Chronolog($\mathbb{Z}$): Linear-time Logic Programming*

Mehmet A. Orgun  William W. Wadge  Weichang Du

Abstract

This paper introduces Chronolog($\mathbb{Z}$), a logic programming language based on a discrete linear-time temporal logic with unbounded past and future. Chronolog($\mathbb{Z}$) is suitable for applications involving the notion of dynamic change, such as modeling non-terminating computations, the simulation of sequential circuits, and temporal databases. The execution of the programs of the language is based on a resolution-type proof procedure called TiSLD-resolution. A modular extension of Chronolog($\mathbb{Z}$) is proposed which can be used to model objects with internal memory.

1 Introduction

Temporal logic has been successfully used as a formalism in concurrent program specification and verification, modeling temporal databases, and various forms of temporal reasoning. In temporal logic [15], the meanings of formulas vary depending on an implicit time parameter and elements from different moments in time can be combined through the use of temporal operators, not by explicit references to time. More recently, a number of logic programming languages based on diverse temporal logics have been proposed: Tempura [12] and Tokio [2] are based on interval logic; Templog [1] and Chronolog [13] are based on linear-time temporal logics and Temporal Prolog [9] is based on linear and branching-time temporal logics.

In Tempura, programs are systematically transformed into a sequence of state descriptions over an interval that satisfies the original program [12]. Templog [1], Temporal Prolog [9] and Chronolog [13] are extensions of logic programming in which temporal logic programs are executed to obtain answers by the use of resolution-type proof procedures. Tokio has a mixed execution mechanism [2].

There are also several attempts at developing semantics of temporal logic languages. Baudinet showed [3] that Templog’s proof procedure is sound and complete; and established the equivalence of the declarative and the operational semantics of Templog programs. Orgun and Wadge provided [13, 14] the semantics of Chronolog programs and outlined a complete proof procedure for the language. Gabbay [10] defined a resolution procedure for Temporal Prolog and proved its soundness. These results suggest that temporal logic programming is feasible.

This paper in particular introduces the temporal language Chronolog($\mathbb{Z}$), which is an extension of Chronolog [13], based on a linear-time temporal logic with unbounded past and future with the set $\mathbb{Z}$ of integers as the collection of moments in time. The temporal logic of Chronolog($\mathbb{Z}$) has an operator to look into the past as well as the operators $\text{first}$ and $\text{next}$ of Chronolog. The “execution” of Chronolog($\mathbb{Z}$) programs is based on a resolution-type proof procedure called TiSLD-resolution. In [14], it is also known that the declarative and the operational semantics of Chronolog($\mathbb{Z}$) programs coincide.

Consider the following program that defines the predicate $f$ which is true of the $(t+1)$-st

---

Fibonacci number for time $t \geq 0$ and no other.

\[
\begin{align*}
\text{first } f(1). \\
\text{first next } f(1). \\
\text{next next } f(N) &\leftarrow \text{next } f(X), f(Y), \text{ } N \text{ is } X + Y.
\end{align*}
\]

Read all clauses as assertions true at all moments in time. The first two clauses define the first two Fibonacci numbers as 1; and the last clause defines the current Fibonacci number as the sum of the previous two. For example, the answer to the given goal $\leftarrow \text{first next next } f(X)$ is a (substitution) instance of the goal with $X$ replaced by 2.

In the following, we first outline the temporal logic of Chronolog($\mathbb{Z}$). We then discuss several applications of the language, including modeling non-terminating computations and the simulation of sequential circuits. We also propose a modular extension of Chronolog($\mathbb{Z}$) for specifying objects with internal memory.

## 2 Temporal Logic of Chronolog($\mathbb{Z}$)

In the temporal logic of Chronolog($\mathbb{Z}$), the collection of moments in time is the set $\mathbb{Z}$ of integers. The temporal logic offers three temporal operators: first, prev and next. Informally, the temporal operators refer to the initial moment, the previous moment and the next moment in time, respectively. We choose a linear-time temporal logic with unbounded past and future as the underlying logic of Chronolog($\mathbb{Z}$), because certain time-dependent properties are more natural and easier to express with past operators and the symmetry between past and future operators is best expressed in such a temporal logic.

The syntax of the temporal logic of Chronolog($\mathbb{Z}$) extends that of first-order logic with three new formation rules: if $A$ is a formula, so are first $A$, prev $A$ and next $A$. We write next($n$) and prev($n$) for $n$ successive applications of next and prev, respectively. From here on, we refer to the underlying logic as TL.

### 2.1 Temporal Interpretations

The semantics of formulas of TL are provided by temporal interpretations. A temporal interpretation of TL assigns meanings at all moments in time to all basic elements of the language, such as function and predicate symbols, and variables. Interepretations are extended upward to all terms and formulas of TL by a satisfaction relation $|=\cdot$. The meaning of a formula of TL varies in time. However, we restrict the discussion to those temporal interpretations in which the values of variables and function symbols are “rigid”. The value of a rigid term is an invariant of time.

We assume a standard definition of the satisfaction relation $|=\cdot$ in terms of temporal interpretations. In the following, the fact that a formula $A$ is true at a moment $t$ in time in a temporal interpretation $I$ is denoted as $|=_{I,t} A$. The formal semantics of temporal operators of TL are given as follows:

- $|=_{I,t} \text{first } A$ iff $|=_{I,0} A$.
- $|=_{I,t} \text{prev } A$ iff $|=_{I,t-1} A$.
- $|=_{I,t} \text{next } A$ iff $|=_{I,t+1} A$.

where $A$ is a formula, and $t \in \mathbb{Z}$.

If a formula $A$ is true in a temporal interpretation $I$ at all moments in time, we say that $A$ is true in $I$, or $I$ is a model of $A$, and denote this fact as $|= I A$. We regard $\lnot$, $\land$ and $\forall$ as primitives and assume the usual definitions of $\lor$, $\rightarrow$, $\leftrightarrow$ and $\exists$ in terms of these primitives.
2.2 Axioms of Temporal Logic

Let the notation \( \vdash A \) denote the fact that \( A \) is a theorem of TL. The notion of deducibility can be characterized in terms of theoremhood: \( \Gamma \vdash A \) means that the formula \( A \) is deducible from a set \( \Gamma \) of formulas in TL. The following axioms state some of the important properties of the temporal operators. Let \( \nabla \) stand for any of \texttt{first}, \texttt{prev} and \texttt{next}.

1. Temporal operator cancellation rules.
   - \textbf{E1.} \( \nabla (\texttt{first } A) \leftrightarrow \texttt{first } A \).  
   - \textbf{E2.} \( \texttt{next } \texttt{prev } A \leftrightarrow A \).  
   - \textbf{E3.} \( \texttt{prev } \texttt{next } A \leftrightarrow A \).

   The first axiom (E1) states that the initial truths persist. The axioms E2 and E3 capture the fact that \texttt{prev} and \texttt{next} are complete inverses.

2. Temporal operator distribution rules.
   - \textbf{D1.} \( \nabla (A \land B) \leftrightarrow (\nabla A) \land (\nabla B) \).
   - \textbf{D2.} \( \nabla (\neg A) \leftrightarrow \neg (\nabla A) \).

   These axioms state that the temporal operators commute with \( \land \) and \( \neg \).

3. Rigidness of variables.
   - \textbf{V.} \( \nabla (\forall x)(A) \leftrightarrow (\forall x)(\nabla A) \).

   This axiom stipulates that the values of individual variables range over extensions (data values).

4. Rules of inference: we have the following temporal-operator introduction rules.
   - \textbf{R1.} If \( \vdash A \), then \( \vdash \texttt{first } A \).
   - \textbf{R2.} If \( \vdash A \), then \( \vdash \texttt{prev } A \).
   - \textbf{R3.} If \( \vdash A \), then \( \vdash \texttt{next } A \).

   It is straightforward to show the correctness of the axioms and the rules of inference (that is, the soundness of the axiomatic system).

\textbf{Lemma 1 ([14])}. \textit{All of the axioms and the rules of inference are valid with respect to the semantics scheme of TL.}

We assume that the rules of inference given above are extended to consider the notion of deducibility from a set of formulas. The resolution-type proof procedure of Chronolog(Z) is based on these axioms and rules of inference [14].

2.3 Temporal Logic Programs

In programs of Chronolog(Z) such as the Fibonacci program given earlier, program clauses can contain applications of temporal operators to atomic formulas. We call atomic formulas with a number (possibly 0) of applications of temporal operators as \textit{temporal atoms}. All variables in program clauses are assumed to be universally quantified. For convenience, we use upper-case letters for variables and lower-case letters for function and predicate symbols. A temporal logic program of Chronolog(Z) is the conjunction of a set of program clauses regarded as assertions true at all moments in time. Programs of Chronolog(Z) are executed using a resolution-type proof procedure called TiSLD-resolution [14].
We need the notion of a canonical formula: Given a formula $A$, by systematic applications of $R1$, $R2$ and $R3$, the following initial truths can be formed:

- $\text{first } A$ (by $R1$),
- $\text{first prev } A$ (by $R2$, $R1$),
- $\text{first next } A$ (by $R3$, $R1$),
- $\text{first prev prev } A$ (by $R2$, $R2$, $R1$),
- $\text{first next next } A$ (by $R3$, $R3$, $R1$),

and so on. Initial truths obtained from a formula $A$ in the above fashion are called initial instances of $A$.

As the following lemma states, the value of a given formula in a temporal interpretation can be expressed in terms of the values of its canonical instances.

**Lemma 2** ([14]). Let $A$ be a formula and $I$ a temporal interpretation of $TL$. \( \models_I A \) if and only if \( \models_I A_t \) for all canonical instances $A_t$ of $A$.

TiSLD-resolution is applied to a set of canonical program and goal clauses. We assume that, in canonical instances of program clauses, all superfluous applications of temporal operators are eliminated by axioms $E1$–$E3$. For a given temporal logic program, canonical instances of program clauses are obtained by rules $R1$–$R3$ and axioms $D1$ and $D2$. When the canonicality restriction is lifted for goal clauses, they can be open-ended. Open-ended goal clauses are used to initiate non-terminating computations (see below). Note that a thorough exposition to TiSLD-resolution can be found in [14].

### 3 Applications

#### 3.1 Non-terminating Computations

In temporal logic programming, non-terminating computations can be modeled by time-varying predicates. Consider Hamming numbers, which are multiples of 2, 3 and 5 of the form $2^i3^j5^k$ for some $i$, $j$ and $k$.

We write a Chronolog($Z$) program to produce the sequence of Hamming numbers in increasing order without any repetitions. In the program, the $\text{hamming}$ predicate is true of the $(t+1)$-st Hamming number at time $t \geq 0$ and no other.

$$\text{hamming}(X) \leftarrow \text{residue}([X \mid L]).$$

$$\text{first residue}([1]).$$

$$\text{residue}(L_1) \leftarrow$$

$$\text{prev residue}([X \mid L]),$$

$$\text{multiply}(2, X, X_2), \text{multiply}(3, X, X_3),$$

$$\text{multiply}(5, X, X_5), \text{merge}([X_2, X_3, X_5], L, L_1).$$

$$\text{merge}(L_1, L_2, L_3) \leftarrow \cdots$$

$$\text{multiply}(X, Y, Z) \leftarrow \cdots$$

We assume a standard definition for $\text{merge}$ in which two given ordered lists are merged to produce a third ordered list with duplicates removed. The $\text{residue}$ duplicate hols all those Hamming numbers produced but not yet consumed by the $\text{hamming}$ predicate. In the program, the $\text{hamming}$ predicate is specified for the non-negative part of the linear time.
According to the program, the residue predicate is true of the residue list [1] at time 0, which means that, by the first clause, 1 is the first Hamming number. At time 1, by the third clause, 1 is removed from the residue list and then multiples of 1 are merged with the rest of the residue list (the empty list []). Thus the residue predicate represents the residue list [2, 3, 5] at time 1, meaning that the second Hamming number is 2. At time 2, by the third clause, 2 is removed from the residue list and multiples of 2 are merged with the rest of the residue list (the list [3, 5]). The residue predicate represents the residue list [3, 4, 5, 6, 10] at time 2, meaning that the third Hamming number is 3. The computation proceeds along these lines.

The open-ended goal clause ← hamming(N) triggers a non-terminating computation of Hamming numbers. The goal is interpreted as an infinite series of canonical subgoals into the future of the form first hamming(N), ← first next hamming(N), ← first next(2) hamming(N), and so forth. The answers to canonical subgoals obtained by TiSLD-resolution in fact constitute the answers to the original goal [14]. The sequence of answer substitutions for N represents the sequence of Hamming numbers.

An implementation of Chronolog(Z) produces answers to canonical subgoals as long as time and resources permit. The starting point and direction of the computation may be determined by users or detected automatically by the implementation. The hamming predicate depends on the residue predicate; the residue predicate is defined by an initial clause and a program clause with recursive past dependencies. Therefore, a non-terminating computation in the negative direction cannot produce any results.

3.2 Circuit Simulation

Clocksin [5] outlined a logic programming method for specifying and reasoning about sequential circuits. A sequential circuit contains memory elements, such as flip-flops, and the next state of the circuit is a function of its current state plus its inputs. We now write a Chronolog(Z) program to simulate a synchronous 3-bit counter (see Figure 1). The counter consists of two “and” gates and three JK flip-flops. In the counter, JK inputs are connected to logic high.

In circuit specifications, a circuit unit with n ports can be represented by an n-ary predicate specifying the behavior of the unit. In the synchronous 3-bit counter circuit, there are two primitive units, that is, an “and” gate and a JK flip-flop. The and(In1, In2, Out) predicate specifies the truth table of an “and” gate with two input ports In1 and In2 and an output port Out; for example, and(0, 1, 0) holds where the constants 0 and 1 stand for logic low and logic high, respectively. The jk(J, K, Ck, Q, Qnext) predicate specifies a clocked JK flip-flop with J-input J, K-input K, clock input Ck, output Q and the next state Qnext. We have left out the negated Q output and S and R inputs, usually available in JK flip-flops, to simplify the specification. We now give the
definition of the \( jk \) predicate:

\[
\begin{align*}
jk(0, 0, 1, Q, Q) \\
jk(0, 1, 1, Q, 0) \\
jk(1, 0, 1, Q, 1) \\
jk(1, 1, 1, 0, 1) \\
jk(1, 1, 1, 0, 0) \\
jk(\ldots, 0, Q, Q)
\end{align*}
\]

In Chronolog\((Z)\), the frequency of the clock can be defined by making the use of the linear
time of the logic. For instance, the program clause \( \text{clock}(1) \) specifies a clock synchronized with
Chronolog\((Z)\) time under the assumption that the clock signal is high when it is sampled at each
each moment in time, but it is low between any two consecutive moments in time. In the specification
of the counter circuit, the input to the circuit is taken from the \( \text{clock} \) predicate. The following is
the definition of the \( \text{counter} \) predicate:

\[
\begin{align*}
\text{first \ counter}(0, 0, 0). \\
\text{next \ counter}(Q_1, Q_2, Q_3) \leftarrow \\
&& \text{clock}(Ck), \ \text{counter}(I_1, I_2, I_3), \\
&& jk(1, 1, Ck, I_3, Q_3), \ \text{and}(Ck, I_3, C_3), \\
&& jk(1, 1, C_3, I_2, Q_2), \ \text{and}(Ck, I_3, T), \\
&& \text{and}(T, I_2, C_2), \ jk(1, 1, C_3, I_1, Q_1).
\end{align*}
\]

The first clause states that the counter is reset at the initial moment in time. The second clause
specifies the behavior of the circuit in terms of its current values and the clocked input. Shared
variables represent connections among the units in the circuit. In the second clause, two \( \text{and} \) units are used to model an \( \text{and} \) unit with three inputs. As in the Hamming-numbers example, the
open-ended goal \( \leftarrow \text{counter}(X, Y, Z) \) starts a non-terminating computation.

The temporal-logic-programming approach to circuit simulation has the advantages of the
logic-programming method of Clocksin [5]. It supports modularity and circuit specifications are
directly executable by Chronolog\((Z)\) implementations. Since it is based on a linear-time temporal
logic, it also offers additional advantages: feedback loops can be more naturally specified, and
delays and timing requirements can be modified in a rigorous manner.

### 3.3 Temporal Deductive Databases

We now consider a subset of Chronolog without any function symbols, that is Temporal Datalog.
Programs or Temporal Datalog can be regarded as “temporal deductive databases”. The meaning
of a Temporal Datalog program is considered as the temporal database the program models intensionally, in which each predicate in the program represents a temporal relation. Slices of temporal
relations are finite at particular moments in time or over periods of time, owing to the restriction
to the function-free subset of logic programs, but they are infinite in the aggregate.

Consider a simple database for a library. Suppose that new books do not go into circulation
until two weeks after their arrival. After then, they stay in circulation on a daily basis. Here the
logical time is interpreted as days. Let the \( \text{inCirc} \) predicate represent all those books that are in
circulation. The following is the partial definition of \( \text{inCirc} \).

\[
\begin{align*}
\text{first \ inCirc}(\text{mobyDick}). \\
\text{first \ inCirc}(\text{lesMiserables}). \\
\text{first \ inCirc}(\text{theHobbit}). \\
\ldots \\
\text{next}(14) \ \text{inCirc}(\text{Book}) \leftarrow \text{newArrival}(\text{Book}). \\
\text{next \ inCirc}(\text{Book}) \leftarrow \text{inCirc}(\text{Book}).
\end{align*}
\]
The first three rules define the books *mobyDick*, *lesMiserables* and *theHobbit* to be in circulation on the first day (time 0). The last two rules formalize our assumptions about the books in circulation. Note that the *newArrival* predicate is not formalized in the program, but supplied as input.

Bottom-up evaluation strategies, such as fixed-point computation, are used to compute the meaning of a given Datalog program [4]. In Temporal Datalog, only slices (finite portions) of temporal relations represented by time-varying predicates can be computed by bottom-up evaluation strategies.

### 3.4 Modular Chronolog(\( \mathbb{Z} \))

In a modular extension of Chronolog(\( \mathbb{Z} \)), we can specify software modules with internal memory. The extension is based on Fitting’s proposal [8] for modular logic programming. As an example, we give the specification of a stack module with unlimited capacity [13]. The actions on the stack are controlled by supplying a sequence of commands through the *cmd* predicate. Items are added to the stack with *push* commands and the top item is removed with *pop* commands. We assume that at any moment in time, exactly one action is allowed: we can push an item onto the stack or remove the top item. The *pop* command has no effect if the stack is empty. The output of the module is the *top* predicate, which is true of the item on top of the stack (if there is one) at any given moment in time.

```
module mod :
  import cmd; export top.

  first stack([\]).
  next stack([X | S]) ← cmd(push(X)), stack(S).
  next stack(S) ← cmd(pop), stack([X | S]).
  next stack([\]) ← cmd(pop), stack([\]).

  top(X) ← stack([X | S]).

end.
```

The first clause states that the stack is initially empty. The next three clauses specify the next state of the stack, depending on its current state and the action indicated by *cmd*. The last clause specifies the output predicate *top*. The *stack* predicate models the internal memory for the module, which is not observable from the outside world. Thus the notion of a “state” is captured within the module.

### 4 Conclusions

We have introduced the temporal logic language Chronolog(\( \mathbb{Z} \)) and outlined its features. Other applications of non-classical logics to logic programming include the multidimensional language Intense [11] and the modal framework Molog [6]. For more details on temporal and modal logic programming, we refer the reader to the literature; for example, see [7]. We believe that these non-classical languages will open up new application areas for logic programming.

### References


