Executable Temporal Logic for Distributed A.I.

Michael Fisher[*]
Department of Computer Science
Victoria University of Manchester
Oxford Road
Manchester M13 9PL
U.K.
EMAIL: `michael@cs.man.ac.uk`

Michael Wooldridge
Department of Computing
Manchester Metropolitan University
Chester Street
Manchester M1 5GD
U.K.
EMAIL: `mikew@sun.com.manp.ac.uk`

DAI'93 Themes:
societies and organisations of agents;
modeling through communication in adversarial and cooperative systems

Abstract

This paper describes Concurrent METATEM, a programming language based on temporal logic, and applies it to the study of Distributed Artificial Intelligence (DAI). A Concurrent METATEM system consists of a number of asynchronously executing objects, which are able to communicate through broadcast message-passing. Each individual object directly executes a specification of its desired behaviour. Such specifications are given using a set of temporal logic 'rules', determining how the object may generate 'commitments', which it subsequently attempts to satisfy.

This language provides a novel and powerful approach to representing DAI systems, where individual 'agents' are specified in a natural way using temporal logic, while groups of agents communicate by broadcasting information.

The paper begins by discussing and justifying the Concurrent METATEM approach, and then describes objects and their execution in more detail. Several examples are presented, demonstrating the utility of Concurrent METATEM for DAI applications. Finally, the language is contrasted with other contemporary DAI testbeds.

1 Introduction

This paper describes a novel programming language based upon executable temporal logic, and demonstrates the utility of this language for tackling the problems faced by designers and implementors of Distributed Artificial Intelligence (DAI) systems. This language, called Concurrent METATEM [13, 12], incorporates two innovations which together distance it from other DAI programming languages.

1. *It is based on the idea of executable temporal logics*. In common with other, more traditional logic programming languages, (e.g., PROLOG), Concurrent METATEM has both a well-understood logical foundation, and an intuitively simple operational semantics. However, its logical foundation (temporal logic) is more powerful than the corresponding classical logics. In spite of this extra expressive power, temporal logic execution remains computationally viable: several other languages based upon executable temporal logics have been developed and applied in a variety of domains [23, 16, 1, 3].

[*]Supported by the SERC under research grant GR/H/18449.

2. *It is based on a novel model of concurrency for executable logics*. This model differs radically from those found in other logic-based languages (e.g., [27]), as it uses the notion of active, autonomous agents, each with significant computational power, communicating via broadcast message-passing: in Concurrent METATEM, the notion of a communicating intelligent agent is given an intuitively simple, well-understood logical foundation. We are aware of no other DAI language/testbed that has this property.

The remainder of this paper is structured as follows. In §1.1, we outline the motivation for our research, and present a rationale for the Concurrent METATEM approach. In §2, we present a brief description of the Concurrent METATEM language itself, while in §3, we present some example program fragments, demonstrating the utility of the language for DAI applications. We show how a variety of cooperative structures can be described and implemented using Concurrent METATEM. In §4, we discuss how Concurrent METATEM stands in relation to other DAI/concurrent object programming languages and testbeds. Finally, in §5, we present some conclusions, and identify some avenues for future research.

1.1 Motivation

Distributed AI is a relatively young discipline, which has developed its own tools for building, experimenting, and evaluating ideas, theories, and applications. Over the past decade, many frameworks for building DAI systems have been reported. While some, such as the DVMT, allow experimentation only with one specific scenario [8], and others provide extensions to existing AI languages, such as LISP [17], comparatively few fundamentally new languages have been proposed for DAI. Arguably, the most successful DAI languages have been based — at least in part — on the Actor model of computation [20, 2], for example [10] and [5].

It is our contention that while frameworks for DAI based on extensions to existing AI languages are useful for experimentation, they will not, ultimately, be viable tools for building production DAI systems. This argument is justified below; to further motivate our discussion, we introduce the notion of a Pnuelian *reactive system*[1]:

> 'Reactive systems are systems that cannot adequately be described by the *relational* or *functional* view. The relational view regards programs as functions … from an initial state to a terminal state. Typically, the main role of reactive systems is to maintain an interaction with their environment, and therefore must be described (and specified) in terms of their on-going behaviour … [E]very concurrent system … must be studied by behavioural means. This is because each individual module in a concurrent system is a reactive subsystem, interacting with its own environment which consists of the other modules'. [25]

DAI systems are reactive, in precisely this sense:

- the applications for which a multi-agent approach seems well suited, such as (e.g., distributed sensing [8]), are non-terminating, and therefore cannot simply be described by the functional view;

- multi-agent systems are necessarily concurrent, and as Pnueli observes (above), each agent should therefore be considered as a reactive system.

[1]There are at least three current usages of the term *reactive system* in computer science. The first, oldest, usage is that by Pnueli and followers (see, e.g., [25], and the description above). Second, researchers in AI planning take a reactive system to be one that is capable of responding rapidly to changes in its environment — here the word 'reactive' is taken to be synonymous with 'responsive' (see, e.g., [21]). More recently, the term has been used to denote systems which respond directly to the world, rather than reason explicitly about it (see, e.g., [7]). In this paper the term is used in its Pnuelian sense, except where otherwise indicated.

Both LISP and PROLOG — the classic AI languages — lend themselves more readily to the relational view of computation than the reactive view: LISP is actually a functional language, with computation viewed as function evaluation; PROLOG is based on the view of computation as goal-directed backward theorem proving, which does not map comfortably into the reactive view of systems. In an attempt to extend the utility of both these languages, several concurrent extensions have been defined. However, since we have just argued that DAI systems are inherently reactive, it seems that both languages are unsuited at a fundamental level to DAI system construction. What seems to be required is a completely new approach to conceptualising and building intelligent reactive systems; one to which the notion of reactivity is intrinsic.

In a 1977 paper, Pnueli proposed temporal logic as a tool for reasoning about reactive systems [24]: when describing a reactive system, we often wish to express properties such as 'if a request is sent, then a response is eventually given'. Such properties are easily and elegantly expressed in temporal logic. As we observed above, each object in a Concurrent METATEM system directly executes a temporal logic specification of its desired behaviour. The concept of a reactive system is therefore at the very heart of the language.

Concurrent METATEM is based on a novel model for concurrency in executable logic. Whereas most previous concurrent logic paradigms are based on fine-grained AND-OR parallelism, (e.g., [6]), concurrency in Concurrent METATEM is achieved via coarse-grained computational entities called objects. Moreover, objects in Concurrent METATEM have a well-motivated logical foundation, which is at once simple, powerful, and intuitively appealing. As a corollary of giving objects a logical foundation, formally specifying and verifying the properties of Concurrent METATEM systems is a realistic possibility [15].

2 Concurrent METATEM

In this section, we describe Concurrent METATEM in detail: we begin with a brief overview[2]. A Concurrent METATEM system contains a number of concurrently executing *objects*, which are able to communicate through asynchronous broadcast message passing. Each object is programmed by giving it a temporal logic specification of the behaviour it is required to exhibit. Objects directly execute their specifications, where a specification consists of a set of 'rules', which are temporal logic formulae of the form

$$\text{antecedent about past} \Rightarrow \text{consequent about future.}$$

Objects maintain a record of the messages they send, and any (internal) actions they perform: object execution proceeds by a process of continually determining which of the past-time antecedents of rules are *satisfied* by this recorded *history*. The antecedents of any rules that do fire become *commitments* which the object subsequently attempts to satisfy. It is possible that an object's commitments cannot all be satisfied simultaneously, in which case unsatisfied commitments are carried over into the next 'moment' in time.

Inter-object communication is managed by *interfaces*, which each object possesses. An interface determines what messages an object may send, and what messages, if sent by another object, it will accept. Whenever an object satisfies a commitment internally, it consults its interface to see whether this commitment corresponds to a message that should be sent; if it does, then the message is broadcast to all other objects. On receipt of a message, an object will consult its interface to see whether the message is one that should be accepted: if it is, then the message is added to the history; otherwise, it is ignored.

[2]An alternative, but preliminary, outline is given in [13], while a complete description of Concurrent METATEM can be found in [12].

A more complete description of Concurrent METATEM will now be given; in order to do this, we must first introduce both temporal logic and (non-concurrent) METATEM. We will only give a brief introduction to these topics; for more details, see [9] and [14].

2.1 Temporal Logic

Throughout this paper we use a discrete, linear temporal logic, called First-order METATEM Logic (FML): a complete definition of FML can be found in [14]. This logic introduces several new connectives to classical logic to enable us to reason about the time dependent properties of reactive systems. In order to do this, FML incorporates a particular model of time where 'states', representing moments in time, form an infinite discrete linear sequence. This, together with the constraint that there is a unique state representing the 'beginning of time', ensures that models of FML formulae are isomorphic to the natural numbers ($\mathbb{N}$).

The classical component of FML is used to describe each state. For example, the formula

$$(\textit{optimistic} \wedge \textit{submit-paper})$$

will be satisfied in a state if we are both optimistic about the acceptance of this paper and we have just submitted it. The temporal connectives relate such states together. For example, $\varphi \mathcal{U} \psi$ is satisfied in a state s if the formula ψ is satisfied in a state t in the future of state s, and the formula φ is satisfied in every state between s and t. Thus, φ must be satisfied *until* ψ is satisfied. The dual of this *until* connective is the *since* connective ($\mathcal{S}$). This mirrors the behaviour of $\mathcal{U}$ in the past, so that $\varphi \mathcal{S} \psi$ is satisfied in a state s if the formula ψ is satisfied in a state r in the past of state s, and the formula φ is satisfied in every state between states r and s. Hence φ has been satisfied *since* ψ was satisfied.

For example, we will be optimistic about the acceptance of this paper if we were initially optimistic when we submitted the paper, and the paper has not been rejected since that moment. Thus at any particular state the following temporal constraint would apply.

$$(\neg \textit{rejected})\, \mathcal{S}\, (\textit{optimistic} \wedge \textit{submit-paper}) \;\Rightarrow\; \textit{optimistic}$$

The other two basic temporal connectives in FML are the *weak last-time* connective, '$\bullet$', and the *next-time* connective, '$\bigcirc$'. If there was a last state, and φ was satisfied in that state, then $\bullet \varphi$ is satisfied in the current state, while $\bigcirc \varphi$ is satisfied in the current state if φ is satisfied in the next state in the sequence.

The remaining temporal connectives can be defined in terms of this basic set. '$\Diamond$' is the *sometime* connective, so that $\Diamond \varphi$ will be satisfied in the current state if φ is satisfied either in the current state or in some future state; '$\Box$' is the *always* connective, so that $\Box \varphi$ will be satisfied in the current state if φ is satisfied in the current state and in all future states; '$\blacklozenge$' and '$\blacksquare$' are the *sometime in the past* and *always in the past* connectives, mirroring the sometimes and always connectives in the past. $\mathcal{W}$ is the binary *unless* connective, which is somewhat similar to the *until* connective, except that its second argument need never be satisfied; $\mathcal{Z}$ is the *zince* operator, which is similar to the *since* connective except that its second argument need never have been satisfied. Note that, in this form of temporal logic, we use *strict* past-time connectives. For example, $\blacklozenge \varphi$ is satisfied if φ was satisfied in some state in the past (i.e., it is strict), whereas $\Diamond \varphi$ is satisfied if φ is satisfied at some state in the future, *or in the present state*.

Finally, the *strong last-time* connective, '$\circledcirc$', is similar to the *weak last-time* connective, except that if there was no last state, then no formula of the form $\circledcirc \varphi$ can be satisfied.

2.2 METATEM

METATEM is a general framework for executing temporal logics [3]. However, METATEM programs are not just arbitrary FML formulae: for our purposes, a program is a formula with the following general form.

$$\square \bigwedge_{i=1}^{n} \forall \overline{x_i}.\ P_i(\overline{x_i}) \;\Rightarrow\; \exists \overline{z_i}.\ F_i(\overline{x_i}, \overline{z_i})$$

Here, '$\overline{x_i}$' represents a vector of variables, x_{i_1}, x_{i_2}, …, x_{i_m}, and each of the conjoined implications is called a *rule*. The P_i and F_i are restricted in that each P_i is a (non-strict) *past-time* temporal formula (i.e., a formula including past-time and present-time constraints) and each F_i is a (non-strict) *future-time* formula (i.e., a formula including both present-time and future-time constraints). Thus a METATEM program is a set of rules of the form $P(\overline{x}) \;\Rightarrow\; F(\overline{x}, \overline{z})$, as above.

The process of executing a METATEM program involves constructing a model for the corresponding FML formula, in the presence of input from the program's *environment*. The model is constructed iteratively, starting from the initial state (state 0). At each step, the program rules are consulted, to see which of past-time antecedents (P_i) are satisfied by the partial model constructed so far[3]. The future-time components (F_i) of those rules with satisfied antecedents are collected together. These *constraints* on the present and future properties of the model, together with any outstanding constraints generated on previous steps, are then used to construct the current state. Any outstanding constraints are passed on to the next step.

At each state there may be several levels of non-determinism. First, there is non-determinism in the execution of a classical disjunction: a constraint of the form $\varphi \vee \psi$ may be satisfied by executing either φ or ψ. Second, the execution process must also decide when to satisfy outstanding commitments, such as those generated by the execution of $\Diamond$-formulae. Note that, to satisfy the formula $\Diamond\varphi$, then φ must be satisfied at *some* state in present or the future. Usually, we choose to satisfy such formulae as soon as possible. However, the satisfaction of one $\Diamond$-formula might conflict with the satisfaction of others. The basic algorithm followed in this version of METATEM is to attempt to satisfy eventualities (i.e., φ in formulae such as $\Diamond\varphi$ and $\psi\,\mathcal{U}\,\varphi$) as soon as possible and, if any such formulae are passed forward from previous states, to attempt to satisfy the oldest outstanding eventuality first [3, 14].

If a contradiction is generated (i.e., **false** is to be executed) then the system may backtrack to a previous choice-point and attempt to construct a model for the program in a different way. This backtracking undoes actions and returns the execution to a previous choice point. If there are no more choice points left, the execution fails, signifying that the program is unsatisfiable.

2.3 A Computational Model

Concurrent METATEM provides an operational model for communicating METATEM processes, based on the *CMP Model* [12]. As mentioned earlier, a Concurrent METATEM system contains a number of asynchronously executing objects, which are able to communicate through broadcast message passing. Each object has two main attributes:

- an *interface* definition, which defines how an object may interact with its environment (i.e., other objects), and in particular what messages it may send and receive;

- an *internal* definition, which defines how the object may act — this definition is actually a METATEM program (i.e., a set of rules in the '*past* $\Rightarrow$ *future*' form).

An object's interface definition contains three components:

- a name for the object;

- a set of predicate symbols called *environment* predicates — these symbols correspond to messages the object will *recognise*;

[3]It is in this process of 'checking the past' that input from the environment may be required as environment predicates can only appear in the P_i component.

- a set of predicate symbols called *component* predicates — these symbols correspond to messages the object may *send*.

For example, the interface for a '`stack`' object might be defined as follows.

```
stack(pop,push)[popped,stackfull].
```

Here, {`pop`, `push`} is the set of environment predicates, (i.e., messages the object recognises), while {`popped`, `stackfull`} is the set of component predicates (i.e., messages the object might produce itself). Note that these sets need not be disjoint — an object may broadcast messages that it also recognises: in this case, messages sent by an object to itself are recognised immediately.

During the execution of each object's rules, the two types of predicate mentioned above now take on a specific operational interpretation, as follows.

- *Environment predicates represent incoming messages.* An environment predicate can be made true if, and only if, the corresponding message has just been received. Thus, a formula containing an environment predicate, such as '*request*(x)', where x is a universally quantified variable, is only satisfied if a message of the form '`request(b)`' has just been received (for some argument '`b`').

- *Component predicates represent messages broadcast from the object.* When a component predicate is satisfied, it has the (side-)effect of broadcasting the corresponding message to the environment. For example, if the formula '*offer(e)*' is satisfied, where *offer* is a component predicate, then the message '`offer(e)`' is broadcast.

Note that some predicates used by an object are neither environment nor component predicates; these *internal* predicates have no external effect. They are used as part of the internal computation of the object and, as such, do not correspond either to message-sending or message reception.

Once an object has commenced execution, it continually follows a cycle of reading incoming messages, collecting together the rules that 'fire' (i.e., whose left-hand sides are satisfied by the current history), and executing one of the disjuncts represented by the conjunction of right-hand sides of 'fired' rules, as described in §2.2. Individual objects execute asynchronously, and are autonomous in that they may execute independently of incoming messages and may change their interface dynamically.

Objects may backtrack, with the proviso that an object may not backtrack past the broadcasting of a message. Consequently, in broadcasting a message to its environment, an object effectively *commits* the execution to that particular path. Thus, the basic cycle of operation for an object can be thought of as a period of internal execution, possibly involving backtracking, followed by appropriate broadcasts to its environment.

2.4 Current Status

At the time of writing, a full implementation of propositional Concurrent METATEM has been developed. This implementation has been used to develop and test numerous example systems, including a simulation of a railway network [11], and propositional versions of all the examples presented in this paper (see below). Experience with this initial implementation is being used to guide an implementation of full first-order Concurrent METATEM.

3 Examples

In this section, we illustrate the utility of Concurrent METATEM for building DAI systems by demonstrating the ease and elegance with which a range of strategies for cooperative interaction can be coded in the language. *Note that our aim is not to propose these strategies as viable techniques for cooperative interaction, but rather to demonstrate how plausible strategies might be coded within the language.*

Space restrictions prevent the inclusion of larger examples; they will be presented in the full paper, if desired.

3.1 Requesting Help

Consider an agent, call it Q, who requires help, such as in solving a particular problem that it can not solve on its own. Now imagine that other agents exist who can solve the problem for agent Q. Thus, Q will make a request to its environment for any offers of help with a problem, p, by broadcasting the message '*request(Q,p)*'. Offers of help may be be forthcoming from certain other agents and they will broadcast their offers using the message *offer(Q,R,p)*, representing an offer by object R to solve problem p for object Q.

Using these two predicates we can construct networks of Concurrent METATEM objects which co-operate in various different ways.

3.1.1 Benevolence

Probably the simplest strategy for cooperative interaction is *benevolence.*

Benevolence: Offer help if an agent requests it, and not otherwise.

This strategy may be encoded in just two program rules. The object skeleton in Figure 1 is all that is required to achieve this strategy. (Note that rule numbers, listed down the left hand side, are for illustration only, and are *not* part of the language; similarly for quantifiers.)

`obj(request)[offer]:`

1. $\forall i.\forall a. \quad \odot\, request(i,a) \Rightarrow \Diamond offer(i,self,a);$
2. $\forall i.\forall a. \quad (\neg request(i,a))\ \mathcal{Z}\ (offer(i,self,a) \wedge \neg request(i,a)) \ \Rightarrow\ \neg offer(i,self,a).$

Figure 1: The Concurrent METATEM Program for a 'Benevolent Agent'

Assuming that the object's name is provided by 'self', then rule (1) encodes the significant part of the strategy, ensuring that help is eventually given wherever it is requested. Rule (2) simply ensures that an object never offers any help unless it has been requested to. In reactive systems terminology, rule (1) represents a *liveness* property, while rule (2) represents a *safety* property. Note that no indication is given in these rules of *when* the object is to '*offer*'; as long as the object *eventually* '*offer*'s', then its behaviour is conformant with its rules. However, in this example, there are no other rules constraining the object's behaviour, and so the algorithm used to determine how commitments are expedited would immediately '*offer*' (see [3]). If we removed the '$\Diamond$' connective then the agent would be required to offer help as soon as it receives a request.

In Concurrent METATEM, the default strategy for assigning truth values to predicates is to assume that if a predicate is not otherwise constrained, it is false. Thus, we do not actually need rule (2) in Figure 1, and so we will omit such rules from now on.

Finally, if multiple requests are received by the agent, then it will broadcast offers for all these requests. A more realistic strategy would be to restrict the agent so that it only makes at most one offer at every step. Again, such a restriction can easily be coded in Concurrent METATEM.

3.1.2 Qualified Benevolence

Although undoubtedly simple, benevolence is hardly a suitable strategy upon which to base interactions in the world. Any realistic scenario must encompass the possibility of non-helpful (possibly antagonistic) objects/agents. How is one to remain a useful member of such a society without being exploited? A simple solution is to introduce some mechanism for *retaliation*, whereby non-cooperative behaviour is 'punished'. A simple strategy that punishes non-cooperative behaviour can be represented as follows.

> *Qualified benevolence:*
>
> 1. *Initially assume every agent is friendly.*
> 2. *If I have requested something, and an agent has not yet responded, then that agent is not friendly; otherwise it is friendly.*
> 3. *Offer help to any friendly agent that requests it, and not otherwise.*

(Note that this strategy is 'suspicious'; agents are only regarded as friendly if they have offered to help us since we last asked.) It is a simple matter to code this strategy as Concurrent METATEM program rules. First, we need an additional predicate, *friendly(i)*, representing the fact that object '*i*' is regarded as being 'friendly'. Omitting the rules used to ensure that each agent is initially regarded as being friendly, a program skeleton that effects the qualified benevolence strategy then consists of two rules, as shown in Figure 2. Rule (1) in this example defines the conditions under which another object is

```
obj(request)[offer]:
```

$$1.\quad \forall i.\forall a.\quad ((\neg request(self,a))\ \mathcal{Z}\ (\neg request(self,a)) \wedge offer(self,i,a)) \Rightarrow friendly(i);$$

$$2.\quad \forall i.\forall a.\quad \circledcirc(request(i,a) \wedge friendly(i)) \Rightarrow \Diamond offer(i,self,a).$$

Figure 2: The Concurrent METATEM Program for a 'Qualified Benevolent Agent'

considered 'friendly', i.e., any agent that is friendly must have responded positively to the previous requests. Rule (2) is an enhanced version of the 'benevolence' liveness rule, requiring that agents must be friendly for them to be offered assistance.

Obviously, more complex behaviours can be devised and implemented in Concurrent METATEM, particularly those taking account of the asynchronous nature of agents within the system, and the allegiance of agents to various organisations. An important aspect of Concurrent METATEM systems is that, due to the broadcast nature of messages, an agent can observe the messages that other agents send. This can form the basis of both cooperative and competitive scenarios, where an agent can form a strategy based upon what it sees other agents doing.

Lack of space prevents us from giving a larger example of a competitive scenario (but see [12]), however we below outline a simple cooperative system based upon the broadcasting of messages in Concurrent METATEM.

3.2 Distributed Problem Solving

The final example we will look at is taken from [15], and defines a simple, abstract distributed problem solving system. This example is particularly useful for exhibiting the use of interface definitions within objects. Here, a single agent, called *executive*, broadcasts a problem to a group of problem solvers. Some of these problem solvers can solve the particular problem completely, and some will reply with a solution.

`executive(solution1)[problem1,solved1]:`

1. $\textbf{start} \Rightarrow \Diamond \mathit{problem}1;$
2. $\circledcirc \mathit{solution}1 \Rightarrow \mathit{solved}1.$

`solvera(problem2)[solution2]:`

1. $\circledcirc \mathit{problem}2 \Rightarrow \mathit{solution}2.$

`solverb(problem1)[solution2]:`

1. $\circledcirc \mathit{problem}1 \Rightarrow \Diamond \mathit{solution}1.$

`solverc(problem1)[solution1]:`

1. $\circledcirc \mathit{problem}1 \Rightarrow \Diamond \mathit{solution}1.$

Figure 3: A Distributed Problem Solving System

`solverd(problem1,solution1.2)[solution1]:`

1. $(\circledcirc \mathit{solution}1.2 \wedge \blacklozenge \mathit{problem}1) \Rightarrow \Diamond \mathit{solution}1.$

`solvere(problem1)[solution1.2]:`

1. $\circledcirc \mathit{problem}1 \Rightarrow \Diamond \mathit{solution}1.2.$

Figure 4: Additional Problem Solving Agents

We define such a Concurrent METATEM system in Figure 3. Here, *solvera* can solve a different problem from the one *executive* poses, while *solverb* can solve the desired problem, but doesn't announce the fact (as *solution*1 is not a component proposition for *solverb*); *solverc* can solve the problem posed by *executive*, and will *eventually* reply with the solution.

We now look at a refinement of this system, where *solverc* has been removed and replaced by by two agents which together can solve *problem*1, but can not manage this individually. These agents, called *solverd* and *solvere* are defined in Figure 4.

Thus, when *solverd* receives the problem it cannot do anything until it has heard from *solvere*. When *solvere* receives the problem, it broadcasts the fact that it can solve part of the problem (i.e., it broadcasts *solution*1.2). When *solverd* sees this, it knows it can solve the other part of the problem and broadcasts the whole solution.

4 Related Work

The purpose of this section is to place Concurrent METATEM in the context of contemporary DAI research: to show how it stands in relation to other work in the field.

Concurrent METATEM is in some respects similar to Shoham's AGENT0 system [29]. AGENT0 is a first attempt to build an *Agent Oriented Programming* (AOP) language. AOP is a 'new programming paradigm, based on a societal view of computation' [29]: central to AOP is the idea of agents/objects as cognitive entities, whose state is best described in terms of *mentalistic* notions: belief, choice, commitment, and so on[4]. Both AGENT0 and Concurrent METATEM are based on temporal logics, though these logics have quite different forms. In Concurrent METATEM, a tense logic approach is adopted: the language of Concurrent METATEM rules is a classical logic augmented by a set of modal operators for describing the dynamic nature of the world. In AGENT0, the 'method of temporal arguments' is used; the language contains terms for directly referring to time. Predicates and modal operators are then 'date stamped' by the time at which they were true. Both styles of temporal logic have advantages and disadvantages; this paper will not go into the relative merits of each approach. AGENT0 and Concurrent METATEM are both rule-based language, though each makes novel use of the concept of a rule. In both languages, the rules an object/agent possesses determine how that object/agent makes *commitments*. In AOP, commitment is given a mentalistic interpretation, based on the formalisations in [28], [30]. In contrast, Concurrent METATEM gives commitment a precise computational meaning: see §2, above.

Despite these similarities, AGENT0 and Concurrent METATEM differ in many significant respects. An obvious distinguishing feature is the nature of rules in the two languages. In Concurrent METATEM, rules have an explicit logical semantics, and are based on the separated (*past* $\Rightarrow$ *future*) form. In AGENT0, rules do not have such a well-developed logical semantics.

Another system used extensively for DAI is Georgeff and Lansky's Procedural Reasoning System (PRS) [19], [18], which employs some elements of the Belief-Desire-Intention (BDI) architecture partly formalised by Rao [26]. The PRS also has much in common with Concurrent METATEM: both systems maintain 'beliefs' about the world (in Concurrent METATEM these beliefs are past-time temporal formulae); knowledge areas in the PRS loosely resemble Concurrent METATEM rules; and PRS intentions resemble Concurrent METATEM commitments. There are many points of difference, however: notably, the PRS does not have a logical basis for execution that is as elegant as that in Concurrent METATEM. However, the elegance of Concurrent METATEM does not come cheap: the PRS seems to be more flexible in execution than Concurrent METATEM, and is likely to have higher potential in time-critical applications.

The computational model underlying Concurrent METATEM is somewhat similar to that employed in the 'autonomous agent model' of Maruichi *et al.* [22]. More generally, there are also some similarities with the Actor model [20, 2]; the key differences are the ability of objects in Concurrent METATEM to act in a non-message driven way, and the use of broadcast, rather than point-to-point message passing.

Finally, a comment on the use of mentalistic terminology. Both AGENT0 and the PRS make free with terms such as 'belief', 'desire', and 'intention'. Shoham argues that such terminology provides a useful degree of abstraction for complex systems [29, pp5–6]. Although Concurrent METATEM employs terminology such as 'commitment', (see §2, above), no attempt is made to relate this terminology to a deeper theory of agency (as Shoham hopes to do in AOP [29]).

5 Comments and Conclusions

In this paper, we have described Concurrent METATEM: a programming language based on executable logic that is well suited to DAI applications. We have given several examples of prototypical DAI systems and have shown that Concurrent METATEM is useful, particularly for cooperative and/or competitive groups of agents. At the time of writing, a full implementation of propositional Concurrent METATEM has been constructed; experience with this system is being used to guide the development of a full first-order implementation.

Other ongoing and future work topics include: a complete formal definition of the language semantics (see also [12]); techniques for specifying and verifying Concurrent METATEM systems (see also [15]); organisational structure in Concurrent METATEM; (more) efficient algorithms for object execution; meta-level reasoning in METATEM and Concurrent METATEM (see also [4]); and a language for object rules containing a belief component.

[4]We comment on the use of mentalistic notions below.

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, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A Framework for Programming in Temporal Logic. In *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Mook, Netherlands, June 1989. (Published in *Lecture Notes in Computer Science*, volume 430, Springer Verlag).

[4] H. Barringer, M. Fisher, D. Gabbay, and A. Hunter. Meta-Reasoning in Executable Temporal Logic. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Cambridge, Massachusetts, April 1991. Morgan Kaufmann.

[5] T. Bouron, J. Ferber, and F. Samuel. MAGES: A Multi-Agent Testbed for Heterogeneous Agents. In Y. Demazeau and J. P. Muller, editors, *Decentralized AI 2 – Proceedings of the Second European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW)*. Elsevier/North Holland, 1991.

[6] K. Clark and S. Gregory. A Relational Language for Parallel Programming. In E. Shapiro, editor, *Concurrent Prolog–Collected Papers*, chapter 1, pages 9–26. MIT Press, 1987.

[7] D. Connah and P. Wavish. An Experiment in Cooperation. In Y. Demazeau and J. P. Muller, editors, *Decentralized AI — Proceedings of the* 1*st* *European Workshop on Modelling Autonomous Agents in Multi-Agent Worlds (MAAMAW '89)*. Elsevier/North Holland, 1990.

[8] E. Durfee. *Coordination of Distributed Problem Solvers*. Kluwer Academic Press, 1988.

[9] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier, 1990.

[10] J. Ferber and P. Carle. Actors and Agents as Reflective Concurrent Objects: a MERING IV Perspective. *IEEE Transactions on Systems, Man and Cybernetics*, December 1991.

[11] M. Finger, M. Fisher, and R. Owens. METATEM at Work: Modelling Reactive Systems Using Executable Temporal Logic. *Submitted to the Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-93)*, 1992.

[12] M. Fisher. Concurrent METATEM — A Language for Modeling Reactive Systems. Submitted to *Parallel Architectures and Languages, Europe (PARLE-93)*, 1992.

[13] M. Fisher and H. Barringer. Concurrent METATEM Processes — A Language for Distributed AI. In *Proceedings of the European Simulation Multiconference*, Copenhagen, Denmark, June 1991.

[14] M. Fisher and R. Owens. From the Past to the Future: Executing Temporal Logic Programs. In *Proceedings of Logic Programming and Automated Reasoning (LPAR)*, St. Petersberg, Russia, July 1992. (Published in *Lecture Notes in Computer Science*, volume 624, Springer Verlag).

[15] M. Fisher and M. Wooldridge. Specifying and Verifying Distributed Intelligent Systems. Submitted to *European Symposium on Validation and Verification of Knowledge Based Systems (EUROVAV-93)*, 1992.

[16] M. Fujita, S. Kono, T. Tanaka, and T. Moto-oka. Tokio: Logic Programming Language based on Temporal Logic and its compilation into Prolog. In *3rd International Conference on Logic Programming*, pages 695–708, July 1986. (Published at Lecture Notes in Computer Science, vol 225, Springer-Verlag.).

[17] L. Gasser, C. Braganza, and N. Hermann. MACE: A Flexible Testbed for Distributed AI Research. In M. Huhns, editor, *Distributed Artificial Intelligence*. Pitman/Morgan Kaufmann, 1987.

[18] M. P. Georgeff and F. F. Ingrand. Decision-Making in an Embedded Reasoning System. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI)*, Detroit, USA, 1989. Morgan Kaufmann.

[19] M. P. Georgeff and A. L. Lansky. Reactive Reasoning and Planning. In *Proceedings of the American Association for Artificial Intelligence (AAAI)*. Morgan Kaufmann, 1987.

[20] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[21] L. P. Kaelbling. An Architecture for Intelligent Reactive Systems. In M. P. Georgeff and A. L. Lansky, editors, *Proceedings of the 1986 Workshop on Reasoning About Actions and Plans*. Morgan Kaufmann, 1986.

[22] T. Maruichi, M. Ichikawa, and M. Tokoro. Modelling Autonomous Agents and their Groups. In Y. Demazeau and J. P. Muller, editors, *Decentralized AI – Proceedings of the First European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW)*. Elsevier/North Holland, 1990.

[23] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, U.K., 1986.

[24] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the Eighteenth Symposium on the Foundations of Computer Science*, 1977.

[25] A. Pnueli. Specification and Development of Reactive Systems. In *Information Processing '86*. Elsevier/North Holland, 1986.

[26] A. S. Rao and M. P. Georgeff. Modeling Agents within a BDI-Architecture. In R. Fikes and E. Sandewall, editors, *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Cambridge, Massachusetts, April 1991. Morgan Kaufmann.

[27] E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. In E. Shapiro, editor, *Concurrent Prolog–Collected Papers*, chapter 29, pages 251–273. MIT Press, 1987.

[28] Y. Shoham. Time for Action: on the relation between time, knowledge and action. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI)*, Detroit, USA, 1989. Morgan Kaufman.

[29] Y. Shoham. Agent Oriented Programming. Technical Report STAN–CS–1335–90, Department of Computer Science, Stanford University, California, USA, 1990.

[30] B. Thomas, Y. Shoham, A. Schwartz, and S. Kraus. Preliminary Thoughts on an Agent Description Language. *International Journal of Intelligent Systems*, 6:497–508, 1991.