ON THE COMPUTATIONAL COMPLEXITY OF STRATIFIED NEGATION IN LINEAR-TIME TEMPORAL LOGIC PROGRAMMING

COSTAS D. KOUTRAS
Graduate Program in Logic, Algorithms & Computation (MPLA)
University of Athens, Department of Mathematics,
Panepistimioupolis 157 84 Zographou, Hellas
e-mail: ckoutras@cc.ece.ntua.gr

CHRISTOS NOMIKOS
Graduate Program in Logic, Algorithms & Computation (MPLA)
University of Athens, Department of Mathematics,
Panepistimioupolis 157 84 Zographou, Hellas
e-mail: nomikos@cs.ntua.gr

A lot of formal approaches to the “right” semantics of negation in logic programming have been proposed during the last two decades and the importance of this topic for the areas of Non-Monotonic Reasoning and Deductive Databases has been stressed. In sharp contrast, only a few papers have been devoted to the semantics of negation in temporal logic programming, despite the intuitive importance and the practical implication of this issue. Recently, a simple syntactic criterion has been proposed in 
12: it is based on the cycle sum test and singles out the Chronolog programs for which a well-defined semantics of stratified negation can be given. In this paper, we show that this test is computationally affordable, and thus of great practical importance too: we sketch an algorithm for the cycle sum test, whose time requirements are linearly related to the “size” of the input Chronolog program.

1 Introduction

The treatment of negation in Logic Programming has been a major research issue for the last twenty years and is closely related to methods and techniques developed in the area of Non-Monotonic Reasoning. An overview of the approaches suggested in both areas is given in 3 and 1. On the contrary, only a few approaches exist to the treatment of negation in Temporal Logic Programming. In this paper, we examine the computational cost of the cycle sum syntactic check for temporal logic programs in the language Chronolog; programs that pass successfully this test are guaranteed to possess a well-defined negation semantics.

In a recent paper 12, the issue of stratified negation in the temporal logic programming language Chronolog 16 9 10 has been investigated. Stratified
negation was originally suggested by Apt, Blair and Walker \(^2\) and has been generalized by T. Przymusinski to local stratification in \(^{11}\). In \(^{12}\) Rondogiannis argues that the simple stratification test of \(^2\) is very restrictive for negation in temporal logic programming, since even very simple programs with a clear meaning fail to pass the test. For example, the following simple Chronolog program is mentioned:

\[
\begin{align*}
\text{first } & p(a). \\
\text{next } & p(X) \leftarrow \neg p(X).
\end{align*}
\]

This program fails to pass the test because of the circularity in the second clause; however, it is not truly circular and an appropriate test for Chronolog should take into account this kind of temporal circularities. Thus, P. Rondogiannis suggests using the cycle sum test \(^{15}\) and shows that the Chronolog programs which pass this test have a classical logic programming analog which is locally stratified, and thus have a unique perfect model. The cycle sum test was originally suggested by W. Wadge \(^{15}\) and it was used to ensure that a given temporal functional program of the language Lucid is deadlock free. It was later extended by S. Matthews \(^7\) to a wider context in the area of functional programming. We investigate here the complexity of this test in the Chronolog context and show that its time requirements are reasonable for practical implementations.

The rest of this paper is organized as follows: in Section 2 we give the necessary definitions for the Chronolog language and a model-preserving translation of Chronolog programs into classical ones taken directly from \(^{12}\). Section 3 provides the necessary formal details for the cycle sum test. Section 4 discusses its computational complexity and Section 5 concludes the paper with a discussion of future research directions.

## 2 The Chronolog Language

### 2.1 The Linear-Time Temporal Logic \(TL\)

The programming language Chronolog \(^{16}\) \(^9\) \(^{10}\) is based on the simple temporal logic \(TL\), which uses a linear notion of time with unbounded future. The time model employed is the set \(\mathcal{N}\) of natural numbers. The operator first is used to express the first moment in time (i.e. time 0), while next refers to the next moment in time. The syntax of \(TL\) extends the syntax of first-order logic with two additional formation rules: if \(A\) is a formula, then so are first \(A\) and next \(A\).
The semantics of temporal formulas of $TL$ is formally expressed in the following definition:

**Definition 2.1** A *temporal interpretation* $I$ of the temporal logic $TL$ comprises a non-empty set $D$, called the *domain* of the interpretation, over which the variables range, together with an element of $D$ for each variable; for each $n$-ary function symbol, an element of $[D^n \rightarrow D]$; and for each $n$-ary predicate symbol, an element of $[N \rightarrow 2^D]$. Following, we can define the ternary *satisfaction relation* $\models_{I,t} A$ denoting that a formula $A$ is true at a moment $t$, in some temporal interpretation $I$.

**Definition 2.2** The following recursive definition of $\models_{I,t} A$ gives the formal details for the semantics of $TL$:

1. If $f(e_0, \ldots, e_{n-1})$ is a term, then
   \[ I(f(e_0, \ldots, e_{n-1})) = I(f)(I(e_0), \ldots, I(e_{n-1})) \]

2. If $p$ is an $n$-ary predicate symbol and $e_0, \ldots, e_{n-1}$ are terms then
   \[ \models_{I,t} p(e_0, \ldots, e_{n-1}) \text{ iff } \langle I(e_0), \ldots, I(e_{n-1}) \rangle \in I(p)(t) \]

3. $\models_{I,t} \neg A$ iff it is not the case that $\models_{I,t} A$

4. $\models_{I,t} A \land B$ iff $\models_{I,t} A$ and $\models_{I,t} B$

5. $\models_{I,t} A \lor B$ iff $\models_{I,t} A$ or $\models_{I,t} B$

6. $\models_{I,t} (\forall x)A$ iff $\models_{I[d/x],t} A$ for all $d \in D$, where the interpretation $I[d/x]$ is the same as $I$ except that the variable $x$ is assigned the value $d$.

7. $\models_{I,t} \text{ first } A$ iff $\models_{I,0} A$

8. $\models_{I,t} \text{ next } A$ iff $\models_{I,t+1} A$

If a formula $A$ is true in a temporal interpretation $I$ at all moments in time, it is said to be true in $I$ (we write $\models_I A$) and $I$ is called a model of $A$.

### 2.2 The Chronolog Programs

We obtain the syntax of *Chronolog programs* by extending the syntax of classical logic programs with two temporal operators, namely *first* and *next*. A temporal reference is a (possibly empty) sequence of the above operators; we denote by $\text{next}^k$ a sequence of $k$ next operators. A *canonical temporal reference* is a temporal reference of the form $\text{first } \text{next}^k$. An *open temporal reference* is a temporal reference of the form $\text{next}^k$. A *temporal atom* is...
an atom preceded by either a canonical or an open temporal reference. We consider an extension of Chronolog that allows negation in the bodies of the rules.

A \textit{temporal clause} in Chronolog is a formula of the form:

$$H \leftarrow A_1, \ldots, A_k, \neg B_1, \ldots, \neg B_m.$$ 

where $H, A_1, \ldots, A_k, B_1, \ldots, B_m$ are temporal atoms and $k, m \geq 0$. If $k = m = 0$, the clause is said to be a \textit{unit temporal clause}. A \textit{temporal program} is a finite set of \textit{temporal clauses}. As a simple example, the following program from \cite{15} simulates the operation of the traffic lights:

\begin{verbatim}
first light(green).
next light(amber) \leftarrow light(green).
next light(red) \leftarrow light(amber).
next light(green) \leftarrow light(red).
\end{verbatim}

\subsection*{2.3 A Translation of Chronolog into Classical Logic Programming}

A Chronolog program can be easily transformed into a classical one, in a way that “preserves” its model theory. Intuitively, given a temporal program $P$, we obtain its classical counterpart $P^*$ by adding to the predicates of $P$ an explicit representation of time.

The transformation comprises the following steps:

- Replace every canonical temporal atom $\textbf{first} \ \textbf{next}^k \ p(e_0, \ldots, e_{n-1})$ in $P$ by the classical atom $p(s^k(0), e_0, \ldots, e_{n-1})$.

- Let $T$ be a variable that does not appear in $P$. Replace every open temporal atom of the form $\textbf{next}^k \ p(e_0, \ldots, e_{n-1})$ by the classical atom $p(s^k(T), e_0, \ldots, e_{n-1})$.

- In the body of every clause that contains at least one open temporal atom, add the atom $\textbf{nat}(T)$; its purpose is to restrict the time parameter so that it takes only natural number values. Add to the program the axiomatization of $\textbf{nat}$, as well (see cite \cite{12}).

The result of this transformation is called a \textit{time-classical} logic program. Also, terms of the form $s^k(0)$ are called (ground) \textit{time-terms}.

A Herbrand interpretation of a time-classical program is called \textit{normal} if:

- the only atoms contained regarding the predicate $\textbf{nat}$, are all the atoms of the set $\textbf{Nat} = \{ \textbf{nat}(s^k(0)) \mid k \geq 0 \}$. 

all the other atoms that it contains are of the form \( p(s^k(0), e_0, \ldots, e_{n-1}) \),
where \( k \geq 0 \).

In \(^{12}\), the following is shown:

**Theorem 2.3** Let \( P \) be a temporal logic program and \( P^* \) be its classical counterpart. Then, there is a one-to-one correspondence between the temporal Herbrand models of \( P \) and the normal Herbrand models of \( P^* \).

### 3 Stratified Negation and the Cycle Sum Test

Time-classical programs have a specific structure: the first argument of each predicate corresponds to the implicit time parameter of the initial temporal program. As it is shown in \(^{12}\), the special structure of these programs allows us to define a syntactic test, which when passed, ensures that the time-classical program is locally stratified \(^{11}\). Locally stratified logic programs have a unique perfect model, which is taken as their intended meaning. We can then take the corresponding temporal Herbrand model as the intended meaning of the initial temporal program. Let \( A \) be an atom appearing in a time-classical logic program; then, \( time(A) \) is the term that corresponds to the first argument of \( A \).

**Definition 3.1** Let \( P^* \) be a time-classical logic program and \( C \) be a clause in \( P^* \). Let \( H \) be the head of \( C \) and \( A \) be an atom (different from \( \text{nat}(T) \)) in the body of \( C \). The **temporal difference** \( \text{dif} \) between \( H \) and \( A \) is defined as follows:

\[
\text{dif}(H, A) = \begin{cases} 
  k - m, & \text{if } time(H) = s^k(0) \text{ and } time(A) = s^m(0) \\
  k - m, & \text{if } time(H) = s^k(T) \text{ and } time(A) = s^m(T) \\
  -\infty, & \text{if } time(H) = s^k(T) \text{ and } time(A) = s^m(0) \\
  k - m, & \text{if } time(H) = s^k(T) \text{ and } time(A) = s^m(T) 
\end{cases}
\]

Informally speaking, \( \text{dif}(H, A) \) expresses how far, that is how many timepoints in the worst case, the head \( H \) of a clause leads the atom \( A \) in the body of the clause. The value \(-\infty\) used in the last case of the above definition, signifies that in this case it is not possible to determine a finite integer value by which the head leads the atom in the body in the worst case (because the head refers to a specific moment in time while the atom in the body has an open temporal reference).

The \( \text{dif} \) measure is used for the construction of the **cycle sum graph** of a given time-classical logic program. The following definition provides the formal details:
Definition 3.2 Let \( P^* \) be a given time-classical logic program. The cycle sum graph of \( P^* \) is a directed weighted graph \( CG_{P^*} = (V, E) \). The set \( V \) of vertices of \( CG_{P^*} \) is the set of predicate symbols appearing in \( P^* \). The set \( E \) of edges consists of triples \((p, q, w)\), where \( p, q \in V \) and \( w \in \mathbb{Z} \cup \{-\infty\} \). An edge \((p, q, w)\) belongs to \( E \) if in \( P^* \) there exists a clause with an atom \( H \) as its head and an atom \( A \) in its body, such that the predicate symbol of \( H \) is \( p \), the predicate symbol of \( A \) is \( q \) and \( \text{dif}(H, A) = w \).

We can now state the cycle sum test:

Definition 3.3 A time-classical logic program \( P^* \) passes the cycle sum test if the sum of weights across every cycle in \( CG_{P^*} \) is positive.

The following theorem from \(^{12}\) justifies the use of the cycle sum test and reveals its importance for the semantics of negation in temporal logic programming:

Theorem 3.4 Let \( P^* \) be a time-classical logic program that passes the cycle sum test. Then \( P^* \) has a unique perfect model which is also normal.

4 The Computational Complexity of Stratifiability Test for Chronolog Programs

In this section we sketch a decision procedure for checking whether a Chronolog program passes the cycle sum test. The algorithm we sketch creates the cycle sum (multi)graph and constructs out of it a suitable sub-graph, on which a shortest-path algorithm is executed. The cycle sum test can be actually performed using any shortest-path algorithm operating on graphs with negative weights (like the Bellman-Ford algorithm for instance) \(^{4}\); if after the execution of the algorithm a candidate “shortest path” can be further shortened, then a negative cycle exists in the graph.

In our approach we employ a fast scaling algorithm of Gabow and Tarjan from \(^{5}\), which runs in \( O(\sqrt{|V|} \cdot |E| \cdot \log(|V| \cdot W)) \) time, where \( W \) is the magnitude of the largest-magnitude weight of any edge in the graph. Typical scaling algorithms for shortest paths solve initially the problem by considering only the highest order bits of the weights. They progressively increase the number of bits considered and refine the solution. After \( \log W \) scales (refinements) the optimum solution is computed (all the bits have been considered). The Gabow-Tarjan algorithm uses a different scaling method: at each scale, instead of an optimal solution, it computes an approximate solution (which is easier to compute) and it uses \( \log |V| \) additional scales to ensure that the last approximate solution is exact.

The shortest path algorithm is not executed directly in the cycle sum graph \( P^* \) for the following reasons:
• $CG_P$- is actually a directed multigraph, as it may contain parallel edges with the same direction. This happens in the case an atom $A$ appears in the body of two different clauses with head $H$. For the purposes of the cycle sum test, parallel edges can be eliminated, reducing significantly the size of the graph.

• In this sub-graph we have to replace the $-\infty$ value with an appropriate finite negative value. The obvious solution would be to employ the smallest number available in the computer on which the algorithm is run. But such a solution will affect the efficiency of the scaling algorithm. So, we have to make a different choice: assume that $b$ is the largest positive weight value occurring in $SCG_P$. Our algorithm proceeds to replace every $-\infty$ value with $-(|V| - 1) * b$; as shown below the output of the cycle sum test algorithm remains correct.

• We must take care of cycles of length zero. The cycle sum test fails for graphs containing cycles with zero weight, while shortest path algorithms work properly in this case. We overcome this difficulty with a simple trick: we decrease each negative or zero weight by a small amount (we have chosen it to be $\frac{1}{|V|}$). In order to eliminate fractional weights, we multiply the weight of every edge with $|V|$.

We can now define the graph $SCG_P$- on which the cycle sum test is performed.

**Definition 4.1** Given a cycle sum graph $CG_P$- of a Chronolog program $P$ with set of vertices $V$, define $SCG_P$- to be the graph constructed from $CG_P$- using the following procedure:

1. Delete all parallel edges (with the same direction) connecting two vertices, except for the one with minimum weight.

2. Replace weights equal to $-\infty$ with $-(|V| - 1) * b$.

3. Subtract $\frac{1}{|V|}$ from every non-positive weight.

4. Multiply all weights with $|V|$.

The following theorem makes the previous observations precise.

**Theorem 4.2** The cycle sum graph $CG_P$- of a Chronolog program $P$ contains a non-positive cycle if and only if $SCG_P$- contains a negative cycle.

**Proof:** Consider the vertices of any cycle in the original multigraph. These vertices also form a cycle in the graph constructed after the first step of the
transformation, because elimination of parallel edges does not affect adjacency. Furthermore if the original cycle was non-positive, then the same holds for the cycle in the new graph.

The selection of the value $-(|V|-1) \times b$ to represent $-\infty$ in the second step, guarantees that whenever an edge with original weight $-\infty$ is contained in a cycle, the cycle cannot be positive: in the extreme case where the cycle is a Hamiltonian cycle and all the other edges are weighted with $b$, the weight of this cycle is 0. Therefore, the graph constructed in the first two steps contains a non-positive cycle if and only if the original graph does so.

After completion of the third step, a negative cycle has remained negative (obviously!). Also a cycle with zero weight cannot contain only positive weight edges. Thus after the decrement of negative and zero weights this cycle will become negative. Further, a positive cycle must contain at least one positive edge and at most $|V|-1$ non-positive edges. Since its original weight was at least 1, the new weight after this step is at least $\frac{1}{|V|}$, i.e. it remains positive. Finally, the multiplication by a positive constant in the fourth step does not affect the sign of the weight of any cycle.

Consequently if $CG_{P^-}$ has a non-positive cycle then there is a negative cycle in $SCG_{P^-}$. On the other hand if $CG_{P^-}$ contains only positive cycles, all cycles in $SCG_{P^-}$ are positive.

In order to run the Gabow-Tarjan algorithm on $SCG_{P^-}$, we must also specify a proper source for the shortest paths. The source can be selected arbitrarily if $SCG_{P^-}$ is strongly connected. However, since all vertices of any cycle must reside in the same connected component, if $SCG_{P^-}$ is not strongly connected, we can perform the cycle sum test independently for every strongly connected component. Strongly connected components can be found in time $O(|V| + |E|)$ \(^4\). Thus we can assume, without loss of generality, that $SCG_{P^-}$ is strongly connected.

We proceed now to sketch the decision procedure: given a Chronolog program $P$, we denote by $L$ the length of $P$, by $r$ the number of rules in $P$, by $m$ the maximum number of predicates that appear in a rule of $P$ and by $p$ the total number of predicate names that appear in $P$, which is also the number of vertices in $SCG_{P^-}$.

**Algorithm CST**($P$)

**Input:** a Chronolog program $P$

**Output:** $true$ if $P$ passes the cycle sum test, $false$ otherwise

**Step 1:** construct $CG_{P^-}$ out of $P$,

and simultaneously eliminate parallel edges
Step 2: to get the weight assignment of $SCG_P$:

for each edge $(v_1, v_2)$:

if $w((v_1, v_2)) = -\infty$ then $w((v_1, v_2)) = -(p - 1) \cdot p \cdot b - 1$
else if $w((v_1, v_2)) \leq 0$ then $w((v_1, v_2)) := w((v_1, \overline{v_2})) \cdot p - 1$
else $w((v_1, v_2)) := w((v_1, v_2)) \cdot p$

Step 3: choose arbitrarily a source $s$

run GABOW-TARJAN on the output of step 2, with source $s$

if a negative-weighted cycle is detected then return false
else return true

We are now ready to state the main result of this paper.

**Theorem 4.3** If the algorithm CST returns true, the input program $P$ passes the cycle sum test and thus has a well-defined negation semantics. Moreover, algorithm CST runs within time $O(L + \sqrt{p} \cdot \min(p^2, r \cdot m) \cdot \log(p \cdot b))$.

**Proof:** The first part of the theorem (correctness of the algorithm) follows immediately by the results of $^{12}$, Theorem 4.2 and the properties of the GABOW-TARJAN algorithm $^5$.

We will now compute the running time of CST. The first step can be carried out in $O(L)$ time, actually during the parsing of the Chronolog program. The graph constructed has $|V| = p$ vertices (one for each predicate name). The number of edges is $O(\min(p^2, r \cdot m))$, since for each one of the $r$ rules, at most $m$ edges can be added to $SCG_P$, and the total number of edges in a graph with $p$ vertices is at most $\frac{p(p - 1)}{2}$. The second step simply examines all edges, so it runs in $O(\min(p^2, r \cdot m))$ time. Finally, the last step, runs the GABOW-TARJAN algorithm, in graph $SCG_P$. The value of the parameter $W$ is $(p - 1) \cdot p \cdot b + 1$ (the absolute value of the weight that replaced $-\infty$). The complexity of this step is $O(\sqrt{p} \cdot \min(p^2, r \cdot m) \cdot \log(p^3 \cdot b))$. Thus the total time required for the three steps (using the fact that $\log(p^3 \cdot b) = O(\log(p \cdot b))$) is $O(L + \sqrt{p} \cdot \min(p^2, r \cdot m) \cdot \log(p \cdot b))$.

5 Conclusions and Further Research

The results of this paper contribute to the important topic of negation in temporal logic programming, and in particular its computational properties. There is a variety of research directions that call for our attention. Interesting issues would certainly be:

- The further exploration of negation semantics in Chronolog.
- Full identification of possible connections between the negation semantics for classical logic programs and temporal ones.
• Investigation of other computational complexity questions in the area of temporal LP, which remain largely unexplored since most of the work in this field is mainly concerned with the richness of the languages proposed and the correctness of the corresponding proof procedures.

Many interesting questions remain open at the level of the underlying temporal logic $TL$, both from the logical as well as from the computational perspective. This logic is a simple, yet somewhat “strange” bimodal temporal logic; this is mainly due to the first operator which does not make actual use of the accessibility relation (just evaluates in the first moment). For this logic a complete axiomatization has been recently found along with some results on the complexity of the validity and the entailment problems. It is very interesting to examine whether these results can be extended to the branching time framework of $BTL$, which underlies the programming language Cactus. Also, a very challenging problem seems to be the investigation of the modal model theory of $TL$ and $BTL$:

• Do we have any useful notion of truth-preserving model-theoretic constructions (such as bounded morphisms or bisimulation)?

• What is the correspondence of these logics to (fragments of) classical predicate logic?

Clearly there are many interesting problems worth investigating.

References


\[ ^a \text{This question was raised during the symposium. The results of P.-Y. Schobbens and J.-F. Raskin were brought to our attention by M.Gergatsoulis and P.Rondogiannis.} \]


