An Introduction to Executable Modal and Temporal Logics

Michael Fisher & Richard Owens

Abstract. In recent years a number of programming languages based upon the direct execution of either modal or temporal logic formulae have been developed. This use of non-classical 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 process control and social modelling.
This paper provides an introduction to some of the basic concepts of executable modal and temporal logics.

1 Introduction

In this paper we provide an introduction to the basic concepts involved in the direct execution of modal and temporal logics. Although previously seen as esoteric, this area of research is beginning to expand, with a variety of languages being developed and applied. We do not expect the reader to gain a deep understanding of the mechanisms for executing modal and temporal logics from this paper; our intention is simply to introduce the underlying concepts, and to provide a set of references that can be followed if desired. Consequently, along with an outline of the logics and some of the common execution methods used, we provide a brief survey of implemented languages based upon the execution of modal or 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

First, we 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 relation

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

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

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

Although the logic might be expensive to reason in, it has been shown, for example in Prolog, that efficiency can be gained by specialising the theorem-proving mechanism. Although this may introduce some non-logical operational constraints into the execution mechanism, the language still retains its desirable properties, such as its declarative, high-level descriptive capabilities.

Why Execute Modal Logic? Many problems, particularly in knowledge representation, involve the division of the problem space into a number of sub-spaces. For example, in representing problems involving distributed reasoning, the total knowledge of the system might be the sum of the knowledge of its components. Alternatively knowledge might only be propagated through the relationships between the sub-spaces — for example local groups of components may share knowledge which is not made available to remote components.

Classical first-order logic expresses relationships between terms representing members of a domain. The truth values of statements in the logic are derived with respect to a particular interpretation of the symbols of the logic to members of some domain. Because there is a single domain, and one interpretation into that domain, classical logic is restricted to statements about a *global* problem space.

This is a strength because, since the problem space can be arbitrarily complex, classical first-order logic can express any first-order condition over arbitrarily complex problem spaces. However, it is also a weakness, as there are no language constructs for handling structural complexity within the problem space.

While the universal nature of classical first-order logic allows such problems to be specified, statements about the general underlying problem space must be incorporated into statements relating to the particular problem being considered. For example, we might state that "for all members m of the domain, if m is an agent, and m is connected to a particular agent a, then if a knows that P is true, m knows that P is true". It is far more preferable if we state that (in the context of agent a) "if P is known to be true then all connected agents know that P is true".

Any theorem-proving or execution in such structured problem spaces would ideally have knowledge of the structure, and exploit the structure to optimise the execution.

Without a means to make such structural optimisations, much effort is spent in repeatedly proving axioms about the structure, which are already known.

Modal logics offer a means to overcome such problems by use of additional connectives available in the logic (the *modalities*) to represent aspects of the structure of the problem space, and by providing theorem-proving techniques which utilise properties of the structure of the problem space to optimise deductions.

Why Execute Temporal Logic? Given that executing logical statements (both classical and modal) 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. Temporal logics are useful for reasoning about a changing world and, as the temporal order of actions/events can be described, they are useful for representing dynamic systems. Consequently, temporal logics are widely used in the specification and verification of *reactive* systems [53], and in applications where the concept of time is central, such as temporal planning [4], temporal representation [6] and temporal databases [22]. 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, 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. Similarly, by using temporal logics, we can describe high level properties of systems, e.g., some condition occurring *sometime* in the future, or occurring *infinitely often* (c.f., liveness and fairness).

Finally, as temporal logics are complex to reason with [62], 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 these notes

We begin our discussion by briefly outlining, in §2, the basic principles of modal logics. In §3, we outline the predominant mechanism for executing modal logics, based upon logic programming using modal resolution. As a wider range of executable temporal logics have been developed, we consider two alternative temporal logics, namely *discrete* and *interval* temporal logics in §4, both of which provide bases for execution. A range of execution options for these temporal logics are described in §5. Finally, in §6, we provide some concluding remarks.

2 Modal Logics

Modern modal logics extend classical logics with (generally) a new connective $\Box$ and its derivable counterpart $\Diamond$. These are known as *necessity* and *possibility* respectively. If a formula $\Box p$ is true, it means that p is necessarily true, i.e. true in every possible scenario, and $\Diamond p$ means that p is possibly true, i.e. true in at least one possible scenario. We can define $\Diamond$ in terms of $\Box$:

$$\Diamond p \Leftrightarrow \neg \Box \neg p$$

so that p is possible exactly when its negation is not necessarily true.

In order to give meaning to $\Box$ and $\Diamond$, models for modal logics are usually based on 'possible worlds', which are essentially a collection of connected models for classical logic. The possible worlds are linked by a relation which determines which worlds are accessible from any given world. It is this *accessibility relation* which determines the nature of the modal logic.

Each world is given a unique label, taken from a set S, which is usually countably infinite. The accessibility relation R is a binary relation on S. The pairing of S and R defines a *frame* or structure which underpins the models for the modal logic. To complete the model we add an interpretation, $h\colon S \times PROP \to \{\text{true}, \text{false}\}$, of symbols in each state so that

$$\langle S, R, h \rangle \models_s a \text{ iff } h(s, a) = \text{true}.$$

This is read as a is true in the model $\langle S, R, h \rangle$ in world s iff h maps a to true in world s. In general when a formula φ is true in a world s in a model $\mathcal{M}$ it is denoted by

$$\mathcal{M} \models_s \varphi$$

and if it is true in every world in the set S, it is said to be true in the model, and denoted

$$\mathcal{M} \models \varphi$$

The boolean connectives are given the usual meaning:

$$\langle S, R, h \rangle \models_s \varphi \vee \psi \text{ iff } \langle S, R, h \rangle \models_s \varphi \text{ or } \langle S, R, h \rangle \models_s \psi.$$
$$\langle S, R, h \rangle \models_s \varphi \Rightarrow \psi \text{ iff } \langle S, R, h \rangle \models_s \varphi \text{ implies } \langle S, R, h \rangle \models_s \psi.$$

The frame enters the semantic definition only when the modality $\Box$ is used, as the formula $\Box\varphi$ is true in a world s exactly when every world t in S which is accessible from s (i.e. we have sRt) has φ true. More formally,

$$\langle S, R, h \rangle \models_s \Box\varphi \text{ iff for all } t \in S, \quad sRt \text{ implies } \langle S, R, h \rangle \models_t \varphi$$

Examples of modal formulae To illustrate the use of modalities to represent semantic structure, we give a few examples:

- If, in all alternative scenarios, I am rich then I am currently rich: $\Box rich \Rightarrow rich$
- Either p has to be true or p has to be false: $\Box p \vee \Box\neg p$ (note that this is distinct from just $p \vee \neg p$, because we are insisting that either p is true in every possible scenario, or it is false in every possible scenario).
- If I am rich then it is necessarily possible for me to be rich: $rich \Rightarrow \Box\Diamond rich$.

For further references to modal logics, and correspondences between modal axioms and the structure of frames, see [48, 18, 67].

3 Programming with Modal Logics

As discussed above, we wish to execute modal logic in preference to classical logic in some situations, because it is easier and more natural to describe systems which involve notions such as distributed reasoning, knowledge, and need in a language which has additional syntactic units (the modalities) to abstract away generic problem space structure. Execution mechanisms can be tailored to exploit specific meanings given to those modalities.

Execution of modal logics has progressed along a similar path as execution of classical logic, namely by developing mechanical proof procedures, and turning them into automated deduction techniques. This has often involved extending existing classical proof procedures to handle modalities, for example semantic tableaux have been extended to contain representations of possible worlds [39], where the extension is based on the use of the tableaux expansion rules for universal and existential quantification. Connection matrices, derived from tableaux have also been extended to modal logics [68].

Below we consider an approach to execution based upon the logic programming paradigm.

3.1 Modal Logic Programming — MOLOG

Not surprisingly, the predominant approach to the execution of modal logics is based upon standard logic programming. This is extended so that *modal* Horn Clauses are defined, based upon the clauses used in modal resolution, while the goal reduction process itself is based upon modal, rather than classical, resolution.

In order to provide a little background to this approach, we will outline the modal resolution method upon which modal logic programming is typically based.

Resolution and Modal Logics Resolution theory for classical logic has been extended to modal logics [25]. The conjunctive normal form on which resolution is based is expanded in modal resolution to be conjunctions of disjunctions of the form

$$L_1 \vee \cdots \vee L_{n_1} \vee \Box D_1 \vee \cdots \vee \Box D_{n_2} \vee \Diamond A_1 \vee \cdots \vee \Diamond A_{n_3}$$

where the L_i are literals (atoms or negated atoms), the D_i are formulae in the form of the disjunction above. The A_i are formulae in conjunctive normal form. All variables are universally quantified.

Additional resolution rules are required to handle the modalities. In particular, due to the complexity of modal logics, modal resolution resolvents may be obtained from a single clause. This is known as *unary* resolution. For example, the following formulae, both in the normal form, are logically equivalent:

$$A \vee \Diamond((A \vee \Box B) \wedge \neg A \wedge \Diamond C)$$
$$A \vee \Diamond(\Box B \wedge \Diamond C)$$

The A and $\neg A$ in the scope of the $\Diamond$ can be used as the basis of the resolution, to obtain the second formula as a resolvent of the first.

The modal extension of the classical resolution rule takes two forms. The first resolves on two $\Box$ modalities: if we have two normal formula of the form $\Box E$ and $\Box F$ we will have E and F together in each accessible world. Therefore if there is a resolvent obtainable from E and F, that resolvent will be in every possible world. For example,

$$\Box(A \vee B)$$
$$\Box(\neg A \vee C)$$

will resolve on A and $\neg A$ to produce $\Box(B \vee C)$ as resolvent. The second modal extension of the classical resolution rule resolves on a $\Box$ and a $\Diamond$ modality: If we have two normal formula of the form $\Box E$ and $\Diamond F$ we will have E in every accessible world, and F in at least one accessible world. Therefore if there is a resolvent obtainable from E and F that resolvent will be in at least one possible world. For example,

$$\Box(A \vee B)$$
$$\Diamond(\neg A \vee \neg D)$$

will resolve on A and $\neg A$ to produce $\Diamond(B \vee \neg D)$ as resolvent.

We can summarise the unary resolution (denoted Γ), and the modal form of classical resolution (denoted Σ) by the definition below:

1. $\Sigma(A, \neg A) = \{\}$ (empty clause)
2. $\Sigma(D_1 \vee D_2, F) = \Sigma(D_1, F) \vee D_2$
3. $\Sigma(D_1 \wedge F_1, D_2 \wedge F_2) = \Sigma(D_1, D_2) \wedge F_1 \wedge F_2$
4. $\Sigma(\Box E, \Box F) = \Box\Sigma(E, F)$
5. $\Sigma(\Box E, \Diamond F) = \Diamond(E, F)$
6. $\Gamma(\Diamond(D_1 \wedge D_2 \wedge F)) = \Diamond(\Sigma(D_1, D_2) \wedge F)$

Different modal logics are obtained by varying the definitions of Σ and Γ.

A non-clausal form of modal resolution was developed by Abadi and Manna [2]. It is based on three sets of rules. A resolution rule yields resolvents by creating a disjunction of two formulae which have sub-formulae in common, and replacing the common sub-formula by 'true' in one disjunct, and 'false' in the other. Simplification rules are used to remove falsities from formulae, and to push negations onto atomic formulae. Finally deduction rules introduce logical deductions into the set of formulae being resolved. It is the members of this latter set that determine the precise nature of the modal logic. Deductions in the non-clausal modal resolution are non-trivial, and the method used to handle variables and quantification in the first-order case requires paramodulation to handle equalities that arise.

We now turn to a particular programming language based upon this approach. While, in general, few executable modal logics have been developed (though see [23, 20]), a large body of work has been carried out at IRIT developing MOLOG, a modal logic programming language of the the above form [26], and mechanisms for the extension of standard logic programming to non-classical logics [9, 8].

MOLOG itself uses modal Horn Clauses derived from the above clausal form, together with goal reduction rules based upon Modal SLD-resolution. For example,

$$\frac{\begin{array}{r} \leftarrow \Diamond p, G \\ \Box p \leftarrow B \end{array}}{\leftarrow B, G}$$

In extending this work, the group at Toulouse have developed a framework for the generation of inference mechanisms based upon non-classical extensions to logic programming, called the *Toulouse Inference Machine* [9], together with an abstract machine model for the implementation of these mechanisms, called TARSKI [8].

4 Temporal Logics

We now turn to temporal logics. As the languages described later are generally based on two different forms of temporal logic, namely *discrete* temporal logics and *interval* temporal logics, we (briefly) describe each of these separately.

4.1 Discrete Temporal Logics

This section describes a standard propositional temporal logic [24], called PTL, based on a discrete, linear model of time, with a finite past and infinite future [42]. Here, 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 can be seen as that of classical logic extended with various modalities. Although this obviously has close links with modal logic, it can be seen as a particular form of modal logic whereby the reachability relation between worlds is taken as an earlier/later (i.e. temporal) relation. In addition to the '$\Diamond$' and '$\Box$' operators used in modal logics, PTL also uses the *next-time* operator, '$\bigcirc$'. In PTL, the intuitive meaning of these connectives is as follows:

- $\Diamond A$ 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. However, rather than representing the model structure in this way, we will define a model, σ, as

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

where

- $\mathbf{N}$ is the Natural Numbers, which is 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 $\{\mathsf{T}, \mathsf{F}\}$, giving 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 interpreted.

A 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:

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

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.

$$\begin{array}{lll}
\langle \sigma, i \rangle \models \bigcirc A & \text{iff} & \langle \sigma,\; i+1 \rangle \models A \\
\langle \sigma, i \rangle \models \Diamond A & \text{iff} & \text{there exists } j \in \mathbf{N} \text{ such that } j \geq i \text{ and } \langle \sigma, j \rangle \models A \\
\langle \sigma, i \rangle \models \Box A & \text{iff} & \text{for all } j \in \mathbf{N}, \text{ if } j \geq i \text{ then } \langle \sigma, j \rangle \models A
\end{array}$$

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.

$$\begin{array}{lll}
\langle \sigma, i \rangle \models A \,\mathcal{U}\, B & \text{iff} & \text{there exists } k \in \mathbf{N}, \text{ such that } k \geq i \text{ and } \langle \sigma, k \rangle \models B \text{ and} \\
& & \quad \text{for all } j \in \mathbf{N}, \text{ if } i \leq j < k \text{ then } \langle \sigma, j \rangle \models A
\end{array}$$

$$\langle \sigma, i \rangle \models A \mathcal{W} B \quad \text{iff} \quad \langle \sigma, i \rangle \models A \mathcal{U} B \quad \text{or} \quad \langle \sigma, i \rangle \models \Box A$$

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

$$\langle \sigma, i \rangle \models \textbf{start} \quad \text{iff} \quad i = 0$$

Examples of PTL Formulae To give some idea 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: $p \wedge (\bigcirc q \vee \bigcirc r)$
- Using the *until* operator to represent intervals of time: $\mathit{nervous}\,\mathcal{U}\;\mathit{started}$
- Using the *unless* operator: $\mathit{poor}\,\mathcal{W}\,\mathit{lottery\text{-}win}$

– Temporal indeterminacy through the *sometime* operator: $born \Rightarrow \Diamond(rich \wedge happy)$
– Links with the representation of standard (imperative) computation:

$$x = 0 \wedge \bigcirc(x = x + 1) \wedge \ldots$$

– Something occurring infinitely often: $\Box\Diamond tick$
– One form of fairness: $\Box\Diamond ask \;\Rightarrow\; \Box\Diamond receive$
– Another (weaker) form of fairness: $\Box\Diamond ask \;\Rightarrow\; \Diamond receive$
– 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 [19, 52, 24, 53, 48, 18, 67].

4.2 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 [56]. Its syntax contains the basic temporal operators of PTL, together with the *chop* operator, ';', which is used to concatenate intervals.

The semantics of ITL are given over a sequence, σ, as defined for PTL. However, statements are interpreted in a sub-sequence of, rather than at a point within, σ. Thus, the semantics of ITL operators can be given as below.

$\langle\sigma_b, \ldots, \sigma_e\rangle \models P \wedge Q$	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 $b \leq i \leq e.$ $\langle\sigma_i, \ldots, \sigma_e\rangle \models P$
$\langle\sigma_b, \ldots, \sigma_e\rangle \models \bigcirc P$	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	exists $b \leq i \leq e.$ $\langle\sigma_b, \ldots, \sigma_i\rangle \models P$ and $\langle\sigma_i, \ldots, \sigma_e\rangle \models Q$

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

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

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 [56], both the languages described below 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 [51].

For further references to interval temporal logics and their applications in such diverse areas and AI and Hardware Verification, see [5, 64, 58, 7]

5 Programming with Temporal Logics

While there have been relatively few executable modal logics produced, many researchers have developed executable temporal logics. Unfortunately, there are two major problems associated with the execution of (discrete) temporal logics — complexity and incompleteness.

1. Executing PTL is complex (PSPACE-complete [62]), while executing first-order temporal logic is undecidable (and *highly* complex [54]). Consequently, two approaches to the execution of 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 [65, 3], 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 [54] 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 approaches. In §5.1, 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. Also, 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. Although logic programming does not utilise the full power of classical logics, it comprises a simple fragment of the logic (Horn Clauses), together with operational rules based upon standard resolution. Thus, once resolution rules for temporal logics were developed [2], it was not surprising that temporal logic programming languages were developed. Unfortunately, not only is the fragment of temporal logic that is used very restrictive (even more so than Horn Clauses), but efficient implementations are difficult to develop.

An alternative approach, outlined in §5.2, 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 briefly outlined in §5.3 and §5.4.

5.1 Temporal Logic Programming — TEMPLOG

As with modal logics, discrete temporal logics can be executed using the logic programming paradigm. Unfortunately, due to 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 [1, 14, 3]. Hence the search for a subset of temporal Horn Clauses logic that can be executed successfully using the logic programming paradigm.

The predominant approach to temporal logic programming is TEMPLOG [1]. Here, Temporal Horn Clauses can be categorised as either *initial* clauses (effectively if they contain **start**), or global clauses, and are restricted still further using the following constraints.

1. The '$\Diamond$' operator can only occur in the either the bodies of rules or in goals.
2. The '$\Box$' operator can only occur in what are termed, *initial definite permanent* clauses, which can be characterised as

$$\Box e \leftarrow d, \mathbf{start}, c.$$

These restrictions basically ensure that no $\Diamond$-formulae appear in the program clauses. Thus, the idea behind these restrictions 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 [2]. These are the temporal analogue of the standard resolution rule and are termed *Temporal SLD* (TSLD) resolution rules [2, 1]. In a simplified form, some of these rules are:

$$\frac{\begin{array}{r}\leftarrow \bigcirc p, G\\ p \leftarrow B \quad\quad\end{array}}{\leftarrow \bigcirc B, G} \qquad \frac{\begin{array}{r}\leftarrow \Diamond p, G\\ p \leftarrow B \quad\quad\end{array}}{\leftarrow \Diamond B, G} \qquad \frac{\begin{array}{r}\leftarrow \Diamond p, G\\ \bigcirc p \leftarrow B \quad\quad\end{array}}{\leftarrow \Diamond B, G}$$

Properties of TEMPLOG There are various technical results relating to TEMPLOG, which can be summarised as follows.

Expressive Power Baudinet [14, 13] 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 above), shows that these rules are complete. Further, she shows how propositional TEMPLOG programs represent a fragment of the temporal fixpoint calculus [10]. 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.

Characterising TEMPLOG The basic TEMPLOG form of temporal Horn Clause, together with its operational model based upon TSLD-resolution, can be characterised in a variety of different forms:

1. Orgun and Wadge [60, 59], describe a fragment of temporal Horn Clauses (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 to the restricted from of TEMPLOG called TL1 [1, 14].
2. Brzoska [16] 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 showing that TSLD-resolution is a (restricted) form of the CLP-derivation mechanism over $\mathcal{A}$.
3. Balbiani *et. al.* [9] 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 on top of Prolog.

5.2 Imperative Future — METATEM

The idea behind temporal logic programming, and TEMPLOG in particular, is to identify a subset of temporal logic that can not only be efficiently 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 complex computation.

In order to do this, we abandon the logic programming paradigm 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 analytic tableaux. In particular, the general idea behind the *imperative future* approach [43] is to rewrite each formula to be executed into a set of formulae of the form

$$\textit{condition on the past} \Rightarrow \textit{constraint on the future}$$

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

Initially, Gabbay developed the language USF [43], which provides a small set of temporal operators ('until', 'since', and 'sometime') and relies upon the Separation Theorem [50, 43] to derive executable formulae of the above form. The METATEM language [11], is a development of USF consisting of a larger range of operators, a better defined execution mechanism [31] and a more practical normal form [30] derived from the normal form used in clausal temporal resolution [36, 37].

The basic idea behind METATEM [11, 31] is to directly use the formula to be executed in order to build a model (in the case of PTL, a sequence) for the formula. The formula to be executed is transformed into a normal form consisting of *rules* 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 the *next* state.

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: $rich \Rightarrow \Box happy$
- Defining transitions between states: $(sad \wedge \neg rich) \Rightarrow \bigcirc sad$
- 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 eventually yields 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 inital 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 non-deterministic, the execution mechanism has a choice as to which states in the model to label in order to make the formula true. We can express this choice by the equivalence:

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

So to make $\Diamond a$ true, the interpreter can either

1. make a true immediately, or,
2. make $\Diamond a$ true 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 successful rules.

Thus, the interpreter has a strategy not only for choosing between disjuncts, but also for choosing when to satisfy formulae of the form $\Diamond a$ (called *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 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 (see below), the language has applications in system modelling [27], databases [28] and meta-level representation and planning [12].

5.3 Imperative Interval Temporal Logic — Tempura

Tempura [57, 45] is an executable temporal logic based upon the forward chaining execution of ITL. 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 above, Tempura incorporates various derived constructs, often with connotations in imperative programming:

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

In general, Tempura constructs intervals imperatively, rather than declaratively. It provides various advantages over the execution of discrete temporal logics, such as the ability to represent 'real-time' and to sequence operations (via ';'). However, 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 formula φ of ITL is a Tempura program if, after eliminating quantifiers and expanding predicates, it can be written in the canonical form [45], i.e.,

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

where p_i is a conjunction of assignments to basic propositions.

5.4 Interval Temporal Logic Programming — Tokio

Tokio [41, 51] 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 $\vee$.

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. Rather than giving a detailed description of the execution mechanism, we simply give a few sample goals, outlining the output produced within Tokio [41].

Tokio Examples

Goal: $\leftarrow \mathit{write}(1)$
Result: `1`

Goal: $\leftarrow \bigcirc \mathit{write}(2), \mathit{write}(1)$
Result: `12`

Goal: $\leftarrow \mathit{length}(5), \square \mathit{write}(1)$
Result: `11111`

Goal: $\leftarrow \mathit{length}(8), (\mathit{length}(5), \square \mathit{write}(0); \mathit{write}(1))$
Result: `000001111`

5.5 A Brief Survey of Other Approaches to Executable Temporal Logics

Temporal Prolog (Gabbay) Gabbay [44] introduces an alternative extension of the logic programming paradigm to temporal logics. Apart from considering temporal logics that incorporate the past, his principal difference to TEMPLOG is that he uses a subset of temporal Horn Clauses where only $\Diamond p$, $\bigcirc p$ and p (p being a proposition) are allowed in either the head or the body of a rule. Gabbay provides a semantics and various technical results relating to his system, but does not describe the implementation in any detail. However, he does extend the framework to a variety of different models of time and collections of temporal operators.

Executing Temporal Formulae Efficiently Merz [54] discusses the issues involved in the tradeoff between efficiency and expressiveness in the execution of temporal formulae. In particular, he defines a class of temporal logic formulae, which is both sufficiently abstract to support high-level descriptions of algorithms and yet, whose model construction problem is tractable.

Concurrent METATEM Although related to METATEM, Concurrent METATEM [38], is a refinement that is particularly suited to the modelling and implementation of distributed and concurrent systems. It imposes more practical operational constraints upon this execution and places the executing processes within a concurrent object-based framework. Concurrent METATEM has been applied to problems in distributed problem-solving [34], modelling societies [32], distributed AI [33], transport simulation [27], hospital simulation [61] and protocols for cooperative action [35]. For a longer survey of potential applications, see [29].

Metric Temporal Logic Programming Brzoska [15] presents an extension to temporal logic programming of the TEMPLOG form, incorporating not only past-time operators, but also metric temporal operators. He gives the correctness of the logic programming system in his framework and shows how it can be translated into a constraint logic programming [49] system over an appropriate algebra. This approach forms the basis for the more ambitious LIMETTE (Logic programming Integrating METric Temporal Extensions) system currently under development [17, 63].

Branching Temporal Logic Programming Tang [66], rather than providing an extension of Prolog that incorporates a form of temporal logic, extends the verification method for temporal logic to incorporate logic programming. In particular, the states of a model checker for CTL (a branching-time temporal logic [21]) are extended with Prolog like statements.

Temporal Prolog (Hrycej) Hrycej [46, 47] describes a completely different *Temporal Prolog* from Gabbay's (see above), which is based upon interval temporal logic. He extends Prolog using a form of Allen's interval temporal logic [6] and applies it to temporal knowledge representation and temporal planning problems.

Annotated Constraint Logic Programming Frühwirth [40] outlines another method for implementing temporal logics as an extension to logic programming, this time by defining a *temporal annotated logic*. A clausal fragment of annotated logics can be mapped directly on to a particular form of constraint logic programming.

6 Concluding Remarks

We have introduced a variety of methods for executing modal and temporal logics. In addition to outlining some of the more common execution mechanisms, we have indicated some of the applications and references for further study.

These areas, particularly that of executable temporal logics, are expanding. Consequently, we can expect to see, in the future, both a proliferation of new techniques and languages, and the refinement and application of some of those techniques outlined here.

Acknowledgements

The authors would like to thank all the other contributors to this volume, together with Michael Wooldridge, for their comments and suggestions regarding this paper.

References

1. M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8: 277–295, 1989.
2. M. Abadi and Z. Manna. Nonclausal Deduction in First-Order Temporal Logic. *ACM Journal*, 37(2):279–317, April 1990.
3. M. Abadi. The Power of Temporal Proofs. *Theoretical Computer Science*, 64:35–84, 1989.
4. J. Allen and J. Koomen. Planning using a temporal world model. In *Proc. IJCAI-83*, pages 741–747, Karlsruhe, August 1983.
5. J. Allen. Maintaining Knowledge about temporal intervals. *Comm. ACM*, 26(11):832–843, November 1983.
6. J. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.
7. J. Allen and P. Hayes. A Common Sense Theory of Time. In *Proc. IJCAI-85*, pages 528–531, Los Angeles, California, August 1985.
8. J-M. Alliot, A. Herzig and M. Lima-Marques. Implementing Prolog Extensions: a Parallel Inference Machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Japan, 1992.
9. P. Balbiani, A. Herzig, and M. Lima-Marques. TIM: The Toulouse Inference Machine for Non-Classical Logic Programming. *Lecture Notes in Computer Science*, 567, 1991.
10. B. Banieqbal and H. Barringer. 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, November 1986.
11. H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A Framework for Programming in Temporal Logic. In *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Mook, Netherlands, June 1989. (Published in *Lecture Notes in Computer Science*, volume 430, Springer Verlag).
12. H. Barringer, M. Fisher, D. Gabbay, and A. Hunter. Meta-Reasoning in Executable Temporal Logic. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Cambridge, Massachusetts, April 1991. Morgan Kaufmann.
13. M. Baudinet. A Simple Proof of the Completeness of Temporal Logic Programming. In L Fariñas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*. Oxford University Press, 1992.
14. M. Baudinet. Temporal Logic Programming is Complete and Expressive. In *POPL16*, Austin, Texas, January 1989. ACM.
15. C. Brzoska. Temporal Logic Programming with Metric and Past Operators. (In this volume.)
16. C. Brzoska. Temporal logic programming and its relation to constraint logic programming. In *Proceedings of International Symposium on Logic Programming (ILPS)*, San Diego, U.S.A., November 1991. (Published by MIT Press.).

17. C. Brzoska and K. Schäfer. LIMETTE: Logic programming integrating metric temporal extensions, language definition and user manual. Interner Bericht 9/93, Fak. für Informatik, Universität Karlsruhe, 1993.
18. R. Bull and K. Segerberg. Basic Modal Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II*. Reidel, 1984.
19. J. Burgess. Basic Tense Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II*. Reidel, 1984.
20. M. Cavalcanti. Solving Air-Traffic Problems with "Possible Worlds". (In this volume.)
21. E. M. Clarke and E. A. Emerson. Using Branching Time Temporal Logic to Synthesise Synchronisation Skeletons. *Science of Computer Programming*, 2:241–266, 1982.
22. T. Dean. Large-Scale Temporal Data Bases for Planning in Complex Domains. In *Proc. IJCAI-87*, pages 860–866, Milan, August 1987.
23. N. Den Haan. Investigations into the Application of Deontic Logic. (In this volume.)
24. E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier, 1990.
25. L. Fariñas del Cerro. Resolution modal logics. *Logique et Analyse*, 152–172, 1985.
26. L. Fariñas del Cerro. MOLOG: A system that extends Prolog with modal logic. *New Generation Computing*, February 1986.
27. M. Finger, M. Fisher, and R. Owens. METATEM at Work: Modelling Reactive Systems Using Executable Temporal Logic. In *Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-93)*, Edinburgh, U.K., June 1993. Gordon and Breach Publishers.
28. M. Finger, P. McBrien, and R. Owens. Databases and Executable Temporal Logic. In *Proceedings of the ESPRIT Conference*, November 1991.
29. M. Fisher. A Survey of Concurrent METATEM — The Language and its Applications. In *First International Conference on Temporal Logic (ICTL)*, July 1994.
30. M. Fisher and P. Noël. 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, U.K., February 1992.
31. M. Fisher and R. Owens. From the Past to the Future: Executing Temporal Logic Programs. In *Proceedings of Logic Programming and Automated Reasoning (LPAR)*, St. Petersberg, Russia, July 1992. (Published in *Lecture Notes in Computer Science*, volume 624, Springer-Verlag).
32. M. Fisher and M. Wooldridge. A Logical Approach to the Representation of Societies of Agents. In *Proceedings of Second International Workshop on Simulating Societies (SimSoc)*, Certosa di Pontignano, Siena, Italy, July 1993.
33. M. Fisher and M. Wooldridge. Executable Temporal Logic for Distributed A.I. In *Twelfth International Workshop on Distributed A.I.*, Hidden Valley Resort, Pennsylvania, May 1993.
34. M. Fisher and M. Wooldridge. Specifying and Verifying Distributed Intelligent Systems. In *Portuguese Conference on Artificial Intelligence (EPIA)*. Springer-Verlag, October 1993.
35. M. Fisher and M. Wooldridge. Specifying and Executing Protocols for Cooperative Action. In *Proceedings of International Working Conference on Cooperating Knowledge-Based Systems (CKBS)*, Keele, June 1994.
36. M. Fisher. A Resolution Method for Temporal Logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)*, Sydney, Australia, August 1991. Morgan Kaufman.
37. M. Fisher. A Normal Form for First-Order Temporal Formulae. In *Proceedings of Eleventh International Conference on Automated Deduction (CADE)*, Saratoga Springs, New York, June 1992. (Published in *Lecture Notes in Computer Science*, volume 607, Springer-Verlag).

38. M. Fisher. Concurrent METATEM — A Language for Modelling Reactive Systems. In *Parallel Architectures and Languages, Europe (PARLE)*, Munich, Germany, June 1993. Springer-Verlag.
39. M. Fitting. *Proof methods for modal and intuitionistic logics*. Reidel Publishers, 1983.
40. T. Frühwirth. Temporal Logic and Annotated Constraint Logic Programming. (In this volume.)
41. M. Fujita, S. Kono, T. Tanaka, and T. Moto-oka. Tokio: Logic Programming Language based on Temporal Logic and its compilation into Prolog. In *3rd International Conference on Logic Programming*, London, July 1986. (Published in *Lecture Notes in Computer Science*, volume 225, Springer-Verlag).
42. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. The Temporal Analysis of Fairness. In *Proceedings of the Seventh ACM Symposium on the Principles of Programming Languages*, pages 163–173, Las Vegas, Nevada, January 1980.
43. D. Gabbay. Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of Colloquium on Temporal Logic in Specification*, pages 402–450, Altrincham, U.K., 1987. (Published in *Lecture Notes in Computer Science*, volume 398, Springer-Verlag).
44. D. Gabbay. Modal and Temporal Logic II (A Temporal Prolog Machine). In T. Dodd, R. Owens, and S. Torrance, editors, *Logic Programming—Expanding the Horizon*. Intellect Books Ltd, 1991.
45. R. Hale and B. Moszkowski. Parallel programming in temporal logic. In *Parallel Architectures and Languages Europe (PARLE)*, Eindhoven, The Netherlands, June 1987. (Published as Lecture Notes in Computer Science, volume 259, Springer Verlag, Berlin.).
46. T. Hrycej. Temporal Prolog. In Yves Kodratoff, editor, *Proceedings of the European Conference on Artificial Intelligence*, pages 296–301. Pitman Publishing, August 1988.
47. T. Hrycej. A temporal extension of Prolog. *The Journal of Logic Programming*, 15(1 & 2):113–145, January 1993.
48. G. Hughes and M. Cresswell. *A Companion to Modal Logic*. Methuen (UP), 1984.
49. J. Jaffir and J-L. Lassez. Constraint Logic Programming. In *Proceedings of the Fourteenth ACM Symposium on the Principles of Programming Languages*, pages 111–119, Munich, West Germany, January 1987.
50. J. A. W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, May 1968.
51. S. Kono. A Combination of Clausal and Non Clausal Temporal Logic Programs. (In this volume.)
52. F. Kröger. *Temporal Logic of Programs*. Springer-Verlag, 1987.
53. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.
54. S. Merz. Efficiently Executable Temporal Logic Programs. (In this volume.)
55. S. Merz. Decidability and Incompleteness Results for First-Order Temporal Logics of Linear Time. In *Journal of Applied Non-Classical Logics*, 2(2), 1992.
56. B. Moszkowski. Reasoning about digital circuits. PhD Thesis, Stanford University, July 1983.
57. B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, U.K., 1986.
58. B. Moszkowski and Z. Manna. Reasoning in Interval Temporal Logic. *Lecture Notes in Computer Science*, 164, 1984.
59. M. Orgun and W. Wadge. Theory and Practice of Temporal Logic Programming. In L Fariñas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*. Oxford University Press, 1992.

60. M. Orgun and W. Wadge. Towards a unified theory of intensional logic programming. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):413–440, 1992.
61. M. Reynolds. Towards First-Order Concurrent METATEM. (In this volume.)
62. A. P. Sistla and E. M. Clarke. Complexity of propositional linear temporal logics. *ACM Journal*, 32(3):733–749, July 1985.
63. K. Schäfer. Entwicklung einer temporallogischen Sprache zur Beschreibung von Abläufen in Straßenverkehrsszenen. Diplomarbeit, Universität Karlsruhe, Inst. für Logik, Komplexität und Deduktionssysteme, 1993.
64. R. Schwartz, P. Melliar-Smith and F. Vogt. An Interval-Based Temporal Logic. *Lecture Notes in Computer Science*, 164, 1984.
65. A. Szalas and L. Holenderski. Incompleteness of First-Order Temporal Logic with Until. *Theoretical Computer Science*, 57:317–325, 1988.
66. T. Tang. Temporal Logic CTL + Prolog. *Journal of Automated Reasoning*, 5:49–65, 1989.
67. J. van Bentham. Correspondence Theory. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II*. Reidel, 1984.
68. L. Wallen. *Automated Deduction in Non-Classical Logics*. MIT Press, 1990.