An Overview of Temporal and Modal Logic Programming

Mehmet A. Orgun[1] and Wanli Ma[2]

[1] Department of Computing, Macquarie University, Sydney, NSW 2109, Australia
[2] Computer Sciences Laboratory, RSISE, The Australian National University, Canberra, ACT 0200, Australia

Abstract. This paper presents an overview of the development of the field of temporal and modal logic programming. We review temporal and modal logic programming languages under three headings: (1) languages based on interval logic, (2) languages based on temporal logic, and (3) languages based on (multi)modal logics. The overview includes most of the major results developed, and points out some of the similarities, and the differences, between languages and systems based on diverse temporal and modal logics. The paper concludes with a brief summary and discussion.

Categories: Temporal and Modal Logic Programming.

1 Introduction

In logic programming, a program is a set of Horn clauses representing our knowledge and assumptions about some problem. The semantics of logic programs as developed by van Emden and Kowalski [96] is based on the notion of the least (minimum) Herbrand model and its fixed-point characterization. As logic programming has been applied to a growing number of problem domains, some of its limitations have also started to surface, especially in those problem domains requiring the notion of dynamic change. On the one hand, we would like to be able to broaden applications of logic programming to, say, temporal reasoning, deductive databases, knowledge representation, and dataflow computation. On the other hand, we would like to remain true to the original goals of logic programming outlined by Kowalski [61]: Logic programs should specify only *what* is to be computed; not how it is to be computed.

In order to overcome some of the limitations of logic programming, many non-logical constructs in the form of annotations, extra system predicates, and infinitary objects have been introduced; for instance, see [91] for a comprehensive survey on concurrent logic languages. These languages are an important contribution to the field of programming languages and their implementations, but the problem with them is that extended "logic" programs are no longer logic. The declarative meaning of an extended program cannot be reasoned about from its logical reading; we must know the underlying execution mechanism of a particular extension to understand what a program really *does* [50, 53, 100].

One argument for these extensions is that expressiveness and efficiency of logic programming systems have priority over declarative features of logic programs such as the minimum model semantics and completeness. For instance, logic programming with infinite terms does not have a minimum model semantics and therefore the intended meaning of an infinitary logic program is not characterized by the set of logical consequences of the program [97, 64]. And this is a major problem: what an infinitary program does is not what it says declaratively. But then how can we attack problems from different domains and still keep the declarative features of logic programming if the expressiveness of our tools is so limited?

In our opinion, the solution lies not in extending logic programming with non-logical tools, but in employing more powerful logical tools. In particular, if we want to model the notion of change in time in a given application (such as that in temporal reasoning) and the dynamic properties of certain problems such as simulation, why not use temporal logic in the first place? Temporal logics have already been used in program specification and verification [65, 63], temporal reasoning [86, 32], and temporal databases [34, 95, 47]. Or if we want to model knowledge and belief, why not use modal and/or epistemic logics? These logics have been extensively studied in philosophy and mathematics and applied in many problem domains (including those in artificial intelligence) with success [8, 57, 92, 93]. Therefore we advocate extending logic programming with temporal and modal logics (or with other non-classical logics) whenever appropriate.

Recently, several researchers have proposed extending logic programming with temporal logic, modal logic and other forms of intensional logic. There are a number of modal and temporal logic programming languages: Tempura [70, 52] and Tokio [9] are based on interval logic; THLP [99, 100], Chronolog [83, 74], Templog [3, 4] and Temporal Prolog [44] are based on temporal logic; there is also another Temporal Prolog [54] based on reified temporal logic; Brzoska [26] proposed temporal logic programming based on metric temporal logic; Molog [39] is based on user-elected modal logics; Modal Prolog [87] is based on modal logic; InTense [68] is a multi-dimensional language with temporal and spatial dimensions. There are also some multi-modal approaches to temporal and modal logic programming [7, 36]. Some other non-classical extensions of logic programming include multiple-valued logic programming schemes of Blair et al [24] and of Fitting [43]. However, these two approaches deal with non-classical semantics for logic programming.

The declarative and operational semantics of some of these languages have already been worked out, and some languages have already been implemented. For instance, Baudinet [16, 17, 18] shows the completeness of the proof procedure of Templog and provides the declarative semantics of Templog programs. Orgun and Wadge [74, 75, 76] develop the model-theoretic semantics of Chronolog and describe a general framework to deal with several intensional programming systems. It is shown in [78] that Chronolog admits a sound and complete resolution-type proof procedure. Balbiani et al [13] provide a declarative and operational semantics for a class of Molog programs. Gabbay [44] showed the soundness of a

computation procedure for Temporal Prolog, based on branching time. Fitting [43] employs topological bilattices to treat the semantics of multiple-valued logic programming. For other languages such as InTense [67, 68] and Tokio [9, 60], some kind of extended operational semantics are usually provided. Progress on implementing modal extensions of Prolog has been reported in [15].

In short, there are a variety of logic programming languages based on diverse temporal and modal logics. The aim of this paper is to present a timely overview of the development of the field of temporal and modal logic programming. Some related languages and systems are not covered, for example, non-classical extensions based on intuitionistic logic, linear logic, multiple-valued logic, paraconsistent logic, fuzzy logic, etc, because our focus is on languages based on extensions of the classical logic such as temporal and modal logics, not on "alternatives" to the classical logic. In the sequel, we start by providing background on temporal and modal logics. Readers who are familiar with temporal and modal logics can skip this section. We then review temporal and modal logic programming languages under three headings: (1) languages based on interval logic, (2) languages based on temporal logic, and (3) languages based on (multi)modal logics. This classification is naturally provided by the logics these languages are based upon. The paper concludes with a brief summary and discussion.

2 Temporal and Modal Logics

Modal logic [56, 29] is the study of context-dependent properties such as necessity and possibility. In modal logic, the meaning of expressions depends on an implicit context, abstracted away from the object language. Temporal logic [82, 28] can be regarded as an instance of modal logic where the collection of contexts models a collection of moments in time. Therefore the following discussion on modal logic also applies to temporal logic. A modal logic is equipped with modal operators through which elements from different contexts can be combined. The underlying language of modal logic is obtained from a first-order language by extending it with formation rules for modal operators.

The collection of contexts is also called the *universe* or the set of *possible worlds*, and denoted by $\mathcal{U}$. The set of possible worlds $\mathcal{U}$ is not a disorganized collection. For instance, in temporal logic, $\mathcal{U}$ can be regarded as a linearly ordered set. In Kripke-style semantics, there is one ordering relation over $\mathcal{U}$, called an *accessibility* relation, associated with each modal operator [51, 82]. If $\bigtriangledown$ is a unary modal operator, the ordering relation R associated with $\bigtriangledown$ is a set of pairs of possible worlds from $\mathcal{U}$; in other words, $R \subseteq \mathcal{U} \times \mathcal{U}$. For any $< w, v > \in R$, it means that v is *accessible* from w.

Let ML denote the underlying modal language of a modal logic. A (modal) interpretation of ML basically assigns meanings to all elements of ML at all possible worlds in $\mathcal{U}$. A modal interpretation can also be viewed as a collection of first-order interpretations (Tarskian structures) one for each possible world in $\mathcal{U}$. Here the denotations of variables and function symbols are *extensional* (a.k.a. *rigid*), that is, independent of the elements of $\mathcal{U}$. This is not generally so

in modal logic; but it is quite satisfactory for application to logic programming: in logic programming, we deal with Herbrand interpretations over the Herbrand universe which consists of uninterpreted terms.

Let $P(X)$ denote the set of all subsets of the set X and $[X \rightarrow Y]$ the set of functions from X to Y. Then the formal definition of a modal interpretation can be given as follows.

Definition 1. A modal interpretation I of a modal language ML comprises a non-empty set $\mathbf{D}$, called the domain of the interpretation, over which the variables range, and for each variable, an element of $\mathbf{D}$; for each n-ary function symbol, an element of $[\mathbf{D}^n \rightarrow \mathbf{D}]$; and for each n-ary predicate symbol, an element of $[\mathcal{U} \rightarrow P(\mathbf{D}^n)]$.

All formulas of ML are *intensional*, that is their meanings may vary depending on the elements of $\mathcal{U}$. The fact that a formula A is true at world w in some modal interpretation I will be denoted as $\models_{I,w} A$. The definition of the satisfaction relation $\models$ in terms of modal interpretations is given as follows (bar the semantics of modal operators). Let $I(E)$ denote the value in $\mathbf{D}$ that I gives an ML term E.

Definition 2. The semantics of elements of ML are given inductively by the following, where I is a modal interpretation of ML, $w \in \mathcal{U}$, and A and B are formulas of ML.

(a) If v is a variable, then $I(v) \in \mathbf{D}$. If $f(e_0, \ldots, e_{n-1})$ is a term, then $I(f(e_0, \ldots, e_{n-1})) = I(f)(I(e_0), \ldots, I(e_{n-1})) \in \mathbf{D}$.
(b) For any n-ary predicate p and terms $e_0, \ldots, e_{n-1}$, $\models_{I,w} p(e_0, \ldots, e_{n-1})$ iff $< I(e_0), \ldots, I(e_{n-1}) > \in I(p)(w)$.
(c) $\models_{I,w} \neg A$ iff $\not\models_{I,w} A$.
(d) $\models_{I,w} A \wedge B$ iff $\models_{I,w} A$ and $\models_{I,w} B$.
(e) $\models_{I,w} (\forall x)A$ iff $\models_{I[d/x],w} A$ for all $d \in \mathbf{D}$ where the interpretation $I[d/x]$ is just like I except that the variable x is assigned the value d in it.

Furthermore, $\models_I A$ means that A is true in I at all worlds, that is, I is a *model* of A, and $\models A$ means that A is true in any interpretation of ML. We regard the above definition as a framework for modal logics, which is enjoyed by most of the temporal and modal languages discussed in the sequel.

The definition is, however, incomplete. We must define the semantics of modal operators available in the language. For instance, consider two classical modal operators: $\Box$ (*necessary*) and $\Diamond$ (*possible*). In Kripke-style semantics for modal logic, the meanings of $\Box$ and $\Diamond$ are determined by an accessibility relation R. One R is enough, because $\Diamond$ can be defined using $\Box$ and $\neg$; and $\Box$ using $\Diamond$ and $\neg$, depending on which of the two modal operators is chosen as a primitive operator.

Informally, $\Box A$ is true at a world w if and only if A is true at all worlds accessible from w; and $\Diamond A$ is true at w if and only if A is true at some world accessible from w. More formally, given a modal interpretation I and $w \in \mathcal{U}$,

$\models_{I,w} \Box A$ iff $\models_{I,v} A$ for all v where $< w, v > \in R$, and

$\models_{I,w} \Diamond A$ iff $\models_{I,v} A$ for some v where $< w, v > \in R$.

Note that $\Diamond A$ and $\neg \Box \neg A$ are logically equivalent; in other words, $\Diamond$ is the dual of $\Box$. If R is an equivalence relation, this gives a Kripke-style semantics for the modal logic S5 [56].

The traditional Kripke approach is, however, too restrictive, because it limits us to a dual pair of modal operators. We could extend it with extra modalities in the obvious way, by allowing a family of dual pairs, each with its own accessibility relation. This is better but still not truly general because, as Scott [90] and others have pointed out, there are many natural modal operators that cannot be defined in terms of an accessibility relation alone.

There are more general approaches to the semantics of modal logic, including "neighborhood" semantics which is usually attributed to Scott [90] and Montague [69]. For a detailed exposition of more general approaches and their relative strengths, we refer the reader to the literature, for example, see [27] and [103]. Neighborhood semantics is used in [77] to provide a unifying theoretical framework for intensional (temporal and modal) logic programming languages. In most of the other reported works, the standard Kripke-style semantics is usually employed.

As mentioned above, temporal logic [82, 28] is regarded as the *modal* approach to time. In temporal logic, there are usually two sets of modalities, one referring to the past, P (*sometime in the past*) and H (*always in the past*); and the other referring to the future, F (*sometime in the future*) and G (*always in the future*). The semantics of formulas with temporal operators (for example, $P\,A$) are defined as above using accessibility relations. The accessibility relations associated with these temporal or tense operators exhibit the properties of a time-line: (usually) discrete, linear, branching, unbounded (in either direction), dense, and so on.

3 Interval Logic Programming

Interval logic is a form of temporal logic in which the semantics of formulas are defined using temporal interpretations and *intervals*, that is, pairs of moments in time which represent a duration of time. There are two temporal languages based on interval logic, namely, Tempura [70, 52] and Tokio [9]. The execution of a program in Tempura is a reduction or transformation process. In other words, Tempura is a temporal logic programming language in a broad sense; it is not based on the "logic programming" paradigm (resolution and unification). The execution of a program in Tokio is also a reduction process, but one which is combined with resolution and unification. The very nature of the execution mechanisms of these two languages and the properties of their underlying logic called ITL [70] set them apart from the other temporal languages. Therefore we discuss them under a separate heading of interval logic programming.

3.1 Tempura

Originally proposed by B. C. Moszkowski [70], Tempura is a programming language based on discrete-time Interval Temporal Logic (ITL). The fundamental temporal operators in ITL are $\Box$ (*always*) and $\Diamond$ (*sometimes*). To partition a time interval, the *chop* or *sequential* operator, "; (*semicolon*)", is introduced. If we write $w_1 ; w_2$, it means that there is an interval σ which can be divided into two consecutive subintervals, say σ_1 and σ_2, and $w_1 ; w_2$ is satisfied on σ if and only if w_1 is satisfied on σ_1 and w_2 on σ_2. In contrast to the *sequential* operator, Tempura uses *conjunction*, $\wedge$, for concurrency. The conjunction is written as `and` in Tempura programs. For instance, $w_1 \wedge w_2$ means that both w_1 and w_2 are satisfied concurrently in the interval σ. As the intervals in Tempura are discrete, the next time operator $\bigcirc$ is also used to change the time to the next subinterval (the current interval without the first state).

The roots of Tempura are in the functional programming, imperative programming, and logic programming paradigms. As a descendant of imperative programming languages, it has a "destructive" assignment statement. Unlike the assignment statement in conventional imperative programming languages, Tempura's assignment only affects those variables that are explicitly mentioned; the values of other variables, which are not mentioned, will never be concerned. This is a special aspect of Tempura, which distinguishes it from the other temporal logic programming languages.

The programs of Tempura are a conjunction of executable ITL formulas, which are composed by the Tempura's operators. Disjunction and negation are not allowed for the sake of efficiency. To simplify writing Tempura programs, new and expressive operators are derived from the fundamental operators. Some of them are [52, page 97]:

`empty` $\equiv_{def} \neg\bigcirc$ *true*	(the zero-length interval)
`skip` $\equiv_{def} \bigcirc$ `empty`	(the unit-length interval)
`keep` w $\equiv_{def} \Box$ ($\neg$ `empty` $\rightarrow$ w)	(on all but the last state)
A `gets` B $\equiv_{def}$ `keep` ($(\bigcirc$ A) = B)	(unit delay)
`halt` C $\equiv_{def} \Box$ (C $\equiv$ `empty`)	(interval termination)
`fin` w $\equiv_{def} \Box$(`empty` $\rightarrow$ w)	(on the last state)
A $\leftarrow$ B $\equiv_{def} \exists$ x:(x=B $\wedge$ `fin` A=x)	(assignment)

Let us give some examples to show the meanings of the operators defined above. Suppose A, B, and C are temporal variables, $< seq >$ represents an interval with states seq. The values of A, B, and C over the states s, t, and u are given in Figure 1. Note that the value of a temporal variable is a sequence varying along the time axis, and that an interval with a single state is of length 0. Then we have:

$< u > \models$ `empty`
$< st > \models$ `skip`
$< ststsu > \models$ `fin`$(B = 0)$
$< ststsu > \models B \leftarrow A$

st. / var.	s	t	s	t	s	u
A	0	0	0	0	0	1
B	1	0	1	0	1	0
C	1	1	1	1	1	1

Fig. 1. The values of three variables over three states

Hale [52] gives a number of applications of Tempura, including motion representation, data transmission, and hardware design. The following program is an example of a Tempura program, which solves the "*Towers of Hanoi*" problem [52, page 100]:

```
/* Tempura solution to the "Towers of Hanoi" problem */
define hanoi(n) = exists L, C, R : {
  L = [0..n-1] and
  C = [ ] and
  R = [ ] and
  move_r(n, L, C, R) and
  always display(L, C, R)
}.
/* Move n rings from peg A to peg B */
define move_r(n, A, B, C) = {
  if n=0 then empty
  else {
    move_r(n-1, A, C, B);
    move_step(A, B, C);
    move_r(n-1, C, B, A)
  }
}.
/* Move the topmost ring from peg A to peg B */
define move_step(A, B, C) = {
  skip and
  A <- tail(A) and
  B <- append([head(A)], B) and
  C <- C
}.
```

The execution of a Tempura program is a transformation (or reduction) process. The program is repeatedly reduced according to the operations on time intervals until no interval can be divided, that is, the current subinterval contains a single state. For example, consider the following program which calculates the value of 2^m:

```
(M=4) ∧ (N=1) ∧ halt(M=0) ∧ (M gets M-1) ∧ (N gets 2*N).
```

It will be reduced to:

```
(M=4 ∧ N=1)
  ∧◯ (M=3 ∧ N=2)
  ∧◯◯ (M=2 ∧ N=4)
  ∧◯◯◯ (M=1 ∧ N=8)
  ∧◯◯◯◯ (M=0 ∧ N=16 ∧ empty).
```

The final formula is regarded as a state-description that satisfies the original formula. Therefore, the value sequence of `M` is `4, 3, 2, 1, 0` and that of `N` is `1, 2, 4, 8, 16`, over an interval of length 5.

Moszkowski [70] described an interpreter for Tempura with the details of the way in which the temporal constructs are implemented. Some applications of Tempura including algorithm description and hardware specification are also given.

3.2 Tokio

Tokio was proposed by Aoyagi, Fujita, and Moto-oka [9] for the description of computer hardware. It is based on the first-order local Interval Temporal Logic (ITL), influenced by Tempura. It is also a superset of Prolog [35]. Tokio is based on discrete linear-time.

The temporal operators in Tokio are:

- *concurrency*(,): The clause P :- Q, R means that Q and R are executed at the beginning of a time interval concurrently.
- *chop*(&&): This operator divides a time interval into two subintervals. The clause P :- Q && R means that Q will be executed at the first subinterval and R will be executed at the second subinterval.
- *next*(@): The clause P :- @Q means that Q will be executed at the time interval after the current time interval.
- *always*(#): The clause P :- #Q means that Q will be executed at all subintervals which make up the P's interval.
- *sometime*(<>): The clause P :- <> Q means that the execution of goal Q will be at some time in the interval in which P is executed.
- *keep*: The clause P :- `keep`(Q) means that Q will be executed at every subinterval of P's except the final one.
- *final*(`fin`): The clause P :- `fin`(Q), in contrast to `keep`(Q), means that Q will only be executed at the final subinterval of P's.

As in Tempura [70], variables in a Tokio program may have different values at different time intervals. In other words, the value of a variable varies with time. This makes the unification in Tokio more complicated. There are two kinds of unification in Tokio: One is concerned with unifying two Tokio variables, that is, unifying the entire sequences of values for the two variables. The second one is concerned with unifying the values of Tokio variables at specific moments in

time through the use of special unification primitives. For example, over a given interval, `X <- Y` means that the value of `Y` at the first state of the interval is unified with the value of `X` at the last state of the interval.

Intervals can be manipulated using certain builtin operators, such as `length`, `empty`, and `notEmpty`. For instance, the `length` operator is used to determine the length of an interval.

The execution of a Tokio program is a mixture of resolution and transformation (or reduction). There were two Tokio interpreters, one written in Prolog, and the other in C [60].

To the best of our knowledge, there have been no attempts at developing either the declarative or the operational semantics of Tokio programs. In order to give a formal semantics to Tokio, one would need to combine the semantics of ITL with a semantics of Prolog that explicitly represents the execution mechanism (e.g., that of Baudinet [20]).

Below are some examples of Tokio sentences and the results of their execution:

1. `length(2), @write(0) && length(2), #write(1).`

Time	0	1	2	3	4
Result	–	0	1	1	1

2. `length(5), <>write(1).`

Time	0	1	2	3	4	5
Result	–	1	1	1	1	–

3. `length(2), keep(write(0)) && length(3), #write(1).`

Time	0	1	2	3	4	5
Result	0	0	1	1	1	–

For an application of Tokio to hardware specification, we refer the reader to Masahiro *et al* [66].

4 Temporal Logic Programming

This section discusses logic programming languages and systems, based on linear- and branching-time temporal logics. The languages such as Templog [3, 4], Chronolog [74, 83], Gabbay's Temporal Prolog [44], and Sakuragawa's Temporal Prolog [88] directly extend logic programming with temporal operators. One common feature of these languages is that they all use temporal versions of resolution-based proof procedures. Some other languages such as Hrycej's Temporal Prolog [54] and Starlog [33] model time-dependent properties using reified temporal logics and additional time-parameters. MTL [26] uses a method which translates into the CLP-scheme. Starlog uses a connection-graph theorem prover.

4.1 Templog

Templog was originally proposed by Abadi and Manna [3, 4]. It uses a discrete linear-time axis with an unbounded future based on the set of natural numbers. As a programming language, Templog imposes some restrictions on its syntax for efficiency reasons. Templog adopts the syntax of Prolog, except that it has three temporal operators: $\bigcirc$ (*the next moment in time*), $\Box$ (*from now on*), and $\Diamond$ (*sometime in the future*). With these temporal operators, some new concepts appear:

- *next-atom*: an atom, which has the same syntax as in Prolog, with a prefix of next operators, for example, $\bigcirc^k A$ where A is an atom and $\bigcirc^k$ is a k-folded application of the next-time operator (for $k \geq 0$). A formula composed by next-atoms is called a *next-formula*.
- *initial clause*: $\forall x_1, \ldots, \forall x_n (A_1 \wedge \ldots \wedge A_n \rightarrow B)$ or $B \leftarrow A_1, \ldots, A_n$ where $A_1, \ldots, A_n$, and B are *next-formulas*.
- *permanent clause*: $\forall x_1, \ldots, \forall x_n \Box (A_1 \wedge \ldots \wedge A_n \rightarrow B)$ or $B \Leftarrow A_1, \ldots, A_n$ where $A_1, \ldots, A_n$, and B are *next-formulas*.

A Templog program is a conjunction of initial and permanent clauses. A goal is a conjunction of next-formulas.

The following Templog program simulates the states of a computer's cpu, which is adapted from a Chronolog program given in Orgun and Wadge [76]. Suppose that the definition of the predicate `job_queue` is defined elsewhere.

```
cpu(idle,0) ←
◯cpu(idle,0) ⇐ cpu(S,0), job_queue([ ])
◯cpu(X,N) ⇐ cpu(S,0), job_queue([[X,N]|R])
◯cpu(S,N) ⇐ cpu(S,s(N))
```

The binary predicate `cpu/2` represents the state of a cpu. Its first parameter shows the cpu state (either `idle` or the current job's name) and the second parameter indicates how many more units of time the current job will run. The cpu will be available at the next moment if either its status is `idle` or the execution of the current job has ended.

To enhance its specification ability, Templog also permits the $\Diamond$ operator to appear in the body of a clause, and $\Box$ to appear in the head of an initial clause. For instance, the following are Templog's program clauses:

```
□ employee(X) ← employee(X)
reachable(X,Y) ⇐ ◇(at(X), ◇ at(Y))
```

The first clause says that an employee who is currently employed will always be employed in the future. The second clause says that position `Y` is reachable from position `X` if there eventually exists a case that we are at position `X` and eventually get to position `Y`.

The operational semantics of Templog is given in terms of a resolution-type proof procedure, called TSLD-resolution, which is based on a restricted form of

a non-clausal temporal deduction system [2]. It has a number of rules for dealing with temporal operators. A Templog goal $\leftarrow G$ will be satisfied by an infinite sequence of related goals: $\leftarrow G$, $\leftarrow \bigcirc G$, $\leftarrow \bigcirc^2 G$, $\leftarrow \bigcirc^3 G$, and so on. Consider the following program which generates the sequence of Fibonacci numbers:

```
fib(0) ←
◯fib(1) ←
◯² fib(X) ⇐ fib(Y), ◯fib(Z), X is Y+Z
```

and the goal $\leftarrow$ `fib(X)`. At the first instant of time, `fib(X)` unifies with the first clause of the program, and yields the answer substitution $\{$`X` $\leftarrow 0\}$. And then, the next goal, $\bigcirc$`fib(X)` unifies with the the second clause of the program, and so the answer substitution is $\{$`X` $\leftarrow 1\}$. After these, a goal of the form $\bigcirc^k$`fib(X)` unifies with the third clause, and produce new goals of $\bigcirc^{k-2}$`fib(Y)` and $\bigcirc^{k-1}$`fib(Z)`. Hence we obtain a sequence of answer substitutions for `X` from the original goal:

$$\{\texttt{X} \leftarrow 0\}, \{\texttt{X} \leftarrow 1\}, \{\texttt{X} \leftarrow 1\}, \{\texttt{X} \leftarrow 2\}, \{\texttt{X} \leftarrow 3\}, \ldots$$

representing the infinite sequence of Fibonacci numbers: $< 0, 1, 1, 2, 3, 5, \ldots >$.

Baudinet [16, 17, 18] showed that TSLD-resolution is a sound and complete proof procedure for Templog, and also developed the declarative semantics of Templog programs as a temporal extension of van Emden-Kowalski semantics [96]. It is shown that Templog enjoys the minimum model semantics based on temporal Herbrand models, and its fixed-point characterization. It is also shown that the expressiveness of Templog queries, in the propositional case, corresponds to a fragment of μTL of Vardi [98] allowing only least fixed-points to be applied to positive formulas. Note that Templog is in fact equivalent in expressive power to a fragment of itself, called TL1, in which the only temporal operator is the next-time operator $\bigcirc$ [18, section 5].

Brzoska [25] showed that Templog can be considered as an instance of the CLP scheme of Jaffar and Lassez [58] over a suitable algebra $\mathcal{A}$. Templog programs are translated into classical logic programs and Templog goals into classical goals through a meaning preserving transformation, Π. Translated programs contain additional function and predicate symbols and a temporal context is added to all the predicate symbols in a Templog program. The algebra $\mathcal{A}$ consists of the free term algebra of a Templog program plus the algebra of natural numbers with functions $+1$ (successor) and $+$ (addition). It is shown by Brzoska that for any given Templog program P and goal G, $P \models G$ if and only if $\Pi(G) \models_\Pi \Pi(P)$. In Π-structures, the meaning of the additional symbols are fixed: they are interpreted over the algebra $(\mathcal{N}, 0, +1, +, =_\mathcal{N})$ where $\mathcal{N}$ is the set of natural numbers. TSLD-derivations are simulated using $(\Pi, \mathcal{A})$ derivations by introducing extra equational constraints on temporal contexts. Then the constraints are solved using the constraint-solving mechanism of the CLP-scheme. An alternative approach to the declarative and fixed-point semantics for Templog programs, as opposed to the direct approach of Baudinet [16, 17], is also given via translation. The meaning-preserving translation of Templog programs

into CLP-programs yields a new proof procedure for the derivation of Templog goals.

These results suggest that a similar translation approach is possible for other languages such as Chronolog [74] and Temporal Prolog [44]. It would be interesting to investigate the sufficient conditions under which a temporal or modal logic programming language can be considered to be an instance of the CLP-scheme.

Chomicki and Imieliński [31] proposed an extension of Datalog which has the same expressive power as (the function-free subset of) TL1. Their language (called $Datalog_{1S}$) is not directly based on temporal logic: all the predicates are extended with an extra parameter for time. The time parameter can be constructed using a specific unary function symbol denoting the *successor* function, and can be viewed as interpreted over the natural numbers. The effects of temporal operators of TL1 are simulated by manipulating the extra time parameters in predicates. It is mentioned in [19] that $Datalog_{1S}$ can be considered as a syntactical variant of (function-free) Templog, because Templog programs can be translated into TL1 programs. It is also mentioned in [19] that there is a translation from $Datalog_{1S}$ into Templog. These results suggest that a variant of SLD-resolution based on a two-sorted logic can also be used as a proof procedure for (translated) Templog programs.

4.2 Chronolog

Wadge [99, 100] proposed a tensed extension of Horn logic programming (called THLP), which was later developed into Chronolog [74, 76, 83]. The design of Chronolog was influenced by the dataflow language Lucid [101]. Hence its original target application was modelling non-terminating dataflow computations. Chronolog has two temporal operators (borrowed from Lucid): `first` refers to the initial moment in time, and `next` to the next moment in time. Like Templog, Chronolog uses the set of natural numbers as the collection of moments in time.

Chronolog adopts a C-Prolog-like syntax [35]. It accepts all the syntax of a first-order language with two new extensions: if A is a formula, so are `first` A and `next` A. The temporal operators can only apply to formulas, not to terms of the language.

Any formula, say A, in Chronolog has a sequence of values along time axis. The formula `first` A means the initial or first value of A, and the `next` A means the value of A at the next instant of time. In other words, `next` has the same semantics as the next-time operator of Templog. The accessibility relations associated with both `first` and `next` are *functional*, and hence these two operators are self-dual. In other words, we have that `first` $A \leftrightarrow \neg$`first` $\neg A$ (and for `next`).

The following Chronolog program defines the predicate `fib` which at each time t is true of the $t + 1^{th}$ Fibonacci number:

```
first fib(0).
first next fib(1).
next next fib(N) <- next fib(X), fib(Y), N is X+Y.
```

The first clause says that at the first moment in time (time 0) the number 0, the first Fibonacci number, makes `fib(0)` true; the second clause gives the second Fibonacci number, 1. The third clause gives the general frame: the current Fibonacci number (at times greater than 1) is obtained as the sum of the previous two.

Thus, from the Fibonacci program, we can prove: `first fib(0)`, `first next fib(1)`, `first next`^{2} `fib(1)`, `first next`^{3} `fib(2)` and so on. As in Templog, a goal like `<- fib(X)` triggers an attempt to prove `fib(X)` at all moments in time. It will lead to a non-terminating computation.

Now we give another Chronolog program which has appeared in Templog; the simulation of a computer's cpu states [76].

```
first cpu(idle,0).
next cpu(idle,0) <- cpu(S,0), job_queue([]).
next cpu(X,N) <- cpu(S,0), job_queue([[X,N]|R]).
next cpu(S,N) <- cpu(S,s(N)).
```

The meaning of the program is the same as that in Templog. In Chronolog, all program clauses are read as assertions true at all moments in time; hence there is an implicit always operator $\Box$ applied to all program clauses. Using the terminology from Templog, all program clauses in Chronolog are permanent. Although there are no initial clauses a la Templog in Chronolog, program clauses in which all atoms have `first` applied to them can be regarded as initial clauses, for example, the first program clause above is an initial clause.

The declarative semantics of Chronolog programs are developed using temporal Herbrand interpretations [74, 76] as an extension of van Emden-Kowalski semantics for ordinary logic programs [96, 64]. It is shown that every Chronolog program has a unique minimum temporal Herbrand model, which is regarded as the canonical meaning of the program. A fixed-point characterization of the minimum model semantics is also given.

The implementation of Chronolog is based on its operational semantics, TiSLD-resolution (a Timely SLD-resolution), which is a temporal extension of the classical SLD-resolution. It is applied to a set of *canonical instances* of program clauses and goals in which all atoms are in the scope of `first` [78]. For example, consider a program specifying the simulation of a traffic light modeled by the time-varying `light` predicate:

```
first light(green).
next light(amber) <- light(green).
next light(red) <- light(amber).
next light(green) <- light(red).
```

It says that traffic light goes `green`, `amber`, `red`, `green`, `amber`, `red`, `green`, and so on. A TiSLD-refutation for a goal `<- first next light(Color)` is:

G_0 = `<- first next light(Color),`
C_0 = `first next light(amber) <- first light(green),`
θ_0 = {`Color` $\leftarrow$ `amber`}.

G_1 = `<- first light(green)`θ_0,
C_1 = `first light(green)`,
θ_1 = {}.

C_0 is a canonical instance of the second program clause, and C_1 is the same as the first clause in the program. Thus we obtain that `first next light(Color)` is true of the program under the substitution θ_0 = {`Color` $\leftarrow$ `amber`}.

Chronolog($\mathcal{Z}$) is an extension of Chronolog with an unbounded past [78, 80], in which the set of integers $\mathcal{Z}$ is the collection of moments in time. The only extra operator in Chronolog($\mathcal{Z}$) is `prev` (*the previous moment in time*), which is the complete inverse of `next`. It is shown in [78] that TiSLD-resolution extended with rules for `prev` is a sound and complete proof procedure for Chronolog($\mathcal{Z}$), and hence for Chronolog.

When compared to Templog, Chronolog seems to lack expressive power because the operators $\Box$ and $\Diamond$ are not allowed in Chronolog. Recall that Templog has the same expressive power as one of its subsets, called TL1, in which the only allowed temporal operator is the next time operator $\bigcirc$ [18]. There is a simple transformation from TL1 into Chronolog: apply `first` to all initial clauses in Templog to preserve their meaning, and replace $\bigcirc$ by `next` and $\Leftarrow$ by `<-` in all clauses. The resulting set of clauses is a Chronolog program. It follows that Chronolog also has the same expressive power as Templog; but, a direct transformation from Chronolog to TL1 is not possible. Some clauses in Chronolog such as `p <- first q` cannot be directly transformed into TL1, because TL1 lacks `first`. These results are, however, more of a theoretical nature without any practical implications.

An early implementation of Chronolog, called μChronolog, can be found in Mitchell [67]; it is based on meta-interpretation in Prolog. Rolston [85] recently proposed an *eductive* implementation technique for ordinary logic programming (Prolog), which can also be used in implementing temporal languages such as Chronolog in a dataflow environment. Eduction [10, 42] is a standard demand-driven computation model for implementing intensional languages such as Lucid.

Another implementation based on a hybrid dataflow model is currently in progress [104]. The model is called CHEM (CHronolog Execution Model). A warehouse mechanism (such as a cache or a context-associative memory) is used to prevent repeated proving of the same goals so that the efficiency of the implementation can be improved. CHEM is a temporal extension of DIALOG [105] which is a parallel execution model for Prolog based on dataflow computation.

A non-deterministic extension of Chronolog is also proposed [76] and its semantics studied in detail in [79]. This extension is suitable for modeling certain resource-sharing problems such as that of the Dining Philosophers problem and non-deterministic dataflow computations. Rolston employed temporal logic programming to mitigate the frame problem in artificial intelligence [84]. Orgun [72] suggested that the function-free subset of Chronolog (called Temporal DATALOG) extended with temporal modules can form the basis of a temporal deductive database system.

4.3 Temporal Prolog (Gabbay)

Temporal Prolog proposed by Gabbay [44] is an extension of logic programming to allow modal and temporal connectives such as P (*sometime in the past*), F (*sometime in the future*), and $\Box$ (*always*). This language is very expressive as it also allows nested implications (and even negation as failure rule). But it also introduces a restriction: unlike in Templog, $\Box$ is not allowed in the heads of program clauses in Temporal Prolog. Thus the two languages are seemingly tailored for different kinds of applications.

The syntax of Temporal Prolog programs is defined as follows [44, definition 2.1]:

1. A *program* is a set of clauses.
2. A *clause* is either an ordinary clause or an always clause.
3. An *always clause* is $\Box A$ where A is an ordinary clause.
4. An *ordinary clause* is a head or an $A \rightarrow H$ where A is a body and H is a head.
5. A *head* is either an atomic formula or FA or PA where A is a conjunction of ordinary clauses.
6. A *body* is either an atomic formula, a conjunction of bodies, an FA or PA where A is a body.

The following are examples of program clauses:

$$\Box(\Diamond(A(x) \wedge FB(y)) \rightarrow R(z)),$$
$$a \rightarrow F((b \rightarrow Pq) \wedge F(a \rightarrow Fb)),$$

but $a \rightarrow \Box b$ is not.

Gabbay [44] outlined a computation procedure for Temporal Prolog. The soundness of the computation procedure is also shown, but it is not mentioned whether the procedure is complete or not. The computation rules are given for branching time temporal logic, and it is suggested that the procedure can be extended to consider linear-time temporal logic. The language is also extended with negation as failure, and the computation procedure is modified to handle negation in the propositional case. It is not clear whether a proof strategy on top of the computation procedure can be defined so that we can have a Prolog-like inference mechanism for Temporal Prolog.

As can be seen from the above definition, Temporal Prolog allows P and F to appear in the heads of program clauses. For instance, consider the following program:

$$A(a)$$
$$A(x) \rightarrow FB(y)$$

and the goal $\leftarrow F(B(a) \wedge B(b))$. From the second program clause, we can deduce either $B(a)$ or $B(b)$ but not both at the same time, therefore the goal fails. This is because the variable y is quantified over the temporal operator F, so the future point at which $B(y)$ is true depends on y, and may be different for different

y's. The computation procedure for Temporal Prolog requires that constants be substituted for all variables as soon as a computation rule is first applied, so that y would have a fixed value in the second clause. We do not have to know what value y is going to have when the rule is first applied, provided that y is replaced by a dummy constant. The actual value might be determined at later stages of a computation. For instance, given the goal $\leftarrow FB(z) \wedge A(z)$, we first use the second clause to deduce $FB(z)$ and replace the variables y in the second clause and z in the goal by a dummy constant, say $\overline{y}$, and then we can replace $\overline{y}$ by a using the first clause.

Temporal Prolog does not enjoy a minimum model semantics because of the use of operators such as F and P in the head of a program clause. Orgun and Wadge [77] showed that F and P are not *conjunctive*, and that the model intersection property does not hold for (intensional) logic programs when an operator which is not conjunctive is allowed to appear in the head of a program clause. The model intersection property is essential for the minimum model of a logic program to exist [77, 64]. For instance, Templog does have a minimum model semantics, because the use of $\Diamond$ is restricted to the body of a clause. Chronolog also has a minimum model semantics, because both `first` and `next` are conjunctive.

We conjecture that the meaning of the above Temporal Prolog program can be characterized with respect to the *minimal model* semantics. In any minimal model of the program, either $B(a)$ or $B(b)$ would be true at any given moment in time, but not both so that the minimality condition is satisfied. Hence either result would be justified with respect to a particular minimal model of the program. Of course, the computation procedure always tries to find the right one, that is, a minimal model in which the goal is true of the given program.

Gabbay [45] also proposed labeled deduction systems as the basis of a temporal logic programming machine. The language considered is basically Temporal Prolog enriched with extra temporal operators such as G (*always in the future*), H (*it has always been the case*), $\bigcirc$ (*the next moment in time*), and $\otimes$ (*the previous moment in time*). Also, time indicators are used to represent temporal data. For instance, a statement like "If $A(x)$ is true at t, then it will continue to be true" is represented as

$$t : A(x) \rightarrow GA(x).$$

Some restrictions are imposed on temporal program clauses to ensure tractability. For instance, Skolem functions are added to eliminate the existential connectives such as F and P. It is shown that labeled deduction systems can deal with three main flows of time: (1) general partial orders (including branching time), (2) linear orders, and (3) the integers or the natural numbers. Each flow of time requires the use of a different set of temporal rules. The soundness of the rules are also established [45].

Chen and Lin [30] studied the complexity of the satisfiability problem of temporal Horn clauses in the propositional case. They consider the future fragment of Temporal Prolog (in which the past temporal operators such as P are not

allowed) and extend it with the next-time operator $\bigcirc$ of Templog. In particular they established two complexity results:

- The satisfiability problem for the future fragment of Temporal Prolog with only $\Box$ and $\Diamond$ is NP-complete.
- The satisfiability problem for Temporal Prolog with $\Box$, $\Diamond$ and $\bigcirc$ is PSPACE-complete.

They also claim that the second result also holds for Templog programs.

4.4 Temporal Prolog (Hrycej)

Hrycej [54] proposed an extension of Prolog (also called Temporal Prolog) capable of handling temporally referenced logical statements and temporal constraints. Temporal Prolog is not directly based on temporal logic: it uses Allen's temporal constraint model [8] for reasoning about time intervals and their relationships, but with some modifications for efficiency reasons [55, section 3].

As Temporal Prolog is built on top of Prolog, it introduces two additional clause types into Prolog:

1. *Temporal references* are used to assert that a certain statement holds exactly during a time interval, for example, *P in T* where *P* is a fact or a rule and *T* is an interval identifier (such as `morning`, `afternoon`, etc.) Temporal references are retrieved using the predicates *P dur T* or *P mkdur T*.
2. *Temporal constraints* axiomatize the relationships between time intervals. They are asserted by the predicate *constrain_rel(T,S,R)* where *T* and *S* are intervals and *R* is a relationship such as "<" (before), ">" (after), "`m`" (meets), "`mi`" (met by), "`o`" (overlaps), "`di`" (contains) and so on. Interval relations are retrieved by the predicate *get_rel(T,S,R)*.

The following Temporal Prolog program is from Hrycej [54]:

```
is_to_speak(X) :- at_home(X).
is_to_speak(X) :- on_visit_at(X,Y), has_telephone(Y).

(is_to_speak(X) :- at_work(X)) in working_time.

at_home(tom) in morning.
on_visit_at(tom,john) in evening.
at_work(tom) in afternoon.

has_telephone(john).
```

The program contains both ordinary Prolog clauses and temporal references. A number of temporal constraints can be added to the program to express interval relationships, for example, `constrain_rel(afternoon,14,[di])` (*afternoon* contains time 14).

The axioms of the modified formalism are also implemented as Prolog rules using a number of additional predicates, one for each logical connective [55, section 5.2]. Temporal Prolog also requires the implementation of a temporal constraint solver (a slightly modified version of Allen's constraint solver [8]). The link between the constraint solver and Temporal Prolog is established through the *constraint-rel* predicate. The temporal constraint solver can be implemented either in Prolog, or in a procedural language with an interface to Prolog. The performance of a C-based implementation of the constraint solver is reported in Hrycej [55]: it outperforms a Prolog implementation by a thousand times.

For an application of Temporal Prolog to database queries and planning, we refer the reader to [54]. Hrycej [55] claimed that one of the potential application areas of Temporal Prolog is qualitative physics.

4.5 Temporal Prolog (Sakuragawa)

In [3, 4], there is a mention of another Temporal Prolog proposed by Sakuragawa. Although we do not have a copy of Sakuragawa's original paper [88], we include a brief review of this language based on a discussion in Abadi and Manna [3, 4] so that some common features of Temporal Prolog and Chronolog($\mathcal{Z}$) can be identified.

In Temporal Prolog, the meaning of predicates in the future depends on their meaning in the past; in other words, all the program clauses are *causal* as in Starlog [33] (see below). Just as in Chronolog, all program clauses in Temporal Prolog are permanent. Past temporal operators such as "$\bullet$" (*previous moment in time*) may occur in the body of a clause while future operators may occur in the head of a clause. Temporal Prolog is quite expressive. For instance, consider the following Temporal Prolog program:

$\Box$ *alarm* $\Leftarrow$ *dangerous(X).*

It says that "If something is dangerous, alarm will always stay on." This program cannot be directly written in Templog, Gabbay's Temporal Prolog, or in Chronolog, because $\Box$ is not allowed in the heads of permanent clauses in Templog and in Gabbay's Temporal Prolog, and Chronolog lacks $\Box$. The following clauses in Chronolog can, however, be written to the same effect:

```
alarm <- dangerous(X).
next alarm <- alarm.
```

There is also a simple translation from these clauses into equivalent Templog clauses by replacing `next` by $\bigcirc$ and `<-` by $\Leftarrow$.

In [3, 4], it is mentioned that the suggested implementation method for Temporal Prolog is by translation into ordinary Prolog. Translation is performed after all program clauses are put into a normal form where the only allowed temporal operator is the previous time operator.

If we assume that Temporal Prolog is based on a temporal logic with an unbounded past and future, TiSLD-resolution of Chronolog($\mathcal{Z}$) can be directly

applied to programs of Temporal Prolog in normal form. Therefore a restricted version of TiSLD-resolution, in which the only allowed temporal operator is `prev`, provides an operational semantics for Temporal Prolog. It can also form the basis of a direct implementation, but not necessarily an efficient one for the full language, based on unification and backtracking.

4.6 MTL

MTL (temporal logic programming with metric and past operators) proposed by Brzoska [26] has linear and unbounded (in both directions) time attributes. The set of integers $\mathcal{Z}$ is the collection of moments in time. The formulas of MTL are built with ordinary logical connectives and some temporal operators, which can only apply to formulas, not to terms. The basic temporal operators are:

1. $\Box_t$: *always within t time points*, $t \in \mathcal{Z} \cup \{-\infty, +\infty\}$;
2. $\Diamond_t$: *sometime within t time points*, $t \in \mathcal{Z} \cup \{-\infty, +\infty\}$;
3. $\circ$: *the next moment in time*;
4. $\bullet$: *the previous moment in time*.

There are also a number of derived temporal operators. Some of them are listed below:

$$\begin{array}{lll}
\Box A & \equiv_{def} \Box_{+\infty}\Box_{-\infty} A & \text{(unrestricted always)} \\
\Diamond A & \equiv_{def} \Diamond_{+\infty}\Diamond_{-\infty} A & \text{(unrestricted sometimes)} \\
\Box_+ A & \equiv_{def} \Box_{+\infty} A & \text{(always in the future)} \\
\Box_- A & \equiv_{def} \Box_{-\infty} A & \text{(always in the past)} \\
\Diamond_+ A & \equiv_{def} \Diamond_{+\infty} A & \text{(sometime in the future)} \\
\Diamond_- A & \equiv_{def} \Diamond_{-\infty} A & \text{(sometime in the past)}
\end{array}$$

A program of MTL is composed by a set of Prolog-like clauses with the temporal operators applied to their formulas [26, definition 2.3], and an MTL goal is a Prolog-like formula prefixed with some temporal operators (possibly none). As is the case in Templog, program clauses in MTL do not contain applications of $\Box$-operators in bodies, or $\Diamond$-operators in heads. MTL is, however, more expressive than either Templog or Chronolog.

It is shown that MTL can be considered as an instance of the CLP-scheme over a suitable algebra [26]. This work is in fact a continuation of the earlier results reported by Brzoska [25], but the translation function Π in MTL is different from that of [25]. It is based on the free-term algebra of an MTL program plus the algebra $(\mathcal{Z}, 0, +1, -1, +, =_{\mathcal{Z}}, \leq_{\mathcal{Z}})$ where $\mathcal{Z}$ is the set of integers. The function Π maps an MTL program P and a goal $\leftarrow G$ to the corresponding classical logic program $\Pi(P)$ and goal $\leftarrow \Pi(G)$. It is established that $P \models G$ if and only if $\Pi(P) \models_{\Pi} \Pi(G)$.

The operational semantics of MTL-programs is based on MTL-resolution, which is a restriction of the CLP-derivations over the considered algebra for tractability reasons. The soundness and completeness of MTL-resolution are also shown [26].

4.7 Starlog

Starlog [33] is a logic programming language which can handle some time-dependent problems. It is claimed that Starlog is a temporal programming language, but it is not directly based on temporal logic and it does not have any temporal operators. It simply adds an additional argument to every Prolog predicate. The first argument position of every predicate is reserved for this time-argument. For example, the statement "John is at school at time t" is expressed as `school(t,John)`. If we want to say that "John is at school from time 8.5 to 10.3" then:

```
school(t,John) <- t>=8.5, t<=10.3.
```

Note that Starlog uses the real number time axis and hence time-arguments can be real numbers. Program clauses are required to be *causal*, that is, the time values in the head of a clause must be greater than or equal to the time values in its body. Starlog does not use a standard resolution-type proof procedure for the execution of its programs, instead, it uses a variant of a connection-graph theorem prover and an intelligent arithmetic constraint solver.

5 Modal Logic Programming

This section discusses modal logic programming languages. Temporal logic languages (including interval languages) are tailored for modelling time-dependent and dynamic properties of certain problems such as simulation, temporal (historical) databases, dataflow computation, temporal reasoning and so on. On the other hand, modal logic languages target application areas involving the notions of knowledge, belief, and assumption. They also lift the requirement that the universe of possible worlds exhibit a certain intrinsic structure (such as a time-line in temporal logic), but at the same time they introduce some other requirements that the accessibility relations be reflexive, transitive, symmetric or antisymmetric, etc. Depending on the properties of the accessibility relations (or the corresponding set of axioms), we have a different modal logic.

We first discuss three modal logic programming languages (or schemes): Molog [39] (which has evolved into TIM [14, 15]), Modal Prolog [87], and Akama's proposal of modal logic programming [6]. In these works, some traditional modal logics such as B, D, S4, S5 and so on are considered, and proof procedures specialized for these modal logics are proposed. There are also multi-dimensional languages such as InTense [68, 67] and 3-Dimensional (3-D) logic programming [77] with extra spatial dimensions. These languages are based on a (multi)modal logic in which the accessibility relation associated with each modality is functional. Two other more standard approaches to (multi)modal logic programming based on translation [7, 36] are also summarized.

5.1 Molog

Molog is a modal system proposed by Fariñas del Cerro [39]. Molog extends Prolog to a modal logic programming language so that it can express concepts

of belief, knowledge, or assumption. The user fixes a modal logic, and defines the rules to deal with modal operators; in this respect, Molog is a framework which can be "instantiated" with particular modal logics. Modal operators are grouped in two categories: universal operators (such as $\Box$), and existential operators (such as $\Diamond$). Standard Kripke-style semantics is employed for the semantics of modal operators.

The basic structure of Molog is a *modal Horn clause*, which is an ordinary Horn Clause governed by modal operators. Modal operators can only be applied to formulas. The general form of a modal Horn clause is as follows:

Modality (Modal atomic formula $\leftarrow$ Conjunctive modal formula).

A sequence of modal operators is called a *Modality*, which can qualify a modal atomic formula, a conjunction of them, or a whole modal Horn clause. An example of a *Modal Horn Clause* is:

knows(paul)(come_back(X) $\leftarrow$ comp(paul) it_rains & tired(X)).

It says that "Paul *knows* that if someone is tired and it is *compatible* with Paul's knowledge that it is raining then this person should come back." [21, page 762] A Molog program is a set of modal Horn clauses, and a goal is a conjunctive modal formula.

Molog uses a modal resolution method whose roots are in a deduction method for modal logics [38]. The general modal resolution rule (inference step) in Molog has the following form [39, page 40]:

$$\frac{\begin{array}{l} M_1(E_1 \leftarrow B) \\ \quad \leftarrow M_2(E_2) \end{array}}{\leftarrow M_3(E_3, B); \text{if } T(M_1E_1, M_2E_2) \text{ is verified.}}$$

where T possesses an associated procedure reflecting the corresponding modal proof theory (based on selected rules for modal operators) for a user-elected modal logic and M_1, M_2, and M_3 are modalities.

Molog requires the use of a different modal resolution method for different modal logics. Since the resolution rules can be created by the user (or selected from a library of resolution rules), Molog is very flexible, but it is not clear how the consistency (and the completeness) of the resulting resolution method can be guaranteed. As for standard modal logics such as S5, it is shown that an instance of Molog, based on S5, has a complete modal resolution method [39]. The following example of an S5-Molog program is taken from [39, page 42]:

knows(pierre) knows(jean) q
knows(pierre)($p \leftarrow$ knows(jean) q)

where "knows(a)" for any term a is regarded as a universal modal operator. Using the modal resolution method for S5, it can be shown that, for example, the goal "$\leftarrow$ knows(pierre) p" follows from the program.

Fariñas del Cerro [39] gives another deduction method for S5-Molog. It is based on a technique called *compilation*, by which all modal Horn clauses are

first transformed into a simple form in which no modalities are applied to a whole modal Horn clause, and then a Prolog-like inference rule based on resolution and unification is used. The completeness of the deduction method by compilation is also shown.

The complexity of the satisfiability of modal Horn clauses, in the propositional case, is also studied [37]. A bottom-up algorithm is given for testing the satisfiability of sets of propositional modal Horn clauses. It is shown that, for some modal logics such as S5, the algorithm works in polynomial time, while for other modal logics such as K, Q, T, and S4 the worst case complexity can be considerably higher.

Balbiani et al [13] discussed the declarative semantics of an instance of Molog based on modal logic Q (in which all accessibility relations associated with modal operators are serial). There, it is shown that the interpretations of a Q-Molog program can be represented by a tree and each Q-Molog program has a minimum Kripke model, which is the limit (the least fixed-point) of a certain transformation over trees. A modal SLD-resolution method is also defined, and its completeness is shown. In order to ensure that modal resolution rules are closed, Skolem techniques are employed to eliminate the occurrences of the existential modal operator $\Diamond$. The results are also extended to instances of Molog based on modal logics T and S4. Orgun and Wadge [77] provided a general model-theory for intensional logic languages which can also be applied to these instances of Molog. They showed, using certain semantic properties of modal operators, that Molog programs considered in [13] enjoy the minimum model semantics, and its fixed-point characterization.

At first, there were two main assumptions in the design of Molog [14]:

- to keep the fundamental logic programming mechanisms: backward chaining, depth-first strategy, backtracking, and unification;
- to parameterize the inference step: it is the user who specifies how to compute a new goal from a given one, in logic programming style.

A further extension makes it possible for a user to select clauses which can not exactly unify with the current goal, but just resemble it in some way. This gave birth to a more general modal logic programming system—TIM (*Toulouse Inference Machine* [14]). As its predecessor, TIM is a general framework for modal logic programming. It can produce a concrete logic programming language, such as Prolog, Templog, modular Prolog, and an epistemic logic programming language etc., by a certain specification: the user specifies the inference rules for modal operators, and then provides a mechanism for clause selection. It is not mentioned whether it can produce languages such as Gabbay's Temporal Prolog in which P and F are allowed in the head of a clause.

In [14, 15], it is reported that a prototype implementation of TIM was written in LISP, and a compiled version in C, following the same principles as WAM (Warren's Abstract Machine [102, 5]). The underlying abstract machine of TIM implementations is called TARSKI (*Toulouse Abstract Reasoning System for Knowledge Inference*). These implementations of TIM led to a distributed implementation on a network of workstations linked by Internet sockets. In the

distributed implementation, which is written in Ada, each workstation runs a TARSKI machine; the early performance of a top-down configuration of the network is also reported [15].

Owing to its parameterized inference mechanism, TIM can produce working implementations of modal logic programming languages (such as Q- or S5-Molog) with minimal effort.

5.2 Modal Prolog

Modal Prolog [87] extends Prolog to give it the abilities to express modularity, hierarchy, and/or structure. A modal logic program consists of two parts, a possible world description and a relationship description upon the possible worlds. Possible worlds descriptions of Modal Prolog resemble program clauses with time indicators (labels) in Gabbay's labeled deduction systems [45].

1. *The description of a possible world*: This is like a Prolog program with modal operators applied to atoms. The modal operators are $\Box$ (necessity) and $\Diamond$ (possibility). Modal operators can not be used in the head of a clause. Every world is identified by a unique name, and program clauses for a possible world form the axioms of that world. There are only finitely many possible worlds. Therefore the use of $\Box$ in the body of a clause (similar to the use of the universal quantifier $\forall$) does not cause any problems in Modal Prolog.
2. *Relationship description*: This gives the accessibility relations between the possible worlds and their attributes such as reflexivity, symmetry, transitivity etc. A relation atom is of the form $re(w_1, w_2)$, where w_1 and w_2 are the names of possible worlds. A relation clause is just like a Prolog clause, except that it describes the attributes of the accessibility relations. The following example describes an accessibility relation re which is reflexive, symmetric, and transitive:

   $re(X, X)$;
   $re(X, Y) \leftarrow re(Y, X)$;
   $re(X, Z) \leftarrow re(X, Y), re(Y, Z)$;

   The following Modal Prolog program is taken from [87, page 87]:

```
relation of
  re(w1,w2); re(w1,w3);
fo.
world w1 of
  p(X) ← □q(X);
  r(X) ← ◇s(X);
fo.
world w2 of
  q(a); q(b); s(a);
fo.
world w3 of
```

```
  q(b); q(c); s(b);
fo.
```

With the description of the relationship of the possible worlds, we can prove that in world `w1`, `□q(b)`, `◇q(a)`, `◇q(c)`, `◇s(a)`, `◇s(b)`, `p(b)`, `r(a)`, and `r(b)` are all true.

Every possible world description corresponds to a module of the program. The interactions among modules are described in the relationship definition, and values from different worlds are combined using modal operators. Thus modal logic programming introduces a hierarchic structure into logic programming. We think that a similar approach can also be applied to "distributed logic programming [81]" where parts of a given program are stored at different sites in a distributed environment.

Modal Prolog enjoys a *possible-world model* and its operational semantics is based on resolution and unification. The soundness and completeness of the proof procedure of Modal Prolog are shown [87]. A simple meta-interpreter for the language, written in Prolog, is also given.

5.3 Modal Logic Programming (Akama)

Akama [6] proposed an extension of Prolog based on modal logic S5. A standard Kripke-style semantics is employed for modal operators. The basic building blocks of modal logic programs are *modal definite clauses*, which consist of a conjunction of modal literals with only one positive literal. A modal literal is either of the form $\bigtriangledown A$ or of the form $\neg(\bigtriangledown A)$ where $\bigtriangledown$ is a modality (a sequence of modal operators $\Box$ and $\Diamond$) and A is an atomic formula. A modal logic program is a set of modal definite clauses. As a proof procedure for modal logic programs, an extension of the standard SLD-resolution with two additional inference rules for modal operators is defined, and its completeness is established.

A generalization of the modal logic programming to (multi)modal logic programming is also considered, but no realistic implementation of (multi)modal logic programming is suggested. Modal definite clauses considered in this work are analogous to those obtained by the compilation technique for S5-Molog [39], but, since there are no restrictions on the use of modal operators in Akama's work [6], the usefulness of the modal SLD-resolution in defining a satisfactory interpreter for the language is questionable. If restrictions are imposed (as in Templog or in Molog), we can conclude that Akama's proposal of modal logic programming is subsumed by Molog.

5.4 InTense

The multi-dimensional logic language InTense was invented by Mitchell and Faustini [68, 67]. It was inspired by both Chronolog and Lucid [101]. Lucid is a functional programming language, in fact, an intensional extension of ISWIM [11, 101]. In Lucid, as in Tempura, variables represent a sequence of values over the time dimension.

InTense has a large repertoire of Lucid-like intensional operators such as `asa` (*as soon as*), `wvr` (*whenever*), and `upon`, in addition to the temporal operators of Chronolog. Formulas in an InTense program vary not only along the time axis, but also along the spatial axis. The language can, in theory, accommodate a (possibly infinite) number of temporal and spatial dimensions; thus a possible world in InTense is a point in a time-space hyperfield, namely, $\mathcal{Z}^\omega$.

InTense is also a Prolog-like programming language. An InTense program consists of a set of intensional Horn clauses, which are first-order Horn clauses with intensional operators applied to formulas. All program clauses are interpreted as assertions true at *all* possible worlds (i.e., points in $\mathcal{Z}^\omega$). In other words, they are all permanent as in Chronolog.

Intensional operators are `prev`, `first`, and `next` on the time axis and `prior`, `initial`, and `rest` on the space axis. Temporal operators `prev`, `first`, and `next` are just like the operators of Chronolog($\mathcal{Z}$). The operator `initial` refers to the original point in space (point 0). Spatial operators `prior` and `rest` are analogous to `prev` and `next`, except that they operate on the space axis. There is also a dimension indicator attached to these operators. If it is omitted, intensional operators are for the default dimensions (that is, the first time and space dimensions). Note that the accessibility relations associated with the operators `initial`, `prior`, `rest`, `first`, `prev`, and `next` are functional; hence all the operators are self-dual.

The following InTense program is taken from Mitchell [67, page 32] (it uses the Sieve of Eratosthenes to produce the sequence of prime numbers). Note that InTense retains the extra-logical and non-logical features of Prolog, for instance, it has the cut "`!`", and `assert` and `retract`.

```
initial ints(2) :- !.
rest ints(X) :- ints(Y), X is Y+1.

first sieve(X) :- ints(X), !.
next sieve(X) :- initial sieve(Y),newsmallest(X),not(0 is X mod Y).

newsmallest(X) :- sieve(X).
newsmallest(X) :- rest newsmallest(X).

prime(X) :- initial sieve(X).
```

As shown in Figure 2, the `sieve` predicate represents the sequence of prime numbers along the time dimension at point 0 in space. The program makes use of the first time and space dimensions only.

There are two reported InTense interpreters [67]. The first one is written in Prolog and translates InTense programs into Prolog programs. The other is written in C, which is based on Warren's Abstract Machine WAM [102] with intensional extension. In the C implementation, up to four temporal dimensions and four spatial dimensions are supported. It is not clear how the Lucid-like operators (such as `wvr` and `asa`) can be given an operational semantics and proof

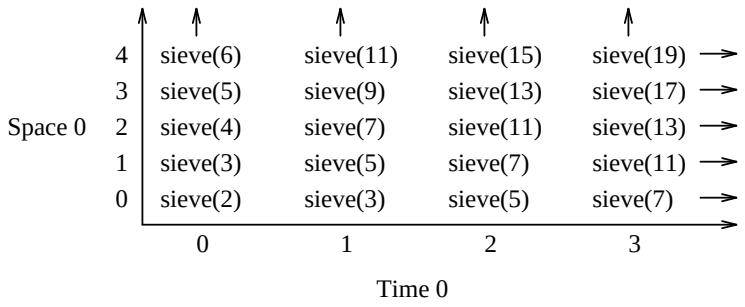

Fig. 2. The `sieve/1` relation in two dimensions

rules can be defined for them, and yet they are implemented. The correctness issues of the implementation are not addressed. Note that, in InTense, Chronolog programs can be run unchanged.

Another approach similar to InTense is 3-D Logic Programming Language of Orgun and Wadge [77], which extends Chronolog with 2 spatial dimensions. There are some modal operators on the extra spatial dimensions, namely, `side`, `edge`, `north`, `south`, `west`, and `east`. The operators `side` and `edge` are analogous to `initial` for the first two spatial dimensions in InTense. Hence the 3-D language can be considered as a pure subset of InTense restricted to 3-D. It is shown that the 3-D logic programming enjoys the minimum model semantics [77], but no completeness results are offered for the language. Another multi-dimensional language which is in spirit similar to "pure" InTense is considered by Orgun and Du [73]. They propose a resolution-based proof procedure for the language and outline a spreadshet interface which can be used for both query formulation and the display of results.

5.5 Multi-Modal Approaches

A Meta-logical Foundation (Akama, 1989): Akama [7] proposed a foundation for modal logic programming. It is based on the simulation of a model (or proof) theory of modal logic in the Horn clause logic programming, using meta-programming techniques. It is a departure from the previous work by Akama [6] in which a modal resolution method was defined.

Modal programs are translated into programs of a two-sorted first-order logic by introducing world paths as extra parameters to function and predicate symbols. A meta-interpreter is given for the execution of translated programs. The translation method does not require the axiomatization of the accessibility relations associated with modal operators such as $\Box$ and $\Diamond$ in modal logic programs, rather it directly encodes the accessibility relations into the unification algorithm. This encoding technique is borrowed from Ohlbach's resolution calculus for modal logics [71]. It is claimed that the encoding technique improves the efficiency of the implementations (meta-interpreters) of modal logic programming,

but it requires special unification algorithms for different types of modal logics. Considered modal logics are T, S4, S5, B, D, D4 and DB.

Although it is shown that a modal Herbrand property holds for translated modal formulas, the declarative semantics of modal logic programs are not given. A version of the minimum model semantics does not seem to hold for the class of modal logic programs considered; in fact some of the examples given in Akama [7] contain the use of negation in the heads of program clauses.

Multimodal Logic Programming (Debart et al, 1992): Debart *et al* [36] extended the automated theorem proving method in modal logic of Aufray and Enjalbert [12] to multimodal logic. The resulting method is called Σ-E-resolution. Then a restricted version of Σ-E-resolution is applied to modal logic programming. Modal logics considered are KD, KT, KD4, KT4, and KF. When restricted to a single modality, this approach to modal theorem proving is, in spirit, similar to Akama's [7].

For each modality i (chosen from the above mentioned modal logics), there is a pair of modal operators, $\Diamond_i$ and $\Box_i$, and $\Diamond_i$ is the dual of $\Box_i$. Standard accessibility relations are used to define the semantics of modal operators, and it is required that the relations be serial. (Multi)modal formulas are first translated into formulas of order-sorted equational theories, preserving satisfiability, and then Σ-E-resolution is used to show satisfiability of the translated formulas. The translation of modal formulas involves the addition of as many extra arguments (world paths) to function and predicate symbols as the total number of modalities. Translated formulas are called *path formulas*. Because of the extra world path arguments, a special unification algorithm is developed. The properties of modal operators are encoded in the unification algorithm as in Akama's work [7]; hence there is no need to axiomatize the accessibility relations associated with modal operators in translated programs.

When path formulas are restricted to Horn clauses, we have a logic programming language called Pathlog. Debart et al [36] showed that Σ-E-resolution with the restriction to Horn clauses is a sound and complete proof procedure for Pathlog. This result provides a basis for multimodal logic programming for the case where multimodal logic programs can be translated into the Horn subset of path formulas. In particular, it is shown that for any given Templog program [3, 4], there is a corresponding Pathlog program; hence it is suggested that Pathlog can be the target language for "compiling" Templog programs. Whether such a compilation technique would provide an efficient interpreter for Templog or not is not discussed.

6 Discussion

An overview of the progress to date in temporal and modal logic programming has been presented. The three classes of interval logic languages, temporal logic languages, and (multi)modal logic languages are *logical*, at least at this stage. The classes are naturally identified by the logics these languages are based upon.

In our opinion, considering that temporal and modal logic programming is a relatively new field of research which gained momentum in the mid-eighties, the depth and variety of the reported work have been quite extensive. The origins of most of the languages are in modal and temporal theorem proving, and logic programming (Prolog), but languages such as Tempura [70] and Tokio [9] have certain features also found in imperative languages. Figure 3 summarizes all the reviewed languages with a few exceptions (such as Sakuragawa's Temporal Prolog [88] and multi-dimensional logic programming [73]).

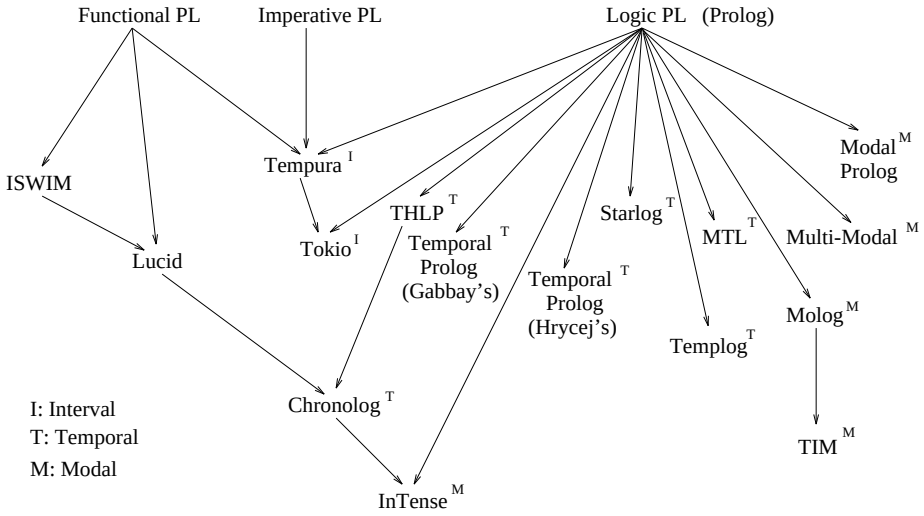

Fig. 3. Temporal and modal logic programming languages

Results on the completeness of temporal and modal logic programming are very encouraging, given that first-order temporal and modal logics are inherently incomplete [1, 94]. We now know that there are complete subsets of temporal and modal logics which can be used as the basis of logic programming. In standard logic programming, the declarative and operational semantics of logic programs coincide [96, 64]. The completeness results (such as those of Templog [16, 17, 18], Chronolog [74, 78], Molog [13], MTL [26]) guarantee that the declarative and operational semantics of temporal and modal logic programs of those languages also coincide. As a consequence, we can use the tools and techniques from temporal and modal logics to *reason about* programs and study their properties, but this is not yet a well-explored area. Similar completeness results are also established for other non-classical languages; we refer the reader to the literature for more details on languages based on linear, paraconsistent, intuitionistic, defeasible logics, and so on, and their applications [23, 89, 41, 49, 59].

There are also some general semantic frameworks for temporal and modal

logic programming. Orgun and Wadge [77] provide a unified model-theory for intensional (temporal and modal) logic programming languages, based on Scott-Montague neighborhood semantics for intensional operators. It explains the reasons behind the restriction on the uses of $\Box$ and $\Diamond$ in Templog and Molog. $\Diamond$ is not a *conjunctive operator*, and conjunctivity is related to the model intersection property. If $\Diamond$ is used in the head of a clause, the minimum model semantics no longer applies. For example, Gabbay's Temporal Prolog does not enjoy the minimum model semantics, because it allows the use of non-conjunctive P and F in the head of a clause. $\Box$ is not a *continuous* operator, and continuity is related to computability. If $\Box$ is used on the body of a clause, the fixed-point semantics no longer applies. Kriaučiukas [62] also proposed a semantics framework for non-classical logic programming based on Kripke-style semantics for modal logic, and investigated the connections between logic programming and dynamic logic. Another general framework is provided by Blair et al [24, 22] for non-classical extensions of logic programming.

We believe that the utility of temporal and modal logic programming has been demonstrated in the works reviewed in this paper; we are, however, yet to see their use in more realistic applications. One of the key features of temporal and modal logic programming is that it provides an abstraction, which we call *context abstraction*. Context abstraction plays an important rôle in many applications involving the notion of dynamic change, time, belief, and knowledge. Through context abstraction, temporal and modal logic programming can be used to describe dynamic and context-dependent properties of certain problems in a natural and problem-oriented way. It also introduces a form of parallelism, that is, *context parallelism*: goals at different contexts (possible worlds) can be executed in parallel. This form of parallelism is in addition to the standard AND- and OR-parallelism existing in logic programs.

Other related issues such as meta-logics, meta-interpretation, translation vs direct implementation of temporal and modal logics are also extensively discussed in the literature; for example, see [48, 44, 46, 36, 40]. Many implementations of temporal and modal languages are built on top of Prolog, either with some modifications on the unification algorithm, or with some extra rules for temporal and modal operators. We are in favour of direct implementations of temporal and modal logic programming so that we can, for instance, exploit context-parallelism, but translation and also meta-interpretation provide a good basis for rapid-prototyping some experimental systems. Meta-interpretation can also be used to investigate the language features which may or may not be feasible and/or practical for inclusion in production languages. In this respect, TIM [14, 15] is a very promising system which has the ability to produce many concrete non-classical languages with minimal effort.

We include a timeless quotation from Scott [90, page 143] which we believe is also valid for temporal and modal logic programming:

> "One often hears that modal (or some other) logic is pointless because it can be translated into some simpler language in a first-order way. Take no notice of such arguments. There is no weight to the claim that

the original system must therefore be replaced by the new one. What is essential is to single out important concepts and to investigate their properties."

In the near future, as the implemented systems mature and efficient parallel implementations emerge, we think we are going to see more and more languages applied to a broader range of applications with success.

Acknowledgements

Wanli Ma has been partially supported by an Overseas Postgraduate Research Award (OPRA) from the Australian Government. Thanks are due to Lee Flax, Jan Hext and Kang Zhang for their useful comments and suggestions on an earlier draft of this paper.

References

1. M. Abadi. The power of temporal proofs. *Theoretical Computer Science*, 65(1989):35–83, 1989.
2. M. Abadi and Z. Manna. Nonclausal temporal deduction. In R. Parikh, editor, *Proc. of Conference on Logics of Programs*, volume 193 of *LNCS*, pages 1–15. Springer-Verlag, 1985.
3. M. Abadi and Z. Manna. Temporal logic programming. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 4–16, San Fransisco, Calif, 1987. IEEE Computer Society Press.
4. M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8:277–295, 1989.
5. Hassan Ait-Kaci. *The WAM: A (Real) Tutorial.* Paris Research Laboratory, Digital Equipment Corporation, Paris, France, 1990.
6. Seiki Akama. A proposal of modal logic programming (extended abstract). In *Proc. of the 6th Canadian Conference on Artificial Intelligence*, pages 99–102, École Polytechnique de Montréal, Montréal, Québec, Canada, May 1986. Presses de l'Université du Québec.
7. Seiki Akama. A meta-logical foundation of modal logic programming. 1-20-1, Higashi-Yurigaoka, Asao-ku, Kawasaki-shi, 215, Japan, December 1989.
8. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, November 1983.
9. T. Aoyagi, M. Fujita, and T. Moto-oka. Temporal logic programming language Tokio. In E. Wada, editor, *Logic Programming'85*, volume 221 of *LNCS*, pages 138–147. Springer-Verlag, 1986.
10. E. A. Ashcroft, A. A. Faustini, and B. Huey. Eduction: A model of parallel computation and the programming language Lucid. In *Proc. of Phoenix Conference on Computers and Communications*, pages 9–15. IEEE Computer Society Press, 1985.
11. E. A. Ashcroft and W. W. Wadge. Lucid – a formal system for writing and proving programs. *SIAM Journal on Computing*, 5:336–54, September 1976.
12. Y. Aufray and P. Enjalbert. Modal theorem proving: an equational viewpoint. To appear in *Journal of Logic and Computation*, 1992.

13. Philippe Balbiani, Luis Fariñas del Cerro, and Andreas Herzig. Declarative semantics for modal logic programs. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 507–514. ICOT, 1988.
14. Philippe Balbiani, Andreas Herzig, and Mamede Lima-Marques. TIM: The Toulouse inference machine for non-classical logic programming. In *PDK'91: International Workshop on Processing Declarative Knowledge*, volume 567 of *LNAI*, pages 365–382. Springer-Verlag, 1991.
15. Philippe Balbiani, Andreas Herzig, and Mamede Lima-Marques. Implementing Prolog extensions: a parallel inference machine. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 833–842. ICOT, 1992.
16. M. Baudinet. On the semantics of temporal logic programming. Technical Report STAN–CS–88–1203, Computer Science Department, Stanford University, Stanford, Calif, June 1988.
17. M. Baudinet. Temporal logic programming is complete and expressive. In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, pages 267–280, Austin, Texas, January 1989. The Association for Computing Machinery.
18. 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*, pages 51–83. Oxford University Press, 1992.
19. M. Baudinet, J. Chomicki, and P. Wolper. Temporal deductive databases. In A. Tansel and et al, editors, *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1993.
20. Marianne Baudinet. Proving termination properties of Prolog programs: a semantic approach. *Journal of Logic Programming*, 14(1&2):1–30, October 1992.
21. Pierre Bieber, Luis Fariñas del Cerro, and Andreas Herzig. MOLOG: A Modal Prolog. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction*, pages 762–763. Springer-Verlag, 1988.
22. H. A. Blair, A. L. Brown, and V. S. Subrahmanian. Monotone logic programming. In L. Fariñas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*, pages 1–22. Oxford University Press, 1992.
23. H. A. Blair and V. S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.
24. H.A. Blair et al. A logic programming semantics scheme, Part I. Technical Report LPRG–TR–88–8, Logic Programming Research Group, Syracuse University, 1988.
25. Christoph Brzoska. Temporal logic programming and its relation to constraint logic programming. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Logic Programming Symposium*, pages 661–677, San Diego, Calif, October 28-31 1991.
26. Christoph Brzoska. Temporal logic programming with metric and past operators. Universität Karlsruhe, P.O.Box 6980, D-7500 Karlsruhe, Germany, January 1992.
27. R. Bull and K. Segerberg. Basic modal logic. In D. M. Gabbay and F. Guethner, editors, *Handbook of Philosophical Logic, Vol. II*, pages 1–88. D. Reidel Publishing Company, 1984.
28. J. P. Burgess. Basic tense logic. In D. M. Gabbay and F. Guethner, editors, *Handbook of Philosophical Logic, Vol. II*, pages 89–134. D. Reidel Publishing Company, 1984.
29. B. F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.
30. Cheng-Chia Chen and I-Peng Lin. The computational complexity of satisfiability of temporal Horn formulas in propositional linear-time logic. *Information*

Processing Letters, 45:131–136, March 1993.

31. Jan Chomicki and Tomasz Imieliński. Temporal deductive databases and infinite objects. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 61–73. The Association for Computing Machinery, 1988.
32. Eugène Chouraqui. Formal expression of time in a knowledge base. In P. Smets, A. Mamdani, D. Dubois, and H. Prade, editors, *Non-Standard Logics for Automated Reasoning*, pages 81–103. Academic Press, 1988.
33. John G. Cleary and Vinit N. Kaushik. Updates in a temporal logic programming language. Technical report, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, 1991.
34. James Clifford and David S. Warren. Formal semantics for time in databases. *ACM Transactions on Database Systems*, 8(2):214–254, June 1983.
35. W. Clocksin and C. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.
36. Françoise Debart, Patrice Enjalbert, and Madeleine Lescot. Multimodal logic programming using equational and order-sorted logic. *Theoretical Computer Science*, 105(1992):141–166, 1992.
37. L. Fariñas del Cerro and M. Penttonen. The complexity of the satisfiability of modal Horn clauses. *Journal of Logic Programming*, 4:1–10, March 1987.
38. Luis Fariñas del Cerro. A simple deduction method for modal logic. *Information Processing Letters*, 14(2), 1982.
39. Luis Fariñas del Cerro. MOLOG: A system that extends PROLOG with modal logic. *New Generation Computing*, 4:35–50, 1986.
40. Luis Fariñas del Cerro and Andreas Herzig. Metaprogramming through intensional deduction: some examples. In *Meta-Programming in Logic: Proc. of the Third International Workshop, META-92, Uppsala, Sweden*, pages 11–25. Springer-Verlag, June 1992.
41. Luis Fariñas del Cerro and Martti Penttonen, editors. *Intensional Logics for Programming.* Oxford University Press, 1992. ISBN 019-853775-1.
42. A. A. Faustini and W. W. Wadge. An eductive interpreter for pLucid. Technical Report TR-006-86, Department of Computer Science and Engineering, Arizona State University, 1986.
43. Melvin Fitting. Logic programming on a topological bilattice. *Fundamenta Informaticae*, XI:209–18, 1988.
44. D. M. Gabbay. Modal and temporal logic programming. In A. Galton, editor, *Temporal Logics and Their Applications*, pages 197–237. Academic Press, 1987.
45. D. M. Gabbay. A temporal logic programming machine [modal and temporal logic programming, Part 2]. Department of Computing, Imperial College, November 1989.
46. D. M. Gabbay. Metalevel features in the object level: modal and temporal logic programming III. In L. Fariñas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*, pages 85–123. Oxford University Press, 1992.
47. Dov Gabbay and Peter McBrien. Temporal logic & historical databases. In *Proceedings of the 17th Very Large Data Bases Conference*, pages 423–430, Barcelona, Spain, September 1991. Morgan Kauffman, Los Altos, Calif.
48. A. Galton. Temporal logic and computer science: an overwiev. In A. Galton, editor, *Temporal Logics and Their Applications*, pages 1–52. Academic Press, 1987.
49. A. Galton, editor. *Temporal Logics and Their Applications.* Academic Press, 1987.

50. R. Gerth, M. Codish, Y. Lichtenstein, and E. Y. Shapiro. Fully abstract denotational semantics for Flat Concurrent Prolog. Technical Report CS88–03, Department of Applied Math. and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, April 1988.
51. R. Goldblatt. *Logics of Time and Computation.* CSLI – Center for the Study of Language and Information, Stanford University, 1987. Lecture Notes no:7.
52. R. Hale. Temporal logic programming. In A. Galton, editor, *Temporal Logics and Their Applications*, pages 91–119. Academic Press, 1987.
53. C. Hewitt and Gul Agha. Guarded Horn clause languages: Are they deductive and logical? In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 650–657. ICOT, 1988.
54. Tomas Hrycej. Temporal Prolog. In *Proc. of the European Conference on Artificial Intelligence*, pages 296–301, Munich, Germany, 1988.
55. Tomas Hrycej. A temporal extension of Prolog. *Journal of Logic Programming*, 15(1&2):113–145, January 1993.
56. G. E. Hughes and M. J. Creswell. *An Introduction to Modal Logic.* Methuen and Co Ltd, London, 1968.
57. P. Jackson, H. Reichgelt, and F. van Harmelen, editors. *Logic-Based Knowledge Representation.* MIT Press, 1989.
58. J. Jaffar and J-L. Lassez. Constraint logic programming. In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, 1987. ACM Press.
59. Max I. Kanovich. Linear logic as a logic of computations. In *Proc. of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 200–210, Santa Cruz, Calif, June 1992. IEEE Computer Society Press.
60. S. Kono, T. Aoyagi, M. Fujita, and H. Tanaka. Implementation of temporal logic programming language Tokio. In E. Wada, editor, *Logic Programming'85*, volume 221 of *LNCS*, pages 138–147. Springer-Verlag, 1986.
61. R. A. Kowalski. Predicate logic as programming language. In *Proceedings of IFIP'74*, pages 569–574, Amsterdam, 1974. North–Holland.
62. Valentinas Kriaučiukas. Non-classical models for logic programs. In *PDK'91: International Workshop on Processing Declarative Knowledge*, volume 567 of *LNAI*, pages 179–190. Springer-Verlag, 1991.
63. Fred Kroger. *Temporal Logic of Programs.* Springer-Verlag, Berlin Heidelberg, 1987.
64. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.
65. Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In Boyer and Moore, editors, *Correctness Problem in Computer Science*, pages 215–273. Academic Press, 1981.
66. Fujita Masahiro et al. Using the temporal logic programming language Tokio for algorithm description and automatic CMOS gate array synthesis. In E. Wada, editor, *Logic Programming'85*, volume 221 of *LNCS*, pages 246–255. Springer-Verlag, 1986.
67. W. H. Mitchell. Intensional Horn clause logic as a programming language – it's use and implementation. Master's thesis, Department of Computer Science and Engineering, Arizona State University, Tempe, Arizona, 1988.
68. W. H. Mitchell and A. A. Faustini. The intensional logic language InTense. In *Proceedings of the 1989 International Symposium on Lucid and Intensional Programming*, Arizona State University, May 8 1989.

69. R. Montague. *Formal Philosophy, Selected Papers of Richard Montague*. Yale University Press, 1974. edited by Richmond Thomason.
70. B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.
71. Hans Jürgen Ohlbach. A resolution calculus for modal logics. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction*, pages 500–516. Springer-Verlag, 1988.
72. M. A. Orgun. On temporal deductive databases. Technical Report 93-140C, Department of Computing, Macquarie University, Sydney, NSW 2109, Australia, December 1993.
73. M. A. Orgun and W. Du. Multi-dimensional logic programming. In *Proceedings of ICCI'94: The Sixth International Conference on Computing and Information*, Trent University, Peterborough, Ontario, Canada, May 26–28 1994. To appear.
74. M. A. Orgun and W. W. Wadge. Chronolog: A temporal logic programming language and its formal semantics. Department of Computer Science, University of Victoria, Victoria, B.C., Canada, January 1988.
75. M. A. Orgun and W. W. Wadge. A theoretical basis for intensional logic programming. In *Proceedings of the 1988 International Symposium on Lucid and Intensional Programming*, pages 33–49, Sidney, B.C., Canada, April 7-8 1988.
76. M. A. Orgun and W. W. Wadge. Theory and practice of temporal logic programming. In L. Fariñas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*, pages 23–50. Oxford University Press, 1992.
77. M. A. Orgun and W. W. Wadge. Towards a unified theory of intensional logic programming. *Journal of Logic Programming*, 13(4):413–440, August 1992.
78. M. A. Orgun and W. W. Wadge. Chronolog admits a complete proof procedure. In *Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*, pages 120–135, Université Laval, Québec City, Québec, Canada, April 26–27 1993.
79. M. A. Orgun and W. W. Wadge. Extending temporal logic programming with choice predicates non-determinism. To appear in *Journal of Logic and Computation*, 1994.
80. M. A. Orgun, W. W. Wadge, and W. Du. Chronolog($\mathcal{Z}$): Linear-time logic programming. In *Proceedings of ICCI'93: The Fifth International Conference on Computing and Information*, Laurentian University, Sudbury, Ontario, Canada, May 27–29 1993. IEEE Computer Society Press.
81. R. Ramanujam. Semantics of distributed logic programs. *Theoretical Computer Science*, 68:203–220, 1989.
82. N. Rescher and A. Urquhart. *Temporal Logic*. Springer-Verlag, 1971.
83. D. W. Rolston. Chronolog: A pure tense-logic-based infinite-object programming language. Department of Computer Science and Engineering, Arizona State University, Tempe, Arizona, August 1986.
84. D. W. Rolston. Toward a tense-logic-based mitigation of the frame problem. In F. M. Brown, editor, *Proceedings of the 1987 Workshop on the Frame Problem in AI*, Lawrence, Kansas, April 1987. Morgan Kaufmann, Los Altos, Calif.
85. D. W. Rolston. *Parallel Logic Programming Using an Intensional Model of Computation*. PhD thesis, Department of Computer Science and Engineering, Arizona State University, Tempe, Arizona, 1992.
86. F. Sadri. Three approaches to temporal reasoning. In A. Galton, editor, *Temporal Logics and Their Applications*, pages 121–168. Academic Press, 1987.

87. Y. Sakakibara. Programming in modal logic: An extension of PROLOG based on modal logic. In E. Wada, editor, *Logic Programming'86*, volume 264 of *LNCS*, pages 81–91. Springer-Verlag, 1987.
88. Takahashi Sakuragawa. Temporal Prolog. In *Proc. of RIMS Conference on Software Science and Engineering*. Springer-Verlag, 1987.
89. P. Schroeder-Heister, editor. *International Workshop on Extensions of Logic Programming*, volume 475 of *LNAI*. Springer-Verlag, 1991.
90. D. Scott. Advice on modal logic. In K. Lambert, editor, *Philosophical Problems in Logic*, pages 143–173. D.Reidel Publishing Company, 1970.
91. E. Shapiro. The family of concurrent logic programming languages. In Friedrich L. Bauer, editor, *Logic, Algebra, and Computation*, pages 359–485. Springer-Verlag, Berlin Heidelberg, 1991.
92. Yoav Shoham. *Reasoning About Change*. MIT Press, 1988.
93. P. Smets, A. Mamdani, D. Dubois, and H. Prade, editors. *Non-Standard Logics for Automated Reasoning*. Academic Press, 1988.
94. A. Szalas. Concerning the semantic consequence relation in first-order temporal logic. *Theoretical Computer Science*, 47:329–334, 1986.
95. Alexander Tuzhilin and James Clifford. A temporal relational algebra as a basis for temporal relational completeness. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 13–23, Brisbane, Australia, August 13–16 1990. Morgan Kaufmann Publishers Inc., Los Altos, Calif.
96. M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23:733–42, 1976.
97. M.H. van Emden and M.A. Nait Abdallah. Top-down semantics of fair computations of logic programs. Technical Report CS–84–27, Department of Computer Science, University of Waterloo, October 1984.
98. M. Y. Vardi. A temporal fixpoint calculus. In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, pages 250–259, San Diego, Calif, January 1988. ACM Press.
99. W. W. Wadge. Tense logic programming: a respectable alternative. Department of Computer Science, University of Victoria, Victoria, B.C., Canada, 1985.
100. W. W. Wadge. Tense logic programming: a respectable alternative. In *Proceedings of the 1988 International Symposium on Lucid and Intensional Programming*, pages 26–32, Sidney, B.C., Canada, April 7-8 1988.
101. W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.
102. David H. D. Warren. An abstract Prolog instruction set. Technical report, SRI International, Menlo Park, Calif, October 1983.
103. R. Wojcicki. *Theory of Logical Calculi*. Kluwer Academic Publishers, 1988.
104. K. Zhang and M. A. Orgun. Parallel execution of temporal logic programs using dataflow computation. In *Proceedings of ICCI'94: The Sixth International Conference on Computing and Information*, Trent University, Peterborough, Ontario, Canada, May 26–28 1994. To appear.
105. Kang Zhang and R. Thomas. DIALOG – A dataflow model for parallel execution of logic programs. *Future Generation Computer Systems*, 6(4):373–388, September 1991.

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