J. Symbolic Computation (1989) **8**, 277–295

Temporal Logic Programming

MARTÍN ABADI† AND ZOHAR MANNA‡

Computer Science Department, Stanford University, Stanford, CA 94305, *USA*

(*Received* 15 *September* 1987)

Temporal logic, often used as a specification language for programs, can serve directly as a programming language. We propose a specific programming language, TEMPLOG, which extends the classical PROLOG-like languages to include temporal operators. PROLOG progams are collections of classical *Horn clauses* and they are efficiently interpreted by *SLD-resolution*. Similarly, TEMPLOG programs are collections of *temporal Horn clauses*, and we interpret them with *temporal SLD-resolution*, a restricted form of a general temporal resolution method.

1. Introduction

Temporal logic is a formalism for reasoning about a changing world. Because the concept of time is directly built into the formalism, temporal logic has been widely used as a specification language for programs where the notion of time is central (e.g., Pnueli, 1981; Lamport, 1983). For the same reason, it is natural to write such programs directly in temporal logic. We describe a temporal logic programming language, TEMPLOG. TEMPLOG extends classical logic programming languages, such as PROLOG (e.g., Clocksin & Mellish, 1981), to include programs with temporal constructs. A PROLOG program is a collection of classical *Horn clauses*. A TEMPLOG program is a collection of *temporal Horn clauses*, that is, Horn clauses with certain temporal operators. An efficient interpreter for PROLOG is based on *SLD-resolution* (e.g., Lloyd, 1983). In previous works we described a resolution system for the full first-order temporal logic (Abadi & Manna, 1985, 1986); we base an interpreter for TEMPLOG on a restricted form of our temporal resolution system, *temporal SLD-resolution*.

Let us consider two simple examples of temporal-logic programs.

EXAMPLES:

- Fibonacci numbers:

  We treat the Fibonacci numbers as a sequence in time,

  $$0, 1, 1, 2, 3, 5, \ldots,$$

  where the first number in the sequence is 0, the next number is 1, and at each time the appropriate number is obtained by adding the two previous numbers. We use the unary predicate symbol *fib*; informally, we want *fib*(x) to hold at time n if x is the

† Present address: Martin Abadi, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA.

‡ Also with the Computer Science Department, The Weizmann Institute of Technology, Rehovot, Israel.

0747–7171/89/090277+19 $03.00/0

n-th Fibonacci number. The sequence of Fibonacci numbers is computed with the temporal program

$$\begin{aligned} &\mathit{fib}(0) \leftarrow, \\ &\bigcirc \mathit{fib}(1) \leftarrow, \\ &\bigcirc\bigcirc \mathit{fib}(x) \Leftarrow \mathit{fib}(y),\ \bigcirc \mathit{fib}(z),\ x \text{ is } y+z, \end{aligned}$$

which may be read

- 0 has property *fib* at time 0 (now);
- 1 has property *fib* at time 1 (at the next time instant);
- for any time n, if y has property *fib* at time n, z has property *fib* at time $n+1$, and $x = y+z$, then x has property *fib* at time $n+2$.

When the single call

$$\mathit{fib}(x)$$

is made, the program proves that

$$\mathit{fib}(0),\quad \bigcirc \mathit{fib}(1),\quad \bigcirc\bigcirc \mathit{fib}(1),\quad \bigcirc\bigcirc\bigcirc \mathit{fib}(2),\quad \ldots$$

while it returns the whole sequence of Fibonacci numbers, that is,

$$0, 1, 1, 2, \ldots$$

- Memory operations:

We define a contents operation for a storage device, in such a way that if the data d is written at address a, *write*(a, d), then d is at a, *in*(a, d), until some other data d' is written at the same location, *write*(a, d'). We assume that the *write* operation is defined elsewhere, and that the initial contents of the device are also given.

$$\begin{aligned} &\bigcirc \mathit{in}(a, d) \Leftarrow \mathit{write}(a, d), \\ &\bigcirc \mathit{in}(a, d) \Leftarrow \mathit{in}(a, d),\ \neg \mathit{write}(a, d'). \end{aligned}$$

In words, a contains d at time $n+1$ if either d was just written at a at time n, or a contained d at time n and there have been no modifications.

If a_0 is an address, the call

$$\mathit{in}(a_0, x)$$

yields the sequence of values stored at a_0. ■

PROLOG-like programs for similar purposes typically involve terms to refer to time or, alternatively, to the state of the computer and the world. For instance, *fib*(x, n) may be defined classically to mean that x is the Fibonacci number for time n, and *in*(a, d, n) may be defined classically to mean that d is stored at address a at time n. In our temporal approach, time is expressed directly by the logic. The direct treatment of time makes some temporal programs easier to understand logically. For instance, we can write simple and logically clear programs that do not terminate, or that interact with a surrounding world that changes because of the actions of concurrent processes.

Classical resolution with explicit time parameters can simulate temporal resolution; therefore, the interpretation of PROLOG-like programs with explicit time parameters can resemble that of the corresponding temporal programs. However, as we show in section 7, this simulation typically involves additional steps for reasoning about time.

Several other temporal logic programming languages have been proposed and used, such as TEMPURA (Moszkowski, 1984), TOKIO (Fujita *et al.*, 1986), two different TEMPORAL PROLOGS (Gabbay, 1987; Sakuragawa, 1989), and Wadge's (1985) language. Each language contains a different repertoire of temporal operators and has a semantics of a different style. Even though these languages are all based on temporal logic, their interpretation does not exploit temporal proof techniques. The proposed interpreters that use proof methods basically rely on a translation into PROLOG and on classical proof techniques. In contrast, we describe a purely temporal approach based on temporal SLD-resolution. A more detailed comparison is given in section 8.

In the next section we present the variant of temporal logic we will consider. In section 3 we give a declarative semantics for temporal programs. In section 4 we point out difficulties in the execution of some temporal constructs and define a fragment of TEMPLOG; we discuss the interpretation of programs in this fragment of TEMPLOG in section 5. We define the full language in section 6; we discuss the interpretation mechanisms for the full TEMPLOG language in section 7.

In a recent paper, Baudinet (1989) provides a fix point semantics for TEMPLOG, proves the completeness of the interpreter, and studies the expressive power of the language.

2. Temporal logic

The temporal logic we will be working with is first-order temporal logic, FTL. The formalism of FTL is that of the predicate calculus, with the additional temporal operators $\bigcirc$ ("next"), $\Box$ ("always"), $\Diamond$ ("eventually"), $\mathcal{U}$ ("until"), and $\mathcal{P}$ ("precedes"). In the intended models time is discrete, linear, and extends infinitely toward the future.

For formulas u and v,

- $\bigcirc u$ means "u is true at the next time instant";
- $\Box u$ means "u is always true (from now on)";
- $\Diamond u$ means "u is eventually true"; that is, $\Diamond u \equiv \neg\Box\neg u$;
- $u\,\mathcal{U}\,v$ means "u is true until v is true"; in particular, u is true forever if v is never true (therefore, $\mathcal{U}$ is often called "weak until" or "unless");
- $u\,\mathcal{P}\,v$ means "u precedes v"; in particular, v may never be true, and then u must eventually be true; that is, $(u\mathcal{P}v) \equiv \neg((\neg u)\mathcal{U}v)$.

We denote a string of k $\bigcirc$'s by $\bigcirc^k$. In languages that make an essential use of function symbols (unlike TEMPLOG), it may be convenient to apply $\bigcirc$ to function symbols and terms as well as to formulas. Thus,

$$\bigcirc(f(t_1, \ldots, t_h))$$

denotes the meaning of $f(t_1, \ldots, t_h)$ at the next time instant, and

$$(\bigcirc f)(t_1, \ldots, t_h)$$

denotes the meaning of f at the next time instant applied to the values of $t_1, \ldots, t_h$ at the present time instant.

For simplicity, we do not discuss the addition of past temporal operators. Note, however, that the future operators that we have introduced can also be interpreted as past operators. For instance, $\Diamond$ can be read "at some point in the past".

Each constant, function, and predicate symbol is either a *rigid* (time-independent) symbol or a *flexible* (time-dependent) symbol. For instance, if *queue* is a flexible constant

symbol, and *length*, *max-length*, and $<$ are rigid function, constant, and predicate symbols, respectively, then

$$\Box[\mathit{length}(\mathit{queue}) < \mathit{max\text{-}length}]$$

may denote that the length of the queue is permanently smaller than some fixed maximum length. Similarly,

$$\Diamond[\mathit{length}(\mathit{queue}) < \mathit{length}(\bigcirc \mathit{queue})]$$

may denote that eventually the current queue is shorter than the queue at the next time instant.

As usual in first-order temporal logics, the meaning of variable symbols does not depend on time. Therefore,

$$\forall x \forall y . [(x = y) \equiv \bigcirc(x = y)]$$

is a valid sentence.

3. Formulas as programs

In this section we give a declarative semantics for temporal logic programs. This semantics applies to all temporal logic formulas, including those that we do not regard as programs because they would be hard to execute.

In classical logic programming, a program P is a sentence (i.e., a formula without free occurrences of variables), which expresses some facts and some rules. A call to the program is a formula c with free variables $x_1, \ldots, x_k$; intuitively, c is a question, or a request for suitable values for the free variables $x_1, \ldots, x_k$. Finally, the substitution

$$\{x_1 \leftarrow t_1, \ldots, x_k \leftarrow t_k\}$$

is a correct output of P for c if P implies that c holds when $x_1, \ldots, x_k$ are taken to equal $t_1, \ldots, t_k$, respectively. For instance, if P is

$$\mathit{even}(0) \wedge \forall x . [\mathit{even}(x) \supset \mathit{even}(x+2)]$$

and c is

$$\mathit{even}(x_1),$$

then the substitution

$$\{x_1 \leftarrow 6\}$$

is one of infinitely many correct outputs.

These definitions are also the basis of our approach to the declarative semantics of temporal programs. Here, however, the meaning of a program P and of a query c may be time-dependent. Thus, different terms may be answers to c at different time instants. This opens several new possibilities in the declarative semantics of programs. For instance, we may interpret c as a request for values to satisfy c in the present, or as a request for values to satisfy c eventually. In this work, we consider a more general scenario, where the system computes a sequence of values to satisfy c at each time instant. As in PROLOG, more than one answer can be given for each time instant if desired. Sometimes we are only interested in certain time instants, and, in practice, we may indicate this with some special notation.

More precisely, given the temporal sentence P, the temporal formula c with free variables $x_1, \ldots, x_k$, and the terms $t_1, \ldots, t_k$, the substitution

$$\theta = \{x_1 \leftarrow t_1, \ldots, x_k \leftarrow t_k\}$$

is an *answer (substitution)* of P for c if the formula

$$P \supset \forall x_1 \ldots \forall x_k . [(x_1 = t_1 \wedge \ldots \wedge x_k = t_k) \supset c] \quad (*)$$

is valid. In the important special case where all the terms $t_1, \ldots, t_k$ under consideration are rigid, this formula is actually equivalent to $P \supset (c\theta)$.

We regard the string "*no*" as a substitution; *no* is an answer of P for c when no other answer exists.

If the substitution θ_i is an answer of P for $\bigcirc^i c$ for all i, then the sequence of substitutions

$$\theta_0, \theta_1, \ldots, \theta_i, \ldots$$

is an *answer (substitution) sequence* of P for c.

Thus, we may regard the temporal sentence P as a *program* and the temporal formula c as a *call*. To *run* the program P with the call c is to find an answer sequence of P for c.

As in classical logic programming languages, some symbols may have predefined interpretations. Typically these are the symbols for the basic operations on conventional data structures, such as numbers and lists. In temporal logic programming, symbols that represent time-dependent information may also be incorporated in a natural way. The definition of answer can then be extended to take into account that some symbols have such intended interpretations. More precisely, we say that θ is an answer of P for c if the formula (*) holds in all models where the predefined symbols have the intended interpretation.

EXAMPLE:

- First-in first-out service:

  Consider the formula

$$\begin{aligned} &\forall x \forall y . [(request(x) \mathscr{P} request(y)) \supset (serve(x) \mathscr{P} serve(y))] \\ &\quad \wedge \\ &\forall x . [\neg serve(x) \mathscr{U} request(x)] \\ &\quad \wedge \\ &\Box \forall x . [request(x) \supset (serve(x) \mathscr{P} \neg busy)], \end{aligned}$$

  where *busy*, *request*, and *serve* are flexible predicate symbols. That is, if request x precedes request y then x is served before y; a request is not served until it is made; and, if a request is made then it is served before the server is not busy.

  Presumably the values for *busy* and *request* would be obtained from some lower system layer to which the answers for *serve* could be passed back. If all requests are distinct, then this formula could be construed as a simple routine to serve a stream of requests on a first-request first-serve basis. More precisely, the call

$$serve(x)$$

  may produce a sequence of substitutions of the form

$$\{x \leftarrow t_1\}, \{x \leftarrow t_2\}, \ldots, \{x \leftarrow t_i\}, \ldots,$$

  where t_i is the request served at time i. ■

As this example illustrates, temporal logic programs provide high-level descriptions of computations. In the remainder of the paper, these descriptions are restricted syntactically in order to make an implementation more viable. We define the tractable class of formulas that constitute the programming language TEMPLOG. We present a fragment of TEMPLOG at first (section 4), and the full language later (section 6).

4. A fragment of TEMPLOG

Some temporal constructs do not seem amenable to efficient execution. For instance, consider the proposed program

$$P_0\colon\ (p(a) \wedge q(a)) \wedge \Box[(p(a) \wedge q(a)) \supset \bigcirc(p(a) \wedge q(a))],$$

where p and q are flexible predicate symbols and a is a rigid constant symbol. The call

$$c_0\colon\ \Box p(x)$$

seems perfectly reasonable. In fact, we would expect to obtain the answer sequence

$$\{x \leftarrow a\}, \{x \leftarrow a\}, \ldots$$

The computation must involve an implicit inductive proof of

$$\Box(p(a) \wedge q(a)),$$

that is, $p(a) \wedge q(a)$ holds initially, and if it holds at time n then it also holds at time $n+1$, hence it always holds. The decision to use induction and the choice of the inductive formula

$$p(a) \wedge q(a)$$

can be handled by a general theorem prover. However, we would not expect efficient programming systems to carry out such inductive proofs routinely. Our programming language does not include programs and calls that require such intricate executions.

The syntax of temporal logic can be restricted in a number of natural ways to obtain reasonably efficient programming languages. We present just one such restriction in this section and relax it in section 6. Throughout, we imitate the PROLOG style of programming. The only function symbols with no predefined interpretation that we allow are rigid constant symbols.

A formula is called *next-atomic* if it is of the form $\bigcirc^k A$, where $k \geqslant 0$ and A is some atomic formula, that is, a formula without connectives, temporal operators, and quantifiers.

An *initial (temporal Horn) clause* is a sentence of the form

$$\forall x_1 \ldots \forall x_k . [A_1 \wedge \ldots \wedge A_n \supset B],$$

where $A_1, \ldots, A_n, B$ are next-atomic formulas. It is convenient to denote such a formula by

$$B \leftarrow A_1, \ldots, A_n.$$

We call B and $A_1, \ldots, A_n$ the *head* and the *body* of the clause, respectively.

A *permanent (temporal Horn) clause* is a sentence of the form

$$\forall x_1 \ldots \forall x_k . \Box[A_1 \wedge \ldots \wedge A_n \supset B],$$

where $A_1, \ldots, A_n, B$ are next-atomic formulas. It is convenient to denote such a formula by

$$B \Leftarrow A_1, \ldots, A_n.$$

Again, B and $A_1, \ldots, A_n$ are the *head* and the *body* of the clause, respectively.

A TEMPLOG *program* is a conjunction of initial and permanent temporal Horn clauses. A *call* is a conjunction of next-atomic formulas. (As usual, the conjunction symbol $\wedge$ is often replaced with a comma.)

EXAMPLES:

- Block manipulation:

  We wish to reverse a (possibly infinite) stack of blocks s in order to construct a new stack of blocks, and to output the stack under construction at each time. For example, if s is ABC (A is the top block), and *empty* denotes the empty stack, then the output would be the sequence of stacks

  $$empty, A, BA, CBA, CBA, CBA, \ldots$$

  Suppose that the predefined rigid symbols *empty*, *top*, *pop*, and *push* are available; $top(x, y)$ means that x is the top block in the stack y, $pop(x, y)$ means that y is the stack x without its top block, and $push(x, y, z)$ means that the result of putting the block x on top of y is z.

  We define $r(x, y, z)$ to mean that x is the stack constructed ("reversed") from z so far and that y is the part of z not yet considered (the "rest"):

  $$r(empty, z, z) \leftarrow,$$
  $$\bigcirc r(x, y, z) \Leftarrow r(u, w, z), top(v, w), pop(w, y), push(v, u, x),$$
  $$\bigcirc r(x, empty, z) \Leftarrow r(x, empty, z),$$

  that is, initially the reversed stack is empty and the rest is identical to the original stack; x and y are the reversed stack and the rest obtained from z at time $n+1$ if u and w are the reversed stack and the rest obtained from z at time n, v is the top block of w, y is the result of popping w, and x is the result of putting v on u; x and *empty* are the reversed stack and the rest obtained from z at time $n+1$ if x and *empty* are the reversed stack and the rest obtained from z at time n. Then we define $r^*(x, z)$ to mean that x is the reversed stack obtained from z so far:

  $$r^*(x, z) \Leftarrow r(x, y, z).$$

  The call $r^*(x, s)$ produces the desired output, that is, a sequence of substitutions of the form

  $$\{x \leftarrow t_1\}, \{x \leftarrow t_2\}, \ldots, \{x \leftarrow t_i\}, \ldots,$$

  where t_i is the reversed stack obtained from s at time i.

- System maintenance and backups:

  Temporal logic programs may look ahead. We can use this capability, for instance, to schedule system backups. Suppose that *maintenance* is a predefined flexible predicate symbol; $maintenance(x)$ denotes that computer system x undergoes maintenance. We define the flexible predicate symbol *backup* so that $backup(x)$

always holds right before *maintenance*(x) holds:

$$backup(x) \Leftarrow \bigcirc maintenance(x),$$

that is, if *maintenance*(x) holds at time $n+1$ then *backup*(x) holds at time n. If m is the name of a machine, *backup*(*m*) yields the empty substitution right before *maintenance*(*m*) holds, and *no* at all other times.

- Interaction between processes:

The simple fragment of TEMPLOG we have just described suffices to program many of the processes that one would usually specify in temporal logic, such as processes that interact with other processes. For instance, consider a process that receives some inputs of the form $\langle s, a\rangle$, where a is a process name and s is a message. Then it sends the message s to the process a. If s is not received from a (as an acknowledgement) after a certain time, then s is sent to a again.

Suppose that the flexible symbols *maxT* and *input* have been defined: *maxT* denotes the maximum time the process would currently wait for an acknowledgement; *input*(S, a) holds at time n if S is the set of messages that have arrived from a at time n.

We define *wait*(s, a, t) to mean that the process has been waiting for an acknowledgement for message s from a for t units of time:

$$wait(s, a, 0) \Leftarrow send(s, a),$$
$$\bigcirc wait(s, a, t) \Leftarrow t' \text{ is } t-1,\ wait(s, a, t'),\ input(S, a),\ s \notin S,$$

that is, the process has been waiting for 0 units of time when the message is first sent; if at time n it has been waiting for $t-1$ units of time and the input from a at time n contains no acknowledgement, then at time $n+1$ it has been waiting for t units of time.

Then we define *send*(s, a) to hold when the message s should be sent to a:

$$send(s, a) \Leftarrow input(S, a'),\ \langle s, a\rangle \in S,$$
$$send(s, a) \Leftarrow wait(s, a, maxT),$$

that is, s is sent to a whenever the input from some process a' contains $\langle s, a\rangle$, and s is sent to a again whenever the process has been waiting for an acknowledgement for s from a for *maxT* units of time. If a_0 is a process name, the call

$$send(s, a_0)$$

produces a sequence of messages forwarded to a_0.

- Motion:

Consider the program

$$step(right) \leftarrow ahead(goal),$$
$$\bigcirc step(right) \Leftarrow \bigcirc ahead(goal),\ step(left),$$
$$\bigcirc step(left) \Leftarrow \bigcirc ahead(goal),\ step(right).$$

If *right*, *left*, *goal*, and *ahead* are predefined (or defined appropriately by some other formulas), then this program and the call *step*(x) yield a simulation of a sequence of steps toward a goal. Intuitively, a step with the right leg is taken at the initial time if

the goal is ahead at the initial time; a step with the right leg is taken at time $n+1$ if the goal is still ahead at time $n+1$ and a step with the left leg was taken at time n; and a step with the left leg is taken at time $n+1$ if the goal is still ahead at time $n+1$ and a step with the right leg was taken at time n. ■

We discuss occurrences of additional temporal constructs in section 6; many other temporal constructs can be allowed as abbreviations, much as in Sakuragawa's TEMPORAL PROLOG. Moreover, the basic temporal logic programming language we describe can be extended in many of the ways PROLOG has been extended (e.g., DeGroot & Lindstrom, 1986), for example to include negation and to handle more general formula structures. In particular, defined rigid function symbols can be introduced as in TABLOG (Malachi *et al.*, 1986), for instance. As for flexible function symbols, substitutions into modal contexts give rise to technical difficulties (Abadi, 1987). Finally, as in an extended version of TEMPORAL PROLOG for real-time process control (Hattori *et al.*, 1986), the use of modules and other modern programming techniques is essential for writing significant programs.

5. The interpretation of TEMPLOG programs

Classical logic programs are often interpreted with classical resolution systems. Similarly, we use our general temporal resolution system (Abadi & Manna, 1985; 1986) to evaluate temporal logic programs. The general resolution method handles all temporal formulas, with arbitrary combinations of temporal operators and quantifiers. For efficiency, we restrict the general resolution method to handle only TEMPLOG programs. The proof system obtained is a restricted form of the general one, analogous to classical SLD-resolution. (We only consider a particular atom-selection rule for sake of simplicity.)

Given a call $C_1, \ldots, C_m$, we consider the sequence of *goals*

$$\begin{array}{c} C_1, \ldots, C_m, \\ \bigcirc C_1, \ldots, \bigcirc C_m, \\ \vdots \end{array}$$

The i-th substitution θ_i in the answer sequence is obtained from a resolution proof of the goal

$$\bigcirc^i C_1, \ldots, \bigcirc^i C_m$$

from the clauses of the program. The proof is constructed with the following two rules:

- The resolvent of the goal

  $$\bigcirc^{i_1} A_1, \ldots, \bigcirc^{i_k} A_k$$

  and the initial clause

  $$\bigcirc^{i_1} B \leftarrow \bigcirc^{j_1} B_1, \ldots, \bigcirc^{j_n} B_n$$

  is the new goal

  $$\bigcirc^{j_1} B_1\theta, \ \ldots, \ \bigcirc^{j_n} B_n\theta, \ \bigcirc^{i_2} A_2\theta, \ \ldots, \ \bigcirc^{i_k} A_k\theta,$$

  where θ is a most-general unifier of the atomic formulas A_1 and B, with the variables of the initial clause standardised apart from those of the goal. (Of course, the goal and the clause have no resolvent if no suitable θ exists.)

  In the important special case where the predicate symbol in A_1 and B is rigid, the

number of $\bigcirc$'s in $\bigcirc \ldots \bigcirc A_1$ and $\bigcirc \ldots \bigcirc B$ is totally ignored, since the deduction really is time-independent.

- Similarly, if $i_1 \geqslant j$, the resolvent of the goal

$$\bigcirc^{i_1} A_1, \ldots, \bigcirc^{i_k} A_k$$

and the permanent clause

$$\bigcirc^{j} B \Leftarrow \bigcirc^{j_1} B_1, \ldots, \bigcirc^{j_n} B_n$$

is the new goal

$$\bigcirc^{j_1+(i_1-j)} B_1\theta, \ \ldots, \ \bigcirc^{j_n+(i_1-j)} B_n\theta, \ \bigcirc^{i_2} A_2\theta, \ \ldots, \ \bigcirc^{i_k} A_k\theta,$$

where θ is a most-general unifier of the atomic formulas A_1 and B, with the variables of the permanent clause standardised apart from those of the goal.
Intuitively, we have the leeway of requiring that i_1 be greater than j (and not equal, as for resolution with initial clauses), because we can extract some $\bigcirc$'s from the $\Box$ implicit in a permanent clause. More precisely,

$$\bigcirc^{j} B \Leftarrow \bigcirc^{j_1} B_1, \ldots, \bigcirc^{j_n} B_n$$

entails

$$\bigcirc^{i_1} B \leftarrow \bigcirc^{j_1+(i_1-j)} B_1, \ldots, \bigcirc^{j_n+(i_1-j)} B_n$$

if i_1 is greater than j.
In the case where the predicate symbol in A_1 and B is rigid,

$$\bigcirc^{j} B \Leftarrow \bigcirc^{j_1} B_1, \ldots, \bigcirc^{j_n} B_n$$

is equivalent to

$$B \leftarrow \Diamond[\bigcirc^{j_1} B_1, \ldots, \bigcirc^{j_n} B_n].$$

We postpone the discussion of permanent clauses with rigid predicate symbols in their heads to section 7, where we show how to handle clauses with bodies where $\Diamond$ occurs.

Thus, we generalize the SLD-resolution strategy to apply to temporal logic programs. For each i, the goal

$$\bigcirc^{i} C_1, \ldots, \bigcirc^{i} C_m$$

is resolved with the clauses in the program until the empty goal is obtained. The proof search strategy is depth-first with backtracking. Clauses are considered in the order in which they appear in the program. As usual, the i-th substitution returned is the composition of the unifiers in the i-th proof, and *no* is returned when the proof search terminates without discovering a proof. If the i-th proof search does not terminate then the $(i+1)$-th proof is never attempted.

EXAMPLE:

- Consider the program to compute Fibonacci numbers that was given in section 1. When the call *fib*(x) is made, the goal *fib*(x) is generated and resolved with the first clause of the program, to obtain the empty goal and output the first substitution $\{x \leftarrow 0\}$.

Next the goal $\bigcirc fib(x)$ is generated. The resolution rule is not applicable with the first clause, hence the second clause is considered. The resolvent of the goal and the second clause is the empty goal, and we obtain the substitution $\{x \leftarrow 1\}$.

Then the goal $\bigcirc\bigcirc fib(x)$ is generated. Now the resolution rule is not applicable with the first and the second clauses, and hence the third clause is considered. The resolvent of the goal and the third clause is $(fib(y), \bigcirc fib(z), x \text{ is } y+z)$. We resolve the first conjunct with the first clause and the second conjunct with the second clause. We obtain the substitutions $\{y \leftarrow 0\}$ and $\{z \leftarrow 1\}$, and add $0+1$ to obtain $\{x \leftarrow 1\}$.

The interpretation of the program continues in this fashion, to yield the sequence of substitutions

$$\{x \leftarrow 0\}, \{x \leftarrow 1\}, \{x \leftarrow 1\}, \{x \leftarrow 2\}, \ldots,$$

and hence the Fibonacci numbers

$$0, 1, 1, 2, \ldots \quad \blacksquare$$

According to this simple approach, each of the substitutions is computed separately. This may give rise to some redundant computations. For instance, in the example presented above, where the sequence of Fibonacci numbers is computed, the computation of each Fibonacci number would require the recomputation of all smaller ones (even exponentially many times!). Fagin (1984) and Wagner-Dietrich & Warren (1986) proposed caching intermediate results in classical logic program evaluations. This technique would significantly speed up the evaluation of temporal logic programs as well.

6. Further TEMPLOG constructs

The fragment of TEMPLOG presented in section 4 is quite general. However, it does not enable us to write directly rules with some complicated temporal expressions, such as "if at some point she becomes president then she must be rich" and "if he is from Norway then he will always be tall for his age." Rules of these forms do arise naturally, though, for instance in querying historical databases (e.g., Snodgrass & Ahn, 1985; Clifford & Tanzel, 1985). We extend TEMPLOG so that these can be easily formulated.

As a first step, we allow $\Box$'s in the heads of initial clauses. If B is a next-atomic formula, and A is a body with free variables $x_1, \ldots, x_l$, we simply regard the clause

$$\Box B \leftarrow A$$

as a convenient abbreviation for the two clauses

$$B \Leftarrow r(x_1, \ldots, x_l),$$
$$r(x_1, \ldots, x_l) \leftarrow A,$$

where r is a fresh rigid predicate symbol.

A more substantial step is to allow $\Diamond$'s in the bodies of clauses. The class of bodies of clauses is now the smallest class of formulas containing all next-atomic formulas and closed under conjunction and application of $\Diamond$. (In contrast, the class of bodies is not closed under application of $\Diamond$ in section 4.) For instance, the formula

$$\Diamond(p \wedge (\Diamond q) \wedge \bigcirc p)$$

is a body, which we may also denote by $\Diamond(p, (\Diamond q), \bigcirc p)$.

Thus, we now write clauses of the forms

$$B \leftarrow A,$$
$$\Box B \leftarrow A,$$
$$B \Leftarrow A,$$

where A is a body, possibly with occurrences of $\Diamond$, and B is a next-atomic formula. As usual, a TEMPLOG program is a conjunction of temporal Horn clauses and a call is a body. Despite the temptation to go one step further and allow arbitrary modal operators in bodies and heads, we do not have a reasonable computational interpretation of $\Diamond$'s in the heads of clauses and $\Box$'s in the bodies of clauses. For instance, we can attribute the difficulties in interpreting the program P_0 and the call c_0 (given in section 4) to the occurrence of $\Box$ in c_0. Similarly, we can attribute the difficulties in interpreting the program

$$P_1\colon\ \Box(\bigcirc p \supset p) \wedge \Diamond p$$

and the call

$$c_1\colon\ p$$

to the occurrence of $\Diamond$ in P_1. The situation is analogous to that in classical logic programming, where it would be inefficient to interpret bodies with universal quantifiers and heads with existential quantifiers.

EXAMPLES:

- Historical queries:

  Classical logic programming languages have been used as query languages for databases. Analogously, a temporal logic programming language may be the basis for a logical query system for databases with some temporal information, such as historical databases.
  For instance, consider the following example, taken from Gadia (1986). We have a database with some facts about the history of a store after a certain initial date, such as who worked in what department and with what salary. We would like to obtain a list of all people x employed in the Toy Department with their salaries y after the initial date and while John was a manager. In temporal logic, we may simply write the call

$$\Diamond \begin{bmatrix} \mathit{manager}(\mathit{John}), \\ \mathit{in\text{-}department}(x, \mathit{Toy}), \\ \mathit{salary}(x, y) \end{bmatrix},$$

  provided the flexible predicate symbols *manager*, *in-department*, and *salary* have been suitably defined. The answers for the present are the desired output.
  Similarly, we may want to know how much John's salary increased when he went from salesman to manager. We write the program

$$\mathit{increase}(x, y) \Leftarrow \mathit{salary}(x, y_1),\ \bigcirc \mathit{salary}(x, y_2),\ y \text{ is } (y_2 - y_1),$$

  that is, the increase of x's salary from time n to time $n+1$ is the difference between

x's salary at time n and x's salary at time $n+1$, and the call

$$\Diamond \begin{bmatrix} salesman(John), \\ \bigcirc manager(John), \\ increase(John, y) \end{bmatrix}.$$

Finally, we may want express that all founding employees are always employees:

$$\Box employee(x) \leftarrow employee(x).$$

- Reachability:

  Consider the following scenario. A vehicle goes through a set of locations. Location y is reachable from location x if the vehicle goes by x and then by y. We assume that the flexible predicate symbol *at* has been defined. Then we may define reachability with the program

$$reachable(x, y) \Leftarrow \Diamond[at(x), \Diamond at(y)]. \quad \blacksquare$$

7. The interpretation of TEMPLOG programs, continued

The interpretation strategy for programs in the full language is basically the same as that for programs in the fragment considered in section 4. We only need to specify how to resolve goals and clauses in the cases where these have new forms. In a preliminary version of this work (Abadi & Manna, 1987) we discussed separately the rules for initial clauses with occurrences of $\Box$; here we prefer to treat these clauses as abbreviations. Thus, we only need to specify how to handle bodies with occurrences of $\Diamond$.

For simplicity, we assume that, as soon as a new goal is generated, all $\bigcirc$'s are pushed inwards, using the valid equivalences

$$\bigcirc\Diamond u \equiv \Diamond\bigcirc u$$

and

$$\bigcirc(u_1 \wedge u_2) \equiv (\bigcirc u_1 \wedge \bigcirc u_2).$$

Our notation is more concise than in section 5; often a single symbol represents a body of arbitrary complexity. Throughout, A and B are atomic formulas with most general unifier θ (after the usual variable renaming), and A', A'', $A_1, \ldots, A_k$, and B' are arbitrary (possibly empty) bodies.

Suppose that we consider the goal $A_1, \ldots, A_k$. As in section 5, if the resolvent of A_1 and the clause C is the body R, and the unifier used in the resolution is θ, then the resolvent of $A_1, \ldots, A_k$ and C is the body $R\theta, A_2\theta, \ldots, A_k\theta$. In particular, when A_1 is next-atomic, we may just use the rules of section 5.

The handling of goals where $\Diamond$ is the main operator (or the main operator of the first conjunct, of course) is somewhat more complicated, but still efficient. When a goal of the form

$$\Diamond[(\Diamond A_1), \ldots, (\Diamond A_k)]$$

appears, we immediately replace it with the equivalent goal

$$(\Diamond A_1), \ldots, (\Diamond A_k).$$

Therefore, we focus on goals of the form

$$\Diamond[A', \bigcirc^i A, A'']$$

where A', $\bigcirc^i A$, A'' has $\bigcirc^i A$ as leftmost next-atomic conjunct.

While it is most orthodox to respect the left-to-right order and to consider $\bigcirc^i A$ after A', this may lead to complications. For instance, suppose that we have the goal

$$\Diamond[(\Diamond p), \bigcirc p]$$

and the fact $\bigcirc^5 p$. If $\Diamond p$ is considered first, we reduce the problem to proving that $\bigcirc p$ holds at some time i, with the constraint $i \leqslant 5$. Unfortunately, this constraint is hard to express succinctly in temporal logic, and it would represent an additional burden even if it were kept implicit, or if it were made explicit in a classical system. On the other hand, if the conjunct $\bigcirc p$ is considered first, we just obtain the new goal

$$\Diamond \bigcirc^{(5-1)} p.$$

Thus, in the following rules, we prefer to consider next-atomic conjuncts first, for simplicity. We replace goals of the form

$$\Diamond[A', \bigcirc^i A, A'']$$

with

$$\Diamond[\bigcirc^i A, A', A''].$$

Therefore, we focus on goals of the form

$$\Diamond[\bigcirc^i A, A'],$$

where $\bigcirc^i A$ is next-atomic.

- If $j \geqslant i$, the resolvent of the goal

  $$\Diamond[\bigcirc^i A, A']$$

  and the initial clause

  $$\bigcirc^j B \leftarrow B'$$

  is the new goal

  $$B'\theta, \bigcirc^{j-i} A'\theta.$$

  Intuitively, we strengthen the goal

  $$\Diamond[\bigcirc^i A, A']$$

  to

  $$\Diamond[\bigcirc^j A, \bigcirc^{j-i} A'],$$

  then to

  $$\bigcirc^j A, \bigcirc^{j-i} A',$$

  and then, since B' suffices to prove that A holds at time j (up to unification), we replace $\bigcirc^j A$ with B'.
- If $j \geqslant i$, the resolvent of the goal

  $$\Diamond[\bigcirc^i A, A']$$

and the permanent clause

$$\bigcirc^j B \Leftarrow B'$$

is the new goal

$$\Diamond[B'\theta, \bigcirc^{j-i} A'\theta].$$

Intuitively, we strengthen the goal

$$\Diamond[\bigcirc^i A, A']$$

to

$$\Diamond[\bigcirc^j A, \bigcirc^{j-i} A'],$$

and then, since B' always suffices to prove $\bigcirc^j A$ (up to unification), we replace $\bigcirc^j A$ with B'.

- On the other hand, if $i \geqslant j$, the resolvent of the goal

$$\Diamond[\bigcirc^i A, A']$$

and the permanent clause

$$\bigcirc^j B \Leftarrow B'$$

is the new goal

$$\Diamond[\bigcirc^{i-j} B'\theta, A'\theta].$$

Intuitively, we first weaken the permanent clause to

$$\bigcirc^i B \Leftarrow \bigcirc^{i-j} B'$$

and then proceed as above.

As usual, in the special case where the predicate symbol in A and B is rigid, the number of $\bigcirc$'s in $\bigcirc \ldots \bigcirc A$ and $\bigcirc \ldots \bigcirc B$ is totally ignored. Also, the goal

$$\Diamond[A', \bigcirc^i A, A'']$$

is interpreted as

$$A, \Diamond[A', A''],$$

and the permanent clause

$$\bigcirc \ldots \bigcirc B \Leftarrow B'$$

is interpreted as

$$B \leftarrow \Diamond B'.$$

Like the rules in section 5, these rules are the basis for a new temporal extension of SLD-resolution. This extension is rather tractable both conceptually and computationally. The similarities between classical SLD-resolution and temporal SLD-resolution suggest that temporal SLD-resolution is reasonably efficient.

Furthermore, temporal SLD-resolution is particularly suitable to deal with temporal programs. More precisely, the classical SLD-resolution approach does not always handle programs with explicit time parameters as well. As we mentioned in section 1, classical resolution with explicit time parameters can simulate temporal resolution, but this simulation typically involves additional steps.

For instance, consider the goal

$$\Diamond\bigcirc\bigcirc p$$

and the rule

$$\bigcirc p \Leftarrow q.$$

We can immediately unify the goal and the head of the rule. In contrast $p(n+2)$ and $p(n'+1)$ are not unifiable. Intermediate deductions are necessary.

Similarly, consider the use of the rule

$$p \Leftarrow \Diamond q.$$

If the rule is translated as

$$p(n) \leftarrow q(n'), n' \geqslant n,$$

the classical interpreters will not use $n' \geqslant n$ until $q(n')$ is proved, and this may lead to some useless proofs of $q(n')$ where n' is too small. On the other hand, if the rule is translated as

$$p(n) \leftarrow n' \geqslant n, q(n'),$$

a proof of $n' \geqslant n$ is attempted immediately; this only succeeds with n and n' instantiated, and many different instantiations may be necessary. The interpretation of TEMPLOG programs gives rise to neither of these problems. Other translations of the rule may yield efficient code, but require more care.

Thus, some extensions would be desirable for temporal reasoning within a classical SLD-resolution framework. These extensions include ad hoc unification mechanisms, for instance to obtain the substitution $\{n \leftarrow n'+1\}$ as unifier of $n+2$ and $n'+1$, and mechanisms to manipulate temporal constraints of the form $n' \geqslant n$. These extensions are directly built into temporal SLD-resolution.

8. Related work

In this section, we briefly describe some languages not based on temporal logic, such as ELEPHANT and LUCID. Then we present some languages based on temporal logic: TEMPURA, TOKIO, two different versions of TEMPORAL PROLOG, and a language proposed by Wadge. Even though these are logic languages, the interpreters described do not exploit efficient temporal proof techniques. In general, either they do not use proof procedures at all, or they rely (at least temporarily) on a translation into PROLOG and the classical resolution engine PROLOG provides. In contrast, TEMPLOG is based on temporal SLD-resolution. Also, the languages differ in which temporal constructs they allow, and, therefore, in which classes of problems they can express naturally. Finally, we discuss the MOLOG system, a general attempt to program in modal logics.

The functional programming language ELEPHANT represents time instants with terms (McCarthy, 1984). Kowalski & Sergot (1985) have proposed a similar approach to logic programming, where events are represented explicitly. Many temporal logic programs can be translated into these languages by adding time parameters. For instance,

$$\bigcirc p \Leftarrow q \quad \text{becomes} \quad p(t+1) \leftarrow q(t).$$

LUCID is a language based on streams (Ashcroft & Wadge, 1985); it is particularly suitable to compute the values of objects on a sequence of time instants iteratively. For instance, the sequence of the powers of 2 is computed by the program

$$First\ I = 1,$$
$$Next\ I = 2 \times I.$$

TEMPLOG differs from LUCID in its theorem-proving roots and methods. In particular, LUCID does not involve unification and backtracking.

TEMPURA is an *interval* temporal logic programming language (Moszkowski, 1984). Programs may include the *chop* operator, denoted by a semicolon. The formula $u; v$ means that u holds for an interval and then v holds for an interval. Execution of a formula is based on constructing a model to satisfy the formula, and not on a proof; unification and backtracking are not central to this construction. For instance, consider the formula

$$quick\text{-}partition(L, pivot);\ serial\text{-}sort\text{-}parts(L, pivot).$$

This formula is part of a quicksort program that sorts a list L using the element *pivot* to break L into two parts. The execution of this formula involves building a sequence of instants where in some initial interval *quick-partition* takes place and then *serial-sort-parts* takes place. In short, the temporal constructs in TEMPURA are essentially a way to express control mechanisms. The values of most predicates do not depend on time.

TOKIO is a temporal logic programming language with unification and backtracking (Fujita *et al.*, 1986). Programs may include the chop operator, and temporal constructs mostly express control strategies, much as in TEMPURA. TOKIO allows only programs where one output is produced based on computations about the future. As an example, the authors gave the rule

$$qs(X) \leftarrow split(X, H, L);\ qs(H),\ qs(L).$$

This rule indicates that, in order to quicksort X, one should first split X into H and L and then quicksort H and L. The execution of this rule would yield a single output, an answer for *qs* in the present. Currently, TOKIO programs are compiled into PROLOG.

Wadge (1985) has described a temporal logic programming language closely related to LUCID. The language is similar to the TEMPLOG fragment of section 4. Unlike LUCID, this language involves unification and backtracking. As in LUCID, the computation for each time instant is based on the computation for the previous one. In particular, we can define the sequence of powers of 2 with a TEMPLOG-like program,

$$First\ power(1) \Leftarrow,$$
$$Next\ power(x) \Leftarrow power(y),\ x\ is\ 2 \times y.$$

Wadge suggested that programs may be interpreted using a translation into PROLOG that introduces time parameters.

In Gabbay's TEMPORAL PROLOG, as in TEMPLOG, there is a distinction between initial clauses and permanent clauses (Gabbay, 1987). However, the two languages can express very different classes of problems, because in TEMPORAL PROLOG only $\Diamond$ and its past analogue ("at some point in the past") occur in the bodies and in the heads of clauses. Neither $\Box$ nor $\bigcirc$ may occur within the clauses. This explains why occurrences of $\Diamond$ in heads do not give rise to some of the problems we encountered in section 6 (e.g., in P_1). Gabbay sketched an operational semantics of TEMPORAL PROLOG, and it seems to suggest

that an interpreter may be based on some temporal proof technique with unification and backtracking. However, this operational semantics does not immediately fully define an interpreter, because it does not account for the linearity of time and because a proof-search strategy is not specified. Furthermore, it is unclear whether any simple proof-search strategy could be added to define a satisfactory interpreter.

In Sakuragawa's (1989) TEMPORAL PROLOG, the meaning of predicates in the future depends on their meaning in the past. More precisely, all clauses are permanent, and past operators may occur in the bodies of clauses while future operators may occur in their heads. Thus, the following two equivalent programs may be written:

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

and

$$\begin{aligned} p &\Leftarrow dangerous(X), \\ p &\Leftarrow \bullet p, \\ alarm &\Leftarrow p \end{aligned}$$

(here $\bullet$ is read "at the previous time instant"). Note that the first program cannot be written in TEMPLOG while the second one can, with just a minor modification:

$$\begin{aligned} p &\Leftarrow dangerous(X), \\ \bigcirc p &\Leftarrow p, \\ alarm &\Leftarrow p. \end{aligned}$$

This language is very expressive and no realistic implementation has been proposed. Sakuragawa suggested putting programs into a normal form and then translating them into PROLOG. The only temporal operator allowed in programs in normal form is $\bullet$. For instance, the first program would be mapped to the second program and then this second program would be compiled into PROLOG. Unfortunately, the transformation into normal form gives rise to inefficient programs. In TEMPLOG we can write and execute directly all programs in normal form (when $\bullet$'s are traded for $\bigcirc$'s), and many others (for instance, programs with $\Diamond$'s in bodies of clauses). We have chosen not to include any constructs without a direct implementation.

Arthaud *et al.* (1986) have studied programming in arbitrary modal logics and have implemented the MOLOG system. Just as we rely on a temporal resolution system to interpret temporal logic programs, Arthaud *et al.* rely on resolution systems for arbitrary modal logics to interpret modal logic programs. In MOLOG, the user chooses a modal logic and defines the rules to handle modal operators. In this sense, MOLOG is a framework rather than a language. For the sake of flexibility, the system guarantees neither the correctness nor the efficiency of the rules given by the user.

We are grateful to Alur Rajeev, Marianne Baudinet, Tom Henzinger, Bengt Jonsson, and Eric Muller for critical reading of the manuscript, to Richard Waldinger and Pierre Wolper for fruitful discussions, to Cynthia Hibbard for editorial help, and to the anonymous referees for useful comments. This research was supported in part by the National Science Foundation under grants DCR-84-13230 and CCR-88-12595, by the United States Air Force Office of Scientific Research under contracts AFOSR 87-0149 and 88-0281, and by the Defense Advanced Research Projects Agency under Contract N00039-84-C-0211.

References

Abadi, M. (1987). *Temporal-Logic Theorem Proving*, PhD Thesis, Computer Science Department, Stanford University.

Abadi, M., Manna, Z. (1985). Nonclausal temporal deduction. In *Logics of Programs* (ed. R. Parikh), Springer-Verlag LNCS 193, pp. 1–15.

Abadi, M., Manna, Z. (1986). A timely resolution. *Symposium on Logic in Computer Science*, pp. 176–186.

Abadi, M., Manna, Z. (1987). Temporal logic programming. *Fourth Symposium on Logic Programming*, pp. 4–16.

Arthaud, R., Bieber, P., Fariñas del Cerro, L., Henry, J., Herzig, A. (1986). MOLOG: Manuel d'utilisation, L.S.I.—Université Paul Sabatier, Toulouse, February 1986.

Ashcroft, E., Wadge, W. (1985). LUCID, *the Dataflow Programming Language*, Academic Press, London.

Baudinet, M. (1989). Temporal logic programming is complete and expressive. *Symp. Principles of Programming Languages*, Austin, TX.

Clifford, J., Tanzel, A. U. (1985). On an algebra for historical relational databases: two views. *ACM-Sigmod International Conference on Management of Data*, pp. 247–265.

Clocksin, W., Mellish, C. (1981). *Programming in PROLOG*, Springer-Verlag, Berlin.

DeGroot, D., Lindstrom, G., eds. (1986). *Logic Programming: Functions, Relations, and Equations*, Prentice Hall, Englewood Cliffs, NJ.

Fagin, B. (1984). Issues in caching PROLOG goals, Report No. UCB-CSD-84-204, Computer Science Division (EECS), University of California at Berkeley, November 1984.

Fujita, M., Kono, S., Tanake, H., Moto-oka, T. (1986). TOKIO: logic programming language based on temporal logic and its compilation to PROLOG. *Third International Conference on Logic Programming*.

Gabbay, D. (1987). *Modal and temporal logic programming*, unpublished draft.

Gadia, S. (1986). Weak temporal relations. *ACM-Sigmod Conference on the Principles of Database Systems*, pp. 70–77.

Hattori, T., Nakajima, R., Sakuragawa, T., Niide, N., Takenaka, K. (1986). RACCO: a modal-logic programming language for writing models of real-time process-control systems. Report No. RIMS-558, Research Institute for Mathematical Sciences, Kyoto University, December.

Kowalski, R., Sergot, M. (1985) A logic-based calculus of events. Unpublished manuscript.

Lamport, L. (1983). Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, **5**, April 1983, pp. 190–222.

Lloyd, J. W. (1983). *Foundations of Logic Programming*, Springer-Verlag, Berlin.

Moszkowski, B. (1984). Executing Temporal Logic Programs, Technical Report No. 55, Computer Laboratory, University of Cambridge, August.

McCarthy, J. (1984). The ELEPHANT Language for proving and maybe even programming, unpublished manuscript, Computer Science Department, Stanford University.

Malachi, Y., Manna, Z., Waldinger, R. (1986). *Logic Programming: Functions, Relations, and Equations* (D. DeGroot & G. Lindstrom, eds.), Englewood Cliffs, NJ: Prentice-Hall, pp. 365–394.

Pnueli, A. (1981). The temporal semantics of concurrent programs. *Theoretical Computer Science*, **13**, 45–60.

Snodgrass, R., Ahn, I. (1985). A taxonomy of time in databases, *ACM–Sigmod International Conference on Management of Data*, pp. 236–246.

Sakuragawa, T. (1989). TEMPORAL PROLOG. *RIMS Conference on Software Science and Engineering*, Springer-Verlag LNCS, In press.

Wadge, W. (1985). Tense logic programming: a sane alternative. Unpublished manuscript.

Wagner-Dietrich, S., Warren, D. S. (1986). Extension tables: memo relations in logic programming. Technical Report 86/18, Computer Science Department, SUNY at Stony Brook.