defining difference between linear and classical logic can be understood in its rejection of weakening and contraction rules, which are fundamental to classical and intuitional logics.

For example the classical logic statement:

\[
\text{If the grass is wet, THEN it rained last night}
\]

can be shown through the weakening rule to be exactly equivalent to the following statements:

\[
\text{If the grass is wet AND the grass is wet THEN it rained last night}
\]
\[
\text{If the grass is wet AND the grass is wet AND \ldots THEN it rained last night}
\]

In this sense one can understand classical logic as dealing with statements of truth. A statement that is true once is true any number of times and thus can be duplicated arbitrarily. Likewise the first statement can be derived from the second using the rule of contraction, enabling the destruction repeated statements. In practice, logic systems based on classical logic can lead to uncontrolled duplication of clauses in the process of a proof. Additionally the semantics of the logic do not fit cleanly with many problem domains, requiring special mechanisms to correct for logical misfits (22). For example if we wish to represent the Blocks World scenario using classical logic, we may represent two blocks with the clause A AND A. The difficulty with representing the world in this way is the application of the contraction rule which says A AND A \rightarrow A. In practice, applying contraction in this case will cause the robot to forget the existence of one of the blocks, clearly not an acceptable outcome.

Likewise the application of the weakening rule can lead to problems, demonstrated with the valid statement A \rightarrow A AND A. Weakening can be applied any number of times, possibly leading to a circumstance where the robot believes that it has many (possible infinite) hands that are holding block A! Clearly the semantics of our representation do not fit the scenario we intend.

One way around the aforementioned problems is by correcting flows in the logic to maintain the semantics of the problem domain (22). For example we may restrict the use of tautologies like those presented above, in practice meaning we would prevent the use of the weakening and contraction rules in certain circumstances. Another approach might be to introduce a new predicate or temporal parameter that identifies the situation in which a rule applies. The problem with the former solution lies in the fact that it would likely be very difficult to characterise in formal logic, and can arguably be seen as a 'band-aid’ solution to an inherently inappropriate model. The latter solution tends to lack flexibility and may lead to overly complex and a difficult to maintain model.

Linear logic offers an elegant and formal solution to the above problems through its property of ‘maintenance of space’ (15). By restricting weakening and contraction rules and defining a set of new operators it offers resource oriented reasoning applicable to many problem domains. In addition, linear logic is strong, in the sense that it can fully accommodate other logics (including classical logic) through the use of special operators called exponentials. The logic offers both cumulative and non-cumulative conjunctions that allow us to deal with resource issues or to overlook them respectively. It also offers two dual forms of disjunction, the first resembling the behaviour of classical disjunction and the second allowing a formal notion of concurrency. The operators are summarised below:

<table>
<thead>
<tr>
<th>Connective</th>
<th>Multiplicative</th>
<th>Additive</th>
<th>Exponential</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conjunction</td>
<td>( \odot ) times/cross</td>
<td>&amp; with</td>
<td>! ofCourse</td>
</tr>
<tr>
<td>Disjunction</td>
<td>( \otimes ) par</td>
<td>( \oplus ) plus</td>
<td>? whyNot</td>
</tr>
<tr>
<td>Implication</td>
<td>( \rightarrow ) lollipop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The cross operator \( \odot \) can intuitively be thought of as combining two resources together. As an example, consider a situation in which we visit a restaurant to order a meal. We might represent a combo meal in the form Burger \( \odot \) Fries \( \odot \) Coke. The cross operator resembles the classical AND operator with one major difference - it does not obey the weakening/contraction rules of the latter. This illustrates linear logics
'resource sensitive' nature clauses cannot be duplicated or deleted whilst maintaining equivalence. For example $\text{Burger} \neq \text{Burger} \otimes \text{Burger}$

The with operator $\&$ allows us to duplicate resources in a controlled way. Intuitively we can think of it as a choice that we ourselves can make. For example, our meal choice might consist of $\text{Burger} \otimes (\text{Fries} \oplus \text{Nuggets}) \otimes (\text{Coke} \oplus \text{Sprite})$. In this context we have a choice between either a Coke or a Sprite with our meal. The choice is arbitrary and we can be sure that both are valid choices.

The plus operator $\oplus$ resembles the classic OR operator, encoding a choice but one which is not ours to make. For example, depending on what the restaurant has available our meal may consist of $\text{Burger} \otimes (\text{Fries} \oplus \text{Nuggets}) \otimes (\text{Coke} \oplus \text{Sprite})$. In this circumstance our meal will come with either fries or nuggets, but we do not get to make the choice as to which we receive. Intuitively we can think of the operator as designating somebody else’s choice (in this case the restaurant).

The par operator $\oslash$ is the dual of the $\otimes$ operator, but has less intuitive definition. In practice it defines a concurrent operation whereby the context can be split between both sides of the operator, allowing us to share resources. Par may be used it to accumulate a supply (debt) of resources in the same way we accumulate resources with the cross operator. For example when we order a meal, the perspective of the restaurant can be represented as $\text{Burger} \oslash (\text{Fries} \oplus \text{Nuggets}) \oslash (\text{Coke} \oplus \text{Sprite})$

The negation ($\perp$) operator is used to represent supply or debt and is discussed later. Note that in this example the $\&$ and $\oplus$ operators are switched from the restaurant’s perspective, illustrating the fact that the choice between resources changes according to perspective.

The lolli operator $\multimap$ is the linear version of classical implication. Implication in linear logic has different semantics to its classical counterpart. In classical logic by the law of modus ponens, the implication $A, A \rightarrow B$ tells us that both $A$ and $B$ are true, i.e. $A$ AND $B$. The equivalent modus ponens law in linear logic no longer preserves equivalence (9). So for example, the linear implication $A, A \rightarrow B$ tells us that given $A$ we can obtain $B$, but that we must ’consume’ $A$ to get it. In this respect, linear implication can be considered as more like an exchange or a committed choice than a mapping of truth. Returning to our restaurant example, we clearly need to pay for our meal which can be represented as $\text{Dollar} \otimes \text{Dollar} \multimap \text{Burger} \otimes (\text{Fries} \oplus \text{Nuggets}) \otimes (\text{Coke} \oplus \text{Sprite})$. Here we have stated that by consuming/exchanging two dollars we can get the combo meal specified. This clearly illustrates the resource oriented nature of the logic.

Negation
Like classical logic, linear logic allows for negation of formula, represented as $F \perp$. Negation can be thought of as changing a consumption of some resource into a supply of that resource, or a debt that must be paid. As illustrated earlier, negations can be accumulated through the use of the $\oslash$ connective in the same way that resources can be accumulated with the $\otimes$ connective.

Exponentials
The exponential operators offer a means for recovering classical logic behaviour within linear logic. The $!$ (of course) operator specifies that a formulae should behave classically, in other words it should be treated as an unlimited resource that can be used any number of times. For example if our restaurant has a bottomless drink policy we can specify this with the formulae $\text{Dollar} \otimes \text{Dollar} \multimap \text{Burger} \otimes (\text{Fries} \oplus \text{Nuggets}) \otimes !\text{Drink}$. This behaviour is characteristic of classical logic where formulae can be arbitrarily duplicated as necessary. The dual of the $!$ operator is the $?$ (why not) operator which represents the possibility for infinite resources. For example we can specify the consumption of any number (including 0) burgers with the formula $?\text{Burger}$.

Units
Each of the four connectives as an associated unit, as outlined in the following table:
Linear logic has been successfully applied to a variety of problems that are otherwise difficult to represent in classical and intuitional logics. For example, linear logic has been applied to various state transition problems, graph problems, database updates, planning, concurrent processing, etc (19; 11; 18). Applied to agents, linear logic offers a natural and elegant solution for many problems due to its resource sensitive nature, its well understood concurrent properties, its ability to represent knowledge and its ability to model state transitions. Linear logic provides a natural notion of actions (difficult in traditional approaches) (8), in the sense that an action can be considered a state transition from a set of states to another set of states. The resource exchange semantics of linear implication allow us to cleanly specify the change of one state to another as a result of applying actions. Additionally linear logics concurrency properties seem well suited to the Agent paradigm, which is inherently concurrent.

### 2.3 Lygon

Lygon is a declarative logic programming language created by Michael Winikoff that is based on Linear Logic. Lygon attempts to follow the mantra Lygon=Prolog+Linear Logic (1), in the sense that it is a strict extension of pure Prolog that incorporates Linear Logic reasoning capabilities. Like its linear semantics, Lygon allows resources to be used only once during a computation. This makes it an excellent tool for problem domains which are resource oriented. Lygon also implements concurrent behaviour that is inherent in the par (⊗) operator, enabling the specification of asynchronous behaviour that is characteristic of Agent Oriented systems. Like traditional logic programming languages such as Prolog, Lygon interprets programs consisting of a sequence of Horn clauses of the form D ← G.

The difference between Lygon clauses and those of classical logic languages such as Prolog, is their interpretation under linear implication rather than traditional implication. In this sense we can consider D to be resources that must be present for a clause to be applicable, and G to be resources that will replace D when the clause is evaluated. A clause in Lygon is actually non-linear by default, meaning that it can be reused any number of times. These semantics are in line with the way most users will specify a program, but can be changed by prefixing a rule with the modifier ‘linear’. The advantage of Lygons linear properties come into their own with the introduction of the various linear connectives associated with linear logic. The mapping of these connectives to ASCII characters is outlined below:

<table>
<thead>
<tr>
<th>Connective</th>
<th>ASCII</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊗</td>
<td>* &amp;</td>
<td>Provable only in empty context</td>
</tr>
<tr>
<td>⊸</td>
<td># &lt;</td>
<td>Cannot be proved, but can be weakened away</td>
</tr>
<tr>
<td>1</td>
<td>&amp; &amp;</td>
<td>Not provable</td>
</tr>
</tbody>
</table>

As a language, Lygon allows the programmer to specify programs at a very high, declarative level. (23). This has the advantage of providing an expressive yet concise grammar that improves programmer efficiency and tends to reduce programmer error through simple semantics. A tradeoff for such brevity usually comes in the form of reduced efficiency of the resulting program. As a rule however, programming abstractions tend to rise over time and the gap between performance characteristics becomes smaller, suggesting the need to explore higher abstraction levels. As a high level abstraction itself, it seems natural to try to merge the goal oriented properties of agent programming with those of logic programming.

A Lygon program consists of a sequence of clauses of the form D ← G that conform to the general grammar outlined below (23; 1):

\[
G ::= G \rightarrow G | G \rightarrow G | G \rightarrow G | G \rightarrow G | G \rightarrow D | 1 | TOP | BOT | A | \neg A \\
D ::= [linear](A_1 \# A_2 \# ... \# A_n ← G)
\]

For example, to specify a program which exchanges two dollars for a meal we may write:

\[
meal ← dollar * dollar.
\]
To specify that buying a meal consumes two dollars and provides a burger, fries and coke we might define the clause:

\[ buymeal \leftarrow \text{dollar} \ast \text{dollar} \ast (\neg \text{burger} \# \neg \text{fries} \# \neg \text{coke}) \]

To 'run' a program we need to specify a goal and possibly supply some resources. For example to consume our two dollars and supply a meal we might specify a goal:

\[ \neg \text{dollar} \# \neg \text{dollar} \# \text{buymeal} \]

Executing the above program and goal, one would observe that the evaluation does in fact fail. The reason this occurs is because the program context must be empty at the end of evaluation for a goal to succeed, in other words all resources must be consumed. In this scenario we have generated three new facts (\neg \text{burger}, \neg \text{fries}, \neg \text{coke}) in the context which have not been consumed. This behaviour, although correct for the purposes of determining the truth of a linear clause, is lacking for circumstances where we wish to determine that state of the world after evaluating some program. This issue and its relation to agent programming will be discussed in Section 3.2.

The Lygon interpreter reduces goals using a simple cycle and heuristics designed to reduce non-determinism inherent in the logic statements (23). The cycle consists of two steps:

- Select formula to be reduced
- Reduce the selected formula

This cycle repeats recursively (where formulas can contain sub-formula that require reduction) until there are no goals left to reduce. In illustrative terms we can view this procedure as a depth-first descent into a tree where the head of a branch consists of a formula joined by a linear connective and a leaf is an atom resolved by an axiom or built in mechanism. The reduction of formula by the interpreter operates on the following rules, where C represents the linear context at the point of reduction.

\[ A \ast B: \text{Split C into two contexts C1 and C2, then look for proof of (C1,A) and (C2,B).} \]
\[ A \# B: \text{Find proof of (C,A,B). This illustrates the concurrent nature of the # operator.} \]
\[ A & B: \text{Look for proofs of both (C,A) and (C,B).} \]
\[ A @ B: \text{Look for a proof of either (C,A) or (C,B). In practice (C,B) is only evaluated if a proof of (C,A) cannot be found.} \]
\[ \text{top: The goal succeeds. This can be thought of as 'consuming' all resources available to it.} \]
\[ \text{bot: Look for a proof of (C).} \]
\[ 1: \text{The goal will succeed if the context C is empty.} \]
\[ A: \text{Resolve atom.} \]
\[ ? \neg A: \text{Adds A to the program and looks for proof of (C).} \]
\[ !A: \text{If C is empty then look for proof of (A), otherwise fail.} \]

The resolution of an atom is determined by the type of atom encountered. These may include:

- **Built in predicates**: These will only succeed in an empty context and in this way are similar to the unit 1, however they can also fail based on the outcome of the builtin operation.
- **Axiom rule**: A context of the form (A,\neg B) where A and B unify is axiomatically provable. This can be thought of as the consumption A of that same supplied resource \neg B (where A=B).
- **Program clause**: An atom may match a program clause of the form A \leftarrow B. In this case the clause is selected, variables are created for it, its head A is unified with the atom and the atom is replaced by the body B.

Lygon in its present form has been shown to be useful in the definition of agents. As demonstrated by Abdullah Amin (1), we can encode Lygon agents in the form agent (ID,State).

An agent responding to a message may be written as:
Here our agent responds to a message and evaluates its body as a plan that updates its knowledge base and potentially sends one or more messages to other agents. The agents knowledge can be encoded in the form of negations (facts) that can be supplied (as discussed previously) in the form $\neg \text{KB(List)}$...#

Facts known by an agent can be consumed in a message of the form:

$\text{agent(ID,State)} \# \text{message(ID,Args)} \leftarrow \text{KB(List)} * \ldots$

As a mechanism for defining agents, the previous examples demonstrate that Lygon is well suited to the task. The implementations presented thus far however, do not scale to large systems due to their syntactic awkwardness and lack of agent abstractions as first class citizens. Further there is no clear understanding of how we can construct plans consisting of actions that fulfil some goal, nor how such an agent could respond in a reactive way to a dynamic environment. These requirements suggest the need for enhancements to the Lygon syntax.

2.4 Forward and Backward Chaining

In logic programming, theorem provers and other Artificial Intelligence applications, the standard goal resolution technique has long been that of backward chaining (9). Given a set of formula and a goal, backward chaining seeks to prove it by starting at the goal and recursively decomposing it into subgoals, attempting to resolve them using depth-first recursion and backtracking. Take the following set of implications:

- $X \text{ flies} \rightarrow X \text{ has wings}$
- $X \text{ has a beak} \rightarrow X \text{ is a bird}$
- $X \text{ has wings} \rightarrow X \text{ is a bird}$

Given these rules we may wish to prove that $X$ is a bird, given that we know it flies. Initially given the goal $X$ is a bird, we would select the second and third rules, since we wish to prove that $X$ is a bird which is the consequent of these rules. Following the inference backwards from the consequent to the antecedent we can then determine that the rules $X$ has a beak and $X$ has wings infer the goal. Continuing by following the second rule backwards, we determine that $X$ having wings is implied by $X$ being able to fly. Therefore based on these rules we can conclude that since $X$ flies, it has wings and is therefore a bird.

A backward chaining algorithm may operate like so (8; 9):

1. Begin with goal $G$ and work recursively as follows
2. Given goal $G$, find inference rules such that $A \rightarrow G$, establishing premises which if true establish the truth of the goal, then select one of them to evaluate.
3. If $A$ is an axiom (an unconditional statement of truth), succeed.
4. If $A$ is not an axiom, recursively search for matching inference rules and repeat from step 2, failing if none are found.
5. If the rule fails, backtrack to the last point where an inference rule was selected and evaluate the next matching inference.

In a linear logic context, a backward chaining proof tells us that we can achieve some state (resources) $G$, given some initial state or resources and a set of valid exchanges. The backward chaining approach is usefully applied to many applications such as the querying of databases, solving for a set of constraints and decomposing goals and plans for agents (8). In an agent context, backward chaining may be viewed as a method to approach rational behaviour - it enables us to ask 'what-if' questions about the world that may prove useful for proactive planning approaches. Backward chaining does not however offer us reactive behaviours, offering little guarantees for timeliness and potentially expending an infinite amount of resources. To successfully implement a reactive system, we require another approach.

In contrast to backward chaining which operates by following inferences from consequent to antecedent, forward chaining operates by following antecedents to consequences as one would do naturally by deduction. For example, consider the following rules:
There is smoke → There is a fire
There is a fire → There is an emergency

Given these implications we can conclude that if we observe smoke we can conclude there is a fire and can therefore conclude there is an emergency. Unlike backward chaining, forward chaining has no notion of success or failure (13). Mappings are made from certain conditions or states to other conditions or states. A linear logic interpretation tells us that given some resources A, we can exchange them for another set of resources B – in practice this means that given resources A we can remove them from the program context and replace them with resources B. This enables us to model and reason about state changes in the world.

One useful application of forward chaining to agent systems is its natural resemblance to reactive behaviours, characteristic of a reactive agent architecture. Given some situation, we can define a rule that reactively applies some action or plan. For example we may define a reactive rule:

There is fire → Sound alarm

The advantage of using such an approach to defining agent behaviour is its simplicity and timeliness. Strong guarantees can be made as to the amount of resources that will be consumed evaluating the rule, and the amount of time the rule takes to evaluate. In certain contexts such as real-time control applications, such guarantees are necessary for a system to be feasible.

In application, a reactive system may be made up of many forward chaining rules. These rules can be seen as a collection of stimulus-response, in-out or condition-action rules (21). This approach makes an agent sensitive to changes in the environment around it, and thus ideal for dynamic real-world scenarios.

2.5 Abduction and mixed mode computation

The backward chaining approach to goal resolution we have so far examined is appropriate for asking the question ‘is this true’. In addition to this, most backward chaining logic languages support unification which allows us to ask ‘what is’ questions containing variables such as the following:

\[
\begin{align*}
\text{fun}(X) & \leftarrow \text{show}(X). \\
\text{show}(\text{circus}). \\
\text{show}(\text{movies}).
\end{align*}
\]

Given the above rules, we may ask the question ‘what is fun?’ by posing the goal fun(X). Through backward chaining and unification, the goal will be resolved by binding the variable X to either X=circus or X=movies. This method is known as unification and has many practical applications. In some circumstances however, we may wish to ask questions of the form ‘how do I make this goal true?’ Essentially we want to determine how we can make a goal which fails (is not currently true) into one that succeeds by manipulating the environment (8). A method for achieving these ends is called \textit{abduction}.

Logical deduction can be considered a projection forward from cause to effects that enable us to predict the logical outcome of a set of inferences. Abduction on the other hand allows us to project backwards from effects to causes to abduce possible explanations for our observations (31). A classic example of this is demonstrated by the following (5):

\[
\begin{align*}
\text{grass is wet} & \leftarrow \text{rained last night} \\
\text{grass is wet} & \leftarrow \text{sprinkler was on} \\
\text{shoes are wet} & \leftarrow \text{grass is wet}
\end{align*}
\]

Given the above rules, we may observe that our shoes are wet and attempt to abduce an explanation for why this is so. By recursively computing explanations for our observations we determine that our shoes may be we because it rained last night or because the sprinkler was on. An important consideration to note is that a derived explanation for an observation is a weak inference – the explanation is only one of potentially many, and should therefore be considered a probational adoption (5).

In an agent context, abduction provides us with a method for generating plans through the decomposition of goals, in other words determining a course of action to achieve a given goal. By taking a goal and
projecting from effects to causes, we can determine some set of causes which sufficiently explain the goal. The potentially many such explanations represent alternative methods for achieving the goal, enabling us to apply some selection mechanism to choose the most appropriate or least costly. An implementation of such a mechanism may operate as follows (8):

1. Use backward chaining and resolution to decompose our goal into subgoals.
2. If subgoals correspond to additional rules, decompose these from step 1.
3. Otherwise find a set of actions that achieve the goal.

In linear logic, connectives between subgoals determine their relation to one another. A disjunction of the form \( G_1 \oplus G_2 \) tells us that the success of either \( G_1 \) or \( G_2 \) is sufficient for the overall goal to succeed. This would determine that our agent can take actions to achieve either \( G_1 \) or \( G_2 \) but does not need to achieve both. A conjunction of the form \( G_1 \land G_2 \) requires that actions be taken to achieve both \( G_1 \) and \( G_2 \) for the overall goal to succeed.

In (8) a method for combining the proactive and reactive properties of the various agent architectures is proposed for application using linear logic. Backward chaining and abduction methods are proposed as a mechanism for implementing the proactive aspects of an agent – finding ways to achieve goals. Forward chaining methods are proposed as a mechanism for integrating events and percepts in order to respond reactively to changes in the environment. In traditional agent architecture approaches, rational agents focus on ‘thinking’ about explicit representations of their environment rather than reacting to dynamic events (21). In practice this approach is not appropriate for many real-world scenarios.

A straightforward way to accommodate both proactive and reactive behaviours is through the use of forward and backward chaining. Given a program \( P \) and a goal \( G \), a backward chaining approach can proactively search for a proof \( P \vdash G \) to achieve the given goal \( G \). Intuitively we can think of such an approach as the ‘thinking’ aspect of an agent, able to determine how best to achieve some goal given the circumstances it finds itself. Given a program \( P \), a forward chaining approach of the form \( P \Rightarrow P' \) can be applied to produce some state in the world \( P' \) given the present state \( P \). This approach is more characteristic of a reactive approach, allowing an agent to respond directly to circumstances in a dynamic environment.

As discussed in (21), reconciling reactive and proactive approaches in an agent context poses a number of difficulties. Central to these difficulties is the disparate nature of the representation of goals. In a rational approach, goals are represented directly and explicitly. In a reactive approach goals are implicit, consisting of stimulus-response specifications that combine to move the agent towards desired states. Reconciling these differences can be challenging and requires an understanding of the relationships between explicit goals and condition action rules in the applied dynamic context. A proactive approach allows the specification of agents that work towards specific outcomes and where there is a strict measure for success. A reactive approach accommodates circumstances in which goals are nebulous or where there are many desirable outcomes that should be accommodated.

3 Agent Programming in Lygon

A variety of extensions were implemented to the Lygon language to accommodate the proposed agent oriented architecture. These mechanisms and associated implementation issues are discussed in the following section.

3.1 Sequentiality in Agent planning

A requirement that quickly becomes apparent when designing an agent program is the need to specify sequentiality of actions and plans. For example we may wish to define a program for a cleaning robot that should first move to a room then vacuum the floor and then polish the floor. Completing these steps in any alternative would clearly be invalid (or at least undesirable) for achieving a clean room. Linear logic, and Lygons capacity for describing sequentiality may at first appear to be simply a matter of using the cross (*) operator and defining the statement \( A*B \), or the par (#) operator by defining \( A#B \). Indeed Lygon does implement the * operator in such a way that the left side will always be evaluated before the right. Part of the problem in using this approach lies in the fact that this behaviour is a consequence of the
implementation of the Lygon interpreter, rather than a property of Linear Logic. In logical terms the cross operator says nothing about sequence, but rather is a property of two resources being found together. In addition, it is a property of Linear Logic that additions to the context in A (generation of new facts) will not be visible when evaluating B (23). As a consequence, A can only consume resources and is unable to generate new state in the program (a property which is highly desirable in the context of defining agent programs).

Likewise the # operator fails to accommodate our requirements. As discussed earlier, the # operator can be viewed as an aggregation of negations (accumulation of debts) and has concurrent properties such that both A and B will be added to the linear context and evaluated in potentially arbitrary order (as demonstrated by activating fair rule selection in Lygon). With this behaviour, although changes to the context made by A will be visible to B, the 'old state' made available to A will also be available to, and potentially consumed by B (23). This is clearly undesirable behaviour for our purposes and thus an alternative approach must be found.

One potential solution to the sequentiality problem is to encode it with the 'continuation passing style idiom' as devised by Michael Winikoff (23). This method works by evaluating condition A and recursively passing sequential goals as a clause parameter, which are then evaluated using the call() function. For example we may define a program like so:

\[
\text{at}(X,C) \leftarrow \text{at}(X)\ast\text{call}(C).
\]

\[
\text{at}(X,C) \leftarrow \text{at}(Y,\text{moveto}(X)\ast\text{call}(C)).
\]

\[
\text{dirty}(X,C) \leftarrow \text{dirty}(X)\ast\text{call}(C).
\]

\[
\text{clean}(X,C) \leftarrow \text{dirty}(X, (\text{at}(X, (\text{neg at}(X) \# \text{neg clean}(X) \# \text{call}(C))))).
\]

Goals which are to be evaluated after another are passed as the last parameter to each rule. We define two at() rules, the first to consume facts in the form at(X) then evaluate the passed goal, and the second to move the robot then evaluate another goal. The clean rule nests multiple sequential goals, first evaluating dirty(X,C) rule which consumes a fact of the form dirty(X), then calling the at(X,C) rule and then generating some new facts which is passed to the next goal C. Providing all rules define the evaluation of the C parameter, goals can be nested arbitrarily to define long sequences.

A difficulty in using the continuation passing style is its potential to grow in complexity, and poor maintainability as the program evolves. Programs of any significant complexity will rapidly become unwieldy and difficult to understand as use of recursive nesting increases. In addition it is difficult to see how the use of the call() operator would be characterised in formal logic.

There are a number of options to consider when determining a solution to the sequentiality problems discussed. One may simply take the existing continuation passing style and define new syntax and rule transformations which take simple sequences in a program definition and covert them into rules of the form shown. Such a system could provide user friendly 'syntactic sugar' to the user, whilst performing complex rule transformations 'behind the scenes'. A more elegant solution would be the specification of a new operator with sequential properties that can be characterised in a formal logical framework. Such a solution was devised in the sequence (\texttt{\textgreater\textless\textgreater}) operator.

**The sequence operator \texttt{\textgreater\textless\textgreater}**

The \texttt{\textgreater\textless\textgreater} operator defines an intuitive notion of a sequence of formulae. When combined with the agent paradigm it can be used to define sequences of actions or plans whose outcomes rely on those that come before. Implicit in this definition is the need for the connective to pass state between its left and right sides the output state after achieving the left side should feed into the right side. For example, we may wish to specify that plan A be executed to accomplish some state in the world, followed by plan B to give us a final state. This can be represented as \texttt{plan A \textgreater\textless\textgreater plan B}.

To implement such an operator we can identify four requirements:

1. A must be evaluated before B (sequential property).
2. A can return a context containing any number of negations (facts), but must completely reduce all goals.
3. B should operate on the context returned by A
4. B must return an empty context.
The necessity of the first requirement is obvious, the second requirement originates from the need to apply some changes to the world that partially accomplish our goal (but complete all specified goals), and which will effect the context of B. This effect implies the need for the third requirement (operating on a partially updated context). The fourth requirement encapsulates our expectation that the clause as a whole is expected to be fully realised as a goal (all state must be consumed). This last requirement is in fact a loose one, since it may be overridden by nesting. For example consider the following program:

plan A ← plan B >> plan C.
plan D ← plan A >> plan E.

Given the previous requirements, one might imagine that evaluating plan D as a goal would require that the context be empty after the evaluation of plan A, since plan C is on the right side of a sequence. However, since plan A is nested on the left side of a sequence, requirement 2 takes precedence over requirement 4, allowing plan C to return state in its context (partially update the state of the world). This gives intuitive results - when evaluating plan D, plan A partially updates the state of the world and plan E completes the evaluation, returning an empty context. Thus a fifth requirement is identified:

5. In nested sequences, requirement 2 overrides requirement 4.

The semantics for implementing the sequential properties of the connective are straightforward, however complications arise when we consider the latter requirements. Recalling our discussion earlier, the abstract Lygon interpreter cycle operates in a two step cycle:

1. Select a formula to be reduced.
2. Reduce the selected formula.

When a formula of the form $A >> B$ is selected by the interpreter engine, it is reduced in the following fashion:

- Pass the entire context $C$ into $A$ and look for a proof of $(C,A)$, marking the proof as allowing the return of negations.
- Take the return context $C_2$ from $A$ and look for proof of $(C_2,B)$.

The marking of a proof for negation (state) returns is complicated by the need for nested sequential connectives, as we must track the depth of nesting to determine when we need to prevent the return of state. This is accomplished through the use of a depth counter. The depth counter is incremented before the evaluation of $A$ and decremented upon return. When the depth counter is greater than 0, we allow the return of contexts which contain negations. Recalling our nesting example earlier, if we were to evaluate plan D, the proof process would operate as follows:

- Reduce plan D to plan A >> plan E.
- Evaluating plan A >> plan E, increment the depth counter to 1.
- Evaluate plan A, reducing it to plan B >> plan C.
- Evaluating plan B >> plan C, increment the depth counter to 2.
- Find proof of plan B, allowing state in return context (depth counter > 0).
- Decrement depth counter to 1.
- Find proof of plan C given output context of plan B, allowing state in return context.
- Decrement depth counter to 0.
- Find proof of E given output context of plan A, requiring empty context on return.

Note that the use of a sequential depth counter enables us to override requirement 4 for nested sequential operators in a straightforward way.

Intuitively we can think of the sequential operator as allowing some goal $A$ to modify the state of the world, and then applying goal $B$ to further update the state of the world. The combination of the two sequential goals work to achieve some higher goal. This can be characterised formally as $P \vdash P' \vdash G$. 

15
In this representation, \( \rightarrow \) represents a computation changing the state of the world (actions performed by the agent). Given some initial state \( P \), our agent performs one or more actions to update that state to \( P' \), achieving the desired goal \( G \).

**The choice sequence operator \(<\to>\)**

In some cases we may wish to specify a sequence, but are not concerned with the order in which that sequence occurs. For example, we may wish to specify that our vacuum robot clean the lounge and bedroom in some sequence but we do not care which comes first. One way we could specify this using existing operators is a rule like the following:

\[
(\text{plan clean(lounge)} \gg \text{plan clean(bedroom)}) @ (\text{plan clean(bedroom)} \gg \text{plan clean(lounge)}).
\]

This rule fully defines the possible sequences in this circumstance. However it is clearly not scalable for more than a few possible sequences due to the necessity of defining all possible sequence options to choose from. A solution to this problem can be found in the choice sequence operator, denoted \( A \,<\to,B \).

The choice sequence operator has a relatively simple implementation given the existing sequential operator. To resolve the clause \( A \,<\to,B \), we proceed as follows:

- Attempt to prove \( A \gg B \), and if it succeeds return
- Otherwise attempt to prove \( B \gg A \)

A noticeable characteristic of this implementation is that it will always return the sequence \( A \gg B \) first if this is possible, resorting to \( B \gg A \) only if this is not possible. In practice we can iterate over multiple possible sequences using the backtracking mechanism.

A pleasing property of the choice sequence operator is its ability to express large numbers of possible sequences concisely and intuitively. For example, we may wish to express that five rooms be cleaned in any order as follows:

\[
\text{clean(room1)} \,<\to\, \text{clean(room2)} \,<\to\, \text{clean(room3)} \,<\to\, \text{clean(room4)} \,<\to\, \text{clean(room5)}.
\]

In practice, the number of sequence combinations implicit in the above statement is \( 5! = 120 \). Expressing such combinations directly grows exponentially as additional options are added. At an implementation level, the order in which the operators are specified will always be returned first, providing that sequence is possible.

### 3.2 Programming for Agents

An agent system may consist of a set of arbitrarily complex plans that define the behaviour of an entity. This implies the need to break down such plans into sequences of atomic actions which can be executed in the external environment. In essence, we wish to search for some hypothesis \( H \) (consisting of sequences of actions) that proves goal \( G \), given a set of beliefs \( B \), a set of plans \( P \) and a set of atomic actions \( A \).

Before we discuss how the planning problem can be approached, we will discuss some of these important components.

#### 3.2.1 Dealing with Facts

As discussed earlier, a goal in Lygon is expected to consume all available resources to be successful. In the context of an agent system, we require some mechanism for collecting the state of the world once a goal has been applied. Traditionally this could be accomplished by specifying a series of clauses which explicitly consume any state values that are available, and possibly printing them to the console. In the present implementation of Lygon, there is little recourse for achieving the desired behaviour using alternative methods. The following blocks world extract demonstrates (23):

\[
\begin{align*}
\text{showall}(\{\text{ontable}(X)\} | R) & \leftarrow \text{ontable}(X) \ast \text{showall}(R). \\
\text{showall}(\{\text{clear}(X)\} | R) & \leftarrow \text{clear}(X) \ast \text{showall}(R). \\
\text{showall}(\{\text{on}(X,Y)\} | R) & \leftarrow \text{on}(X,Y) \ast \text{showall}(R).
\end{align*}
\]
showall({hold(X)|R}) <- hold(X) * showall(R).
showall({empty|R}) <- empty * showall(R).
showall({}).
show <- showall(R) * output(R).

The showall clauses in this program will explicitly consume any facts of the form ontable(X), clear(X),
on(X,Y), hold(X) or empty by recursively calling itself and adding any consumables to a list. The show
clause starts the process off and then outputs the obtained list to the console. A goal which makes use
of the show mechanism may look something like go#show, proving go and then consuming whatever
facts remain once this is complete. Although this solution works for the purposes of allowing the goal to
succeed, it does not suit our purposes in an agent environment.

One of the problems with this method is the cumbersome and error prone syntax required to consume
facts. Not only do we have to define a rule for each fact we wish to consume, but we must do so for all
possible facts for the method to be successful. In an agent system where there are potentially many possible
states, this is unscalable and has significant potential for introducing programmer error. Another problem
with this approach is its inefficiency. Consuming a single fact has the potential to cause evaluation of up
to four of the clauses before it is consumed. Each clause failure requires backtracking to be performed so
that the next clause can be tried. Although negligible in a simple program such as the one presented, this
inefficiency can potentially be problematic when there are many fact types and many facts to consume.

Ideally we would like a more efficient and convenient way to collect facts. Such a solution was imple-
mented using the keyword 'collect'. The collect statement simply takes all facts (negations) in the context
available to it and consumes these. It then adds these into a special collect context list which is available
at the interpreter engine level. The collect operator couples intuitively as the right side of the sequence
operator for collecting the output state of an operation. For example we may apply some plan A and wish
to collect the state of the world at the end. Such a goal would take the form:
plan A >> collect.

The implementation of the collect operator is relatively simple, operating like so:

1. When resolving collect, consume all atoms of the form neg X from the context C, giving the new
context C2.

2. Check that context C2 is returnable and succeed if so, otherwise failing.

The order in which the collect operator is evaluated is important. Evaluating collect before plan A in
the previous example would consume the initial state and leave the output state of the plan untouched. For
this reason it is important to consider the order of evaluation inherent in connectives that are joined to the
collect operator.

3.2.2 Representing Agent Beliefs

An agent system consists of a set of known facts or beliefs that form a knowledge base and which are
obtained through perceptions. Beliefs in the knowledge base define an agents perspective on the current
state of the world, and are used to guide plan selection. In a BDI system as outlined in (35), an important
component is the belief revision function. This function takes perceptual inputs that the agent is exposed
to (generally obtained through sensors), and determines a new set of beliefs. In Lygon, beliefs about the
world can be stored in the form of negations that may be supplied by the main driver rule when querying
for a goal. For example, we may request a proof of goal G, given some facts (or beliefs in this case) A, B
and C, taking the form neg a#neg b#neg c#g.

Internally the above rule concurrently adds the beliefs a,b and c into the context along with goal g, then
evaluates g. In a BDI implementation, there is the potential for goals to be evaluated every cycle which
require access to the current state of the world. For this reason, we would like Lygon to maintain the state
that the world is in between multiple goals. In the standard method shown in the previous example, this is
difficult to accomplish without considerable manual bookkeeping and data entry of facts in between goals.
The solution to this problem was to introduce a specialised agent knowledge base for storing beliefs about
world state, that persists between goals, and which are available to appropriate goals and plans. This store
is by necessity implemented at the Lygon engine level to allow its use by internal agent components. The
store can be interacted with directly by the user through the Lygon console, or potentially interfaced with
external systems. The applicable command syntax is outlined below:

- **worldset([fact1, fact2, ...]):** This command sets the state of the world to the list of
  facts specified. Note that neg _ is not required before each fact, as this is implied.
- **worldadd(fact):** Adds a single new fact into the current world state.
- **worldadd([fact1, fact2, ...]):** Adds multiple facts into the current world state.
- **showworld:** This command displays the current state of the world to the user.

The implementation of the world belief store is as a simple list of negations in the form [neg fact1, neg
fact2, ...]. When the world state applies to a goal, we simply append this list into the context and then
evaluate the goal. This has the equivalent effect as writing neg fact1#neg fact2#...#goal.

One of the attractions of linear logic is its ability to model state changes. A direct consequence of this
property is the consumption of resources when they are used in a goal. In some cases however, we may
wish to test for the existence of a fact without consuming it. For example if we are defining a conditional
plan we may wish to specify that it applies only in the presence of a certain fact, but does not actually
consume that fact during evaluation. One way to model this in Lygon syntax is to reassert the facts after
we have consumed them like so:

gain ← fact1 >> neg fact1 >> fact2 >> neg fact2 >> dosomething.

The evaluation of the above rule leads to the deletion of fact1 and fact2 and then their reassertion
immediately after, and before calling the goal dosomething. The non-existence of the facts causes the
consumption rules to fail, giving the desired behaviour. Although this approach works, in practice it may
not be clear which facts need to be reasserted in the presence of disjunctions, leading to complex syntax.
For this reason, and for syntactic brevity, we have defined an operator called see(). The see() operator
takes a specified fact and tests for its existence without consuming it. The syntax is demonstrated in the
following example:

```plaintext
goal ← see(fact1) >> see(fact2) >> dosomething.
```

The operation of the see operator is intuitive. Internally the implementation is identical to the resolution
of atoms by the axiom rule (described in (23)), with exception that the specified negation is not removed
from the context before returning. Where the specified fact does not exist in the context, the operator fails.

### 3.2.3 Actions

An action is an atomic description of state changes that can be made by an agent in the world. As discussed
in (22), linear logic offers a natural way to represent actions and their effects on the world in which an
agent operates. A conjunction action system consists of a set of rules of the form I \(\rightarrow\) G where I consists
of the initial state of the world in which the action applies, and G consists of new facts that are generated
after the action is applied. Implicitly, facts which are not found in I \(\rightarrow\) G are not affected by the action. In
Lygon we have defined a simple conjunctive action system that allows the specification of a set of action
rules, and allows us to elegantly define many agent behaviours. The action syntax takes the general form:

```plaintext
act actionName: precondition1 *precondition2*... ⇒ postcondition1*...
```

Returning to our cleaning robot example, we may wish to define an action that specifies what happens
when we recharging our battery. Such an action may take the form:

```plaintext
act recharge: at(charger)•charge(C) ⇒ at(charger)•charge(100).
```

The preconditions of an action determine whether it is applicable in a specific circumstance. In this case
we can perform the recharge battery if we are currently at the charger. The second precondition consumes
the current charge fact (removing it from our current beliefs) so that we can replace it in the postcondition
with a new value. The postconditions define what new state is expected to be generated once the action has
been completed. In this case we are asserting that we are still at the charger and that our charge is now at
100%.

Internally action rules will be transformed into backward chaining clauses for use in plan construction
by abduction methods (described later in the paper). The general form of the transformation looks like this:
The semantics of this rule are as follows:

- Each of the preconditions are consumed from the program context.
- The action is emitted for action sequence construction (see later discussion).
- Each of the postconditions are asserted as facts into the return context.

The emitact() operator is a special internal syntax which indicates to the interpreter the presence of an action during plan construction. Its syntax is not intended for direct use. The effects of this operator are illustrated later in our discussion on plan decomposition by abduction.

An intuitive way to think of an action is as an input-output mechanism. We input a certain state of the world into an action (specified by preconditions) and get another state of the world out when it is complete, as is seen in Figure 1. Implicit in this definition is the understanding that only the applicable state is input into an action – an existing state which is not specified in the preconditions is retained and appended to the output state.

3.2.4 Plans

The new agent syntax allows the specification of plans in two forms: explicitly defined plans which define sequences of subplans and actions required to accomplish them, and implicit achieve() plans which attempt to derive solutions automatically based on action rules and heuristics. The two approaches to planning are outlined in the following section.

An explicit plan defines sequences of subplans and actions which achieve some system goal. A plan consists of a set of preconditions which determine whether that plan applies, and a body which describes how to achieve the plan. Multiple plans can be specified with the same name – plans are evaluated in top to bottom order, moving from one to the next if a plan fails or when backtracking for additional solutions. The general syntax of a plan is defined as follows:

\[
\text{plan planName}(\text{vars}): \text{preconditionClause} \Rightarrow \text{bodyClause}.
\]

Specifying a plan as a goal instructs Lygon to construct a sequence of actions which achieve the specified plan given a current state. Such a goal may take the form:

\[
\text{neg fact1}\#\text{neg fact2}\#\text{plan planName} >> \text{collect}.
\]

This command applies the plan in the context of the specified facts, then collects the final state after the plan has been applied by using a sequence and collect operator. The successful result of evaluating such a rule generates two sets of values - a sequence of actions which will achieve the specified plan, and a set of facts that were collected at the end of the plan by the collect operator. For example, a robotic vacuum program might consist of the following actions and plans:

\[
\text{act move}(Y) : \text{at}(X) \Rightarrow \text{at}(Y).
\]

\[
\text{act vacuum}(X) : \text{at}(X)\#\text{dirty}(X) \Rightarrow \text{at}(X)\#\text{clean}(X).
\]

\[
\text{plan clean}(X) : \text{see}(\text{at}(X))\#\text{see}(\text{dirty}(X)) \Rightarrow \text{act vacuum}(X).
\]

\[
\text{plan clean}(X) : \text{see}(\text{at}(Y))\#\text{see}(\text{dirty}(X)) \Rightarrow \text{act move}(X) >> \text{act vacuum}(X).
\]
The first plan specifies how to clean a dirty room that we are currently in. The second plan specifies how to clean a room that we are not currently in. The order of the definitions is important in this case as we try to clean a room that we are currently in first, and only if this fails will we attempt to move to clean another room. This prevents unnecessary movement actions. Given an initial state of the world in which we are currently in the lounge and there is a dirty lounge and dirty kitchen, we may specify the following goal:

```
neg at(lounge) # neg dirty(lounge) # neg dirty(kitchen) #
plan clean(lounge) >> plan clean(kitchen) >> collect.
```

This generates the sequence of actions:
```
act vacuum(lounge), act move(kitchen), act vacuum(kitchen)
```

As we would expect, the state of the world obtained by the collect clause looks like so:
```
neg clean(lounge), neg clean(kitchen), neg at(kitchen)
```

The utility of the sequence operator becomes apparent when we examine the body of a plan. This enables us to specify any number of actions (or subplans) inside the body of a plan or as a goal and get the expected result. Our use of see() in plan preconditions is important also as it allows us to test for facts without consuming them. Optionally we can choose to omit the see clause which will lead to consumption of the specified resources. Such an approach may be convenient in cases where the fact is not required in the body of the plan.

Recursive plans are a powerful feature, enabling us to call another plan (including the current plan) from within the body of a plan. We may extend our cleaning robot example with the following plan:

```
plan cleanall: not(dirty(_)).
plan cleanall: see(dirty(X)) ⇒ plan clean(X) >> plan cleanall.
```

This new plan checks for the existence of dirty rooms, calling the clean(X) plan to clean it then recursively recalling the current plan to clean the next dirty room. The first rule is called the termination rule, and specifies that the plan should succeed when there are no dirty rooms. Note that this rule contains no body, its only function is to cause the plan recursion to terminate once all rooms have been cleaned. It’s failure (in the case when there are dirty rooms) leads to the evaluation of the second rule, which will apply appropriate action to clean the dirty room. This second rule demonstrates our ability to nest plans, both external plans as well as recursively nesting the current plan.

Some plans can be achieved in a variety of ways. This can be expressed with multiple rules of the form:

```
plan planName(vars): preconditions ⇒ body1.
plan planName(vars): preconditions ⇒ body2.
```

These rules specify that a plan planName, having a set of preconditions (where preconditions are equivalent) can be achieved using either body1 or body2. A more concise definition can be specified using standard Lygon connectives. For example an equivalent definition of the above scenario can be defined as follows:
```
plan planName(vars): preconditions ⇒ body1 @ body2.
```

In practice, plan bodies can contain any of the usual Lygon connectives with the expected results.

Planning operates on the principle of backward chaining and abduction discussed later in the paper. For this purpose we are required to transform plan definitions into an appropriate clausal form. The general form of the transformation is relatively simple as follows:
```
plan planName(vars) ← preconditions * body.
```

Our use of the cross operator between preconditions and body is notable. One might imagine that we would make use of the sequence operator to evaluate the preconditions followed by the body. In practice the cross operator operates in a similar fashion, however does not allow new facts to be generated in the preconditions and passed into the body. This limitation is by design, reflecting the fact that the preconditions clause is designed to be a test of context, and should not be generating any new state.

Also notable in the above transformation is the new plan() prefix syntax attached to the head of the clause. This prefix unambiguously distinguishes plans from regular Lygon clauses, and allows us to query goals of the form plan planName(vars). Internally the resolution of a plan atom is very similar to regular clause resolution as outlined in (23), and occurs as follows:
• Select an appropriate plan clause in the order defined.
• Create fresh variables.
• Unify the plan head with the goal atom.
• Replace the atom with the clause body in the context.

Action definitions specify preconditions and postcondition states that are associated with it. An alternative way to specify a plan is to notice that we can accomplish any postcondition in an action definition by consuming all its preconditions and taking the rest of the postconditions as side-effects. For example, given the movement action definition act vacuum(X):
\[
\text{at}(X) \cdot \text{dirty}(X) \Rightarrow \text{at}(X) \cdot \text{clean}(X)
\]
we can construct a clause that will achieve a clean room in the form:
\[
\text{achieve clean}(X) \leftarrow \text{at}(X) \cdot \text{dirty}(X) \cdot \text{act vacuum}(X) \gg \neg \text{at}(X).
\]
Such clauses can be constructed automatically from action definitions. This provides a convenient shorthand mechanism that complements existing syntax.

A problem with using achieve causes in practical applications is that they can be highly indeterminate, leading to long, possibly infinite recursions. For example, we might specify a program with four movement actions like the following:
\[
\begin{align*}
\text{act move(north):} & \quad \text{at}(X,Y) \cdot \text{is}(\text{NewY},Y+1) \Rightarrow \text{at}(X,\text{NewY}). \\
\text{act move(east):} & \quad \text{at}(X,Y) \cdot \text{is}(\text{NewX},X+1) \Rightarrow \text{at}(\text{NewX},Y). \\
\text{act move(south):} & \quad \text{at}(X,Y) \cdot \text{is}(\text{NewY},Y-1) \Rightarrow \text{at}(X,\text{NewY}). \\
\text{act move(west):} & \quad \text{at}(X,Y) \cdot \text{is}(\text{NewX},X-1) \Rightarrow \text{at}(\text{NewX},Y).
\end{align*}
\]
From these action definitions we can extract a series of achieve clauses of the form:
\[
\text{achieve at}(X,\text{NewY}) \leftarrow \text{achieve at}(X,Y) \cdot \text{is}(\text{NewY},Y+1) \cdot \text{act move(north)}.
\]
A feature of this derivation is that it is recursive, enabling the achieve clause to handle multiple steps when we call achieve at(X,Y). Multiple clauses are derived from the action definitions due to multiple actions containing at() in their postconditions. Since clauses are chosen in definition order, this can lead to an infinite recursion through the first clause due to the lack of terminating conditions. An effective way to deal with such situation would be to specify which clause to use for each recursion. Such a mechanism has been implemented with the heuristic syntax.

Achieve heuristics can be specified against achieve clauses to allow for selection of clauses at each recursive step. A heuristic definition specifies a comparison function that compares two states to determine an ordering preference. To define a heuristic that enables the achieve at() clauses to try to follow the shortest path based on distance we could specify the following:
\[
\begin{align*}
\text{heuristic achieve at}(X,Y): & \quad \text{this}(\text{at}(X_1,Y_1)) \ast \text{other}(\text{at}(X_2,Y_2)) \\
& \ast \text{distance}(X_1,Y_1,X,Y,D_1) \ast \text{distance}(X_2,Y_2,X,Y,D_2) \ast \text{lt}(D_1,D_2).
\end{align*}
\]
This heuristic calculates the distance between the target location and current location after each possible clause is applied. If the selected clause (prefixed by \textit{this}) has a shorter distance to the target than the comparison clause (prefixed by \textit{other}), then it will be ordered first. Heuristic clauses can contain standard Lygon clauses. During decomposition of the achieve clause, each available clause is stepped once on the current program context to obtain a new context. For each unique pair of clauses, states resulting from two different achieve clauses are made available with the prefixes \textit{this} (the current clause being tested) and \textit{other} (the comparison clause). Success of the heuristic determines that the current clause has higher preference than the comparison clause. Failure determines that the current clause is equal or lower than the comparison clause. By evaluating multiple combinations we can obtain an ordered sequence of clauses. Ordered clauses are then evaluated in the defined order, only moving to the next clause on failure of the one before. The heuristic algorithm can be summarised as follows:

• Until achieve goal succeeds
• For each pair of applicable clauses.
– Evaluate a single step for both clauses on current context to obtain contexts C1, C2.
– Aggregate C1 and C2 into new context CH, prefixing this and other.
– Evaluate the heuristic clause on CH to determine success.
  * On success, order first clause before second.
– If ordering is incomplete, move to next pair of clauses.
– If ordering is complete.
  * For each clause in order.
    · Recursively evaluate next achieve step.
    · If evaluation succeeds, exit succeeding.

3.3 From plans to a plan of action

Now that we have defined sufficient syntax relating to the specification of agent plans, we can discuss how these components can be combined to help an agent decide what actions to perform to accomplish arbitrary goals. An important unifying component in the specified architecture that is so far missing, is the ability to decompose a goal into a set of actions that accomplish it. The method this is achieved is discussed below.

To decide on course of action, we wish to find some cause A (a set of subplans or actions), given a consequence G (our goal) and a set of explanations (planning clauses) of the form A → B, B → C, C → D, etc. One method of decomposing a goal into actions is through the use of abduction. Abduction does depth-first decomposition of a specified goal to determine sequences of actions or subplans that achieve it given the present state of the world. The granularity at which we calculate A can vary depending on our requirements. For example we may wish to reduce goal G into sequences of high level sub-goals whose consequence achieve G. Alternatively we may wish to fully decompose our goal into a sequence of atomic actions. A key factor in deciding how much decomposition to do is that of performance. Generating plans with high granularity can potentially consume far greater resources and take longer to complete than the alternative. The risk in maintaining a low granularity is the potential that decomposed goals may be unachievable in certain circumstances. For example, we may decompose our cleaning robot goal cleanall into a sequence of high level goals:

plan clean(lounge), plan clean(kitchen), plan clean(bedroom).

As we execute this plan, we are required to further decompose these subplans into actions from left to right. If we imagine that our robot has a limited battery life, we may encounter a situation where we do not have enough battery to complete the above breakdown. Since we have not sufficiently decomposed our goals to determine this early on, we are required to resort to a reactive approach – modifying our actions when the exceptional condition arises. The approach we have taken to implement agents in Lygon is to decompose goals fully into actions. Despite the additional resource requirements, this has proven to be a flexible and attractive option.

Abduction is a problem of 'finding the the best explanation for a given set of observations' (6). In planning, since we are envisioning some possible future we would like to achieve, we can equate such 'observations' with this desired state (in other words our goal). Our agents ability to change state in the world is limited to the actions it is able to perform, and therefore is the only means to explain the desired state. In this respect we are trying to find the best explanation consisting of a sequence of actions, for a given observation consisting of a goal.

The abduction method implemented consists of a depth first recursive algorithm that traverses a tree of clauses representing all the possible means to achieve the specified goal. As we descend the tree we wish to collect sequences of actions at the leaves of that tree. Abduction can operate on any standard Lygon clause, in our case we wish to work with the plan and action clause transformations specified earlier. A special action sequence structure is maintained throughout the resolution process, communicating action sequences constructed by subclauses. To illustrate the mechanism, imagine that we were to specify the goal clean(bedroom) to our cleaning robot program given that we are currently in the bedroom. The decomposition would operate like so:
1. Selecting the first clean(X) rule (which vacuums the current room), we decompose the clean(bedroom) goal by replacing its associated clause body:
   see(at(bedroom)) * see(dirty(bedroom)) * act vacuum(bedroom).
2. We resolve the first two see() atoms, succeeding, then attempt to resolve act vacuum(bedroom).
3. We decompose act vacuum(bedroom) by replacing it with its associated clause body:
   at(bedroom) * dirty(bedroom) * emitact vacuum(bedroom) * (neg at(bedroom) # neg clean(bedroom)).
4. The appropriate facts are consumed [at(bedroom), dirty(bedroom)] from context C, new facts are created [at(bedroom), clean(bedroom)] in the return context C2, and the action is emitted to the action sequence structure.
5. We return with new context, aggregating actions into a single sequence (in this case consisting of one action).

A very important aspect of the above decomposition is the use of the internal emitact() syntax. This clause is called to indicate to the abduction mechanism that an action leaf has been encountered and that it should be added into the action sequence structure. In practice this operator only exists inside action clauses. The operators is equivalent to the unit 1, succeeding where the context is returnable, but modifies the action sequence structure before return.

Aggregating actions across connectives occurs at the end of evaluation for each connective. The sequence operators have clear semantics as to the order in which sub action sequences can be combined. Other connectives are less obvious. A summary of how actions are handled for each connective is summarised below:

- A >> B: The action sequence B is appended to the end of sequence A, giving [A,B].
- A <=> B: Action sequence A can be appended to action sequence B or vice versa depending on the chosen sequence, [A,B] or [B,A].
- A*B: Actions joined with the cross operator have no implicit ordering, therefore we may join them as [A,B] or [B,A]. We select [A,B] by default.
- A@B: Either [A] or [B] depending on the successful subclause.
- A&B: Either [A] or [B]. Both actions are valid choices.
- A#B: Either [A,B] or [B,A]. No ordering is implicit to the operator.

An important mechanism of the abduction process is dealing with failures. Since we are planning in a 'what if' type scenario, a failure simply means that we have come across an unsuccessful solution to the goal. For this reason it is reasonable for us to backtrack and try a different solution until we find one that works. For example, we may modify our cleaner program to take into account the current battery charge, and modify actions to consume some amount of charge for each action. The specified program is shown below:

act move(Y): at(X) * charge(C) * is(C2, C-10) * gt(C2, 0) ⇒ at(Y) * charge(C2).
act vacuum(X): at(X) * dirty(X) * charge(C) * is(C2, C-20) * gt(C2, 0)
⇒ at(X) * clean(X) * charge(C2).
plan clean(X): see(at(Y)) * see(dirty(X)) ⇒ act move(X) >> act vacuum(X).

Given the goal plan clean(kitchen) and the context [neg dirty(kitchen), neg charge(25), neg at(lounge)], this program will attempt to decompose into the action sequence [act move(kitchen), act clean(kitchen)]. Unfortunately this will fail due to the action precondition gt(C2, 0), which ensures that given a reduction in charge for the action, that charge should stay above 0. The immediate response to this condition is to backtrack for an alternative solution, but since there is no alternative ways to clean the kitchen given the available 25% charge, the goal will fail. One way to deal with this situation may be to add an additional clean rule and a recharge action in the form:
act docharge: at(charger) • charge(C) ⇒ at(charger) • charge(100).
plan clean(X): charge(C) • lt(C,100) ⇒ act move(charger) >> act docharge >> plan clean(X).

Given this situation, failure caused by a lack of charge will cause backtracking on the clean(X) plan. Since the first clean(X) case is unachievable, our new alternative will be selected, causing the robot to move to the charger and recharge. This will generate the action sequence [act move(charger), act docharge]. The new plan then recursively calls plan clean(X) to complete the cleaning process, producing the final action sequence [act move(charger), act docharge, act move(kitchen), act vacuum(kitchen)].

3.4 Implementing reactive behaviour

The mechanisms covered so far provide a powerful and flexible platform for defining a variety of rational agent behaviours. Given a set of specifications and a goal, we can decompose these into sequences of atomic actions that will achieve that goal. What our system is lacking so far however, is the means to define reactive behaviours, suitable for highly dynamic and real-time environments. One way to specify such behaviours is by allowing the specification of events, which enable an agent to carry out actions in response to state changes in the world.

Events tend to be pure forward chaining mechanisms that act as the primary reactive component in BDI architectures (27; 26; 8). In many cases it may be desirable to interrupt the current execution of a sequence of actions to respond to an important occurrence. For example our cleaning robot may have sensors which detect the presence of a fire and activate an alarm. Clearly it is inappropriate to wait till the end of the current goal execution to respond in such a circumstance. As discussed earlier, reactive events can be characterised in the form:

situation → action

Since events tend to be dynamic, there is little rationale in trying to plan for them. To accommodate an event, we need to periodically interrupt the execution of the current action sequence to test whether any situation preconditions on an event are met. The mechanism combining such rational and reactive behaviours is accommodated by a specialised BDI cycle, discussed in the next section.

Events are evaluated in the order they are defined, giving the definition order a priority like property. Given its forward chaining properties and absence from the planning mechanism, an action is not transformed into a clause, but is stored as a tuple consisting of a name, a situation clause and an execution clause. Each of these fields are present in the event definition syntax. For example our fire alarm event may be written as so:

event firedetected: see(fire) ⇒ act alarm.

When the BDI cycle is testing for events, the situation clause is evaluated in the usual way against the current state of the world. This is accomplished by adding the current world state into a new context along with a clause of the form (where situation is the event situation clause):

situation >> collect.

Attempting to prove this clause in the current world context either succeeds if the situation conditions are met and the event applies, or fails if they are not. When a situation clause succeeds, the associated action clause will be executed.

The execution clause of an event can consist of more than a single action. In some circumstances we may wish to respond to an event by carrying out a series of actions, or even adopting a new goal thatrationally reasons about an event and responds appropriately. The former approach is more in keeping with the reactive flavour of an event – it simply replaces the agents current intentions with one or more actions in a dynamic fashion. The latter approach provides a powerful synergy of reactive event handling with deliberative planning, that can overrides the current goal when one of higher priority is encountered. Taking our fire detection event, the system would respond in the following fashion during the event phase:
• Check that the firedetected event is applicable by attempting to prove see(fire) >> collect.
• If the clause succeeds, replace current intentions with [act alarm].
• Once alarm action has completed, return to the originating goal.

Sometimes we may wish respond rationally to a situation by setting a new goal when an event fires. For example, imagine that our robot has fire extinguishing capabilities. We could define an event that activates a fire extinguishing mode when a fire is detected. This could be expressed as

\[
\text{event firedetected: } \text{see(fire(X))} \Rightarrow \text{plan firerespond(X)}.
\]
\[
\text{plan firerespond(X): } \text{see(fire(X))} \Rightarrow \text{act move(X)} >> \text{act extinguish}.
\]

This event overrides the current goal of the agent with plan firerespond(X) when a fire is detected in room X. When the plan is evaluated, it moves to the appropriate room and activates its extinguish action to put out the fire. In actuality, the execution clause of an event can be practically any valid Lygon clause that can be decomposed into a series of actions. The specified clause replaces the current goal in the BDI cycle, such that the agent will now work towards achieving it rather than its original goal. Only when the new goal completes or is unachievable will the agent return to its original goal. The mechanism for decomposing such clauses into sequences of actions is the abduction mechanism discussed earlier.

Using arbitrary clauses inside event bodies should be considered carefully. One of the prime purposes of an event is to provide responsive behaviours in dynamic situations. As discussed previously, the planning mechanism used to evaluate such clauses provides no strong guarantees for resource consumption and timeliness. In this respect it is difficult to determine how responsive such an event will be. As a general rule, non-action entities in the execution clause should be kept as simple as possible where responsiveness is important.

### 3.5 The BDI cycle

The components we have discussed so far enable us to accommodate rational and reactive agent behaviours in the linear logic framework of Lygon. A mechanism for integrating these components to work as a whole is an important step in developing a functioning agent system. This integrating mechanism is found in the form of the BDI agent cycle. A high level view of the BDI execution cycle defines it as the repetition of three phases:

- **Observe:** In this phase we observe the current state of the world to update our current beliefs.
- **Think:** The agent considers its alternatives to construct intentions that work towards achieving its goals.
- **Act:** The agent acts upon its intentions to change the state of the world.

The concrete implementation of the BDI cycle varies between systems. We have defined a custom cycle that implements this three phase cycle that accommodates both reactive and rational behavioural components defined earlier. The operation of the cycle can be summarised as follows:

Given an expected world state E, current intentions I and a goal G

1. **Observe the world to obtain current beliefs W.**
2. **Sequentially test each applicable event (where events are active).**
   - If the event applies, recursively call BDI cycle with the event body as goal, deactivating events.
   - Reobserve the world to obtain updated beliefs W
3. **If expected world E ⊈ observed world W**
   - Construct a new plan of action with goal G, replacing intentions I
   - If construction fails, terminate indicating failure
4. If current intentions $I$ is empty
   - Terminate cycle, indicating success

5. Otherwise
   - Obtain the first action $A$ in intentions $I$
   - Simulate action $A$ on current beliefs $W$ to get next expected world $E_2$
   - Execute action $A$
   - If action $A$ execution fails
     - Restart cycle with goal $G$, no expected world state and no intentions
   - Otherwise
     - Remove $A$ from intentions, getting $I_2$
     - Restart cycle with new expected world $E_2$, intentions $I_2$, goal $G$

This cycle has a number of important properties.

- The observation function obtains the current state of the world from the world state store. This enables state to persist between cycles and allows the user to specify a state at the command line, or integrate external sensors in a relatively simple way.

- Events are evaluated before planning occurs. This emphasises the reactive nature of events—they always take priority over the current goal. Events can be deactivated, for example if we are currently processing an event in the cycle we do not want to introduce infinite recursion.

- The execution of an event recursively spawns its own BDI cycle. This effectively takes over the current BDI cycle, trying to achieve the execution clause of the event. It is important to note that although the execution clause is evaluated using the abductive planning mechanism, this does not significantly compromise responsiveness for simple action bodies. Where the clause contains actions only, the planning mechanism will deterministically generate an action list from the specified action clause. At the end of the new BDI cycle we are returned to the original cycle and therefore the original goal.

- Constructing a plan of action fully decomposes the goal into a sequence of atomic actions. This guarantees its validity and allows the application of constraints described later.

- It is possible that no solution can be found to achieve goal $G$, given the current world $W$. In this case we terminate the cycle with a failure indication.

- The plan of action that is determined to achieve our goal constitutes our current intentions. Success is determined when there are no more actions left to do (we have no intentions). A world in which goal $G$ is already accomplished will return an empty plan of action if it is evaluated.

- Unnecessary replanning is avoided by maintaining an expected state of the world. This expected state is obtained by ‘simulating’ the present action on current beliefs and collecting the resulting state using the clause act theAction $>>$ collect. Where the world is equivalent to the expected state, our current intentions are still valid. External changes to the world may cause us to discard current intentions (which are no longer achievable) and generate a new valid set of intentions.

- Actions can fail. Internally actions are implemented with a 20% chance of failure. Failing an action is handled by re-evaluating the original goal from scratch.

- The cycle is reiterated after the execution of each action. This means that events have the opportunity to interrupt execution in between each action.

The observation phase in this implementation occurs at the first statement, obtaining the current state of the world from our world store. The thinking phase integrates both reactive properties by testing and reacting to events, and proactive properties by constructing a plan of action that forms current intentions. The action phase simply executes the first action in our intentions, before cycling back to the observation phase.
### 3.6 Maintenance goals

In many circumstances we would like to introduce additional goals to an agent program that should be maintained throughout execution, but for which it is difficult to specify using traditional achievement goals. For example, our cleaning robot may require that its battery never becomes fully discharged during the cleaning process. Since every action reduces the charge in the battery, it would be cumbersome to define conditional plans that check its state in between each action. A more convenient method would be a mechanism allowing the explicit specification of one or more goals to be maintained throughout execution. There are two potential approaches to dealing with such a situation, a reactive approach and a proactive approach.

As discussed in (33), a reactive maintenance goal reacts only when a maintenance condition has been violated, remaining inactive at all other times. For example we may define a reactive goal that interrupts our robot and instructs it to recharge when the battery drops below some threshold. In our new agent syntax, this behaviour can be achieved through the use of events. To instruct our robot to recharge once it has dropped below 20% charge, we might specify the following rule:

```plaintext
event lowbattery: see(charge(X)) • lt(C,20) ⇒ plan recharge.
```

As discussed in the previous section, the BDI cycle processes events before the execution of plan actions. This means that the lowbattery event will have the opportunity to execute before actions in the clean program, triggering whenever the battery discharges below 20%. This particular example demonstrates the utility in allowing event bodies to contain complex goals. Since the event body can override the present goal, the recharge plan can be as complex as we need, moving the robot to the charger and activating it as necessary. Once the recharge plan has completed, the robot will return to its original goal of cleaning, now fully charged.

As one would expect, reactive maintenance goals are a forward chaining mechanism. The immediate consequence of this fact is that the goal cannot be accounted for in planning. Given the goal cleanall (defined previously in the paper), a sequence of actions will be generated that moves the robot between the various rooms and cleans them. Unfortunately in the context of a robot having limited charge, the generated sequence may not be possible due to the large consumption of charge consumed per room. In practice the reactive maintenance goal will modify the sequence to maintain charge, however this poses an issue in the event that a single action uses too much charge. If we were to begin a cycle with 21% charge the maintenance goal would not fire. It may then be determined that we should clean a room, consuming 20% charge. In the next cycle the event will fire, with a charge of only 1%. The unfortunate outcome of this circumstance is that moving to the charger costs more than 1% charge, leading to the robot becoming fully discharged (and therefore stuck) despite the presence of the reactive constraint.

The inability of reactive maintenance goals to guarantee their success in all circumstances makes them unsuitable for some applications. For example, although it may be acceptable for a robotic vacuum cleaner to occasionally become discharged (it can be rescued by its owner), the same may not be true for the Mars rover. In safety applications there is an obvious need for strict guarantees regarding maintenance goals. An elevator program may require the maintenance of a goal that prevents doors being open while the elevator is not stopped at that floor. A train program may require that doors are closed while the train is in motion. Scenarios such as these require a more robust approach.

A better approach to maintenance goals would be one that can be evaluated during the planning phase, eliminating action sequences that lead to the goal being violated. Such a proactive approach largely removes the need for reactive maintenance goals by providing strong guarantees of conformance during action sequence construction. This makes the approach suitable for many applications such safety, resource management and maintenance of legal requirements. One of the more pleasing outcomes of implementing an agent framework based in linear logic, has been the realisation that such a mechanism can be implemented efficiently and in a relatively straightforward fashion within the architecture so far specified. In (33) Duff demonstrates mechanisms for enforcing proactive maintenance goals within JACK using the proactive-check algorithm. A weakness in this approach that Duff failed to address was its inability to deal with arbitrarily defined goals. Ideally we would like to explicitly specify constraints on the state of the world that are enforced pro-actively throughout the full planning phase. Such an approach is concise.
and intuitive for the programmer and provides guarantees of conformance for action sequences. It is our
contention that the mechanisms we have implemented within our agent framework provide such a solution.

A proactive maintenance constraint can be defined against any plan specification, and can specify arbitrary
test conditions. When action sequences are being constructed, these constraints are tested to determine
the outcome of actions up to that point. In circumstances where the constraint is violated, the current se-
quence construction fails, triggering the backtracking mechanism to try the next sequence. For example,
consider the following addition to our cleaner program, using the constraint feature:

\[
\begin{align*}
\text{act move}(Y) & : \text{at}(X) \cdot \text{charge}(C) \cdot \text{is}(C2,C-10) \Rightarrow \text{at}(Y) \cdot \text{charge}(C2). \\
\text{act docharge} & : \text{at}(\text{charger}) \cdot \text{charge}(C) \Rightarrow \text{at}(\text{charger}) \cdot \text{charge}(100). \\
\text{act vacuum}(X) & : \text{at}(X) \cdot \text{dirty}(X) \cdot \text{charge}(C) \cdot \text{is}(C2,C-20) \Rightarrow \\
& \quad \text{at}(X) \cdot \text{clean}(X) \cdot \text{charge}(C2).
\end{align*}
\]

plan clean(X): see(at(X)) \cdot see(dirty(X)) \Rightarrow act vacuum(X).
plan clean(X): see(at(Y)) \cdot see(dirty(X)) \Rightarrow act move(X) \gg act vacuum(X).

plan cleanall: not(dirty(_)).
plan cleanall: see(at(X)) \cdot see(dirty(X)) \Rightarrow plan clean(X) \gg plan cleanall.
plan cleanall: see(dirty(X)) \Rightarrow plan clean(X) \gg plan cleanall.
plan cleanall: see(charge(C)) \cdot lt(C,100) \Rightarrow \\
& \quad act move(charger) \gg act docharge \gg plan cleanall.

discharged <- see(charge(C)) \cdot lt(C,10).
constrain plan cleanall: not(discharged).

The first feature to note is the addition of an extra cleanall plan variant. This rule simply specifies that if
the robot has less that 100% charge, it can return to the charger for a recharge. The positioning of this rule is
important. Being last it is only evaluated when other rules have failed. This prevents arbitrary, unnecessary
recharges. The constraint syntax is specified in the last line. Constraint specifications are straightforward,
having the general form: constrain plan planName: condition.

The condition in a constraint can be any valid Lygon clause, and has the current state of the world
available in its context at evaluation. Returning to the earlier example, our constraint simply specifies that
the battery should not be discharged, referring to a regular Lygon clause that checks whether the current
charge is below 10%. The action rules specify an estimated reduction in charge they require to complete
by consuming charge(C) and subtracting some value from C, before reasserting it in the postconditions.
During the abduction planning process, constraints are tested immediately after an action is added to the
current action sequence. The success or failure of the constraint determines whether or not the current se-
quence should continue to be pursued. In this respect, constraints can be considered a method of ‘pruning’
the action sequence tree, eliminating all candidate sequences that violate the specified condition.

An important property of the abductive goal decomposition mechanism presented thus far is its ability

An important property of the abductive goal decomposition mechanism presented thus far is its ability
to construct action sequences in the order in which they are naturally executed. For example, given the
goal jump \gg spin, abduction proceeds by decomposing first the jump clause into a sequence of actions,
followed by the spin clause. In addition, it can be noted that the program context before and after a de-
composition represents the input and output states of the world, with reference to the current clause. For
example consider the decomposition of the goal:

\[
\text{act move(kitchen)} \gg \text{act vacuum(kitchen)} \gg \text{collect}.
\]

Given a starting context of \( \neg \text{at(lounge)}, \neg \text{dirty(kitchen)} \), decomposition begins by
passing this context (the present state of the world) into \text{act move(kitchen)}. As discussed earlier,
actions are decomposed into a fact consumption and assertion clause, in this case the following:

\[
\text{act move}(Y) \leftarrow \text{at}(Y) \cdot \text{emitact(move}(Y)) \cdot \{\neg \text{at}(Y)\}.
\]

The effect of this rule is, given some input context, to consume the current \text{at(lounge)} fact, emit
\text{move(kitchen)} into the current action sequence and then assert the new fact \text{at(kitchen)} into the
return context. The decomposition then proceeds to \text{act vacuum(kitchen)}, evaluating the clause:

\[
\text{act vacuum}(X) \leftarrow \text{at}(X) \cdot \text{dirty}(X) \cdot \text{emitact(vacuum}(X)) \cdot \{\neg \text{at}(X) \# \neg \text{clean}(X)\}.
\]
Remembering back to our discussion of the sequence operator, a central property of its decomposition is the passing of output context (a set of facts) from its left as the input context on the right. In this case, evaluation takes the context [neg at(kitchen), neg dirty(kitchen)] output by the move clause and generates the new context [neg at(kitchen), neg clean(kitchen)]. At the end of sequence evaluation, action sequences constructed for its subgoals are appended together. This gives us the final sequence [act move(kitchen), act vacuum(kitchen)].

An intuitive way to think of this process is as a flow of input-output states between the various actions making up a sequence. This property affords us the ability to test for conformance to maintenance goals at strategic locations throughout the decomposition. Since it is the state of the world that we are most interested in, and given that actions constitute our agents ability to change the state of that world, the most obvious place to perform such checks is on the output context of action clauses. In the case of our present example, we might represent the flow of these constraint checks with the following diagram:

At an implementation level, the constraint check mechanism runs immediately after each evaluation of the act() clause. It is at this point when the return context of the action clause contains the expected output state of the world after the effects of the action have been applied. Performing a constraint check is a relatively simple process of evaluating all applicable constraint clauses given an input state. For example, imagine that our robot has been barred from entering the kitchen to clean, using the constraint clause:

\[
\text{constrain plan clean}(X): \text{not}(\text{at}(\text{kitchen})).
\]

Specifying the goal clean(kitchen) would lead to its decomposition into the subgoal act move(kitchen) >> act clean(kitchen).

When the first action is evaluated it might output the context [neg at(kitchen), neg dirty(kitchen)]. At this point, the constraint checking mechanism activates, attempting to prove not(at(kitchen)) on the context of the action output. Since the context contains neg at(kitchen), the constraint check fails. The failure of a constraint has the effect of causing the action clause to fail, which triggers back-tracking to find the next possible solution. If one imagines action sequence construction as a search of a tree of possible sequences, constraint checking can be thought of as a method of trimming inappropriate branches before they are descended. Figure 3 illustrates such a decomposition for the goal clean(\_) in the context [neg dirty(kitchen), neg dirty(lounge), neg dirty(bedroom), neg at(bedroom)], which cleans any dirty room.
Constraint propagation is a central concept in constraint programming (28; 12; 2). Propagation of constraints refers to their aggregation into more restrictive constraints, as new constraints are encountered in a solution search. For example, nesting constrained plans within constrained plans requires that all constraint tests accommodate the tests of both sets of constraints. As constraints are nested at deeper levels, the number of constraints increases and become more restrictive in the states that they will accept. Often there are circumstances where multiple constraints can be reduced into simpler, more efficient equivalents that improve the speed of checks. For example the constraints $X > 5$ and $X < 7$ can be simplified into a single constraint $X = 6$.

Constraint propagation in an agent context is less demanding than pure constraint programming languages. In most cases constraints in an agent program will be limited to the elimination of a small number of exceptional conditions, and for providing conformance assurances. In this context there is unlikely to be large sets of varying constraints. This reduces the need for complex constraint propagation techniques such as simplification. When a new constraint is encountered it is appended to the current list of constraints, providing an equivalent constraint is not already in the list. Preventing the repeated addition of equal constraints helps to improve the efficiency of constraint checks within recursive plans. Constraint testing is simply a matter of checking each of the constraints in the list against the current context. If any of the constraints fail then the current sequence of actions is invalid.

3.7 Implementation

The mechanisms we have described in this section were implemented as extensions to the linear logic programming language Lygon (23). The extensions are well commented and sparsely arranged, adding an approximate 1100 lines of Prolog code to the existing 720 lines. This sums to a total of approximately 1800 lines. The current implementation has been developed and tested using SWI-Prolog.

4 Experiments and Evaluation

The brevity and expressive power of the Lygon extensions we have implemented become apparent when used in tandem and applied to complex agent problems. The syntax allows us to express complex scenarios in a very concise and highly abstracted fashion. This affords us clear productivity advantages over more prevalent imperative languages such as JACK. A decisive advantage of the Lygon extensions are its ability to express complex high level abstractions using expressive and concise syntax. This makes the tool suitable for rapid prototyping scenarios.

4.1 Robotic cleaner program

To evaluate the effectiveness of the implemented extensions to Lygon in the context of real problems, it is worth exploring its application to some more complex problems. The first problem extends upon the ideas presented in previous discussions for a robotic cleaning program. In the following example we combine these ideas and extend upon them to accommodate more complex reactive and proactive scenarios. Features and requirements that we may like to incorporate into such an application are summarised below:

- We would like our robot to accommodate the cleanliness of multiple rooms, scheduling for the vacuuming of all rooms in an appropriate sequence.
- We want our robot to deal with obstacles preventing it from cleaning. To accommodate this requirement we will equip our robot with a powerful ray gun, able to vaporize pesky targets.
- We need to maintain battery charge to prevent the robot running flat, and to extend the life of the batteries (by preventing full discharge).
- We wish to reactively respond to the presence of an intruder by activating the alarm.
- We wish to respond to the detection of a fire by replacing the current goal with a fire-fighting goal and returning to cleaning only when the fire has been extinguished. This new goal should allow for excessive emergency discharge of the battery, but should be able to perform an emergency rapid charge in the case that there is not enough power to extinguish the fire.
An implementation that demonstrates these requirements is outlined in full below:

```prolog
act move(Y): at(Y) * charge(C) * is(C2, C-10) ⇒ at(Y) * charge(C2).
act vacuum(X): at(X) * dirty(X) * charge(C) * is(C2, C-20) ⇒ at(X) * clean(X) * charge(C2).
act docharge: at(charger) * charge(C) ⇒ at(charger) * charge(100).
act dofastcharge: at(charger) * charge(C) * is(C2, 50) ⇒ at(charger) * charge(C2).
act extinguish: fire(X) * at(X) ⇒ smoke(X) * at(X).
act soundalarm: alarm(\_) ⇒ alarm(on).
act stopalarm: alarm(\_) ⇒ alarm(off).
act vaporize(X): at(X) * obstacle(X) * charge(C) * is(C2, C-20) ⇒ at(X) * charge(C2).

The cleaner program demonstrates a variety of features of the implemented extensions in Lygon. It has reactive and proactive properties, making use of deliberative planning techniques to determine appropriate action sequences, and events to respond to dynamic changes in the environment. It applies constraints in both the proactive planning phase to prevent inappropriate behaviour, and dynamically to react to exceptional conditions. The program also demonstrates the integration of standard Lygon clauses to define common functionality shared between components. The details of the implementation are discussed below.

An implementation that demonstrates these requirements is outlined in full below:

```prolog
act move(Y): at(Y) • charge(C) • is(C2, C-10) ⇒ at(Y) • charge(C2).
act vacuum(X): at(X) • dirty(X) • charge(C) • is(C2, C-20) ⇒ at(X) • clean(X) • charge(C2).
act docharge: at(charger) • charge(C) ⇒ at(charger) • charge(100).
act dofastcharge: at(charger) • charge(C) • is(C2, 50) ⇒ at(charger) • charge(C2).
act extinguish: fire(X) • at(X) ⇒ smoke(X) • at(X).
act soundalarm: alarm(_) ⇒ alarm(on).
act stopalarm: alarm(_) ⇒ alarm(off).
act vaporize(X): at(X) • obstacle(X) • charge(C) • is(C2, C-20) ⇒ at(X) • charge(C2).

The cleaner program specifies a number of atomic actions which characterise the agents interaction with the environment. The move rule allows the robot to change its location to an arbitrarily defined position. In practice the implementation of such an action will be encapsulated in an external system which handles all appropriate logic. For the purposes of determining its outcome, we presume that a single movement consumes 10% of the current charge. In a more precise implementation, we might specify less generic movement actions with unique discharge rates that reflect the relative distances between locations. For example we may wish to specify that moving between the lounge and bedroom consumes 12% charge and between the bedroom and kitchen 9% like so:

```prolog
act move(lounge, bedroom): at(lounge) • charge(C) • is(C2, C-12) ⇒ at(bedroom) • charge(C2).
act move(bedroom, kitchen): at(bedroom) • charge(C) • is(C2, C-9) ⇒ at(kitchen) • charge(C2).

The ability of the action clauses to accommodate both variables and constants in its parameter clauses affords us a great deal of flexibility in defining actions at various degrees of granularity. For simplicity and brevity purposes we retain the more generic movement specification, and apply the same approach to subsequent action definitions.
It is important to realise in the previous scenario that the amount of charge that is consumed by any particular action is highly dependent on the exact circumstances of its execution, such as the specific path taken, speed, temperature, age of batteries, etc. It would be impractical to model all of these factors in a meaningful way. Ultimately the specification of state changes made by action definitions should be considered an approximation of its expected outcome in normal circumstances. In the imperfect world, action outcomes are likely to vary at times, and may even fail completely. For example, we may consume more or less charge than expected when cleaning a room. An important factor to consider in defining actions is what variations can occur between specification and execution. If we were to significantly underestimate the battery charge consumed by our actions, it would be possible for constraint enforcement to fail to maintain charge. Faulty assumptions associated with actions can lead to action sequences having unexpected outcomes that are potentially irrecoverable. This also has implications for efficiency, due to the potentially significant amount of resources consumed for re-planning. When an action failure does occur or has an unexpected outcome, the BDI execution cycle will attempt to compensate by re-evaluating the present goal, taking actual state into account. This enables goals to be achieved even in the presence failures in many cases.

Internally a simulated execution of an action will fail randomly approximately 20% of the time. Internal action failures are modelled as complete failures they report failure without making any changes to the state of the world. In a typical simulated failure scenario a move action may fail to complete, leaving the robot in its original location. When such a failure occurs, the BDI cycle responds by immediately restarting the cycle for re-planning. In a complete failure scenario this process will usually result in a the construction of a plan identical to the previous (since the world has not changed). This effectively causes the robot to retry its previous action until it succeeds. In a real-world scenario we could imagine action failures making some partial state change in the world. For example a movement action between the bedroom and kitchen may fail halfway, leaving the robot in the lounge. When such a scenario occurs the BDI cycle will re-plan given the new state of the world generating a new plan that accommodates the partial state changes made by the action.

The vacuum action is our robots means to change a room from the dirty state to the clean state. It requires that the robot be in the specified room and consumes 20% battery charge. This action rule consumes facts of the form dirty(X) and asserts a facts of the form clean(X) to indicate the updated state of the room. It is expected that the dirty and clean facts will initially be generated by some external system able to monitor the cleanliness of the various rooms. A real world system might implement additional plans to search and discover which rooms are dirty or clean through observation (its knowledge is incomplete). A requirement of our robot is the maintenance of battery charge, which is consumed by the various actions it can execute. This implies a means to recharge the battery, encapsulated in the docharge action. This action requires that the robot be at the location labelled ‘charger’ and increases the charge level to 100%. In addition we specify a fast charge action called dofastcharge. This enables the robot to rapidly obtain 50% charge in emergency situations, but should be used sparingly as it reduces battery life over time.

Our robots emergency handling requirements specify that it must be able to put out fires and respond to intruders with the sounding of an alarm. The robots fire-fighting abilities are engaged through the use of the extinguish action. Given a fire in our present location, the presumed outcome of this action is the observation of residual smoke (logic to direct the extinguisher, etc is presumed external). Additionally the robot can activate and deactivate an alarm with the soundalarm and stopalarm actions. The extinguish action demonstrates an important design principle. In system design we need to determine where the boundaries of our system are. These may be specified at a high level (as in this case) where much of the logic is external to the system, or at a lower level by specifying the logic explicitly. An action can be considered an interface to some external system whose implementation is largely unimportant, only its effects on the environmental state.

In order to remove obstacles from our path, we need to be able to eliminate them through the use of our ray gun. This is accommodated by the vaporize(X) action, causing the elimination of an obstacle(X) fact in the specified location and consuming 20% of our charge.

The intuitive and concise specification of actions in the new syntax underlines the resource oriented nature of linear logic, appropriate in an agent context. A limitation in the new syntax is our inability to specify actions which only consume state or generate state without preconditions. In practice such actions are likely to be rare, however one can imagine circumstances in which an action would be used to specifi-
ally trigger a state (for use by external/ internal system), or remove a state. For example we may wish to model the activation of an alarm as the existence of a linear fact ‘alarm’. To turn the alarm on, we would simply generate the fact ‘alarm’, and to turn it off consume the same fact. Providing such extensions to the existing syntax are likely trivial, but as a practical feature we have so far found little convincing incentive to implement them.

To initiate the specified program, we would specify the command execute(plan cleanall). The cleanall plan represents the top level goal seeking to ensure all rooms have been appropriately vacuumed and are free of obstacles. The plan is defined recursively, cleaning a single room and calling itself to deal with any additional dirty rooms. The terminating condition is specified first and activated when there are no dirty rooms observed, triggering the success of the plan. Two cases are identified for cleaning requirements - when we are currently in a dirty room and when there is a dirty room elsewhere that requires us to travel. The ordering of these plans is specified such that cleaning the room the robot is currently in will take precedence over cleaning external rooms. Each of these rules calls the subplan clean(X) to handle both cleaning and obstacle removal in the current room.

The decomposition of plans occurs in the order in which they are defined and the first successful sequence is selected for execution. This is intuitive to the programmer and gives them some control over the which action sequences are derived to accomplish a plan. This capability was demonstrated in the previous example, which specified a preference for cleaning immediate rooms over external rooms. The limitation in this approach is the presumption that we can always order clauses in such a way as to obtain better action sequences first. In many cases this may be impractical if not impossible, leading to less than optimal approaches to achieving a plan. For example we may wish to order cleaning to reduce the distance travelled by our robot. This would require that we specify all factorial combinations of cleaning order sorted by distance, leading to a large, difficult to maintain and inefficient implementation. An improvement to the system might provide some means of heuristic selection enabling lower cost action sequences to be selected in preference.

The clean(X) plan is called when we would like to clean the room that we are currently in. It identifies two cases - when we are blocked by an obstacle (requiring it to be vaporized) and where we are free to clean. An alternative implementation to the program we have specified might be to require the user to specify an order in which to clean rooms using goals of the form clean(lounge) <> clean(kitchen) <> ... Considering such an implementation we might imagine a scenario in which a requested goal is unachievable. For example we may specify that the lounge, kitchen, bedroom and bathroom be cleaned in any order. Unfortunately it is not possible to have enough charge to complete four consecutive rooms, and is thus unachievable. A limitation in the current system is the way in which unachievable goals are handled. Requesting such an execution would simply return the result:

Goal is unachievable!

A more useful response may be a description of why the goal cannot be achieved. For example, we may indicate that the goal is unachievable because all planning paths fail when charge is determined to fall below 0, preventing the calling of additional actions. Given a list of reasons for failure (extracted from the various failure paths during abduction), it may even be possible to determine how to best partially achieve the specified goal (choosing the least undesirable failure). For example we could determine that we can clean the lounge and kitchen but not the bedroom and bathroom.

The requirement that we maintain battery charge during the cleaning process motivates two distinct mechanisms. Attached to the cleanall plan, we specify the constraint not(batterylow). This constraint proactively ensures that we do not generate any action sequences which will cause us to reduce battery charge below 25%. In addition we specify a reactive lowbattery event, which commands the robot to recharge when the low battery state is detected at the beginning of a BDI cycle. The combination of these two approaches provide a robust solution to maintaining battery charge that covers a variety of situations.

An important consideration when specifying constraints is the realisation that in their present form they are a solution trimming feature. In the context of our lowbattery constraint, this means that any combination of actions leading to a low battery will cause the current plan decomposition to fail. Without additional plan options it may be impossible to clean multiple rooms without creating a low battery situation. For this
reason we must provide additional plan options that enable the cleaning decomposition to continue even when the battery becomes low (i.e. recharge). This motivates the addition of a fourth cleanall plan option:

\[
\text{plan cleanall: } \text{see(charge(C))} \land \text{lt(C,100)} \Rightarrow \text{act move(charger)} \gg \text{act docharge} \gg \text{plan cleanall.}
\]

When planning decomposition has generated sufficient actions to cause the battery constraint to fail, the cleanall plan will be unable to continue cleaning rooms until it is recharged. This will lead to the failure of the first three plan cases. Backtracking will eventually reach the fourth plan option, enabling the robot to perform a recharge before continuing with the cleanall plan. Placement of the rule is again important, as we only wish to attempt a recharge after cleaning attempts have failed. This approach can be used as a general mechanism to accomplish constraint recovery in most circumstances. One limitation of this approach however is in the presence of nested plans. When nesting plans, all constraints are inherited and will therefore continue to trim solutions, however the additional parent plan options are not available for recovery. This can lead to the complete failure of the sub-plan even where recovery is possible by appropriate actions. The solution when this is a problem is to duplicate the additional plan options on subplans, in the same way that we did for the parent plans.

In addition to the proactive approach, we also specify a reactive constraint in the form of the lowbattery event. Given the proactive maintenance goal already specified, this event may seem redundant. In an ideal world a purely proactive constraint approach would be sufficient, however in the practical scenarios there may be circumstances in which the battery becomes low in ways that we cannot plan for. For example the owner may install a new battery that is mostly depleted or an action execution may consume more charge than expected. In such cases replanning may not be the best approach, firstly because it consumes significant resources which will consume additional battery charge, and secondly because assumptions about the amount of battery charge consumed may be incorrect. This rule takes care of such situations by reactively overriding the robots current plan and moving it to the charger to perform a recharge. In essence, this event can be considered a fail-safe mechanism that compensates for the imperfect nature of planning. Using a combination of pro-active and reactive maintenance goals ensures a robust system that is able to handle a variety of failure scenarios.

A subtle limitation of the current battery maintenance constraint on the cleanall plan can be seen during analysis of the programs behaviour. It can be observed that plans which reduce the battery charge below 35% and then attempt to recharge will fail, even though we might expect them to succeed providing charge stays above 25%. This situation arises due to the implementation of the recharge option in the cleanall plan. This plan consists of two actions - moving to the charger and then charging. Because the constraint still applies during the execution of this plan, constraints are enforced immediately after the movement action. Since movement costs 10% charge, by the time the robot arrives at the charger it is expected to have a low battery and thus the constraint indicates that the plan should fail. A simple solution in this case is to enable the constraint to succeed if we are currently at the charger, like so:

\[
\text{constraint plan cleanall: not(batterylow) @ at(charger)}
\]

This constraint ensures that the battery low constraint only applies when we are not currently at the charger. Although conceptually simple, the updated constraint is a little unintuitive. A improved approach might be a mechanism which excludes constraints in certain scenarios, or to specify a clause that is executed in response to a constraint violation (on which the constraint itself is not enforced).

### 4.2 Reactive Behaviour

The cleaning robot can deal with a number of dynamic scenarios through the use of reactive events, defined in order of priority. An important function for our agent is the handling of emergency situations. At the highest priority is the handling of fires when they occur. The fire event responds by activating the alarm and then calling upon the fireresponse(X) plan. Recalling our discussion on the implemented BDI cycle, events always take priority over proactive plans - replacing whatever plan the agent is currently pursuing and asserting the fireresponse(X) plan. This provides an intuitive mechanism for prioritising goals. Given some initial goal, events can replace that goal when certain conditions arise, returning to the original goal only when event goals have been achieved. Since events are evaluated in definition order, the programmer can specify their priority by the order in which events are defined. A limitation in the current BDI cycle
is its inability to nest multiple events. When an event is fired, all other events are deactivated until that event has finished execution. An improvement on this approach may be to allow events of higher priority to override those of lower priority, deactivating only events lower than the current priority. For example by specifying the intruder event above the low battery event, even if we are currently executing the recharge event we can still respond to the higher priority intruder event to activate the alarm.

The fireresponse plan identifies three scenarios - when there is no fire (in which case we just succeed), when there is a fire in the current room and when there is a fire in an external room. This plan makes use of the extinguish action to put out the fire, the expected outcome of which will be the consumption of the fire fact, but generation of smoke. Like the cleanall plan, this plan is constrained to prevent over discharging of the battery. In this case we would like to allow maximum discharge to fight the fire due to the emergency situation (we obviously don’t want to attempt an unnecessary recharge whilst a fire is raging). For this reason we specify that charge should not fall below 1%. Like the cleanall scenario, the addition of this constraint motivates an additional planning case in which the robot can return for a recharge if it is expected to become discharged. In this circumstance however, the plan will only be used when a full discharge is imminent (which would prevent firefighting), and performs a rapid charge to reduce time away from the fire.

An important property to notice about this implementation is the possibility for the cleanall rule to become unachievable in circumstances where a fire event occurs. When the fire event is triggered, the robot responds by executing the firerespond(X) plan. Although the constraints on the plan ensure that we will not fully discharge the battery during the fire fighting process, it does allow the circumstance where the battery can become close to depleted by the time the fire is put out. In some circumstances the battery may be discharged below 10%, a level at which the robot does not have enough power to move to the charger on conclusion of the fire-fighting plan. This behaviour is by design and reasonable in this circumstance - we do not wish the robot to attempt charging a low battery in circumstances where we are fighting a fire and have enough charge to put it out. In practice the reactive lowbattery event will attempt a recharge once fire-fighting is complete, but is potentially unachievable, leading to the robot becoming stuck (a small price to pay for putting out the fire). The presented scenario demonstrates the subtle interaction of multiple constrained plans in the presence of events. Although a constraint should ensure the maintenance of that condition throughout planning, it does not guarantee that the final state achieved won’t lead to the violation of constraints in other plans. This subtle behaviour should be considered when specifying multiple plans that contain constraints.

In addition to the fire event, we also specify an intruder event. This reactive event simply activates the alarm when an intruder is detected. Since it only performs a single action, it is highly reactive, able to offer timeliness guarantees. Its immediate effect is to replace the agents current intentions with [act soundalarm]. Reactive events of this form are suitable for real-time interactions. Since no planning is required, they are a purely forward chaining mechanism. As a final reactive measure, to accommodate the switching off of the alarm we define the safe event. This event is activated when the alarm is on but there is no longer a threat (fire or intruder). This enables the automatic deactivation of the alarm at the end of an emergency.

In some scenarios we may wish to specify goals which we would like to be met, but which are not essential. Such ‘soft’ goals can be thought of as preferences that the interpreter will attempt to achieve but has permission discard if they are unachievable. Such goals are useful for situations such as optimisation. For example, we may wish to specify that only rooms which are occupied should be cleaned if possible. This will cause the robot to clean all unoccupied rooms first (maintaining the soft goal). When there are no unoccupied rooms left to clean, the soft goal will be discarded and the robot will resort to cleaning the occupied rooms in order to achieve cleanall. In the present implementation, soft goals cannot be explicitly defined, however similar behaviour can be imitated in some circumstances using clever plan specifications. For example, we could define two cleaning options - the first to clean unoccupied rooms and the second (evaluated after the first) to clean the occupied rooms. Such an implementation might look like the following:

\[
\text{plan cleanall: } \text{see(dirty(X))} \cdot \text{see(at(X))} \cdot \text{see(unoccupied(X))} \\
\Rightarrow \text{plan clean(X)} >> \text{plan cleanall.}
\]

\[
\text{plan cleanall: } \text{see(dirty(X))} \cdot \text{see(at(X))} \cdot \text{see(occupied(X))} \\
\Rightarrow \text{plan clean(X)} >> \text{plan cleanall.}
\]
4.3 Gold mining program

The next program is based upon a game used by the CLIMA agent programming contest (see the website http://centria.di.fct.unl.pt/clima/ for more information), that demonstrates the use of implicit achieve goals and heuristics within an open world environment where knowledge is incomplete. It also makes use of prioritised events containing plans that override the agents behaviour under certain circumstances. The program implements a scenario in which the agent must collect as much gold as possible within a grid and return to a depot. Each cell in the grid is defined as a linear fact in the form cell(X,Y). As an agent moves around the grid, it makes observations about the content of cells, marking them with facts in the form observed(X,Y). The amount of gold being carried by the agent is stored as holdgold(G) where G is the number of gold currently held. When gold is observed it is represented as goldat(X,Y).

An important property of the following implementation is its reliance on an external server (known as the CLIMA server). It is the servers responsibility to provide all percepts to the agent relating to cells and gold that are visible from its present location. We presume that the server has access to the world context and is able to insert new facts accordingly. Actions are used to communicate to the server, indicating how to move the agent around in the grid, pick up and return gold to the depot, etc. Initial facts about available cells in the grid, the agents position, gold held, etc are expected to be created by the CLIMA server on start-up. As the agent moves around the grid the server generates the associated gold and obstacle facts when they are in observation range.

```
act pick: at(X,Y)*goldat(X,Y)*holdgold(G)*is(NG,G+1)
⇒ at(X,Y)*holdgold(NG).
act drop: depotat(X,Y)*at(X,Y)*holdgold(_) ⇒ depotat(X,Y)*at(X,Y)*holdgold(0).
act move(north): at(X,Y)*is(NewY,Y+1)*clear(X,NewY) ⇒ at(X,NewY).
act move(east): at(X,Y)*is(NewX,X+1)*clear(NewX,Y) ⇒ at(NewX,Y).
act move(south): at(X,Y)*is(NewY,Y-1)*clear(X,NewY) ⇒ at(X,NewY).
act move(west): at(X,Y)*is(NewX,X-1)*clear(NewX,Y) ⇒ at(NewX,Y).
heuristic achieve at(X,Y): this(at(X1,Y1))*other(at(X2,Y2))
distance(X1,Y1,X,Y,D1)*distance(X2,Y2,X,Y,D2)*lt(D1,D2).
event returngold: see(holdgold(3)) *see(depotat(X,Y)) ⇒ plan returngold.
event pickgold: see(at(X,Y)) *see(goldat(X,Y))*see(holdgold(G))*lt(G,3)
⇒ act pick.
plan collectgold: not(goldat(_,_)).
plan collectgold: closestgold(X,Y) ⇒ achieve at(X,Y) >> plan collectgold.
plan returngold: see(holdgold(G))*gt(G,0)*see(depotat(X,Y))
⇒ achieve at(X,Y) >> act drop.
plan findgold: goldat(_,_).
plan findgold: not(goldat(_,_)) ⇒
closestunobserved(X,Y) >> achieve at(X,Y) >> findgold.
plan play: see(goldat(_,_)) ⇒ plan collectgold >> plan play.
plan play: not(goldat(_,_)) ⇒ plan findgold >> plan play.
plan play: not(cellsunobserved).
clear(X,Y) ← not(obstacle(X,Y)).
distance(X,Y,X2,Y2,M) ← is(MX,X2-X)*is(MY,Y2-Y)
abs(MX,AMX)*abs(MY,AMY)*is(M,AMX+AMY).
abs(V,AV) ← (lt(V,0)*is(AV,-V)) @ (gt(V,0)*is(AV,V)) @
(eq(V,0)*is(AV,0)).
closestgold(X,Y) ← see(goldat(X,Y)) *not(isclosergold(X,Y)).
isclosergold(GX,GY) ← see(at(X,Y)) *see(goldat(GX,GY))
distance(X,Y,GX,GY,D1) *distance(X,Y,GX,GY,D2) *lt(D2,D1).
closestunobserved(X,Y) ← see(cell(X,Y)) *not(observed(X,Y)) *
not(iscloserunobserved(X,Y)).
iscloserunobserved(OX,OX) ← see(at(X,Y)) *see(cell(CX,CY)) *
distance(X,Y,CX,CY,D1) *distance(X,Y,OX,OX,D2) *lt(D1,D2).
cellsunobserved ← see(cell(X,Y)) *not(observed(X,Y)).
```
This implementation demonstrates the brevity and convenience of using achieve clauses in combination with heuristics to specify potentially complex tasks in a concise and intuitive fashion. The heuristic works by determining which of the four possible movement actions gets the agent closer to its destination. The clauses are then ordered and recursively pursued from the most desirable to least desirable. A condition on each of the movement action rules is that they cannot move to a location containing an obstacle (the cell must be clear). When the shortest direction leads the player through a cell containing an obstacle, the action evaluation will fail, triggering backtracking for the next action in the sequence ordering. This theoretically enables the agent to move around obstacles.

An issue with this implementation is the observation that backtracking will favour moving the agent in the direction it came rather than backtracking to previous planning steps. In practice this means that the planning decomposition will never backtrack more than a single step, generating an additional step to move in reverse rather than revising the current path. This can lead to infinite recursions and inefficient path calculations. One way to deal with this issue might be to keep track of which cells we have stepped on and do not move to them again for the current path calculation. For example we can modify our action definitions like so:

\[
\text{act move(north): } \text{at}(X,Y) \cdot \text{is}(NewY,Y+1) \cdot \text{clear}(X,NewY) \cdot \text{not}(\text{beenat}(X,NewY)) \\
\Rightarrow \text{at}(X,NewY) \cdot \text{beenat}(X,Y).
\]

This generates the state \text{beenat}(X,Y) each time an agent moves to a new cell. The preconditions specify that a movement action is not valid if we have already been there in a previous step. Once we have planned a path to the desired location, we simply need to consume all \text{beenat()} facts for the next path planning phase. Given the specified modification, a plan causing an agent to become stuck will leave no action options available, causing backtracking to previous steps. Previous steps will then choose the next action from available options based on distance preference.

An alternative to using the above achieve goal and heuristic for movement is the use of explicit plans. For example we might define the following set of plans:

\[
\begin{align*}
\text{plan moveto}(X,Y): \text{see}(\text{at}(AtX,AtY)) \cdot \text{eq}(X,AtX) \cdot \text{eq}(Y,AtY). \\
\Rightarrow \text{act move(north)} & \equiv \text{act move(east)} \Rightarrow \text{plan moveto}(X,Y). \\
\text{plan moveto}(X,Y): \text{see}(\text{at}(AtX,AtY)) \cdot \text{lt}(AtY,Y) \cdot \text{lt}(AtX,X) \cdot \text{clear}(X,Y) \\
\Rightarrow \text{act move(north)} & \equiv \text{act move(east)} \Rightarrow \text{plan moveto}(X,Y). \\
\text{plan moveto}(X,Y): \text{see}(\text{at}(AtX,AtY)) \cdot \text{gt}(AtY,Y) \cdot \text{gt}(AtX,X) \cdot \text{clear}(X,Y) \\
\Rightarrow \text{act move(south)} & \equiv \text{act move(west)} \Rightarrow \text{plan moveto}(X,Y). \\
\text{plan moveto}(X,Y): \text{see}(\text{at}(AtX,AtY)) \cdot \text{gt}(AtY,Y) \cdot \text{lt}(AtX,X) \cdot \text{clear}(X,Y) \\
\Rightarrow \text{act move(south)} & \equiv \text{act move(east)} \Rightarrow \text{plan moveto}(X,Y). \\
\text{plan moveto}(X,Y): \text{see}(\text{at}(AtX,AtY)) \cdot \text{lt}(AtY,Y) \cdot \text{eq}(X,AtX) \cdot \text{clear}(X,Y) \\
\Rightarrow \text{act move(north)} \Rightarrow \text{plan moveto}(X,Y). \\
\text{plan moveto}(X,Y): \text{see}(\text{at}(AtX,AtY)) \cdot \text{lt}(AtX,X) \cdot \text{eq}(Y,AtY) \cdot \text{clear}(X,Y) \\
\Rightarrow \text{act move(east)} \Rightarrow \text{plan moveto}(X,Y). \\
\text{plan moveto}(X,Y): \text{see}(\text{at}(AtX,AtY)) \cdot \text{gt}(AtY,Y) \cdot \text{eq}(X,AtX) \cdot \text{clear}(X,Y) \\
\Rightarrow \text{act move(south)} \Rightarrow \text{plan moveto}(X,Y). \\
\text{plan moveto}(X,Y): \text{see}(\text{at}(AtX,AtY)) \cdot \text{eq}(X,AtX) \cdot \text{clear}(X,Y) \\
\Rightarrow \text{act move(west)} \Rightarrow \text{plan moveto}(X,Y).
\end{align*}
\]

This implementation defines a set of preconditions and the directions in which we want to travel based on those conditions. Each call moves the agent one or two steps (diagonal) providing the cell is clear of obstacles. The implementation demonstrates a practical use of the sequence choice (\(\equiv\)) operator. Rather than specifying how to move in a diagonal direction using two alternative plans (or a long body), we simply specify that we do not care which order the movement actions are executed in. The sheer number of plan options required to define movement demonstrates the utility in using the more concise achieve and heuristic syntax.

Since not all information is available about the environment in this scenario, we need to implement a search strategy for finding gold. The top level plan defining an agents behaviour is the play plan. This plan
has three cases - when we can see gold, when we cannot see gold but there are unobserved cells and when there are no cells left to observe (the terminating condition). When there is gold visible, the collectplan plan is called to go pick it up. This recursively moves the player to the closest observed gold, picking it up until there is no more observed gold available. The first case of this plan is the terminating case. The second case demonstrates the use of a standard Lygon (closestgold) clause in the preconditions. This clause iterates over all gold to find the one closest to the players current position, then returns it in X,Y.

A noticeable omission from the collectgold plan is the use of the pick action. Rather than explicitly picking gold in this plan, we have chosen to model gold picking as an event. At the beginning of a BDI cycle the pick event determines if we are currently standing on gold and have carrying capacity. If so if replaces current intentions with the pick action before returning to the original plan. The advantage in this approach is that we do not need to explicitly seek to pick up gold - if we are ever in the same cell as some gold and have carrying capacity then it will automatically be picked up. The collectgold plan simply moves us to the location of gold so that this event can be triggered. Returning gold has also been modelled as an event. When a player has reached maximum gold, this event replaces the current plan with the returngold plan. Once the gold has been returned the BDI cycle returns to the original play plan.

When there is no visible gold in the grid, the play plan calls upon the findgold plan. This plan simply finds the closest unobserved cell and moves the agent towards it. The preconditions on this plan ensure that it is only valid when we cannot see gold. Once gold has been spotted, the terminating condition enables the plan to succeed and return to the play plan. The play plan terminates when there is no longer any unobserved cells.

4.4 Reflection

One of the primary goals we set out to achieve in this research was to determine how effectively forward and backward chaining could be integrated within a linear logic agent system. We believe that the implementation we have developed shows conclusively that the framework proposed by Harland and Winikoff (8) is not only feasible, but has many valuable properties for agent systems. The syntax we implemented on top of the Lygon interpreter offers an intuitively simple yet very powerful framework for specifying agent behaviours. Linear logic has proven to be an effective framework for application to agent oriented programming, effectively modelling actions and resource oriented logic suitable for agents. Specifying plans in linear logic turns out to be relatively intuitive once the appropriate tools have been specified. Reactive behaviours have been effectively modelled in the form of events, enabling a powerful synergy of agent behaviours suitable for many applications. One of the more pleasing artefacts of the implemented agent extensions was the relatively straightforward means by which proactive constraints could be implemented on top of the framework. Proactive constraints provide an extremely powerful mechanism for arbitrarily restricting agent behaviours in an intuitive way. The constraint mechanism effectively implements many of the ideals proposed by Simon Duff in (33) for proactive maintenance goals in an agent context.

5 Conclusions and Future Work

The use of the Agent Oriented paradigm in programming is becoming increasingly prevalent for a variety of real world applications. The paradigm is appropriate for applications that are dynamic and that require goal directed and timely behaviours. The combination of reactive and deliberative approaches promises to be a flexible and adaptive framework for developing such applications. Linear logic offers an intuitive and powerful framework for modelling many agent concepts such as actions, plans and multi-agent concurrency. In this paper we have demonstrated how such agent oriented extensions can be implemented within a linear logic programming framework. We have developed a set of tools that have proven to be a valuable tool for defining agent systems in a concise, high level an intuitive fashion. In section 3 we showed how sequentiality can be implemented within a linear logic framework, providing an intuitive and powerful way to express a variety of behaviours in agent planning. We demonstrated how agent planning can be accommodated within a linear logic framework, developing appropriate syntax on top of the Lygon interpreter. We demonstrated a natural mechanism for expressing actions and applied these to agent problems. We demonstrated the use of reactive event syntax in an agent framework and discussed its implementation in
Lygon. We have compared and contrasted reactive and proactive agent behaviours, demonstrating the implementation of associated mechanisms and their integration within a uniform BDI architecture. Finally we introduced the concept of using reactive and proactive maintenance goals within a linear logic agent system and demonstrated associated implementations within Lygon. The effectiveness and expressive power of the implemented tools have shown conclusively that applying linear logic in combination with forward and backward chaining provides a very powerful framework for agent system development. These properties suggest the need for further research and development.

In the current BDI implementation, the decomposition of plans into action sequences is relatively limited in the sequences that will be generated. Decompositions occur in a depth-first, in-order approach according to the sequence in which plans, actions and clauses are defined. This gives the programmer some control over the solutions derived, but is limited by its lack of consideration for potentially superior alternatives. In practical applications it is likely to be difficult or impossible to extract optimal sequences through naive clause ordering. In the current implementation, multiple equivalent action sequences can be derived simply by backtracking for alternative solutions. This capability is limited however by the lack of appropriate mechanisms to decide which of the alternatives is superior. A more powerful solution would be one which considers such multiple alternatives, deriving some measure of cost to each and selecting the best solution. For example, if we wish to calculate the best order to clean a set of rooms we could derive a cost function that takes factors such as distance, size and charge into account. This function could then be applied to multiple alternative sequences to determine the best course of action. Alternatively or in addition to such a mechanism, we could provide more flexible clause selection mechanism that dynamically orders the selection of clauses during decomposition to obtain favourable sequences first. This approach could provide benefits for many applications, but in practice it may not be feasible to determine which clauses are preferred during the decomposition stage.

A possible implementation of the discussed selection mechanism could allow for the association of arbitrary cost values with individual actions. These costs could be statically defined or calculated dynamically with a programmer defined cost function. During decomposition of plans into sequences, it would be relatively trivial to obtain the associated cost of an action sequence up to that point by accumulating cost values for each prior action. These accumulated costs could then be compared for multiple potential sequences to determine the best option.

One of the properties of the implemented BDI cycle is its re-evaluation of the current goal and events after each action. In a dynamic environment this allows for reactive behaviours and plan changes when the state of the world changes. In practice however, atomic actions may be small units of functionality that are accomplished in short periods of time relative to expected changes in the environment. In such cases the single action cycle becomes inefficient and unnecessary. Often it may be the case that few changes are expected in the environment during the period of an action execution. A simple extension that could improve upon the current situation is to allow the aggregation of multiple actions between cycles. One possible implementation of such a mechanism could take the form of a transaction syntax. For example, to specify that a program activate an alarm and send a distress signal in the same BDI cycle we could specify a syntax like the following:

\[
\text{event emergency: condition} \Rightarrow \text{transaction(acl alarm >> act distress).}
\]

The implementation of transaction mechanism is potentially straightforward. When a transaction block is encountered, one simple approach would be to cycle through the intention execution phase of the BDI cycle until the end of the transaction block is reached.

The demonstrated proactive constraint mechanism has proven to be a powerful addition to the agent syntax. A limitation in the current approach however has been the need to explicitly provide plan alternatives to accommodate failures caused by constraint violation. For example, the cleanall plan in the demonstrated cleaning program requires the addition of a charging alternative to facilitate planning failures caused by a low battery. A better approach would be the provide an explicit constraint violation response clause, that performs some action or plan enabling the original goal to continue. For example, we could specify the execution of a recharge plan in the case that the battery maintenance constraint is violated. Once the recharge plan has completed, the original cleaning plan can continue with the current action sequence.
Such a mechanism would provide a more flexible and scalable approach, and could simplify the use of global constraints. A global proactive constraint applies to all plans and events in a program, limiting action sequences that can be produced by causing them to fail when a constraint is violated. For example, we could specify a global constraint to maintain battery charge in our robot, and a recovery constraint to initiate a recharge using the following syntax:

\[\text{constrain: not(lowbattery) } \Rightarrow \text{ plan recharge.}\]

In practical application, such a recovery condition could probably be implemented by decomposing the recovery goal into a series of actions and inserting them into the action sequence at the point of constraint violation. Decomposition can then return to the original goal to finalise the sequence. In some cases we would like to specify a constraint as a pure solution trimming mechanism that eliminates invalid action sequences. For example in an elevator safety system we may wish to remove from consideration all action sequences that lead to unsafe states (there may not be a recovery action that is appropriate). This behaviour is more characteristic of the present implementation. One solution to modelling such scenarios within the context of the proposed extensions may take the following syntax:

\[\text{constrain: safe } \Rightarrow \text{ fail.}\]

The power of the specified constraint mechanism is potentially vast, offering syntactically and conceptually simple mechanisms with a great deal of expressive power for practical applications.

A well known property of linear logic is its ability to natively express concurrency using the par operator. This property suggests applications in multiple agent scenarios. In its present form, Lygon provides some very limited multi-agent capabilities that are potentially compatible with the implemented agent extensions. One of the issues that requires addressing to make this practical however, is a means to model interaction between multiple agents in a cooperative fashion, and how these relate to the syntactic extensions implemented. In theory, one might imagine running parallel plans by specifying the goal \(\text{plan1#plan2}\). Experimentation has proven this simple approach to contain a number of difficulties. One such issue is how to appropriately decompose the multiple parallel plans fairly such that both agents execute their actions concurrently. Another issue is how plans and actions can be shared between agents in such a way that they can update their own state independent of the other. One possible approach may be to pass the agent identity as a parameter into plans and storing facts in a similar way. Such a mechanism could be handled automatically to simplify the syntax, transforming plan, event and action definitions in the compile stage.

As a concurrent programming paradigm, agent oriented programs can be very difficult to debug. A valuable avenue for further exploration might be methods and methodologies for debugging such programs in Lygon. A simple debugging trace tool might simply print out each plan and action in a decomposition, identifying sequences that fail, and the states in which they fail. In the present implementation of Lygon, some tracing features are available which allows the user to specify variables to watch and to break execution whilst programs are executing. Unfortunately these features do not currently extend to cover the new agent syntax in an effective way. Additionally work is required to determine how debugging should occur within the forward chaining BDI cycle.

References


