Executing Logical Agent Specifications

Michael Fisher and Anthony Hepple

Department of Computer Science, University of Liverpool, U.K.

EMAIL: MFisher@liverpool.ac.uk; A.J.Hepple@liverpool.ac.uk

To appear in

Multi-Agent Programming: Languages, Platforms and Applications, Vol. 2, edited by *Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni* and published by Springer.

Abstract

Many agent-oriented programming languages are based on the Prolog-like logical goal reduction approach where rules are used to reduce, in a depth-first way, a selected goal. The ability of agents to change between goals means that such languages often overlay the basic computational engine with a mechanism for dynamically changing which goal is selected.

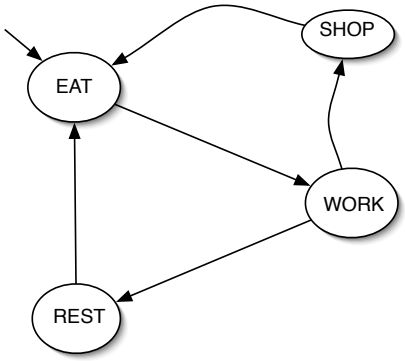
Our approach is different. The basic computational approach we use is that of model building for logical formulae, but the underlying formulae are *temporal*. This allows us to capture the dynamic nature of the agent explicitly. In addition, the temporal basis provides us with ways of having multiple active 'goals' and being able to achieve several at once. As in most agent-oriented languages *deliberation* is used to choose between goals when not all can be satisfied at once.

This basic execution of temporal formulae provides us with the foundation for agent programming. In order to deal with multi-agent systems in an equally straightforward way we also incorporate a very simple, but flexible, model of organisational structuring.

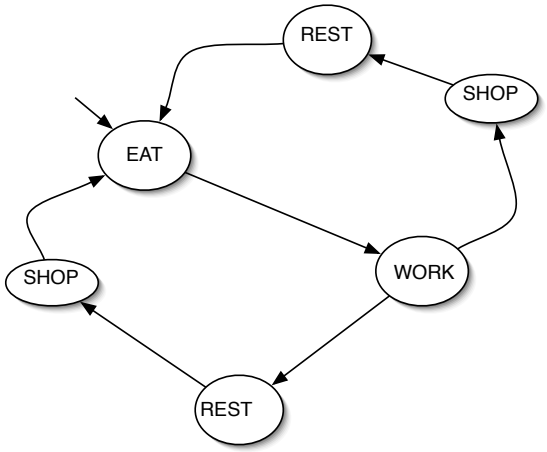
These two aspects provide the core of the language implemented. There are, however, many extensions that have been proposed, some of which have been implemented, and all of which are mentioned in this article. These include varieties of agent belief, resource-bounded reasoning, the language's use as a coordination language, and the use of contextual constraints.

1 Motivation

What does an agent do? It has a number of choices of how to proceed, most simply represented by a finite-state machine, for example:



Here the agent begins in the ‘EAT’ state and can take any transition to move to another state. But the agent also has *choices*. In the example above, when the agent is in the ‘WORK’ state it can then either move to the ‘REST’ state or to the ‘SHOP’ state. How shall the agent decide what to do at these choice states? The agent could act randomly, effectively tossing a coin to see which branch to take. Or we could fix the order in which choices are made, effectively modifying the state machine to capture a fixed set of choices, for example:



Here, we are restricted about which choices to take, for example being disallowed from following the sequence

$$\text{EAT} \longrightarrow \text{WORK} \longrightarrow \text{REST} \longrightarrow \text{EAT} \longrightarrow \text{WORK} \longrightarrow \text{REST} \longrightarrow \text{EAT} \longrightarrow \ldots$$

which was allowed in the original state machine. Not only is this prescription of choices against the spirit of agent-oriented computing, where choices are made dynamically based

on the prevailing conditions, but this is also very inflexible. State machines become large and complex if we try to capture all possible legal sequences of choices within them.

Thus, agents, particularly *rational* agents, have various *motivational attitudes* such as goals, intentions, or desires, which help them make choices at any point in their computation. These motivations not only help direct an agent's choices, but are dynamic themselves with new motivations being able to be added at any time. Thus, if we consider the first state machine and provide some additional motivations (let us call them *goals* here) then we can avoid 'bad' sequences just by using these goals to direct our choice. If the agent's main goals are to REST and SHOP then, at the WORK state the most important goal will direct the choice of next state. For example, let us say we move to the REST state. When we again come back to the WORK state, we still have a goal to SHOP, and so we may well choose that branch. And so on. Each time we visit the WORK state, the current goals modify which choices we take.

This is not only a simple view, but it is also a very flexible one. These ideas are at the heart of, for example, BDI languages where the desires and intentions act as motivations for choosing between options [29, 30].

We choose to use a *formal logic* to capture exactly this fundamental choice. This simple view allows us to describe both the potential choices and the agent motivations. If we recall the first state machine above, then we can actually capture what is happening via a set of logical formulae, as follows.

$$\begin{aligned} \mathsf{EAT} &\Rightarrow \bigcirc\mathsf{WORK} \\ \mathsf{WORK} &\Rightarrow \bigcirc(\mathsf{REST} \vee \mathsf{SHOP}) \\ \mathsf{REST} &\Rightarrow \bigcirc\mathsf{EAT} \\ \mathsf{SHOP} &\Rightarrow \bigcirc\mathsf{EAT} \end{aligned}$$

These are actually *temporal logic* formulae, with the temporal operator '$\bigcirc$' meaning "at the next moment in time". As you can see from the formulae above, we can treat these as being 'rules' telling us what to do next. However, we can go beyond just simple modelling of finite-state machines. (Note that, to precisely model the state machine we also need to add rules ensuring that exactly one of EAT, WORK, REST, or SHOP is true at any moment in time.) As we have the full power of temporal logic (propositional temporal logic, in this case) we can define additional propositions and use them to define other aspects of the computation. For example, rather than

$$\mathsf{WORK} \Rightarrow \bigcirc(\mathsf{REST} \vee \mathsf{SHOP})$$

we might instead have

$$\begin{aligned} (\mathsf{WORK} \wedge \mathit{tired}) &\Rightarrow \bigcirc\mathsf{REST} \\ (\mathsf{WORK} \wedge \neg\mathit{tired} \wedge \mathit{rich}) &\Rightarrow \bigcirc\mathsf{SHOP} \\ (\mathsf{WORK} \wedge \neg\mathit{tired} \wedge \neg\mathit{rich}) &\Rightarrow \bigcirc(\mathsf{REST} \vee \mathsf{SHOP}) \end{aligned}$$

Importantly, all of *tired*, SHOP, etc., are propositions within the language.

As described above, as well as the basic structure of choices, a core component of agent computation is some representation of motivations such as goals. Fortunately, temporal logic provides an operator that can be used to give a simple form of motivation, namely the "sometime in the future" operator '$\Diamond$'. This operator is used to ensure that some property becomes true at *some* point in the future; possibly now, possibly in the next moment, possibly in 5 moments time, possibly in 400 moments time, but it will definitely happen. Consider again our original example, but now with such *eventualities* added:

$$\begin{aligned} \mathsf{WORK} &\Rightarrow \Diamond\mathsf{REST} \\ \mathsf{WORK} &\Rightarrow \Diamond\mathsf{SHOP} \\ \mathsf{WORK} &\Rightarrow \bigcirc(\mathsf{REST} \vee \mathsf{SHOP}) \end{aligned}$$

These formulae describe the fact that there is still a choice between REST or SHOP, once the WORK state is reached, but also ensures that eventually (since we keep on visiting the WORK state) REST will become true and eventually SHOP will become true. Importantly, we can never take the same choice forever.

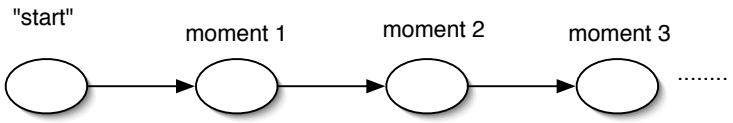
Thus, the above is the basic motivation for our agent language. There are clearly many other aspects involved in the programming of rational agents (and, indeed, we will discuss these below) but the use of temporal logic formulae of the above form to describe computation, and the execution of such formulae in order to implement agents, is the underlying metaphor.

2 Language

The basic 'METATEM' approach was developed many years ago as part of research into formal methods for developing software systems [2]. Here, the idea is to *execute* a formal specification by building a concrete model for the specification; since the specifications are given in temporal logic, model-building for temporal formulae was developed. Throughout, the *imperative future* view was followed [24, 1], essentially comprising forward chaining from initial conditions and building the future state by state.

2.1 Specifications and Syntactical Aspects

Thus, the core language we use is essentially that of *temporal logic* [11, 17]. This is a formal logic based on the idea that propositions/predicates can be true/false *depending* on what moment in time they are evaluated. Thus, the temporal structure linking moments in time is important. While there are many different possibilities for such temporal structures [17], we adopt one of the simplest, namely a linear sequence of discrete moments in time with a distinguished 'start' moment (finite past).



This sequence of moments can go on for ever. Importantly, the basic propositions/predicates within the language can have different truth values at different moments. Thus, a propo-

sition '*hungry*' might be false in the 'start' moment, true in moment 1, true in moment 2, false in moment 3, etc.

Obviously we need a language to be able to describe such situations. Again, there are many varieties of temporal language, even for the simple model above. So, we choose a basic, but flexible, variety involving the logical operators:

'$\bigcirc$' . . . . . "in the next moment in time";

'$\Box$' . . . . . "at every future moment";

'$\Diamond$' . . . . . "at some future moment".

These operators give us useful expressive power and, even with such a simple temporal logic as a basis, we are able to describe both an agent's individual dynamic behaviour, and also (see later) how an agent's beliefs or goals evolve. As is demonstrated by the following simple example of conversational behaviour, which captures a subtle preference for listening over speaking by allowing models with repeated listening states but preventing uninterrupted speaking!

$\Box(\mathsf{SPEAK} \vee \mathsf{LISTEN})$
$\Box\neg(\mathsf{SPEAK} \wedge \mathsf{LISTEN})$
$\mathsf{LISTEN} \Rightarrow \Diamond\mathsf{SPEAK}$
$\mathsf{SPEAK} \Rightarrow \bigcirc\mathsf{LISTEN}$

2.1.1 Basic Execution

Given a temporal description, using the above language, we adopt the following basic execution approach:

- transform the temporal specification into a *normal form* [13];
- from the initial constraints, *forward chain* through the set of temporal rules constraining the *next* state of the agent; and
- constrain the execution by attempting to satisfy eventualities (aka goals), such as $\Diamond g$ (i.e. g eventually becomes true). (This, in turn, involves some strategy for choosing between such eventualities, where necessary.)

The basic normal form, called Separated Normal Form (SNF) [13], essentially categorises formulae into 3 varieties: *initial rules*, of the form **start** $\Rightarrow \varphi$, which indicate properties of the initial state; *step rules*, of the form $\psi \Rightarrow \bigcirc\varphi$, which indicate properties of the *next* state; and *sometime rules*, of the form $\psi \Rightarrow \Diamond\phi$, which indicate properties of the future. In each case φ is a disjunction of literals, ψ is a conjunction of literals and ϕ is a positive literal. In summary, the transformation to this normal form ensures that all negations apply only to literals, that all temporal operators other than $\bigcirc$ and $\Diamond$ are removed, and that all occurrences of the $\Diamond$ operator apply only to literals.

Since we allow first-order predicates, implied quantification of variables, and non-temporal rules, the specific normal form used is more complex. Below, we provide the normal form of each rule type implemented, a corresponding example in first-order temporal logic and, with the implied quantification, an equivalent example in the implemented METATEM syntax.

START RULE:

General $\quad start \Rightarrow \exists \bar{x}. \bigvee_{i=1}^{a} p_i(\bar{x})$

Example $\quad start \Rightarrow \exists x.[p(x) \vee q(x)]$

Code
```
start => p(X) | q(X);
```

STEP RULE:

General $\quad \forall \bar{x}. \left[\left[\bigwedge_{i=1}^{a} p_i(\bar{x}) \wedge \exists \bar{y}. \bigwedge_{j=0}^{b} q_j(\bar{y},\bar{x}) \right] \Rightarrow \bigcirc \exists \bar{z}. \bigvee_{k=1}^{c} r_k(\bar{z},\bar{x}) \right]$

Example $\quad \forall x. \left[\left[p(x) \wedge \exists y. q(y,x) \right] \Rightarrow \bigcirc \exists z. [r(z,x) \vee s(z,x)] \right]$

Code
```
p(X) & q(Y,X) => NEXT r(Z,X)|s(Z,X);
```

SOMETIME RULE:

General $\quad \forall \bar{x}. \left[\left[\bigwedge_{i=1}^{a} p_i(\bar{x}) \wedge \exists \bar{y}. \bigwedge_{j=0}^{b} q_j(\bar{y},\bar{x}) \right] \Rightarrow \Diamond \exists \bar{z}. r(\bar{z},\bar{x}) \right]$

Example $\quad \forall x. \left[\left[p(x) \wedge \exists y. q(y,x) \right] \Rightarrow \Diamond \exists z. r(z,x) \right]$

Code
```
p(X) & q(Y,X) => SOMETIME r(Z,X);
```

NON-TEMPORAL (PRESENT-TIME) RULE:

General $\quad \forall \bar{x}. \left[\left[\bigwedge_{i=1}^{a} p_i(\bar{x}) \wedge \exists \bar{y}. \bigwedge_{j=0}^{b} q_j(\bar{y},\bar{x}) \right] \Rightarrow \bigvee_{k=1}^{c} r_k(\bar{x}) \right]$

Example $\quad \forall x. \left[\left[p(x) \wedge \exists y. q(y) \right] \Rightarrow \left[r(x) \vee s(x) \right] \right]$

Code
```
p(X) & q(y) => r(X) | s(X);
```

We are able to omit explicit quantification symbols from the program code by making the following interpretations.

- Any variable appearing positively in the antecedents is universally quantified.
- Any variables that remain after substitution of matching, and removal of non-matching, universal variables are existentially quantified.
- Of the existentially quantified variables, those that appear only negatively in the antecedents are ignored.
- Existential variables in the consequent of an otherwise grounded rule, which cannot be matched are grounded by Skolemisation.

Existentially quantified variables are not allowed in the consequent of a present-time rule, preventing circumstances in which present-time rules fire repeatedly (possibly infinitely) as a result of new terms generated by repeated grounding by Skolemisation.

Next, we consider a number of examples, exhibiting the basic execution mechanism.

Examples of Basic Execution

We now consider several basic examples, describing how execution of such scenarios occurs.

Example 1

Consider a machine capable of converting a raw material into useful widgets, that has a hopper for its raw material feed which, when empty, prevents the machine from producing widgets. A simple specification for such a machine, presented in the normal form described above, is as follows (each rule is followed by an informal description of its meaning):

`start => hopper_empty;`

The hopper is initially empty.

`true => power;`

The machine has uninterrupted power.

`hopper_empty => NEXT fill_hopper;`

If the hopper is empty, then it must be refilled in the next moment in time.

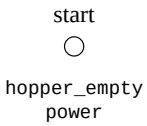
`fill_hopper => NEXT ( material | hopper_empty ) ;`

Filling the hopper is *not* always successful.

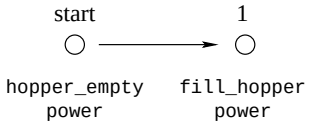
`( material & power ) => NEXT widget;`

If the machine has power and raw material then, in the next moment in time a widget will be produced.

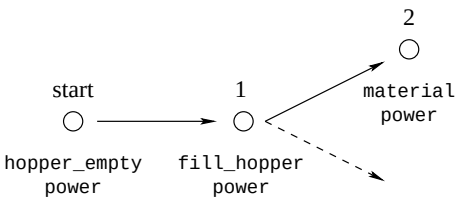
Execution begins with the construction of an initial state which is constrained by the start rules and any present-time rules. Thus, in the *start* state our machine has an empty hopper and power:



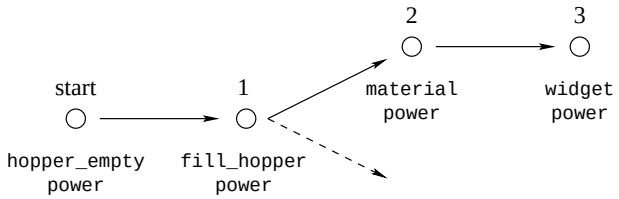
The interpretation of each state is used to derive constraints on the next state. Applying the above rules to this initial state produces the constraint `fill_hopper`, which must be true in any successor state. The METATEM execution algorithm now attempts to build a state that satisfies this constraint and is logically consistent with the agent's present-time rules. In this example we have only one present-time rule, which does not contradict our constraints but does introduce another constraint, hence state 1 is built:



State 1 provides the METATEM agent with its first choice point. Evaluation of the agent's rules constrains the next state to satisfy the disjunction, `((material ∧ power) ∨ (hopper_empty ∧ powe`. Without any preferences or goals to guide its decision, the METATEM agent is able to choose either alternative and makes a non-deterministic choice between disjuncts. For this example we will assume that `material` is made true in state 2:



In this state, our machine has both the power and material necessary to produce a widget in the next state:



Note. Without explicit rules, be they temporal or non-temporal, the machine no longer believes it has its raw material. Hence, evaluation of the agent's temporal rules with the interpretation of state 3 produces no constraints and the agent will produce no further states.

Example 2

This example illustrates the backtracking nature of the METATEM algorithm when it encounters a state that has no logically consistent future. Staying with our widget machine, we modify its non-temporal rule and provide an additional rule:

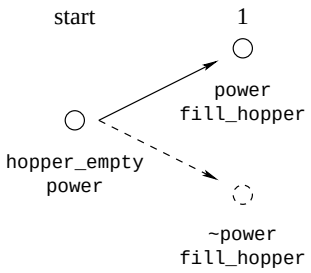
`true => power | ~power;`

Power can now be switched 'on' or 'off'.

`( fill_hopper & power ) => NEXT false;`

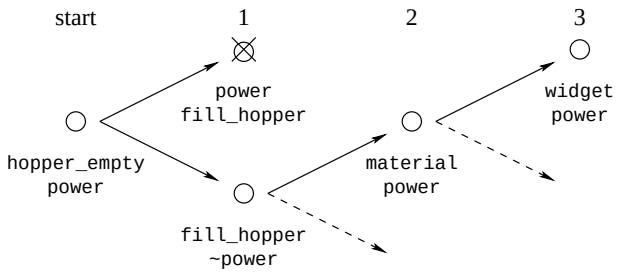
Filling the hopper with the power switched on causes irrecoverable problems in the next state!

Execution now begins in one of two states, `(hopper_empty ∧ power)` or `(hopper_empty ∧¬power)` due to the conjunction introduced by the modified present-time rule. Let us assume that the former is chosen, though it is inconsequential to our example. Again our agent has a choice when constructing the next state, it can fill the hopper with the power on or with the power off. Each of these choices has a consistent present but only one has a consistent future! Let us assume that the 'wrong' choice is made and the 'correct' choice is retained for future exploration;



Now, evaluation of state 1's interpretation constrains all future states to include `false` — this state has no future. It is at this point, when no consistent choices remain, that METATEM backtracks to a previous state in order to explore any remaining choices[1]. Execution then completes in much the same way as the previous example:

[1]In this example, the agent's ability to fill its hopper is considered to be internal and *reversible*. However, the current METATEM implementation does distinguish between internal and external (those that cannot be reversed and hence cannot be backtracked over) abilities. The sending of messages is an important example of an external ability.



At this point it should be emphasised that the above executions are, in each case, only one of many possible models that satisfy the given temporal specification. Indeed, many models exist that produce no widgets at all. To ensure the productivity of our widget machine we must introduce a goal in the form of an eventuality. For the next example we return to our conversational agent to demonstrate the use of temporal eventualities.

Example 3

For this example, we re-write the specification presented at the end of Section 2.1 into the executable normal form, removing all necessity operators ($\Box$), and conjunctions from the consequents of all rules. The result of which is:

```
true => NEXT ( speak | listen );
```

Always speaking or listening...

```
speak => ~listen;
listen => ~speak
```

...but never at the same time.

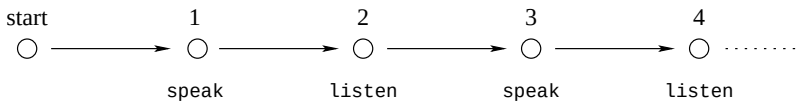
```
listen => SOMETIME speak;
```

Eventually speak after listening.

```
speak => NEXT listen;
```

Always pause to listen, after speaking.

The model resulting from execution of this specification is one which alternates between listening and speaking in successive states;



Although intuitively we may expect to see multiple listening states between each speaking state, the METATEM algorithm endeavours to satisfy outstanding eventualities *at the earliest opportunity*. That is, providing it is logically consistent to do so, an eventuality (such as "$\Diamond$*speak*") will be made true without being explicitly stated in the consequents of a *next* rule. There are no conflicting commitments and therefore there is no need to delay its achievement.

2.1.2 Strategies and Deliberation

Where there are multiple outstanding eventualities, and where only a subset of these can be satisfied at the same time, then some strategy for deciding which eventualities to satisfy now, and which to hold over until future states, is required. As we have seen in the previous

section, we are not able to require both `speak` and `listen` to be true at the same point in time. Thus, if we require both "`sometime listen`" and "`sometime speak`" to be made true many times, then we must decide when to make `speak` true, when to make `listen` true, and when to make neither true.

The basic strategy for deciding between conflicting eventualities is provided directly by the original METATEM execution algorithm. This is to choose to satisfy the eventuality that has been outstanding (i.e. needing to be satisfied, but as yet unsatisfied) the longest. This has the benefit that it ensures that no eventuality remains outstanding forever, unless it is the case that the specification is unsatisfiable [1].

There are, however, a number of other mechanisms for handling such strategies that have been developed. The most general is that described in [14]. To explain this, let us view the outstanding eventualities at any moment in time as a list. The eventualities will then be attempted in list-order. Thus, in the basic METATEM case we would order the list based on the age of the eventuality. When an eventuality is satisfied, it is removed from the list; when a new eventuality is generated, we add this to the end of the list.

With this list view, our strategy for deciding which eventualities to satisfy next is just based on the order of eventualities within a list. Thus, if the agent can *re-order* this list between states then it can have a quite sophisticated strategy for *deliberation*, i.e. for dynamically choosing what to tackle next. This approach is discussed further in [14, 18] but, unless we put some constraints on the re-ordering we might apply, then there is a strong danger that the completeness of the execution mechanism will be lost [18].

In the current implementation, rather than using this quite strong, but dangerous, approach we adopt simpler, and more easily analysable, mechanisms for controlling (or at least influencing) the choice of eventuality to satisfy. These mechanisms are characterised by the predicates/directives `atLeast`, `atMost` and `prefer`.

The `atLeast` predicate places a minimum constraint on the number of instances of positive predicates, whilst `atMost` places a maximum constraint on the number of instances of positive predicates in a given temporal state, in the style of the capacity constraints described by [10]. Besides providing the developer with the ability to influence an agent's reasoning, when applied judiciously `atMost` and `atLeast` can simplify the fragment of the logic considered and hence can increase the execution performance of a METATEM agent.

As an example of the use of predicate constraints we provide some code snippets from an example included with the METATEM download, which specifies the behaviour of a lift. The lift responds to calls from floors above and below it and, when more than one call remains outstanding, must decide which call to serve first, changing direction if necessary. Each discrete moment in time of our temporal model denotes the lift's arrival at a floor and the transition between temporal states is analogous to the lift's transition between floors. The following rules specify that the lift starts at the ground floor and must satisfy all calls before it can achieve the `waiting` state:

```
start => atFloor(0);
true => SOMETIME waiting;
call(X) => ~waiting;
```

Clearly, it is desirable that the lift visits a floor in each state of our model. This behaviour could be specified by the rule

```
true => NEXT atFloor(X);
```

which states that there must exist an `X` such that `atFloor(X)` is satisfied in each moment in time. However, our lift must visit *one and only one* of a limited number of *valid* floors.

The above rule is logically too general as it allows multiple `X`'s in any moment in time and implies an infinite domain of `X` [2]. Therefore our lift specification does not use the rule given immediately above, but instead employs predicate constraints. These ensures that the lift visits one and only one floor at each moment, without introducing an existential variable. The following declarations in an agent description file achieve this.

```
at_most 1 atFloor true;
at_least 1 atFloor true;
```

The construction of each temporal state during the execution of a METATEM specification generates a logical interpretation that is used to evaluate the antecedents of each temporal rule. The consequents of all the rules that fire are conjoined (and transformed into disjunctive normal form) to represent the agent's choices for the next temporal state, each conjunction being a distinct choice, one of which is chosen and becomes the interpretation of the next temporal state, from which the next set of choices are derived. This process is repeated, and conjunctions that are not chosen are retained as alternative choices to be taken in the event of backtracking. As mentioned earlier, a number of fundamental properties of the formulae in each conjunction affect the choice made. For example, an agent will always satisfy a commitment if it is consistent to do so, and will avoid introducing commitments (temporal 'sometime' formula) if able to, by making a choice containing only literal predicates. These preferences are built-in to METATEM, however the `prefer` construct allows the developer to modify the outcome of METATEM's choice procedure by re-ordering the list of choices according to a declared pair of predicates (e.g. `prefer(win,lose)`) after the fundamental ordering has been applied. We refer to the `prefer` construct as a deliberation meta-predicate and the architecture of the current METATEM allows the implementation of further deliberation meta-predicates as 'plug-ins'.

Each of these constructs can be declared as applicable in all circumstances or as context dependent, that is, only applicable when a given formula is true. Typically this formula might be an `inContext/1` or `inContent/1` predicate when one is expressing an agent's preferences or constraints when acting under the influence of another agent (the concept and purpose of Context/Content is explained in Section 2.1.3). Furthermore, each preference is assigned an integer weighting, within the (arbitrary) range of 1–99, which allows a fine-grained ordering of preferences[3].

For example, the following snippets are two alternative applications of the `prefer` construct to the lift example described above, to encourage the lift to continue moving in the same direction when appropriate;

```
prefer downTo to upTo when moving(down) weight 50;
prefer upTo to downTo when moving(up) weight 50;

prefer("downTo","upTo","moving(down)",50)
prefer("upTo","downTo","moving(up)",50)
```

The first two directives above are examples of those that appear in the preamble of an agent definition file[4], these preferences apply from time, $t = 0$. The latter two directives are examples of meta-predicates that, when appearing in the consequents of a temporal

[2] Indeed, the current implementation considers existential variables on the right-hand side of future rules on an open-world principle, implementing a form of Skolemisation by, when necessary, creating new terms. In this example our lift could disappear to an imaginary floor!

[3] We reserve the weighting values 0 and 100 for built-in preferences.

[4] A more detailed description of the agent file is given in Section 3

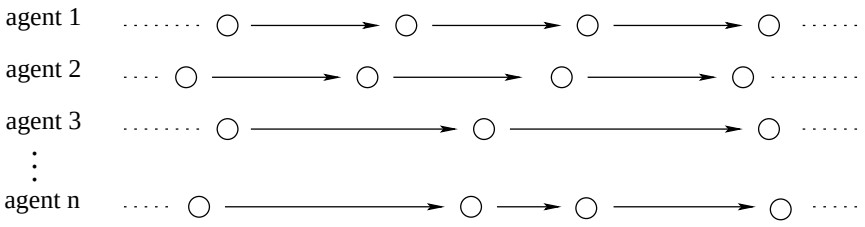


Figure 1: Typical asynchronous agent execution.

NEXT rule, will provide the agent with the declared preference from the temporal state following the state in which the rule was fired. The former type is simply a syntactic convenience for a rule of the type

```
start => prefer("downTo","upTo","moving(down)",50)
```

Once a preference is applied it is upheld for all future states and there is no mechanism for explicitly deleting it, instead preferences can be *overriden* by an otherwise identical preference which declares a higher priority value or *counteracted* by an opposing preference. However, the use of context dependent preferences is encouraged as leaving a context provides the effect of deleting a preference but with the benefit that the preference will be reinstated upon entering the relevant context. We believe this is a natural interpretation of preferences.

2.1.3 Multiple Agents

METATEM supports the asynchronous, concurrent execution of multiple agents which are able to send one another messages that are guaranteed to arrive at some future moment in time. Each agent has its own concept of time and the duration of each time step for an individual agent is neither fixed nor constant throughout execution. Conceptually then, the transition of multiple agents between successive temporal states is as depicted in Fig. 1.

Note. The form of asynchronous execution seen in Fig. 1 is a little problematic for propositional temporal logic to represent straight-forwardly. However, as described in [12] a temporal logic based on the *Real Numbers* rather than the Natural Numbers, provides an appropriate semantic basis. Importantly, the propositional fragment of such a *Temporal Logic of the Reals* required still remains decidable [26].

An agent sends a message to another agent by making the action predicate $send(Recipient, Message)$ true in one of its own states. This guarantees that at some future moment the predicate $receive(From, Message)$ will be true in at least one state of the recipient agent (where $Recipient$, $From$ and $Message$ are all terms and are substituted by the recipient agent's name, the sending agent's name and the message content, respectively). The *send* predicate is an example of a special 'action' predicate which, when made true, prevents subsequent backtracking over the state in which it holds. For this reason, the use of a deliberate–act style of programming is encouraged in which an agent explores multiple execution paths, performing only retractable internal actions, backtracking when necessary, before taking a non-retractable action.

Although METATEM agents exist as individual threads within a single Java™ virtual machine there are no other predefined agent containers or agent spaces that maintain a centralised structuring of multiple agents. Instead METATEM follows an agent-centred approach to multi-agent development with the only implemented interactions between autonomous agents being message passing. Support for the abstract structuring of agent

societies is provided by internal (to each agent) constructs and is discussed in detail in the next section.

Agent structuring: Content and Context.

At the implementation level, and as the default abstraction, agents occupy a single flat structure (potentially physically distributed). However, each agent maintains two sets of agent references named 'Content' and 'Context' which provide the flexibility to represent a wide range of inter-agent relationships and multi-agent structuring [22, 8], as can be represented as in Fig. 2.

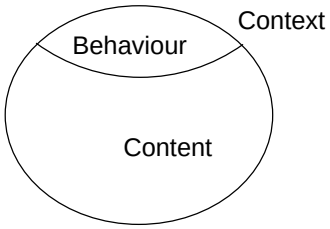


Figure 2: An abstract representation of a single METATEM agent.

The actual meaning of a relationship between an agent and the members of its Content and its counterpart Context is entirely flexible and dependent on the application being tackled. That is, during the initial analysis stage of an agent-oriented software engineering process when the most appropriate abstraction(s) are determined, it may be declared that the Content set of an agent who acts as a team leader are the agents that comprise its team. Alternatively, an agent may be declared to represent an aspect of the environment, and those agents who have the 'environment' agent in their Content set have that aspect of the environment within their sphere of influence. An agent specification then, contains its temporal specification (its behaviour) along with its Content and Context sets.

Structurally, an agent's Content set are those agents an agent is said to "contain" and its Context are those it is "contained by". But abstractly, and capturing the social nature of multi-agent systems, an agent's Context should be viewed as the set of agents that influence its own behaviour and its Content as those agents whose behaviour it has some influence over. These agents sets are used to store the multi-agent structure in a truly distributed manner that not only allows a wide range of structures and abstractions to be implemented

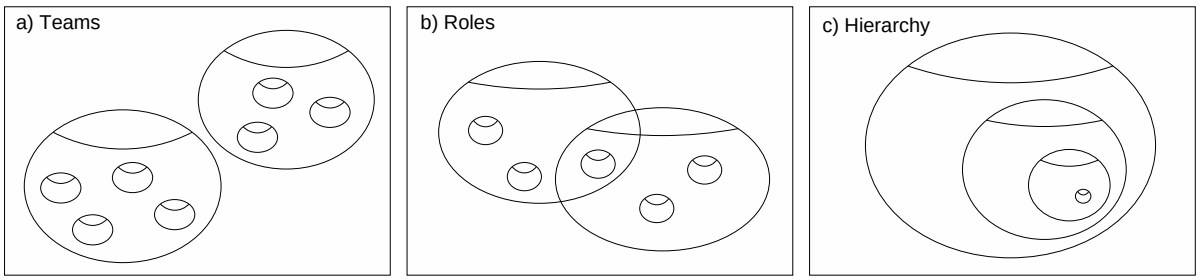


Figure 3: A selection of METATEM agent structures, with possible interpretations; a) two teams of agents, b) five agents fulfilling two roles (one agent fulfills both roles), and c) a nested hierarchy of agents.

Predicates	Actions
`inContent(Agent)`	`addToContent(Agent)`
`inContext(Agent)`	`enterContext(Agent)`
`enteredContent(Agent)`	`removeFromContent(Agent)`
`enteredContext(Agent)`	`leaveContext(Agent)`
`leftContent(Agent)`	
`leftContext(Agent)`	

Table 1: Some of the system predicates and actions that can be used by METATEM agents to reason about, and modify, their Content and Context sets.

(as illustrated in Fig. 3) but also allows dynamic structural changes at run-time without central organisation.

Crucially, an agent is able to reason about, and modify, its relationship with other agents at any given moment in time using the system predicates and action predicates listed in Table 1. In the case of the `inContent/1` and `inContext/1` predicates, they hold true for all agents that are members of Content (and Context respectively) and for all moments in time that they remain members. The actions listed can only be made true by the agent whose Content (and Context respectively) is modified; in each case making the action true entails that in the next moment in time the modification has taken effect, i.e. for Content, the following rules apply.

```
addToContent(a) => NEXT inContent(a);

removeContent(a) => NEXT ~inContent(a);.
```

Multicast message passing

One of the most advantageous practical benefits of the Context/Content grouping described above, is their use in message passing. For, as well as sending messages to individual named agents, a METATEM agent can address messages to sets of agents using a number of supported set expressions. The terms `Content` and `Context` are in fact literal set expressions which can be used to build larger expressions using the operators `UNION`, `INTERSECTION` and `WITHOUT`, for instance an agent can send a message to the set `Content UNION Context`. Additionally, an agent can send a message to all agents who share the same Context, *without maintaining an explicit reference to those agents*. The three fundamental multicasts are depicted in Fig. 4.

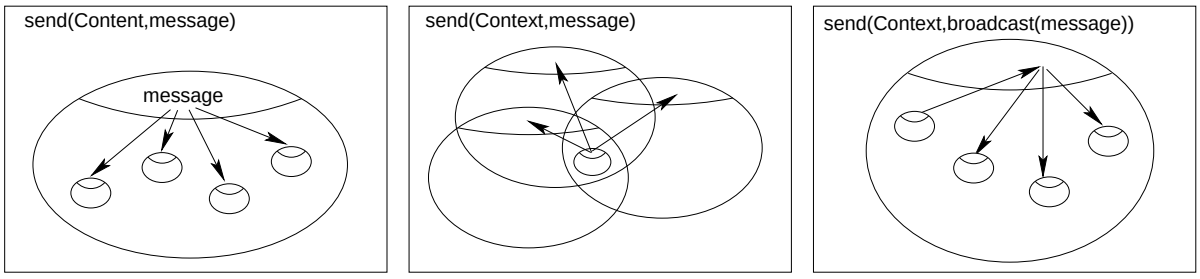


Figure 4: The three fundamental forms of multicast messaging across METATEM's multi-agent structure.

It is expected that the membership of an agent's Content and Context sets changes at runtime, as is appropriate for the agent's activity in the modelled system. The inclusion of a third set of agents called the `Known` set contains all agents an agent has ever encountered. It retains a references to those agents who once belonged to `Content` or `Context` sets after they have left [5].

2.2 Semantics and Verification

Programs in METATEM essentially comprise formulae in SNF normal form [13], together with annotations concerning communication and organisation. It is important to note that *any* temporal logic formula can be translated to an equivalent one in SNF (be it propositional, or first-order) in polynomial time. Once we have a set of SNF formulae representing the behaviour of an agent, we can begin execution. Without external interaction, such execution essentially corresponds to model construction for the formulae. Thus, an important theorem from [2, 1] concerning propositional temporal logic is

Theorem 1 *If a set of SNF rules, R, is executed using the* METATEM *algorithm, with the proviso that the oldest outstanding eventualities are attempted first at each step, then a model for R will be generated if, and only if, R is satisfiable.*

Once we add deliberation, for example re-ordering of outstanding eventualities, then this theorem becomes:

Theorem 2 *If a set of SNF rules, R, is executed using the* METATEM *algorithm, with a fair*[6] *ordering strategy for outstanding eventualities, then a model for R will be generated if, and only if, R is satisfiable.*

Thus, the *fair ordering strategy* restriction imposes a form of *fairness* on the eventuality choice mechanism [18].

It is also important to note that, once we move to either the execution of full first-order temporal specifications, or we consider the execution of an agent in an unknown environment, then completeness *can not* be guaranteed. At this stage we are only *attempting* to build a model for the temporal formula captured within the program.

If, however, we have specifications of all the participants in the multi-agent system, together with a strong specification of the communication and execution aspects, then we can, in principle, develop a full specification for the whole system. If this specification is within a decidable fragment, then we can analyse such specifications automatically.

An alternative approach is to consider the *operational semantics* of METATEM and use model checking. In [7] a common semantic basis for many agent programming languages was given, with METATEM being one of the languages considered. In [3], a model checker was developed for this common core and so, in principle, METATEM agents can be verified via this route also (although model-checking for METATEM agents has not yet been carried out).

[5] The intended purpose of an agent's `Known` set is that of a list of contacts or address book and is partly implemented for efficiency reasons.

[6] By *fair* we mean a strategy that ensures that if an eventuality is outstanding for an innite number of steps, then at some point in the execution that eventuality will continually be attempted.

2.3 Software Engineering Issues

In common with many agent-oriented languages, the aim of METATEM is to capture the highest level of deliberative behaviour required of a system and to provide a clear and concise set of constructs with which to express it. It is anticipated that METATEM code will account for only a small minority of a given system's code-base, whilst the majority of functionality will be engineered with whichever traditional (or otherwise) technique is most appropriate for the application domain concerned. Software engineering issues for METATEM therefore reduce to the following questions. Which methodologies are recommended/suited to the engineering of systems that employ METATEM agents? Can a METATEM agent be interfaced with other technologies? In this section we attempt to answer these questions.

2.3.1 Methodology

Although much work can be done to improve our understanding of this important aspect of agent technology, we are able to make some constructive comments about techniques that have been employed during METATEM's evolution. The METATEM approach can be considered a generalisation of a number of programming techniques, for instance, from object-oriented programming; agents can be readily organised into access-controlled groups [21] (akin to packages) and a type system that implements a kind of inheritance has been adopted as a method of ensuring that certain behavioural contracts are fulfilled.

Whilst informal agent-oriented design methodologies such as Gaia [31] and Prometheus [28] are well suited to capturing global system behaviour and agent interactions, we feel that the behaviour of an individual METATEM agent requires a more formal design approach if the principle benefit of the execution method is to be realised (that of direct execution of a logical specification). One way to arrive at a precise representation of an individual agent's behaviour is to model it as a finite-state machine [16]. The derivation of temporal formulae from such a model is a largely mechanical process.

In other work [15], an iterative approach to system design has been explored that makes use of the agent grouping structures to make iterative refinement of a design, by decomposing atomic agents into groups of sub-ordinate agents. In this way, by adopting appropriate organisation abstractions during each iteration, complex structures and relationships are developed. This is a promising approach but needs further development.

2.3.2 Integration

A truly agent centred philosophy has been adopted throughout the evolution of the METATEM approach to multi-agent programming, that considers all entities to be an agent with an associated temporal specification. Thus, system components that are developed and implemented with technology other than agent-oriented technology[7] must be wrapped inside a METATEM agent. A straight-forward method for achieving this via a `Java` API is provided, such that any technology that can be interfaced with `Java`, can also be interfaced with METATEM.

2.4 Extensions

A number of extensions to METATEM have been explored, by the authors and others, all of which have previously published theory and some of which have benefited from an imple-

[7] In fact, any technology other than METATEM!

mentation. However, as none are yet included in the current METATEM implementation we simply provide a list and refer the reader to relevant publications.

1. Belief predicates and epistemic properties have been explored in [14] and are expected to be included in the current METATEM implementation in the future.
2. Bounded belief and resource-bounded reasoning is described in [19, 20] and it is intended that this is implemented alongside item 1 above.
3. Probabilistic belief was explored by Ferreira et al. in [6, 5].
4. The use of METATEM as a high-level process coordination language has been explored in [25].
5. More recently, the notion of context and its applicability to agent organisation is the subject of ongoing research [8, 9].

3 Platform

A METATEM implementation, its documentation and some simple examples are available from the following URL.

`http://www.csc.liv.ac.uk/~anthony/metatem.html`

3.1 Defining Programs

The METATEM system is a `Java` application that has no dependencies other than a `Java` Runtime Environment that supports `Java 6.0` code. Defining a multi-agent system with METATEM involves creating the following source files (each are plain-text files):

1. System file;
2. Agent file;
3. Rule file.

System file. The system file declares the agents and the initial relationships between agents. For each agent, a name is declared and a type is given that corresponds with an agent file. Any initial relationships between the declared agents are declared by enumerating the members of each agent's `Content`, `Context` and `Known` sets. An example system file is shown below.

```
agent fred :  "example/robot.agent";
agent Barney :  "example/robot.agent";
agent wilma :  "example/boss.agent";

fred {
   Context :  wilma;
}

barney {
   Known :  wilma, barney;
}

wilma {
```

```
   Content :  wilma;
   Known :  barney;
}
```

Agent file. An agent file is defined for each agent type. It defines a METATEM program by declaring and importing a series of temporal constraints and rule blocks. In most cases, the majority of an agent file's content will be any number of rule blocks, each containing the runtime behaviour of an agent and described using the three rule types described in Section 2.1. In the preamble to these rule blocks a number of abilities, runtime options, meta-predicates and include statements can be declared. An example agent file is shown below.

```
// a traffic light controller
type agent;

ability timer:  metatem.agent.ability.Timer;

at_least 1 red;
at_most 1 green;
at_most 1 amber;

startblock:  {
    start => red(east_west);
    start => red(north_south);
}

ruleblock:  {

    // illegal combinations
    amber(X) => ~green(X);
    red(X) => ~green(X);

    // when red it must at some time turn to amber...
    red(X) => SOMETIME amber(X);

    //...  but not before the other light is red only
    red(X) & amber(Y) & X\=Y => NEXT ~amber(X);
    red(X) & green(Y) & X\=Y => NEXT ~amber(X);

    // red must hold when amber is shown
    red(X) & ~amber(X) => NEXT red(X);

    // once amber is shown, green can be displayed
    amber(X) & red(X) => NEXT green(X);
    amber(X) & red(X) => NEXT ~amber(X);

    // amber follows green
    green(X) => NEXT amber(X) | green(X);
    green(X) => NEXT ~red(X);
```

```
    // red follows a single amber
    amber(X) & ~red(X) => NEXT red(X);
}
```

Rule file. Rule files are a programming convenience that allows multiple agent types to re-use rule-blocks via 'include' statements. They contain only rule blocks.

The full syntax of each file type, in Backus-Naur form, is provided in the system documentation.

3.1.1 Built-in predicates

On top of the support for first order logic, METATEM provides a number of constructs commonly found in Prolog implementations and sometimes known as built-in predicates. Arithmetic is supported by the `is/2` and `lessThan/2` predicates. The declaration and manipulation of sets is achieved with set expressions that support *set union*, *set intersection* and *set difference* operators. Finally, *quoted* terms are supported along with a number of built-in predicates that allow a formula to be represented as a term if required.

3.1.2 Agent abilities

METATEM's mechanism for giving agents the ability to act on their environment is achieved via special predicates we call *abilities*. The platform provides `send` and `timer` abilities which allow agents to send messages to other agents immediately and to themselves after a period of delay, respectively. Application specific actions are supported by the METATEM API and involve the creation of a `Java` class that extends the API's `AgentAbility` class for each action. Once declared in the header of an agent file, abilities may appear wherever it is valid for a conventional predicate to appear. Abilities have the same logical semantics, on a state by state basis, as conventional predicates. However, a special 'external' ability type exists which, when executed in a state, prevents the METATEM interpreter from backtracking over that state. Characterisation of abilities as internal or external is of course dependent upon the application but as an example, a database select query might be implemented as an internal ability whereas an update query might be considered external. Internal abilities are reversible in the sense that it is safe to backtrack over them without the need for an equal and opposite action—as the action is deemed to have no side-effects.

3.2 Available tools and documentation

The current version of METATEM has a default command-line execution and output, however a basic graphical visualisation tool is also provided that provides a dynamic visualisation of the relationships between agents during execution and a monitoring facility that provides the ability to isolate individual agents and monitor their logical state. Proposals for the near future include a web interface for online demonstration and teaching purposes. A screen-shot of the tool showing the visualisation of example multi-agent structures is given in Fig. 5.

Developer's documentation is included in the download available from the project web page.

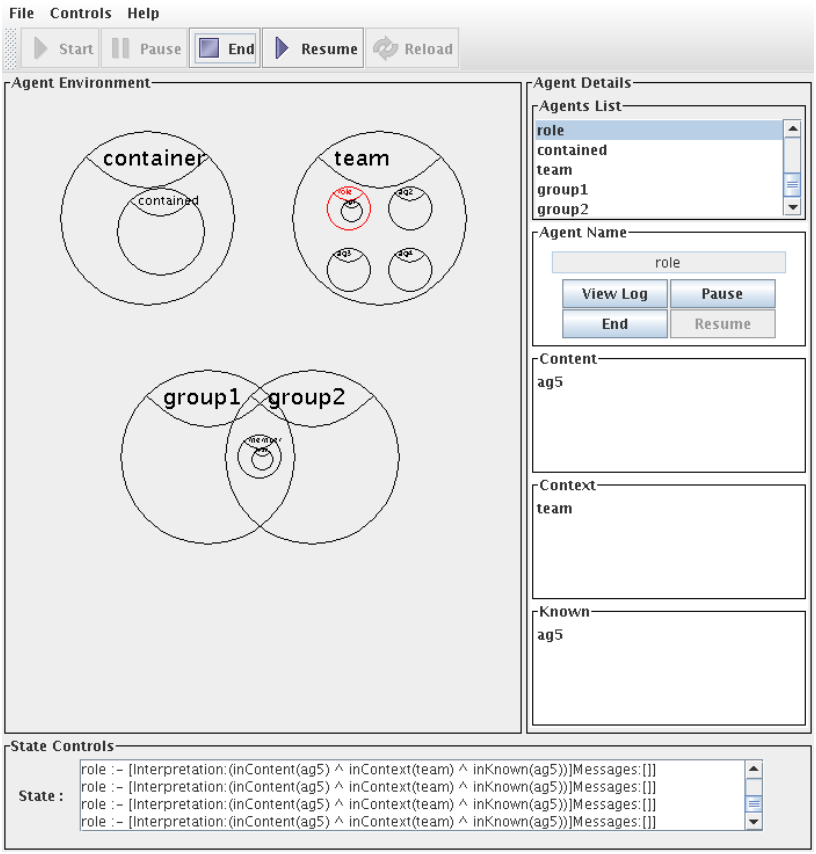


Figure 5: A visualisation tool for the control and monitoring of METATEM agents.

3.3 Standards compliance, interoperability and portability

The current METATEM implementation is written entirely in `Java` using no platform dependent APIs and is thus portable to any platform supporting a `Java 1.6` (or later) runtime environment. Although METATEM does not support heterogeneous agents directly, the creation of METATEM wrapper agents for heterogeneous agents is possible and covered by the developers' documentation.

METATEM agents are executed concurrently, each having a thread of their own in the same virtual machine. There are no plans to distribute agents across a network but the current implementation does not preclude this extension if it were required.

It is important to note that the use of logical notations, together with strong notions of execution and completeness, ensure that the language has a close relationship with its semantics. The flexibility of general first-order predicates, and of the content/context structures, ensures that semantics are both user definable and transparent.

3.4 Other features of the platform

METATEM's most distinctive feature is the inclusion of structuring constructs in the core of the language (Context and Content sets) which enable agent organisation techniques that have the flexibility to be user-definable.

The nature of the declarative logic that describes the behaviour of a METATEM agent can lead, in cases where agents are faced with many choices, to slow execution. In these cases, one can often ameliorate the effects of logical complexity by appropriate use of the deliberation meta predicates discussed in Section 2.1.2.

4 Applications supported by the language and/or the platform

We should make it clear that the current METATEM implementation is a prototype implementation that has been put to experimental use and has not been used for any real-world applications, though we now feel it is mature enough to be considered for use in the wider academic world. Its formal underpinnings and strong semantics make it a natural choice for application areas that require a high degree of clarity at a high-level of abstraction, particularly where time features prominently in the specification or where verification of system properties may be required. Features of the implementation such as meta-deliberation make METATEM a candidate language when agents must have the ability to reason about and/or modify their own reasoning, whilst the built-in agent grouping mechanism aims to support applications that comprise a significant number of agents with overlapping concerns. The natural handling of concurrency as autonomous METATEM agents, each defined by a formal temporal specification, lends itself to use as a language for coordinating processes between which dependencies exist—the specification of a coordination model [25]. The authors believe that METATEM will prove useful for the specifcation of highly distributed systems such as those found in pervasive computing scenarios, this is therefore the focus of their current application research.

Though the current platform has not been applied widely, the METATEM language itself has been used in (or at least has inspired work in) several areas. For example, in tackling planning, temporal logics have been increasingly used to control search. In [27] direct execution of temporal formula is used to directly implement planning. METATEM,

and specifically the underlying normal form, provide a concise description of temporal behaviours and this has been used in [4] to implement a form of agent verification. Similarly, the encoding of verification problems using the METATEM rule form has been shown to be beneficial to the efficiency of model checkers [23].

Acknowledgements. Many people have influenced the current METATEM implementation but Benjamin Hirsch and Chiara Ghidini must be thanked for their significant contribution to the current implementation and agent grouping strategies, respectively. In addition, thanks go to Michael Cieslar for developing the graphical visualisation tool.

References

[1] Barringer, H., Fisher, M., Gabbay, D., Owens, R., Reynolds, M. (eds.): The Imperative Future: Principles of Executable Temporal Logics. John Wiley & Sons, Inc., New York, NY, USA (1996)

[2] Barringer, H., Fisher, M., Gabbay, D.M., Gough, G., Owens, R.: METATEM: An introduction. Formal Aspects of Computing **7**(5), 533–549 (1995)

[3] Bordini, R.H., Dennis, L.A., Farwer, B., Fisher, M.: Automated Verification of Multi-Agent Programs. In: Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 69–78 (2008)

[4] Costantini, S., Dell'Acqua, P., Pereira, L.M., Tsintza, P.: Specification and Dynamic Verification of Agent Properties. In: Proc. Ninth International Conference on Computational Logic in Multi-Agent Systems (2008)

[5] de Carvalho Ferreira, N., Fisher, M., van der Hoek, W.: Specifying and Reasoning about Uncertain Agents. International Journal of Approximate Reasoning **49**(1), 35–51 (2008)

[6] de Carvalho Ferreira, N., Fisher, M., van der Hoek, W.: Logical Implementation of Uncertain Agents. In: Proceedings of 12th Portuguese Conference on Artificial Intelligence (EPIA), *Lecture Notes in Computer Science*, vol. 3808. Springer (2005)

[7] Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M., Wooldridge, M.: A Common Semantic Basis for BDI Languages. In: Proc. Seventh International Workshop on Programming Multiagent Systems (ProMAS), *Lecture Notes in Artificial Intelligence*, vol. 4908, pp. 124–139. Springer Verlag (2008)

[8] Dennis, L.A., Fisher, M., Hepple, A.: A Common Basis for Agent Organisation in BDI Languages. In: Proc. 1st International Workshop on LAnguages, methodologies and Development tools for multi-agent Systems (LADS), *Lecture Notes in Computer Science*, vol. 5118, pp. 171–188. Springer (2008)

[9] Dennis, L.A., Fisher, M., Hepple, A.: Language Constructs for Multi-Agent Programming. In: Proc. 8th Workshop on Computational Logic in Multi-Agent Systems (CLIMA), *Lecture Notes in Artificial Intelligence*, vol. 5056, pp. 137–156. Springer (2008)

[10] Dixon, C., Fisher, M., Konev, B.: Temporal Logic with Capacity Constraints. In: Proc. 6th International Symposium on Frontiers of Combining Systems, *Lecture Notes in Computer Science*, vol. 4720, pp. 163–177. Springer (2007)

[11] Emerson, E.A.: Temporal and Modal Logic. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, pp. 996–1072. Elsevier (1990)

[12] Fisher, M.: A Temporal Semantics for Concurrent METATEM. Journal of Symbolic Computation **22**(5/6), 627–648 (1996)

[13] Fisher, M.: A Normal Form for Temporal Logic and its Application in Theorem-Proving and Execution. Journal of Logic and Computation **7**(4), 429–456 (1997)

[14] Fisher, M.: Implementing BDI-like systems by direct execution. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI), vol. 1, pp. 316–321. Morgan Kaufmann, San Fransisco, CA, USA (1997)

[15] Fisher, M.: Towards the Refinement of Executable Temporal Objects. In: H. Bowman, J. Derrick (eds.) Formal Methods for Open Object-Based Distributed Systems. Chapman & Hall (1997)

[16] Fisher, M.: Temporal Development Methods for Agent-Based Systems. Journal of Autonomous Agents and Multi-Agent Systems **10**(1), 41–66 (2005)

[17] Fisher, M.: Temporal Representation and Reasoning. In: F. van Harmelen, B. Porter, V. Lifschitz (eds.) Handbook of Knowledge Representation, *Foundations of Artificial Intelligence*, vol. 2. Elsevier Press (2007)

[18] Fisher, M.: Agent Deliberation in an Executable Temporal Framework. Technical Report ULCS-08-014, Department of Computer Science, University of Liverpool, UK (2008)

[19] Fisher, M., Ghidini, C.: Programming Resource-Bounded Deliberative Agents. In: Proc. 16th International Joint Conference on Artificial Intelligence (IJCAI), pp. 200–205. Morgan Kaufmann (1999)

[20] Fisher, M., Ghidini, C.: Exploring the Future with Resource-Bounded Agents. Journal of Logic, Language and Information **18**(1), 3–21 (2009)

[21] Fisher, M., Ghidini, C., Hirsch, B.: Programming Groups of Rational Agents. In: Proc, International Workshop on Computational Logic in Multi-Agent Systems IV (CLIMA), *Lecture Notes in Artificial Intelligence*, vol. 3259, pp. 16–33. Springer-Verlag, Heidelberg, Germany (2004)

[22] Fisher, M., Kakoudakis, T.: Flexible Agent Grouping in Executable Temporal Logic. In: Proceedings of Twelfth International Symposium on Languages for Intensional Programming (ISLIP). World Scientific Press (1999)

[23] Frisch, A.M., Sheridan, D., Walsh, T.: A Fixpoint Based Encoding for Bounded Model Checking. In: Proc. 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD), *Lecture Notes in Computer Science*, vol. 2517, pp. 238–255. Springer (2002)

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

[25] Kellett, A., Fisher, M.: Concurrent METATEM as a Coordination Language. In: Coordination Languages and Models, *Lecture Notes in Computer Science*, vol. 1282. Springer-Verlag (1997)

[26] Kesten, Y., Manna, Z., Pnueli, A.: Temporal Verification of Simulation and Refinement. In: A Decade of Concurrency, *Lecture Notes in Computer Science*, vol. 803, pp. 273–346. Springer-Verlag (1994)

[27] Mayer, M.C., Limongelli, C., Orlandini, A., Poggioni, V.: Linear Temporal Logic as an Executable Semantics for Planning Languages. Journal of Logic, Language and Information **16**(1) (2007)

[28] Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. John Wiley and Sons (2004)

[29] Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In: Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR), pp. 473–484. Morgan Kaufmann, San Fransisco, CA, USA (1991)

[30] Rao, A.S., Georgeff, M.P.: BDI agents: From theory to practice. In: Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS), pp. 312–319. IEEE Press, Washington, DC, USA (1995)

[31] Wooldridge, M., Jennings, N.R., Kinny, D.: The gaia methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems **3**(3), 285–312 (2000)