1 Using meta-logic to reconcile reactive with rational agents

R.A. Kowalski

Abstract
In this paper I outline an attempt to reconcile the traditional Artificial Intelligence notion of a logic-based rational agent with the contrary notion of a reactive agent that acts "instinctively" in response to conditions that arise in its environment. For this purpose, I will use the tools of meta-logic programming to define the observation-thought-action cycle of an agent that combines the ability to perform resource-bounded reasoning, which can be interrupted and resumed any time, with the ability to act when it is necessary.

1.1 Introduction

The traditional notion of an intelligent agent in Artificial Intelligence is that of a rational agent that has explicit representations of its own goals and of its beliefs about the world. These beliefs typically include beliefs about the actions that the agent can perform and about the effects of those actions on the state of the world.

This traditional notion of intelligent agent has been challenged in recent years by the contrary notion of an agent that reacts "instinctively" to conditions in its immediate environment. A reactive agent need possess neither an explicit representation of its own goals nor any "world model".

In this paper, I shall outline an attempt to reconcile these two conflicting views. For this purpose, I will extend the notion of knowledge assimilation, Kowalski [Kow79], using the tools of meta-logic programming, to combine assimilation of inputs, reduction of goals to subgoals and execution of appropriate subgoals as actions.

Expressed in informal, simplified, procedural, meta-logic programming style, the definition of the top-most level of a logic-based agent might take the form:

to "cycle" at time T,
  observe one input at time T,
  assimilate the input from time T to $T + m$,
  reduce current goals to subgoals from time $T + m$ to $T + m + n$,
  perform any requisite atomic action from time $T+m+n$ to $T+m+n+1$,
  "cycle" at time $T + m + n + 1$.

The parameters m and n adjust the resources allocated for rational processing of inputs and goals respectively, in relation to those allocated to making observations and performing actions. If the parameters m and n are relatively small, the agent will have little time to "think" before acting. The behaviour will be similar to that of a reactive agent. If m and n are sufficiently large, the agent will be able to generate a complete plan before beginning to act. The behaviour will be similar to that of a traditional, rational agent. For intermediate values of m and n the agent will be able to plan several steps ahead before committing to a particular course of action.

The definition of the *cycle* predicate is similar to the procedural characterisation of a deliberate agent given by Genesereth and Nilsson [GN87]. The most important difference between our procedure and theirs is that we aim to give the definition both a procedural and a declarative interpretation in the spirit of logic programming. As we will see later, a key factor contributing to the declarative interpretation of the procedure is the organisation of inputs and outputs into input and output streams. Both the agent and the agent's environment are interpreted as logically defined processes which communicate with each other by means of these input and output streams. Input and output are symmetric in the sense that the input stream of the agent is the output stream of the environment and vice versa.

Before discussing the *cycle* predicate in greater detail, I will discuss the changes necessary to the more conventional demo and assimilate predicates.

1.2 The proof predicate, demo

The familiar, conventional demo predicate for a Horn clause object language is defined non-deterministically:

$$demo(KB, P) \leftarrow axiom(KB, P \leftarrow Q) \wedge demo(KB, Q)$$
$$demo(KB, P \wedge Q) \leftarrow demo(KB, P) \wedge demo(KB, Q)$$
$$demo(KB, true)$$

Here $demo(KB, P)$ expresses that conclusion P can be demonstrated from "knowledge base" KB; $axiom(KB, P \leftarrow Q)$ expresses that $P \leftarrow Q$ is a clause represented explicitly as an axiom in KB. For simplicity, I use the ambivalent syntax of Kowalski and Kim [KK91] and Jiang [Jia94]. Moreover, I have considered only the propositional case. The general case can be reduced to the propositional case either by using a standard definition of unification, or by adding a clause

$$\begin{aligned} demo(KB, P') \leftarrow\ & demo(KB, forall(X, P)) \\ & \wedge substitute(X, P, Y, P') \end{aligned}$$

as discussed by Kowalski [Kow90]. The predicate $substitute(X, P, Y, P')$ holds when P' results from substituting the *term* Y for the variable X in P.

The definition is non-deterministic because it is not determined, in the first clause of the definition, what axiom in the knowledge base, having conclusion P, might be needed to demonstrate P. The search for the necessary axiom is performed by the non-deterministic "inference engine" which executes the definition rather than by the definition itself.

We can add an extra argument to indicate the resources needed to construct a proof.

$$\begin{aligned}
&demo(KB, P, R+1) \leftarrow axiom(KB, P \leftarrow Q) \wedge demo(KB, Q, R)\\
&demo(KB, P \wedge Q, R+S) \leftarrow demo(KB, P, R) \wedge demo(KB, Q, S)\\
&demo(KB, true, 0)
\end{aligned}$$

However, this argument counts only the number of steps in a proof rather than the number of steps in a search for a proof.

To count the number of steps in the search for a proof, it is necessary to represent the search space explicitly. This can be done by means of a deterministic definition of the *demo* predicate, in which alternative branches of the search space are represented by means of disjuncts:

$$\begin{aligned}
demo(KB, InGoals, OutGoals, R+1) \leftarrow\ & InGoals \equiv (G \wedge Rest) \vee AltGoals\\
& \wedge definition(KB, G \leftarrow D)\\
& \wedge demo(KB, (D \wedge Rest) \vee AltGoals, OutGoals, R)\\
demo(KB, InGoals, OutGoals, 0) \leftarrow\ & InGoals \equiv true \vee AltGoals\\
& \wedge OutGoals = true\\
demo(KB, InGoals, OutGoals, 0) \leftarrow\ & \neg InGoals \equiv true \vee AltGoals\\
& \wedge OutGoals = InGoals
\end{aligned}$$

Here $demo(KB, InGoals, OutGoals, R)$ expresses that the search space of goals, $InGoals$, can be reduced to the search space of subgoals, $Outgoals$, in R steps. For simplicity the definition does not count the resources needed to select the atomic subgoal G and to find its definition $G \leftarrow D$. As we will see in the example definition of $\equiv$ below, the number of steps involved in executing the definition of $\equiv$ may be non-trivial.

The predicate $definition(KB, G \leftarrow D)$ expresses that $G \leftarrow D$ is the complete definition of G in KB. In the general case D is a disjunction of all the conditions of all the clauses having conclusion G. In addition to having its logical meaning as disjunction, $\vee$ can be interpreted as an infix list constructor, terminated by $false$. Thus every disjunction has a final disjunct $false$. In particular, if G is the conclusion of no clause in KB, then D is just $false$. However, if G is abducible, i.e. can be assumed or "made" true, then G has no definition at all. As we will see later, actions which can be performed by the agent are represented by such abducible goals.

Similarly, $\wedge$ can be interpreted as an infix list constructor, terminated by $true$. Thus every conjunction has a final conjunct $true$. In particular, if G is defined by a conditionless clause, then the condition of that clause is taken to be $true$ instead.

The infix predicate, $\equiv$, is any predicate which deterministically expresses the logical equivalence of its two arguments. Procedurally, $\equiv$ can be viewed as selecting both a branch $(G \wedge Rest)$ of the search space and a goal G in the branch. Different deterministic definitions of $\equiv$ give rise to different selection strategies, which in turn give rise to different strategies for searching the search space. For example, the following definition gives rise to Prolog-style depth-first search:

$$
\begin{aligned}
&((D1 \vee D2) \wedge Rest) \vee AltGoals \equiv D' \vee AltGoals' \leftarrow \\
&\qquad\qquad (D1 \wedge Rest) \equiv D' \\
&\qquad\qquad \wedge\ (D2 \wedge Rest) \vee AltGoals \equiv AltGoals' \\
&(false \wedge Rest) \vee AltGoals \equiv AltGoals \\
&((C1 \wedge C2) \wedge Rest) \equiv (C1 \wedge Rest') \leftarrow (C2 \wedge Rest) \equiv Rest' \\
&(true \wedge Rest) \equiv Rest
\end{aligned}
$$

Notice that the first two clauses are like the definition of $append$ for lists constructed using $\vee$; whereas the last two clauses are like the definition of $append$ for lists constructed using $\wedge$. [1]

The infix predicate $=$ is simple identity, defined by the clause $X = X \leftarrow$.

The $demo$ predicate has an argument (the third parameter) which records the state of the search space after the resources allocated to goal reduction have been exhausted. This makes it possible for goals to persist from one cycle to the next, and for execution of the $demo$ predicate to resume when additional resources are made available in later cycles.

1.3 Knowledge assimilation, integrity constraints and goals

The conventional assimilate predicate deals with four cases:

- The input can be demonstrated from the knowledge base, in which case the next state of the knowledge base is identical to the current state.

[1] Notice that the example definition of $\equiv$ does not distinguish between abducible and non-abducible goals. Abducible goals should be treated in $demo$ as though they were true, without actually being replaced by $true$. Catering for abducibles can be done by modifying the definitions of $demo$ and/or of $\equiv$. I leave the details to the reader.

- The input together with one part of the knowledge base can be used to demonstrate the remaining part. In this case the next state of the knowledge base is the input together with the first part of the current state of the knowledge base.
- The input is inconsistent with the knowledge base. Together, the input and the knowledge base need to be revised to restore consistency.
- The input and the knowledge base are logically independent. In this case either the input is added to the knowledge base directly, or, if it is more appropriate, an abductive explanation of the input is added instead.

The formalisation of such knowledge assimilation as a meta-logic program is straightforward, Kowalski [Kow79]. Unfortunately it is quite different from the kind of *assimilate* predicate needed by an active, resource-bounded agent.

An agent needs to process its inputs in real time, both to update its knowledge base and to determine whether any immediate action is required to respond to the input. For example, the representation of the sentence

"If it is raining, carry an umbrella"

should result in the agent attempting to carry an umbrella soon after observing that it is raining.

It is such "online" processing of inputs that is required in the top-level cycle of our agent, rather than the "offline" restructuring of the knowledge base with which the conventional notion of knowledge assimilation is concerned.

In the remainder of this paper, we shall assume that "offline" knowledge assimilation is an activity which takes place either in parallel with the top-level cycle of the agent or at times when the rate of input is very low. We shall focus instead on the resource-bounded assimilation of inputs that needs to take place "online".

The example of carrying an umbrella when it rains is typical of the kind of knowledge an active agent would use to relate conditions it observes to actions it performs. Many other examples readily come to mind:

" In an emergency, press the alarm signal button."

" Give up your seat, if someone else needs it more than you do."

" If it is after 10.00pm and there is no good reason to stay awake, go to sleep."

It is natural to formalise such sentences by means of condition-action production rules. In this paper, we shall explore the possibility that they can be formalised by means of integrity constraints instead. [2]

Integrity constraints in database systems express obligations and prohibitions that all states of a database must satisfy. Such conditions can be expressed by sentences of first-order logic. The obligations and prohibitions associated with such first-order sentences are implicit in the semantics of integrity constraints rather than explicit, as they would be if they were written as sentences of deontic logic, e.g. Jones and Sergot [JS93].

Thus, for example, we might formalise the "obligation" to carry an umbrella when it is raining by the integrity constraint:

$$holds(rain, T) \rightarrow holds(carry(self, umbrella), T)$$

If this were an ordinary sentence in the knowledge base, it would allow the agent to conclude that it is carrying an umbrella whenever it rains, whether it is actually carrying an umbrella or not. As an integrity constraint, however, the sentence imposes an obligation on the ordinary sentences to establish that the conclusion holds, independently of any integrity constraints. This can be done by abducing that some event of putting up an umbrella happens when it first starts raining and by preventing any event of putting down the umbrella while it is still raining.

But it is not enough simply to add to the knowledge base a sentence stating that the agent is performing an action. As we will see in the next section, before the assertion can be added to the knowledge base, the agent needs to output the action to the environment, and the environment needs to confirm that the attempted action has been successful.

Although the distinction between integrity constraints and ordinary sentences is intuitively clear, there have been many different attempts to give the "semantics" of integrity constraints a formal characterisation. For deductive databases, these formalisations include the *theoremhood view*, Lloyd and Topor [LT85], that integrity constraints are theorems that should be logical consequences of the completion of the database, the *consistency view*, Sadri and Kowalski [SK87], that they should be consistent with the completion, and the views that integrity constraints should be understood as *epistemic*, Reiter [Rei90] or *metalevel*, Sadri and Kowalski [SK87] statements about what the database "knows" or can demonstrate.

Despite the differences between these formalisations, the proof procedures that have been developed for verifying integrity constraints generally treat them as goals to be satisfied and are similar in practice. Indeed, it is exactly such a relationship between

[2] This interpretation of production rules in terms of integrity constraints resembles the transformation of Rashid [Ras94]. The exact relationship, however, between our interpretation and this transformation needs to be investigated further.

integrity constraints and goals which we will take as the operational semantics of integrity constraints used for "online" assimilation of inputs. We will regard integrity constraints as passive goals that become active when they are "triggered" by appropriate inputs.

For the sake of simplicity, we will assume that integrity constraints are stored in the knowledge base, but are distinguished from ordinary sentences by their syntax. We will assume, in particular, that all such integrity constraints are written in the form:

$$I \rightarrow C$$

where I is an atomic formula and C is a "complete" conjunction of all the constraints that the knowledge base should satisfy when I holds. Operationally, if an input matches I (unifies with I) then the integrity constraint is "triggered" (resolved with the input) and the appropriate instance of C (resolvent) is added to the current search space of goals. More formally (and more simply, ignoring unification), we can define such online assimilation of inputs by:

$$\begin{aligned} assimilate(InKB, InGoals, Input, OutKB, OutGoals, T) \leftarrow \\ constraint(KB, Input \rightarrow C) \\ \wedge\, OutKB = (Input \wedge InKB) \\ \wedge\, OutGoals = (C \wedge InGoals) \end{aligned}$$

Here the assimilate predicate expresses the relationship that holds between the states of the knowledge base and of the search space of goals, before and after observing *Input* at time T. To simplify the definition of *assimilate*, as a matter of convention, a lack of input is recorded instead as an input of *true*.

The predicate *constraint* expresses that C is a conjunction of all the constraints that should hold when *Input* holds. If there are no such constraints then C is assumed to be *true*. As a result of these conventions

$$constraint(KB, true \rightarrow true)$$

holds as a special case.

The simplified representation of integrity constraints assumes that they have been "precompiled" so that inputs trigger integrity constraints directly in one step without any intermediate deductions. This assumption simplifies the definition of *assimilate* (and *cycle*) because it means that it is unnecessary to record forward deductions from the input that have not yet resolved with the integrity constraints. It also facilitates the agent's ability to process inputs in real time. As a result of this simplifying assumption, assimilation of the input consumes only one unit of resource rather than the more general m units we assumed would be necessary in the Introduction.

Notice that we have also assumed that integrity constraints can be written as implications which have only a single atomic condition. This assumption can be relaxed in various ways; for example, by allowing conclusions C which contain negative literals or implications. Thus the constraint

$$A \wedge B \rightarrow C$$

could be written as

$$A \rightarrow (B \rightarrow C)$$

and/or

$$B \rightarrow (A \rightarrow C)$$

whereas

$$\neg(A \wedge B)$$

could be written as

$$A \rightarrow \neg B$$

or

$$A \rightarrow (B \rightarrow false).$$

Any such relaxation would require appropriate extension of the *demo* predicate. A proof procedure which incorporates many of the features required of such an extension has been developed by Fung [Fun93], following an initial proposal of Kowalski [Kow92]. Further discussion of this matter is beyond the scope of this paper.

1.4 The cycle predicate reconsidered

We can now formulate the recursive clause of the *cycle* predicate more precisely and more formally:

$$\begin{aligned}
&cycle(KB,\ Goals,\ Input.InRest,\ try(Act, Result).OutRest,\ T) \leftarrow \\
&\qquad assimilate(KB, Goals, Input, KB', Goals', T) \\
&\qquad \wedge\ demo(KB', Goals', Goals'', R) \\
&\qquad \wedge\ R \leq n \\
&\qquad \wedge\ try\text{-}action(KB', Goals'', try(Act, Result), KB'', Goals''', T+R+2) \\
&\qquad \wedge\ cycle(KB'', Goals''', InRest, OutRest, T+R+3)
\end{aligned}$$

Our intention is that the definition should have both procedural and declarative readings. The procedural reading requires the *cycle* predicate to be executed as a process concurrently with an environment process. The declarative reading specifies what sequences of inputs and outputs constitute acceptable behaviour.

Here the inputs and outputs are represented by an input stream and output stream respectively. The first item, $Input$, in the input stream represents the actual input at time T. The first item, $try(Act, Result)$, in the output stream represents the next output, if any, attempted at time $t + R + 2$. If the action Act is not "nil". then it is "tried" at the moment the output item is "consumed" by the environment. At the same time, the environment instantiates the variable $Result$ indicating whether the attempted action succeeds or fails.

The time parameter, T, is local to the agent and behaves as an internal clock used to "time stamp" inputs and outputs when they are recorded in the knowledge base. For the sake of simplicity, we have assumed that time is measured in terms of inference steps, and that each inference step takes one unit of time. For simplicity, we have also assumed that observing and assimilating the input together take two time units, whereas trying an (atomic) action takes only one.

The constant n is the amount of resource available for goal reduction, whereas R is the actual resource consumed. Notice that, for the *cycle* predicate to execute efficiently as it is written, the condition $R \leq n$ needs to be executed as a constraint or to be coroutined with the execution of the *demo* condition. In a more elaborate version of the *cycle* predicate, n might be computed by the agent, varying in a manner which is appropriate to the circumstances.

The *try-action* predicate analyses the search space of goals $Goals''$ to determine whether it contains any action which the agent can try to execute. There are three cases:

- There is such an action, and the attempt to execute it succeeds. In this case, the agent commits to the branch of the search space containing the action.
- There is such an action, but the attempt to execute it fails. In this case the branch containing the action is discarded.
- There is no such action, in which case the search space is unchanged.

The *try-action* predicate can be defined more formally as follows:

$$
\begin{aligned}
&try\text{-}action(KB, Goals, try(Act, Result), KB', Goals', T) \leftarrow \\
&\qquad\qquad action\text{-}needed(Goals, Act, Rest, AltGoals, T) \\
&\qquad\qquad \wedge\ Result = success \\
&\qquad\qquad \wedge\ Goals' = Rest \\
&\qquad\qquad \wedge\ KB' = (do(self, Act, T) \wedge KB)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{try\text{-}action}(KB, Goals, try(Act, Result), KB', Goals', T) \leftarrow \\
&\qquad \mathit{action\text{-}needed}(Goals, Act, Rest, AltGoals, T) \\
&\qquad \wedge\ Result = failure \\
&\qquad \wedge\ Goals' = AltGoals \\
&\qquad \wedge\ KB' = KB \\
&\mathit{try\text{-}action}(KB, Goals, try(nil, nil), KB', Goals', T) \leftarrow \\
&\qquad \neg\exists Act, Rest, AltGoals\ \mathit{action\text{-}needed}(Goals, Act, Rest, AltGoals, T) \\
&\qquad \wedge\ Goals' = Goals \\
&\qquad \wedge\ KB' = KB \\
&\mathit{action\text{-}needed}(Goals, Act, Rest, AltGoals, T) \leftarrow \\
&\qquad Goals \equiv (do(self, Act, T) \wedge Rest) \vee AltGoals
\end{aligned}
$$

Here the goal, $do(self, Act, T)$, is abducible in the sense that it has no definition in KB (before it is added to the knowledge base, in the case that the attempt to execute *Act* succeeds). Alternatively, it may be viewed as having a definition ($true$ in case $Result = success$, $false$ in case $Result = failure$) which is held externally in the environment. This second view is similar to that of query-the-user, Sergot [Ser83].

Operationally, the calls to the predicate *action-needed* in the first two clauses select one branch of the search space from among all the branches containing an action which the agent can try to execute at that time. This instantiates the variable *Act* to a concrete value. The partially instantiated term $try(Act, Result)$ where *Result* is a variable, is now available as an output on the output stream to the environment.

For the *cycle* predicate to behave effectively, it needs to be executed as a process concurrently with other processes corresponding to the environment and to other agents. The agent and the environment communicate with one another by means of terms in input and output streams. The input stream of the agent is an output stream of the environment; and, conversely, the output stream of the agent is an input stream of the environment. As in concurrent logic programming languages, a process can send a message, such as $try(Act, Result)$, containing a variable, which is instantiated by the process receiving the message. In this case, the environment process instantiates the variable, *Result*, to one of the values $success$ or $failure$.

If the result of the attempted action is $success$, then the agent commits to the action and discards the alternative branches of the search space. This commitment is similar to the non-deterministic commitment of concurrent logic programming languages. It is tempting to speculate that the declarative reading of the *try-action* predicate definition, embedded within the definition of the *cycle* predicate, might give a (meta)logical semantics to such non-deterministic commitment.

The *try-action* predicate combines committed choice, when actions succeed, with search, when actions fail. The second clause in the definition deals with the case when actions fail. The branch (being a conjunction equivalent to $false$) containing the selected and failed action is discarded and the agent enters the next cycle with the search space consisting of the remaining alternative branches.

Notice that the definitions of the *demo* and *action-needed* predicates are neutral with respect to the search strategy used to select branches in the search space. Thus backtracking upon failure is only one of the many possible search strategies covered by our definitions.

Another possibility is to employ an evaluation function to evaluate alternative courses of action represented by alternative branches of the search space. The same evaluation function could be used both to direct the search towards the most promising part of the search space in the definition of *demo* and to select the most promising next action in the definition of *try-action*. Such use of an evaluation function would need to be taken into account in determining the total amount of resources consumed by the agent within a given cycle.

As we will see in greater detail in the next section, the branch

$$do(self, Act, T) \wedge Rest,$$

whose first subgoal is selected for attempted execution at time T, represents a partial plan for accomplishing the agent's goals. The action associated with the first subgoal represents the first step of the plan; $Rest$ represents the remainder of the plan. Depending upon how much resource is available for generating the plan before the first action is needed, $Rest$ will contain more or less detail about the rest of the plan. The less resource, the less detail; and the more the agent behaves reactively. The more resource, the more detail; and the more the agent behaves deliberately.

The advantage of deliberation is that it allows the agent to look ahead, compare alternative partial plans and try the most promising alternative. In many cases it can avoid trying an unproductive action by foreseeing that it would eventually lead to failure. The disadvantage is that in many situations the need to perform an action is so urgent that there simply is no time for such deliberation. Moreover, when the future is unpredictable, planning can be a waste of time.

The value of the parameter n determines the balance between deliberation and reactivity. As we have already remarked, it might be useful for this parameter to be computed and for its value to depend upon the circumstances. But the balance between deliberation and reactivity also depends upon the kind of knowledge that is represented in the knowledge base, as we will see in the following section.

1.5 Knowledge representation matters

The feasibility of the agent architecture outlined above depends crucially upon the way in which knowledge is represented in the knowledge base. It must be represented, in particular, in such a way that the backward reasoning performed by the *demo* predicate generates plans in a forward direction, starting with an action that can be performed in the current state. Conventional logic-based representations of actions and their effects behave instead in such a way that backward reasoning corresponds to reasoning backwards in time while forward reasoning corresponds to reasoning forwards in time.

Consider, for example, the goal of going from one location to another. A typical logic-based representation, of the kind normally associated with the situation calculus of McCarthy and Hayes [MH69] or the event calculus of Kowalski and Sergot [KS86], might employ, along with frame axioms or persistence axioms, a clause of the following simplified form:

$$\begin{aligned} holds(loc(Agent, Y), T+1) \leftarrow\ & holds(loc(Agent, X), T) \\ & \wedge\ next\text{-}to(X, Y) \\ & \wedge\ holds(clear(Y), T) \\ & \wedge\ do(Agent, step(X, Y), T) \end{aligned}$$

Using such a sentence in the definition of the *demo* predicate to reduce goals to subgoals would generate plans backwards, starting from the last action to the first. The agent would not be able to execute any action until it generates a complete plan.

What we need instead is a representation such as:

$$\begin{aligned} go(Agent, X, Z, T) \leftarrow\ & holds(loc(Agent, X), T) \\ & \wedge\ next\text{-}to(X, Y) \\ & \wedge\ holds(clear(Y), T) \\ & \wedge\ T' \leq T + n' \\ & \wedge\ do(Agent, step(X, Y), T') \\ & \wedge\ go(Agent, Y, Z, T') \\ go(Agent, X, X, T) \leftarrow\ & holds(loc(Agent, X), T) \end{aligned}$$

which includes both the current state and goal state in the same predicate. Here n' is a parameter (sufficiently larger than $n+3$) regulating the rate of movement from one location to the next. Using such a representation backwards to reduce goals to subgoals generates plans forwards, starting from the first action in a plan. It can be interrupted any time after the first action has been generated, to try executing that action, even if no part of the rest of the plan has yet been generated.

Notice that even with this representation, however, conventional axioms such as

$$\begin{aligned} holds(loc(Agent, Y), T2) \leftarrow\ & do(Agent, step(X, Y), T1) \\ & \wedge T1 < T2 \\ & \wedge \neg\exists T^*[do(Agent, step(Y, Z), T^*) \wedge\ T1 < T^* < T2] \end{aligned}$$

of the kind used in the event calculus, are still required (for example, to solve the condition $holds(loc(Agent, X), T)$ in the definition of go). The crucial matter is that the goal of the agent's going at time T to a destination Z from its current location X should be represented as

$$go(self, X, Z, T)$$

rather than as

$$holds(loc(self, Z), T).$$

Notice, however, that, although the definition of go might work in theory, it gives rise to a brute-force search, starting from the current location X, that will not work in practice. With this representation, unless the value of n is sufficiently large to allow the generation of a complete plan, the agent will move about at random, totally ignoring the destination Z.

For the agent to behave effectively, the choice of the location Y next to X in the definition of go needs to take the destination Z into account. In traditional AI approaches this is done by using heuristic functions to evaluate alternatives in the search space and to select more promising alternatives in preference to less promising ones. In expert system approaches, on the other hand, such knowledge is more commonly incorporated into the object-level knowledge base itself. In our case this can be done simply by adding an extra condition, $towards(X, Y, Z)$, to the definition of go to restrict the choice of next location Y to one whose distance to the destination is closest among all the locations next to X which are clear at time T. This extra condition can be specified in the form:

$$\begin{aligned} towards(X, Y, Z) \leftarrow \forall Y'[&next\text{-}to(X, Y') \wedge holds(clear(Y'), T) \rightarrow \\ & dist(Y, Z) \leq dist(Y', Z)] \end{aligned}$$

which can be converted into conventional logic programming form in standard ways.

1.6 Logic-based multi-agent systems

A preliminary version of the resource-bounded, logic-based agent architecture described above has been implemented in a multi-agent environment by Davila [DQ94]. Several agents are placed at various initial locations on a rectangular grid and are given the goal of going from their initial locations to different destinations.

Given a grid with no obstacles, except for those created by one agent temporarily blocking another, the implementation confirmed our expectation that planning confers no advantages over purely reactive behaviour. This is because, without an agent having a sophisticated model of the behaviour of other agents, in this environment the obstacles created by agents blocking one another are totally unpredictable.

The implementation was written in a combination of Prolog, used for programming the logic-based cycle of the individual agents, and April, for programming the interactions between the agents and the environment. April, McCabe and Clark [MC94], is a process-oriented symbolic language, which has grown out of the experience of using concurrent logic programming languages such as Parlog. The discussion of the previous section of this paper, showing how inputs and outputs can be implemented using input-output streams containing variables, suggests that an implementation combining Prolog and a concurrent logic programming language might be more appropriate from a logical point of view.

Although planning had no value in our simple multi-agent experiment, we anticipate that it will be important in other applications where predictions can be made reliably. To predict the future, an agent needs to model other agents as well as to communicate with other agents both to avoid conflicts and to achieve common goals.

It was in fact for the purpose of modelling other agents that we earlier proposed the use of meta-logic programming to solve the puzzle of the three wise men, Kowalski and Kim [KK91]. More recently, we have begun to investigate the use of argumentation to resolve conflicts between different agents, Kowalski and Toni [KT94].

1.7 Conclusions and future work

The proposal outlined in this paper is a first step towards the development of a resource-bound, logic-based agent architecture. Although it is firmly based on the use of logic both at the object level to represent domain knowledge and at the metalevel to control the observation-thought-action cycle, it makes important concessions to the anti-logic, reactive agent school. In particular, it concedes, that when the future is unpredictable, rational planning is not only a waste of time but interferes with the ability to act effectively and in a timely manner.

None the less, compared with purely reactive architectures, our logic-based agent model can exploit reliable knowledge about the future, to avoid short-term actions that ultimately and predictably fail to achieve long term goals. Moreover, it can be extended to make the future more predictable, both by exploiting meta-logic to reason about other agents and by allowing agents to communicate with each other to negotiate co-ordinated plans of action. Integrating such extensions with the simplified agent model outlined in this paper is an important direction for future research.

There are at least two other important extensions to be considered. One is to investigate how several agents can be combined so that to an external observer they behave as though they were a single agent. The other is to investigate how goal-oriented behaviour can emerge as a property of the behaviour of a single agent or collection of agents.

It seems that much of the work needed for the first of these extensions has already been done in the work on using meta-logic for combining theories Brogi et al. [BMPT94, BT95]. This needs to be developed further to take integrity constraints, goals, subgoals and actions into account.

The second of these extensions seems to be related to the use of integrity constraints to obtain the behaviour of condition-action rules. This, in turn, is related to logic programming representations in which conclusions of implications represent goals. These relationships need to be investigated further to take account of the properties that emerge when agents interact in multi-agent systems.

Acknowledgements

This work was partly supported by Fujitsu Laboratories. I am grateful to Krzysztof Apt, Jacinto Davila, Murray Shanahan and Franco Turini for their helpful comments on an earlier draft of this paper, and to Rodney Brooks and Alan Macworth for alerting me to the importance of the problem investigated in this paper.

Bibliography

[BMPT94] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, 1994.

[BT95] A. Brogi and F. Turini. Fully abstract compositional semantics for an algebra of logic programs. 1995. To appear in *Theoretical Computer Science*.

[DQ94] J.A. Davila Quintero. *Knowledge assimilation in multi-agent systems*. MSc. Thesis, Imperial College, London, 1994.

[Fun93] T. H. Fung. *Theorem proving approach with constraint handling and its applications on databases*. MSc. Thesis, Imperial College, London, 1993.

[GN87] M.R. Genesereth and N.J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann Publishers Inc., 1987.

[Jia94] Y. Jiang. Ambivalent logic as the semantic basis of metalogic programming: I. In *Proc. Eleventh International Conference on Logic Programming*, pages 387–401, 1994.

[JS93] A. Jones and M. Sergot. On the characterisation of law and computer systems: the normative systems perspective. In J.-J.Ch. Meyer and R.J. Wieringa, editors, *Deontic logic in computer science: normative system specification*, chapter 12. Wiley, 1993.

[KK91] R.A. Kowalski and J.S. Kim. A metalogic programming approach to multi-agent knowledge and belief. In Lifschitz V., editor, *Artificial intelligence and mathematical theory of computation*, pages 231–246. Academic Press, 1991.

[Kow79] R.A. Kowalski. *Logic for problem solving*. Elsevier, New York, 1979.

[Kow90] R.A. Kowalski. Problems and promises of computational logic. In J.W. Lloyd, editor, *Proc. Symposium on Computational Logic*. Springer Verlag Lecture Notes in Computer Science, 1990.

[Kow92] R.A. Kowalski. A dual form of logic programming. 1992. Lecture Notes, Workshop in Honour of Jack Minker, University of Maryland.

[KS86] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[KT94] R.A. Kowalski and F. Toni. Argument and reconciliation. In *Proc. Workshop on Legal Reasoning, International Symposium on Fifth Generation Computer Systems*, Tokyo, Japan, 1994.

[LT85] J.W. Lloyd and R.W. Topor. A basis for deductive database system. *Journal of Logic Programming*, 4:93–109, 1985.

[MC94] F. McCabe and K.L. Clark. April – agent process interaction language. In *Proc. ECAI94 Workshop on Agent Theories, Architectures, and Languages*, 1994. To be published by Springer Verlag.

[MH69] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[Ras94] L. Rashid. A semantics for a class of stratified production system programs. *Journal of Logic Programming*, 21(1):31–57, 1994.

[Rei90] R. Reiter. On asking what a database knows. In J.W. Lloyd, editor, *Computational Logic*, pages 96–113. Springer Verlag, Esprit Basic Research Series, 1990.

[Ser83] M. Sergot. A query-the-user facility for logic programming. In Degano and Sandwell, editors, *Integrated interactive computer systems*, pages 27–41. North Holland Press, 1983.

[SK87] F. Sadri and R.A. Kowalski. An application of general purpose theorem-proving to database integrity. In J. Minker, editor, *Foundations of deductive databases and logic Programming*, pages 313–362. Morgan Kaufmann, 1987.