The Knowledge Engineering Review, Vol. 11:1, 1996, 43–56

An introduction to executable temporal logics

MICHAEL FISHER

Department of Computing, Manchester Metropolitan University, Manchester M1 5GD, UK
(Email: M.Fisher@doc.mmu.ac.uk)

Abstract

In recent years a number of programming languages based upon the direct execution of temporal logic formulae have been developed. The use of such logics provides a powerful basis for the representation and implementation of a range of dynamic behaviours. Though many of these languages are still experimental, they are beginning to be applied, not only in computer science and AI, but also in less obvious areas such as user interfaces, process control and social modelling.

This article provides an introduction to some of the basic concepts of executable temporal logics, together with an overview of the main approaches being pursued.

1 Introduction

Although previously seen as esoteric, research into the direct execution of temporal logics is on the increase, with a variety of languages being developed and applied. In this article, we provide an introduction to the basic concepts involved in this area of research. We do not expect the reader to gain a deep understanding of the mechanisms for executing temporal logics from this article; our intention is simply to introduce the underlying concepts, and to provide a set of references through which further details are available. Consequently, along with an outline of the logics, we provide a brief description of some of the common execution methods used for temporal logics. For the sake of simplicity, we restrict our description to the propositional versions of the logics and their execution mechanisms.

1.1 Motivation

Let us first consider the question of why we might want to directly execute logics *at all*.

Why *execute* logical formulae?

In general, what does it mean to execute a formula, F, of logic, L? In logical terms, the execution of a formula corresponds to building a model for that formula. Thus, the model constructed, call it $\mathcal{M}$, satisfies the semantic relation

$$\mathcal{M} \models_L F$$

In the simplest case, execution can be carried out using basic theorem-proving techniques applicable to the logic L. However, if external constraints are placed upon F, then $\mathcal{M}$ must be constructed in a slightly different way (e.g. by incorporating some interaction with an environment). Thus, languages such as Prolog effectively build a model for the formula by attempting to prove the negation of a goal, interacting with the environment (for example, via `read` and `write`) in the process.

This execution of logical formulae can also be viewed in two alternative ways, as follows:

- As F represents a declarative statement, then producing $\mathcal{M}$ can be seen as execution within the declarative language L.
- As F is a specification, then constructing $\mathcal{M}$ can also be seen as prototyping an implementation of that specification.

On the question of the efficiency of direct execution, we note that although the logic might be expensive to reason with, it has been shown, for example in Prolog, that efficiency can be gained by specialising the underlying theorem-proving mechanisms. Although this may introduce some non-logical operational constraints into the execution mechanism, the language can still retain many of its desirable properties, such as its declarative, high-level descriptive capabilities. As we will see later, several executable temporal logics do, indeed, utilise extra-logical constraints in order to improve efficiency.

Why execute *temporal* logic?

Given that executing logical statements in general has some benefits, then why do we choose to execute *temporal* logic?

Perhaps the main reason is to provide a high-level language with significant expressive power to enable the representation of dynamic behaviours. Temporal logics are useful for reasoning about a changing world, and, as the temporal order of actions/events can be described using such notations, they are useful for representing dynamic systems. Consequently, temporal logics are widely used in the specification and verification of *reactive* systems (Manna and Pnueli, 1992), and in applications where the concept of time is central, such as temporal planning (Allen and Koomen, 1983), temporal knowledge representation (Allen, 1984) and temporal databases (Dean, 1987). Thus, programming languages that provide access to such temporal concepts have a wide range of applications.

Further, the logical power of the language allows us to express complex temporal properties of systems. For example, by using temporal logics, we can describe high-level properties of systems, e.g., a condition occurring *sometime* in the future (cf. liveness), or occurring *infinitely often* (cf. fairness). Similarly, if *discrete* temporal logics are used, where time is represented as a sequence of distinct *moments*, then temporal formulae can effectively represent the individual steps of an execution.

Finally, if we consider the standard approach to formal software development, we see that high-level specifications are successively *refined* into lower level specifications that are close to an intended implementation (Jones, 1986). As temporal logics are generally complex to reason with (Sistla and Clarke, 1985), then any formal specification and refinement technique is likely to encounter problems in verifying program transformations. If we are able to execute the temporal specifications directly, then this avoids the necessity of proving a large number of refinements correct (although the cost of execution itself is likely to be high).

1.2 Structure of this article

We begin our discussion, in section 2, by considering two alternative classes of temporal logic, namely *discrete* and *interval* temporal logics, both of which provide bases for execution. A range of execution options for these temporal logics are described in section 3, while, in section 4, we provide some concluding remarks and further references.

2 Temporal logics

The programming languages described in section 3 are generally based on two different forms of temporal logic, namely *discrete* temporal logics and *interval* temporal logics. Although these logics have much in common, they also have significant differences. Consequently, we (briefly) describe

each of these separately. As there has been a great deal of discussion, some would say argument, over the pros and cons of various different forms of temporal logic, we will avoid such controversy here. For readers interested in, for example, the merits of branching-time versus linear-time, and point-based versus interval-based temporal logics, papers such as (Emerson and Halpern, 1986; Van Bentham, 1988) may be consulted. In general, however none of these particular logics is 'better' than the others; each has its advantages in certain cases.

2.1 Discrete temporal logics

We present a standard propositional temporal logic (Emerson, 1990) called PTL, based on a discrete, linear model of time, with a finite past and infinite future (Gabbay et al., 1980). Within this logic, time can be visualised as a sequence of discrete moments, starting with a distinguished 'beginning of time' and extending infinitely into the future.

The language of PTL is that of classical logic extended with various modalities. This obviously has close links with modal logic (Bull and Segerberg, 1984), and can be seen as a particular form of modal logic where the reachability relation between worlds is read as an earlier/later (i.e., temporal) relation. In addition to the '$\Diamond$' and '$\Box$' operators regularly used in modal logics, PTL also incorporates the *next-time* operator, '$\bigcirc$'. In PTL, the intuitive meaning of these connectives is:

$\Diamond A$ is satisfied now if A is satisfied *sometime* in the future;
$\Box A$ is satisfied now if A is satisfied *always* in the future;
$\bigcirc A$ is satisfied now if A is satisfied at the *next* moment in time.

We can also introduce other operators, such as the *until* operator, '$\mathcal{U}$', and its weakened version, the *unless* operator, '$\mathcal{W}$'. The intuitive meaning of these binary connectives is:

$A \mathcal{U} B$ is satisfied now if B is satisfied at some future moment, and A is satisfied until then;
'$\mathcal{W}$' is a binary connective similar to '$\mathcal{U}$', allowing for the possibility that the second argument never becomes satisfied.

Intuitively, the models for PTL formulae are based on discrete, linear structures having a finite past and infinite future, i.e., sequences of the form

$$s_0, s_1, s_2, s_3, \ldots$$

where each s_i, called a *state*, provides a propositional valuation. Thus, we define a model, σ, as

$$\sigma = \langle \mathbf{N}, \pi_p \rangle$$

where

$\mathbf{N}$ is the set of Natural Numbers, used to represent the sequence $s_0, s_1, s_2, s_3, \ldots$, and,
π_p is a map from $\mathbf{N} \times \mathcal{L}_p$ (the set of all proposition symbols) to $\{\mathsf{T}, \mathsf{F}\}$, which gives a propositional valuation for each state in the sequence.

An interpretation for this logic is defined as a pair $\langle \sigma, i \rangle$, where σ is the model and i the index of the state at which the temporal statement is to be evaluated.

The semantics for well-formed temporal formulae is a relation between interpretations and formulae, and is defined inductively as follows, with the (infix) semantic relation being represented by '$\models$'. The semantics of a proposition is defined by the valuation given to it at a particular state:

$$\langle \sigma, i \rangle \models p \quad \text{iff} \quad \pi_p(i, p) = \mathsf{T} \quad [\text{where } p \in \mathcal{L}_p].$$

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

$$\langle \sigma, i \rangle \models A \vee B \quad \text{iff} \quad \langle \sigma, i \rangle \models A \quad \text{or} \quad \langle \sigma, i \rangle \models B.$$

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

$\langle \sigma, i \rangle \models \bigcirc A$ iff $\langle \sigma, i+1 \rangle \models A$
$\langle \sigma, i \rangle \models \Diamond A$ iff there exists $j \in \mathbf{N}$ such that $j \geq i$ and $\langle \sigma, j \rangle \models A$
$\langle \sigma, i \rangle \models \Box A$ iff for all $j \in \mathbf{N}$, if $j \geq i$ then $\langle \sigma, j \rangle \models A$.

Additionally, the syntax includes two binary future-time temporal operators, $\mathcal{U}$ and $\mathcal{W}$. The first of these is interpreted as follows, the second is an obvious weakening of this:

$\langle \sigma, i \rangle \models A \mathcal{U} B$ iff there exists $k \in \mathbf{N}$ such that $k \geq i$ and $\langle \sigma, k \rangle \models B$ and
for all $j \in \mathbf{N}$, if $i \leq j < k$ then $\langle \sigma, j \rangle \models A$
$\langle \sigma, i \rangle \models A \mathcal{W} B$ iff $\langle \sigma, i \rangle \models A \mathcal{U} B$ or $\langle \sigma, i \rangle \models \Box A$.

Finally, we also introduce the '**start**' operator, which is used to distinguish the unique "beginning of time", as follows:

$\langle \sigma, i \rangle \models$ **start** iff $i = 0$.

Examples of PTL formulae

To give an indication of the type of properties that can be represented in PTL, we provide the following (simple) examples:

- A simple choice of what should occur next: $rich \wedge (\bigcirc happy \vee \bigcirc poor)$
- Using the *until* operator to represent intervals of time: $nervous \mathcal{U} started$
- Using the *unless* operator: $poor \mathcal{W} lottery\text{-}win$
- Temporal indeterminacy through the *sometime* operator: $born \Rightarrow \Diamond(rich \wedge happy)$
- Something occurring infinitely often: $\Box\Diamond sunrise$
- One form of fairness: $\Box\Diamond ask \Rightarrow \Box\Diamond receive$
  The combination '$\Box\Diamond$' represents "infinitely often". Thus, the above formula effectively states that if requests are made infinitely often, then receptions will occur infinitely often.
- Another (weaker) form of fairness: $\Box\Diamond ask \Rightarrow \Diamond receive$
  In other words, if requests are made infinitely often, then something will eventually be received.
- Induction:
  $[(rich \Rightarrow happy) \wedge \Box(happy \Rightarrow \bigcirc happy)] \Rightarrow (rich \Rightarrow \Box happy)$

For further references to temporal logics, seen both from logic and computer science viewpoints, see Burgess (1984), Kröger (1987), Emerson (1990), and Manna and Pnueli (1992).

2.2 Interval temporal logic

While PTL is essentially based upon the notion of points, the interval temporal logic introduced here is concerned with the truth of statements over *intervals*. This logic is called ITL and was originally developed by Moszkowski in order to model digital circuits (Moszkowski, 1983). Its syntax contains the basic temporal operators of PTL, together with the *chop* operator, ';', which is used to concatenate intervals.

The semantics of ITL is given over a sequence, σ, as defined for PTL. However, statements are interpreted in a sub-sequence (defined by $\sigma_b, \ldots, \sigma_e$) of, rather than at a point within, σ. Thus, the semantics of ITL operators can be given as follows:

$\langle \sigma_b, \ldots, \sigma_e \rangle \models P \wedge Q$ iff $\langle \sigma_b, \ldots, \sigma_e \rangle \models P$ and $\langle \sigma_b, \ldots, \sigma_e \rangle \models Q$
$\langle \sigma_b, \ldots, \sigma_e \rangle \models P \vee Q$ iff $\langle \sigma_b, \ldots, \sigma_e \rangle \models P$ or $\langle \sigma_b, \ldots, \sigma_e \rangle \models Q$
$\langle \sigma_b, \ldots, \sigma_e \rangle \models \Box P$ iff for all i, if $b \leq i \leq e$ then $\langle \sigma_i, \ldots, \sigma_e \rangle \models P$
$\langle \sigma_b, \ldots, \sigma_e \rangle \models \bigcirc P$ iff $e > b$ and $\langle \sigma_{e+1}, \ldots, \sigma_e \rangle \models P$
$\langle \sigma_b, \ldots, \sigma_e \rangle \models P;Q$ iff there exists i, such that $b \leq i \leq e$ and both
$\langle \sigma_b, \ldots, \sigma_i \rangle \models P$ and $\langle \sigma_i, \ldots, \sigma_e \rangle \models Q$

In addition, '$\Diamond$' can be derived, this time in terms of ';', i.e.

$\Diamond P \Leftrightarrow \textbf{true};P$

meaning that there is some sub-interval in which **true** is satisfied that is followed by a sub-interval in which P is satisfied.

Although full ITL is undecidable (Moszkowski, 1983), the programming languages described in section 3.3 and section 3.4 both use a subset of ITL called *Local ITL*, where the truth of propositions does not depend upon their values at the end of an interval (Kono, 1995).

Examples of ITL formulae

Simple examples of formulae in ITL are given below:

- Property persisting throughout an interval: $\Box$*raining*
- Conditional persistence: *if manchester then* $\Box$*raining*
- Steps within intervals: *rich* $\wedge \bigcirc$*poor* $\wedge \bigcirc\bigcirc$*rich* $\wedge \bigcirc\bigcirc\bigcirc$*poor*
- Choice within intervals: *if born then* $\Box$(*rich* $\vee$ *poor*)
- Sequences of intervals: $\Box$*january;* $\Box$*february;* $\Box$*march;* . . .

For further references to interval temporal logics and their applications in such diverse areas and AI and Hardware Verification, see Allen (1983), Schwartz et al. (1983), Moszkowski and Manna (1984) and Allen and Hayes (1985).

3 Programming with temporal logics

While many researchers have investigated the execution of temporal formulae, their languages have had to overcome two problems associated particularly with discrete temporal logics, namely *complexity* and *incompleteness*:

1. Executing PTL is complex (PSPACE-complete (Sistla and Clarke, 1985)), while executing first-order temporal logic is undecidable (and *highly* complex (Merz, 1995)). Consequently, two approaches to the execution of discrete temporal logic have been followed: to restrict the logic and provide a more efficient execution mechanism for the restricted fragment; or to execute the full logic and to embed more efficient heuristics within the execution mechanism.
2. First-order discrete temporal logic is incomplete (Szalas and Holenderski, 1988; Abadi, 1989), so not every formula can be successfully executed. Again the same two approaches have been followed: to restrict the logic; or to disregard completeness and to view execution as simply an *attempt* to build a model for the formula.

Thus, as Merz (1995) suggests:

> "the design of a temporal logic based programming language involves a tradeoff between expressiveness and (efficient) implementability".

The particular approaches that we outline below exemplify two alternative responses to the above problems. In section 3.1, a restriction of PTL that follows the standard logic programming paradigm is described. Logic programming has been successfully applied in a wide variety of areas. Not only do standard logic programming languages, such as Prolog, satisfy desirable formal properties, but it has been shown that relatively efficient implementations can be produced for these languages (Aït-Kaci, 1991). Although logic programming does not utilise the full power of classical logic, it comprises a simple fragment (Horn Clauses), together with operational rules based upon standard resolution. Thus, as resolution rules for temporal logics have been developed (Abadi and Manna, 1990), it is not surprising that temporal logic programming languages have also been produced. Unfortunately, not only is the fragment of temporal logic used very restrictive (even more so than Temporal Horn Clauses), but efficient implementations are difficult to develop.

An alternative approach, outlined in section 3.2, is to ignore the problem of completeness and implement the full logic directly. In doing this it was recognised that certain temporal formulae would be expensive to execute. However, many common temporal "idioms" fit into this style of execution and can be implemented relatively efficiently.

While Interval Temporal Logics, such as ITL do not exhibit the problems of PTL (or at least not to the same degree), two similar approaches to the execution of ITL have been followed. These are outlined in sections 3.3 and 3.4.

3.1 Temporal logic programming – TEMPLOG

We now consider the extension of standard logic programming to temporal logics. We assume the reader is familiar with the concepts of logic programming, as exemplified by Prolog (Sterling and Shapiro, 1987).

As with classical logics, discrete temporal logics can be executed using the logic programming paradigm. Unfortunately, due to the incompleteness of first-order temporal logic, the Horn Clause fragment of temporal logic is still too powerful to satisfy the desirable properties of logic programming, so such a fragment must be restricted even further (Abadi and Manna, 1989; Baudinet, 1989; Abadi, 1989). Hence, the search for a subset of Temporal Horn Clauses that can be executed successfully using the logic programming paradigm.

The predominant approach to the extension of the logic programming paradigm to temporal logic is TEMPLOG (Abadi and Manna, 1989). Here. Temporal Horn Clauses, which can be categorised as either *initial* clauses (effectively if they contain **start**) or global clauses, are restricted still further using the following constraints.

1. The '$\Diamond$' operator can not occur in the head of a clause.
2. The '$\Box$' operator can only occur in the head of what are termed, *initial definite permanent* clauses, which can be characterised as

   $\Box e \leftarrow d$, **start**, c.

Essentially, these restrictions ensure that no positive occurrences of $\Diamond$-formulae appear in the program clauses. Thus, the idea here is to allow goals and bodies of program rules to consist of formulae such as p, $\bigcirc q$ and $\Diamond r$, while heads of program rules may consist of formulae such as p, $\bigcirc q$ and $\Box r$.

Given a particular goal, together with a set of rules, the goal can be successively reduced using reduction rules based upon temporal resolution (Abadi and Manna, 1990). These are the temporal analogue of the standard resolution rule and are termed *Temporal SLD* (TSLD) resolution rules (Abadi and Manna, 1990; Abadi and Manna, 1989). While we will not give a full description of the TSLD rules, we provide several, in a simplified form, below:

$$\begin{array}{lccc} \text{GOAL:} & \leftarrow \bigcirc p, G & \leftarrow \Diamond p, G & \leftarrow \Diamond p, G \\ \text{REDUCTION RULE:} & p \leftarrow B & p \leftarrow B & \bigcirc p \leftarrow B \\ \hline \text{NEW GOAL:} & \leftarrow \bigcirc B, G & \leftarrow \Diamond B, G & \leftarrow \Diamond B, G \end{array}$$

Thus, given a goal which is a next-time formulae, and a rule which allows us to reduce the predicate to which the '$\bigcirc$' operator applies, we can replace the predicate by the new body within the context of the next-time operator. If there is a goal of '$\Diamond p$', and we have a rule that enables us to reduce p to B, then we can derive the new goal '$\Diamond B$'. Finally, if we again have '$\Diamond p$' as a goal, but instead have a rule reducing '$\bigcirc p$' to B, then we can legitimately reduce the goal to '$\Diamond B$'. This is because we must still ensure that B occurs at some point in the future in order to ensure that p is satisfied in the successor state. Other TSLD rules for TEMPLOG follow a similar pattern.

As in Prolog, when there are a variety of rules that can be used to reduce a particular goal, an ordering of the rules is used. If the goal cannot be successfully reduced using the rules chosen, backtracking is employed as usual. As, in essence, TEMPLOG introduces computation over the

temporal dimension into the TSLD rules, this backtracking can sometimes be over the construction of the *temporal* sequence.

Properties of TEMPLOG

There are various technical and practical results relating to TEMPLOG which concern its expressive power, characterisation and implementation. These will be outlined below.

Baudinet (1989, 1992) provides two alternative formulations of (first-order) TEMPLOG's declarative semantics and, by relating these to TEMPLOG's operational semantics (as given by the TSLD rules of the form above), shows that these rules are complete. Further, she establishes that propositional TEMPLOG programs represent a fragment of the temporal fixpoint calculus (Banieqbal and Barringer, 1986). These important results show that, not only does TEMPLOG represent a (relatively) efficient execution mechanism for a fragment of PTL, but that, as in standard logic programming where a variety of semantic definitions coincide, both minimal model and fixpoint semantics coincide with TEMPLOG's operational semantics.

The basic TEMPLOG form of Temporal Horn Clause, together with its operational model based upon TSLD-resolution, can be characterised, and thus implemented, in a variety of different ways, as follows:

- Orgun and Wadge (1992a,b), describe a fragment of Temporal Horn Clause (called CHRONOLOG) that is expressively equivalent to TEMPLOG and that can be seen as a form of intensional logic programming. This fragment includes all the restrictions of TEMPLOG but, in addition, restricts clauses so that no '$\Diamond$' operators are allowed in the body of any clause and no '$\Box$' operators are allowed in *any* heads. CHRONOLOG is equivalent in expressive power to the restricted form of TEMPLOG called TL1 (Abadi and Manna, 1989; Baudinet, 1989).
- Brzoska (1991) describes the relationship between TEMPLOG and a form of constraint logic programming. In particular, he shows that TEMPLOG programs can be considered as $CLP(\mathcal{A})$ programs over a suitable algebra $\mathcal{A}$. This involves defining the particular algebra $\mathcal{A}$, providing a translation from Temporal Horn Clauses, and TEMPLOG clauses in particular, to elements of $\mathcal{A}$, and establishing that TSLD-resolution is a (restricted) form of the CLP-derivation mechanism over $\mathcal{A}$.
- Balbiani et al. (1991) describe the Toulouse Inference Machine (TIM), which provides a meta-language for defining extensions of logic programming based upon non-classical logics. The basic idea is to add the notion of *contexts* to standard logic programming and allow meta-rules to manipulate these. In particular, they show how the TSLD-resolution rules can be coded in the meta-language and hence how TEMPLOG can be implemented as a meta-level extension to Prolog.

3.2 Imperative future – METATEM

The idea behind temporal logic programming, and TEMPLOG in particular, is to identify a sub-set of temporal logic that can not only be effectively executed, but provides properties analogous to those found in standard logic programming. An alternative view is to attempt to execute *any* formula that can be written in temporal logic. This means that certain temporal constructs (e.g., '$\bigcirc$') are efficiently implementable, while others (e.g., '$\Diamond$') may involve more complex computation.

To achieve this, the logic programming paradigm is abandoned as it restricts the form in which logical formulae can be represented and constrains their execution. Thus, rather than using an execution mechanism based upon resolution and refutation, as in logic programming, we now look at one based upon model construction techniques related to temporal semantic tableaux (Wolper, 1985). In particular, the idea underlying the *imperative future* approach (Gabbay, 1987) is to rewrite each formula to be executed into a set of 'rules' of the form

condition on the past $\Rightarrow$ *constraint on the future*

then treat such rules as templates showing how the future can be generated given the past constructed so far. Thus, the term *imperative future* derives from the general idea that, to construct the next state in the model, we check conditions on the previous states and, if necessary, apply new constraints to the construction of the next and future states.

Initially, Gabbay developed the language USF (Gabbay, 1987), which follows this imperative future approach. USF provides a small set of temporal operators ('until', 'since' and 'sometime') and relies upon the Separation Theorem (Kamp, 1968; Gabbay, 1987) to derive executable rules of the above form. The METATEM language (Barringer et al., 1989) is a development of USF consisting of a larger range of operators, a better defined execution mechanism (Fisher and Owens, 1992) and a more practical normal form (Fisher and Noël, 1992) derived from the normal form used in clausal temporal resolution (Fisher, 1991, 1992).

In METATEM (Barringer et al., 1989; Fisher and Owens, 1992), as in the imperative future approach in general, the rules to be executed are directly used to construct a model (in the case of PTL, a sequence) for the original formula. These rules are in a normal form and are either of the form

$$\textbf{start} \Rightarrow \bigvee_j l_j$$

or

$$\bigwedge_k l_k \Rightarrow \bigcirc\left(\bigvee_l t_l \vee \bigvee_n \Diamond t_n\right)$$

where l_j, l_k, t_l and t_n are literals.

The intuition behind this form of rules is that the former (called *initial rules*) provide constraints upon the initial state, while the latter (called *global rules*) provide constraints upon both the *next* and the future states.

Examples of METATEM rules

Below are some simple examples showing some of the properties that might be represented directly as METATEM rules.

- Specifying initial conditions: $\textbf{start} \Rightarrow sad$
- Specifying initial goals (eventualities): $\textbf{start} \Rightarrow \Diamond happy$
- Introducing permanent properties: $rich \Rightarrow \bigcirc happy$
  $happy \Rightarrow \bigcirc happy$
- Defining transitions between states: $(sad \wedge \neg rich) \Rightarrow \bigcirc(sad \vee poor)$
- Introducing new goals: $(\neg resigned \wedge sad) \Rightarrow \Diamond famous$

METATEM execution

The execution of a METATEM program is an iterative process of labelling a model structure with the propositions true in each state, which would eventually yield a model for the formula (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.

Thus, if we wish to construct the initial state, the initial rules are consulted, while if we wish to construct any other state, we generate constraints on this state from the global rules whose antecedents are satisfied in the previous state. As the '$\Diamond$' operator is a non-deterministic, the execution mechanism has a choice as to which states in the model to label in order to satisfy formulae of the form $\Diamond a$ (called *eventualities*). We can express this choice by the equivalence:

$$\Diamond a \Leftrightarrow (a \vee \bigcirc\Diamond a)$$

Thus, to satisfy $\Diamond a$, the implementation can either

1. satisfy a immediately, or,
2. satisfy $\Diamond a$ in the next state.

The latter is achieved by passing a commitment '$\Diamond a$' from the current state to the next state to be conjoined with the consequents of the applicable rules.

Thus, the interpreter has a strategy not only for choosing between disjuncts, but also for choosing when to satisfy eventualities. In METATEM, the execution mechanism attempts to satisfy as many eventualities as possible. In the case of conflicting eventualities, e.g., $\Diamond a$ and $\Diamond \neg a$, the oldest outstanding eventuality is satisfied first. Any unsatisfied eventualities are passed on to the next state.

If a contradiction is generated within a state, i.e., execution of the rules has forced us to make both a proposition and its negation true in the current state, then a form of 'backtracking'—undoing previous choices—occurs. This backtracking undoes the model construction 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.

Applications

Although much of the development of METATEM has been suspended in favour of Concurrent METATEM (Fisher, 1993, 1994; Fisher and Wooldridge, 1993b), the basic METATEM language has applications in system modelling (Finger et al., 1993), databases (Finger et al., 1991) and meta-level representation and planning (Barringer et al., 1991).

3.3 Imperative interval temporal logic – TEMPURA

TEMPURA (Moszkowski, 1986; Hale and Moszkowski, 1987) is an executable temporal logic based upon the forward chaining execution of ITL, similar to that of METATEM described above. In this sense it was the precursor of the METATEM family of languages and provides a simpler and more tractable alternative to these approaches. In addition to the ITL operators outlined in section 2.2, TEMPURA incorporates various constructs, often with connotations in imperative programming, as follows:

$$
\begin{aligned}
\mathit{empty} &\equiv \neg\bigcirc\mathbf{true} \\
\mathit{fin}(p) &\equiv \Box(\mathit{empty} \Rightarrow p) \\
\mathit{if}\ C\ \mathit{the}\ P\ \mathit{else}\ Q &\equiv (C \Rightarrow P) \wedge (\neg C \Rightarrow Q) \\
\mathit{for}\ n\ \mathit{times}\ \mathit{do}\ P &\equiv \mathit{if}\ n = 0\ \mathit{then}\ \mathit{empty} \\
&\qquad \mathit{else}\ [p;\ \mathit{for}\ n - 1\ \mathit{times}\ \mathit{do}\ p]
\end{aligned}
$$

Thus, for example, an empty interval is one where there is no *next* element in the interval, while *fin* is an operator that ensures that its argument is satisfied at the end of each interval (i.e., when the interval is empty).

Many more operators can be derived from the basic ITL primitives (see Moszkowski (1986) for details). These give a wide variety of constructs for expressing properties of computations (via properties of intervals). A particular advantage of using this language, rather than those based upon, discrete temporal logics is the ability to represent both the sequencing of processes (via ';') and a form of 'real-time' (via a combination of '○' and ';').

TEMPURA execution

TEMPURA programs are written using combinations of the statements above. In general, the TEMPURA execution mechanism constructs intervals imperatively, rather than declaratively. The

basic execution consists of a transformation of the statement to be executed into a simple normal form (see below) and an iterative process of constructing the appropriate intervals.

Note, however, that TEMPURA executes a restricted version of ITL, one that is essentially deterministic. In particular, the '$\Diamond$' and '$\vee$' operators are not defined in TEMPURA. Thus, a TEMPURA program, φ, is translated (through quantifier elimination and macro expansion) to a canonical form (Hale and Moszkowski, 1987), i.e.,

$$\varphi \equiv \bigwedge_{i=0}^{n} \bigcirc^i p_i$$

where p_i is a conjunction of assignments to basic propositions. This effectively provides constraints upon the current moment, the next moment, the one after that, etc. Each state in the interval is then constructed iteratively using these, necessarily deterministic, formulae. Since the formulae are deterministic, no backtracking is required during execution.

TEMPURA examples

Simple examples of TEMPURA programs, together with the interval created when they are executed, are given below (taken from Moszkowski (1986)):

Statement: $fin(p) \wedge \bigcirc\bigcirc\bigcirc empty$
Result: an interval of length 3 at the end of which p is satisfied.

Statement: *while t do s* $\equiv$ *if t then* (*s;* [*while t do s*]) *else empty*
Result: example of the definition of new operators—the *while* construct.

Statement: *while* $(M \neq 0)$ *do* $([M \leftarrow N \bmod M] \wedge [N \leftarrow M])$
Result: N is assigned the greatest common divisor of M and N (where '$\leftarrow$' represents assignment).

Statement: *for 12 times do* [*if month(2) then fin(28) else . . .*]
Result: intervals representing months (month number 2 is February) are sequentially composed.

Applications of TEMPURA include the simulation of hardware designs (Moszkowski, 1986), the representation of multimedia (King, 1995), and parallel programming in general (Hale and Moszkowski, 1987).

3.4 Interval temporal logic programming – TOKIO

TOKIO (Fujita et al., 1986; Kono, 1995) is a logic programming language based on the extension of Prolog with ITL formulae. It provides a powerful system in which a range of applications can be implemented and verified. While TEMPURA executes a deterministic subset of ITL, TOKIO executes an extended subset incorporating the non-deterministic operators '$\Diamond$' and '$\vee$'.

TOKIO execution

Since TOKIO is an extension of Prolog, if no temporal statements are given, the execution of TOKIO reduces to that of Prolog. When ITL statements *are* present, the execution mechanism is extended to construct intervals satisfying those temporal constraints. As described in Fujita et al. (1986), *facts* are true for all intervals, whereas rules, such as

$$a \leftarrow b, c, d$$

specify that the interval a must satisfy the constraints b, c and d. In processing constraints upon intervals, the initial constraints (those satisfied 'now') are processed before the constraints relating to the next element of the interval, and so on. This iterative process continues until the construction

reaches the end of an interval (i.e. *empty* is generated). This is guaranteed as only finite intervals are allowed in TOKIO.

TOKIO examples

The following simple goals produce the output bwlow within TOKIO (Fujita et al., 1986):

Goal: $\leftarrow$ *write*(1)
Result: 1

Goal: $\leftarrow \bigcirc$*write*(2), *write*(1)
Result: 1 2

Goal: $\leftarrow$ *length*(5), $\Box$*write*(1)
Result: 1 1 1 1 1

Goal: $\leftarrow$ *length*(8), (length(5), $\Box$*write*(0); *write*(1))
Result: 0 0 0 0 0 1 1 1 1

The more complex goal below (Kono, 1995) states that either p and q must occur within five states of the interval, or the interval is six states long and s occurs in the next interval. In either case, r is true throughout the following interval:

$\leftarrow a; \Box r$
$a \leftarrow \Diamond p, \Diamond q$, *length*(5)
$a \leftarrow$ *length*(6); s

Applications of TOKIO include the prototyping of user interfaces (Johnson and Harrison, 1992), hardware simulation and verification (Kono, 1995).

4 Concluding remarks

This article has provided an introduction to the direct execution of temporal logics. In addition to outlining some of the more common execution mechanisms, we have indicated application areas together with references for further study. It should be apparent that this is a strong and expanding research area. Consequently, we expect to see both a proliferation of new techniques and the further refinement and application of the approaches outlined here.

Although there are a variety of different approaches, it is difficult to compare the languages, each having its merits in different application areas. Consequently, a purely theoretical comparison of such languages is unlikely to be useful in itself—such a study must take into account the ease of representation, the suitability of the language to the particular application area and, possibly, the efficiency of its implementation.

As this article was not intended to provide a survey of executable temporal logics, merely an introduction to some of the basic mechanisms, there are obviously languages utilising this paradigm that we have not mentioned. The interested reader is encouraged to investigate the descriptions of languages based upon logic programming provided in Gabbay (1991), Brzoska (1995), Brzoska and Schäfer (1993), Schäfer (1993), Tang (1989), Hrycej (1988, 1993) and Frühwirth (1995), and further work on the imperative view of executable temporal logics provided in Merz (1995), and Fisher (1993, 1994).

Finally, we also refer the reader to a good survey article describing a wider range of languages (Orgun and Ma, 1994), together with a more detailed account of some of the current research in this area which can be found in the proceedings of the first international workshop on executable modal and temporal logics (Fisher and Owens, 1995).

References

Abadi, M, 1989. "The power of temporal proofs" *Theoretical Computer Science* **64** 35–84.

Abadi, M and Manna, Z, 1989. "Temporal logic programming" *Journal of Symbolic Computation* **8** 277–295.

Abadi, M and Manna, Z, 1990. "Nonclausal deduction in first-order temporal logic" *ACM Journal* **37**(2) 279–317.

Aït-Kaci, H, 1991. *Warren's Abstract Machine—A Tutorial Reconstruction,* MIT Press.

Allen, J, 1983. "Maintaining knowledge about temporal intervals" *Comm. ACM* **26**(11) 832–843.

Allen, J, 1984. "Towards a general theory of action and time" *Artificial Intelligence* **23**(2) 123–154.

Allen, J and Hayes, P, 1985. "A common sense theory of time". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* 528–531, Los Angeles, CA.

Allen, J and Koomen, J, 1983. "Planning using a temporal world model". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* 741–747, Karlsruhe, Germany.

Balbiani, P, Herzig, A and Lima-Marques, M, 1991. "TIM: The Toulouse Inference Machine for non-classical logic programming" *Lecture Notes in Computer Science* 567, Springer-Verlag.

Banieqbal, B and Barringer, H, 1986. "A study of an extended temporal language and a temporal fixed point calculus" Technical Report UMCS-86-10-2, Department of Computer Science, University of Manchester.

Barringer, H, Fisher, M, Gabbay, D, Gough, G and Owens, R, 1989. "METATEM: a framework for programming in temporal logic". In: *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness,* Mook, Netherlands. (Published in *Lecture Notes in Computer Science* 430. Springer-Verlag.)

Barringer, H, Fisher, M, Gabbay, D and Hunter, A, 1991. "Meta-reasoning in executable temporal logic". In: Allen, J, Fikes, R and Sandewall, E, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)* Cambridge, MA. Morgan Kaufmann.

Baudinet, M, 1989. "Temporal logic programming is complete and expressive". In: *Proceedings of the Sixteenth ACM Symposium on the Principles of Programming Languages (POPL)* Austin, Texax. ACM Press.

Baudinet, M, 1992. "A simple proof of the completeness of temporal logic programming". In: del Cerro, LF and Penttonen, M, editors, *Intensional Logics for Programming.* Oxford University Press.

Brzoska, C, 1991. "Temporal logic programming and its relation to constraint logic programming". In: *Proceedings of International Symposium on Logic Programming (ILPS)* San Diego, CA. MIT Press.

Brzoska, C, 1995. "Temporal logic programming with metric and past operators". In: Fisher, M and Owens, R, editors, *Executable Modal and Temporal Logics: vol 897 of Lecture Notes in Artificial Intelligence.* Springer-Verlag.

Brzoska, C and Schäfer, K, 1993. "LIMETTE: Logic programming integrating metric temporal extensions—language definition and user manual". Interner Bericht 9/93, Fak für Informatik, Universität Karlsruhe.

Bull, R and Segerberg, K, 1984. "Basic modal logic". In: Gabbay, D and Guenthner, F, editors, *Handbook of Philosophical Logic (II):* Vol. 165 of *Synthese Library*, Chapter II.1, pp. 1–88. Reidel.

Burgess, J, 1984. "Basic tense logic". In: Gabbay, D and Guenthner, F, editors, *Handbook of Philosophical Logic (II):* Vol. 165 of *Synthese Library*, Chapter II.2, pp. 89–134.

Dean, T, 1987. "Large-scale temporal data bases for planning in complex domains". In: *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)* 860–866, Milan, Italy.

Emerson, E, 1990. "Temporal and modal logic". In: van Leeuwen, J, editor, *Handbook of Theoretical Computer Science* 996–1072. Elsevier.

Emerson, E and Halpern, J, 1986. " 'Sometimes' and 'Not Never' revisited: on branching versus linear time temporal logic" *Journal of the ACM* **33**(1) 151–178.

Finger, M, Fisher, M and Owens, R, 1993. "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)* Edinburgh, UK. Gordon and Breach.

Finger, M, McBrien, P and Owens, R, 1991. "Databases and executable temporal logic". In: *Proceedings of the ESPRIT Conference* Brussels, Belgium.

Fisher, M, 1991. "A resolution method for temporal logic". In: *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)* Sydney, Australia. Morgan Kaufmann.

Fisher, M, 1992. "A normal form for first-order temporal formulae". In: *Proceedings of Eleventh International Conference on Automated Deduction (CADE)* Saratoga Springs, New York. (Published in *Lecture Notes in Computer Science, 607,* Springer-Verlag.)

Fisher, M, 1993. "Concurrent METATEM—A language for modeling reactive systems". In: *Parallel Architectures and Languages, Europe (PARLE)* Munich, Germany. (Published as *Lecture Notes in Computer Science, 694,* Springer-Verlag.)

Fisher, M, 1994. "A survey of concurrent METATEM—The language and its applications". In: *First International Conference on Temporal Logic (ICTL)* Bonn, Germany. (Published in *Lecture Notes in Computer Science, 827*, Springer-Verlag.)

Fisher, M and Noël, P, 1992. "Transformation and synthesis in METATEM—Part I: Propositional METATEM. Technical Report UMCS-92-2-1, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK.

Fisher, M and Owens, R, 1992. "From the past to the future: executing temporal logic programs". In: *Proceedings of Logic Programming and Automated Reasoning (LPAR)* St. Petersberg, Russia. (Published in *Lecture Notes in Computer Science, 624*, Springer-Verlag.)

Fisher, M and Owens, R, editors, 1995. *Executable Modal and Temporal Logics:* vol. *897* of *Lecture Notes in Artificial Intelligence.* Springer-Verlag.

Fisher, M and Wooldridge, M, 1993b. "Executable temporal logic for distributed A.I." In: *Twelfth International Workshop on Distributed A.I.* Hidden Valley Resort, Pennsylvania.

Frühwirth, T, 1995. "Temporal logic and annotated constraint logic programming". In: Fisher, M and Owens, R, editors, *Executable Modal and Temporal Logics,* vol. *897* of *Lecture Notes in Artificial Intelligence.* Springer:Verlag.

Fujita, M, Kono, S, Tanaka, T and Moto-oka, T, 1986. "Tokio: Logic programming language based on temporal logic and its compilation into Prolog". In: *3rd International Conference on Logic Programming* London, UK. (Published in *Lecture Notes in Computer Science, 225*, Springer-Verlag.)

Gabbay, D, 1987. "Declarative past and imperative future: executable temporal logic for interactive systems". In: Banieqbal, B, Barringer, H and Pnueli, A, editors, *Proceedings of Colloquium on Temporal Logic in Specification* 402–450, Altrincham, UK. (Published in *Lecture Notes in Computer Science, 398*, Springer-Verlag.)

Gabbay, D, 1991. "Modal and temporal logic II (a temporal Prolog machine)". In: Dodd, T, Owens, R and Torrance, S, editors, *Logic Programming—Expanding the Horizon.* Intellect Books.

Gabbay, D, Pnueli, A, Shelah, S and Stavi, J, 1980. "The temporal analysis of fairness". In: *Proceedings of the Seventh ACM Symposium on the Principles of Programming Languages (POPL)* 163–173, Las Vegas, NV. ACM Press.

Hale, R and Moszkowski, B, 1987. "Parallel programming in temporal logic". In: *Parallel Architectures and Languages Europe (PARLE)* Eindhoven, The Netherlands. (Published as *Lecture Notes in Computer Science, 259*, Springer-Verlag.)

Hrycej, T, 1988. "Temporal Prolog". In: Kodratoff, Y, editor, *Proceedings of the European Conference on Artificial Intelligence* 296–301. Pitman.

Hrycej, T, 1993. "A temporal extension of Prolog" *Journal of Logic Programming* **15**(1 & 2) 113–145.

Johnson, CW and Harrison, MD, 1992. "Using temporal logic to support the specification and prototyping of interactive control systems" *International Journal of Man-Machine Studies* **36**.

Jones, CB, 1986. *Systematic Software Development Using VDM* Prentice Hall.

Kamp, JAW, 1968. *Tense Logic and the Theory of Linear Order* PhD thesis, University of California.

King, P, 1995. "Towards an ITL based formalism for expressing temporal constraints in multimedia documents". In: *Proceedings of IJCAI Workshop on Executable Temporal Logics* Montreal, Canada.

Kono, S, 1995. "A combination of clausal and non-clausal temporal logic programs". In: Fisher, M and Owens, R, editors, *Executable Modal and Temporal Logics,* vol *897* of *Lecture Notes in Artificial Intelligence.* Springer-Verlag.

Kröger, F, 1987. *Temporal Logic of Programs* Monographs on Theoretical Computer Science. Springer-Verlag.

Manna, Z and Pnueli, A, 1992. *The Temporal Logic of Reactive and Concurrent Systems: Specification* Springer-Verlag.

Merz, S, 1995. "Efficiently executable temporal logic programs". In: Fisher, M and Owens, R, editors, *Executable Modal and Temporal Logics, vol 897* of *Lecture Notes in Artificial Intelligence.* Springer-Verlag.

Moszkowski, B, 1983. "Reasoning about digital circuits" Technical report, Stanford University.

Moszkowski, B, 1986. *Executing Temporal Logic Programs* Cambridge University Press.

Moszkowski, B and Manna, Z, 1984. "Reasoning in interval temporal logic". In: *AMC/NSF/ONR Workshop on Logics of Programs* Berlin, Germany. (Published as *Lecture Notes in Computer Science, 164*, Springer-Verlag.)

Orgun, M and Ma, W, 1994. "An overview of temporal and modal logic programming". In: *First International Conference on Temporal Logic (ICTL)* Bonn, Germany. (Published in *Lecture Notes in Computer Science, 827*, Springer-Verlag.)

Orgun, M and Wadge, W, 1992a. "Theory and practice of temporal logic programming". In: del Cerro, LF and Penttonen, M, editors, *Intensional Logics for Programming* Oxford University Press.

Orgun, MA and Wadge, W, 1992b. "Towards a unified theory of intensional logic programming". *Journal of Logic Programming* **13**(1, 2, 3 and 4) 413–440.

Schäfer, K, 1993. "Entwicklung einer temporal logischen Sprache zur Beschreibung von Abläufen in Straßenverkehrsszenen" Diplomarbeit, Universität Karlsruhe, Inst. für Logik, Komplexität und Deduktionssysteme.

Schwartz, R, Melliar-Smith, P and Vogt, F, 1983. "An interval-based temporal logic" *Lecture Notes in Computer Science* **164** 443–457.

Sistla, A and Clarke, E, 1985. "Complexity of propositional linear temporal logics" *ACM Journal* **32**(3) 733–749.

Sterling, L and Shapiro, E, 1987. *The Art of Prolog* MIT Press.

Szalas, A and Holenderski, L, 1988. "Incompleteness of first-order temporal logic with until" *Theoretical Computer Science* **57** 317–325.

Tang, T, 1989. "Temporal Logic CTL + Prolog" *Journal of Automated Reasoning* **5** 49–65.

Van Bentham, J, 1988. "A logician's point of view concerning the use of temporal logic in computer science". In: *REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*. (Published as *Lecture Notes in Computer Science, 354*, Springer-Verlag.)

Wolper, P, 1985. "The tableau method for temporal logic: an overview" *Logique et Analyse* 110–111, 119–136.