# Representing and Executing Agent-Based Systems

Michael Fisher

Department of Computing
Manchester Metropolitan University
Manchester M1 5GD
United Kingdom

M.Fisher@doc.mmu.ac.uk

**Abstract.** In this paper we describe an approach to the representation and implementation of agent-based systems where the behaviour of an individual agent is represented by a set of logical rules in a particular form. This not only provides a logical specification of the agent, but also allows us to directly execute the rules in order to implement the agent's behaviour. Agents communicate with each other through a simple, and logically well-founded, broadcast communication mechanism. In addition, agents can be *grouped* together. This not only restricts the extent of broadcast messages, but also provides a structuring mechanism within the agent space.

The purpose of this paper is threefold: to motivate the use of our particular computational model; to show that a logic-based approach is both possible and potentially very powerful; and to assert that by narrowing the gap between the agent theory and the agent programming language, we are able to bring the prospect of formal specification and verification of multi-agent systems nearer.

## 1 Introduction

Multi-Agent Systems are being developed and applied in a variety of contexts, from traditional AI, through Operating Systems and concurrent programming languages, to Robotics and Artificial Life. In spite of this, not only is there little agreement on what the core attributes of a general computing agent should be, but also there is, as yet, little evidence of an *engineering* approach to the development of multi-agent systems. For example, both development methods and verification techniques for multi-agent systems are rare. In fact, the definitions of agents are often so difficult to understand that they (a) cannot be effectively analysed by humans, and (b) are too complex for verification purposes.

In this paper, we outline an approach to the representation, development and implementation of multi-agent systems that we are developing. This is based upon the notion of concurrent object-based systems where each object executes a temporal specification of its behaviour whilst communicating with other objects using a form of *broadcast* message-passing.

In this paper we will address some issues relating to our approach, in particular:

– we will motivate the use of our particular computational model, by showing how the combination of executable specifications, broadcast message-passing and object

grouping is able to represent a range of behaviours of both individual agents and multi-agent systems;
- we show that the use of a logic-based language in the description and implementation of agents is not only desirable but, as we use a *temporal logic*, is also potentially very powerful, particularly since the logic contains simple elements that are fundamental to the representation of dynamic behaviour;
- we outline how this work, combining executable specifications within a powerful logical framework, not only narrows the gap between the theory and implementation of multi-agent systems, but also provides a further step towards the goal of formal specification, verification and development of such systems.

In §2 and §3 we describe our approach to the modelling of individual agent behaviours and agent communication and grouping respectively. In §4, we give a brief outline of an implementation mechanism for these logic-based descriptions. Applications of this approach are indicated in §5, while the work towards the formal development of multi-agent systems is described in §6. Finally, in §7, we summarise the general utility of our approach and comment on related work.

## 2 Representing Agent Behaviour

In this section, we will motivate our approach to the representation of behaviour within individual agents. This will encompass both an outline of the advantages of a general logic-based approach and a more specific description of the benefits of using *temporal logic* as as a notation for representing dynamic activity.

First of all, however, we will outline the context in which we view individual agents.

### 2.1 Agents as Objects

An 'agent' is typically described as "an encapsulated entity with 'traditional' AI capabilities". We take the view that the distinction between an object, as used in concurrent object-based systems, and an agent, as defined above, is so vague and flexible to be useless. Hence, we believe that a multi-agent system is simply a system consisting of concurrently executing objects. Although some researchers, for example Maruichi et. al. [17], have attempted to distinguish agents from objects by stating that agents have control over their own execution (i.e., have meta-level capabilities) while objects do not, we take the view that everything is an object and, while many objects have some degree of meta-level control, there is nothing special that distinguishes an 'agent' from any other object. However, for the sake of this discussion, we will refer to objects that exhibit some form of 'intelligence' as 'agents' (although we must recall at certain times that the system is not able to distinguish between 'intelligent' and 'dumb' objects — they are all treated the same).

### 2.2 Requirements for Agent Description Languages

In representing the internal behaviour of an individual agent, it can be argued that we require a notation satisfying some, if not all, of the following criteria.

– It should be high-level, yet concise, consisting of a small range of powerful constructs.
– It should possess a semantics that is both intuitive and, if possible, obvious from the syntax of the language.
– It should be able to represent not only the static, but the *dynamic*, behaviour of agents.
– It should impose as few operational constraints upon the system designer as possible (for example, concurrent activities within a single agent should be able to be defined).

In representing an individual agent's behaviour, we choose to utilise a formal logic. One of the advantages of following such an approach is that the notation has a well-defined, and usually well understood, semantics. The use of a formal logic language also allows us to narrow the gap between the agent descriptions and the agent theory in that the semantics of an agent is close to that of its logical description. This allows for the possibility of employing both specification and verification techniques based upon formal logic in the development of multi-agent systems.

### 2.3 Representing Dynamic Behaviour Using Temporal Logic

While a general logic-based approach satisfies many of the above criteria, we choose instead to use temporal logic as the basis of our formal description of agent behaviour. Temporal Logic is a form of non-classical logic where a model of time provides the basis of the system. In our case, a simple discrete, linear sequence of states is used as the basic temporal model. Each state in this temporal sequence can be thought of as a model for classical logic. Such a temporal logic is more powerful than the corresponding classical logic and is useful for the description of dynamic behaviour in reactive systems. As such, it has many advantages as a description technique for agents in multi-agent systems, some of which we outline below.

– The discrete linear model structure that is the basis of the logic is very intuitive matching, as it does, steps in an execution sequence with an identified starting state and an infinite execution.
– The logic contains the core elements for describing the behaviour of basic dynamic execution. For example, it contains three main descriptive elements:
  - a declarative description of the current state;
  - an imperative description of transitions that might occur between the current and the *next* states;
  - a description of situations that will occur at some, unspecified, state in the future.

  Thus, using this logic, we are able to describe the behaviour of an agent now, in transition to the next moment in time and at some time in the future.
– This basic set of concepts is sufficient as more complex temporal properties can be translated into a normal form consisting of these concepts [13]. Thus, a general temporal specification can be given and transformed into a set of rules of this basic form.

Of particular importance, both in the representation of dynamic behaviour and in the execution of such temporal formulae, is the simplicity of the logic. In spite of this simplicity, it contains three core elements that are characteristic of general computation: the state of the agent at the current moment; the transitional constraints upon an agent as it moves forward to the next moment in time; and temporally indeterminate properties which must become true at some point in the agent's future. Thus, we can think of these three core elements as expressing the current state of an agent, what it can do next, and what it's longer term goals are.

We will not provide a detailed description of the temporal logic used, nor of the exact rules used in our approach (for a more detailed description, see [9]). Rather, we will present a simplified version as follows. The behaviour of an agent, both static and dynamic, is represented by a set of temporal *rules*. Each rule is of the form

'formula about the present' $\Rightarrow$ 'formula about the future'

Simple examples include

$$(p \wedge q) \;\Rightarrow\; \bigcirc \neg r$$

representing the rule that if both $p$ and $q$ are true in the present state, then $r$ must be false in the next state, and,

$$s \;\Rightarrow\; \Diamond t$$

representing the rule that if $s$ is true now, then $t$ must be true at *some* time in the future.

The form of these rules is such that any formula in the logic can be rewritten into a set of rules consisting of initial constraints (which are true at the start state) and rules of one of the following two forms [13].

$$\bigwedge_{i=1}^{m} p_i \Rightarrow \bigvee_{j=1}^{n} \bigcirc q_j$$

$$\bigwedge_{i=1}^{m} p_i \Rightarrow \Diamond r$$

Here, $p_i$, $q_j$ and $r$ are simply literals.

Note that, since any formula of the logic can be rewritten as a set of rules in this form, this does not represent a restriction of the logic such as Temporal Horn Clauses [1] — the full power of temporal logic can be utilised. Although we will see later that some additional operational constraints are used, the simple form of the rules means that the behaviour of the system is usually obvious from its description.

Finally, the temporal rules themselves can be seen to capture a mixture of both declarative and imperative aspects of the system. They comprise a declarative description of the state of the agent (classical logic), an imperative description of allowable transitions between states (e.g. '$\bigcirc$') and a declarative description of things that must happen at some time in the future (e.g., '$\Diamond$').

## 3 Agent Communication

We now turn to the representation of communication between agents. As we have argued above, multi-agent systems should be essentially based upon a form of concurrent object-based system. Thus, the key attributes of our system are those of objects (or agents) and message-passing. However, though most concurrent object-based systems employ point-to-point message-passing as the basic communication mechanism, we will argue below that such an approach restricts the power of multi-agent systems. In particular, we advocate a combination of agent grouping structures and broadcast message-passing in order to achieve communication that is both flexible and adaptable, yet is not prohibitively inefficient.

### 3.1 Point-to-point Message-Passing

Point-to-point message-passing is widely used in concurrent object-based systems, for example in actor systems [2]. Messages are sent to a specific address (the 'receiver') which must be known by the sender. The advantages of this approach are:

- an agent 'knows' where a message is being sent to;
- security controls are easily introduced as an agent can ensure that important information is never sent to the 'wrong' agents;
- this style of message-passing is both common and efficiently implemented in concurrent object-based systems.

The disadvantages of this simple form of message-passing are:

- an agent *must* 'know' where a message is to be sent to;
- in such a framework, it is hard to model *open* systems, for example the disappearance of the message recipient is usually problematic.

A final disadvantage, more specific to logic-based approaches, is that this form of communication does not fit naturally into the view of computation as a limited form of theorem-proving. Thus, we have examined other approaches to communication in an attempt to find a mechanism that is both more natural and more flexible than point-to-point message passing.

### 3.2 Broadcast Message-Passing

Broadcast message-passing is a natural mechanism to consider as it not only matches the logical view of computation that we utilise, but it is becoming more widely used in distributed computer systems. Broadcast message-passing involves sending a message, not to a specific address, but to all agents at once. It's advantages are:

- it is *compositional* in that a particular agent in a system can be replaced by another agent having observationally equivalent behaviour and the behaviour of the system as a whole will be unchanged (even though the name/address of the agent *has* changed);

- it is ideal for systems where tasks are announced and agents either compete or cooperate for the ‘contract’, for example Contract Nets [21];
- it is widely used in adaptable and fault-tolerant systems, for example in distributed operating systems, agents that ‘die’ can be replaced by ‘shadows’ allowing the system to continue even if certain processors fail [5].

In spite of these advantages, for many years broadcast message-passing has been avoided due to its perceived disadvantages:

- broadcast is not *secure* — any agent can examine the contents of any message;
- in a distributed system, broadcast has been perceived as *prohibitively* inefficient;
- it is perceived as being difficult to program with.

### 3.3 Agent Groups and Multicast Message-Passing

An obvious way to avoid some of the perceived problems of broadcast, and one that has been developed both within distributed operating systems and DAI is that of structuring the agent space into *groups*. Thus, each agent is a member of at least one group and if that agent broadcasts a message, it will be sent to all members of one or more (depending on the system) of the agent’s groups. In such a way, full broadcast message-passing is replaced by *multicast* message-passing. In distributed operating systems, such an approach, called *process groups* is used to ensure replication and fault-tolerance [6], while in DAI it is used, for example in the *organisational model*, as a method for structuring and organising agents [17].

Thus, by utilising a form of group structuring, together with broadcast message-passing, we are able to retain many of the advantages of full broadcast while avoiding its major drawbacks. A bonus is that, in recent years, low-level mechanisms for efficient broadcast have been developed in many computer systems [4]. Further, not only is broadcast one of the basic communication mechanisms on local area networks [19], but also the advent of novel parallel architectures (e.g. data parallelism [14, 15]) has meant that more powerful programming techniques based upon broadcast communication are beginning to be developed.

## 4 Implementing Agent Behaviour

Having considered both the representation of behaviour within individual agents, and the communication mechanism between agents, we will now describe how these behaviours are *implemented* is our system. Whilst using a logic-based language gives us a clear link between the agent theory and the agent description language, we may not be certain that the language is implemented according to the semantics of the logic. One way to ensure this is to directly execute the logical statements. This move towards executable logic specifications further narrows the gap between the actual implementation of the language and the theory underlying the system.

Unfortunately, one particular problem with executable logics in general is that as the expressiveness of logic being used increases, so the execution mechanism becomes correspondingly more expensive to implement. This is one of the reasons that the more popular logic-based languages, such as PROLOG, restrict both the logic and the operational model so that the full power of the logic is not available and the execution remains relatively efficient.

### 4.1 Executing Logical Descriptions

The basis for the execution of logical statements is to construct a model for the set of statements. This, in general, involves theorem-proving. For example, in PROLOG, the model constructed represents a particular refutation constructed through SLDNF-resolution. Unfortunately, in representing agent behaviours concisely, we must often use forms of logic more expressive still than Horn Clauses. Even in PROLOG, decidability is not a property of the executable logic. When more expressive logics are used, completeness is also lost!

Thus, we advocate the use of logic-based languages for representation of agent behaviours, together with the direct execution of these languages, but with the following provisos.

- In abstract, the execution of a formula is just an *attempt* to build a model for that formula.
- We want to avoid *full* theorem-proving – it is often too expensive. Although we might use theorem-proving like mechanisms in the model construction process, *operational* constraints are also imposed in order to make the execution more efficient.
- Some formal properties of the logic are lost, not only through the operational restrictions, but also through the power of the logic used. For example, completeness and decidability might both be lost. Similarly, although execution of a formula is analogous to model construction for that formula, we might be unable to construct a *finite* model. Again, in these, more expressive, logics execution is the process of *attempting* to construct a model.

More important than the formal logical properties of the language are its pragmatic attributes. For example, although the execution mechanism should be sound, it should also be (relatively) efficient. Most importantly, the core logical features of the language must be both concise and applicable.

Given that we wish to directly execute the set of logical rules representing an agent's behaviour, we must decide what particular execution mechanism to employ. In the next section, we outline the approach we adopt, contrasting it with the predominant approach to the execution of logical formulae, namely logic programming.

### 4.2 Execution within an Agent

Whilst we advocate the use of logic languages as agent descriptions and the direct execution of these languages in order to provide agent behaviour, we argue that the standard logic-programming model is inappropriate for executing general agent behaviours, for the following reasons.

- In many cases, we require programs to be non-terminating. In such cases, we cannot expect the execution mechanism to return a finite object (a model). This leads us away from standard logic programming towards, if necessary, concurrent logic languages.
- More importantly, agents often want to specify, and thus attempt to achieve, several simultaneous goals. It is difficult, if not impossible, to represent this directly in a (single) goal directed framework such as logic programming.
- Horn Clauses, or similar classes of formulae often required for logic programming, may be too restrictive.
- In representing and executing agent descriptions where interaction with an environment is required, we must certainly disallow backtracking over observable actions. However, we would like to have some backtracking *within* agents.

The execution mechanism we employ is both natural for the style of logic we advocate (temporal logic) and avoids many of the above problems. The essential features of this execution mechanism are as follows.

- It is based upon *forward chaining* from the initial constraints through the set of temporal rules representing the agent's behaviour.
- This execution is constrained by the execution of eventualities. Thus, when a formula such as $\Diamond e$ is executed, $e$ becomes an *eventuality* that must be made true at some point in the future. If $e$ cannot be satisfied immediately, it is added to the list of outstanding eventualities. When the execution mechanism has a non-deterministic choice (for example, when it has to execute a disjunction), the outstanding eventualities constrain this choice such that the execution mechanism attempts to satisfy the eventualities as soon as possible, with the oldest outstanding eventuality being attempted first.

  Thus, in terms of agents, these eventualities can be seen as goals that the agent attempts to satisfy as soon as it can.
- Each agent is an independent asynchronously executing object. In order to implement communication in a natural (and logical) way, the predicates in each agent's rules are split into three categories: *environment*, *internal* and *component*. Environment predicates are under the control of the agent's environment, while the other categories of predicate can be made true or false by the agent itself. Thus, when an agent's execution mechanism makes an internal predicate true, it just records the fact in its internal memory, while when it makes a component predicate true, it also broadcasts this predicate to all other agents. Finally, if an environment predicate is to be executed, the agent must *wait* for the predicate's value to be provided by the environment (i.e., by broadcasting its value). This allows agents to *synchronise* with other agents on selected messages.
- Agents are allowed to backtrack. As the agent's execution mechanism has a range of non-deterministic choices, it can, if it finds a contradiction, backtrack to a previous choice point and continue executing but on the basis of a different choice.

  However, as agents are part of an open object-based system, we do not allow an agent to backtrack past the broadcast of a message. Thus, once an agent has broadcast a message, it has effectively *committed* its execution to that choice. This allows agents to carry out search through backtracking internally, but avoids the problem of attempting to rollback actions in a distributed system.

Having discussed and motivated the execution and communication mechanisms for individual agents in our system, we now give a flavour of one of the aspects currently under development, namely the addition of grouping into this framework.

### 4.3 Implementing Agent Groups

Recall that a *group* is essentially a set of agents. These are used to restrict the extent of broadcast communication and thus to structure the agent space. The basic properties we require of groups are that agents should be able to

- send a message to a group,
- add an agent to a group,
- ascertain whether a certain agent is a member of a group,
- remove a specified agent from a group, and,
- construct a new subgroup.

There are other properties that might be useful, such as the ability to list the members of a group, but they are not essential.

Two alternative approaches to the implementation of groups of agents within CONCURRENT METATEM are currently being investigated. While these will not be described in great detail, we provide an outline of them in order to show a little of the range of representations that are available. These simple mechanisms allow us to represent a group of agents either

1. as a set of named agents, e.g

$$\{ \texttt{manager, solver1, solver2, worker, ...} \}$$

2. or as a set construction formula, where any agent satisfying the formula is considered as a member of the group, e.g

$$\texttt{agent(X)} \wedge \texttt{solver(X)}$$

The former involves generating the group from the bottom up, the latter from the top down (by using a declarative description). Note, however, that we are able to carry out both forms using the more expressive second style, e.g.

$$\texttt{agent(manager)} \vee \texttt{agent(solver1)} \vee \texttt{agent(solver2)} \vee \texttt{...}$$

Thus, this formula is true for any of the members of the group explicitly represented in (1) above. Similarly, we can either add an agent to the group by adding a disjunct, e.g, '$\vee$ `agent(added)`' to the defining formula, or remove an agent from the group by adding a negated conjunct, e.g., '$\wedge\ \neg$ `agent(removed)`' to the formula.

However, checking whether a particular agent is in the group is slightly more difficult in the second case as it involves some logical manipulation, rather than just string matching. Further, if we ask the group object to give us a list of its members, this is *much* more difficult using the second approach as there is no explicit representation of the agents within the group, simply a statement of what properties group members must have!

The discussion of which particular technique is most appropriate continues. The two approaches to group computation described above are being evaluated at present, both for their practical efficiency and for their logical consistency.

## 5 Applications in Multi-Agent Systems

In this section, we briefly indicate how our approach can be applied in a variety of different agent-based scenarios. These cover both the representation and implementation of single agents, and of societies of dynamically interacting agents. As the applications of our language have been presented elsewhere, we will only give an indication of how it can be applied.

The purpose of this section is to indicate to the reader that not only does our approach have some logical merit, but it can also be applied in a range of simple multi-agent scenarios. Further, the two main elements of the system, namely the executable temporal specifications and the communication/grouping mechanism, can often be applied separately. Thus, the communication/grouping mechanism is useful for structuring the agent space, while the temporal logic is useful for representing and executing dynamic behaviour in individual objects.

### 5.1 Dynamic Objects

By utilising the power of temporal logic, we are able to specify the internal behaviour of *dynamic* objects. In particular, we can represent agents whose internal state changes over time, whilst interacting with its environment. Rather than giving a range of examples here, we refer the reader to our survey of CONCURRENT METATEM applications [9].

### 5.2 Reaction versus Deliberation

There seems to be a continuing debate about the relative merits of *reactive* architectures versus *deliberative* architectures (typically based upon planning). Rather than arguing for one approach or the other, we take the view that *both* types of behaviour should be possible within agents. In particular, we argue that, in allowing concurrent activities within an individual agent, we are able to represent behaviour that is both reactive and deliberative. Agents can react immediately to certain stimuli, but can be carrying out a longer term planning process in the background.

For example, an agent can contain a range of transition rules representing reactive situations, such as

$$\begin{aligned} \texttt{stimuli1} &\Rightarrow \bigcirc\texttt{response1} \\ \texttt{stimuli2} &\Rightarrow \bigcirc\texttt{response2} \end{aligned}$$

Note that a response occurs here in the next step of the agent and so a variety of immediate responses can be represented. As well as being useful for reactive architectures in DAI, such rules can be used as part of more traditional applications, such as process control [8].

We are also able to represent typical deliberative activities such as planning, for example by

$$\begin{aligned} \texttt{problem} &\Rightarrow \Diamond\texttt{plan} \\ \texttt{plan} &\Rightarrow \texttt{broadcast(plan)} \end{aligned}$$

which states that at some time in the future the agent will have generated a plan to solve a particular problem and, when it does, the agent will broadcast this plan. There are two ways in which the agent might construct the plan, as follows.

1. Use $\Diamond\texttt{plan}$, as above, but add rules constraining the production of the plan, for example

   $$\begin{aligned} \neg\ \texttt{pre1} &\Rightarrow \neg\ \texttt{plan} \\ \neg\ \texttt{pre2} &\Rightarrow \neg\ \texttt{plan} \end{aligned}$$

   which states that the plan cannot be achieved until the preconditions `pre1` and `pre2` have been achieved. These subgoals can in turn be solved by adding the rules

   $$\begin{aligned} \texttt{problem} &\Rightarrow \Diamond\ \texttt{pre1} \\ \texttt{problem} &\Rightarrow \Diamond\ \texttt{pre2} \end{aligned}$$

   Thus, we can attempt to utilise the deductive and backtracking aspects of the system in order to achieve the construction of the plan.
2. An alternative approach is to plan without backtracking. Here, we use a continuation-based approach to represent the *rest* of the search space. This approach is more expensive, but safer (see below). However, such a planner would have to be built *on top* of CONCURRENT METATEM — it would not directly exploit the execution mechanism of the system.

There is a general problem with the first approach in that it may be difficult to mix reactive and deliberative aspects within the same agent. This is because, if the planning process depends upon backtracking of some sort, the reactive rules might be such that they broadcast a message at some point thus effectively committing the execution to a certain path and stopping the system backtracking past this point. The simplest solution to this problem is, when we require an agent that has both planning and reactive capabilities, to spawn a separate 'planning' agent which carries out the planning activity in parallel with the original agent. The original agent acts reactively to its environment, having spawned the planning agent, but once the planning agent has succeeded in producing a plan the original agent is at liberty to act upon it.

This type of behaviour leads us on to other examples of multi-agent systems where cooperative problem-solving can occur.

### 5.3 Distributed Problem Solving

As shown in [12], we are able to define a range of simple, abstract distributed problem solving systems. By utilising the power of temporal specifications for individual objects simple problem-solving agents can be implemented in CONCURRENT METATEM. These agents can then be organised to form a variety of problem-solving architectures. In particular, we can represent hierarchical, cooperative and competitive problem-solving. It is the combination of broadcast message-passing together with agent grouping that again allows us to develop such rich structures.

### 5.4 Simulating Societies

Combining the definition of individual dynamic agents, as above, together with broadcast message passing and grouping structures enables us to develop more complex agent societies. (For more details, see [11].) In particular, we are able to represent a range of group behaviours from cooperation through to competition, not only between individual agents but between groups of agents. Further, the grouping structures are dynamic as well — new agents can be added to groups and other agents can be removed.

### 5.5 Physical Attributes

As well as allowing the development of complex structuring within the object space, the grouping mechanism provides the potential for innovative applications, such as its use to represent physical properties of objects. For example, if we assume that any two objects in the same group can 'see' each other, then movement broadcast from one object can be detected by the other object. Similarly, if an object moves 'out of sight', it moves out of the group and thus the objects that remain in the group will 'lose sight' of it.

### 5.6 Fault-tolerance

As described above, the notion of groups, together with broadcast message-passing allows us to represent a powerful form of fault-tolerance. Thus, we are able to build this in to our multi-agent systems in addition to their purely functional behaviour. Note also, that the possibility of having many agents recognising and broadcasting the same messages can be used to develop more powerful forms of competition and adaptability, such as those seen in applications of genetic algorithms and artificial life.

## 6 The Development of Multi-Agent Systems

Formal methods have been used productively in the development of a variety of software and hardware systems. Particularly for complex and safety-critical systems, a formal approach is seen as vital. We believe that, as multi-agent systems become more complex and are applied in ever more critical areas, there will increasingly be a requirement for some form of formal validation/verification.

Our long term goal with this work is the provision of a formal framework for the specification, animation and development of distributed multi-agent systems. We have already identified many important features of multi-agent systems through our work on the abstract modelling of such systems and, by extending our executable temporal logics with a variety of these features, we will be able to model and prototype a range of practical systems. By basing our work on a simple temporal logic, for which there are already a variety of proof methods, we have already outlined an approach to the specification and verification of multi-agent systems [12].

In this section we will briefly identify some of the key areas that need to be developed. We are actively investigating several of these, though others remain as long term goals.

### 6.1 Temporal Specifications

Temporal logics have been widely used in the specification and verification of reactive systems [16]. Recent and future work in this area will feed into our development framework as multi-agent systems are just a particular class of reactive system. In particular, the work on compositional temporal specification and verification will have some bearing on our ability to effectively verify large multi-agent systems. However, the use of an object-based computational model, together with broadcast message-passing, will avoid some of the problems associated with the large-scale verification of multi-agent systems.

### 6.2 Agent Transformation

Currently, we are able to specify and verify properties of both individual agents and simple multi-agent systems. Rather than verifying the properties of both agents and systems at every stage of the development process, we intend to provide suitable *transformation* rules for both individual agents and groups of agents. These rules will preserve certain temporal properties. In particular, we will be able to transform an agent's specification to another specification that is logically equivalent (or at least implies the original specification). Further, *structural* transformations will be utilised, enabling the refinement of groups of agents. This will allow both the merging and splitting of agent groups during the development process.

It is important to note that any formula in the logic can be transformed into a set of temporal rules of the required form. However, the system developer would like some control of this process as two logically equivalent sets of rules might not execute at comparable speeds. Similarly, a logical specification might not have the identical behaviour, when executed, as the same specification distributed across multiple agents.

### 6.3 More Expressive Descriptions

The temporal logic described in this paper provides a simple, yet powerful, basis for the description of dynamic agents. However, there is the potential for the extension of this logical base with further features, including both reflection and meta-level capabilities, and common modalities such as those of intention, knowledge and belief. In the future, we intend to investigate these extended logics, particularly the simple extensions towards knowledge and belief. Although these more powerful logics are applicable in a wider range of complex systems, their logical properties are unknown though it is likely that the manipulation of agent descriptions will be much more expensive in such frameworks. Further, there are the questions of (a) are these additional modalities *really* necessary, and (b) if they are, is there a *core* set of them?

Given that, at least in certain cases, we might wish to specify agents using these more complex descriptions, we must also decide whether the concepts (i.e., intention, belief, etc) are transferrable between agents. If they are not, then a purely *temporal* specification may be carried out, with elements of intention, belief, etc, being introduced at a later stage of (single) agent refinement. If, however, we expect one agent to be able to send messages using these concepts and have the receiving agent understand them using *exactly* the same semantics, then this is likely to require a much more complex specification framework.

### 6.4 Heterogeneous Agents

An important aspect of our approach is that the two main elements, namely the executable temporal logic and the communication/grouping mechanism, can be separated. This is particularly useful when we consider heterogeneous agents. Here, the temporal logic description of each agent is simply a specification. A variety of programming languages can be used to implement individual agents. For example, one agent might be implemented in LISP, and other in C, and so on. As long as each agent satisfies its temporal specification and obeys the common communication rules, then these heterogenous agents are able to form a multi-agent system.

Two important aspects relating to this need to be studied. The first is the refinement of a temporal specification to a particular programming language, which requires appropriate refinement rules for a variety of languages. The second is the verification of properties of agents already implemented in standard programming languages. This, we may be able to achieve through a process of *temporal model-checking* [7] which allows us to check whether simple finite-state systems satisfy certain temporal properties.

Finally, we are also investigating the direct translation of a temporal specification into a lower-level programming language (e.g. C). This is likely to improve efficiency and would be used as part of the compilation process.

### 6.5 Verification System

The logic we use at present is relatively simple. However, as agents execute asynchronously and as each agent can effectively refer to an infinite domain, the logical base is more complex when we consider the verification of global properties of multi-agent systems. Not only is the logic undecidable, but first-order temporal logic is incomplete [22]. Further, due to the asynchrony of execution, we require a temporal logic based upon a *dense* model of time, rather than a discrete one [3, 10].

This, together with the possibility of using more powerful augmented logics means that the development of a verification system based upon our approach is likely to require further research into proof methods for temporal logics (particularly links to inductive and semi-automatic theorem-proving). We have made some progress in this area, though much is yet to do. We note that, even when using our simple temporal logic, we are able to verify many useful properties of multi-agent systems at present. However, we are not yet able to tackle either large systems or systems where the agents are particularly powerful.

In addition, the proof rules we have developed do not, as yet, incorporate the full range of *operational constraints* occurring in CONCURRENT METATEM. In proofs about the system, we assume that the execution mechanism perfectly implements the semantics of temporal formulae. However, we do not attempt to implement temporal logic completely, even in the propositional case. Thus, in order to reason about the implemented systems, rather than just their specifications, we must incorporate the operational constraints inherent within CONCURRENT METATEM into the proof system. As an example, the formula $\Diamond \phi$ in temporal logic means that the formula $\phi$ *will* be satisfied at some point in the future, while when this is executed in CONCURRENT METATEM we simply *attempt* to satisfy $\phi$ as soon as we can.

This problem, which is being studied at present, is related to the more general problem of whether to use a logical formula purely declaratively, or to add operational constraints in order to be able to execute it efficiently. These constraints in turn affect the declarative semantics of the original formula.

## 7 Summary

In this paper, we have motivated the use of executable temporal logics, together with broadcast message-passing and grouping mechanisms. We have discussed their use in the development of a range of multi-agent systems and have justified our approach in both logical and practical terms. As well as the notion of autonomous objects, CONCURRENT METATEM also provides a larger structuring mechanism through the ‘groups’ extension. This not only restricts the extent of an object’s communications, but also provides an extra mechanism for the development of strategies for organisations. In particular, this provides the basis for agent cooperation, competition and interaction.

We have shown that a logic-based approach is not only theoretically desirable, but can also be practical. Further, the language we have developed captures a range of agent behaviours in a natural way. We believe that, regardless of the utility of our combined approach, the notions of executable agent specifications and group-based agent communication may be used separately in future systems.

Finally, we have shown that by narrowing the gap between theory and implementation, we are able to retain the elements of agent description that we require, while opening up the possibility of the formal development of multi-agent systems.

### 7.1 Related Work

While a wide variety of logical theories, purporting to represent agents, have been proposed few, if any, have provided the basis for an agent programming language. For example, languages such as `April` [18] and AgentSpeak [24] do not appear to be directly based upon a logical theory of agency. The closest work to that described in this paper is that of Shoham [20] who has proposed *Agent-Oriented Programming* (AOP). This framework consists of a logical system for defining the mental state of agents, together with an interpreted language for programming those agents [23]. While the logical basis of AOP contains notions of time, as well as other modalities such as belief, intention and capability, and the programming language provides syntactic support for such concepts, it is not clear what the formal link between the language and the logic is [25]. While, as mentioned above, we are not expecting to *exactly* implement the semantics of our logical specifications, we believe that CONCURRENT METATEM provides a closer link between its logic and programming language than AOP in that is *attempts* to implement as much as is practical of the logical basis.

## References


1. M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8: 277–295, 1989.
2. G. Agha. *Actors - A Model for Concurrent Computation in Distributed Systems*. MIT Press, 1986.
3. H. Barringer, R. Kuiper, and A. Pnueli. A Really Abstract Concurrent Model and its Temporal Logic. In *Proceedings of the Thirteenth ACM Symposium on the Principles of Programming Languages*, St. Petersberg Beach, Florida, 1986.
4. K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.
5. K. Birman. The Process Group Approach to Reliable Distributed Computing. Techanical Report TR91-1216, Department of Computer Science, Cornell University, July 1991.
6. A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault Tolerance. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, New Hampshire, October 1983. ACM. (In ACM Operating Systems Review, vol. 17, no. 5).
7. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
8. M. Finger, M. Fisher, and R. Owens. METATEM at Work: Modelling Reactive Systems Using Executable Temporal Logic. In *Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-93)*, Edinburgh, U.K., June 1993. Gordon and Breach Publishers.
9. M. Fisher. A Survey of Concurrent METATEM — The Language and its Applications. In *First International Conference on Temporal Logic (ICTL)*, July 1994.
10. M. Fisher. Towards a Semantics for Concurrent METATEM. In M. Fisher and R. Owens, editors, *Executable Modal and Temporal Logics*. Springer-Verlag, 1995.
11. M. Fisher and M. Wooldridge. A Logical Approach to the Representation of Societies of Agents. In N. Gilbert and J. Doran, editors, *Artificial Societies*. UCL Press Ltd, London, 1995.
12. M. Fisher and M. Wooldridge. Specifying and Verifying Distributed Intelligent Systems. In *Portuguese Conference on Artificial Intelligence (EPIA)*. Springer-Verlag, October 1993.
13. M. Fisher. A Resolution Method for Temporal Logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)*, Sydney, Australia, August 1991. Morgan Kaufman.

14. W. D. Hillis. The Connection Machine: A computer architecture based on cellular automata. *Physica D*, 10:213–228, 1984.
15. W. D. Hillis and G. L. Steele. Data parallel algorithms. *Comm. ACM*, 29(12):1170–1183, December 1986.
16. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.
17. T. Maruichi, M. Ichikawa, and M. Tokoro. Modelling Autonomous Agents and their Groups. In Y. Demazeau and J. P. Muller, editors, *Decentralized AI 2 – Proceedings of the* $2^{nd}$ *European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW '90)*. Elsevier/North Holland, 1991.
18. F. McCabe and K. Clark. `April` — agent process interaction language. (In this volume.)
19. R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Comm. ACM*, 7(19), July 1976.
20. Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60(1):51–92, 1993.
21. R. G. Smith. *A Framework for Distributed Problem Solving*. UMI Research Press, 1980.
22. A. Szalas and L. Holenderski. Incompleteness of First-Order Temporal Logic with Until. *Theoretical Computer Science*, 57:317–325, 1988.
23. S. R. Thomas. The PLACA Agent Programming Language. (In this volume.)
24. D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a Concurrent Agent-Oriented Language. (In this volume.)
25. M. Wooldridge and N. Jennings. Agent Theories, Architectures and Languages: A Survey. (In this volume.)


This article was processed using the LaTeX macro package with LLNCS style