A Survey of Concurrent METATEM — The Language and its Applications

Michael Fisher

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

`M.Fisher@mmu.ac.uk`

Abstract. In this paper we present a survey of work relating to the Concurrent METATEM programming language. In addition to a description of the basic Concurrent METATEM system, which incorporates the direct execution of temporal formulae, a variety of extensions that have either been implemented or proposed are outlined. Although still in the development stage, there appear to be many areas where such a language could be applied. We present a variety of sample applications, highlighting the particular features of Concurrent METATEM that we believe will make it appropriate for use in these areas.

1 Introduction

Concurrent METATEM is a language based upon the direct execution of temporal formulae [15]. It consists of two distinct aspects: an execution mechanism for temporal formulae in a particular form; and an operational model that treats single executable temporal logic programs as asynchronously executing objects in a concurrent object-based system. The motivation for the development of this language has been provided by many areas. For example, being based upon executable logic, the language can be used as part of the specification and prototyping of reactive systems. Also, as it uses *temporal*, rather than classical, logic the language provides a high-level programming notation in which the dynamic attributes of individual components can be concisely represented. Finally, it incorporates a novel model of concurrent computation which has a range of applications in distributed systems.

The logic used as a basis for Concurrent METATEM is a discrete, linear temporal logic. This presents a simple view of time and, in doing so, provides the systems designer with a direct analogy between the models for the logic and the discrete, linear execution sequences with which he or she is familiar. This, together with the fact that a restricted form of this temporal logic is executed, ensures that the temporal aspects of Concurrent METATEM are manageable, both for the programmer and for the implementation of the system.

Each object executes its own set of temporal formulae and, in doing so, (effectively) generates an infinite sequence of states. As each object executes asynchronously, it constructs a separate execution sequence. Further, within Concurrent METATEM, a mechanism is provided for communication between separate objects. This simply consists of a partition of each object's propositions into those controlled by the object and those controlled by the environment. To fit in with this logical view of communication, whilst also providing a flexible and powerful message-passing mechanism, *broadcast* message-passing is used to pass information between objects.

This relatively straightforward combination of executable temporal logic, a concurrent object-model and broadcast message-passing forms the basis of Concurrent METATEM. Together, these features provide an coherent and consistent programming model within which a variety of reactive systems can be represented and implemented.

This review is structured as follows. Firstly, in §2, we will present the core elements of the language, including basic temporal execution, together with its operational and communication model. In §3, we will outline various extensions to the core language that have either been proposed or implemented. Thus, together, these two sections will provide a survey of the language itself. In §4, we provide a range of sample applications of Concurrent METATEM, utilising both the core features of the language and its extensions. Although, in some cases, these are only *potential* applications, we will argue that the properties that Concurrent METATEM exhibits make it suitable for application in these areas. Thus, the range of examples in this section provide a survey of the current and potential applications of the language. Finally, in §5, we present concluding remarks, including brief overviews of related work, current status of the implementation, and possible future work.

2 **Concurrent** METATEM

In this section, we will provide an introduction to the Concurrent METATEM system, which consists of objects, whose behaviour is implemented using executable temporal logic, communicating via broadcast message-passing. Concurrent METATEM itself was originally developed as an extension of the sequential execution of temporal logic programs provided by METATEM, an executable temporal logic described in [3, 10]. The rules that are executed are based upon the normal form developed for temporal theorem-proving [14], while the concurrent operational model was outlined in [9] and developed to its current state in [15].

2.1 An Overview of the Approach

While it is possible to program objects in Concurrent METATEM using a small range of temporal operators (just the *last-time* and *sometime in the future* operators), a large range of temporal operators are available in the interests of convenience. Most of these operators are eliminated during the transformation from the rules input by the programmer to rules that are actually executed. These transformations follow those used in producing a normal form for temporal formulae [14]. The *transformed* rules are then executed directly, providing the dynamic behaviour of the individual object. The components of this core language represent the basic descriptive elements of our system:

– logical properties of individual states, through the use of classical logic to represent declarative description of a state;

- properties of state transformation steps, through the use of the *last-time* operator in conjunction with constraints on the present state;
- global properties of temporal sequences, through the use of multiple state transformation steps together with the *sometime in the future* operator.

An important aspect of the language is the mixture, within the execution of these temporal formulae, of both declarative and imperative aspects. For example, we might provide a *declarative* description of a particular state through a formula such as $p \wedge q$, while also providing an *imperative* rule describing *how* to generate the current state from the last one, such as $\circledcirc p \;\Rightarrow\; q$ (here, '$\circledcirc$' is the *last-time* operator).

2.2 Temporal Logic

Temporal logic can be seen as classical logic extended with various modalities representing temporal aspects of logical formulae. The temporal logic we use is based on a discrete, linear model and, thus, time is modelled as an infinite sequence of discrete states, with an identified starting point, called 'the beginning of time'. Classical formulae are used to represent constraints within individual states, while temporal formulae represent constraints *between* states. As formulae are interpreted at particular states in this sequence, operators which refer to both the past and future are required.

The logic we use, called First-Order METATEM Logic (FML) is a simple first-order temporal logic, based on discrete, linear models with finite past and infinite future. Below we give an outline of its syntax and semantics (for a more detailed presentation, see [3, 14]).

Syntax As in classical logics, the *terms* the language are constructed from a set of *constant symbols* ($\mathcal{L}_c$), a set of *variable symbols* ($\mathcal{L}_v$) and a set of *function symbols* ($\mathcal{L}_f$). From these elements, the set of terms, $\mathcal{L}_t$, can be generated. The other symbols used in FML are as follows.

- A set, $\mathcal{L}_p$, of *predicate symbols*, contains elements usually represented by strings of lower-case alphabetic characters.
- Classical connectives, $\neg$, $\vee$, $\wedge$, $\Rightarrow$, and $\Leftrightarrow$.
- Future-time temporal operators, including unary operators $\bigcirc$, $\Diamond$ and $\Box$, and binary operators $\mathcal{U}$ and $\mathcal{W}$.
- Past-time temporal operators, including unary operators $\circledcirc$, $\bullet$, $\blacklozenge$ and $\blacksquare$, and binary operators $\mathcal{S}$ and $\mathcal{Z}$.
- Quantifiers, $\forall$ and $\exists$.
- '(' and ')' which are, as usual, used to avoid ambiguity.

The set of well-formed formulae of FML (WFF_f) is defined as follows.

1. If $t_1, \ldots, t_n$ are in $\mathcal{L}_t$, and p is a predicate of arity n, then $p(t_1, \ldots, t_n)$ is in WFF_f.
2. if A and B are in WFF_f, then the following are in WFF_f

$$\begin{array}{llllll}
\neg A & A \vee B & A \wedge B & A \Rightarrow B & (A) & \\
\Diamond A & \Box A & A\,\mathcal{U}\,B & A\,\mathcal{W}\,B & \bigcirc A & \\
\blacklozenge A & \blacksquare A & A\,\mathcal{S}\,B & A\,\mathcal{Z}\,B & \circledcirc A & \bullet A
\end{array}$$

3. If A is in WFF_f and v is in $\mathcal{L}_v$, then $\exists v.\ A$ and $\forall v.\ A$ are both in WFF_f.

Sub-classifications of WFF_f are defined as follows. A *literal* is defined as either a predicate symbol applied to an appropriate term, or the negation of such a predicate.
A *State-formula* is either a literal or a non-temporal combination of other state-formulae. *Future-time formulae* contain only classical and future-time temporal operators, while *past-time formulae* contain only classical and past-time temporal operators. *Strict* versions of both these categorisations can be formed by removing from them formulae that refer to the present.

Semantics The basic models of FML are discrete, linear structures with finite past and infinite future. To this structure a domain, $\mathcal{D}$, and mappings from elements of the language to denotations are added. Thus the full model structure for FML is

$$\mathcal{M} \ = \ \langle \sigma, \mathcal{D}, \pi_c, \pi_f, \pi_p \rangle$$

where

– σ, a sequence of states $s_0,\ s_1,\ s_2,\ s_3,\ \ldots,$
– $\mathcal{D}$ is the object-level domain,
– π_c is a map from $\mathcal{L}_c$ to $\mathcal{D}$,
– π_f is a map from $\mathcal{L}_f$ to $\mathcal{D}^n \longrightarrow \mathcal{D}$, where n is the arity of f, and,
– π_p is a map from $\mathbf{N} \times \mathcal{L}_p$ to $\mathcal{D}^n \longrightarrow \{T, F\}$.

Thus, for a particular state s, and a particular predicate p of arity n, $\pi_p(s, p)$ represents a map from n-tuples of elements of $\mathcal{D}$ to T or F. Note that the *constant domain* assumption is used, i.e. that $\mathcal{D}$ is constant for every state, and that both constant and function symbols have fixed interpretations.

The semantics of FML is given with respect to a model, $\mathcal{M}$, a state, s_i, at which the formula is to be interpreted, and a variable assignment, V. From the model and the variable assignment, we are able to generate a *term assignment*, $\tau_{v\pi}$, which maps every term to its appropriate element of the domain, $\mathcal{D}$.

We first consider the semantics of atomic predicates:

$$\langle \mathcal{M}, s_i, V \rangle \models p(x_1, \ldots, x_n) \quad \text{iff} \quad \pi_p(i, p)(\tau_{v\pi}(x_1), \ldots, \tau_{v\pi}(x_n)) \ = \ T\,.$$

The semantics of the standard propositional connectives is as in classical logic, e.g.,

$$\langle \mathcal{M}, s_i, V \rangle \models \ \varphi \vee \psi \quad \text{iff} \quad \langle \mathcal{M}, s_i, V \rangle \models \varphi \ \text{ or } \ \langle \mathcal{M}, s_i, V \rangle \models \psi$$

The semantics of the unary future-time temporal operators is defined as follows

$$\begin{array}{lll}
\langle \mathcal{M}, s_i, V \rangle \models \ \bigcirc\varphi & \text{iff} & \langle \mathcal{M}, s_{i+1}, V \rangle \models \varphi \\
\langle \mathcal{M}, s_i, V \rangle \models \ \Diamond\varphi & \text{iff} & \text{there exists a } j \geq i \text{ such that } \langle \mathcal{M}, s_j, V \rangle \models \varphi \\
\langle \mathcal{M}, s_i, V \rangle \models \ \Box\varphi & \text{iff} & \text{for all } j \geq i \text{ then } \langle \mathcal{M}, s_j, V \rangle \models \varphi\,.
\end{array}$$

The informal semantics of these operators is as follows: $\bigcirc\varphi$ means that φ must be satisfied in the *next* state; $\Diamond\varphi$ means that φ must be satisfied at *some* state in the future; $\Box\varphi$ means that φ must be satisfied at *all* states in the future.

The two binary future-time temporal operators that we use are interpreted as follows

$$\begin{array}{lll}
\langle \mathcal{M}, s_i, V\rangle \models \varphi\,\mathcal{U}\,\psi & \text{iff} & \text{there exists a } k \geq i \text{ such that } \langle \mathcal{M}, s_k, V\rangle \models \psi \\
 & & \text{and for all } i \leq j < k \text{ then } \langle \mathcal{M}, s_j, V\rangle \models \varphi \\
\langle \mathcal{M}, s_i, V\rangle \models \varphi\,\mathcal{W}\,\psi & \text{iff} & \text{for all } j \geq i \text{ then } \langle \mathcal{M}, s_j, V\rangle \models \varphi \\
 & & \text{or } \langle \mathcal{M}, s_i, V\rangle \models \varphi\,\mathcal{U}\,\psi
\end{array}$$

If past-time temporal formulae are interpreted at a particular state, s_i, then states with indices less than i are 'in the past' of the state s_i. The semantics of unary past-time operators is given as follows:

$$\begin{array}{lll}
\langle \mathcal{M}, s_i, V\rangle \models \bullet\varphi & \text{iff} & i = 0 \text{ or } \langle \mathcal{M}, s_{i-1}, V\rangle \models \varphi \\
\langle \mathcal{M}, s_i, V\rangle \models \circledcirc\varphi & \text{iff} & i > 0 \text{ and } \langle \mathcal{M}, s_{i-1}, V\rangle \models \varphi \\
\langle \mathcal{M}, s_i, V\rangle \models \blacklozenge\varphi & \text{iff} & \text{there exists } j \text{ such that } 0 \leq j < i \text{ and } \langle \mathcal{M}, s_j, V\rangle \models \varphi \\
\langle \mathcal{M}, s_i, V\rangle \models \blacksquare\varphi & \text{iff} & \text{for all } j \text{ such that } 0 \leq j < i \text{ then } \langle \mathcal{M}, s_i, V\rangle \models \varphi
\end{array}$$

Note that, in contrast to the future-time operators, the $\blacklozenge$ and $\blacksquare$ operators are interpreted as being *strict*, i.e. the current index is not included in the definition. Also, as there is a unique start state, termed the *beginning of time*, two different last-time operators are used. The difference between '$\circledcirc$' and '$\bullet$' is that for any formula φ, $\circledcirc\varphi$ is false at the beginning of time, while $\bullet\varphi$ is true at the beginning of time. In particular, $\bullet$**false** is only true when interpreted at the beginning of time; otherwise it is false. Note that, as the formula $\bullet$**false** appears so regularly in Concurrent METATEM, we often abbreviate it with the nullary operator '**start**', thus making obvious its association with the beginning of time (and the beginning of execution).

The semantics for the binary past-time operators $\mathcal{S}$ and $\mathcal{Z}$ relates to that for $\mathcal{U}$ and $\mathcal{W}$ just as the unary past-time operators relate to the unary future-time operators. Finally, the semantics of quantifiers is defined as follows.

$$\begin{array}{lll}
\langle \mathcal{M}, s, V\rangle \models \forall x.\ \varphi & \text{iff} & \text{for all } d \in \mathcal{D}.\ \langle \mathcal{M}, s, V \dagger [x \mapsto d]\rangle \models \varphi \\
\langle \mathcal{M}, s, V\rangle \models \exists x.\ \varphi & \text{iff} & \text{there exists } d \in \mathcal{D}. \text{ such that } \langle \mathcal{M}, s, V \dagger [x \mapsto d]\rangle \models \varphi
\end{array}$$

As the interpretation consists of a triple, comprising model, state, and assignment components, a well-formed formula, φ, is *satisfied* in a particular model, $\mathcal{M}$, at the beginning of time, s_0, under a particular variable assignment, V, if $\langle \mathcal{M}, s_0, V\rangle \models \varphi$.

2.3 Executable Rules

Next, we define the subset of FML that can be used in a Concurrent METATEM program. The description of a Concurrent METATEM object is a set of *rules*, represented by

$$\Box \bigwedge_i R_i$$

where each R_i is, in turn, of the form

'past and present formula' **implies** 'present or future formula'

Taking quantification into account, the general form of these rules becomes

$$\square \bigwedge_{i=1}^{n} \forall \overline{X_i}.\ P_i(\overline{X_i}) \ \Rightarrow\ F_i(\overline{X_i})$$

where '$\overline{X_i}$' represents a vector of variables, $X_{i_1}, X_{i_2}, \ldots, X_{i_m}$.

Each rule is further restricted to be one of the following.

$$\begin{array}{lll}
\forall \overline{X}. \qquad [\textbf{start} \wedge \bigwedge_{b=1}^{h} l_b(\overline{X})] \Rightarrow \bigvee_{j=1}^{r} m_j(\overline{X}) & & (\text{an } \textit{initial } \square\text{-rule}) \\
\forall \overline{X}.\ [(\bigcirc \bigwedge_{a=1}^{g} k_a(\overline{X})) \wedge \bigwedge_{b=1}^{h} l_b(\overline{X})] \Rightarrow \bigvee_{j=1}^{r} m_j(\overline{X}) & & (\text{a } \textit{global } \square\text{-rule}) \\
\forall \overline{X}. \qquad [\textbf{start} \wedge \bigwedge_{b=1}^{h} l_b(\overline{X})] \Rightarrow \Diamond l(\overline{X}) & & (\text{an } \textit{initial } \Diamond\text{-rule}) \\
\forall \overline{X}.\ [(\bigcirc \bigwedge_{a=1}^{g} k_a(\overline{X})) \wedge \bigwedge_{b=1}^{h} l_b(\overline{X})] \Rightarrow \Diamond l(\overline{X}) & & (\text{a } \textit{global } \Diamond\text{-rule})
\end{array}$$

where each k_a, l_b, m_j or l is a literal. Note that the left-hand side of each initial rule is a constraint only on the *first* state, while the left-hand side of each global rule represents a constraint upon a state together with its predecessor. The right-hand side of each $\square$-rule is simply a disjunction of literals referring to the current state, while the right-hand side of each $\Diamond$-rule is a single eventuality (i.e., '$\Diamond$' applied to a literal).

Note also that, although arbitrary FML formulae can be transformed into a set of rules of the form

$$\begin{array}{ll}
\forall \overline{X}. & [(\forall \overline{Y}.\ \textbf{start} \wedge \bigwedge_{b=1}^{h} l_b(\overline{X}, \overline{Y})) \Rightarrow \exists \overline{Z}.\ \bigvee_{j=1}^{r} m_j(\overline{X}, \overline{Z})] \\
\forall \overline{X}. & [(\forall \overline{Y}.\ (\bigcirc \bigwedge_{a=1}^{g} k_a(\overline{X}, \overline{Y})) \wedge \bigwedge_{b=1}^{h} l_b(\overline{X}, \overline{Y})) \Rightarrow \exists \overline{Z}.\ \bigvee_{j=1}^{r} m_j(\overline{X}, \overline{Z})] \\
\forall \overline{X}. & [(\forall \overline{Y}.\ \textbf{start} \wedge \bigwedge_{b=1}^{h} l_b(\overline{X}, \overline{Y})) \Rightarrow \exists \overline{Z}.\ \Diamond l(\overline{X}, \overline{Z})] \\
\forall \overline{X}. & [(\forall \overline{Y}.\ (\bigcirc \bigwedge_{a=1}^{g} k_a(\overline{X}, \overline{Y})) \wedge \bigwedge_{b=1}^{h} l_b(\overline{X}, \overline{Y})) \Rightarrow \exists \overline{Z}.\ \Diamond l(\overline{X}, \overline{Z})]
\end{array}$$

where each k_a, l_b, m_j or l is a literal, we here choose to execute only a subset of this normal form [14].

2.4 Execution within Objects

We now describe how a set of rules for a given object in Concurrent METATEM is executed in order to provide its basic behaviour. Concurrent METATEM uses a set of 'rules' of the above form to represent an object's internal definition. Due to the outer '$\Box$' operator present in this rule form, the rules are applied at every moment in time (i.e., at every step of the execution) during the construction of a model for the formula.

As an example of a simple set of rules for a single object, consider the following. (Note that these rules are not meant to form a 'meaningful' program – they are only given for illustrative purposes.)

$$\begin{aligned} \textbf{start} &\Rightarrow \texttt{popped(a)} \\ \circledcirc\texttt{pop(X)} &\Rightarrow \Diamond\texttt{popped(X)} \\ \circledcirc\texttt{push(Y)} &\Rightarrow \texttt{stack-full()} \lor \texttt{popped(Y)} \end{aligned}$$

Note that the 'X' and 'Y' here represent universally quantified variables. Looking at these program rules, we see that `popped(a)` is satisfied at the beginning of time and whenever `pop(X)` is satisfied in the previous moment in time, a commitment to eventually satisfy `popped(X)` is given. Similarly, whenever `push(Y)` is satisfied in the previous moment in time, then either `stack-full()` or `popped(Y)` must be satisfied.

The temporal language which forms the basis of execution within particular objects in Concurrent METATEM also provides two orthogonal mechanisms for representing choice. These are

– static indeterminacy, through the classical operator '$\lor$', and,
– temporal indeterminacy, through '$\Diamond$', the temporal operator representing *sometime in the future*.

A logical description, containing the '$\lor$' operator, of the properties of a given state represents a choice about the exact nature of the state. Although other constraints upon the state might restrict this choice, the potential for a completely non-deterministic choice is present. A formula such as $\Diamond a$ states that the proposition a must be satisfied *at some time in the future*. Thus, it represents a form of temporal indeterminacy. However, we do not model this as a truly non-deterministic choice. Rather, given a constraint, such as $\Diamond a$, the execution mechanism attempts to satisfy a as soon as possible (taking in to account any other temporal constraints). Thus, a formula such as

$$\Diamond a \land \Diamond b \land \Diamond c$$

when executed, would ensure that a, b and c are all satisfied as soon as possible. If necessary, we can add further temporal formulae representing extra ordering within these constraints, for example to ensure that c can not be satisfied until both a and b have been satisfied.

Once the object has commenced execution, it continually follows a cycle of checking which rules have antecedents satisfied by the previous state, conjoining together the consequents of these rules, rewriting this conjunction into a disjunctive form and choosing one of these disjuncts to execute. From this disjunct a state is constructed, with predicates remaining false unless otherwise constrained. The computation then moves forward to the next state where this cycle begins again. If a contradiction is found, it may be possible to backtrack to a previous choice (but see §2.5 for restrictions). This choice is constrained by the currently outstanding eventualities, i.e., formulae of the form $\Diamond a$ that have not yet been satisfied. As many as possible of these formulae are satisfied in the state constructed, starting with oldest outstanding eventuality.

Note that this basic execution mechanism is similar to that provided for (sequential) METATEM [3, 10].

We now describe the general operational model for Concurrent METATEM objects executing in the above manner. This incorporates the asynchronous execution of individual objects, dynamic attributes of object interfaces, and the communication mechanism between objects.

2.5 Concurrent Object-Based Operational Model

The computational model used in Concurrent METATEM combines the two notions of *objects* and *concurrency*. Objects are here considered to be self contained entities, encapsulating both data and behaviour, and communicating via message-passing. In particular, Concurrent METATEM has the following fundamental properties.

1. The basic mechanism for communication between objects is *broadcast* message-passing.
2. Objects are not message driven — they begin executing from the moment they are created and continue even while no messages are received.
3. Each object contains a set of temporal rules representing its behaviour.
4. Objects execute asynchronously.

So, rather than seeing computation as objects sending mail messages to each other, and thus invoking some activity (as in Actor systems [2]), computation in a collection of Concurrent METATEM objects can be visualised as independent entities *listening* to messages broadcast from other objects.

In addition to this basic framework, the operational model exhibits several important features which are described in more detail below.

Communication Mechanism Broadcast communication is used, not only as it is a flexible and powerful communication mechanism, but also as it has a logical meaning within our system, namely that of passing valuations for predicates throughout the system.

In order to implement communication as an integral part of execution, we categorise the basic predicates used in the language, with several categories of predicate corresponding to messages to and from the object, as follows.

– *Environment* predicates, which 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 ‘`push(Y)`’, is only true if a message of the form ‘`push(b)`’ has just been received (for some argument ‘`b`’, which unifies with ‘`Y`’).

– *Component* predicates, which represent messages broadcast from the object.
  When a component predicate is made true, it has the (side-)effect of broadcasting the corresponding message to the environment. For example, if the formula '`popped(e)`' is made true, where `popped` is a component predicate, then the message '`popped(e)`' is broadcast.
– *Internal* predicates, which have no external effect.
  These predicates are used as part of formulae participating in the internal computation of the object and, as such, do not correspond either to message-sending or message reception.
  This category of predicates may include various *primitive* operations.

Object Interfaces Networks of Concurrent METATEM objects communicate via broadcasting messages and individual objects only act upon certain identified messages. Thus, an object must be able to filter out messages that it wishes to recognise, ignoring all others. The definition of which messages an object recognises, together with a definition of the messages that an object may itself produce, is provided by the *interface definition* for that particular object.

The interface definition for an object, for example '`stack`', is defined in the following way

```
stack(pop,push)[popped,stack-full].
```

Here, {`pop`, `push`} is the set of messages the object recognises, while the object itself is able to produce the messages {`popped`, `stack-full`}. Note that these sets of messages need not be disjoint – an object may broadcast messages that the object itself recognises. In this case, messages sent by an object to itself are recognised immediately. Note also that many distinct objects may broadcast and recognise the same messages.

Backtracking In general, if an object's execution mechanism is based on the execution of logical statements, then a computation may involve backtracking. Objects may backtrack, with the proviso that an object may not backtrack past the broadcasting of a message. Consequently, in broadcasting a message to its environment, an object effectively *commits* the execution to that particular path. Thus, the basic operation of an object can be thought of as a period of internal execution, possibly involving backtracking, followed by appropriate broadcasts to its environment. The analogy with a collections of humans is of a period of thinking, followed by some (broadcasted) action, e.g. speech. Backtracking can occur during thinking, but once an action has been carried out, it cannot be undone.

3 Extensions of the Basic System

Above, we have described the principles behind the basic Concurrent METATEM system. In this section, we will outline various extensions that have ither been implemented, or are actively under development.

3.1 Autonomous Objects

There are a variety of extensions to Concurrent METATEM that allow objects to have control, to some extent, over their own execution. In particular, these extensions allow objects to control their interface with their environment.

Dynamic Interfaces The interface definition of an object defines the initial set of messages that are recognised by that object. However, the object may dynamically change the set of messages that it recognises. In particular, an object can either start 'listening' for a new type of message, or start 'ignoring' previously recognised message types. For example, given an original object interface such as

```
stack(pop,push)[popped,stack-full]
```

the object may dynamically choose to stop recognising '`pop`' messages, for example by executing '`ignore(pop)`'. This effectively gives the object the new interface

```
stack(push)[popped,stack-full].
```

Dynamic Message Queues Each object in the system has, associated with it, a *message queue* representing the messages that the object has recognised, but has yet to process. The number of messages that an object reads from its message queue during an execution step is initially defined by the object's interface. In principle, the execution of an object is based on the set of messages received by the object since the last execution step it completed. Thus, Concurrent METATEM objects process sets of messages, rather than enforcing some linearisation on the order of arrival of messages.

We are developing extensions to enable the object to dynamically modify its own behaviour regarding the manipulation of its message queue. The default behaviour of an object is for it to, at every execution step, read a sequence of messages from its input queue up to either the end of the queue, or the repetition of a message. For example, if the message queue is

```
pop(a), push(c), pop(d), pop(e), push(c), pop(f), pop(a), ...
```

then in one execution step the set of messages read in is

```
{ pop(a), push(c), pop(d), pop(e) }
```

and the message queue remaining is

```
push(c), pop(f), pop(a), ...
```

This behaviour can be modified dynamically so that, for example, objects read only one message at a time from their input queue, or read up to the second occurrence of either of several messages. Complex varieties of message queue manipulation may be defined, depending on the properties of the Concurrent METATEM objects. Such an extension is being developed through the use of extended primitive predicates (an alternative approach is through the use of meta-level features [4]).

3.2 Synchronisation Mechanisms

There are two approaches to the synchronisation of asynchronously executing objects in Concurrent METATEM, outlined as follows.

1. An object asks for something, continues processing, but does something when the answer arrives.
   Thus, a request is made and execution continues, but when a reply is received from the environment, some suitable action is taken.

$$\begin{array}{r@{\;\Rightarrow\;}l} \texttt{....} & \texttt{ask} \\ \circledcirc\ \texttt{answer} & \texttt{do\_it} \end{array}$$

   Here `ask` is a component predicate and `answer` is an environment predicate.
2. An object asks for something, *waits* for an answer, and does something when the answer arrives.
   To suspend execution whilst waiting for a synchronisation message, an extra synchronisation rule is added to (1) above, giving:

$$\begin{array}{r@{\;\Rightarrow\;}l} \texttt{....} & \texttt{ask} \\ \circledcirc\ \texttt{answer} & \texttt{do\_it} \\ \circledcirc\ \texttt{ask} & \texttt{answer} \end{array}$$

   If an `ask` message has been sent, then the only way to satisfy this last rule is to ensure that `answer` is received in the next state. Thus, the object cannot execute further until the required message arrives, and consequently it is suspended.
   Note that, until the appropriate synchronisation message arrives, execution of the process suspends and no further incoming messages are processed (though they are recorded).

3.3 Point-to-point message-passing

Although broadcast message passing has been defined as the primitive mechanism for communication, point-to-point message-passing can be implemented on top of this. Such a mechanism can either be provided as part of the implementation of Concurrent METATEM, and made available through extra primitives, or can be provided by utilising meta-level features within each Concurrent METATEM object (see [4] for an outline of these basic features). To provide point-to-point message-passing using meta-level features, every Concurrent METATEM object (that wishes to take part in such a scheme) must include a 'meta-rule' of the form

$$\texttt{send(me,X)} \;\Rightarrow\; \texttt{X}$$

within its definition. Here '`me`' is the name of that particular object. Thus, whenever an object, `obj1`, wishes to send the message '`p(a)`' to another object, `obj2`, the following message must be broadcast by `obj1`:

$$\texttt{send(obj2,p(a)).}$$

If such a scheme is enforced in all objects, then point-to-point message passing is available throughout the system.

An alternative approach is to add an extra *destination* argument to each message. Thus, to send the message '`p(a)`' to object '`obj2`', we broadcast

```
p(obj2,a)
```

and ensure that the rules in `obj2` itself deal with this appropriately.

3.4 Synchronous Concurrent METATEM

A variation on the basic execution scheme for Concurrent METATEM is the development of *Synchronous* Concurrent METATEM. In this system, each object executes in step and messages sent from any object reach all other objects by the start of the next step. This simplification removes the need for synchronisation using environment predicates, but also reduces the flexibility of the approach (see [5] for an outline of this approach).

3.5 Groups

Finally, objects are also members of *groups*. Each object may be a member of several groups. When an object sends a message, that message is, by default, broadcast to all the members of its group(s), but to no other objects. Alternatively an object can select to broadcast only to certain groups (of which it is a member). This mechanism allows the development of complex structuring within the object space and provides the potential for innovative applications, such as the use of groups to represent physical properties of objects. For example, if we assume that any two objects in the same group can 'see' each other, then movement broadcast from one object can be detected by the other object. Similarly, if an object moves 'out of sight', it moves out of the group and thus the objects that remain in the group 'lose sight' of it. Examples, such as this (see also §4.3), show some of the power of the group concept.

4 Examples

In this section we provide a range of examples exhibiting some of the applications of Concurrent METATEM. Several of these examples represent abstractions of particular application areas and, within each example, we will attempt to identify the properties of Concurrent METATEM that make it suitable for use in that particular area. In addition to those provided here, other 'standard' examples, such as the dining philosophers and producer/consumer problems can be defined using Concurrent METATEM.

4.1 Snow White and The Seven Dwarves — A tale of 8 objects

We will first give a 'toy' example. This not only provides a simple and appealing introduction to Concurrent METATEM, but also exhibits some of the features used later in more complex systems. This example, taken from [15], is a descendent of the 'resource controller' example used in earlier papers on METATEM and, as such, is related to a variety of resource allocation systems. However, the individual objects can be specified so that they show a form of ‘intelligent’ behaviour, and so this example system is also related to applications in Distributed AI [12]. 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 objects 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 objects exhibiting different behaviours, where an individual object’s internal rules can consist of both safety and liveness constraints, and where complex interaction can occur between autonomous objects.

The Program The Concurrent METATEM program for the scenario described above consists of the definitions of 8 objects, given below. To give a better idea of the meaning of the temporal formulae representing the internals of these objects, a brief description will be given with each object’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, upper-case alphabetic characters, such as X and Y represent universally quantified variables.

1. eager(give)[ask] :

$$\textbf{start} \Rightarrow \texttt{ask(eager)}$$
$$\circledcirc\texttt{give(eager)} \Rightarrow \texttt{ask(eager)}$$

Initially, `eager` asks for a sweet and, whenever he has just received a sweet, he asks again.

2. `mimic(ask)[ask]` :

$$\odot\texttt{ask(eager)} \Rightarrow \texttt{ask(mimic)}$$

If `eager` has just asked for a sweet then `mimic` asks for one.

3. `jealous(give)[ask]` :

$$\odot\texttt{give(eager)} \Rightarrow \texttt{ask(jealous)}$$

If `eager` has just received a sweet then `jealous` asks for one.

4. `insistent[ask]` :

$$\textbf{start} \Rightarrow \Box\texttt{ask(insistent)}$$

From the beginning of time `insistent` asks for a sweet as often as he can.

5. `courteous(ask)[ask]` :

$$\begin{bmatrix} (\neg\texttt{ask(courteous)})\,\mathcal{S}\,\texttt{ask(eager)} \wedge \\ (\neg\texttt{ask(courteous)})\,\mathcal{S}\,\texttt{ask(mimic)} \wedge \\ (\neg\texttt{ask(courteous)})\,\mathcal{S}\,\texttt{ask(jealous)} \wedge \\ (\neg\texttt{ask(courteous)})\,\mathcal{S}\,\texttt{ask(insistent)} \end{bmatrix} \Rightarrow \texttt{ask(courteous)}$$

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. `generous(give)[ask]` :

$$\begin{bmatrix} (\neg\texttt{ask(generous)})\,\mathcal{S}\,\texttt{give(eager)} \wedge \\ (\neg\texttt{ask(generous)})\,\mathcal{S}\,\texttt{give(mimic)} \wedge \\ (\neg\texttt{ask(generous)})\,\mathcal{S}\,\texttt{give(jealous)} \wedge \\ (\neg\texttt{ask(generous)})\,\mathcal{S}\,\texttt{give(insistent)} \wedge \\ (\neg\texttt{ask(generous)})\,\mathcal{S}\,\texttt{give(courteous)} \end{bmatrix} \Rightarrow \texttt{ask(generous)}$$

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. `shy(ask)[ask]` :

$$\begin{aligned} \textbf{start} &\Rightarrow \Diamond\texttt{ask(shy)} \\ \odot\texttt{ask(X)} &\Rightarrow \neg\texttt{ask(shy)} \\ \odot\texttt{ask(shy)} &\Rightarrow \Diamond\texttt{ask(shy)} \end{aligned}$$

`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. `snow-white(ask)[give]` :

$$\begin{aligned} \odot\texttt{ask(X)} &\Rightarrow \Diamond\texttt{give(X)} \\ \texttt{give(X)} \wedge \texttt{give(Y)} &\Rightarrow \texttt{X=Y} \end{aligned}$$

If `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 `give(X)` and `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 `ask()` and `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 objects, particularly those using the '$\diamondsuit$' operator to represent multiple goals.

Though this example is fairly simple, it does give some idea of how complex interacting systems can be developed using Concurrent METATEM. It also shows how useful the model is, particularly when 'intelligent' objects (agents) are considered.

We also note that, as the objects behaviour is represented explicitly, and in a logical way, the verification of properties of the system is possible. For example, given the objects' definitions, we are able to prove that every dwarf, except '`shy`' will eventually receive a sweet. (For further work on the verification of properties of such systems, see [13].)

4.2 Cooperative and Competitive Behaviour

We now extend the type of example given above to incorporate the notion of some other resource that the dwarves can attempt to exchange with Snow White for sweets, e.g. money. Note that this example is taken from [11].

The purpose of this extension of the scenario is to show how not only can the behaviours of single objects be developed, but also how more complex social structures can be represented. In particular, we outline how both cooperation and competition can be represented in Concurrent METATEM.

Bidding Initially, we will simply change the `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, `dwarf1` below asks for a sweet, bidding '2'.

$$\begin{array}{l}\texttt{dwarf1()[ask]:}\\ \quad \textbf{start} \Rightarrow \texttt{ask(dwarf1,2)}\end{array}$$

We can further modify a dwarf's behaviour so that it does not 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

$$\circledcirc[\texttt{money(N)} \wedge \texttt{N} \geq \texttt{2}] \Rightarrow \texttt{ask(dwarf1,2)}$$

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, `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 `dwarf1`, then `give(dwarf1,2)` is broadcast.

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

$$\circledcirc[\texttt{money(N)} \wedge \texttt{give(dwarf1,C)}] \Rightarrow \texttt{money(N-C)}$$

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.

$$\begin{array}{r} \textbf{start} \Rightarrow \texttt{money(100)} \wedge \texttt{paid} \\ \bigcirc[\texttt{money(N)} \wedge \bigcirc\bigcirc\bigcirc\bigcirc\texttt{paid}) \Rightarrow \texttt{money(N+100)} \wedge \texttt{paid} \end{array}$$

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

$$\begin{array}{l} \texttt{dwarf1(go)[ask]} : \\ \qquad\qquad \textbf{start} \Rightarrow \texttt{money(100)} \\ \bigcirc(\texttt{money(N)} \wedge \texttt{go}) \Rightarrow \texttt{money(N+100)} \end{array}$$

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, `dwarf2` might watch for any bids by `dwarf1` and then bid more, e.g.,

$$\bigcirc[\texttt{ask(dwarf1,B)} \wedge \texttt{myhigh(M)} \wedge \texttt{B>M}] \Rightarrow \texttt{ask(dwarf2,B+1)} \wedge \texttt{myhigh(B+1)}$$

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.

Borrowing Money Above we showed how individual dwarves might compete with each other for Snow White's sweets. Now, we will consider how dwarves might *co-operate* 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. 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 (`dwarf3` in this case) knows that the highest amount bid for a sweet so far is `X` and he only has `Y`, then he can ask to borrow `X-Y`, possibly as follows.

$$\begin{array}{l} \texttt{dwarf3(lend)[borrow,ask]} : \\ \bigcirc[\texttt{highest(X)} \wedge \texttt{money(Y)} \wedge \texttt{X>Y}] \Rightarrow \texttt{borrow(dwarf3,(X-Y)+1)} \end{array}$$

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:

$$\bigcirc[\texttt{lend(dwarf4,dwarf3,Z)} \wedge \texttt{money(Y)}] \Rightarrow \texttt{money(Y+Z)} \wedge \texttt{owe(dwarf4,Z)}$$

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 objects.

4.3 Societies of Dwarves

As described earlier, as well as the notion of autonomous objects, Concurrent METATEM also provides a larger structuring mechanism through the 'groups' extension. 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 objects.

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 objects 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 object. 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 object strategies and of grouping objects 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 compete for some global resource. Although our examples have been based upon objects 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 objects perform local interactions with each other and their environment.

4.4 Distributed Problem Solving

The next example we will look at is taken from [13], and defines simple, abstract distributed problem solving systems. We will assume that individual problem-solving objects can be implemented in Concurrent METATEM and will look at how such objects can be organised to form useful problem-solving architectures.

Hierarchical Problem Solving Once we have defined individual problem solvers, for example simple planner objects, we can construct distributed problem solving systems. The simplest approach is for one object to assign sub-plans to individual problem-solvers. This can be achieved by defining an appropriate `manager` object which knows about a variety of other problem-solving objects, can split a plan up and can assign a particular sub-problem to a specific object. This approach would involve the use of point-to-point message passing as the `manager` only wishes to pass each task on to one specific object.

Group Problem Solving While the above approach is often used in real problem-solving systems, a more dynamic and flexible architecture can be defined by utilising the facility, in Concurrent METATEM, for having multiple objects recognising the same messages. In particular, we can, in a way similar to the Contract Net approach [22], broadcast sub-problems to be solved to a set of objects. Each object might attempt to solve the problem in its own way, but the top-level object again waits until at least one solution has been returned from the set of objects. Note that the `manager` object in this case need not know exactly what other problem-solving objects exist.

We might define such a Concurrent METATEM system below.

```
manager(solution1)[problem1,solved1]:
    start ⇒ ◇problem1;
    ◉solution1 ⇒ solved1.

solvera(problem2)[solution2]:
    ◉problem2 ⇒ solution2.

solverb(problem1)[solution2]:
    ◉problem1 ⇒ ◇solution1.

solverc(problem1)[solution1]:
    ◉problem1 ⇒ ◇solution1.
```

Here, `solvera` can solve a different problem from the one `manager` poses, while `solverb` can solve the desired problem, but doesn't announce the fact (as `solution1` is not a component proposition for `solverb`); `solverc` can solve the problem posed by `manager`, and will *eventually* reply with the solution.

Cooperative Problem-Solving We now look at a refinement of the above system, where `solverc` has been removed and replaced by by two objects which together can solve `problem1`, but can not manage this individually. These objects, called `solverd` and `solvere` are defined below.

```
solverd(problem1,solution1.2)[solution1]:
 (◉solution1.2 ∧ ◆problem1) ⇒ ◇solution1.

solvere(problem1)[solution1.2]:
  ◉problem1 ⇒ ◇solution1.2.
```

Thus, when `solverd` receives the problem it cannot do anything until it has heard from `solvere`. When `solvere` receives the problem, it broadcasts the fact that it can solve part of the problem (i.e., it broadcasts `solution1.2`). When `solverd` sees this, it knows it can solve the other part of the problem and broadcasts the whole solution.

More complex problem-solving architectures can be developed in a similar way.

4.5 Process Control

Next, we will describe an example system presented in [8] which models simple rail networks and the movement of trains within them. The essential properties of this example are that each station is represented by a Concurrent METATEM object, and each object knows the identity of the next stations on its line(s). We then define a general protocol in terms of permissions for trains to enter stations and the general goal of each station to move trains on. Finally, we add initial conditions such as the initial placement of trains within the system. As the set of Concurrent METATEM objects execute, they communicate with each other in order to organise the movement of trains around the system.

The Scenario First we will give a brief outline of the problem, then we show how the elements can be abstractly modelled and prototyped using Concurrent METATEM. Here, we model, *abstractly*, the behaviour of several networks of stations within a railway system. We do not intend to describe *all* the details, but simply present executable temporal logic programs that characterise the general behaviour of the system. Consequently, the model we use will not correspond directly to a real-life transport system. It represents an abstraction of such a system exhibiting several features fundamental to the behaviour of transport systems. We will look at various configurations of lines and stations, and will use the following simplifying restrictions.

– Each station can be occupied by at most one train.
– Each train is assigned to a particular line, and can only ever travel on that line.
– Trains can only travel in one direction on a particular line, and each line has a direction associated with it.
– Networks consists only of stations, connected to each other.

Sample Network We will now show how a particular rail configuration can be represented in Concurrent METATEM by constraints within objects. This example, while being simple, represents an abstraction of the typical properties of many networks. We describe the network using the following predicates:

`station(S)` — the names of the stations.
`line(T,L)` — the line `L` of each train `T`.
`has(S,T)` — a predicate storing the name of the train `T` currently at station `S`.
`next(S1,L,S2)` — a predicate which stores, for each station `S1`, and line `L`, the next station `S2`.

Whereas, in general, predicates can change value as the execution progresses, we constrain the `station` and `next` predicates as being constant throughout time.

The particular network given here consists of two lines, one moving clockwise and one moving anti-clockwise, as shown in Figure 1. Thus, the lines join at station ‘D’,

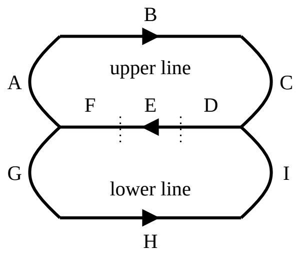

Fig. 1. A double ring network

split at station ‘F’, and use same track at station ‘E’. So, if ‘upper’ is the line running through A–B–C–D–E–F–A and ‘lower’ is the line running through G–H–I–D–E–F–G, then the definitions of `next` are as follows

```
next(A,upper,B), next(D,upper,E), next(D,lower,E), next(G,lower,H),
next(B,upper,C), next(E,upper,F), next(E,lower,F), next(H,lower,I),
next(C,upper,D), next(F,upper,A), next(F,lower,G), next(I,lower,D)
```

The initial configuration uses two trains on each line: trains ‘1’ and ‘2’ run only on the upper line, while trains ‘3’ and ‘4’ run only on the lower line. The initial placement of these four trains is train ‘1’ at station ‘C’, train ‘2’ at station ‘E’, train ‘3’ at station ‘F’, and train ‘4’ at station ‘H’.

Station ‘D’ differs from the other stations in this example in that it has *two* entry points — station ‘C’ on the upper line and station ‘I’ on the lower line. This ‘merging’ station requires more sophisticated control rules than the single-entry stations. The initial configuration for the double ring is:

```
has(train1,C),   has(train3,F),
has(train2,E),   has(train4,G)
```

The Program We will now show how the rail configurations described above together with the movement of trains between stations can be modelled in Concurrent METATEM. Each station will be represented as a separate Concurrent METATEM object and we will introduce a communication protocol which will enable stations to negotiate the passing of trains between stations. Thus, rather than having a global model of the system, the state of each station will be represented locally, with the station’s `has` and `next` constraints being private to that station. Thus, for example, the Concurrent METATEM object representing station ‘F’ in the double ring network described above would contain the following initial constraints `station(F),    has(train3)`

together with the following network details

`next(upper, A),    next(lower, G)`

Each separate station will use the same set of Concurrent METATEM rules. The only differences between stations will be their local `next` and `has` constraints. These formulae representing initial placement and network configuration will thus be partitioned amongst the appropriate objects. The rules will use the predicates defined earlier, together with the extra predicates `request()`, `permit()`, and `moved()`, whose effect can be described as follows.

`request(S,T)` — a request message which is broadcast asking for permission for train `T` to move to station `S`.
`permit(S,T)` — a message which is broadcast permitting train `T` to move to station `S`.
`moved(S,T)` — a message which is broadcast as train `T` moves to station `S`.

Thus, in order for a train to move, the station it occupies must request the 'next' station for permission to move the train on. A station can only give permission for a train to move to it if the station is currently empty and no permission has been given to another train. The interface definition for each station object is simply

```
station(request,permit,moved)[request,permit,moved].
```

showing that each station object both recognises and broadcasts instances of `request`, `permit` and `moved` messages. The internal definition of each station is represented by a set of 9 rules, described below. Note that, as usual, all upper-case letters represent universally quantified variables. Also, the variables `T1`, `T2` and `T3` represent trains, while `S1` and `S2` represent stations.

1. $\circledcirc[\texttt{has(T1)} \wedge \texttt{line(T1,L)} \wedge \texttt{next(L,S1)} \wedge \neg\texttt{moved(S1,T1)}] \Rightarrow \texttt{has(T1)}$
   If a station had a train, and the train has not moved, then the station still has that train. Note that this can be seen as a simple frame condition.
2.
$$\left[\begin{array}{c} \circledcirc[\texttt{has(T1)} \wedge \texttt{line(T1,L)} \wedge \texttt{next(L,S1)}] \\ \wedge \\ (\neg\texttt{request(S1,T1)})\mathcal{Z}(\neg\texttt{has(T1)}) \end{array}\right] \Rightarrow \texttt{request(S1,T1)}$$
   If a station had a train, and a request to move the train on has not been made during this time, then the station will make such a request to the next station on the train's line.
3. $\circledcirc[\texttt{has(T1)}\wedge\texttt{line(T1,L)}\wedge\texttt{next(L,S1)}\wedge\texttt{permit(S1,T1)}] \Rightarrow \texttt{moved(S1,T1)}$
   If a station had a train, and it has just received permission to move the train on to its next station, then the train can be moved to its next station.
4. $\circledcirc[\texttt{station(S2)} \wedge \texttt{moved(S2,T1)}] \Rightarrow \texttt{has(T1)}$
   If a train has just moved to a station, then that station has it.

5. $\odot$[station(S2) $\wedge$ moved(S2,T1)] $\Rightarrow$ $\neg$permit(S2,T2)
   If a train has just moved to a station, then that station cannot give permission to any other trains to move to that station.
6. $\odot$[station(S2) $\wedge$ request(S2,T1)] $\Rightarrow$ $\Diamond$permit(S2,T1)
   If a station receives a request for a train to be moved to the station, then eventually it will give permission for this move.
7. [station(S2) $\wedge$ permit(S2,T2) $\wedge$ permit(S2,T3)] $\Rightarrow$ T2 = T3
   A station can only ever give permission to one move at a time.
8. $\odot$[station(S2) $\wedge$ has(T1)] $\Rightarrow$ $\neg$permit(S2,T2)
   If a station has a train, then it can not give permission to any train wanting to move to the station.
9. $\odot$station(S2) $\wedge$ ($\neg$moved(S2,T1)) $\mathcal{S}$ permit(S2,T1) $\Rightarrow$ $\neg$permit(S2,T2)
   If a station has given permission for a move, but the move has not yet occurred, then the station can not give permission for any move (until the outstanding move has been completed).

Again, though the above examples are simple abstractions of railway networks, they exhibit features found in much larger, more complex rail systems. We also note that the temporal rules within each object encapsulate both the protocol for transferring trains and the goal of moving them along the line. As each station is represented by an object, it has complete control over both the station's state and the permissions granted to other stations to send trains forward.

4.6 Fault Tolerant Systems

We next consider an application which utilises the combination of broadcast message-passing and group structuring found in Concurrent METATEM. Many real-life fault-tolerant systems, such as Distributed Operating Systems [6], utilise the power of broadcast message-passing to provide duplication of important resources. One common approach is that of 'process pairs' [7]. In this section, we outline the use of process pairs for fault-tolerance, exhibiting the prower of the model of communication that we use. It will hopefully be obvious to the reader that such systems can also be developed in Concurrent METATEM.

Consider an 'essential' process in a distributed operating system, for example a network server of some kind. This can be represented as the *primary* process in Figure 2. Here, this process, if represented in Concurrent METATEM, might have an interface of the form

```
primary(r1,...,rn)[a1,...,am]
```

where r1 to rn are the requests that the process recognises, while a1 to am are the replies that it might send. The *secondary* object listens for exactly the same messages — this is possible since all requests are broadcast. Further, the secondary object *simulates* the behaviour of the primary object, with the exception that it does not broadcast any messages of its own. Thus the secondary object would have the following interface.

```
secondary(r1,...,rn)[ ]
```

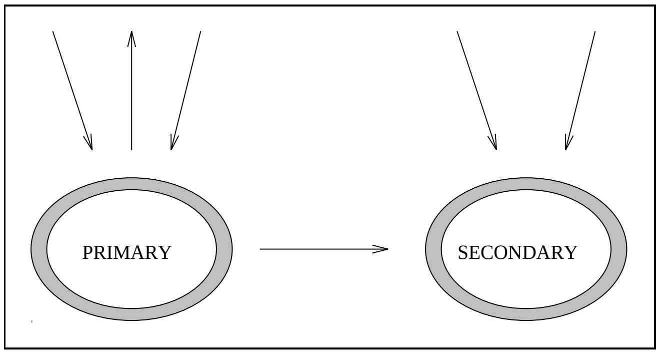

Fig. 2. A Process and its ‘shadow object’

Now, the secondary object also notes the primary object’s activity. If the primary object has not responded to a request for a specific amount of time, the secondary object assumes that the primary object’s processor has ‘crashed’ and takes over the duties associated with the service. This requires changing the object’s interface so that it now broadcasts replies. At this stage, the secondary object has become the primary one and it spawns a new copy of itself to act as the new secondary one. This final configuration is shown in Figure 3. In this way, fault-tolerance can be added ‘seamlessly’ to certain

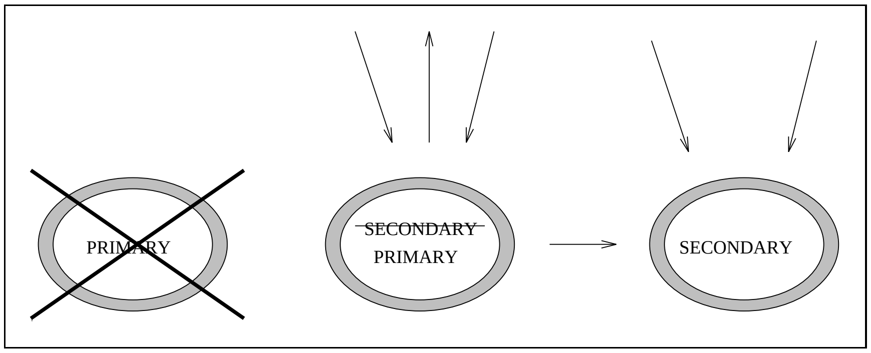

Fig. 3. Secondary object takes over.

types of distributed systems. Note that, to implement this in Concurrent METATEM we require both the ability to dynamically change an object’s interface and the ability to create new objects. It should be clear from this example, that it is the use of *broadcast* message-passing that is the main reason for the utility of such an approach.

4.7 Heterogeneous Systems

Finally, we note that the set of rules in each Concurrent METATEM object is effectively a specification of the behaviour of that object (under a particular operational interpretation). Thus, we can also view Concurrent METATEM as a framework for specifying heterogeneous systems, with each object being implemented in several possible ways. As long as each object satisfies its specification and observes the protocols regarding communication, then the *real* implementation of the object can be anything: a Cobol program, a Prolog program, a ‘real’ item of hardware, even a human.

5 Concluding Remarks

Concurrent METATEM not only provides a novel model for the simulation and implementation of reactive systems, but also incorporates executable temporal logic to implement individual objects. Consequently, such objects have explicit logical semantics and the behaviour of certain types of system is easily coded as temporal logic rules. Concurrent METATEM has potential applications in a wide range of areas, for example simulation and programming in concurrent and distributed systems, the development of distributed algorithms, distributed process control, distributed learning/problem solving, and multi-agent AI. An advantage of this approach, at least when representing certain systems from Distributed AI, is that the model follows the way in which humans communicate and cooperate.

5.1 Implementation

An implementation of (propositional) Concurrent METATEM has been developed. This program provides a platform on which simple experiments into both synchronous and asynchronous systems can be carried out. Experience with this system is directing the implementation of full (first-order) Concurrent METATEM. Future implementation work will include the development of efficient techniques for recognising and compiling both point-to-point message-passing and groups via multi-cast message-passing.

5.2 Related Work

In [16], Gehani describes Broadcasting Sequential Processes (BSP) which is also based on the asynchronous broadcasting of messages. As in Concurrent METATEM, objects may screen out certain messages, but not only is the identity of the sender always incorporated into a message, but also objects cannot manipulate their message queue as is intended in Concurrent METATEM. One, more fundamental, difference between BSP and our approach is that objects in BSP are message driven. The Linda model [17] has some similarities with this approach in that the shared data structures represented in the Linda tuple space can be seen as providing a broadcast mechanism for data. However, our computational model fixes much more than just the basic communication and distribution system. As mentioned earlier, the Concurrent METATEM model has some similarities with the Actor model, the main differences being the ability to act in a non message-driven way, the ability to process sets of incoming messages, and the ability to synchronise with other objects. Maruichi et. al. [18] use a model of computation similar to Concurrent METATEM for their investigation in DAI systems, while several distributed operating systems also use the notion of process groups (what we call 'groups') in order to group processes (objects) together [6].

Various executable temporal logics have been developed, for example [19, 1], but few have incorporated the notions of concurrency and we know of none that are based upon a computational model similar to the one described here. However, one comparable approach is Shoham's work on *Agent Oriented-Programming (AOP)* [21]. Here, individual agents are represented within a multi-modal logic, which is more concerned with the *beliefs*, *intentions* and actions of agents that we are here. Further, both the logical basis and the model of computation use in AOP are different to that considered here.

5.3 Future work

Given an object's interface definition, the internal computation method can be defined in any way that is consistent with the interface. The possibility of using a mixture of languages for the each object's internal computation will be investigated. Within the Concurrent METATEM model, both point-to-point message-passing and synchronous processes can be developed. The current Concurrent METATEM interpreter already includes the possibility of executing objects synchronously, but more support needs to be added to ensure that, if point-to-point message-passing is defined using broadcast message-passing, it remains efficient.

We also intend to look at the incorporation of dynamic object creation into the Concurrent METATEM system. On the formal side, we are developing a specification and development framework for Concurrent METATEM systems and looking at giving an algebraic semantics for Concurrent METATEM, for example based upon [20].

5.4 Acknowledgements

The author would like to thank Michael Wooldridge, Richard Owens, Howard Barringer and Marcello Finger who have collaborated on several of the papers summarised here, and Mark Reynolds who provided useful suggestions relating to the language.

References

1. M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8: 277–295, 1989.
2. G. Agha. *Actors - A Model for Concurrent Computation in Distributed Systems*. MIT Press, 1986.
3. H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A Framework for Programming in Temporal Logic. In *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Mook, Netherlands, June 1989. (Published in LNCS volume 430, Springer Verlag).

4. H. Barringer, M. Fisher, D. Gabbay, and A. Hunter. Meta-Reasoning in Executable Temporal Logic. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Cambridge, Massachusetts, April 1991.
5. H. Barringer and D. Gabbay. Executing temporal logic: Review and prospects (Extended Abstract). In *Proceedings of Concurrency '88*, 1988.
6. K. Birman. The Process Group Approach to Reliable Distributed Computing. Techanical Report TR91-1216, Department of Computer Science, Cornell University, July 1991.
7. A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault Tolerance. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, New Hampshire, October 1983. ACM. (In ACM Operating Systems Review, vol. 17, no. 5).
8. M. Finger, M. Fisher, and R. Owens. METATEM at Work: Modelling Reactive Systems Using Executable Temporal Logic. In *Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-93)*, Edinburgh, U.K., June 1993. Gordon and Breach Publishers.
9. M. Fisher and H. Barringer. Concurrent METATEM Processes — A Language for Distributed AI. In *Proceedings of the European Simulation Multiconference*, Copenhagen, June 1991.
10. 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 LNCS volume 624, Springer Verlag).
11. M. Fisher and M. Wooldridge. A Logical Approach to the Representation of Societies of Agents. In *Proceedings of Second International Workshop on Simulating Societies (SimSoc)*, Certosa di Pontignano, Siena, Italy, July 1993.
12. M. Fisher and M. Wooldridge. Executable Temporal Logic for Distributed A.I. In *Twelfth International Workshop on Distributed A.I.*, Hidden Valley Resort, Pennsylvania, May 1993.
13. M. Fisher and M. Wooldridge. Specifying and Verifying Distributed Intelligent Systems. In *Portuguese Conference on Artificial Intelligence (EPIA)*. Springer-Verlag, October 1993.
14. M. Fisher. A Normal Form for First-Order Temporal Formulae. In *Proceedings of Eleventh International Conference on Automated Deduction (CADE)*, Saratoga Springs, New York, June 1992. (Published in LNCS volume 607, Springer Verlag).
15. M. Fisher. Concurrent METATEM — A Language for Modeling Reactive Systems. In *Parallel Architectures and Languages, Europe (PARLE)*, Munich, Germany, June 1993.
16. N. Gehani. Broadcasting Sequential Processes. *IEEE Transactions on Software Engineering*, 10(4):343–351, July 1984.
17. D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in Linda. In *International Conference on Parallel Processing*, August 1985.
18. T. Maruichi, M. Ichikawa, and M. Tokoro. Modelling Autonomous Agents and their Groups. In *Decentralized AI 2 – Proceedings of the* 2^{nd} *European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW)*. Elsevier/North Holland, 1991.
19. B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, U.K., 1986.
20. K. V. S. Prasad. A Calculus of Broadcasting Systems. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, pages 338–358, Brighton, U.K., April 1991. (Published in LNCS volume 493, Springer Verlag).
21. Y. Shoham. Agent Oriented Programming. Technical Report STAN–CS–1335–90, Department of Computer Science, Stanford University, California, USA, 1990.
22. R. G. Smith and R. Davis. Frameworks for Cooperation in Distributed Problem Solving. *IEEE Transactions on Systems, Man and Cybernetics*, 11(1):61–70, 1981.

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