Towards a Semantics for Concurrent METATEM

Michael Fisher

Abstract. Concurrent METATEM is a programming language based on the notion of concurrent, communicating objects, where each object directly executes a specification given in temporal logic, and communicates with other objects using asynchronous broadcast message-passing. Thus, Concurrent METATEM represents a combination of the direct execution of temporal specifications, together with a novel model of concurrent computation. In contrast to the notions of predicates as processes and stream parallelism seen in concurrent logic languages, Concurrent METATEM represents a more coarse-grained approach, where an object consists of a set of logical rules and communication is achieved by the evaluation of certain types of predicate. Representing concurrent systems as groups of such objects provides a powerful tool for modelling complex reactive systems.
Being based upon executable temporal logic, objects in isolation have an intuitive semantics. However, the addition of both operational constraints upon the object's execution and global constraints provided by the model of concurrency and communication, complicates the overall semantics of networks of objects. It is this, more complex, semantics that we address here, where a basis for the full semantics of Concurrent METATEM is provided.

1 Introduction

A wide variety of computer systems are *reactive*. A reactive system does not simply read in a set of inputs and produce, on termination, a set of outputs [22, 16]. Reactive systems are typically concurrent or distributed, and contain elements that are constantly *reacting* to stimuli from their environment. In this paper, we provide a semantics for Concurrent METATEM, a language for representing and implementing a subclass of reactive systems [14]. This subclass contains reactive systems that are concurrent and whose elements represent self-contained entities that communicate through message-passing. These entities encapsulate both data and behaviour and, hence, can be termed *objects*.

Such reactive systems are sometimes termed *concurrent object-based systems* [28].

Concurrent METATEM has been developed from the sequential execution of temporal specifications provided by METATEM, an executable temporal logic described in [4, 11]. Thus, individual objects execute temporal specifications and communicate with their environment at certain times by *broadcasting* information.

Adopting this style of language provides the programmer/designer with a flexible way to view concurrent systems, where computation is carried out within groups of objects broadcasting, listening and executing asynchronously. Such executing objects are generally coarse-grained, with an object consisting of a set of logical rules which define constraints upon certain predicates. The logic we use within each individual object is powerful, and has been shown to be useful in describing and implementing various properties of reactive systems [20, 4].

The structure of this paper is as follows. In §2, we outline the main features of Concurrent METATEM, including the internal representation and execution of objects using temporal logic, and the interaction of sets of objects using concurrency and communication. In addition to providing a description of the basic Concurrent METATEM system, we present a range of extensions that have either been proposed or implemented. In §3, we provide a semantics for a basic Concurrent METATEM system using a *dense* temporal logic. Although Concurrent METATEM is based on the execution of a *discrete* temporal logic, we present its semantics in terms of dense temporal logic in order to realistically model the asynchronous execution of individual objects. We then show how this semantics can be refined in order to represent some of the extensions described earlier. Finally, in §4, we present our conclusions together with an outline of future work on the semantics of Concurrent METATEM.

2 Concurrent METATEM

In this section we give an outline of Concurrent METATEM. (The reader is referred to [14, 10] for more details.) We begin by motivating and describing the overall computational model, follow this by examining the internal definition and execution of objects within this model, and conclude with a brief discussion of extensions to the basic language. As the internal definition of Concurrent METATEM objects is based on temporal logic, we also include a brief overview of the logic we use, together with an outline of the execution model for temporal logic formulae provided by METATEM [4, 11].

2.1 The Computational Metaphor

The abstract 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.

The predominant approach to concurrent object-based systems is that of the *actor* model of computation [17, 1]. This model has not only been implemented directly [19], but has also been used as the basis for combining objects and concurrency in other languages, such as ABCL [27] and Concurrent Prolog [24, 18]. The metaphor used in the actor model is of a mail system, with messages being addressed and sent directly to individual actors. Thus, actor systems are based upon point-to-point message-passing and message-driven computation.

Concurrent METATEM is based on a rather different computational metaphor[1]. Although objects are concurrently executing entities communicating through message-passing, they have the following fundamental properties which contrast, for example, with the actor model.

1. The basic method of 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. An object can change its interface (i.e., the messages that it recognises) dynamically.

So, rather than seeing computation as objects sending mail messages to each other, and thus invoking some activity, computation in Concurrent METATEM can be visualised as independent objects *listening* to messages broadcast from other objects.

Although reliable broadcast message-passing may be expensive to implement (but see [6]), and secure communication may be difficult to achieve, using broadcast as the primitive message-passing mechanism provides several advantages over point-to-point message-passing. For example, this model of communication affords the opportunity for higher reliability and availability within the system [3, 2]. The use of the broadcast mechanism provides robustness in that 'duplicate' objects can be easily represented within Concurrent METATEM. These objects can either provide an alternative source for replies to messages or can be used as 'reserve' objects that can take the place of the primary object if its processor crashes. For example, replication mechanisms such as those described in [8] (based on objects which mirror and monitor the primary object in case of disruption) and [25] (based on *opportunistic recovery* from processor crashes) may be represented and implemented within Concurrent METATEM. A further advantage is that Concurrent METATEM may be used in systems where the individual objects do not necessarily know the names/addresses of other objects.

Thus, a system consists of a set of concurrently executing objects which communicate through asynchronous broadcast message-passing. Objects are the basic computational entities in a system and each object has two components:

1. an abstract *interface definition*, and,
2. an *internal definition*.

The interface definition is abstract in that the same definition can be used regardless of the object's internal definition. Although we use executable temporal logic to implement individual objects, this opens up the possibility of developing *heterogeneous* systems consisting of objects implemented in a variety of ways.

[1] Elements of this model of computation have appeared in a variety of guises, from operating systems [7] to A.I. [21], and so we do not claim that it is new — only that it is a useful computational model for executable logics.

Object Interfaces Networks of 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,full].
```

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

Given this general model for concurrent systems, we choose to represent and implement individual objects using an executable temporal logic. Thus, we next give a brief, and relatively informal, introduction to temporal logic followed, in §2.3, by an outline of the execution mechanism used for temporal formulae.

2.2 Temporal Logic

Temporal logic can be seen as classical logic extended with various modalities representing temporal aspects of logical formulae [9]. The temporal logic we use is based on a linear, discrete model of time. 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 future-time temporal operators used in this paper are as follows.

- The *sometime in the future* operator, '$\Diamond$'.
  $\Diamond\varphi$ is true now if φ is true *sometime* in the future.
- The *always in the future* operator, '$\Box$'.
  $\Box\varphi$ is true now if φ is true *always* in the future.
- The *until* operator, '$\mathcal{U}$'.
  $\varphi\mathcal{U}\,\psi$ is true now if ψ is true from now *until* a moment in the future when ψ becomes true.

Similarly, connectives are provided to enable formulae to refer to the *past*. The past-time temporal operators needed for the examples below can be described as follows.

- The *since* operator, '$\mathcal{S}$'.
  $\varphi\mathcal{S}\,\psi$ is true now if ψ was true in the past and φ was true from that moment until (but not including) the present moment.
- The *sometime in the past* operator, '$\blacklozenge$', which is the past-time analogue of '$\Diamond$'.

– The *always in the past* operator, '$\blacksquare$', which is the past-time analogue of '$\square$'.
– The *beginning of time* operator, '**start**'.
  start is only true at the beginning of time.
– The *strong last-time operator*, '$\circledcirc$'.
  $\circledcirc\varphi$ is true if there was a last moment in time and, at the moment, φ was true.

There are many other temporal operators used in temporal logics in general, and in METATEM in particular, though they will not be mentioned here, not only for clarity but also since such temporal formulae can be reduced to a normal form.

A Normal Form for Temporal Formulae Fortunately, although a range of temporal operators can be defined, we are able to transform arbitrary temporal formulae into a normal form called SNF [13] which basically consists of a set of 'rules' each of one of the following forms:

$$\textbf{start} \Rightarrow \bigvee_{j=1}^{r} m_j \qquad \text{(an } \textit{initial } \square\text{-rule)}$$

$$\circledcirc \bigwedge_{i=1}^{q} k_i \Rightarrow \bigvee_{j=1}^{r} m_j \qquad \text{(a } \textit{global } \square\text{-rule)}$$

$$\textbf{start} \Rightarrow \lozenge l \qquad \text{(an } \textit{initial } \lozenge\text{-rule)}$$

$$\circledcirc \bigwedge_{i=1}^{q} k_i \Rightarrow \lozenge l \qquad \text{(a } \textit{global } \lozenge\text{-rule)}$$

where each k_i, 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 the previous state. 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 $\lozenge$-rule is a single eventuality (i.e., '$\lozenge$' applied to a literal).

It should be noted that the use of temporal logic as the basis for the computation rules gives an extra level of expressive power over the corresponding classical logics. In particular, operators such as '$\lozenge$' give us the opportunity to specify future-time (temporal) indeterminacy. Representation in SNF allows us to capture these expressive capabilities concisely.

2.3 METATEM

METATEM is a programming and modelling language based on temporal logic. It uses a set of 'rules', couched in temporal logic, to represent an object's internal definition [4]. These rules are generally of the form [15]

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

Such rules are applied at every moment in time (i.e., at every step of the execution). In our particular case, the rules are in SNF.

In executing a set of SNF rules, the aim is to produce a model for those rules, but to do so using a *forward-chaining* process. Thus, execution in METATEM follows a cycle of checking if the left-hand side of each rule is satisfied in the current execution and, if it is, ensuring that the right-hand side is also satisfied.

The operator used to represent the basic temporal indeterminacy within METATEM is the *sometime* operator, '$\Diamond$'. When a formula such as $\Diamond\varphi$ is executed, the system must try to ensure that φ *eventually* becomes true[2]. As such eventualities might not be able to be satisfied straight away, we must keep a record of the unsatisfied eventualities, retrying them as execution proceeds [11].

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

$$\begin{aligned} \textbf{start} &\Rightarrow \texttt{popped} \\ \bullet\texttt{pop} &\Rightarrow \Diamond\texttt{popped} \\ \bullet\texttt{push} &\Rightarrow \texttt{full} \vee \texttt{popped} \end{aligned}$$

Looking at these program rules, we see that popped is made true at the beginning of time and whenever pop is true in the last moment in time, a commitment to eventually make popped true is given. Similarly, whenever push is true in the last moment in time, then either full or popped must be made true.

For a more detailed description of the METATEM execution mechanism, see [11].

2.4 Concurrent Operational Model

Given processes executing METATEM as above, we develop an operational model for their concurrent execution and communication as follows. This develops and extends METATEM into Concurrent METATEM.

The basic predicates used in the logic are categorised as follows, with several categories of predicate corresponding to messages to and from the object.

- *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', is only true if a message of the form 'push' has just been received.
- *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' is made true, where popped is a component predicate, then the message 'popped' 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.

[2] Thus, such formulae are often called eventualities.

This category of predicates may include various *primitive* operations.

Once the object has commenced execution, it continually follows a cycle of reading incoming messages, collecting together the rules that 'fire' (i.e., whose left-hand sides are satisfied by the current history), and executing one of the disjuncts represented by the conjunction of right-hand sides of 'fired' rules.

Note that, by default, the message-passing mechanism does not guarantee that the order of arrival of messages is the same as the order of sending.

2.5 Extensions

While the above provides the basic Concurrent METATEM system, we here describe some of the extensions to this that have either been proposed or implemented.

Synchronisation Though objects execute asynchronously, provision for the *synchronisation* of objects is often useful. For example, an object might broadcast a request for something, then continue executing until a reply is received. While it is waiting for the reply, the object can process other messages. Alternatively, the object might broadcast its request, then *suspend* until an appropriate reply is received. In this case, it will not process any other messages in between the request being sent and the reply being received. This can be considered as a process of synchronising the object with the reply message.

Within Concurrent METATEM, both approaches can be represented, though here we will only outline the second one where an object sends out a request, *suspends* waiting for an answer, and performs some action when an answer arrives. If we imagine that `ask` is a component predicate, while `answer` is an environment predicate, then we can represent such an object's rules as follows.

$$\begin{array}{r}
\texttt{....} \Rightarrow \texttt{ask} \\
\bigcirc\!\!\!\!\cdot\;\, \texttt{answer} \Rightarrow \texttt{do_it} \\
\bigcirc\!\!\!\!\cdot\;\, \texttt{ask} \Rightarrow \texttt{answer}
\end{array}$$

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. In this way, objects in Concurrent METATEM can synchronise with selected messages.

Groups An extension being developed at present is the provision of structuring using *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 more innovative applications [12].

We assume a group is simply represented by a set of the names of the objects that it contains. This provides us with a simple description of groups that will be used in giving their semantics (see §3.5).

Message Queues Each object in the system is associated with 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. By default, the execution of a Concurrent METATEM object is based on the set of messages received by the object since the last execution step it completed. Thus, objects consume sets of messages, rather than enforcing some linearisation on the order of arrival of messages.

However, it is sometimes useful to provide alternative message-manipulation behaviour, for example if we require that the object processes just one message at a time. Thus, Concurrent METATEM will provide mechanisms for dynamically changing this behaviour, though as yet these have not been implemented.

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,full]
```

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

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

Backtracking In general, if an object's execution mechanism is based on the execution of logical statements, then a computation may involve backtracking. In Concurrent METATEM, 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.

3 Semantics for Concurrent METATEM Systems

As one might expect, given that the language is based upon an executable temporal logic, a *temporal* semantics can be given for Concurrent METATEM. However, because both operational constraints and linguistic extensions have been added to the core language, defining the semantics for a Concurrent METATEM system is not as straightforward as it might seem. In particular, when asynchronous execution is introduced, each object effectively constructs a different temporal model. Because of this, we provide the semantics of (asynchronous) Concurrent METATEM using a *dense* temporal logic even though the individual objects each execute formulae of discrete temporal logic.

We will begin by giving a temporal semantics for one of the simplest versions of Concurrent METATEM, i.e., one that encompasses the following constraints.

1. All objects execute in synchrony
   — the execution steps of each object match exactly.
2. Messages broadcast are instantaneously received by every object
   — there is no delay in transmission.
3. Objects may not synchronise on external messages
   — environment predicates are only allowed in the left-hand side of METATEM rules.
4. There is no structuring of the object-space in to groups
   — all objects occupy the same message/object space.
5. Objects can not dynamically change their interface definitions
   — objects can neither `listen` nor `ignore`.

We will extend this semantics by successively removing each of the above constraints. Note that a semantics can be given for this basic system using a *discrete* temporal logic. However, when we remove the constraint that all objects should execute synchronously, it is more natural to give the temporal semantics using a dense temporal logic.

Object Structures In order to give a semantics for Concurrent METATEM, we assume a simple syntactic structure for representing objects in the language. For the purpose of this discussion, an object (e.g., *Obj*) is represented by a structure indexed by the following fields.

Rules — the set of rules comprising the object.
Env — the set of environment predicates.
Comp — the set of component predicates.
Int — the set of internal predicates.

Thus, for example, we can access the set of component predicates of the object *Obj* via *Obj.Comp*.

Predicate Renaming To avoid any unwanted name clashes between different objects, we rename each predicate, for example p, used in object O by a new predicate p-O. Thus, if we make predicate p true in a particular object, this will not directly affect the truth value of p in a different object. However, we also need to add communication constraints linking certain predicates together. The basic predicates p, q, etc., are used to connect the sender's name for the predicate to the receiver's name for the predicate (see below).

Semantic Definitions In order to simplify the description, we will present a semantics for propositional Concurrent METATEM. This semantics will be given using the semantic function $\mathcal{M}$, with a subscript representing the type of syntactic entity to which this function is applied. For example, $\mathcal{M}_{rule}$ gives the semantics of individual rules, while $\mathcal{M}_{system}$ gives the meaning of the whole system. The types of these semantic functions are obvious, mapping the particular syntactic structure (e.g., rule, object, system) on to a temporal logic formula.

3.1 Basic Semantics

We begin with a simple Concurrent METATEM system encompassing all the constraints described above. We give the semantics of such a system, S, consisting of the Concurrent METATEM objects $O_1, O_2, \ldots, O_n$, as follows.

$$\mathcal{M}_{system}(S) = \bigwedge_{i=1}^{n} \mathcal{M}_{object}(\mathsf{rename}(O_i))$$

$$\mathcal{M}_{object}(O) = \mathcal{M}^{O}_{rules}(O.rules) \wedge \mathsf{comms}(O)$$

where $\mathsf{rename}(O_i)$ carries out the renaming of predicates within object O_i, as described above. The predicate $comms(O)$ generates a communication constraint for messages to and from the object, which is initially defined as follows.

$$\bigwedge_{p-O \in O.env} \square(p \Rightarrow p\text{-}O) \quad \wedge \quad \bigwedge_{q-O \in O.comp} \square(q\text{-}O \Rightarrow q)$$

The intuition behind this constraint is that it ensures that when a message, corresponding to predicate p, arrives from the object's environment, not only is p made true, but also the renamed version of p used within the object is also made true. Similarly, when a component predicate is made true in the object, the appropriate message predicate must be broadcast. We will see in §3.3 that this constraint can be refined in order to represent communication delays.

Before providing the semantics of rules within objects, we will remove the first of our constraints. This allows us to introduce a dense temporal logic in order to model execution; it is in this logic that the semantics of individual rules will be presented.

3.2 Asynchronous Execution

The first constraint we remove is that of objects executing in synchrony. This allows asynchronous execution, where each object may progress at its own speed and using its own local clock. Thus, although each individual object sees time (execution) as a sequence of discrete moments (steps), the behaviour of the whole system can be represented using a dense model, where the intervals over which a particular object is active are labelled by that object's name. Consequently, associated with each object, O, is a clock, modelled by a predicate '*tick(O)*', which provides the model structure for that object. For example, in Figure 6.1, an object's state is mapped directly to the appropriate *tick* interval within the dense model provided by the logic (called TLR — see later).

In this way, we can guarantee that each object's states correspond to *tick* intervals, and can ensure that if a predicate is to be satisfied in a particular state of a particular object's execution, the predicate will be satisfied in the interval corresponding to that state.

As objects execute asynchronously, the '*tick*' intervals for various clocks are allowed to overlap. For example, consider four Concurrent METATEM objects, O_1, O_2, O_3, and O_4. The clocks associated with each particular object might overlap as shown in Figure 6.2.

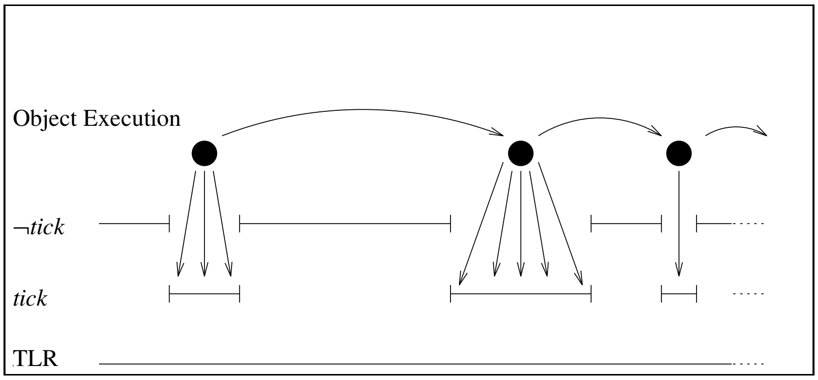

Fig. 6.1. Mapping discrete execution onto a dense model.

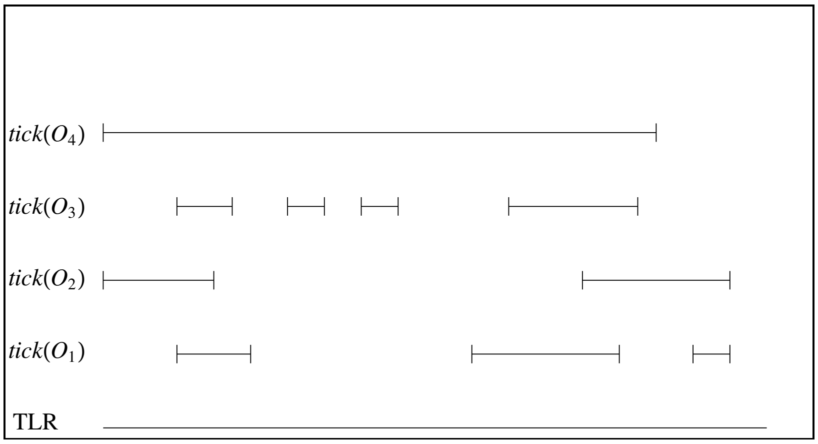

Fig. 6.2. Mapping several discrete executions onto a dense model.

In order to represent this, logically, we use a temporal logic based upon a dense model of time and overlay '*tick*' intervals for various objects upon this. The logic we use is called the 'Temporal Logic of the Reals' (TLR), which was originally developed to enable the *abstract* specification of reactive systems [5]. In dense temporal logics, there is no concept of an unique *next* state — between every two states there are an infinite number of other states. As the name suggests, TLR is based upon the domain consisting of the positive real numbers, while the logic described in §2.2 is based upon the domain of natural numbers. Though a full definition of TLR can be found in [5], we give a brief outline of the logic below.

TLR: As, in TLR, the model of time is isomorphic to the the real numbers, rather than the integers, there is no distinguished *next* moment in time. Thus, the only temporal operators within TLR refer to intervals[3]:

$A\,\mathcal{U}^{+}B$ — meaning A is true at any moment in the interval between now and the moment when B becomes true.

$C\,\mathcal{S}^{-}D$ — meaning C was true at any moment in the interval between the moment when D was true and now.

These are the two basic operators within the logic. The only other constraint added in the original TLR definition [5] that we make use of here is that of *finite variability*. The definition of finite variability states that between any two points, each predicate can only vary its value a finite number of times. This avoids the case where predicates 'toggle' values infinitely within a finite period.

As described above, we will define a new predicate '*tick*' within TLR as follows[4].

$$(\mathbf{start} \Rightarrow tick) \wedge \Box(\neg tick \Rightarrow (\neg tick)\,\mathcal{U}^{+}\,tick) \wedge \Box(tick \Rightarrow tick\,\mathcal{U}^{+}\,(\neg tick))$$

Thus, '*tick* periods' occur infinitely often, separated by '$\neg$*tick* periods'. Note that each *tick* and $\neg$*tick* interval is arbitrary, but finite, as each such interval is followed *sometime in the future* by a period in which the opposite *tick* occurs. Note that, this infinite occurrence, together with the *finite variability* condition, ensures that all the domain is 'covered' by periods of *tick*/$\neg$*tick* (i.e., an infinite number of alternations between *tick*/$\neg$*tick* is not allowed in any finite period).

We extend this definition so that '*tick*' is parameterised by the particular object to which it refers (as shown in Figure 6.2). This represents the fact that individual objects have their own clock. We will model a predicate, p, being satisfied in a state, s, by the predicate *p-Obj* being satisfied in the *tick* associated with s.

Definition of $\mathcal{M}_{rules}$ Earlier, we gave the semantics of an individual object in terms of the semantics of the rules within that object. Now we can give the semantics of individual rules as a TLR formula. This formula is then applied within each interval where the appropriate *tick* occurs. To do this, we require auxiliary predicates next-tick, last-tick and during-tick, which can be described as follows.

next-tick(O, X) — is satisfied if formula 'X' is satisfied within the next 'O' tick.

last-tick(O, X) — is satisfied if formula 'X' is satisfied within the last 'O' tick (**false** if there was no last tick).

during-tick(O, X) — is satisfied if formula 'X' is satisfied throughout the current 'O' tick.

Using these predicates, we can define the semantics of an object's set of internal rules as follows.

$$\mathcal{M}_{rules}(O) \quad = \quad \bigwedge_{R \in O.Rules} \text{next-tick}(O, \mathcal{M}^{O}_{rule}(R)) \quad \wedge \quad \text{next-tick}(O, \mathcal{M}_{rules}(O))$$

[3] Note that we have extended TLR, in the obvious way, to incorporate a finite past.

[4] We assume that both the **start** and '$\Box$' operators, are defined.

Note that $\mathcal{M}_{rules}$ is defined recursively, as we must expand the semantics of each object's set of rules at each state in the object's execution sequence.

Now, as any Concurrent METATEM rule can be transformed into one of the four different types of SNF rule [13], we need only give the semantics for each of these types of rule. These are given below, with C representing a conjunction of (non-temporal) literals, D representing a disjunction of (non-temporal) literals, and l representing a single (non-temporal) literal.

$$\mathcal{M}^{O}_{rule}(\,\mathbf{start} \;\Rightarrow\; D) = [(\neg\text{last-tick}(O, \mathbf{true})) \;\Rightarrow\; \text{during-tick}(O, D)]$$

$$\mathcal{M}^{O}_{rule}(\,\mathbf{start} \;\Rightarrow\; \Diamond l) = [(\neg\text{last-tick}(O, \mathbf{true})) \;\Rightarrow\; \Diamond(tick(O) \wedge \text{during-tick}(O, l))]$$

$$\mathcal{M}^{O}_{rule}(\,\bullet C \;\Rightarrow\; D) = [\text{last-tick}(O, C) \;\Rightarrow\; \text{during-tick}(O, D)]$$

$$\mathcal{M}^{O}_{rule}(\,\bullet C \;\Rightarrow\; \Diamond l) = [\text{last-tick}(O, C) \;\Rightarrow\; \Diamond(tick(O) \wedge \text{during-tick}(O, l))]$$

Note that, in order to ensure that predicates are checked at the appropriate moment, we add the constraint that if a predicate becomes true within a *tick* period, then it remains true until the end of that periond, i.e.,

$$(p \wedge tick(O)) \;\Rightarrow\; p\,\mathcal{U}^{+}\,\neg tick(O)$$

This ensures that, if we need to check whether a certain predicate was true during the last period, we need only check the predicate's value at the end of the period.

Synchronous Semantics Before continuing removing restrictions from the basic Concurrent METATEM system that we are modelling, we note how the synchronous version can be represented in TLR. This is relatively easy, once we have the semantics of the asynchronous version in that all we must do is to 'link' all the local clocks together. Thus, by ensuring that all the local clocks are exactly the same, we have effectively introduced a global clock enforcing synchronous execution. This linkage can simply be carried out by adding the following constraint.

$$\forall i.\ \forall j.\ \Box(tick(O_i) \Leftrightarrow tick(O_j))$$

Here, a '*tick*' within one object is exactly the same as a '*tick*' within any other.

Although we can define the semantics of synchronous execution in this way, it would obviously be preferable, if synchrony were required, to use a discrete model in the first place.

3.3 Delaying Messages

We now remove the constraint that messages are instantaneous. This provides a more natural computational model whereby messages sent are only guaranteed to arrive at *some* time in the future. Further, since we now have asynchronous execution it makes little sense to try to ensure that messages sent are received by all objects at the same time. Thus, to enforce this new model of communication, we simply replace the above definition of *comms*(O) by

$$\bigwedge_{p\text{–}O \in O.env} \Box(p \Rightarrow \Diamond p\text{-}O) \quad \wedge \quad \bigwedge_{q\text{–}O \in O.comp} \Box(q\text{-}O \Rightarrow \Diamond q)$$

Again, p-O and q-O range over the sets of environment and component predicates of object O, respectively. This constraint ensures that when a message, corresponding to predicate p, arrives from the object's environment, a commitment to make true the renamed version of p used within the object is given. Similarly, when a component predicate is made true in the object, the appropriate message predicate must eventually appear in the environment.

The only minor problem with this new definition is what to do about messages that arriving at an object while it is not executing, i.e., when *tick* is false? In order to cope with this, the above definition can be changed to

$$\bigwedge_{p\text{–}O \in O.env} \Box(p \Rightarrow \Diamond(tick(O) \wedge p\text{-}O)) \quad \wedge \quad \bigwedge_{q\text{–}O \in O.comp} \Box(q\text{-}O \Rightarrow \Diamond q)$$

thus ensuring that the received message is recorded during a *tick* of the object.

3.4 Synchronisation

If we now allow environment predicates in the right-hand sides of object rules, we can achieve synchronisation. The side-effect of synchronising objects occurs naturally so long as we ensure that objects can not define the truth value of environment predicates. Under this assumption, then if an object is waiting for an environment predicate to be satisfied, it must wait for the appropriate message to be received. This occurs naturally in the model as the current *tick*/$\neg$*tick* interval can be 'stretched' out for a finite amount of time while waiting for such a message.

In order to ensure that objects can not directly affect the truth value of an environment predicate, we simply add a constraint that says that such a predicate must be false if no new message has been received since the last message of this type was recorded (or if no such message has *ever* been received), i.e.[5]

$$[(\neg p)\,\mathcal{S}^-(tick(O) \wedge p\text{-}O) \ \vee\ \blacksquare \neg p\text{-}O] \quad \Rightarrow \quad \neg p\text{-}O$$

3.5 Structuring the Object Space

By introducing the group structuring of objects, we effectively change the communication constraint so that messages are only broadcast to members of particular groups. To give an idea of how this can be reflected in the semantics, assume that a group is simply represented as a structure containing the names of the objects within the group,

[5] Here, '$\blacksquare$', representing "always in the past", can be defined using '$\mathcal{S}^-$'.

and that there are extra *Group* and *Name* fields within the object structure. Now, consider the following communication constraint in which it is assumed that each object is a member of one group only.

$$\bigwedge_{p\text{-}O \in O.env} \square((p(G) \wedge O.Name \in G) \Rightarrow \Diamond(tick(O) \wedge p\text{-}O))$$
$$\wedge$$
$$\bigwedge_{q\text{-}O \in O.comp} \square(q\text{-}O \Rightarrow \Diamond q(O.Group))$$

Thus, a predicate broadcast includes an extra argument representing the group of the sender. Any receiver can only record such a message if it is also a member of that group.

3.6 Dynamic Interfaces

A more complex situation occurs when we wish to model the ability of an object to dynamically change its interface, for example using `listen` or `ignore`. Here, since execution of a rule might change the syntactic representation of the object itself, for example the *Env* or *Comp* fields, semantic functions such as $\mathcal{M}^{O}_{rule}$ must be redefined to return not only a TLR formula, but the updated version of the object structure that will be used for the semantics of the next execution step.

Thus, we have outlined the basic approach to the construction of a logical semantics for Concurrent METATEM. This semantics can be expanded and extended still further in order to incorporate more complex mechanisms, such as those for ensuring that predicates have particular default values (e.g., default to **false**) and for modelling the manipulation of the message queue by an object.

4 Conclusions and Future Work

While Concurrent METATEM provides a novel model for the simulation and implementation of reactive systems, its semantic clarity is slightly compromised by the operational model used. In particular, the fact that objects can execute asynchronously means that a straight-forward semantics given using a discrete temporal logic is inappropriate. However, we have shown that by utilising a dense temporal logic and overlaying this with the execution sequence for each individual object, we are able to model a range of properties of asynchronous Concurrent METATEM

While we have provided the basis for a semantics, this paper still only represents a step towards a full semantics for Concurrent METATEM. Future work will cover the more esoteric features of Concurrent METATEM (some of which have yet to be implemented), such as dynamic object creation and meta-level manipulation. Similarly, a full semantics would take account of the backtracking and commit that can occur within object execution. Further in the future, we wish to look at alternatives to temporal semantics for Concurrent METATEM. For example, we are considering using an algebraic framework which incorporates the notion of broadcast message-passing [23].

Finally, we note that TLR has been utilised in a similar way in order to provide a semantics for the MYWORLD DAI testbed [26].

References

1. G. Agha. *Actors - A Model for Concurrent Computation in Distributed Systems*. MIT Press, 1986.
2. G. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.
3. H. Bal, J. Steiner, and A. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.
4. H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A Framework for Programming in Temporal Logic. In *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Mook, Netherlands, June 1989. (Published in *Lecture Notes in Computer Science*, volume 430, Springer Verlag).
5. H. Barringer, R. Kuiper, and A. Pnueli. A Really Abstract Concurrent Model and its Temporal Logic. In *Proceedings of the Thirteenth ACM Symposium on the Principles of Programming Languages*, St. Petersberg Beach, Florida, January 1986.
6. K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.
7. K. Birman. The Process Group Approach to Reliable Distributed Computing. Techanical Report TR91-1216, Department of Computer Science, Cornell University, July 1991.
8. 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).
9. E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.
10. M. Fisher. A Survey of Concurrent METATEM — The Language and its Applications. In *First International Conference on Temporal Logic (ICTL)*, July 1994.
11. M. Fisher and R. Owens. From the Past to the Future: Executing Temporal Logic Programs. In *Proceedings of Logic Programming and Automated Reasoning*, St. Petersberg, Russia, July 1992. (Published in *Lecture Notes in Computer Science*, volume 624, Springer-Verlag).
12. 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.
13. 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 *Lecture Notes in Computer Science*, volume 607, Springer-Verlag).
14. M. Fisher. Concurrent METATEM — A Language for Modeling Reactive Systems. In *Parallel Architectures and Languages, Europe (PARLE)*, Munich, Germany, June 1993. Springer-Verlag.
15. D. Gabbay. Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of Col-*

loquium on Temporal Logic in Specification, Altrincham, U.K., 1987. (Published in *Lecture Notes in Computer Science*, volume 398, Springer-Verlag).

16. D. Harel and A. Pnueli. On the Development of Reactive Systems. Technical Report CS85-02, Department of Applied Mathematics, The Weizmann Institute of Science, Revohot, Israel, January 1985.
17. C. Hewitt. Control Structure as Patterns of Passing Messages. In P. H. Winston and R. H. Brown, editors, *Artificial Intelligence: An MIT Perspective (Volume 2)*. MIT Press, 1979.
18. K. Kahn, E. Tribble, M. Miller, and D. Bobrow. Vulcan: Logical Concurrent Objects. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.
19. H. Lieberman. Concurrent Object-Oriented Programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*. MIT Press, 1986.
20. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.
21. T. Maruichi, M. Ichikawa, and M. Tokoro. Modelling Autonomous Agents and their Groups. In Y. Demazeau and J. P. Muller, editors, *Decentralized AI 2 – Proceedings of the 2^{nd} European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW '90)*. Elsevier/North Holland, 1991.
22. A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. *Lecture Notes in Computer Science*, 224, August 1986.
23. K. V. S. Prasad. A Calculus of Broadcasting Systems. In S. Abramsky and T. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Brighton, U.K., April 1991. (Published in *Lecture Notes in Computer Science*, volume 493, Springer Verlag).
24. E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. In E. Shapiro, editor, *Concurrent Prolog–Collected Papers*, chapter 29. MIT Press, 1987.
25. R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.
26. M. Wooldridge. This is MYWORLD: The Logic of an Agent-Oriented Testbed for DAI. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents — Proceedings of the 1994 Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, 1995.
27. A. Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.
28. A. Yonezawa and M. Tokoro. Object-Oriented Concurrent Programming. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*. MIT Press, 1986.