# From the Past to the Future: Executing Temporal Logic Programs

Michael Fisher[1] and Richard Owens[2]

[1] University of Manchester, Department of Computer Science, Manchester M13 9PL, U.K.
[2] Imperial College, Department of Computing, London SW7 2BZ, U.K.

**Abstract.** We describe some of the techniques which have been used to implement METATEM, a programming language based on temporal logic, and address problems such as non-determinism and looping which place constraints on the use of temporal logic as a programming language.

## 1 Introduction

The use of temporal logic as a formal language for reasoning about the behaviour and evolution of systems is becoming widespread in the field of software engineering [12]. Recently there has been interest in extending this use by *executing* specifications written in temporal logic, initially for prototyping, and eventually for final use [2, 5].

A temporal logic specification for a system is a formula expressing conditions upon the permissible states in which the system can be, and the permissible transitions between those states. The atomic sentences of the logic each describe some aspect of the system which can change from state to state. Thus a specification for a simple torch might have propositions to express that the switch is in the 'on' position, and that the bulb is illuminated. The atomic sentences are divided up into those which describe aspects of the state which are under the control of the system (e.g. the bulb being illuminated) and those which are under the control of the environment of the system (e.g. the switch being in the 'on' position). These are known as *component* and *environment* atoms, respectively. The formula constraining the behaviour of the torch might state that for the bulb to be illuminated, the switch must have been in the 'on' position in some previous state of the system, and that the switch must have stayed on in each state of the system since then. In general the formula can move arbitrarily through the possible states of the system.

The essential notion in executing such specifications is that we can express these temporal constraints in a normal form which is a set of rules each of the form:

*condition about the past* $\Rightarrow$ *condition about the present and future*

The normal form formula (henceforth, *program*) can then be seen as a set of constraints upon the possible transitions from one state of the system to another. The program is 'run' by evaluating the conditions about the past with respect to a history of the states of the system, and where a condition evaluates to true, the corresponding condition about the present and future is used to select the next state of the system. When all the rules have been evaluated, the system is moved to the selected next state. Since this state is described in terms of atomic sentences about aspects of the system, the component atoms must be used to change the real state of the system to reflect the computed state. This paradigm is known as the *declarative past and imperative future* [8], where conditions about known states of the system are used to generate imperatives to control future states of the system. This idea has been developed into a framework for temporal logic programming, known as METATEM [2].

In this paper we describe some of the techniques which have been used to implement METATEM, and addresses particular problems which place constraints on the use of this programming paradigm. To begin, we present a brief introduction to temporal logic.

## 2 Temporal Logic

For the purposes of much of this paper, the phenomena that we shall address occur in propositional temporal logic. We shall describe the temporal logic used initially, called Propositional METATEM Logic (PML), in this section, leaving the development of first-order temporal logics until Section 5.

PML introduces two new logic connectives, 'until' ($\mathcal{U}$) and 'since' ($\mathcal{S}$), together with a number of other operators definable in terms of $\mathcal{U}$ and $\mathcal{S}$, to classical logic. The classical component of PML is used to describe states, for example

$$\neg(\text{bulb-lit} \wedge \neg\text{switch-on}).$$

The $\mathcal{U}$ and $\mathcal{S}$ connectives relate states together. $\alpha\,\mathcal{U}\,\beta$ is true in a state $s$ if the formula $\beta$ is true in a state $t$ in the future of state $s$, and the formula $\alpha$ is true in every state between states $s$ and $t$. Thus $\alpha$ will be true *until* $\beta$ is true. The $\mathcal{S}$ connective mirrors the behaviour of $\mathcal{U}$ in the past, so that $\alpha\,\mathcal{S}\,\beta$ is true in a state $s$ if the formula $\beta$ is true in a state $t$ in the past of state $s$, and the formula $\alpha$ is true in every state between states $t$ and $s$. Hence $\alpha$ was true *since* $\beta$ was true. Recall the specification for the torch example, which requires that the switch has been in the 'on' position continually since some previous state, for the bulb to be illuminated current state. We can express this by the following temporal logic rule:

$$\text{bulb-lit} \Rightarrow \text{switch-on}\,\mathcal{S}\,\text{switch-on}$$

which is read as 'if the bulb is lit now, then there must have been a state in the past when the switch was on, and the switch has been on in all the states since then.'

*Well-formed formulae* of PML ($\text{WFF}_p$) are based on a set, $\mathcal{L}_p$, of *proposition* symbols, partitioned into *environment* and *component* propositions. The set $\text{WFF}_p$ is defined as follows.

1. Any element of $\mathcal{L}_p$ is an element of $\text{WFF}_p$.
2. If $A$ and $B$ are in $\text{WFF}_p$, then so are **true**, **false**, $\neg A$, $A \vee B$, $A\,\mathcal{U}\,B$, $A\,\mathcal{S}\,B$, and $(A)$.

We define other classical connectives, such as $\wedge$, $\Rightarrow$, and $\Leftrightarrow$ in terms of the ones given above, and can also define several other useful temporal connectives in terms of $\mathcal{U}$ and $\mathcal{S}$. For example $\Diamond\alpha$, meaning that $\alpha$ is true in some future state, can be defined as $\mathbf{true}\,\mathcal{U}\,\alpha$ which can be read as 'in some future state $\alpha$ will be true, and **true** will be true in all the states until then.' Because **true** is true in all states, $\mathbf{true}\,\mathcal{U}\,\alpha$ simply requires $\alpha$ to be true in some future state. The following list gives all the connectives that we shall use in this paper:

| | | |
|---|---|---|
| $\bigcirc\varphi$ | $\varphi$ is true in the next state | $[\mathbf{false}\,\mathcal{U}\,\varphi]$ |
| $\circledcirc\varphi$ | there was a last state, and $\varphi$ was true in the last state | $[\mathbf{false}\,\mathcal{S}\,\varphi]$ |
| $\bullet\varphi$ | if there was a last state, then $\varphi$ was true in that state | $[\neg\circledcirc\neg\varphi]$ |
| $\lozenge\varphi$ | $\varphi$ will be true in some future state | $[\mathbf{true}\,\mathcal{U}\,\varphi]$ |
| $\blacklozenge\varphi$ | $\varphi$ was true in some past state | $[\mathbf{true}\,\mathcal{S}\,\varphi]$ |
| $\square\varphi$ | $\varphi$ will be true in all future states | $[\neg\lozenge\neg\varphi]$ |
| $\blacksquare\varphi$ | $\varphi$ was true in all past states | $[\neg\blacklozenge\neg\varphi]$ |

As PML formulae refer to discrete linear time-structures, which have a finite past, notice that the formula $\bullet\,\mathbf{false}$ can only be satisfied at the beginning of time and we can thus use this formula to mark that unique moment.

Temporal formulae can be classified as follows. A *state-formula* is either a literal (a proposition or a negated proposition) or a boolean combination of other state-formulae. *Strict future-time* formulae are defined as follows

- If $A$ and $B$ are either state or strict future-time, then $A\mathcal{U}B$ is strict future-time.
- If $A$ and $B$ are strict future-time, then $\neg A$, $A \vee B$, and $(A)$ are all strict future-time.

*Strict past-time* formulae are defined as the past-time duals of strict future-time formulae. *Non-strict* classes of formulae include state-formulae in their definition. Thus $\bullet a \wedge c\mathcal{S}d$ is a strict past-time formula, whereas $p \vee \lozenge q$ is a non-strict future-time formula.

The process of executing a MetateM program essentially builds a model for the PML formula which constitutes the program. The models for PML are given by a *structure*, which consists of a sequence of states, together with an *assignment* of truth values to propositions within states. More formally, a model is a pair $\mathcal{M} = (\sigma, \pi_p)$ where $\sigma$ is the ordered set of states $s_0$, $s_1$, $s_2$, $\ldots$, and $\pi_p : \mathbf{N} \times \mathcal{L}_p \mapsto \{T, F\}$ is a function assigning $T$ or $F$ to each atomic proposition at each moment in time. We use the relation '$\models$' to give the truth value of a formula in a model $\mathcal{M}$, at a particular moment in time $i$. This relation is defined for formulae of PML in Figure 1.
As an interpretation consists of a pair comprising model and time components, we say that a well-formed formula is *satisfied* in a particular model at a moment in time. Alternatively, the formula can be said to be *true* in that model at that moment. The terminology is often extended to include a well-formed formula being *satisfied in a model*. A formula is satisfied

$$
\begin{array}{ll}
\langle \mathcal{M}, i\rangle \models \mathbf{true} & \\
\langle \mathcal{M}, i\rangle \not\models \mathbf{false} & \\
\langle \mathcal{M}, i\rangle \models a & \textit{iff } \pi_p(i, a) = T, \textit{ where } a \textit{ is a proposition} \\
\langle \mathcal{M}, i\rangle \models \neg\varphi & \textit{iff } \langle \mathcal{M}, i\rangle \not\models \varphi \\
\langle \mathcal{M}, i\rangle \models \varphi \vee \psi & \textit{iff } \langle \mathcal{M}, i\rangle \models \varphi \quad \textit{or} \quad \langle \mathcal{M}, i\rangle \models \psi \\
\langle \mathcal{M}, i\rangle \models \varphi\,\mathcal{U}\,\psi & \textit{iff for some } k \textit{ such that } i < k,\ \langle \mathcal{M}, k\rangle \models \psi \textit{ and} \\
 & \qquad\qquad \textit{for all } j, \textit{ if } i < j < k \textit{ then } \langle \mathcal{M}, j\rangle \models \varphi \\
\langle \mathcal{M}, i\rangle \models \varphi\,\mathcal{S}\,\psi & \textit{iff for some } k \textit{ such that } 0 \leq k < i,\ \langle \mathcal{M}, k\rangle \models \psi \textit{ and} \\
 & \qquad\qquad \textit{for all } j, \textit{ if } k < j < i \textit{ then } \langle \mathcal{M}, j\rangle \models \varphi
\end{array}
$$

**Fig. 1.** The semantics of PML

in a model (written $\mathcal{M} \vdash \varphi$) if it is satisfied in that model at the beginning of time (i.e. $\langle \mathcal{M}, 0 \rangle \models \varphi$).

## 3 METATEM

METATEM programs are not just arbitrary PML formulae. They must be in a specific '*past implies future*' form. However, any arbitrary PML formula can be transformed into this form [2]. Thus, a METATEM program, $\mathcal{P}$, is a set of rules of the form $P_i \Rightarrow F_i$, where each $P_i$ is a strict past-time formula and each $F_i$ is a (non-strict) future-time formula. Note that most versions of METATEM follow the restriction outlined in [2], whereby references to environment propositions are not allowed in any $F_i$. Intuitively, this matches the view that a component cannot ensure that its environment will behave as it requires.

If a METATEM program, $\mathcal{P}$, is given as $\{P_i \Rightarrow F_i \mid 1 \leq i \leq n\}$, then the formula representing $\mathcal{P}$ is

$$\varphi_{\mathcal{P}} = \Box \bigwedge_{i=1}^{n} (P_i \Rightarrow F_i).$$

The execution of the program, $\mathcal{P}$, is an iterative process of labelling a model structure with the propositions true in each state, which eventually yields a model for the formula, $\varphi_{\mathcal{P}}$ (if the formula is satisfiable). The model structure produced is a sequence of states, with an identified start point. The initial model may be empty, or it may contain some information already. In either case, execution starts at the initial state 0, and steps through each state in the structure in turn. When this process is complete (at some trans-finite moment), the relationship $\langle \mathcal{M}, 0 \rangle \models \varphi_{\mathcal{P}}$ will hold.

Consider an arbitrary state, $t$, in the structure. When a rule such as $P \Rightarrow F$ is executed in state $t$, it is being made true with respect to the model $\mathcal{M}$, i.e. the relationship $\langle \mathcal{M}, t \rangle \models (P \Rightarrow F)$ is being maintained. This is achieved by performing the following steps:

1. The antecedent of the rule, $P$, is evaluated in the model. Since $P$ must be a strict past time formula, it can only describe states that have already been executed, so that $P$ can be checked against the states of model $\mathcal{M}$ which have already been labelled.
2. If $P$ is evaluated to false, then there is nothing more to be done with this rule, by virtue of the meaning of '$\Rightarrow$', because **false** $\Rightarrow \alpha$ is true regardless of $\alpha$.
3. If $P$ is evaluated to true, i.e. $\langle \mathcal{M}, t \rangle \models P$, then we must ensure that $\langle \mathcal{M}, t \rangle \models (P \Rightarrow F)$ is preserved by forcing $\langle \mathcal{M}, t \rangle \models F$.
4. Recall that $F$ must be a non-strict future-time formula, which means that the (present and future) states that it refers to have yet to be constructed. $F$ is made true by adding information to $\pi_p$ which forces $\langle \mathcal{M}, t \rangle \models F$. For example, if $F$ is $\Diamond a$, then $\pi_p$ would be extended to label a state $u$ such that $t > u$ and $\pi_p(u, a) = T$.

Note that there are limitations on the changes which can be made to $\mathcal{M}$ in order to make $F$ true. In particular, since $\mathcal{M}$ may already contain information about the future of state $t$, generated by executing an earlier state $s < t$, only *monotonic* changes may be made to $\mathcal{M}$. Thus, if the version of $\mathcal{M}$ after executing state $s$ had the relationship $\langle \mathcal{M}, v \rangle \models \alpha$ for some state $v$ and some formula, $\alpha$, then the version of $\mathcal{M}$ after executing state $t$ must also have that relationship.

When the program consists of several rules, and the interpreter is in a state where the antecedents of more than one rule evaluate to true in the model, the consequents of the

```
interpret(∅, ∅, History).

interpret(Rules, Commitments, History) ⇐
        check-past(Rules, History, Consequents),
        rewrite(Consequents, Commitments, ForceFormula),
        choose(ForceFormula, Rules, NewComponentAtoms, NewCommitments),
        extend-history(History, NewComponentAtoms, NewHistory),
        interpret(Rules, NewCommitments, NewHistory).
```

**Fig. 2.** The core of the interpreter

successful rules are conjoined to form a single formula which must be made true. Thus if the program consists of rules $P_i \Rightarrow F_i$, and $P_1, \ldots, P_n$ evaluate to true in the current model, then the formula to be made true is $F_1 \wedge \cdots \wedge F_n$.

Notice from Figure 1 that the $\mathcal{U}$ and $\mathcal{S}$ operators are non-deterministic — the interpreter therefore has a choice as to which states in $\mathcal{M}$ to label to make $F_1 \wedge \cdots \wedge F_n$ true. We can express this choice by the equivalence: $\alpha \mathcal{U} \beta \Leftrightarrow \beta \vee (\alpha \wedge \bigcirc(\alpha \mathcal{U} \beta))$ So to make $\alpha \mathcal{U} \beta$ true, the interpreter can either (i) make $\beta$ true immediately, or (ii) make $\alpha$ true in the current state, and make $\alpha \mathcal{U} \beta$ true in the next state. This latter step is done by passing a commitment $(\alpha \mathcal{U} \beta)$ from the current state to the next state to be conjoined with the consequents of the successful rules.

The formula $F_1 \wedge \cdots \wedge F_n$ can therefore be rewritten via the above equivalence and normal equivalences for $\wedge$ and $\vee$ into a disjunctive normal form $\bigvee_{i=1}^{m} \phi_i$. The interpreter must choose one of the $\phi_i$ to satisfy according to some *strategy*. The choice that the interpreter makes can critically affect whether it is able to successfully execute the program — several factors constrain the interpreter's choice: (i) The monotonicity requirement may mean that certain choices are not available because of labels already given to states by previous execution steps. (ii) Some choices would, if taken in the present state, lead to a state in which none of the $\phi_i$ in the future state could be made true. In the latter case, a form of 'backtracking' — undoing previous choices — is needed to recover. This backtracking undoes component actions and returns the execution to a previous choice point. If there are no more choice points left, the execution fails, signifying that the program is unsatisfiable.

## 4 Interpreting Propositional METATEM Programs

Here we present the essential components of a basic interpreter for temporal logic formulae, in particular propositional METATEM programs. We use a PROLOG-like pseudo code to illustrate the techniques. The core of the interpreter is a procedure which takes three arguments: a set of rules (i.e. the program) to be run, a set of states already executed (the history), and a set of commitments. This last argument is the means by which monotonic changes to the future are effected, as we shall show. The procedure performs the following steps:

1. The antecedent of each rule is checked against the history, and the consequents of those rules whose antecedents are satisfied are gathered into a set.

2. The conjunction of the consequents and commitments gives the formula whose truth must be ensured. This formula is rewritten to a normal form given by

$$\bigvee_{i=1}^{n} \bigwedge_{j_i=1}^{m_i} (p_{j_i} \wedge \neg q_{j_i}) \wedge \bigcirc \varphi_i$$

   Here, $p_{j_i}$ and $q_{j_i}$ are propositions, and $\varphi_i$ is an (optional) non-strict future-time formula.
3. One of the disjuncts must be forced true according to the strategy in use, for $i = a$, say.
4. A new state is constructed from the

$$\bigwedge_{j_a=1}^{m_a} (p_{j_a} \wedge \neg q_{j_a})$$

   and the values of the environment atoms in the new state. A new set of commitments is given by $\varphi_a$.
5. The core procedure is called recursively, with the program, the new commitments, and the old history augmented with the newly constructed state.

A simple outline of the above procedure is given in Figure 2.

The `check-past` procedure is used to ascertain which rules have past-time components that are satisfied over the given history, and `rewrite` takes the future-time components of these rules and rewrites them into the (disjunctive) normal form represented by `ForceFormula`. The `choose` procedure provides the strategy for deciding which disjunct within `ForceFormula` to execute, while the `extend-history` procedure simply extracts the new state to be built from the disjunct that has been chosen and extends the history with this state. These procedures are considered in more detail below.

### 4.1 Checking the Past

Given a set of rules, and a history, the `check-past` procedure must ascertain whether the antecedent of each rule is satisfied over the given history, according to Figure 1. The efficiency of this procedure lies in the way that the history is recorded. There are a number of methods for maintaining and querying a history which trade-off space against time complexity, depending on the requirement of the application. One such method involves the storing of propositions together with (maximal) intervals of time over which they are true [9]. So the example history above might be stored as:

$$p\ [1,3],\quad q\ [2,2],\quad r\ [1,1],\quad r\ [3,4]$$

Naturally, because the intervals are recorded as being maximal, there may be additional computation involved, particularly when intervals are merged upon the gap between them being filled. For example, should $r$ be made true in state 2 in the example above, then the $r$ [1, 1] and $r$ [3, 4] would have to be merged to become $r$ [1, 4]. The advantages of this approach are twofold — first, the space used to record the history is less (except when propositions tend to be true for intervals of unit length). Second, computation of the truth of formulae using the connectives $\mathcal{S}$ and $\mathcal{U}$ becomes just a matter of establishing the bounds of the intervals containing the arguments of the $\mathcal{S}$ and $\mathcal{U}$ connectives, and performing a simple comparison of those bounds.

### 4.2 Handling Non-determinism

We said earlier that it was necessary for the interpreter to make its choice of how to satisfy non-deterministic formulae such as $\alpha\mathcal{U}\beta$ with care, as it could restrict its scope for taking future actions to the point where the program can no longer be executed. Consider a particular example of that, given by this program:

$$\bullet\mathbf{false} \Rightarrow a\mathcal{U}b$$
$$\circledcirc a \Rightarrow \Box\neg b$$

The $\bullet$**false** is true only at the start of time, so when the interpreter starts it must force $a\mathcal{U}b$ to be true in the first state, either by making $b$ true in the second state, (so that there would be no interval between the current state and the state containing $b$, in which $a$ was true) or by making $a$ true in the second state, and also passing $a\mathcal{U}b$ to the second state as a commitment to be made true. The second rule's antecedent contains $\circledcirc a$ which requires that there be a previous state, so this antecedent cannot be true in the first state.

Suppose that the interpreter chooses the second way of making $a\mathcal{U}b$ true. In the second state the interpreter sees the following configuration:

| $a$ | $b$ | State | Commitments |
|---|---|---|---|
| ? | ? | 0 | |
| **true** | **false** | 1 | $a\mathcal{U}b$ |

At this point, the interpreter evaluates the rules' antecedents again, and discovers that there was an $a$ in the previous state, so that the second rule's consequent, $\Box\neg b$, must be made true. In other words, $b$ must be false in all future states. This is in direct conflict with the interpreter's commitment to perform $a\mathcal{U}b$ which involves making $b$ true sometime in the future. Hence the execution will never succeed without revoking this earlier choice. If however the interpreter had chosen its first course of action in the initial state, namely making $b$ true in state 1, there would have been no problem. To detect that $\Box\neg b$ and $a\mathcal{U}b$ are in conflict, the interpreter needs some algorithm. The mechanism we describe below is a combination of a particular choice strategy and a loop-checking procedure, and not only has the desired effect, but is also reasonably simple to implement.

When the system is executing, it is possible for it to end up in a 'loop'. This is where the same sequence of states is recurring along with the same unsatisfied eventualities. The example illustrates how the interpreter might end up in a 'loop', where the same sequence of states recurs along with the same unsatisfied eventualities. $b$ will be false in every state, while the commitment $a\mathcal{U}b$ will always be outstanding. Thus, it is possible for a METATEM program to be satisfiable, yet, when it is executed, led to a looping situation. (Such loops also occur when unsatisfiable programs are executed.) However, the fact that the execution is looping does not necessary signify that the execution has reached an unsatisfiable situation. The program may require that a certain situation must occur a finite number of times before a different action is taken.

Though avoiding 'bad' loops by making 'intelligent' decisions in the `choose` procedure is essential, loops generated by bad interpreter choices can also be detected by the general loop-checking procedure employed in METATEM. This checks, just before any new state is built, whether the component propositions' values and commitments particular to this state have occurred before. If such a situation has occurred for a bounded number of states, then the execution is forced to fail and backtracking occurs. The bound is derived from the upper bound for the size of the model for the program. As PML has the finite-model property, then we can extract this upper bound directly from the program's formula. Thus, we can use a coarse form of *abstract interpretation* to generate this bound which is, in the worse case, $2^{5|\varphi_{\mathcal{P}}|}$, where $|\varphi_{\mathcal{P}}|$ is the size of $\varphi_{\mathcal{P}}$ (see [2] for more details). Further work, using more sophisticated abstract interpretation techniques, is under way to refine this upper bound.

Given the above loop checking mechanism, it can be shown that the METATEM execution mechanism, when it uses the following strategy for choosing which disjunct to execute, is sound and complete with respect to the semantics of PML. The choice strategy used is to try to execute disjuncts that will satisfy the longest outstanding eventualities first. Thus, an attempt to satisfy each individual eventuality will occur as the execution proceeds. For a longer description of the correctness of propositional METATEM, see [2].

In general it is not possible to have a decidable strategy which always indicates the correct choice of disjunct to satisfy to the interpreter, although certain temporal logics (such as PML) are restricted enough for us to define such strategies. Even where such strategies exist, the upper bound on the running time of the algorithm is too high for the strategy to be of any practical use at run-time. Of greater interest is the use of techniques at compile-time to (partially) determinise the program.

If the program is implementable then there is an algorithm which will synthesise a deterministic automaton (at most double-exponential in the length of the specification) which will implement the specification. Unfortunately, this result cannot easily be extended to predicate temporal logic specifications, since the algorithm is based on the emptiness of tree automata. In [6, 11], a technique which can be extended from propositions to predicates is presented for propositional METATEM which transforms a set of rules to a deterministic set, whilst preserving satisfiability.

### 4.3 Extending the History

Once a disjunct has been selected for execution, the positive and negative propositions in the state-formula part of the disjunct, (i.e. the $\bigwedge_{j_a=1}^{m_a}(p_{j_a} \wedge \neg q_{j_a})$) are used to set the values of those propositions in the current state in the history. Not all of the component propositions need be given a value by the disjunct, in which case a default value (usually false) is given. Values to assign to the environment propositions are then read from the environment.

With each component proposition in the METATEM program is associated a description written in some programming language which causes the intended effect of the proposition. For example, if the torch program were to be implemented, then 'bulb-lit' would have an associated program which would send the appropriate control signals to the torch to light the bulb. Each time the 'bulb-lit' proposition was made true, the associated program would be run.

## 5 First-Order Temporal Logic

We introduce a first-order extension of PML, called FML. This logic is again based on discrete, linear models with finite past and infinite future.

*Well-formed formulae* of FML (WFF$_f$) are generated from the symbols of PML with the following changes: The set, $\mathcal{L}_p$, becomes a set of *predicate* symbols, partitioned into *environment* and *component* predicates. Sets of *variables* and *constants* are introduced, along with the quantifiers, $\forall$ and $\exists$. We define the set of *terms* as the set of all constants and variables. The set $\textsc{wff}_f$ is defined as $\textsc{wff}_p$ is, with the following additions.

1. If $t_1, \ldots, t_n$ are in $\mathcal{L}_t$, and $p$ is a predicate symbol of arity $n$, then $p(t_1, \ldots, t_n)$ is in $\textsc{wff}_f$.
2. If $A$ is in $\textsc{wff}_f$ and $v$ is a variable, then $\exists v.\ A$ and $\forall v.\ A$ are both in $\textsc{wff}_f$.

Classifications of temporal formulae, such as literals, past-time formulae, and future-time formulae in FML are the obvious first-order extensions of the definitions for PML.

The basic models of FML are those of PML, i.e. pairs $\langle \sigma, \pi_p \rangle$, with a domain $\mathcal{D}$ and mappings from elements of the language to denotations added. Thus the full model structure for FML is $\mathcal{M} = \langle \sigma, \mathcal{D}, \pi_c, \pi_p \rangle$ where $\pi_c$ is a map from the constants to $\mathcal{D}$, and $\pi_p$ is now a map from $\mathbf{N} \times \mathcal{L}_p$ to $\mathcal{D}^n \rightarrow \{T, F\}$. Thus, for a particular state $s$, and a particular predicate $p$ of arity $n$, $\pi(s, p)$ represents a map from $n$-tuples of elements of $\mathcal{D}$ to $T$ or $F$. Note that we use the *constant domain* assumption, i.e. that $\mathcal{D}$ is constant for every state. A *variable* assignment $V$ is a mapping from the variables to elements of $\mathcal{D}$. Given a variable assignment and the valuation function $\pi_c$, a term assignment $\tau_{v\pi}$ is a mapping from terms to $\mathcal{D}$ defined in the usual way.

The semantics of a well-formed formula is now given, not only with respect to a model structure and a particular instant in time, but also with respect to a variable assignment. The satisfaction relation, $\models$, relates such tuples to well-formed formulae as in PML, the main differences being as follows: The semantics of a predicate application is given by the truth value of the predicate application as defined in the model:

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

The semantics of the classical and temporal operators are the obvious first-order versions of those given earlier for PML, and the semantics of quantifiers are defined as follows.

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

We will usually deal with closed formulae, i.e. formulae containing no free variables. In this case, the empty mapping, [ ], is used as the initial variable assignment.

## 6 Interpretation of First-Order MetateM Programs

Propositional MetateM programs consist of sets of temporal formulae of the form $P \Rightarrow F$, where $P$ is a strict past-time temporal formula and $F$ is a (non-strict) future-time temporal formula. The set of program rules, $\{P_i \Rightarrow F_i \mid 1 \leq i \leq n\}$, represents the PML formula $\square \bigwedge_{i=1}^{n} (P_i \Rightarrow F_i)$. First-order MetateM programs are again sets of rules in the '$P \Rightarrow F$' form, but now $P$ is a (non-strict) past-time FML formula and $F$ is a (non-strict) future-time FML formula. The set of program rules, $\{P_i(\bar{x}_i, \bar{y}_i) \Rightarrow F_i(\bar{x}_i, \bar{z}_i) \mid 1 \leq i \leq n\}$, where $\bar{x}_i$, $\bar{y}_i$, and $\bar{z}_i$ are (vectors of) free variables, now represents the FML formula

$$\square \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ \exists \bar{y}_i.\ (P_i(\bar{x}_i, \bar{y}_i) \Rightarrow [\exists \bar{z}_i.\ F_i(\bar{x}_i, \bar{z}_i)]).$$

Execution of the first-order rules is in spirit the same as execution of propositional rules, but account must be taken of the bindings for variables. We still must ensure that each rule is true in each state so that, as for the propositional case where we had $\langle \mathcal{M}, t \rangle \models P \Rightarrow F$, we must maintain, for a particular rule,

$$\langle \mathcal{M}, t, [\,] \rangle \models \forall \bar{x}.\ \exists \bar{y}.\ (P(\bar{x}, \bar{y}) \ \Rightarrow\ [\exists \bar{z}.\ F(\bar{x}, \bar{z})]) .$$

So, by the semantics of $\forall$ and $\exists$ we must ensure that for all $\bar{d} \in \mathcal{D}^*$ there exists $\bar{e} \in \mathcal{D}^*$ such that

$$\langle \mathcal{M}, t, [\bar{x} \mapsto \bar{d}, \bar{y} \mapsto \bar{e}] \rangle \models\ (P(\bar{x}, \bar{y}) \ \Rightarrow\ [\exists \bar{z}.\ F(\bar{x}, \bar{z})])$$

and by further expansion of the semantics of $\Rightarrow$, this becomes for all $\bar{d} \in \mathcal{D}^*$ there exists $\bar{e} \in \mathcal{D}^*$ such that

$$\text{if } \langle \mathcal{M}, t, [\bar{x} \mapsto \bar{d}, \bar{y} \mapsto \bar{e}] \rangle \models\ P(\bar{x}, \bar{y}) \text{ then } \langle \mathcal{M}, t, [\bar{x} \mapsto \bar{d}, \bar{y} \mapsto \bar{e}] \rangle \models\ [\exists \bar{z}.\ F(\bar{x}, \bar{z})] .$$

By checking $P(\bar{x}, \bar{y})$ against the history, we can establish which values of $\bar{d}$ make the antecedent true, and thus for which values of $\bar{x}$ the consequent must be made true. Each rule therefore can generate a number of sub-formulae to be made true which we conjoin as follows:

$$\bigwedge_{j=1}^{m} \exists \bar{z}.\ F(\bar{d}_j, \bar{z})$$

and by conjoining these conjunctions for all the rules, we get a conjunction of all the formulae to be made true. As in the propositional case, this conjunction can be rewritten into a normal form of:

$$\bigvee_{k} \exists \bar{z}.\ \bigwedge_{l_k} (A_{l_k}(\bar{d}_{l_k}, \bar{z}) \wedge \neg B_{l_k}(\bar{d}_{l_k}, \bar{z})) \wedge \bigcirc \varphi_k$$

where the $A_{l_k}(\bar{d}_{l_k}, \bar{z})$ and $B_{l_k}(\bar{d}_{l_k}, \bar{z})$ are literals and $\varphi_k$ is an optional non-strict future formula. As before, one of the disjuncts $a$ is selected to be made true, and a new state is constructed from the

$$\bigwedge_{l_a} (A_{l_a}(\bar{d}_{l_a}, \bar{z}) \wedge \neg B_{l_a}(\bar{d}_{l_a}, \bar{z}))$$

Because we still have the existentially quantified variable $\bar{z}$ in the formula, a binding for $\bar{z}$ must be generated. This binding is also used for occurrences of $\bar{z}$ in the commitments $\varphi_a$. The purpose of the $\bar{z}$ is to enable values which are determined by the component predicates to be communicated throughout the disjunct to be made true.

To see how variables are used to pass values around within rules, consider the FML program:

$$\text{●}\mathrm{request}(p, m) \Rightarrow (\mathrm{resource}(m, r) \ \wedge\ \Diamond \mathrm{give}(p, r))$$

The ‘request’ predicate is an environment predicate, and records that the message $m$ has been received from process $p$. The component predicate ‘resource’ is used to discover an identifier $r$ for the resource requested in message $m$. Finally the ‘give’ component predicate hands over resource $r$ to process $p$. The $p$ and $m$ variables play the role of the $\bar{x}$ in the above explanation, and the $r$ corresponds to $\bar{z}$. In this example there is no variable playing the role of $\bar{y}$. Now suppose that we are in state 5, with the following literals true in state 4:

$$\mathrm{request}(123, \mathrm{disk1}), \quad \mathrm{request}(456, \mathrm{tape3})$$

When we evaluate the rule, we see that there are two matches for request($p, m$), given by: [$p \mapsto 123,\ m \mapsto$ disk1] and [$p \mapsto 456,\ m \mapsto$ tape3]. Thus the formula which has to be made true is:

$$\exists r.\ [\text{resource}(\text{disk1}, r)\ \wedge\ \Diamond\text{give}(123, r)] \wedge \exists r.\ [\text{resource}(\text{tape3}, r)\ \wedge\ \Diamond\text{give}(123, r)]$$

This can be rewritten via various equivalences and renaming of variables into:

$$\begin{aligned} & \exists q.\exists r.\ [\text{resource}(\text{disk1}, q)\ \wedge\ \text{give}(123, q) \wedge\ \text{resource}(\text{tape3}, r)\ \wedge\ \text{give}(456, r)] \\ \vee\ & \exists q.\exists r.\ [\text{resource}(\text{disk1}, q)\ \wedge\ \text{give}(123, q) \wedge\ \text{resource}(\text{tape3}, r)\ \wedge\ \bigcirc\Diamond\text{give}(456, r)] \\ \vee\ & \exists q.\exists r.\ [\text{resource}(\text{disk1}, q)\ \wedge\ \bigcirc\Diamond\text{give}(123, q) \wedge\ \text{resource}(\text{tape3}, r)\ \wedge\ \text{give}(456, r)] \\ \vee\ & \exists q.\exists r.\ [\text{resource}(\text{disk1}, q)\ \wedge\ \bigcirc\Diamond\text{give}(123, q) \wedge\ \text{resource}(\text{tape3}, r)\ \wedge\ \bigcirc\Diamond\text{give}(456, r)] \end{aligned}$$

Suppose that the interpreter chooses to select the second disjunct. In this case, a value must be given to $q$ so that resource(disk1, $q$) $\wedge$ give(123, $q$) is true, and a value must be given to $r$ so that both 'resource(tape3, $r$)' and '$\bigcirc\Diamond$give(456, $r$)' are true. These values are generated by the program associated with the 'resource' component predicate, and passed to the program associated with the 'give' predicate. Let us assume that the values given to $q$ and $r$ are 88 and 99 respectively. The disjunct becomes:

$$\text{resource}(\text{disk1}, 88)\ \wedge\ \text{give}(123, 88) \wedge \text{resource}(\text{tape3}, 99)\ \wedge\ \bigcirc\Diamond\text{give}(456, 99)$$

The state formula parts of this disjunct are used to extend the history, so that the component part of state 5 is given by the first three conjuncts, and the commitment $\Diamond$give(456, 99) is passed to state 6 for execution.

## 7 Implementations and Related Work

Two implementations of the imperative future paradigm described in this paper have been produced. The first is a prototype interpreter for propositional METATEM implemented in Scheme [4]. This incorporates the loop-checking and backtracking procedures described in this report. A more robust PROLOG-based interpreter for a restricted first-order version of METATEM has been used as a transaction programming language for temporal databases [3], using the maximal interval technique described in Section 4.1. Case studies using this technology are being conducted.

Several alternative approaches have been taken towards a programming language based on temporal logic, each adopting a particular philosophy. TEMPLOG [1] extends PROLOG to include temporal operators and is based on a restricted clausal form of Abadi & Manna's resolution proof system for propositional temporal logic. Rivals to TEMPLOG include TOKIO [7], which is based on intervals of time, rather than individual moments of time. TEMPLOG and TOKIO are purely declarative; their programs define sets of models which can be checked to see whether any of the models satisfy a given query. Such languages can be seen as the 'declarative past' component of the METATEM paradigm — conditions are evaluated against a model (or set of models) but the models are fixed or static, whereas in METATEM the model is dynamic, with changes to the model being reflected in changes to the 'world'.

This advantage is shared by TEMPURA [10], considered to be one of the furthest developed temporal programming languages. Each TEMPURA program is a formula of interval temporal logic, and a run of a program constructs an interval of time which satisfies the formula. The principal difference between TEMPURA and METATEM is the expressive power of the formulae which constitute the programs. TEMPURA uses an entirely deterministic language without negation to maximise the efficiency of the execution algorithm. In METATEM we make a limited trade-off between efficiency and expressiveness which affects the applications for which METATEM is appropriate. The difference between TEMPURA and METATEM is analogous to the difference between PASCAL and PROLOG: the former represents a lower-level approach amenable to efficient execution, the latter represents a higher-level approach more suited to describing symbolic computation.

**Acknowledgements** Under Basic Research Action 3096 (SPEC), ESPRIT Michael Fisher was supported by Esprit Basic Research Action 3096 (SPEC), and later by the SERC under Research Grant GR/H/18449, who also funded the early stages of the work carried out by Richard Owens under Research Grant GR/D/5939.7, with support being received from ESPRIT under Project No. 2469 (TEMPORA) for the development of the temporal transactions interpreter, jointly produced with Peter McBrien of Imperial College.

## References

1. M. Abadi and Z. Manna. Temporal Logic Programming. In *Symposium on Logic Programming*, pages 4–16. IEEE, 1987.
2. H. Barringer, M. Fisher, G. Gough, D. Gabbay, and R. Owens. METATEM: A Framework for Programming in Temporal Logic. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*. Springer-Verlag, 1990.
3. M. Finger, P. McBrien, and R. Owens. Databases and Executable Temporal Logic. In *Proceedings of the ESPRIT Conference*, November 1991.
4. M. Fisher. Implementing a Prototype METATEM Interpreter. SPEC Project Report, Department of Computer Science, University of Manchester, 1990.
5. M. Fisher and H. Barringer. Concurrent METATEM Processes — A Language for Distributed AI. In *Proceedings of the European Simulation Multiconference*, Copenhagen, Denmark, June 1991.
6. M. Fisher and P. Noel. Transformation and Synthesis in METATEM – Part I: Propositional METATEM. Technical Report, Department of Computer Science, University of Manchester, 1991.
7. M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Tokio: Logic Programming Language Based on Temporal Logic and its Compilation to Prolog. In *Logic Programming: Proceedings of the Third International Conference*, volume 225 of *LNCS*, pages 695–709. Springer-Verlag, 1986.
8. Dov Gabbay. The Declarative Past and Imperative Future. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, *Proceedings of Colloquium on Temporal Logic in Specification, Altrincham, 1987*, volume 398 of *LNCS*. Springer-Verlag, 1989.
9. D. Gabbay and P. McBrien. Temporal Logic and Historical Databases. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, Barcelona, 1991.
10. Ben Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.
11. Philippe Noel. Towards the Determinisation of Propositional Temporal Formulae. (To appear in Proceedings of LOPSTR'91).
12. Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In J. W. de Bakker, W-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency: Overview and Tutorials*, volume 224 of *LNCS*. Springer-Verlag, 1986.
13. A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the Sixteenth Symposium on the Principles of Programming Languages*. ACM, 1989.