A Logical Approach to Simulating Societies

Michael Fisher and Michael Wooldridge

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

EMAIL: {michael,mikew}@sun.com.mmu.ac.uk

Abstract

This paper describes a novel language for programming societies of intelligent artificial agents. In this language, called Concurrent METATEM, individual agents are programmed by giving them a formal, temporal logic specification of their desired behaviour. Each agent then directly executes its specification. We motivate and describe the language in some detail, and go on to demonstrate, through the use of examples, how it may be used to simulate competitive and cooperative systems.

1 Introduction

A number of languages and testbeds for implementing and simulating artificial social systems have been described in the literature of Distributed Artificial Intelligence (DAI) and its related disciplines; see, for example (Hewitt, 1977; Gasser et al., 1987; Doran et al., 1991; Bouron et al., 1991; Agha, 1986; Ferber and Carle, 1991; Shoham, 1990). However, with the possible exception of pure Actor languages ((Hewitt, 1977; Agha, 1986)), these tools have relied on an informal, if not ad hoc notion of agency, with similarly informal techniques for programming agents. As a result, it is difficult to reason, either formally or informally, about the expected behaviour of systems implemented using such tools. It is similarly difficult to derive an a posteriori explanation of such a system’s behaviour. In this paper, we present a language for simulating cooperative and competitive behaviour in groups of intelligent artificial agents. In contrast to the informal languages and testbeds referred to above, this language, called Concurrent METATEM, has a simple, well-motivated logical (and hence mathematical) foundation. In relative terms, it is easy to reason about Concurrent METATEM systems at both formal and informal levels (Fisher and Wooldridge, 1993). Concurrent METATEM is, so far as we are aware, unique in DAI in having such a well developed and motivated logical foundation.

The remainder of this paper is structured as follows. In §1.1, we motivate the language and discuss its origins in more detail. In §2 we present a more detailed, though essentially non-technical introduction to the language and its execution. In §3 we demonstrate, through the use of examples, how the language may be used to model societies. Finally, in §4, we
describe the current state of the language, and describe our intentions with respect to future
developments.

1.1 Motivation

Distributed AI is a relatively young discipline, which has developed its own tools for building,
experimenting, and evaluating theories and applications. Over the past decade, many frame-
works for building DAI systems have been reported. While some, such as the DVMT, allow
experimentation only with one specific scenario (Durfee, 1988), and others provide extensions
to existing AI languages such as LISP (Gasser et al., 1987), 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 (Hewitt, 1977; Agha,
1986), for example (Ferber and Carle, 1991) and (Bouron et al., 1991).

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 is because DAI systems are a subset of the class of systems
known in mainstream computer science as reactive systems. A reactive system in this sense
is one whose purpose is to maintain some ongoing interaction with its environment. (We
therefore do not use the term ‘reactive system’ in the way that it has become fashionable
to use it in AI, to refer to systems that respond directly to their environment – perhaps using
‘situation-action’ rules – without reasoning in any way.) All concurrent or distributed systems
are reactive in the sense we intend, as a module or agent in a concurrent system must maintain
some interaction with the other components in the system (see, e.g., (Pnueli, 1986) for a
discussion of this point). It is for this reason that we claim all DAI systems are reactive.

Contemporary DAI testbeds and languages are usually built as extensions to the classic AI
languages LISP and PROLOG. However, neither of these languages is well suited to building
reactive systems. They are both based on a quite different view of systems, called the func-
tional or relational view. For this reason, we suggest that an obvious development for DAI is
to build languages based on a more reactive view of systems; Concurrent METATEM is such
a language.

In a 1977 paper, Pnueli proposed temporal logic as a tool for reasoning about reactive
systems (Pnueli, 1977): when describing a reactive system, we often wish to express prop-
erties such as ‘if a request is sent, then a response is eventually given’. Such properties are
easily and elegantly expressed in temporal logic. However, proving the properties of reactive
systems using temporal logics is not a trivial matter. The difficulties involved in this
process led, in the early 1980s, to the idea of using temporal logic itself as a programming
language (Moszkowski, 1986). This ideal led directly to the METATEM concept (Barringer
et al., 1989). METATEM is a general framework for executing temporal logic specifications,
where these specifications are expressed as a set of temporal logic ‘rules’. The concept of
a reactive system is therefore at the very heart of METATEM. Although the original META-
TEM proposal did not address the issue of concurrency, the potential value of concurrently
executing METATEM systems – particularly for DAI applications – was immediately recog-
nised (Fisher and Barringer, 1991). Concurrent METATEM is a simple operational framework
which allows societies of METATEM processes to communicate and cooperate.

Note that although Concurrent METATEM may be regarded as a logic programming lan-
guage, (in that it has a well developed and motivated logical foundation), it is quite unlike any
other logic programming language with which we are familiar, and in particular, it 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., (Clark and Gregory, 1987)), concurrency in Concurrent METATEM is achieved via coarse-grained computational entities called agents; each agent is a METATEM process.

2 Concurrent METATEM

In Concurrent METATEM (Fisher, 1993), the behaviour of an agent is defined using a temporal logic formula. Temporal logic is used as, not only is it an ideal formalism in which to represent the dynamic properties of an agent, but it also contains an explicit mechanism for representing and executing goals. As well as providing a declarative description of the agent’s desired behaviour, the temporal formula can be executed directly to implement the agent (Fisher and Owens, 1992). Thus, the basic behaviour of an agent consists of following a set of temporal rules representing the basic dynamics of the agent, introducing new goals, and attempting to satisfy existing goals.

Agents communicate via message-passing. As well as its temporal specification, each agent records two sets of message-types – one representing messages that it is able to send, the other representing messages that it is willing to receive. When an agent sends a message, this message is broadcast. When a message arrives at an agent, the agent will only process that message if it is one of the types of message that it is ‘listening’ for. Agents are truly autonomous; not only do they only react to messages that they want to ‘hear’, but they are able to dynamically change the set of message-types that they will recognise, and are able to control the number of messages consumed at any one time.

Agents are members of groups. Each agent may be a member of several groups. If an agent sends a message, then that message is broadcast to all members of its group(s), but to no other agents. Thus, the basic model of communication in Concurrent METATEM is of agents using broadcast locally, while using more selective methods for non-local communication (for example, via ‘eavesdropping’ agents that are members of multiple groups).

Agents within groups can achieve close interaction. In particular, the use of broadcast communication allows individual agents to observe the messages that other agents send, and thus to modify their behaviour accordingly. Using this type of approach, varieties of cooperative and competitive behaviours can easily be modelled.

In the following subsections, we shall give a more detailed introduction to Concurrent METATEM. We begin with an informal introduction to the type of temporal logic that Concurrent METATEM is based on.

2.1 Temporal Logic

Temporal logics are classical logics augmented by a set of modal operators through which it is possible to describe the time-varying state of the world. Although there are many different types of temporal logic (Emerson, 1990), in this paper we shall consider only one. This logic is based on a linear, discrete model of time, which is bounded in the past and infinite in the future. This means that there is only one ‘timeline’; that time comes in ‘atoms’, such that each moment in time has a unique predecessor and successor; that there was some moment in the past at which time ‘began’; and that time stretches infinitely into the future.
The logic we consider is based on classical first-order logic. In addition to the usual connectives and quantifiers of this language, the logic contains a number of temporal operators. First there are the future time connectives. If \( \varphi \) is a formula of the temporal language, then \( \Diamond \varphi \) is also a formula, which will be satisfied now if \( \varphi \) is satisfied now or at some time in the future. Another unary connective is \( \blacksquare \): the formula \( \blacksquare \varphi \) will be satisfied now if \( \varphi \) is satisfied now and at all times in the future; i.e., now and forever more. Since each moment in time has a unique successor, we can introduce a ‘next-time’ connective: \( \bigcirc \varphi \) will be satisfied now if \( \varphi \) is satisfied in the next moment. There are also some binary connectives in the language: \( \varphi \mathcal{U} \psi \) will be satisfied now if \( \psi \) is satisfied in some future time, and at all time points until that time, \( \varphi \) is satisfied.

The language also contains temporal operators for talking about things that have occurred in the past. The \( \Diamond \) operator mirrors the behaviour of \( \Diamond \) in the past, so \( \Diamond \varphi \) will be satisfied now if \( \varphi \) was satisfied in some prior moment. The \( \blacksquare \) operator mirrors \( \blacksquare \) in the past, so \( \blacksquare \varphi \) will be satisfied now if \( \varphi \) was satisfied at all prior times. It is possible to talk about the previous moment in time: \( \bigcirc \varphi \) will be satisfied if there was a previous moment in time, and \( \varphi \) was satisfied in that moment. (Note that the possibility of ‘now’ being the first moment in time complicates the meaning of this ‘last time’ operator somewhat, and so a special nullary operator, ‘\text{$\text{start}$}’, is used to denote the first moment in time.) Finally, there is a binary ‘since’ connective: \( \varphi \mathcal{S} \psi \) will be satisfied now if \( \psi \) was satisfied at some prior moment in time, and at all times since then, \( \varphi \) has been satisfied.

2.2 Agents

A Concurrent MetateM system contains a number of concurrently executing agents, which are able to communicate through asynchronous broadcast message passing. Each agent is programmed by giving it a temporal logic specification of the behaviour it is required to exhibit. Agents directly execute their specifications, where a specification consists of a set of ‘rules’, which are temporal logic formulae of the form

\[ \text{antecedent about past} \Rightarrow \text{consequent about future.} \]

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

Inter-agent communication is managed by interfaces, which each agent possesses. An interface determines what messages an agent may send, and what messages, if sent by another agent, it will accept. Whenever an agent 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 agents. On receipt of a message, an agent 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.

For example, the interface for a ‘stack’ agent might be defined as follows.

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

During the execution of each agent’s rules, the two types of predicate have 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 ‘\text{request}(x)’, where \(x\) is a universally quantified variable, is only satisfied if a message of the form ‘\text{request}(b)’ has just been received (for some argument ‘\(b\)’).

- **Component predicates represent messages broadcast from the agent.** When a component predicate is satisfied, it has the (side-)effect of broadcasting the corresponding message to the environment. For example, if the formula ‘\text{offer}(e)’ is satisfied, where \text{offer} is a component predicate, then the message ‘\text{offer}(e)’ is broadcast.

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

Once an agent 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. Individual agents execute asynchronously, and are autonomous in that they may execute independently of incoming messages and may change their interface dynamically.

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

Finally, agents are also members of groups. Each agent may be a member of several groups. When an agent sends a message, that message is, by default, broadcast to all the members of its group(s), but to no other agents. Alternatively an agent can select to broadcast only to certain groups (of which it is a member). This mechanism allows the development of complex structuring within the agent space and provides the potential for innovative applications, such as the use of groups to represent physical properties of agents. For example, if we assume that any two agents in the same group can ‘see’ each other, then movement broadcast from one agent can be detected by the other agent. Similarly, if an agent moves ‘out of sight’, it moves out of the group and thus the agents that remain in the group ‘lose sight’ of it. Examples such as this (which will not be explored further here) show some of the power of the ‘group’ concept.
3 Examples

In this section we will give a range of examples, starting with collections of agents acting individually and gradually adding structure and the potential for more complex interactions amongst agents and groups of agents. We will begin with a simple scenario, taken from (Fisher, 1993), where seven agents are each attempting to get a resource from a single agent.

3.1 Snow White and The Seven Dwarves – A tale of 8 agents

To give some idea of how Concurrent METATEM can be used, a simple example showing agents with a form of ‘intelligent’ behaviour will be described. First, a brief outline of the properties of the leading characters in this example will be given.

The Scenario

Snow White has a bag of sweets. All the dwarves want sweets, though some want them more than others. If a dwarf asks Snow White for a sweet, she will give him one, but maybe not straight away. Snow White is only able to give away one sweet at a time.

Snow White and the dwarves are going to be represented as a set of agents in Concurrent METATEM. Each dwarf has a particular strategy that it uses in asking for sweets, which is described below.

1. eager initially asks for a sweet and, from then on, whenever he receives a sweet, asks for another.

2. mimic asks for a sweet whenever he sees eager asking for one.

3. jealous asks for a sweet whenever he sees eager receiving one.

4. insistent asks for a sweet as often as he can.

5. courteous asks for a sweet only when eager, mimic, jealous and insistent have all asked for one.

6. generous asks for a sweet only when eager, mimic, jealous, insistent and courteous have all received one.

7. shy only asks for a sweet when he sees no one else asking.

8. snow-white can only allocate one sweet at a time. She keeps a list of outstanding requests and attempts to satisfy the oldest one first.

   If a new request is received, and it does not occur in the list, it is added to the end. If it does already occur in the list, it is ignored. Thus, if a dwarf asks for a sweet \( n \) times, he will eventually receive at most \( n \), and at least 1, sweets.

This example may seem trivial, but it represents a set of agents exhibiting different behaviours, where an individual agent’s internal rules can consist of both safety and liveness constraints, and where complex interaction can occur between autonomous agents.

\footnote{The names and behaviours of the dwarves are different from the Fairy Tale!}
The Program

The Concurrent METATEM program for the scenario described above consists of the definitions of 8 agents, given below. To give a better idea of the meaning of the temporal formulae representing the internals of these agents, a brief description will be given with each agent’s definition. Requests to Snow White are given in the form of an ask() message with the name of the requesting dwarf as an argument. Snow White gives a sweet to a particular dwarf by sending a give() message with the name of the dwarf as an argument. Finally, uppercase alphabetic characters, such as X and Y represent universally quantified variables.

1. \(\text{eager}(\text{give})[\text{ask}]\):
   \[
   \begin{align*}
   \text{start} & \Rightarrow \text{ask(eager)} \\
   \bigotimes \text{give(eager)} & \Rightarrow \text{ask(eager)}
   \end{align*}
   \]
   Initially, eager asks for a sweet and, whenever he has just received a sweet, he asks again.

2. \(\text{mimic}(\text{ask})[\text{ask}]\):
   \[
   \bigotimes \text{ask(eager)} \Rightarrow \text{ask(mimic)}
   \]
   If eager has just asked for a sweet then mimic asks for one.

3. \(\text{jealous}(\text{give})[\text{ask}]\):
   \[
   \bigotimes \text{give(eager)} \Rightarrow \text{ask(jealous)}
   \]
   If eager has just received a sweet then jealous asks for one.

4. \(\text{insistent}[\text{ask}]\):
   \[
   \text{start} \Rightarrow \square \text{ask(insistent)}
   \]
   From the beginning of time, insistent asks for a sweet as often as he can.

5. \(\text{courteous}(\text{ask})[\text{ask}]\):
   \[
   \begin{align*}
   \neg \text{ask(courteous)} & \Rightarrow \text{ask(eager)} \\
   \neg \text{ask(courteous)} & \Rightarrow \text{ask(mimic)} \\
   \neg \text{ask(courteous)} & \Rightarrow \text{ask(jealous)} \\
   \neg \text{ask(courteous)} & \Rightarrow \text{ask(insistent)}
   \end{align*}
   \]
   If courteous has not asked for a sweet since eager asked for one, has not asked for a sweet since mimic asked for one, has not asked for a sweet since jealous asked for one, and, has not asked for a sweet since insistent asked for one, then he will ask for a sweet.

6. \(\text{generous}(\text{give})[\text{ask}]\):
   \[
   \begin{align*}
   \neg \text{ask(generous)} & \Rightarrow \text{give(eager)} \\
   \neg \text{ask(generous)} & \Rightarrow \text{give(mimic)} \\
   \neg \text{ask(generous)} & \Rightarrow \text{give(jealous)} \\
   \neg \text{ask(generous)} & \Rightarrow \text{give(insistent)} \\
   \neg \text{ask(generous)} & \Rightarrow \text{give(courteous)}
   \end{align*}
   \]
   If generous has not asked for a sweet since eager received one, has not asked for a sweet since mimic received one, has not asked for a sweet since jealous received one, has not asked for a sweet since insistent received one, and, has not asked for a sweet since courteous received one, then he will ask for a sweet!
7. \textit{shy(ask)[ask]}:

\begin{align*}
\text{start} & \Rightarrow \square \text{ask(\textit{shy})} \\
\square \text{ask(X)} & \Rightarrow \neg \text{ask(\textit{shy})} \\
\diamond \text{ask(\textit{shy})} & \Rightarrow \square \text{ask(\textit{shy})}
\end{align*}

\textit{shy} initially wants to ask for a sweet but is prevented from doing so whenever he sees some other dwarf asking for one. Thus, he only succeeds in asking for one when he sees no one else asking and, as soon as he has asked for a sweet, he wants to try to ask again!

8. \textit{snow-white(ask)[give]}:

\begin{align*}
\diamond \text{ask(X)} & \Rightarrow \square \text{give(X)} \\
\text{give(X)} \land \text{give(Y)} & \Rightarrow X = Y
\end{align*}

If \textit{snow-white} has just received a request from a dwarf, a sweet will be sent to that dwarf eventually. The second rule ensures that sweets can not be sent to two dwarves at the same time by stating that if both \text{give(X)} and \text{give(Y)} are to be broadcast, then \(X\) must be equal to \(Y\).

Note that, in this example, several of the dwarves were only able to behave as required because they could observe all the \text{ask()} and \text{give()} messages that were broadcast. The dwarves can thus be programmed to have strategies that are dependent on the behaviour of other dwarves. Also, the power of executable temporal logic is exploited in the definition several agents, particularly those using the \(\square\) operator to represent multiple goals.

We also note that, as the agents behaviour is represented explicitly, and in a logical way, the verification of properties of the system is possible. For example, given the agents’ definitions, we are able to prove that every dwarf, except ‘\textit{shy}’ will eventually receive a sweet. (For further work on the verification of properties of such systems, see \cite{Fisher and Wooldridge, 1993}.)

3.2 Adding ‘Money’

We now extend the type of example given above to incorporate the notion of some resource which the dwarves can attempt to exchange with Snow White for sweets, i.e. money.

**Bidding** Initially, we will simply change the \text{ask} predicate so that it takes an extra argument representing the amount the dwarf is willing to pay for a sweet. This enables dwarves to ‘bid’ for a sweet, rather than just asking for one. For example, \textit{dwarf1} below asks for a sweet, bidding ‘2’.

\begin{align*}
\textit{dwarf1()[ask]}:
\text{start} & \Rightarrow \text{ask(dwarf1,2)}
\end{align*}

We can further modify a dwarf’s behaviour so that it doesn’t bid more than it can afford by introducing some record of the amount of money that the dwarf has at any one time. Thus, the main rule defining the ‘bidding’ behaviour of a dwarf might become something like

\begin{align*}
\square \left[\text{money(N)} \land N \geq 2\right] & \Rightarrow \text{ask(dwarf1,2)}
\end{align*}

Note that the behaviour of Snow White might also change so that all the bids are recorded then a decision over which bid to accept is made based upon the bids received. Once a decision is
made, \texttt{give} is again broadcast, but this time having an extra argument showing the amount paid for the sweet. For example, if Snow White accepts the bid of `2' from \texttt{dwarf1}, then \texttt{give(dwarf1,2)} is broadcast.

Finally, a dwarf whose bid has been accepted, in this case \texttt{dwarf1}, must remember to record the change in finances:

\[
\Diamond [\text{money}(N) \land \text{give(dwarf1,C)}] \Rightarrow \text{money}(N-C)
\]

**Renewable Resources** Dwarves who keep buying sweets will eventually run out of money. Thus, we may want to add the concept of the renewal of resources, i.e., being paid. This can either happen at a regular period defined within each dwarf’s rules, e.g.

\[
\text{start} \Rightarrow \text{money}(100) \land \text{paid}
\]

\[
\Diamond [\text{money}(N) \land \Diamond \Diamond \Diamond \Diamond \Diamond \text{paid}] \Rightarrow \text{money}(N+100) \land \text{paid}
\]

or the dwarf can replenish its resources when it receives a particular message from its environment, e.g.

\[
dwarf1(\text{go})[\text{ask}]:
\]

\[
\text{start} \Rightarrow \text{money}(100)
\]

\[
\Diamond (\text{money}(N) \land \text{go}) \Rightarrow \text{money}(N+100)
\]

**Competitive Bidding** As the bids that individual dwarves make are broadcast, other dwarves can observe the bidding activity and can revise their bids accordingly. We saw earlier that the ‘mimic' dwarf asks for a sweet when it sees the ‘eager' dwarf asking for one. Similarly, \texttt{dwarf2} might watch for any bids by \texttt{dwarf1} and then bid more, e.g.,

\[
\Diamond [\text{ask(dwarf1,B)} \land \text{myhigh}(M) \land B > M] \Rightarrow \text{ask(dwarf2,B+1)} \land \text{myhigh}(B+1)
\]

Although we will not give further detailed examples in this vein, it is clear that a range of complex behaviours based upon observing others’ bids can be defined.

### 3.3 Cooperation

In the last section we showed how individual dwarves might compete with each other for Snow White’s sweets. Here, we will consider how dwarves might \textit{cooperate} in order to get sweets from Snow White. In particular, we consider the scenario where one dwarf on its own does not have enough money to buy a sweet, and thus requires a loan from other dwarves.

**Borrowing Money** In order to borrow money from other dwarves to enable a single dwarf to buy a sweet, the dwarf can broadcast a request for a certain amount. For example, if the dwarf (\texttt{dwarf3} in this case) knows that the highest amount bid for a sweet so far is \texttt{X} and he only has \texttt{Y}, then he can ask to borrow \texttt{X-Y}, possibly as follows.

\[
dwarf3(\text{lend})[\text{borrow,ask}]:
\]

\[
\Diamond [\text{highest}(X) \land \text{money}(Y) \land X > Y] \Rightarrow \text{borrow(dwarf3,}(X-Y)+1)
\]
Now, if another dwarf, say dwarf4, offers to lend a certain amount, say Z, to dwarf3, then another rule recording the loan must be added to dwarf3’s rule-set:

\[ [\text{lend} \text{(dwarf4, dwarf3, Z)} \land \text{money}(Y)] \implies \text{money}(Y+Z) \land \text{owe} \text{(dwarf4, Z)} \]

**Lending Behaviour**  Dwarves might have various strategies of lending and borrowing money. For example, perhaps a dwarf won’t lend any more money to any dwarf who still owes money. Further, a dwarf might be less likely to lend money to any dwarf who has never offered to help his previous requests.

Again, a variety of strategies for lending and borrowing in this way can be coded in Concurrent METATEM. Rather than giving further examples of this type, we next consider the use of groups in the development of structured systems of interacting agents.

### 3.4 Group Structuring

As described earlier, as well as the notion of autonomous objects, Concurrent METATEM also provides a larger structuring mechanism through ‘groups’. This restricts the extent of an object’s communications and thus provides an extra mechanism for the development of strategies for organisations. Rather than giving detailed examples, we will outline how the group mechanism could be used in Concurrent METATEM to develop further cooperation, competition and interaction amongst agents.

Again, we will consider a scenario similar to Snow White and the Seven Dwarves described above, but will assume the existence of a large number of dwarves, and possibly several Snow White’s! We will outline several examples of how the grouping of these agents can be used to represent more complex or refined behaviour.

**Collective Bidding**  If we again have a situation where dwarves bid for sweets, then we can organise cooperation within groups so that the group as a whole puts together a bid for a sweet. If successful, the group must also decide who to distribute the sweet to.

Thus, a number of groups might be cooperating internally to generate bids, but competing (with other groups) to have their bid accepted.

**Forming Subgroups**  Within a given group, various subgroups can be formed. For example, if several members of a group are unhappy with another member’s behaviour, they might be able to create a new subgroup within the old grouping which excludes the unwanted agent. Note that members of the subgroup can hear the outer group’s communications, while members of the outer one cannot hear the inner group’s communications.

Although we have described this as a retributive act, such dynamic restructuring is natural as groups increase in size.

As we have seen above, by using a combination of individual agent strategies and of grouping agents together, we are able to form simple societies. In particular, we can represent societies where individuals cooperate with their fellow group members, but where the groups themselves cooperate for some global resource.
Although our examples have been based upon agents competing and cooperating in order to get a certain resource, many other types of multi-agent system can be developed in Concurrent MetateM.

Finally, it is important to note that there is no explicit global control or global plan in these examples. Individual agents perform local interactions with each other and their environment.

4 Related Work

Concurrent MetateM is in some respects similar to Shoham’s AGENT0 system (Shoham, 1990). 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’ (Shoham, 1990): central to AOP is the idea of agents/agents as cognitive entities, whose state is best described in terms of mentalistic notions: belief, choice, commitment, and so on2.

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 agent possesses determine how that agent makes commitments. In AOP, commitment is given a mentalistic interpretation, based on the formalisations in (Shoham, 1989), (Thomas et al., 1991). 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 ⇒ 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) (Georgeff and Lansky, 1987), (Georgeff and Ingrand, 1989), which employs some elements of the Belief-Desire-Intention (BDI) architecture partly formalised by Rao (Rao and Georgeff, 1991).

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. (Maruichi et al., 1990). More

2We comment on the use of mentalistic notions below.
generally, there are also some similarities with the Actor model (Hewitt, 1977; Agha, 1986); the key differences are the ability of agents 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 (Shoham, 1990, 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 (Shoham, 1990)).

5 Comments and Conclusions

In this paper, we have described the use of Concurrent Metatem, a programming language based on executable logic, in representing and simulating simple societies of artificial agents. We have given several examples of prototypical societies and have shown that Concurrent Metatem is useful, particularly for cooperative and/or competitive groups of agents. The use of a formal language for representing agents’ behaviours has the added advantage that the verification of properties of such societies becomes possible.

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 (Finger et al., 1992), 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. Several features, such as the grouping mechanism described above and the extension of individual agents’ capabilities to incorporate meta-level statements, are still under development.

Other ongoing and future work topics include: a complete formal definition of the language semantics (see also (Fisher, 1993)); techniques for specifying and verifying Concurrent Metatem systems (see also (Fisher and Wooldridge, 1993)); organisational structure in Concurrent Metatem; (more) efficient algorithms for agent execution; meta-level reasoning in Metatem and Concurrent Metatem (see also (Barringer et al., 1991)); a language for agent rules containing a belief component; and the development of a formal framework for the group structuring notion.
References


