Stratified Negation in Temporal Logic Programming and the Cycle Sum Test

P. Rondogiannis

Department of Computer Science, University of Ioannina
P.O. Box 1186, GR-45110 Ioannina, Greece
e_mail: prondo@cs.uoi.gr

Abstract

We consider stratified negation in temporal logic programming. We demonstrate that the *cycle sum test* (which was initially proposed for detecting deadlocks in the context of temporal functional programming) can also be used as a syntactic stratification test for temporal logic programming. Therefore, on the one hand we exhibit a class of temporal logic programs with negation which have a well-defined semantics, and on the other hand we provide further evidence that the cycle sum test is a fundamental one in the area of temporal programming.

Keywords: Temporal Logic Programming, Negation, Stratification, Cycle Sum Test.

1 Introduction

Negation in logic programming has received considerable research attention largely due to its applications in areas such as artificial intelligence and deductive databases. From a semantic point of view, the addition of negation in classical logic programming is far from straightforward and many different approaches have been developed [PP90, AB94]. One of the earliest such approaches is the so-called *stratified negation* [ABW88]. Intuitively, a stratified logic program is one in which negation is not used in a circular way, and this (syntactically determinable) condition ensures that the program has a unique *perfect* model. Stratification was generalized by T. Przymusinski [Prz88] to *local stratification* which is more powerful but can not in general be detected syntactically.

In this paper, we consider stratified negation in temporal logic programming (and more specifically in the context of the temporal logic programming language Chronolog [Wad88, OW92a, OW92b]). The simple stratification test of [ABW88] appears to be too restrictive for temporal logic programming with negation: even the simplest programs, that have an obvious meaning, fail to pass the test. Consider for example the following Chronolog program:

```
first p(a).
next p(X) ← ¬ p(X).
```

The declarative reading of the above program is: "p is true of a at time 0. Moreover, p is true of X at time $t+1$ if p is not true of X at time t". A temporal model of the program that suggests itself is the one in which p is true of a at the time points $0, 2, 4, \ldots$. However, the simple stratification test fails for the above program, due to the circularity in the second clause.

Programs such as the above one are not however truly circular. The meaning of temporal logic programs depends on an implicit time parameter which needs to be taken into consideration or otherwise most programs would have to be rejected. In other words, an effective stratification test for temporal logic programming should also examine for temporal circularities in the program.

In [Wad81] W. Wadge developed the *cycle sum test* which ensures that a given temporal functional program of the language Lucid [WA85], is deadlock free. The test was later extended by S. Matthews [Mat95] to a wider context (but still in the area of functional programming).

We show that the test is also applicable in the area of temporal logic programming with negation, and we demonstrate that programs that pass the test have a well-defined semantics. Our contribution is therefore twofold: on the one hand we exhibit a class of temporal logic programs with negation which have a clear meaning, and on the other hand we provide further evidence that the cycle sum test is a fundamental one in the area of temporal programming in general.

The results presented in this paper initially appeared in preliminary form in [Ron98]. The rest of the present paper is organized as follows: section 2 contains preliminary material on temporal logic programming. Section 3 presents a transformation algorithm from temporal logic programs into classical ones, in such a way that the model theory of the initial programs is preserved. Section 4 defines the cycle sum test for the classical logic programs that result from the transformation. Section 5 demonstrates that programs passing the cycle sum test have a well-defined meaning. Section 6 concludes the paper with a discussion of related work and possible extensions.

2 Preliminaries: Temporal Logic Programming

The temporal logic programming language we consider here is the language Chronolog [Wad88, OW92a, OW92b]. As an informal introduction to the language, consider the following Chronolog [Wad88] program that simulates the operation of the traffic lights:

```
first light(green).
next light(amber) ← light(green).
next light(red) ← light(amber).
next light(green) ← light(red).
```

The syntax of Chronolog programs is an extension of the syntax of classical logic programs [Llo87] with two temporal operators, namely **first** and **next**. The declarative reading of these operators will be discussed shortly. A *temporal reference* is a sequence (possibly empty) of the above operators. We will often write $\mathbf{next}^k$ to represent a sequence of k **next** operators. A *canonical temporal reference* is a temporal reference of the form **first** $\mathbf{next}^k$. An *open temporal reference* is a temporal reference of the form $\mathbf{next}^k$. A *temporal atom* is an atom preceded by either a canonical or an open temporal reference.

In this paper we consider an extension of Chronolog that allows negation in the bodies of the rules of a program (and any reference to Chronolog in the rest of the paper will concern this particular extension).

A *temporal clause* in Chronolog is a formula of the form:

$$\mathbf{H} \leftarrow \mathbf{A}_1, \ldots, \mathbf{A}_k, \neg\mathbf{B}_1, \ldots, \neg\mathbf{B}_m.$$

where $\mathbf{H}, \mathbf{A}_1, \ldots, \mathbf{A}_k, \mathbf{B}_1, \ldots, \mathbf{B}_m$ are temporal atoms and $k, m \geq 0$. If $k = m = 0$, the clause is said to be a *unit temporal clause*. A *temporal program* is a finite set of *temporal clauses*.

Chronolog is based on the relatively simple *temporal logic* TL, which uses a linear and discrete notion of time with unbounded future. The set of time moments can then be modeled by 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 the formulas of TL is an extension of the syntax of first-order logic with two formation rules: if $\mathbf{A}$ is a formula, then so are **first** $\mathbf{A}$ and **next** $\mathbf{A}$.

The semantics of temporal formulas of TL are given using the notion of *temporal interpretation* [OW92a, OW92b]:

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 $[D^n \to D]$; and for each n-ary predicate symbol, an element of $[\mathcal{N} \to 2^{D^n}]$.

In the following definition, the satisfaction relation $\models$ is defined in terms of temporal interpretations. $\models_{I,t} \mathbf{A}$ denotes that a formula $\mathbf{A}$ is true at a moment t in some temporal interpretation I.

Definition 2.2. The semantics of the elements of the temporal logic TL are given inductively as follows:

1. If $\mathbf{f}(\mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$ is a term, then $I(\mathbf{f}(\mathbf{e}_0, \ldots, \mathbf{e}_{n-1})) = I(\mathbf{f})(I(\mathbf{e}_0), \ldots, I(\mathbf{e}_{n-1}))$.
2. For any n-ary predicate symbol $\mathbf{p}$ and terms $\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}$,

   $\models_{I,t} \mathbf{p}(\mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$ *iff* $\langle I(\mathbf{e}_0), \ldots, I(\mathbf{e}_{n-1})\rangle \in I(\mathbf{p})(t)$
3. $\models_{I,t} \neg\mathbf{A}$ *iff it is not the case that* $\models_{I,t} \mathbf{A}$
4. $\models_{I,t} \mathbf{A} \wedge \mathbf{B}$ *iff* $\models_{I,t} \mathbf{A}$ *and* $\models_{I,t} \mathbf{B}$
5. $\models_{I,t} \mathbf{A} \vee \mathbf{B}$ *iff* $\models_{I,t} \mathbf{A}$ *or* $\models_{I,t} \mathbf{B}$
6. $\models_{I,t} (\forall\mathbf{x})\mathbf{A}$ *iff* $\models_{I[d/\mathbf{x}],t} \mathbf{A}$ *for all* $d \in D$*, where the interpretation* $I[d/\mathbf{x}]$ *is the same as* I *except that the variable* $\mathbf{x}$ *is assigned the element* d*.*
7. $\models_{I,t}$ **first** $\mathbf{A}$ *iff* $\models_{I,0} \mathbf{A}$
8. $\models_{I,t}$ **next** $\mathbf{A}$ *iff* $\models_{I,t+1} \mathbf{A}$

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

The semantics of Chronolog are defined in terms of *temporal Herbrand interpretations*. A notion that is crucial in the discussion that follows, is that of *canonical atom*:

Definition 2.3. A *canonical temporal atom* is a temporal atom whose temporal reference is canonical.

As in the theory of classical logic programming [Llo87], the set $U_{\mathbf{P}}$ generated by constant and function symbols that appear in $\mathbf{P}$, called *Herbrand universe*, is used to define *temporal Herbrand interpretations*. Temporal Herbrand interpretations can be regarded as subsets of the *temporal Herbrand Base* $B_{\mathbf{P}}$ of $\mathbf{P}$, consisting of all *ground canonical temporal atoms* whose predicate symbols appear in $\mathbf{P}$ and whose arguments are terms in the Herbrand universe $U_{\mathbf{P}}$ of $\mathbf{P}$. In particular, given a subset H of $B_{\mathbf{P}}$, we can define a Herbrand interpretation I by the following:

$$\langle \mathbf{e}_0, \ldots, \mathbf{e}_{n-1} \rangle \ \in I(\mathbf{p})(t) \quad \textit{iff} \quad \mathbf{first}\ \mathbf{next}^t\ \mathbf{p}(\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}) \in H$$

A *temporal Herbrand model* is a temporal Herbrand interpretation, which is a model of the program. In the rest of the paper when we refer to a "model of a program" we always mean a Herbrand model.

3 The Classical Counterpart of a Temporal Program

In this section we demonstrate that a temporal logic program can be transformed into a classical one whose model theory is closely related to that of the initial program. Intuitively, given a temporal program $\mathbf{P}$ we obtain its *classical counterpart* $\mathbf{P}^*$ by adding to the predicates of $\mathbf{P}$ an extra parameter that represents explicitly the notion of time.

The transformation can be formalized as follows:

- Let $\mathbf{s}$ be a unary function symbol and $\mathbf{0}$ be a constant, that do not appear in $\mathbf{P}$. Replace every canonical temporal atom $\mathbf{first}\ \mathbf{next}^k\ \mathbf{p}(\mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$ in $\mathbf{P}$ by the classical atom $\mathbf{p}(\mathbf{s}^k(\mathbf{0}), \mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$.

- Let $\mathbf{T}$ be a variable that does not appear in $\mathbf{P}$. Replace every open temporal atom of the form $\mathbf{next}^k\ \mathbf{p}(\mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$ by the classical atom $\mathbf{p}(\mathbf{s}^k(\mathbf{T}), \mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$.

- Let $\mathbf{nat}$ be a predicate name that does not appear in $\mathbf{P}$. In the body of every clause of $\mathbf{P}$ that contains at least one open temporal atom, add the atom $\mathbf{nat}(\mathbf{T})$ (whose purpose is to restrict the time parameter to obtaining only natural number values). Also, add to the program the (usual) definition of $\mathbf{nat}$.

- Let $\mathbf{universe}$ be a predicate name that does not appear in $\mathbf{P}$. In the body of every clause of $\mathbf{P}$ add the atoms $\mathbf{universe}(\mathbf{x}_0), \ldots, \mathbf{universe}(\mathbf{x}_{n-1})$, where $\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}$ are the variables appearing in that clause. The purpose of these atoms is to restrict the above variables to obtaining terms from the Herbrand universe of $\mathbf{P}$ (and not terms that contain the symbols $\mathbf{s}$ and $\mathbf{0}$ that are only used for the special new variable $\mathbf{T}$). Also, add to the program the axiomatization of $\mathbf{universe}$[1].

In the following, we will also refer to the programs that result from the above transformation as *time-classical* logic programs. Moreover, atoms that appear in time-classical logic programs and whose predicate symbol is different from $\mathbf{nat}$ and $\mathbf{universe}$, will be called *time-atoms*. Finally, terms of the form $\mathbf{s}^k(\mathbf{0})$ will be called (ground) *time-terms*.

The above transformation algorithm is illustrated by the following example:

[1]Notice that this is similar to the transformation used to obtain type-free first order formulas from typed ones (see for example [Llo87][pages 150-151]).

Example 3.1. Consider the following temporal logic program:

```
first next p(a).
next next p(f(X,X)) ← ¬ next p(X).
```

The transformation described above results in:

```
p(s(0),a).
p(s(s(T)),f(X,X)) ← ¬ p(s(T),X),nat(T),universe(X).
nat(0).
nat(s(X)) ← nat(X).
universe(a).
universe(f(X,Y)) ← universe(X),universe(Y).
```

which is the classical counterpart of the initial program.

Definition 3.1. Let $\mathbf{P}$ be a temporal logic program and $\mathbf{P}^*$ be its classical counterpart. A Herbrand interpretation of $\mathbf{P}^*$ is called *normal* if:

1. The only atoms regarding the predicate **nat** that it contains are all the atoms of the set $Nat = \{\mathbf{nat}(\mathbf{s}^k(\mathbf{0})) \mid k \geq 0\}$.
2. The only atoms regarding the predicate **universe** that it contains are all the atoms of the set $Universe = \{\mathbf{universe}(\mathbf{e}) \mid \mathbf{e} \in U_{\mathbf{P}}\}$.
3. All the other atoms that it contains are of the form $\mathbf{p}(\mathbf{s}^k(\mathbf{0}), \mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$, where $k \geq 0$ and $\mathbf{e}_0, \ldots, \mathbf{e}_{n-1} \in U_{\mathbf{P}}$.

The following theorem can then be easily established:

Theorem 3.1 *Let* $\mathbf{P}$ *be a temporal logic program and* $\mathbf{P}^*$ *be its classical counterpart. Then, there is a one-to-one correspondence between the temporal Herbrand models of* $\mathbf{P}$ *and the normal Herbrand models of* $\mathbf{P}^*$.

Proof: Given a temporal Herbrand model I of $\mathbf{P}$, obtain a classical interpretation I^* by replacing every canonical ground temporal atom $\mathbf{first}\ \mathbf{next}^k\ \mathbf{p}(\mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$ in I by the classical atom $\mathbf{p}(\mathbf{s}^k(\mathbf{0}), \mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$. It can be easily shown that $I^* \cup \{\mathbf{nat}(\mathbf{s}^k(\mathbf{0})) \mid k \geq 0\} \cup \{\mathbf{universe}(\mathbf{e}) \mid \mathbf{e} \in U_{\mathbf{P}}\}$ is a model of $\mathbf{P}^*$. Similarly, when given a normal Herbrand model of the classical program $\mathbf{P}^*$ one can easily obtain a temporal Herbrand model of $\mathbf{P}$ by removing the atoms regarding **nat** and **universe** and transforming the classical atoms into canonical temporal atoms. ■

We see therefore that from a model theory point of view, a temporal logic program is closely related to its classical counterpart. In the rest of this paper, we will consider and analyze the classical counterpart of a given temporal logic program.

4 The Cycle Sum Test

The classical programs that result from the transformation defined in the previous section have a specific structure: the first argument of each predicate corresponds to the implicit time parameter of the initial temporal program. In this section, we show that the special structure of these programs allows us to define a syntactic test (the *cycle sum test*), which when passed, ensures that the time-classical program is *locally stratified* [Prz88]. It is well known [Prz88] that locally stratified logic programs have a unique model (called *perfect* model), which is taken as their intended meaning. Due to the model correspondence Theorem 3.1, we can then take the corresponding temporal Herbrand model as the intended meaning of the initial temporal program.

In the following, we formally define the cycle sum test which is applied to the classical logic programs that result from the transformation of section 3.

The following definitions are needed:

Definition 4.1. Let $\mathbf{A}$ be a time-atom appearing in a time-classical logic program. Then, $time(\mathbf{A})$ is the first argument of $\mathbf{A}$.

Definition 4.2. Let $\mathbf{P}^*$ be a time-classical logic program and $\mathbf{C}$ be a clause in $\mathbf{P}^*$. Let $\mathbf{H}$ be the head of $\mathbf{C}$ and $\mathbf{A}$ be an atom in the body of $\mathbf{C}$. The *temporal difference dif* between $\mathbf{H}$ and $\mathbf{A}$ is defined as follows:

- If both $\mathbf{A}$ and $\mathbf{H}$ are time-atoms, then:

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

- If one or both of $\mathbf{H}$ and $\mathbf{A}$ is not a time-atom[2] (ie. it is **nat** or **universe**) then, $dif(\mathbf{H},\mathbf{A}) = 1$.

Intuitively, the value of $\mathit{dif}(\mathbf{H},\mathbf{A})$ expresses a lower bound on how far (ie. how many time-points) the head $\mathbf{H}$ of a clause leads the atom $\mathbf{A}$ in the body of the clause. In particular, 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).

Example 4.1. Consider the following temporal logic program:

```
first p(a).
next p(X) ← next next q(X),¬ first r(X).
first next next p(X) ← next q(X).
```

[2]Notice that the definition of *dif* for non time-atoms is not of real significance. It is actually only provided so as that the *CG* graph of Definition 4.3 that follows, more closely parallels the *DG* graph of Definition 5.2.

The corresponding time-classical logic program is (we omit the clauses for **nat** and **universe**):

```
p(s(0),a).
p(s(T),X) ← q(s(s(T)),X),¬ r(0,X),nat(T),universe(X).
p(s(s(0)),X) ← q(s(T),X),nat(T),universe(X).
```

Then, using the definition of *dif* we have (we consider only time-atoms):

$$\begin{array}{lcl} \mathit{dif}(\mathtt{p(s(T),X)},\mathtt{q(s(s(T)),X)}) & = & -1 \\ \mathit{dif}(\mathtt{p(s(T),X)},\mathtt{r(0,X)}) & = & 1 \\ \mathit{dif}(\mathtt{p(s(s(0)),X)},\mathtt{q(s(T),X)}) & = & -\infty \end{array}$$

The following definition formalizes the notion of *cycle sum graph* of a given time-classical logic program.

Definition 4.3. Let $\mathbf{P}^*$ be a given time-classical logic program. The *cycle sum graph* of $\mathbf{P}^*$ is a directed weighted multi-graph with self-loops $CG_{\mathbf{P}^*} = (V, E)$. The set V of vertices of $CG_{\mathbf{P}^*}$ is the set of predicate symbols appearing in $\mathbf{P}^*$. The set E of edges consists of triples $(\mathbf{p}, \mathbf{q}, w)$, where $\mathbf{p}, \mathbf{q} \in V$ and $w \in \mathcal{Z} \cup \{-\infty\}$. An edge $(\mathbf{p}, \mathbf{q}, w)$ belongs to E if in $\mathbf{P}^*$ there exists a clause with an atom $\mathbf{H}$ as its head and an atom $\mathbf{A}$ in its body, such that the predicate symbol of $\mathbf{H}$ is $\mathbf{p}$, the predicate symbol of $\mathbf{A}$ is $\mathbf{q}$ and $\mathit{dif}(\mathbf{H}, \mathbf{A}) = w$.

We can now state the cycle sum test:

Definition 4.4. A time-classical logic program $\mathbf{P}^*$ passes the cycle sum test if the sum of weights across every cycle in $CG_{\mathbf{P}^*}$ is positive.

Example 4.2. The lights program of section 2 can also be coded using negation in the bodies of the clauses:

```
first light(green).
next light(amber) ← ¬ light(red),¬ light(amber).
next light(red) ← ¬ light(green),¬ light(red).
next light(green) ← ¬ light(amber),¬ light(green).
```

It can easily be shown that the classical counterpart of the above program passes the cycle sum test.

In the following section we show that every time-classical logic program passing the cycle sum test is locally stratified and therefore has a well-defined meaning. Before we proceed with the proof of this statement, some further discussion on the cycle-sum test is necessary.

There is an important difference between the test described above and the usual stratification tests for classical logic programming. Given a classical logic program with negation, a stratification algorithm usually constructs a program dependency graph [PP90] and the edges of the graph are labeled as *positive* or *negative* depending on whether they "connect" the head of a program clause with a positive or negative literal in the body of the clause. The cycle sum test as defined above, does not take into consideration positive and negative edges, and it is therefore possible that certain positive programs will not be directly acceptable by the test. For example, consider the following:

```
first p(a).
p(X) ← next p(X).
```

This program is not directly acceptable by the test (the cycle sum graph contains a cycle with weight -1) but it obviously has a well-defined meaning under the standard semantics of temporal logic programming.[3] The obvious solution is to define as acceptable all the programs that are either positive or pass the cycle sum test.

We believe however that a refinement of the test that would additionally take into account positive and negative edges is possible although not straightforward (this is further discussed in the concluding section). In this article we restrict attention to the formulation of the test given by Definition 4.4.

We conclude the discussion on the cycle sum test by noting that it is easy to find an efficient (polynomial) algorithm that checks whether a given program passes the test or not. Actually, as it is demonstrated in [CN99] the test can be implemented using any shortest-path algorithm operating on graphs with negative weights (like the Bellman-Ford algorithm [CLR90]): if after the execution of the algorithm a potentially shortest path can be further shortened, then a negative cycle exists in the graph.

5 Justification of the Cycle Sum Test

In this section we show that time-classical logic programs passing the cycle sum test are locally stratified and therefore have a unique perfect Herbrand model. The following (graph theoretic) definition will be used in the rest of this section:

Definition 5.1. [Har72] Let G be a directed graph. A *directed walk* in G is a sequence of vertices and edges, $v_0 e_1 v_1 \cdots e_n v_n$ in which each edge e_i connects v_{i-1} with v_i. A *closed walk* has the same first and last vertices.

The following definitions regarding local stratification are also necessary:

Definition 5.2. [PP90] Let $\mathbf{P}$ be a classical logic program (with negation). The *dependency graph* $DG_{\mathbf{P}}$ of $\mathbf{P}$ is a graph whose vertex set is the Herbrand base $B_{\mathbf{P}}$ of $\mathbf{P}$ and whose edges are determined as follows: if $\mathbf{A}$ and $\mathbf{B}$ are two atoms in $B_{\mathbf{P}}$, there exists a directed edge from $\mathbf{A}$ to $\mathbf{B}$ if and only if there exists an instance of a clause in $\mathbf{P}$ whose head is $\mathbf{A}$ and one of whose premises is either $\mathbf{B}$ or $\neg\mathbf{B}$. In the latter case, the edge is called *negative*.

Definition 5.3. [PP90] Let $\mathbf{P}$ be a classical logic program (with negation). For any two ground atoms $\mathbf{A}$ and $\mathbf{B}$ in $B_{\mathbf{P}}$ we write $\mathbf{A} < \mathbf{B}$ if there exists a directed walk in the dependency graph of $\mathbf{P}$ leading from $\mathbf{A}$ to $\mathbf{B}$ and passing through at least one negative edge. We call the relation $<$ the *priority relation* between ground canonical atoms.

The following theorem from [PP90] actually defines the notion of local stratification in terms of the priority relation $<$:

Theorem 5.1 *[PP90] A logic program* $\mathbf{P}$ *is locally stratified if and only if every increasing sequence of ground atoms under* $<$ *is finite.*

Notice that given a time-classical logic program $\mathbf{P}^*$, there is a close relationship between the structure of the cycle sum graph $CG_{\mathbf{P}^*}$ and the dependency graph $DG_{\mathbf{P}^*}$. This relationship is reflected by the following Lemma:

[3] However, notice that programs such as the above in fact contain some form of deadlock [Wad81] and are in a sense against the spirit of temporal logic programming which views predicates as infinite *streams* [Wad88].

Lemma 5.1 *Let $\mathbf{P}^*$ be a time-classical logic program. Then, a directed walk in $DG_{\mathbf{P}^*}$ corresponds to a directed walk in $CG_{\mathbf{P}^*}$.*

Proof: Let $v_0e_1v_1\cdots e_nv_n$ be a directed walk in $DG_{\mathbf{P}^*}$. The graph $DG_{\mathbf{P}^*}$ is constructed in a similar way as the graph $CG_{\mathbf{P}^*}$ by looking at the structure of the clauses of $\mathbf{P}^*$. The main difference is that the vertices of $DG_{\mathbf{P}^*}$ are atoms belonging to the Herbrand base of $\mathbf{P}^*$ while the vertices of $CG_{\mathbf{P}^*}$ are predicate names appearing in $\mathbf{P}^*$. Therefore, there exists a corresponding sequence of vertices and edges $v'_0e'_1v'_1\cdots e'_nv'_n$ in $CG_{\mathbf{P}^*}$, such that each v'_i refers to the same predicate name $\mathbf{p}_i$ as v_i. ∎

The lemma given below will be used in the following discussion:

Lemma 5.2 *Let W be a closed walk in a directed weighted graph G. Then, there exists a sequence of (not necessarily distinct) cycles $C_1,\ldots,C_k$ of G such that the sum of the weights of the edges of W is equal to the sum of the weights of the edges of $C_1,\ldots,C_k$.*

Proof: The proof is by induction on the length l of the walk W. The induction basis is for $l=1$ and it obviously holds because in this case the closed walk is actually a self-loop. The induction hypothesis assumes that the lemma holds for all closed walks of length $l \leq m$. Consider now a closed walk of length $l=m+1$. If all the vertices of the walk (except for the first and the last one) are distinct, then the walk is a cycle, in which case the lemma holds immediately. If there exist in the walk repetitions of vertices, consider the first vertex (say v) that is encountered twice in the walk. The part of the walk that is determined by the two occurrences of v is a cycle. We remove this cycle from the walk, getting a new walk of length $\leq m$. The induction step then follows immediately. ∎

Example 5.1. Consider the directed weighted graph $G=(V,E)$ with $V=\{v_1,v_2,v_3,v_4\}$ and $E=\{e_1,e_2,e_3,e_4,e_5\}$, where $e_1=(v_1,v_2,w_1)$, $e_2=(v_2,v_3,w_2)$, $e_3=(v_3,v_4,w_3)$, $e_4=(v_4,v_2,w_4)$ and $e_5=(v_3,v_1,w_5)$. Consider the closed walk $W=v_1e_1v_2e_2v_3e_3v_4e_4v_2e_2v_3e_5v_1$. Then, there exist two cycles, namely $C_1=v_2e_2v_3e_3v_4e_4v_2$ and $C_2=v_1e_1v_2e_2v_3e_5v_1$, such that the condition of the above lemma is satisfied.

The following theorem demonstrates that programs passing the cycle sum test are locally stratified. The proof uses Lemmas 5.1 and 5.2.

Theorem 5.2 *Let $\mathbf{P}^*$ be a time-classical logic program that passes the cycle sum test. Then $\mathbf{P}^*$ is locally stratified.*

Proof: Assume that $\mathbf{P}^*$ is not locally stratified. This means that there exists an infinite increasing sequence $\mathbf{A}_1 < \mathbf{A}_2 < \cdots$ of atoms of the Herbrand base of $\mathbf{P}^*$. For each atom in the sequence there exists a clause of the program whose ground instantiation has the atom as its head. As the sequence is infinite, there exist infinitely many atoms of the sequence that correspond to the same clause of the program, say $\mathbf{C}$. These atoms form an infinite subsequence $\mathbf{B}_1 < \mathbf{B}_2 < \cdots$ of the initial sequence.

Consider now two atoms $\mathbf{B}_i$ and $\mathbf{B}_{i+1}$ from the new sequence. The first argument of $\mathbf{B}_i$ is of the form $\mathbf{s}^k(\mathbf{e})$, where $k \geq 0$ and $\mathbf{e}$ is either $\mathbf{0}$ or a non time-term. Moreover, by Definition 5.3, there exists a directed walk from $\mathbf{B}_i$ to $\mathbf{B}_{i+1}$ in the dependency graph of $\mathbf{P}^*$. By Lemma 5.1, this walk corresponds to a walk in the cycle sum graph of $\mathbf{P}^*$, which is in fact a closed walk because $\mathbf{B}_i$ and $\mathbf{B}_{i+1}$ are atoms having the same predicate name. Using Lemma 5.2 and the fact that $\mathbf{P}^*$ passes the cycle sum test, we deduce that the sum of the weights that correspond to the edges of the walk is positive. This implies that the first argument of $\mathbf{B}_i$ leads the first argument of $\mathbf{B}_{i+1}$ by a positive amount (i.e., the first argument of $\mathbf{B}_{i+1}$ is either of the form $\mathbf{s}^m(\mathbf{e})$ or $\mathbf{s}^m(\mathbf{0})$, with $m < k$). Consequently, the first arguments of the members of the sequence $\mathbf{B}$ decrease in complexity and therefore the sequence can not be infinite. This is a contradiction, which implies that program $\mathbf{P}^*$ is locally stratified. ■

Theorem 5.3 *Let $\mathbf{P}^*$ be a time-classical logic program that passes the cycle sum test. Then $\mathbf{P}^*$ has a unique perfect model which is also normal.*

Proof: By Theorem 5.2, $\mathbf{P}^*$ is locally stratified, which means that it has a unique perfect model M^* [Prz88]. This model obviously contains the set $Nat = \{\mathbf{nat}(\mathbf{s}^k(\mathbf{0})) \mid k \geq 0\}$ due to the two clauses in $\mathbf{P}^*$ that define the predicate **nat**. Moreover, it contains the set $Universe = \{\mathbf{universe}(\mathbf{e}) \mid \mathbf{e} \in U_{\mathbf{P}}\}$ due to the clauses that define the predicate **universe**. We need to further show that M^* does not contain any other atom regarding the predicates **nat** and **universe**, and that it only contains atoms of the form $\mathbf{p}(\mathbf{s}^k(\mathbf{0}), \mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$. Consider the interpretation N^* that results if we remove from M^* all atoms regarding the predicate **nat** that do not belong to Nat, all atoms regarding **universe** that do not belong to $Universe$, and all atoms that are not of the form $\mathbf{p}(\mathbf{s}^k(\mathbf{0}), \mathbf{e}_0, \ldots, \mathbf{e}_{n-1})$. We have $N^* \subset M^*$. It can be easily shown that every ground instance of a clause in $\mathbf{P}^*$ is true under N^*. Therefore, N^* is also a model of $\mathbf{P}^*$, which contradicts the fact that perfect models are minimal models [Prz88]. ■

From the above discussion, we conclude that given a temporal logic program $\mathbf{P}$, if its classical counterpart $\mathbf{P}^*$ passes the cycle sum test, then $\mathbf{P}^*$ is guaranteed to have a unique perfect model M^* which is normal. Therefore, by Theorem 3.1, there exists a temporal Herbrand model M of $\mathbf{P}$, which we take as the intended meaning of $\mathbf{P}$.

6 Related and Future Work

In this paper we have developed a syntactic test (the *cycle sum test*) for temporal logic programs with negation. Programs that pass the test are guaranteed to have a well-defined meaning. In particular, the test represents an application of the notion of local stratification in the context of temporal logic programming. In this section we discuss related research results as well as possible future extensions of the proposed technique.

Related Work: During the time that this paper was under review, the author became aware of two other approaches for adding negation to temporal logic programming, namely *XY-stratification* [ZAO93] and *state-stratification* [Lud98]. Both approaches however apply to temporal languages that have a rather restricted syntax, and are therefore in this respect less general than the cycle-sum approach introduced in this paper.

The *XY-stratification* approach was developed for the language *XY-Datalog* [ZAO93], which has been proposed as a formalism for combining active and deductive rules. In XY-Datalog, rules have a distinguished first argument (the *stage argument*), which represents the notion of time. A stage argument can have three possible values, namely `nil`, `I` and `s(I)`. The notion of XY-stratification is defined on programs whose rules are either *X-rules* or *Y-rules*. An XY-Datalog rule r is an:

- *X-rule* if all the stage arguments of r are equal to a single variable `I`.
- *Y-rule* if the head of r has stage argument `s(I)`, some positive goal of r has stage argument `I`, and all the remaining stage arguments are either `I` or `s(I)`.

Clearly, the syntax of the rules on which XY-stratification is defined, corresponds to a language that is restricted when compared to the Chronolog language adopted in this paper. Moreover, the test for determining whether a program is XY-stratified differs from the cycle-sum approach (the former test is based on producing and examining the so-called *primed* version [ZAO93] of the source program).

The *state-stratification* approach applies to the language *Statelog*, which only allows *progressive rules* [Lud98]. A progressive rule is one in which the temporal difference between the head and every atom of the body is positive. More formally, a progressive rule is of the form:

$$[S + k_0]H \;\leftarrow\; [S + k_1]B_1, \ldots, [S + k_n]B_n$$

where $k_0 \geq k_i$, for $i = 1, \ldots, n$. Progressive Statelog rules are clearly a proper subclass of Chronolog rules. Consider for example the following Chronolog program:

```
first p(a).
p(X) ← next q(X).
next next q(X) ← ¬ p(X).
```

The above program is not progressive in the sense of Statelog (due to the second clause) but it has a well-defined meaning under the cycle-sum approach.

It should be noted however that the state stratification idea has an underlying philosophy that is related to the approach presented in this paper: the notion of *leap* defined in [Lud98] is similar to the notion of *temporal difference* introduced in this paper. The difference between the two approaches is that in state stratification the leaps are always positive or zero while in the cycle-sum approach the temporal difference can be any integer value or $-\infty$.

Future Work: One shortcoming of the cycle sum approach is that (as noted in section 4) certain positive programs are not directly acceptable by the test. The obvious solution is to define as acceptable all the programs that are either positive or pass the cycle sum test. We conjecture that an extension of the test can be devised that can handle a significantly broader class of temporal logic programs with negation and which directly accepts positive programs. Such an extension would probably rely on a generalized cycle sum graph, which apart from the integer weights would also contain indications of whether edges are positive or negative. The notion of "sum" across cycles would then have to be generalized to take into account the new indications. Alternatively, one could apply the existing cycle-sum test only on those strongly connected components of the generalized cycle sum graph that contain at least one negative edge.

Another interesting topic for further research would be the consideration of other temporal logic programming languages that use an extended set of temporal operators or that are based on different notions of *time*. One interesting such case is that of *branching-time* logic programming [RGP98] in which the set of possible worlds of the underlying branching-time logic is the set of lists of natural numbers. Such a language would require a more powerful cycle test which would have to be applicable to the more complicated underlying set of possible worlds.

Acknowledgments: The author would like to thank Manolis Gergatsoulis and Christos Nomikos for their valuable comments on earlier versions of the paper.

References

[AB94] K. Apt and R. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19,20:9–71, 1994.

[ABW88] K.R. Apt, H.A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, CA, 1988.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[CN99] C. Coutras and C. Nomikos. On the Computational Complexity of Stratified Negation in Linear-Time Temporal Logic Programming. In *Proceedings of the 12th International Symposium on Languages for Intensional Programming*, pages 222–231, June 1999. Athens, Greece.

[Har72] F. Harary. *Graph Theory.* Addison-Wesley, 1972.

[Llo87] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[Lud98] B. Ludäscher. *Integration of Active and Deductive Database Rules.* PhD thesis, Institut für Informatik, Universität Freiburg, 1998. (available from http://www.sdsc.edu/˜ludaesch).

[Mat95] S.G. Matthews. An Extensional Treatment of Lazy Dataflow Deadlock. *Theoretical Computer Science*, 151:195–205, 1995.

[OW92a] M. A. Orgun and W. W. Wadge. Theory and practice of temporal logic programming. In L. Farinas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*, pages 23–50. Oxford University Press, 1992.

[OW92b] M. A. Orgun and W. W. Wadge. Towards a unified theory of intensional logic programming. *The Journal of Logic Programming*, 13(4):113–145, August 1992.

[PP90] H. Przymusinska and T. Przymusinski. Semantic Issues in Deductive Databases and Logic Programs. In R. Banerji, editor, *Formal Techniques in Artificial Intelligence*, pages 321–367. North Holland, 1990.

[Prz88] T. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[RGP98] P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. Branching-Time Logic Programming: the Language Cactus and its Applications. *Computer Languages*, 24:155–178, 1998.

[Ron98] P. Rondogiannis. Negation in Chronolog. In *Proceedings of the 11th International Symposium on Languages for Intensional Programming*, pages 89–97, May 7-9 1998. Palo Alto, USA.

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[Wad81] W. W. Wadge. An Extensional Treatment of Dataflow Deadlock. *Theoretical Computer Science*, 13:3–15, 1981.

[Wad88] W. W. Wadge. Tense logic programming: A respectable alternative. In *Proc. of the 1988 International Symposium on Lucid and Intensional Programming*, pages 26–32, 1988.

[ZAO93] C. Zaniolo, N. Arni, and K. Ong. Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}++$ Approach. In *Proceedings of DOOD-93*, December 6-8 1993. Phoenix, Arizona, USA.