SPECIFYING AND EXECUTING PROTOCOLS FOR COOPERATIVE ACTION

Michael Fisher and Michael Wooldridge

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

{M.Fisher, M.Wooldridge}@mmu.ac.uk

ABSTRACT

The purpose of this paper is twofold: (i) to illustrate and re-emphasize the use of CONCURRENT METATEM, a programming language based on executable temporal logic, as a viable framework in which to develop Distributed Artificial Intelligence (DAI) applications; and (ii) to present a specific example of a cooperative protocol, give an outline of its implementation in CONCURRENT METATEM, and show how properties of the protocol can be established through formal proof.

1 INTRODUCTION

An important activity in DAI is the construction of *protocols for cooperative action* [3]. Such protocols establish a common frame of reference in which agents can successfully cooperate. They identify both the rules that must be followed in order for cooperation to begin (such as a common syntax for communication languages) and more general rules that determine how agents should behave in various situations. Cooperation protocols are typically designed in order to maximise *coordination* and *coherence* [2, pp19–25]. Probably the best known example of such a protocol is the Contract Net [12]; another well known example is Durfee's partial global planning framework [4].

Although there are analogies between cooperation protocols in DAI and protocols in the better-known network sense, these analogies cannot be pushed too far. Perhaps the most important distinction is that when we specify a network protocol, we aim to completely define the possible state transitions, and indeed these transitions are often represented in some kind of state table, so that a network node can simply use a table lookup procedure to determine its next actions. There is no scope for flexibility, compromise, or non-determinism in such protocols — in general, it is precisely such 'ambiguity' that one aims to avoid. In contrast, the DAI system builder typically aims to make a cooperation protocol as flexible as possible. While the protocol will define *general* rules, agents are typically given the ability to make *reasoned choices* about what actions to perform, based on their own goals, preferences, and allegiances. The ability to make such choices is essential in complex, uncertain, or highly dynamic domains, in which rigid protocols over-constrain the behaviour of agents. Non-determinism is, therefore, a necessary component of good cooperation protocols[1].

One of the problems associated with building a DAI system is that the properties of a cooperation protocol are often hard-wired into a procedural implementation; they are generally not represented explicitly. This makes it difficult to understand or predict how the system will behave in a given situation [10], and makes the formal proof of properties very difficult. To address these issues, we have developed a DAI programming language called CONCURRENT METATEM [5]. A CONCURRENT METATEM system contains a number of concurrently executing objects, able to communicate through asynchronous broadcast message passing. Each object is programmed by giving it a temporal logic specification of the behaviour that it is intended the object should exhibit. Objects generate their behaviour by directly executing their specification. We believe that temporal logic is an ideal tool for representing cooperative protocols [7]: it is an expressive language, well suited to representing the kind of reasoned choice and non-determinism that the DAI system builder aims for. Moreover, the use of a language with a well-defined logical semantics makes the formal verification of properties of cooperative protocols a realistic possibility [8]. In the remainder of this paper, we hope to illustrate and justify these claims. We begin, in §2, with an overview of the CONCURRENT METATEM language (note that this paper does not aim to give a complete introduction either to CONCURRENT METATEM or its underlying temporal logic; for such introductions, the reader is referred to [1, 5, 7]). In §3, we introduce a cooperative protocol, based on the Contract Net [12], which we formalise by expressing it in the CONCURRENT METATEM language. This formalisation serves as an *executable specification* of the protocol. In §4, we show how the properties of the protocol may be derived through formal proof. Finally, some conclusions are presented in §5.

2 CONCURRENT METATEM

A CONCURRENT METATEM system contains a number of concurrently executing *objects*, which are able to communicate through asynchronous broadcast message passing. Each object directly executes a temporal logic specification, given to it as a set of

[1]Ultimately, DAI aims to build systems in which agents can organise themselves, and devise their own cooperation protocols — some preliminary work on generic configurable cooperation protocols is reported in [3]. The notions of trust, honesty, commitments, and conventions may well play an important role in systems which are capable of such behaviour [11].

'rules'. In this section, we describe objects and their execution in more detail. Each object has two main components:

- an *interface*, which defines how the object may interact with its environment (i.e., other objects);
- a *computational engine*, which defines how the object may act.

An object interface consists of three components:

- a unique *object identifier* (or just object id), which names the object;
- a set of symbols defining what messages will be accepted by the object — these are called *environment predicates*;
- a set of symbols defining messages that the object may send — these are called *component predicates*.

For example, the interface definition of a 'stack' object might be [5]:

$$stack(pop, push)[popped, stackfull]$$

Here, *stack* is the object id that names the object, $\{pop, push\}$ are the environment predicates, and $\{popped, stackfull\}$ are the component predicates. Whenever a message headed by the symbol *pop* is broadcast, the *stack* object will *accept* the message; we describe what this means below. If a message is broadcast which is not declared in the *stack* object's interface, then *stack* ignores it. Similarly, the only messages which can be sent by the *stack* object are headed by the symbols *popped* and *stackfull*. (All other predicates used within the object are *internal* predicates, which have no external correspondence.)

The computational engine of an object is based on the METATEM paradigm of executable temporal logics. The idea which informs this approach is that of directly executing a declarative object specification, where this specification is given as a set of *program rules*, which are temporal logic formulae of the form:

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

The past-time antecedent is a temporal logic formula referring to the past, whereas the future time consequent is a temporal logic formula referring either to the present or future. The intuitive interpretation of such a rule is 'on the basis of the past, do the future', which gives rise to the name of the paradigm: *declarative past and imperative future* [9]. The actual execution of an object is, superficially at least, very simple to understand. Each object obeys a cycle of trying to match the past time antecedents of its rules against a *history*, and executing the consequents of those rules that 'fire'.

To make the above discussion more concrete, we will now informally introduce a first-order temporal logic, called First-Order METATEM Logic (FML), in which individual METATEM rules will be given. FML is essentially classical first-order logic augmented by a set of modal connectives for referring to the *temporal ordering* of actions. FML is based on a model of time that is *linear* (i.e., each moment in time has a unique successor), *bounded in the past* (i.e., there was a moment that was the 'beginning of time'), and *infinite in the future* (i.e., there are an infinite number of moments in the future). The temporal connectives of FML can be divided into two categories, as follows.

1. Strict past time connectives: '$\odot$' (last), '$\blacklozenge$' (was), '$\blacksquare$' (heretofore), '$\mathcal{S}$' (since) and '$\mathcal{Z}$' (zince, or weak since).

2. Present and future time connectives: '$\bigcirc$' (next), '$\Diamond$' (sometime), '$\Box$' (always), '$\mathcal{U}$' (until) and '$\mathcal{W}$' (unless).

The connectives $\{\odot, \blacklozenge, \blacksquare, \bigcirc, \Diamond, \Box\}$ are unary; the remainder are binary. In addition to these temporal connectives, FML contains the usual classical logic connectives.

The meaning of the temporal connectives is quite straightforward, with formulae being interpreted at a particular moment in time. Let φ and ψ be formulae of FML. Then $\bigcirc\varphi$ is true (satisfied), at the current moment in time if φ is true at the next moment in time; $\Diamond\varphi$ is true now if φ is true now or at some future moment in time; $\Box\varphi$ is true now if φ is true now and at all future moments; $\varphi\mathcal{U}\psi$ is true now if ψ is true at some future moment, and φ is true until then — $\mathcal{W}$ is a binary connective similar to $\mathcal{U}$, allowing for the possibility that the second argument never becomes true.

The past-time connectives are similar: $\odot\varphi$ is true now if there was a previous moment of time and φ was true at that moment in time; $\blacklozenge\varphi$ will be true now if φ was true at some previous moment in time; $\blacksquare\varphi$ will be true now if φ was true at all previous moments in time; $\varphi\mathcal{S}\psi$ will be true now if ψ was true at some previous moment in time, and φ has been true since then; $\mathcal{Z}$ is the same, but allowing for the possibility that the second argument was never true. Finally, a temporal operator that takes no arguments can be defined which is true only at the first moment in time: this useful operator is called '**start**'.

Any formula of FML which refers strictly to the past is called a *history formula*; any formula referring to the present or future is called a *commitment formula*; any formula of the form

$$\textit{history formula} \Rightarrow \textit{commitment formula}$$

is called a *rule*; an object specification is a set of such rules.

A more precise definition of object execution will now be given. Objects continually execute the following cycle:

1. Update the *history* of the object by receiving messages from other objects and adding them to their history. (This process is described in more detail below.)

2. Check which rules *fire*, by comparing past-time antecedents of each rule against the current history to see which are satisfied.

3. *Jointly execute* the fired rules together with any commitments carried over from previous cycles. This is done by first collecting consequents of newly fired rules and old commitments, which become *commitments*. It may not be possible to satisfy *all* the commitments on the current cycle, in which case unsatisfied commitments are carried over to the next cycle. An object will then have to choose between a number of execution possibilities.

4. Goto (1).

Clearly, step (3) is the heart of the execution process. Making a bad choice at this step may mean that the object specification cannot subsequently be satisfied (see [1, 6]).

A natural question to ask is: how do objects do things? How do they send messages and perform actions? When a predicate in an object becomes *true*, it is compared against that object's interface (see above); if it is one of the object's *component* predicates, then that predicate is broadcast as a message to all other objects. On receipt of a message, each object attempts to match the predicate against the environment predicates in their interface. If there is a match then they add the predicate to their history, prefixed by a ' $\odot$ ' operator, indicating that the message has just been received.

The reader should note that although the use of only broadcast message-passing may seem restrictive, standard point-to-point message-passing can easily be simulated by adding an extra 'destination' argument to each message; the use of broadcast message-passing as the communication mechanism gives the language the ability to define more adaptable and flexible systems. We will not develop this argument further; the interested reader is referred to our earlier work on Concurrent MetateM [5, 7].

3 SPECIFYING A PROTOCOL

We will consider a Contract Net-like protocol and its implementation in Concurrent MetateM — we pre-suppose familiarity with the Contract Net [12]. First, we will describe the notions of *tasks* and *capabilities* that will be used throughout this description. We will represent an individual capability simply as a constant. For example, if an agent is able to move, speak and jump, the capabilities of the agent would be represented by the *capabilities* predicate within the agent's definition, i.e.,

$$capabilities(agent, [move, speak, jump])$$

A task will be represented simply as a function symbol *task* applied to certain arguments, i.e.,

$$task(Name, Description, Requirements, Originator)$$

Predicate	Meaning
announce(*Task*)	announces that a particular task is available for bids
bid(*Task, Bidder*)	a bid for a particular task
award(*Task, Awardee*)	awards the contract for a particular task
completed(*Task, By, Result*)	signals the completion of a particular task

Table 1: Message Predicates

where

- *Name* is the name of the task;
- *Description* is the general description of the task;
- *Requirements* is the list of capabilities required of an agent for it to be able to carry out the task; and
- *Originator* is the agent who announced the task.

We will now outline the CONCURRENT METATEM rules that can be used to describe the behaviour of a simple agent taking part in our contract net-like system. The behaviours of the agent will be split into categories relating to *task announcement*, *bidding*, the *award* of contracts, and the *completion* of contracts.

The basic message predicates used in the system are summarised in Table 1. Along with these predicates representing communication, a number of other internal predicates will be used. In particular, we will define the predicates *competent*, *busy*, *bidded*, and *most-preferable* as follows[2].

$$\begin{aligned} \mathit{capabilities}(A, \mathit{Cap}) \wedge (\mathit{Cap} \cap \mathit{Req} \neq \emptyset) &\Rightarrow \mathit{competent}(A, \mathit{task}(T, D, \mathit{Req}, O)) \\ (\neg \mathit{completed}(T, \mathsf{self}, R))\, \mathcal{S}\, \mathit{award}(T, \mathsf{self}) &\Rightarrow \mathit{busy}(\mathsf{self}) \\ (\neg \mathit{award}(T, A))\, \mathcal{S}\, \mathit{announce}(T) \wedge \mathit{bid}(T, X) &\Rightarrow \mathit{bidded}(T, X) \\ \mathit{most\text{-}preferable}(T, X) &\Leftrightarrow \neg \exists Y.\ \mathit{preferable}(Y, X) \wedge \mathit{bidded}(T, Y) \end{aligned}$$

Note that 'self' refers to the object id of the object in which the rules occur.

Task Announcement

Initially, a prospective manager agent just announces its first task, using the following rule.

$$\mathbf{start} \Rightarrow \mathit{announce}(\mathit{task}(\mathit{Name}, \mathit{Desc}, \mathit{Req}, \mathsf{self})) \qquad (A1)$$

[2]We assume that each agent awarding contracts has an internal selection procedure which is characterised by the predicate *preferable*.

If an agent has been contracted to carry out a task, yet is unable to complete it, then it must sub-contract part of the task. The rule used in this case utilises '*split*', a predicate that splits a task appropriately, given the agent's capabilities (i.e., a task is split into two tasks, the first of which the agent is able to complete, the second of which it must attempt to subcontract).

$$\left(\begin{array}{c} award(task(N, D, Req, O), \mathsf{self}) \\ \wedge\, capabilities(\mathsf{self}, Cap) \\ \wedge\, (Req - Cap \neq \emptyset) \end{array}\right) \Rightarrow \left(\begin{array}{c} split(task(N, D, Req, O), T1, T2) \wedge \\ announce(T2) \wedge \Diamond \exists R.\ result(T1, R) \end{array}\right) \tag{A2}$$

Bidding

The first basic rule involved in the bidding process is that an agent should only define a *possible* task as one that has been announced (and not yet awarded) and which the agent has the capabilities to undertake.

$$(\neg award(T, A))\, \mathcal{S}\, announce(T) \wedge competent(A, T) \quad \Leftrightarrow \quad possible(A, T) \tag{B1}$$

Given this rule, another basic property of bidding agents is that they should not bid for tasks that are not possible, i.e.,

$$\neg possible(\mathsf{self}, T) \quad \Rightarrow \quad \neg bid(T, \mathsf{self}) \tag{B2}$$

We can then add a variety of rules depending upon the behaviour required for the agent. For example, the following rules are needed only if an agent is restricted to bidding for one task at a time.

$$possible(\mathsf{self}, T) \Rightarrow \exists Y.\ bid(Y, \mathsf{self}) \tag{B3}$$

$$bid(X, \mathsf{self}) \wedge bid(Y, \mathsf{self}) \Rightarrow X = Y \tag{B4}$$

The following rule is needed if we restrict the agent's behaviour so it can't bid while it is actively undertaking a task.

$$busy(\mathsf{self}) \quad \Rightarrow \quad \neg bid(X, \mathsf{self}) \tag{B5}$$

Finally, if we want an agent to be able to bid for every task, at any time, we would replace rules *B3*, *B4*, and *B5* by

$$possible(\mathsf{self}, T) \quad \Rightarrow \quad bid(T, \mathsf{self}) \tag{B6}$$

Awarding Contracts

Given that a manager has announced a task then, after a certain time, it must decide which bidding agent to award the contract to. To achieve this, we simply use the rule

$$bidded(T, X) \wedge most\text{-}preferable(T, Y) \quad \Rightarrow \quad award(T, Y) \qquad (W1)$$

Thus the choice amongst those agents that have bid for the the contract is made by consulting the agent's internal list of preferences.

Completion

There are two rules relating to the completion of a task, one for tasks solely carried out within the agent, the other for tasks that were partially sub-contracted.

$$\left(\begin{array}{c} (\neg completed(T, \mathsf{self}, X) \wedge \neg split(T, T1, T2)) \,\mathcal{S}\, award(T, \mathsf{self}) \\ \wedge \circledcirc result(T, R) \end{array}\right) \Rightarrow completed(T, \mathsf{self}, R) \qquad (C1)$$

$$\left(\begin{array}{c} (\neg completed(T, \mathsf{self}, X)) \,\mathcal{S}\, award(T, \mathsf{self}) \wedge \blacklozenge split(T, T1, T2) \\ \wedge \blacklozenge result(T1, R1) \wedge \blacklozenge completed(T2, By, R2) \end{array}\right) \Rightarrow completed(T, \mathsf{self}, R1 \cup R2) \qquad (C2)$$

Thus, in the first case, once the agent has produced a result, the completion of the task is announced, while in the second case completion is only announce once the agent has completed its portion of the task and the sub-contractor reports completion of the remainder.

4 PROVING PROPERTIES OF THE SYSTEM

In this section, we give an outline of how various properties of the above system may be established through formal proof. Rather than giving the detailed proofs, we will present a precis of the proof process for a selection of obvious properties that the system should exhibit.

Throughout this section, we will use statements of the form

$$[Agent1] \vdash \varphi$$

to show that the formula φ can be proved for *Agent1*. We will now give a selection of properties and outline their proofs for our multi-agent system.

Property 1 *If at least one agent bids for a task, then the contract will eventually be awarded to one of the bidders.*

As this is a global property of the system, not restricted to a particular agent, then it can be represented logically as

$$bid(T, A) \;\Rightarrow\; \exists B.\, \Diamond award(T, B).$$

We can prove this statement as follows. Given a particular task, t, together with an agent, a, which makes a bid for the task, then we know that

$$[a] \vdash bid(t, a).$$

Now, from the axioms governing communication between agents, we know that, once broadcast, a message of this form will eventually reach all agents who wish to receive messages of this form. Thus, we know that

$$[m] \vdash \Diamond bid(t, a)$$

where m is the manager agent for this particular task. Similarly, we can derive

$$[m] \vdash \Diamond bidded(t, a).$$

Finally, by the definition of contract allocation, we know that for some bidding agent, p (the 'most preferable'), then the manager will eventually award the contract to p, i.e.,

$$[m] \vdash \Diamond award(t, p).$$

As this information is broadcast, we derive the global statement that

$$\Diamond award(t, p)$$

thus satisfying the desired property.

Property 2 *Agents do not bid for tasks that they cannot contribute to.*

In logical terms, this is simply

$$(announce(T) \wedge \neg competent(A, T)) \;\Rightarrow\; \neg bid(T, A)$$

where T is any task and A is any agent.

If we know that, for some task t and agent a, where the task t has been announced, yet the agent a is not competent to perform the task then we know by rule (B1) that

$$\neg possible(a, t).$$

Now, by rule (B2), then we can derive the fact that agent a will not bid for the task, i.e.,

$$\neg bid(t, a).$$

Property 3 *Agents do not bid unless there is a task announcement.*

In logical terms, this statement can be represented by

$$bid(T, A) \;\Rightarrow\; \blacklozenge announce(T)$$

where T is any task and A is any agent.

In order to prove this statement, we simply show that for there to have been a bid, the agent must have considered the task possible, i.e.,

$$possible(t, a)$$

and, for this to occur, then by (B1)

$$(\neg award(t, B))\, \mathcal{S}\, announce(t)$$

which in turn implies

$$\blacklozenge announce(t).$$

Property 4 *Managers award the contract for a particular task to at most one agent.*

This can be simply represented by

$$award(T, A) \wedge award(T, B) \;\Rightarrow\; A = B.$$

The proof of this follows simply from (W1) which states that the 'most preferable' bidder is chosen. The definition of 'most preferable' in turn utilises the linear ordering provided by the 'preferable' predicate.

Further, more complex properties may also be proved. For example, it is possible to show that if a task is announced and there is sufficient coverage of skills within the group, then the task will eventually be completed.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we have shown how CONCURRENT METATEM, a programming language for DAI based on the notion of executable temporal logic, can be used to elegantly and succinctly express the properties of protocols for cooperative action. We have also shown how the properties of such protocols may be established through formal proof.

This combination of a high-level formal specification, together with both proof and execution methods provides a powerful framework for the development of DAI systems. By a combination of rapid prototyping (through executable specifications) and formal verification, we can develop complex DAI systems that are more likely to be both correct and dependable. Further, the use of a very expressive logic, such as temporal logic, means that we are able to describe a range of complex behaviours for agents themselves.

This framework is particularly suitable for application in the area of cooperative protocols as not only does this require a formal structured protocol to operate effectively, but also the verification of properties of agents in such a system is feasible. Further work in this, and in other areas concerning cooperative action, is planned in the future.

Extending the Logic

There are two particular areas in which the logical basis and the proof rules for our systems could be improved. The first is the extension of the proof rules to incorporate the grouping structures being developed for CONCURRENT METATEM. This would allow us to prove properties of multi-agent systems exhibiting some of the following attributes.

- Cooperative activity between agents, e.g., pooling resources.
- Competitive strategies, e.g., don't bid for contracts from, or award to, 'unfriendly' agents.
- Grouping structures, e.g., forming a team that has the capabilities to cope with most tasks.
- Fault-tolerance, e.g., through duplicate contractor agents, 'shadow' agents, etc.

The other extension required in order to realistically prove properties of large systems that have been implemented using CONCURRENT METATEM is to incorporate *operational constraints* into the proof rules. In the proofs give in §4, we have assumed that the execution mechanism perfectly implements the semantics of temporal formulae. However, not only is the more expressive first-order temporal logic used undecidable, but it is also incomplete. Further, 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 $\Diamond\varphi$ in temporal logic means that the formula φ *will* be satisfied at some point in the future, while when this is executed in CONCURRENT METATEM we simply *attempt* to satisfy φ as soon as we can.

REFERENCES

[1] 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).

[2] A. H. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, 1988.

[3] B. Burmeister, A. Haddadi, and K. Sundermeyer. Generic configurable cooperation protocols for multi-agent systems. In *Proceedings of the Fifth European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW '93)*, 1993.

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

[5] M. Fisher. Concurrent METATEM — A Language for Modeling Reactive Systems. In *Parallel Architectures and Languages, Europe (PARLE)*, Munich, Germany, June 1993. Springer-Verlag.

[6] 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).

[7] M. Fisher and M. Wooldridge. Executable Temporal Logic for Distributed A.I. In *Proceedings of the Twelfth International Workshop on Distributed Artificial Intelligence*, Hidden Valley, Pennsylvania, May 1993.

[8] M. Fisher and M. Wooldridge. Specifying and Verifying Distributed Intelligent Systems. In *6th Portuguese Conference on Artificial Intelligence (EPIA '93)*. Springer-Verlag, October 1993.

[9] D. Gabbay. Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of Colloquium on Temporal Logic in Specification*, pages 402–450, Altrincham, U.K., 1987. (Published in *Lecture Notes in Computer Science*, volume 398, Springer Verlag).

[10] 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.

[11] N. R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. *Knowledge Engineering Review*, 8(3):223–250, 1993.

[12] R. G. Smith. *A Framework for Distributed Problem Solving*. UMI Research Press, 1980.