From Logic Programming towards Multi-agent systems

Robert Kowalski and Fariba Sadri
Department of Computing
Imperial College, London, UK
rak, fs@doc.ic.ac.uk

Abstract

In this paper we present an extension of logic programming (LP) that is suitable not only for the "rational" component of a single agent but also for the "reactive" component and that can encompass multi-agent systems. We modify an earlier abductive proof procedure and embed it within an agent cycle. The proof procedure incorporates abduction, definitions and integrity constraints within a dynamic environment, where changes can be observed as inputs. The definitions allow rational planning behaviour and the integrity constraints allow reactive, condition-action type behaviour. The agent cycle provides a resource-bounded mechanism that allows the agent's thinking to be interrupted for the agent to record and assimilate observations as input and execute actions as output, before resuming further thinking. We argue that these extensions of LP, accommodating multi-theories embedded in a shared environment, provide the necessary multi-agent functionality. We argue also that our work extends Shoham's Agent0 and the BDI architecture.

Keywords: Logic based agents, agent cycle, integrity constraints, reactivity, rationality.

1. Introduction

In the 1980s logic programming (LP) was widely viewed as providing a new foundation for Computing. In particular, the Fifth Generation Project in Japan identified concurrent LP as the core software technology, to implement knowledge intensive applications on highly parallel computer architectures. Prolog, a sequential variant of LP, was the rising star among computer languages, especially for AI applications; and many of us expected deductive databases, a database-oriented form of logic programming, to provide the basis for the next generation of database systems.

These expectations were not fulfilled. Concurrent logic programming failed to solve the problems of the Fifth Generation Project; Prolog became a niche language and was overshadowed by C++ and Java; deductive databases were unable to displace relational databases and to compete with the rising fashion for object-oriented databases. Indeed, object-orientation, rather than LP, has become to be seen as the new unifying paradigm for all of Computing.

Although LP may not have fulfilled its wildest expectations, it has nonetheless achieved a number of important successes in areas such as constraint logic programming, inductive logic programming, knowledge representation (including temporal, metalevel, non-monotonic and abductive reasoning) and logic grammars.

In this paper we will try to identify some of the deficiencies of LP viewed as a comprehensive foundation for all of Computing. We will argue, in particular, that the main problem of LP is that it is suitable only for the "rational" component of a single agent; whereas, to be fully comprehensive, LP needs to be extended to account also for the "reactive" component of agents and to encompass multi-agent systems. The extension to multi-agent systems, in

particular, arguably brings to LP the main benefits of object-orientation, including encapsulation.

Building on earlier extensions of LP, we will present an outline of such a more comprehensive foundation. We will argue that extensions of LP, such as abductive LP, incorporating integrity constraints, provide the necessary reactivity; and extensions of LP, in which agent programs are embedded in a shared environment, provide the necessary multi-agent functionality. As a by-product of these extensions, we will obtain a form of LP, in which input-output and change of state of the environment are justifiably non-declarative.

Although we do not discuss implementation and efficiency issues in this paper, we believe that non-declarative change of state allows a form of logic programming with destructive assignment, which is potentially more efficient than purely declarative logic programming. There have been a number of implementations of the extended logic programming system, most notably the one reported in [4].

The paper is structured as follows. In section 2 we describe an abstract agent cycle. In section 3 we describe in detail the representation of the agent's beliefs and goals. In sections 4, 5, and 6, respectively, we describe the semantics, the proof procedure and a unified agent cycle that combines reactivity with rationality. In sections 7 and 8, respectively, we discuss the correctness of the cycle and the model of inter-agent and agent-environment communication. In sections 9, 10 and 11 we compare our work with the Rao-Georgeff BDI (Belief-Desire-Intention) architecture, Shoham's Agent0 and several other approaches. Finally section 12 concludes the paper and outlines future work.

This paper expands upon [12] in two important respects: First, it treats "observations" as a special kind of belief, different from beliefs that are represented by means of logic programs. In [12], observations were incorrectly treated as goals. Second, it argues that the agent architecture subsumes the functionality of the Rao-Georgeff BDI architecture [18] and Shoham's Agent0 [21].

2. The observe-think-act agent cycle

LP, in its narrowest sense, concerns itself only with the thinking component of a rational agent. Interactions with the environment are limited to inputs that are goals to be solved and to outputs that are solutions to goals.

The deductive database incarnation of LP, on the other hand, is also concerned with more general inputs including updates. Techniques have been devised for assimilating updates and in particular for verifying that updates satisfy integrity constraints. These include those of [19,17]. Knowledge in such deductive databases is partitioned into logic programs, which define the data, and integrity constraints, which constrain the data.

However, compared with intelligent agents and with embedded computer systems, the outputs of deductive databases are restricted to answering queries and to accepting or rejecting updates. They do not encompass the performance of more general actions, such as sending messages or activating peripheral devices. Such outputs can be obtained, however, by the use of active rules of active database systems.

Independently of developments both in LP and in the database field, the notion of intelligent agent has begun to provide a more unified and more coherent approach to problems in the field of Artificial Intelligence (AI).

The behaviour of an intelligent agent can be characterised in abstract, simplified terms as a cycle:

*To **cycle** at time T,*
***observe** any inputs at time T,*
***think**,*
***select** one or more actions to perform,*
act,
***cycle** at time T+n.*

Here *n* is the amount of time taken for a single iteration of the cycle. In a realistic situation, the value of *n* would vary from one iteration of the cycle to another, as would the distribution of the value of *n* between the various components of the cycle.

If more than one action is selected for performance, then conceptually they may be viewed as being performed in parallel.

In this formulation, the four components of the cycle, observation, thinking, selecting actions and performing actions, are treated as taking place in sequence. However, it is also possible to formulate them as concurrent processes. The concurrent process formulation of the cycle is more general and more flexible than the sequential version. However, for simplicity, we will employ the sequential formulation in the remainder of this paper.

Different types of agents and different approaches to AI differ in the way in which they define the various components of the cycle. In this paper, we will argue that the functionality of many of these approaches can be obtained as special cases of a uniform agent architecture, in which thinking combines reasoning with both logic programs and integrity constraints. We will argue, in particular, that the behaviour of Shoham's Agent0 [21] and the BDI [18] agent architecture can both be obtained as such special cases.

3. The thinking component of the unified agent architecture

The thinking component of our agent is a proof procedure which is a variant of [6, 7]. It combines backward reasoning by means of abductive logic programs with forward reasoning by means of integrity constraints. The original proof procedure was developed with the aim of unifying abductive logic programming (ALP), constraint logic programming (CLP) and semantic query optimisation in deductive databases (SQO). Our variant allows the proof procedure to run within a dynamic environment, where changes in the environment (including communications between agents) can be observed as inputs.

The set of predicates is partitioned into four subsets:

- "closed" predicates defined by a set of definitions in iff-form
- "open" predicates representing actions that can be performed by the agent
- "open" predicates representing events that happen in the environment and actions of other agents
- "constraint" predicates such as = and <.

Open predicates are more commonly referred to as "abducibles". The terminology "open predicates" is due to Denecker and De Schreye [6].

The **internal state** of an agent consists of:

1. Beliefs which are iff-definitions for closed predicates

2. Beliefs which are observations, recording positive and negative instances of open or closed predicates
3. Goals in if-then form, representing
   - commands
   - queries
   - obligations and prohibitions
   - integrity constraints
   - condition-action rules
   - commitment rules
   - atomic and non-atomic actions that can be performed by the agent.

It is probably this use of "goals", to represent such conceptually diverse objects, that is the most controversial feature of our agent architecture.

Compared with the BDI model of intelligent agent, we do not have a separate component for intentions. In terms of our agent model, intentions would be atomic and non-atomic actions that the agent is committed to perform in the future. We could modify our model to accommodate such intentions by extending the selection of an action to perform so that it selects an entire conjunction of actions to be performed in the future. Such an extension would be especially useful for multi-agent systems, where agents might communicate their intentions in order to harmonise their interactions.

In the remainder of this section we define the syntax of beliefs and goals formally and illustrate it by means of examples.

3.1 Beliefs represented by logic programs in iff-form

Example 3.1

explore iff [clear & forward & explore] or
[obstacle & right & explore] or
[can-not-see & park]

This example defines the closed predicate "explore" by means of a tail-recursive logic program. The definition reduces the goal of exploring to the subgoals of moving forward and continuing to explore if it is clear ahead , or to the subgoals of turning right and continuing to explore if there is an obstacle ahead, or to the subgoal of parking if it cannot see ahead.

Example 3.2

explore(T1, T2) iff [clear(T1) & forward(T1+1) & explore(T1+1, T2)] or
[obstacle(T1) & right(T1+1) & explore(T1+1, T2)] or
[can-not-see(T1) & park(T2) & T2=T1+1]

Here the predicates have time parameters. A non-atomic action predicate, such as "explore", has two time parameters, standing for the start and end times of the action respectively. Atomic action predicates, such as "forward", "right" and "park", and properties, such as "clear", "obstacle" and "can-not-see", have a single time parameter. For simplicity, atomic actions are assumed to be instantaneous, although this assumption can easily be modified.

The inclusion of time parameters as arguments of atomic actions constrains the selection of atomic actions to perform to those whose times are identical to the cycle time or are at least consistent with the cycle time and with any constraints relating them. This use of time parameters to control the execution of atomic actions is similar to their use to execute "commitments" in Shoham's Agent0.

The definitions in examples 3.1 and 3.2 are related to one another in much the same way that definite clause grammars are related to Prolog programs with explicit difference lists. In the same way that difference lists are hidden in definite clause grammars, it is possible to hide time parameters in the definition of complex actions in terms of lower level actions. Thus definitions of the form 3.1 can be compiled automatically into the form 3.2. Such an approach has, in fact, been incorporated in Davila's implementation [4], following the example of Golog [15].

Although our use of time parameters resembles the use of time in the event calculus [13], our treatment of time is neutral with respect to the underlying treatment of the relationship between events and the properties they initiate and terminate. In particular, it is possible in many cases to avoid explicit (and computationally expensive) reasoning about persistence of properties, by extending the class of open predicates to include properties and by using the environment itself as its own model.

In example 3.2, in particular, the properties "clear", "obstacle" and "can-not-see" can be verified directly by observations in the environment, rather than by reasoning about the actions and events that initiate those properties, and about the persistence of those properties in the absence of terminating actions or events.

Example 3.3

explore(T1, T2) iff [if clear(T1) then [forward(T1+1) & explore(T1+1, T2)]] &
[if obstacle(T1) then [right(T1+1) & explore(T1+1, T2)]] &
[if can-not-see (T1) then [park(T2) & T2=T1+1]]

This is an alternative formulation of example 3.2. As we will see later in example 3.7, this example is interesting because, except for the recursive occurrence of "explore", the right-hand side of the definition has the form of condition-action rules.

It is possible to show that example 3.3 is equivalent to example 3.2, under the assumption that "clear", "obstacle" and "can-not-see" are mutually exclusive and exhaustive.

In general:

[p & q] or [not p & r] is equivalent to [if p then q] & [if not p then r].

Formally, definitions represent the completely defined predicates of an agent's beliefs. They have the form of generalised logic programs in if-and-only-if form:

(i) $G \leftrightarrow D1 \vee ... \vee Dn \qquad n \geq 0.$

Note that in the formal definitions we use symbolic notation for the logical connectives, in contrast with the informal notation used for examples in the paper.

G is an atom in the closed predicates and is said to be the *head* of the definition. The disjunction is said to be the *body*. The disjuncts *Di* are either the logical constant *false* or conjunctions

(ii) $C1 \wedge \ldots \wedge Cm \qquad m \geq 1$

where each conjunct *Cj* is either an atom (possibly *true*) or an implication of the form

(iii) $B1 \wedge \ldots \wedge Bp \rightarrow E1 \vee \ldots \vee Eq \qquad p \geq 0 , q \geq 1$

where the *conditions Bi* are atomic formulae (possibly *true*) and the *conclusion* has the same form as the body of a definition (as in (i) above). A denial $\neg B$ is represented as an implication of the form $B \rightarrow false$.

The quantification of all variables is implicit. All variables that occur in the head of a definition are implicitly universally quantified. The scope of these universal quantifiers is the entire definition.

All variables in an atomic conjunct *Cj* in a disjunct *Di* in the body that are not in the head are implicitly existentially quantified with scope the disjunct *Di*.

All variables in the conditions of an implicational conjunct *Cj* in a disjunct *Di* in the body that are not in the head and are not in an atomic conjunct of *Di* are implicitly universally quantified with scope the implication *Cj*. The quantification of all other variables occurring in the conclusion of the implication *Cj* is determined recursively as though the conclusion were the body of a definition.

The rules for determining the quantification of variables in definitions ensure that definitions satisfy a generalisation of the **range-restriction** (also called "allowedness" in [8] and elsewhere). The generalisation implements a limited form of constructive negation.

Before we illustrate and define the syntax of beliefs that are observations, we deal with the syntax of goals.

3.2 Goals represented in if-then form

We use "goals", expressed in if-then form, to represent such conceptually diverse objects as

- commands
- queries to a database
- integrity constraints
- condition-action rules
- commitment rules
- obligations
- prohibitions
- sub-goals obtained by means of goal-reduction using definitions in iff-form.

Example 3.4

for-all E [if employee(E) then exists M manager(M,E)]

This goal has the character of a classical integrity constraint, which the database (or beliefs of an agent) must satisfy. It can also be regarded as an obligation on the agent.

As we will see later, at the end of this section, the quantification of variables in this example and in the other examples in the paper can be made implicit.

Example 3.5

if *true* then co-operate

This goal can also be understood as expressing an obligation to co-operate. In general, any implicational goal of the form

If *true* then G

with condition *true* can also be understood equivalently as a condition-less goal

G.

Example 3.6

for-all Time [if do(steal, Time) then *false*]

This goal can be viewed as a prohibition.

Example 3.7

for-all T [if clear(T) then forward(T+1)]
for-all T [if obstacle(T) then right(T+1)]
for-all T [if can-not-see (T) then park(T+1)]

Here the parameter T represents time. This example illustrates the use of goals to represent the effect of condition-action rules.

When goals are used to represent condition-action rules, then the time parameter in the conclusion represents a later time than the time parameters in the conditions. However, the syntax of goals places no constraint on the relationship between time parameters in conditions and conclusions. In particular, it is possible for time in the conclusion to be earlier than time in the conditions. This is often needed for representing certain kinds of integrity constraints; for example "anyone who takes an examination for a course must have registered for the course at the beginning of the term in which the course was given".

However, some syntactically acceptable goals are not meaningful in practice; for example "if you are waiting for the bus for longer than twenty minutes after leaving work, then you should take the underground after leaving work." The onus is on the "programmer" to ensure that the use of time is sensible.

Compare the formulation of example 3.7 with the logic programs of 3.2 and 3.3. Notice that, whereas 3.2 and 3.3 are recursive, 3.7 is not. Arguably, the non-recursive formulation is more natural. This illustrates both another deficiency of conventional LP and an attraction of our extended LP language (as well as an attraction of condition-action rules).

Viewed procedurally, we can see that recursion is unnecessary in 3.7, because the "global" recursion of the agent cycle itself achieves the same effect as the "local" recursion in 3.2 and 3.3. Viewed conceptually, on the other hand, we can see that recursion is unnecessary in 3.7, because the higher level goal (in this case "exploration") is implicit (or "emergent") in 3.7, whereas it is explicit in the logic programs of 3.2 and 3.3.

Thus, goals do not necessarily play the role of complete specifications. Such specifications may be only implicit. For example, the explicit goal

if raining then carry-umbrella

implicitly achieves the higher-level goal

stay-dry

which need not be represented explicitly among the agent’s goals.

Example 3.8

for-all Agent, Act, T1, T2
if happens(ask(Agent, do(Act, T2)), T1)
then exists T, T'
[confirm(can-do(Act, T2), [T, T']) &
do(Act, T2) & T1 < T < T' < T2]

This illustrates the use of goals to represent the effect of commitment rules, as in Agent0. The goal states that if at time T1 an agent asks you to do an action at time T2, then you should first confirm that you can do it, and then do the action at time T2. Here "confirm", like "explore" is a non-atomic action, which has a start time and an end time. It needs to be defined by an iff-definition, which reduces the non-atomic action to atomic actions. In contrast, in Agent0, the conclusion of a commitment rule can only be a conjunction of atomic actions.

Here, for simplicity, it is assumed that the Act that has been requested is a simple atomic act, which can be performed in a single time instance.

The goal in example 3.8 commits the agent to do whatever it is asked. Example 3.9, below, allows the agent also to deal with requests for actions that it cannot perform. It also illustrates the fact that the conclusion of a goal can be a disjunction.

Example 3.9

for-all Agent, Act, T1, T2
if happens(ask(Agent, do(Act, T2)), T1)
then exists T, T', T"
[[confirm(can-do(Act, T2), [T, T']) &
do(Act, T2) & T1 < T < T' < T2] or
[confirm(can-not-do(Act, T2), [T, T']) &
do(tell(Agent, can-not-do(Act, T2), T")) &
T1 < T < T' < T" ≤ T2]]

It is important to realise that, although all of the examples have a natural declarative interpretation, they also have a procedural behaviour, which needs to be efficient. Indeed, because agents need to perform actions in a timely manner, the efficiency that is required of their programs is more demanding than it is for conventional logic programs. In particular, it is important that all agent programs be formulated in such a way that any reasoning/computation that needs to be performed, to generate a commitment to an atomic action, is performed before the action is scheduled for execution.

Formally, a *goal* is an implication of the form

$B1 \wedge \ldots \wedge Bp \rightarrow E1 \vee \ldots \vee Eq \qquad p \geq 0, q \geq 1.$

where the *conditions Bi* are atomic formulae (possibly *true*) and the *conclusion* has the same form as the body of a definition. The special case $true \rightarrow G$
is equivalent to the *goal G.*

As in the case of definitions, all quantification of variables is implicit. The rules for determining the quantification of variables in goals are similar to those for subgoals in the body of definitions.

As we will see later, individual goals are conjoined with observations and other goals in disjuncts of problem statements. Such problem statements have the same syntax as the bodies of definitions, and the rules for determining the quantification of variables in problem statements are the same as those for the bodies of definitions.

The rules for determining the quantification of variables in goals ensure that, like definitions, they satisfy a generalisation of the **range-restriction**. In addition to the fact that this restriction allows quantification to be implicit, it also allows variables to be renamed so that different disjuncts of a problem statement have no variables in common. As a result, the special case of the proof procedure consisting of the two inference rules of forward reasoning and splitting implement the theorem-prover SATCHMO [17].

Note that the beliefs and goals of an agent can incorporate a general theory of action. For example in [20] we present a variant of the event calculus formulated as a theory consisting of iff definitions for persistence and for initiation and termination of properties and of integrity constraints for action preconditions. In [11] we present a similar formulation of a variant of the situation calculus. The agent cycle, the language of beliefs and goals, its proof procedure and its semantics are all neutral with respect to any such action theory.

As we observed following the discussion of example 3.2, it is often possible to avoid the use of action theories altogether, using direct observations of properties in the environment instead. Thus there is a distinction between an agent's reasoning about actions and events, which in a logical approach is non-destructive, and the actual changes themselves that occur in the environment, which can legitimately be destructive.

3.3 Observations

Beliefs include observations, which are given dynamically as inputs. Observations record positive and negative instances of open and closed predicates, representing events that happen in the environment, including actions of other agents, as well as the success or failure of the agent's own actions and properties that are observed to hold.

Positive observations are recorded as simple, variable-free atomic predicates. Negative observations are recorded as variable-free implications with conclusion *false.*

Examples 3.10

employee(mary)

if employee(bob) then *false*
i.e. not employee(bob)

happens(ask(mary, john, do(john, go-home, 10)), 1)
i.e. at time 1 Mary asks John to go home at time 10.

if do(john, go-home, 10) then *false*
i.e. John does not go home at time 10.

Observations can also be instances of closed predicates, as in database *view updates:*

Example 3.11

telephone-is-dead

where telephone-is-dead is a closed predicate defined by

telephone-is-dead iff disconnected or wires-cut or technical-problem.

Here the observation has three alternative, abductive explanations, disconnected, wires-cut or technical-problem.

The representation of incompletely defined, open predicates by means of integrity constraints (goals) has been advocated by Denecker and DeSchreye [5]. Here we use the same syntax for instances of open predicates as we do for goals, including integrity constraints. However, unlike Denecker and DeSchreye, we distinguish observations from goals in the semantics, the proof procedure and the agent cycle.

4. Semantics

The proof procedure described in the next section uses backward reasoning with definitions to reduce an initial problem statement, consisting of goals G and observations O, to an equivalent disjunction of answers to the problem. Each answer $\Delta G \,\&\, \Delta O$ is a formula that is irreducible in the sense that it is expressed in the vocabulary of the open and constraint predicates alone. The component ΔG of the answer contains among its subformulae a plan of atomic actions for accomplishing G, whereas ΔO contains an explanation of O. The explanation ΔO can be used as part of the solution of G, but ΔG cannot be used as part of the explanation of O.

Of course, within the agent cycle, observations (including records of the agent's successful and failed **past** actions, as well as observations of **past** external events and their effects) are added to O incrementally. However, for conceptual simplicity, the semantics and the proof procedure are defined under the assumption that all the observations are given in advance. It is possible to show that the interruption and resumption of the proof procedure to make observations and to perform actions does not affect the semantics (see section 7).

The proof procedure uses forward reasoning to check that the observations are consistent with the goals and subgoals. As we have seen in the examples, such forward reasoning may generate atomic or non-atomic actions for the agent to perform. In the case where the goals represent integrity constraints and the observations are database updates, the generation of these actions actively maintains the integrity of the database.

We now define the top-most level of the semantics of the proof procedure. This top-most level is abstract, in the sense that it is non-committal with respect to a number of such important, lower-level matters as the definition of |= . This is partly because the proof procedure is compatible with many different semantics and partly because we wish to keep the options open at this time.

Let D be a conjunction of definitions in iff-form. Let G be a conjunction of goals in if-then form. Let O be a conjunction of positive and negative observations.

$\Delta G \,\&\, \Delta O$ is an *answer* iff it satisfies all four conditions below:

1. ΔG and ΔO are both conjunctions of formulae in the open predicates and constraint predicates.
2. $D \,\&\, \Delta G \,\&\, \Delta O \models G$
3. $D \,\&\, \Delta O \models O$
4. $D \,\&\, \Delta G \,\&\, \Delta O$ is consistent; i.e. it is not the case that $D \,\&\, \Delta G \,\&\, \Delta O \models$ false.

We have investigated a number of different definitions for |=. All of these interpret P |= C as expressing that C is true in all the intended models of P. The definitions differ, however, in their understanding of the notion of intended model.

In [8], for example, the intended models are the three-valued models of D, together with the Clark equality theory (CET) and definitions in iff-form for the open predicates. The correctness and completeness proofs of the iff-proof procedure of [8] can be adapted to the variant of the proof procedure presented in this paper, relative to the same semantics. A similar semantics and proof procedure, with somewhat weaker completeness results, have been given by [6]. They also discuss the completeness of their proof procedure relative to other semantics.

In [14], D is restricted to be a locally stratified logic program, and the unique perfect model of D & ΔG & ΔO is taken as the only intended model. The proof of soundness of the proof procedure of [14] can also be adapted to the proof procedure of this paper.

The top-most level of the semantics above is also non-committal about what, if any, restrictions are imposed upon the formulae in ΔG and ΔO. A variety of different restrictions have been investigated.

Fung and Kowalski [8], as mentioned above, return as answers iff-definitions of ground positive atoms. These are obtained by first generating a disjunction that is equivalent to the initial problem statement, and such that one of its disjuncts is irreducible. The positive atoms of this disjunct are then instantiated, so they contain no variables and so they are consistent with the other formulae in the disjunct. These ground positive atoms are then rewritten as definitions in iff-form.

Denecker & DeSchreye [6], on the other hand, construct more general answers containing variables. Kowalski *et al*. [14] and Wetzel *et al*.[23] also construct answers containing variables, but these are restricted to conjunctions of atoms and of negations of positive atoms.

In this paper we focus on the agent application of the modified proof procedure and refer the reader to [8], [6], and [14] for more technical details of the semantics.

5. The Proof Procedure

The proof procedure is the thinking component of the unified agent cycle. It combines backward reasoning by means of iff-definitions with forward reasoning by means of goals and observations. Whereas such backward reasoning is used to reduce goals to subgoals in logic programming, forward reasoning is used to check integrity constraint satisfaction in deductive databases [19] and also to check that integrity constraints are consistent [17]. Forward reasoning with the observations also produces condition-action and active-rule behaviour.

The proof procedure is based on that of [7, 8, 14, 23], which, in turn, can be viewed as a hybrid of the proof procedures of Console *et al.* [2] and Denecker and De Schreye [6]. It was originally developed with the goal of unifying abductive logic programming, constraint logic programming and semantic query optimisation. A similar proof procedure has been used for abductive planning in the event calculus [13] by [5], [4] and [9].

Our variant of the proof procedure distinguishes observations from beliefs and goals, thus allowing the proof procedure to be embedded in a dynamic environment.

The goals and observations of an agent are organised into *problem statements*. They have the same form

D1 ∨ … ∨ *Dn*

as the body of a definition. Each conjunct in each disjunct *Di* is either a goal, as described in section 3.2, or an observation, as described is section 3.3. The observations are marked to distinguish them from goals.

All variables belonging to an atomic goal are implicitly existentially quantified with scope the entire disjunct in which the goal occurs. All other variables in the problem statement are quantified in accordance with the rules in section 3.2.

Problem statements are a declarative representation of the search space, containing both alternative courses of action that an agent needs to perform to achieve its goals as well as alternative explanations of the observations.

Example 5.1

The problem statement

[do(drop-out, T) & do(take-drugs, T)] or
[do(study-hard, T) & do(get-good-job, T') & do(get-rich, T'') & T ≤ T' & T' ≤ T'']

represents two alternative courses of action, between which a student might need to choose.

In the propositional case, the *proof procedure* has four inference rules, each of which rewrites a problem statement into another problem statement that is equivalent. The proof procedure terminates when no further inference rules can be applied. The resulting problem statement is then equivalent to the initial problem statement.

When the proof procedure is used for planning, when it terminates, each disjunct of the final problem statement which does not contain *false* as a conjunct contains an alternative plan, which solves the agent's goals with respect to one consistent explanation of the observations.

In addition to the inference rules, the proof procedure needs to be augmented by selection rules and search strategies for deciding what inference rules to apply when. In particular, consistency checking by forward reasoning is not even semi-decidable. Therefore the inference rules need to be applied "fairly" in the sense that no inference step is delayed indefinitely.

Backward reasoning uses iff-definitions to reduce atomic goals (and observations) to disjunctions of conjunctions of sub-goals. It contributes to the implementation of the **D & ΔO |= O** and **D & ΔG & ΔO |= G** parts of the semantics.

Example 5.2

Given the atomic goal "co-operate" and the belief

```
co-operate    iff     for-all Agent, Act, T1, T2
                      if      happens(ask(Agent, do(Act, T2)), T1)
                      then    exists T, T'
                              [confirm(can-do(Act, T2), [T, T']) &
                              do(Act, T2) & T1 < T < T' < T2]
```

backward reasoning reduces the goal to the subgoal

for-all Agent, Act, T1, T2
if happens(ask(Agent, do(Act, T2)), T1)
then exists T, T'
[confirm(can-do(Act, T2), [T, T']) &
do(Act, T2) & T1 < T < T' < T2]

which has the form of a commitment rule. Notice that if the original goal had been this subgoal, then "co-operate" would have been an emergent goal, i.e. a goal not explicitly represented within the agent, but one that could be verified externally.

More formally, **backward reasoning** uses a definition

$G \leftrightarrow D1 \vee ... \vee Dn$

(case 1) to replace a problem statement of the form[1]

$(G \wedge G') \vee D$ by $((D\,1 \vee ... \vee D\,n) \wedge G') \vee D.$

(case 2) to replace a problem statement of the form

$((G \wedge G' \rightarrow D') \wedge G'') \vee D$

by $((D\,1 \wedge G' \rightarrow D') \wedge ... \wedge (D\,n \wedge G' \rightarrow D') \wedge G'') \vee D.$

In both cases if G in the problem statement is an observation then each atomic conjunct in each Di in the resulting problem statement will be marked as an observation.

In case 1, goal reduction needs to be followed by the splitting rule, described below, to put the resulting formula in problem statement form:

$(D\,1 \wedge G') \vee ... \vee (D\,n \wedge G') \vee D.$

Case 2 implicitly uses the equivalence

$((B \vee C) \rightarrow A) \leftrightarrow (B \rightarrow A) \wedge (C \rightarrow A).$

Forward reasoning is a form of modus ponens, which tests and actively maintains consistency, contributing to the implementation of the following part of the specification:

D & ΔG & ΔO is consistent.

It matches an observation or atomic goal	p
with a condition of a non-atomic goal or negative observation	$p \wedge q \rightarrow r$
to derive the new goal	$q \rightarrow r.$

The observation or atomic goal p is sometimes called the "trigger" of the inference " and the condition p in $p \wedge q \rightarrow r$ is sometimes called the "invocation condition".

Any further conditions q can be eliminated either by repeated backward reasoning (goal-to-subgoal reduction) or by further steps of forward reasoning. After all of the conditions $p \wedge q$

[1] Here and elsewhere, we assume the commutativity and associativity of conjunction and disjunction. In particular, when we write a problem statement in the form $(G \wedge G') \vee D$, we intend that $G \wedge G'$ may be any disjunct in the problem statement and that G may be any conjunct in the disjunct.

have been eliminated, the conclusion *r* is added as a new subgoal. In general *r*, itself, has the form of a problem statement. If *r* is an atomic goal, it can trigger forward reasoning or it can be reduced to sub-goals by backward reasoning.

More formally, **forward reasoning** replaces a problem statement of the form

$(G \wedge (G \wedge G' \rightarrow D') \wedge G'') \vee D$

by $(G \wedge (G \wedge G' \rightarrow D') \wedge (G' \rightarrow D') \wedge G'') \vee D$.

Logically, when the implication $(G \wedge G' \rightarrow D')$ represents an integrity constraint or an implication derived from an integrity constraint and *G* represents an update, forward reasoning contributes to integrity verification.

Other inference rules

Splitting uses distributivity to replace a formula of the form

$(D \vee D') \wedge G$ by $(D \wedge G) \vee (D' \wedge G)$

Splitting preserves all the markings of observations.

Logical equivalences rewrite a subformula by means of another formula which is both logically equivalent and simpler. These include the following equivalences used as rewrite rules:

$G \wedge true \leftrightarrow G$	$G \wedge false \leftrightarrow false$
$D \vee true \leftrightarrow true$	$D \vee false \leftrightarrow D$.
$[true \rightarrow D] \leftrightarrow D$	$[false \rightarrow D] \leftrightarrow true$

The resulting formulae *G* and *D* retain all the markings of observations.

Factoring merges two copies of the same atomic goal $G \wedge G$ into one copy of *G*. It also merges an atomic goal *G* with an atomic observation *G* into the observation *G*, thereby distinguishing between the different roles of goals and observations in the specification

D & ΔG & ΔO |= G
D & ΔO |= O

In the non-propositional case, factoring rewrites the formula $P(t) \wedge P(s)$ by the equivalent formula

$[P(t) \wedge s = t] \vee [P(t) \wedge P(s) \wedge s \neq t]$

where *P(t)* and *P(s)* are atomic goals or observations. Both of the atoms in the second disjunct $P(t) \wedge P(s)$ of the result retain their original markings, if any, as observations. However, if at least one of the atoms *P(t)* or *P(s)*, before factoring, is an observation, then *P(t)* in the first disjunct of the result is marked as an observation.

The other inference rules of the proof procedure are defined for the non-propositional case in the usual way, using unification and constraint equalities, as presented in [7,8].

Example 5.3

Let D consist of the following iff definition:

write-reference(X, [T1, T]) iff
[get-file(X, T1) & type-reference(X, T2) & send-reference(X, T) & T1<T2<T] or
[tutor(X,Y) & ask-about(Y, X, T1) & type-reference(X, T2) & send-reference(X, T) & T1<T2<T]

i.e to write a reference for some X in the period of time [T1,T] either get X's file or ask X's tutor about X, then type the reference and send it.

Let G be the conjunction of the following goals:

G1 write-reference(johnson, [1, T])

G2 if get-file(X,T+1) & lost-file(X,T) then *false*

G3 if request(X, meeting, T) & student(X)
then answer(X, T+1) & meet(X, T') & T+1 < T' < T+60

G1 represents the goal of writing a reference for Johnson starting at time 1. G2 represents the integrity constraint that one cannot get a file that is lost. G3 represents an obligation that if a student requests a meeting then one should answer within 1 unit of time and one should meet the student within 60 units of time.

Let O be the conjunction of the following observations (marked by *):

O1 lost-file(johnson, 0)*

O2 request(martins, meeting, 2)*

O3 student(martins)*

O4 tutor(johnson, palmer)*

Alternatively, "tutor" could be represented as a closed predicate defined by means of an iff definition listing all pairs of students and their tutors.

The proof procedure starts with the conjunction of the goals G and the observations O as the problem statement S1. One strategy for selecting conjuncts in the problem statement and for selecting inference rules to apply to the selected conjuncts produces the following derivation:

Reasoning forward with O2 and O3 using G3, the conjunction

answer(martins, 3) & meet(martins, T') & 3 < T' < 62

is conjoined to S1, giving a new problem statement S2.

Reasoning backwards with G1 using the iff definition in D, G1 is replaced by

[get-file(johnson, 1) & type-reference(johnson, T2) &
send-reference(johnson, T) & 1<T2<T] or
[tutor(johnson,Y) & ask-about(Y, johnson, 1) &
type-reference(johnson, T2) & send-reference(johnson, T) & 1<T2<T]

in S2. Splitting then results in a problem statement S3. S3 consists of two disjuncts, which are identical, except that where one of the disjuncts has the goal "get-file(johnson, 1)" the other has the two goals "tutor(johnson, Y) & ask-about(Y, johnson, 1)".

The first disjunct reduces to *false* after two steps of reasoning forward with "get-file(johnson, 1)" and O1 using G2.

After factoring the goal "tutor(johnson, Y) with the observation O4, the second disjunct generates the following answer:

ΔO = O1 & O2 & O3 & O4
ΔG = ask-about(palmer, johnson, 1) & type-reference(johnson, T2) &
send-reference(johnson, T) & 1 < T2 < T & answer(martins, 3) &
meet(martins, T') & 3 < T' < 62.

Alternatively, a ground answer could be extracted from the second disjunct by generating consistent ground substitutions for the variables.

6. The Unified Agent Cycle

The proof procedure described in the previous section is only the thinking component of our unified agent cycle. The top-most level of the cycle has the form:

To cycle at time T,
- record any observations at time T,
- resume the proof procedure, giving priority to forward reasoning with the new observations,
- evaluate to *false* any disjuncts containing subgoals that are not marked as observations but are atomic actions to be performed at an earlier time,
- select subgoals, that are not marked as observations, from among those that are atomic actions to be performed at times consistent with the current time,
- attempt to perform the selected actions,
- record the success or failure of the performed actions and mark them as observations,
- cycle at time T+ n.

An earlier version of this cycle has been formalised in meta-logic in [10]. This formalisation has been used as the basis of Davila's implementation of logic-based agents [4].

The proof procedure within the agent cycle gives preference to reasoning forward from new observations. As mentioned earlier, this preference needs to be balanced by fair application of the inference rules. In addition, other constraints need to be imposed on the proof procedure. For example, before evaluating an atomic action subgoal to *false* in step three, all factoring of the subgoal with observations must first be attempted.

Note that selecting atomic actions involves both choosing an alternative branch of the search space (i.e. a disjunct from the current problem statement) and choosing subgoals (i.e. atomic actions from among the conjuncts of the chosen disjunct). These choices are non-deterministic. At one extreme, they can simply be made at random. At another extreme, they can be made by means of a decision-theoretic analysis of probabilities and utilities.

Note that an atomic action that is selected for execution might not succeed. For example, a "send-email" action might fail if the server is down. In general, attempted actions might fail because the agent has incorrect or incomplete knowledge about the environment.

Although the agent model outlined above does not have intentions as a separate component of an agent's internal state, intentions can be approximated by employing suitable search strategies and selection rules. In particular, the combined use of depth-first, heuristic-guided search and heuristic-guided selection of actions will focus the agent's actions on those belonging to a preferred alternative. This focus will have the effect, at any given time, of identifying the preferred alternative as a plan of action, which serves as the agent's current intentions.

Note also that, not only does the unified agent cycle allow the interruption of thinking and planning to react to observations, but it also allows the selection and execution of atomic actions in partial plans, without first requiring that complete plans be generated.

Example 6.1

Let D, G1, G2, G3 be as in example 5.3. Here we will show how the example works within the agent cycle, when we do not have foreknowledge of all the inputs at the start, but we receive them incrementally.

Assume that the cycle starts at time 0. The problem statement at that time consists of the conjunction of G1, G2, G3. Suppose that we then observe and record observations O1 and O4. These will be conjoined to the problem statement and marked as observations, and the proof procedure is then applied.

Assume that the following inference steps are applied in the following order:

- forward reasoning with O1 and G2
- backward reasoning with G1 and D
- splitting the disjunction that is introduced by backward reasoning
- factoring with O4
- splitting the disjunction that is introduced by factoring.

These inferences result in the following problem statement:

[get-file(johnson, 1) & type-reference(johnson, T2) &
send-reference(johnson, T) & 1< T2 < T &
(if get-file(johnson,1) then *false*) & REST] or

[Y = palmer & ask-about(Y, johnson, 1) &
type-reference(johnson, T2) & send-reference(johnson, T) & 1<T2 <T &
(if get-file(johnson,1) then *false*) & REST] or

[tutor(X,Y) & Y ≠ palmer & ask-about(Y, johnson, 1) &
type-reference(johnson, T2) & send-reference(johnson, T) & 1 < T2 <T &
(if get-file(johnson,1) then *false*) & REST]

where "REST" is the conjunction of G2, G3, O1, O4.

Forward reasoning within the first disjunct generates *false*, which reduces the entire disjunct to *false*, leaving only the other two disjuncts. In the first of these, the equality Y = palmer can be eliminated, resulting in the problem statement:

[ask-about(palmer, johnson, 1) &
type-reference(johnson, T2) & send-reference(johnson, T) & 1 < T2 <T &
(if get-file(johnson,1) then *false*) & REST] or

[tutor(X,Y) & Y ≠ palmer & ask-about(Y, johnson, 1) &
type-reference(johnson, T2) & send-reference(johnson, T) & 1 < T2 <T &
(if get-file(johnson,1) then *false*) & REST]

Suppose the atomic action subgoal "ask-about(palmer, johnson, 1)" is now selected for execution, and assume that it is executed successfully. In that case a record of the successful action, i.e. "ask-about(palmer, johnson, 1)*", will be conjoined to the problem statement. In the next iteration of the cycle, after splitting, this observation will be added to both disjuncts.

Suppose that, in that next iteration of cycle, observations O2 and O3 are made. Giving preference to the new observations, the planning for the reference writing is suspended and the cycle reacts to the new observations. This adds the conjunct

answer(martins, 3) & meet(martins, T') & 3 < T' < 62

to the problem statement. After splitting this conjunct is added to both disjuncts.

During this iteration of the cycle, the following problem statement is generated:

[ask-about(palmer, johnson, 1)* &
type-reference(johnson, T2) & send-reference(johnson, T) & 1 < T2 <T &
(if get-file(johnson,1) then *false*) & REST & O2 & O3 & answer(martins, 3) & meet(martins, T') & 3 < T' < 62]

Here the successfully executed subgoal "ask-about(palmer, johnson, 1)" has been factored with the observation "ask-about(palmer, johnson, 1)*". The other disjunct that results from the factoring step contains false equalities, so that entire disjunct is evaluated to *false*.

The second disjunct, left over from the previous cycle, containing the atomic action subgoal "ask-about(Y, johnson, 1)" has also been evaluated to *false*. This is obtained after first factoring the subgoal with the observation of the successfully executed action. One of the resulting disjuncts after factoring contains the false equalities "johnson ≠ johnson" and "1 ≠ 1", and therefore is evaluated to *false*. The other disjunct is evaluated to *false* by the third step of the cycle, which recognises that the time, "1", of the action is past.

The remaining actions of the problem statement will be selected for execution in future iterations of the agent cycle. However, to avoid the entire problem statement being evaluated to false, the atomic action "answer(martins, 3) " should be selected for execution at time 3. In the case of successful executions of this action and of the remaining atomic actions, the time parameters T2, T and T' will be instantiated with actual execution times.

7. The correctness of the agent cycle

Example 5.3 illustrates the proof procedure in a static environment, whereas example 6.1 illustrates the proof procedure in a dynamic setting where the agent's beliefs are incrementally augmented by observations. The cycle correctly incorporates the proof procedure, in the sense that any problem statement generated dynamically can be generated statically by the same sequence of inferences where the initial problem statement includes foreknowledge of all observations. The following theorem states this more formally. Its proof is straightforward.

Theorem 7.1

Let D be a set of iff definitions and let P be a problem statement. Suppose O is the conjunction of all observations recorded during *m* iterations of cycle, including all successful and failed actions. Then the problem statement at the end of the *m* iterations is identical to the problem statement resulting from the uninterrupted application of the same sequence of inference rules, with the same iff definitions D, starting with the problem statement consisting of the conjunction of P and O.

8. The environment and multi-agent systems

The unified agent cycle of section 6 is a special case of the abstract agent cycle of section 2. Inputs are observations, and outputs are actions performed by the agent. Interactions with other agents are special kinds of interactions with the environment.

This model of inter-agent interaction contrasts with the message-passing model, employed in concurrent logic programming, in which agents interact by passing messages between one another. In the concurrent logic programming model a multi-agent system is represented by means of a single logic program. Interactions between agents are represented by bindings of shared variables, generated by one agent and consumed by another. There is no separate environment and no destructive assignment. Any conceptually separate environment has to be implemented declaratively as another agent.

The concurrent logic programming model contrasts with the unified agent model in which a multi-agent system is a system of individual agents that interact with one another through the medium of the shared environment. Each agent has its own logic program (if any), and the environment that "glues" the agents together is non-linguistic. The environment "grounds" the goals and beliefs of an agent and gives them meaning. In this sense, the environment serves as a privileged model of the agents' beliefs, the model which determines whether or not the agents' beliefs are true.

The environment, like the world around us, is non-linguistic, dynamic and need not remember its past. In this sense, because the environment has no need to remember its past, the transition from one state of the environment to another may be truly and justifiably destructive.

Thus, in our framework, a multi-agent system is a collection of individual agents, interacting through a shared and possibly destructive environment. Such multi-agent systems share with object-oriented programs the fact that the internal states of individual agents are encapsulated, like the internal states of objects. Although some intelligent agents might employ a logic-based, unified agent architecture like that outlined above, other agents might use other internal mechanisms, including ones that are non-logical and even destructive.

We illustrate the interaction of a unified agent in such a multi-agent context in the airline agent example in section 10 below.

9. Comparison with the Rao & Georgeff agent model

In [18] and elsewhere, Rao and Georgeff describe a practical system architecture, which is based upon an abstract BDI agent architecture. Both architectures employ an agent cycle like the one outlined in section 2. Here we formulate, in our extended logic programming language, a variant of the example given in [18]:

if happens(become-thirsty, T)
then holds(quench-thirst, [T1, T2]) & T ≤ T1 ≤ T2 ≤ T+10

holds(quench-thirst, [T1, T2])	iff	holds (drink-soda, [T1, T2]) or holds (drink-water, [T1, T2])
holds (drink-soda, [T1, T2])	iff	holds(have-glass, [T1, T']) & holds (have-soda, [T",T2]) & do(drink, T2) & T1 <T"<T2 ≤T'
holds (have-soda, [T1, T2])	iff	do(open-fridge, T1) & do(get-soda, T2) & T1 ≤ T2
holds (drink-water, [T1, T2])	iff	holds(have-glass, [T1, T']) & do(open-tap, T") & do(drink, T2) & T1<T"<T2 ≤T'

The first sentence represents the goal of quenching thirst within 10 units of time whenever the agent becomes thirsty. Becoming thirsty is interpreted as an observation. Thus, the agent's body is treated as part of the environment.

Notice that "plans" are represented as iff-definitions, which reduce non-atomic actions of the form "holds(P, [T1, T2])" to other non-atomic actions or to atomic actions of the form "do(Act, T)". In this example, as in other BDI examples, planning is "planning from second principles": looking up pre-specified plans to match a pre-identified class of planning problems.

Our formulation of the example, compared with the Rao-Georgeff formulation, is typical of the similarities and differences between the two agent models. Both models employ a similar agent cycle and both distinguish goals and beliefs as separate components of an agent's internal state. However, the Rao-Georgeff model also distinguishes intentions as a separate component.

In the Rao-Georgeff model, there are of two kinds of beliefs: "facts", which record the current state of the environment, and plans, which reduce complex actions to simpler, lower-level actions. Our model allows both these kinds of beliefs, as well as historical records of past observations and arbitrary logic programs.

In the Rao-Georgeff model, goals are conjunctions of positive and negative literals. In our model, they are more general implications. Whereas intentions are distinguished from beliefs and goals in the Rao-Georgeff model, in our model they are treated as goals that represent actions to be performed in the future.

Rao & Georgeff [18] employ two agent languages, one as a modal logic specification language and the other as a procedural implementation language. In contrast, we use the same language for both specification and implementation. We use an explicit representation of time, whereas [18] uses modal operators for time in the specification language, but implicit time in the implementation language.

We distinguish beliefs and goals, as separate components of an agent's internal state. However, [18] distinguishes them by modal operators in the specification language and by meta-predicates in the implementation language.

Whereas our proof procedure distinguishes backward from forward reasoning as separate inference rules, the practical architecture of [18] employs a single procedure for both. The procedure matches a "trigger event" (corresponding to a fact or goal) with the "invocation condition" of a "plan". In those cases where the procedure simulates forward reasoning, the trigger event represents a fact and the invocation condition represents a distinguished condition of an implicational goal. In those cases where the procedure simulates backward reasoning, the trigger event represents a goal and the invocation condition represents the head of a definition.

10. Comparison with the Agent0 agent model

Perhaps the most distinguishing characteristic of Shoham's agent model, Agent0 [21], is that the main part of an agent program consists of commitment rules, similar to the rule of example 3.8. These rules are executed by means of an agent cycle, like the abstract cycle of section 2. Thus Agent0 can be regarded as a generalisation of condition-action rule production systems, in which actions are generalised to commitments to actions to be performed in the future.

Like the unified agent cycle and unlike the Rao-Georgeff cycle, commitments are actions to be performed by the agent at explicitly represented times. The cycle executes all commitments that are scheduled for the current time.

Unlike the unified agent cycle and unlike the Rao-Georgeff cycle, all actions are atomic. The selection of actions is deterministic: all commitment rules whose conditions are satisfied are executed when their times become identical to the cycle time. Unlike the unified agent cycle and unlike the Rao-Georgeff cycle, it is assumed that all selected actions succeed.

Here we formulate, in our extended logic programming language, a variant of the airline agent example given in [21]. Perhaps the most obvious difference between our formulation and Shoham's is that ours employs non-atomic actions, which are ultimately decomposed into atomic actions. In Agent0, all actions are atomic. However, a more important difference is that we employ a separate database of reservations rather than an internal set of commitments to issue boarding passes. There is no reason why an Agent0 formulation might not make similar use of an external database.

```
if      happens(ask(Agent, do(Act, T2)), T1)
then    confirm(can-do(Act, T2), [T, T']) &
        do(Act, T2) & T1 < T < T' < T2

if      happens(present(Pass), T1)
then    check(reservation(Pass, Flight#, T''), [T, T']) & T1+20<T''<T1+120 &
        do(issue-boarding-pass(Pass, Flight#, T''), T2) &
        T1 < T < T' < T2 < T1+10

confirm(can-do(make-reservation(Pass, Flight#, T'), T2), [T, T''])
iff     check(remaining-seats(Flight#, T', N) and
              not reservation(Pass, Other-flight#, T'), [T, T'']) &
        N > 0

check(Query, [T, T''])
```

```
iff     do(ask(database, Query), T) &
        happens(tell(database, affirm(Query, T')), T") &
        T < T' < T" < T+5
```

The first commitment rule deals generally with any request to perform an atomic action. It uses the first iff-definition to deal with a request to make a reservation. It confirms that the reservation can be made, by querying a database, to determine that there are remaining seats on the flight and that the passenger does not already have a reservation on any flight leaving at that time.

In general the querying of the database is a non-atomic action, which is decomposed, by means of the second iff-definition, into the atomic action of asking the query, followed by waiting for an affirmative answer. Waiting is constrained to a maximum of 5 minutes. Once it has been confirmed that the reservation can be made, the first commitment rule generates the commitment to make the reservation, which is an update to the database.

Note that the term

"remaining-seats(Flight#, T', N) and not reservation(Pass, Other-flight#, T')"

names a query to the database. The answer received from the database instantiates the variable N, which is then checked to make sure it is greater than 0.

The second commitment rule deals with the arrival of a customer at the check-in desk. It first queries the database to check that the customer already has a reservation on a flight with a departure time no less than 20 minutes and no more than 2 hours away. It then issues the customer with a boarding pass within a maximum of 10 minutes of the customer's arrival.

Our formulation of the example contrasts with the Agent0 formulation in a number of important respects. As already mentioned, perhaps the most important of these is that in our formulation the reservations database is external to the agent and is queried and updated by atomic actions performed by the agent. In the Agent0 formulation, the database is represented as an internal set of commitments to issue customers boarding passes when they arrive at the check-in desk. Arguably, our formulation is more faithful to conventional implementations, in which a separate, external reservations database is shared by many travel agents.

In our formulation, because the database is conceptually part of the environment, the atomic action of making a reservation can change the database destructively. This contrasts with the pure logic programming approach, where all changes need to be non-destructive.

A somewhat less important difference concerns the treatment of "capabilities", which are conditions that need to be satisfied for the agent to be capable of performing an action. In our formulation, capabilities, defined by iff-definitions, are explicitly represented as actions, such as "confirm" and "check", that need to be performed to ensure that other actions can be performed. In Agent0 the requirement that capabilities be satisfied is implicit rather than explicit. They are defined by a separate, internal database of capabilities.

11. Other related work

We have already compared our approach with the Rao-Georgeff agent model and Shoham's Agent0. Here we give a brief comparison with a number of logic-programming related agent models.

Eiter, Subrahmanian and Pick [24] use agent programs, in the style of logic programming, to generalise condition-action rules. These rules extend logic programming by including deontic modalities to indicate that actions are permitted, forbidden, obliged or waived. They also employ integrity constraints on states and actions. In contrast, we employ a uniform representation of integrity constraints to obtain the effect of condition-action rules, commitment-rules and integrity constraints on both states and actions. We also employ iff-definitions for goal-oriented problem-reduction.

Eiter *et al* [24] employ an agent cycle to generate actions in response to input messages. All reasoning takes place and is completed within a single iteration of the agent cycle. In our agent cycle, on the other hand, reasoning can be interrupted both to accept inputs and to generate outputs. Outputs can be, not only actions generated in response to inputs received in iterations of that cycle, but also actions generated reactively or proactively in earlier iterations.

Wagner [22] employs an agent architecture that contains a number of different components: a knowledge base of beliefs represented in extended logic programming form, a set of tasks having the form of queries, reaction rules encoding the reactive and communicative behaviour of the agent, and action rules giving proactive behaviour. The different components have similar, but different semantics and proof procedures. In contrast, we employ a general-purpose formalism, semantics and proof procedure, to integrate these different functionalities within a unified framework.

Wagner employs a metalogical formulation of the agent cycle in the manner of [10]. Although he acknowledges that planning needs to be interrupted for timely execution of actions, he does not show how this can be done in his approach.

Baral, Gelfond and Provetti [1] use logic programming extended with “classical negation” to implement a modification of the action description language, A [25]. The logic program is used proactively to reduce goals to plans. Thus there is no explicit provision for obtaining condition-action rule reactive behaviour.

Baral *et al* employ an agent cycle in which inputs are either observations or goals. Observations are added to the knowledge base, but are not used as triggers for generating actions. Any goals that have been input are added to the current goals, and some subset of the resulting goals is selected to generate a plan using the action logic program. All planning is performed within a single iteration of the agent cycle. However, a new observation can invalidate a current plan, which then necessitates replanning.

Whereas Baral, *et al* [1] use extended logic programming to implement a variant of the action language, A, Li and Pereira [16] use abductive logic programming to implement A modified to deal with concurrent actions. However, although their abductive programming language allows the use of integrity constraints, Li and Pereira do not use them, as we do, to obtain reactive behaviour. In fact, they focus mainly on classical problems of reasoning about actions, and do not concern themselves with the interaction between reasoning about actions and actually performing them.

Jung [9] builds up on the earlier version of our unified agent architecture [12] and explores in further detail the ability of the architecture to interleave partial, abstract planning with execution of partially constructed plans. Although, like us, he uses the iff-proof procedure [8] to execute abductive logic programs, he does not consider the use of integrity constraints to represent condition-action rules or commitment rules.

12. Conclusions and further work

We have argued in this paper that multi-agent systems are a more powerful and more comprehensive model of computation than logic programming; and we have presented an extension of logic programming that has the potential to provide multi-agent functionality. In addition to including the use of backward reasoning to provide the "rationality" of ordinary logic programs, the extension includes forward reasoning by means of integrity constraints to provide "reactive" behaviour.

Whereas logic programming is not committed to any particular operating system, the extension employs an observe-think-act cycle, which interfaces an agent to its environment. The environment gives meaning to the beliefs and goals of an agent and can change destructively. Agents are object-oriented, in the sense that their goals and beliefs are encapsulated and their interactions with other agents are limited to the effects of their actions on the shared environment.

We have presented a proof procedure and semantics for the extended logic programming language, and we have compared our agent model to Rao-Georgeff's BDI model, to Shoham's Agent0 and to a number of logic-programming related agent models.

We have argued that the unified agent cycle and its proof procedure subsume condition-action rule production systems, SATCHMO, the Rao-Georgeff agent model and Agent0. We also believe that they incorporate much of the functionality of the other, logic programming based approaches. It remains future work to make these arguments and conjectures more precise and more formal.

Further work is also needed on the agent cycle: in particular, to investigate the relationship between passive observations and actions that can be regarded as actively generating observations.

We have investigated the dynamic change of beliefs due to the addition of new observations. However, it is also necessary to investigate more radical changes of belief, including learning, belief revision, and garbage collection of observations and of other beliefs that are no longer needed.

Acknowledgements
The authors are grateful to Dr. Francesca Toni for valuable discussions and to the anonymous referees for helpful comments and suggestions. This research was supported by the UK Science and Engineering Research Council.

References

[1] Chitta Baral, Michael Gelfond and Alessandro Provetti. Representing actions: laws, observations and hypotheses. *Journal of Logic Programming,* 31(1-3) (1997) 201-243.

[2] Luca Console, D. Theseider Dupre, Pietro Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation* 2(5) (1991) 661-690.

[3] Jacinto Davila. REACTIVE PASCAL and the event calculus: a platform to program reactive, rational agents. In *Proc. Workshop on Reasoning about Actions and Planning in Complex Environments*, Bonn, 1996.

[4] Jacinto Davila. Agents in logic programming, PhD thesis, Imperial College of Science, Technology and Medicine, 1997.

[5] Marc Denecker and Danny De Schreye. Representing Incomplete Knowledge in Abductive Logic Programming. In *Proc. International Symposium on Logic Programming*, 1993, 147-163.

[6] Marc Denecker and Danny De Schreye. SLDNFA: an Abductive Procedure for Abductive Logic Programs. *The Journal of Logic programming*, 34 (2) (1998) 111-167.

[7] Tse Ho Fung. A modified abductive framework. In *Proceedings of Logic Programming Workshop*, WLP'94, N. Fuchs and G. Gottlob (eds.), 1994.

[8] Tse Ho Fung and Robert Kowalski. The IFF Proof Procedure for Abductive Logic Programming. *The Journal of Logic programming*, 33 (2) (1997) 151-165.

[9] C. Jung. Situated abstraction planning by abductive temporal reasoning. Proceedings of ECAI 98, Edited by H. Prade, Wiley, 1998, 383-387.

[10] Robert Kowalski. Using meta-logic to reconcile reactive with rational agents. In Meta-Logic and Logic Programming, K. Apt and F. turini (eds.), MIT Press, 1995, 227-242.

[11] Robert Kowalski and Fariba Sadri. The situation calculus and event calculus compared, Proceedings of the International Logic Programming Symposium, Ithaca, New York, Bruynooghe M. (Ed), The MIT Press, November 1994, 539-553.

[12] Robert Kowalski and Fariba Sadri. Towards a Unified Agent Architecture that Combines Rationality with Reactivity. *Proceedings of International Workshop on Logic in Databases*, San Miniato, Italy, Springer-Verlag, 1996.

[13] Robert Kowalski and Marek Sergot. A Logic-based Calculus of Events. In *New Generation Computing*, 4 (1) (1986) 67-95.

[14] Robert Kowalski, Francesca Toni and Gerhard Wetzel. Executing Suspended Logic Programs. *Fundamenta Informatica* 34 (1998) 1-22.

[15] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, R.B. Scherl. GOLOG:A logic programming language for dynamic domains, *Journal of Logic Programming, Special issue in reasoning about action and change*, 31(1-3) (1997) 59-83.

[16] Renwei Li and Luis Moniz Pereira. Representing and reasoning about concurrent actions with abductive logic programs. In *Annals of Mathematics and Artificial Intelligence*, volume 21 (1997), 245-303.

[17] R. Manthey and F. Bry. SATCHMO: A Theorem Prover Implemented in Prolog. Proceedings of the 9th Conference on Automated Deduction (CADE), Argonne, Illinois, 1988.

[18] Anand S. Rao and Michael P. Georgeff. An abstract architecture for rational agents. *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, KRR92*, Boston, 1992.

[19] Fariba Sadri and Robert Kowalski. A Theorem Proving Approach to Database Integrity Checking. In Foundations of Deductive Databases and Logic Programming, J. Minker (ed.), Morgan Kaufmann, 1988, 275-312.

[20] Fariba Sadri and Robert Kowalski. Variants of the event calculus, Proceedings of the International Conference on Logic Programming, Kanagawa, Japan, Stirling L. (Ed), The MIT Press, June 1995, 67-81.

[21] Yoav Shoham. Agent-oriented programming. *AI Journal*, vol. 60, no. 1, (1993) 51-92.

[22] Gerd Wagner. Vivid Agents - How they deliberate, how they react, how they are verified. Extended version of G. Wagner: A Logical And Operational Model of Scalable Knowledge- And Perception-Based Agents. In W. Van de Velde and J.W. Perram (Eds.), Agents Breaking Away, Proc. of MAAMAW'96, Springer Lecture Notes in Artificial Intelligence 1038, 1996.

[23] Gerhard Wetzel , Robert A. Kowalski and Francesca Toni. A theorem-proving approach to CLP. In *Workshop Logische Programmierung*, Krall A., Geske U. (eds.), vol. 270 of GMD-Studien, September 1995, 63-72.

[24] Thomas Eiter, V.S.Subrahmanian and George Pick.Heterogeneous active agents, I. *AI Journal*, vol. 108, no. 1-2, (March 1999) 179-255.

[25] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of logic programming*, vol. 17, (1993) 301-322.