Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 342 (2005) 382–415

Theoretical Computer Science

www.elsevier.com/locate/tcs

Temporal stratification tests for linear and branching-time deductive databases

Christos Nomikos[a], Panos Rondogiannis[b,∗], Manolis Gergatsoulis[c]

[a] *Department of Computer Science, University of Ioannina, P.O. Box 1186, 45 110 Ioannina, Greece*
[b] *Department of Informatics and Telecommunications, University of Athens, Panepistimiopolis, 157 84 Athens, Greece*
[c] *Department of Archive and Library Sciences, Ionian University, Palea Anaktora, Plateia Eleftherias, 49100 Corfu, Greece*

Received 15 May 2003

Communicated by G. Levi

Abstract

We consider the problem of extending temporal deductive databases with stratified negation. We argue that the classical stratification test for deductive databases is too restrictive when one shifts attention to the temporal case. Moreover, as we demonstrate, the (more general) local stratification approach is impractical: detecting whether a temporal deductive database is locally stratified is shown to be co-NP hard (even if one restricts attention to programs that only use one predicate symbol and two constants). For these reasons we define *temporal stratification*, an intermediate notion between stratification and local stratification. We demonstrate that for the temporal deductive databases we consider, temporal stratification coincides with local stratification in certain important cases in which the latter is polynomial-time decidable. We then develop two algorithms for detecting temporal stratification. The first algorithm applies to linear-time temporal deductive databases and it is efficient and more general than existing approaches; however, the algorithm sacrifices completeness for efficiency since it does not cover the whole class of temporally stratified programs. The second algorithm applies to branching-time temporal deductive databases (which include as a special case the linear-time ones). This algorithm is more expensive from a computational point of view, but it covers the whole class

∗Corresponding author.
E-mail addresses: cnomikos@cs.uoi.gr (C. Nomikos), prondo@di.uoa.gr (P. Rondogiannis), manolis@ionio.gr (M. Gergatsoulis).

of temporally stratified programs. We discuss the relative merits of the two algorithms and compare them with other existing approaches.

Keywords: Temporal deductive databases; Temporal logic programming; Stratified negation

1. Introduction

Temporal deductive databases [35,21,4,3] (and more generally temporal logic programming languages [22,10]) are promising formalisms that appear to have interesting applications. Although the field of temporal deductive databases is far from new, many notions that have been widely studied for classical deductive databases have only recently been considered for the temporal case. Negation is one such concept: although in the classical case many elegant semantic approaches have been developed during the last 20 years (see for example [26,2]), the temporal case has not been extensively studied and there only exist a few sparse results. The existing approaches focus on deriving a useful notion of stratified negation for temporal deductive databases [34,19,29,12,15,18]. However, in these techniques the syntax of the underlying temporal formalisms is rather restricted. Moreover, the notion of time that is adopted is discrete and linear (although there exist other interesting and useful notions of time).

1.1. The problem

The problem of adding stratified negation to (even linear-time) temporal formalisms is not trivial: if one tries to blindly transfer the classical stratification test [1] to a temporal deductive database setting, then many programs that appear to have a clear semantics must be rejected. Consider for example the simple Chronolog [33] program simulating the operation of the traffic lights: [1]

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

The above program is obviously a meaningful one. However, if one uses the classical stratification approach, this program has to be rejected since it contains cyclic dependencies of a predicate name through negation. A stratification test for a temporal deductive database formalism has to take into careful consideration the (sometimes implicit) time parameter on which such a formalism is based. One idea is to use local stratification [27] instead of simple stratification. This would solve the problem of the time parameter but it would create new problems instead. It is well-known that deciding whether a logic program is locally stratified is undecidable [5,24]; moreover, as we demonstrate in this paper (Theorem 18) deciding

[1] It should be noted that the syntax of Chronolog does not support negation, but the meaning of the above program should be clear.

whether a given Datalog$^{\neg}$ program is locally stratified, is co-NP hard (which easily implies that the local stratification problem for the temporal formalisms we consider is co-NP hard). Therefore, we have to choose between a very restrictive form of stratification and a very broad one which is computationally impractical.

Fortunately, there is a middle road to follow (and this is actually the road implicitly taken by all of the existing approaches [34,19,29,20,15,18]). The basic idea is to find an intermediate notion of stratification which is not trivial and which can be detected efficiently. There still exist, however, two important issues that remain unanswered by all existing approaches:

(1) The linear-time temporal stratification tests that have been proposed so far are in general narrow in scope: the syntax of the underlying temporal deductive database formalisms on which these tests apply is rather restricted. For example, it is often required that the temporal references of the predicates in a temporal program cannot be arbitrary but instead they have to obey to some predefined pattern. Therefore, there still remains the quest for a test that is both efficient and that applies to a very general linear-time temporal deductive database formalism.
(2) All the existing tests treat languages in which time has a linear (and discrete) flow: the set of time-points is actually the set of natural numbers. There exist however richer temporal formalisms. For example, the language *Cactus* [31] (and its function-free subset called *Branching-time Datalog* [30]) is a branching-time logic programming formalism; similarly, $\mathit{Datalog}_{nS}$ [6,7] has an extended notion of time built in its design (and has linear-time as a very special case, called $\mathit{Datalog}_{1S}$ [4]). It should be noted that branching-time languages can express certain problems in a natural way [31] and have recently found interesting applications in the area of Datalog optimizations [30,25]. Therefore, the second issue that arises is the derivation of an efficient temporal stratification test for this more general notion of time.

The above two questions are the main issues tackled in this paper.

1.2. Contributions

The main contributions of the work presented in this paper can be summarized as follows:

(1) We argue that stratification is too restrictive and local stratification is impractical when one considers temporal deductive databases. In particular, we demonstrate that the local stratification problem for the temporal languages we adopt is co-NP hard (even if one restricts attention to programs with one predicate symbol and two constants). On the positive side, we demonstrate that this problem is actually decidable; this is a non-trivial fact since for these languages the temporal Herbrand base is infinite due to the time parameter. It is well-known that for classical logic programs (whose Herbrand base is also infinite) the local stratification problem is undecidable [5,24].
(2) We define the notion of *temporal stratification* and argue that it is an intermediate notion between stratification and local stratification. Moreover, we demonstrate that for the temporal formalisms we consider, temporal stratification coincides with local stratification in certain important cases in which the latter is polynomial-time decidable.
(3) We propose a temporal stratification test for linear-time temporal deductive databases. The proposed approach is an extension of the cycle-sum test [29] and actually

remedies its main shortcoming since it does not reject any positive programs. In fact, the new test accepts a significant class of programs with negation which is strictly greater than the classes of programs accepted by the existing temporal stratification tests [34,19,29,12,15,18]. However, the algorithm sacrifices completeness for efficiency since it does not cover the whole class of temporally stratified programs.

(4) We propose a temporal stratification test for branching-time temporal deductive databases (which include linear-time ones as a special case). More specifically, the test can be applied to the language Branching-time Datalog$^{\neg}$ that supports a branching notion of time (with appropriate modifications the technique can also be applied to other similar temporal formalisms). Since existing stratification tests for temporal languages only apply to linear time, the proposed technique is a generalization and extension of previous approaches. Another unique characteristic of this algorithm is that it covers the whole class of temporally stratified programs. As a trade-off however, the algorithm is computationally more expensive than the one for the linear-time case (but still has a polynomial-time complexity). Finally, this is the first (to our knowledge) temporal stratification test that can successfully cope with programs whose clauses may contain temporally ground atoms (*canonical atoms*).

Summarizing, we believe that the results obtained in this paper can be used in order to add a useful form of negation to temporal deductive databases (of either linear or branching time). Hopefully, the proposed techniques can be embedded in existing temporal systems and enhance their capabilities.

1.3. Structure of the paper

The rest of the paper is organized as follows: Section 2 gives an introduction to the syntax and semantics of the temporal languages used in the paper, and Section 3 introduces the notions of stratification and local stratification for these languages. Section 4 motivates and defines the notion of temporal stratification. Section 5 extends the cycle-sum approach [29] obtaining an extended temporal stratification test for linear-time temporal deductive databases. Section 6 introduces a novel stratification test for branching-time temporal deductive databases. Section 7 compares the two proposed approaches and presents their relative merits. Section 8 gives a comparison with related work and Section 9 discusses directions for future work.

2. Temporal deductive databases

In this section we define the basic notions that will be used in the rest of the paper. In the following, we assume a familiarity with the basic concepts behind deductive databases [28] and logic programming [17].

The languages that will be used throughout the paper are temporal (they have an implicit parameter which encodes the notion of time). In certain points of the paper we will need to refer to the (non-temporal) language Datalog$^{\neg}$, which is the extension of Datalog that allows negative literals in clause bodies (see for example [28]). The two languages that will be the main focus of our study, are:

- The language *Linear-time Datalog*$^\neg$ which is a deductive database language that is based on a linear notion of time.
- The language *Branching-time Datalog*$^\neg$ which is a deductive database language that supports a branching notion of time.

The above two formalisms have their roots in the Chronolog [33,23] and Cactus [31] temporal logic programming languages. The main difference is that the formalisms we consider in this paper support negation (which is not the case for Chronolog and Cactus). Moreover, the two languages we consider here do not have function symbols (without this being an essential restriction since the tests we describe can be lifted to the more general framework of temporal logic programming).

As, it will be discussed later in this section, Linear-time Datalog$^\neg$ can be seen as a special instance of Branching-time Datalog$^\neg$. For this reason, we start by presenting the syntax and the semantics of Branching-time Datalog$^\neg$ (and then obtain as special cases the syntax and the semantics of Linear-time Datalog$^\neg$).

Every atom in a Branching-time Datalog$^\neg$ program is preceded by a *temporal reference*, which is a (possibly empty) sequence of the temporal operators `first` and $\texttt{next}_i$, $i \geqslant 0$. A temporal reference of the form $\texttt{first}\ \texttt{next}_{i_1} \cdots \texttt{next}_{i_k}$, where $k \geqslant 0$, is called *canonical*. A temporal reference of the form $\texttt{next}_{i_1} \cdots \texttt{next}_{i_k}$ is said to be *open*. A *temporal atom* is an atom preceded by either a canonical or an open temporal reference. A *canonical* (resp. *open*) temporal atom is a temporal atom whose temporal reference is canonical (resp. open). Given a temporal atom A, the temporal reference of A is denoted by $\mathit{time}(A)$. A *temporal clause* in Branching-time Datalog$^\neg$ is a formula of the form:

$$H \leftarrow A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m,$$

where $H, A_1, \ldots, A_n, B_1, \ldots, B_m$ are temporal atoms and $n, m \geqslant 0$. The atoms $A_1, \ldots, A_n$ are said to *occur positively* in the clause while the atoms $B_1, \ldots, B_m$ *negatively*. If $n = m = 0$, the clause is said to be a *unit temporal clause*. It is very common in deductive databases to partition the set of predicates/atoms into *intensional* (or IDBs) and *extensional* (or EDBs). However, for the purposes of this paper, this distinction does not play any important role. Therefore, the terminology we adopt in this paper is more closely related to that of logic programming [17].

A Branching-time Datalog$^\neg$ program is a finite set of temporal clauses.[2] A *canonical temporal clause* is a temporal clause in which all atoms that occur in it are canonical. A *canonical temporal instance* of a temporal clause C is a canonical temporal clause which is obtained by applying the same canonical temporal reference to all open atoms of C.

In all the above discussion, the temporal references that are used are either canonical or open. One might possibly wonder why more general forms of temporal references have been excluded (e.g. $\texttt{next}_1\ \texttt{first}\ \texttt{next}_2$). However, disallowing such temporal references in program clauses is not a real restriction since, as it can be easily shown [31], the operators that appear before the rightmost `first` operator are superfluous and can be eliminated.

[2] In other words, we assume that a program in Branching-time Datalog$^\neg$ consists of both rules and facts.

Before presenting the semantics of Branching-time Datalog$^{\neg}$, we give two examples that illustrate the above ideas (the first example does not use negation while the second does).

Example 1. Consider the following program which performs a hypothetical tour of cities:

```
first city(athens).
 next₁ city(X) ← by_sea(Y,X),city(Y).
 next₂ city(X) ← by_air(Y,X),city(Y).
```

The tour starts from athens. At each point of the tour one can consider to move to a next city either by_sea or by_air. The temporal operators next$_1$ and next$_2$ reflect the type of connection between the cities that is followed in each case. Therefore, if by_sea(athens,lisboa) and by_air(lisboa,london) are two facts that are added to the above clauses, then the canonical temporal atom first next$_1$ next$_2$ city(london) is a logical consequence of the above program. Notice that the sequence of the indices of the next operators when read from left to right reflect the types of connections that have been used during the trip.

Example 2. The following program simulates the painting of the nodes of a binary tree with three colors, namely red, green and blue, by following certain simple rules that involve negation:

```
first tree(green).
next₁ tree(red) ← ¬ tree(red).
next₁ tree(green) ← tree(red).
next₂ tree(X) ← color(X), ¬ tree(X), ¬ next₁ tree(X).
color(green).
color(red).
color(blue).
```

The above clauses can be read as follows: "the root of the tree is colored green; the left child of a node of the tree is colored red if the node itself is not red; the left child of a node is colored green if the node itself is red; finally, the right child of a node can be colored with a color that is neither used to color the node itself nor its left child".

In Section 4, the notion of temporal stratification for Branching-time Datalog$^{\neg}$ programs will be defined. The above program is actually a temporally stratified one (intuitively, it does not contain temporal circularities through negation). This fact is not immediately obvious but it can be demonstrated through the technique developed in Section 6. Actually, it can be shown that the intended model of the above program represents a unique and balanced binary tree of infinite depth (which has been appropriately colored).

Branching-time Datalog$^{\neg}$ is based on a relatively simple *branching-time logic* (*BTL*). In *BTL* time has an initial moment and flows towards the future in a tree-like way. The set of moments in time can be modeled by the set *List*(ω) of lists of natural numbers. The empty list [ ] corresponds to the beginning of time and the list $[i|t]$ (that is, the list with head i, where $i \in \omega$, and tail t) corresponds to the ith alternative successor of the moment identified by the list t. *BTL* uses the temporal operators first and next$_i$, $i \in \omega$.

The operator `first` is used to identify the first moment in time, while $\texttt{next}_i$ refers to the *i*th alternative successor of the current moment in time. The syntax of *BTL* extends the syntax of first-order logic with two formation rules: if A is a formula then so are $\texttt{first}\, A$ and $\texttt{next}_i\, A$. The semantics of temporal formulas of *BTL* are given using the notion of *branching temporal interpretation* [31]:

Definition 3. A *branching temporal interpretation* or simply a *temporal interpretation I* of the temporal logic *BTL* comprises a non-empty set D, called the domain of the interpretation, together with an element of D for each variable or constant symbol and an element of $[List(\omega) \rightarrow 2^{D^n}]$ for each n-ary predicate symbol.

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

Definition 4. The semantics of the elements of the temporal logic *BTL* are given recursively as follows:
(1) For any n-ary predicate symbol p and terms $e_0, \ldots, e_{n-1}$, $\models_{I,t} p(e_0, \ldots, e_{n-1})$ iff $\langle I(e_0), \ldots, I(e_{n-1})\rangle \in I(p)(t)$;
(2) $\models_{I,t} \neg A$ iff it is not the case that $\models_{I,t} A$;
(3) $\models_{I,t} A \wedge B$ iff $\models_{I,t} A$ and $\models_{I,t} B$;
(4) $\models_{I,t} (\forall x) A$ 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 element d;
(5) $\models_{I,t} \texttt{first} A$ iff $\models_{I,[\,]} A$;
(6) $\models_{I,t} \texttt{next}_i\ A$ iff $\models_{I,[i|t]} A$.

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.

When we focus on Branching-time Datalog$^{\neg}$ programs, the interpretations we consider are Herbrand ones. As usual, the *Herbrand universe* U_P of a program P is the set of all constant symbols that appear in P. *Temporal Herbrand interpretations* can be regarded as subsets of the *temporal Herbrand base* B_P of P, consisting of all *canonical ground temporal atoms* whose predicate symbols appear in P and whose arguments are terms in the Herbrand universe U_P of P. In particular, given a subset H of B_P, we can define a temporal Herbrand interpretation I by the following:

$$\langle c_0, \ldots, c_{n-1}\rangle \in I(p)([i_1, \ldots, i_k]) \ \text{ iff } \quad \texttt{first}\ \texttt{next}_{i_k} \cdots \ \texttt{next}_{i_1}\ p(c_0, \ldots, c_{n-1}) \in H.$$

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 temporal Herbrand model.

An important subset of Branching-time Datalog$^{\neg}$ is obtained when one considers a single $\texttt{next}_i$ operator. In this case, the language obtained is in fact a linear-time one, and it is more convenient to model the underlying set of time moments by the set ω of natural numbers (the empty list corresponds to 0 and the list $[i, \ldots, i]$ containing k consecutive i’s corresponds to the natural number k). 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. We will often write $\texttt{next}^k$ to represent a sequence of k `next` operators. The language obtained in this way will be called Linear-time Datalog$^\neg$. Since the underlying set of time moments of this language is ω, a temporal interpretation now assigns to each n-ary predicate symbol, an element of $[\omega \to 2^{D^n}]$.

The satisfaction relation $\models$ of the underlying linear-time logic is defined as before, the only difference being the simpler notion of time. The two semantic equations that are simplified are:
(5) $\models_{I,t}$ `first` A *iff* $\models_{I,0} A$
(6) $\models_{I,t}$ `next` A *iff* $\models_{I,t+1} A$
All the other concepts regarding Linear-time Datalog$^\neg$ are special cases of the corresponding concepts for Branching-time Datalog$^\neg$. An example program in this simpler language is the “traffic-lights” one given in the introductory section.

We close this section with a brief discussion on the operational semantics (i.e., proof procedures) that can be defined for branching-time languages. More specifically, if we restrict attention to Branching-time Datalog (i.e., the subset of Branching-time Datalog$^\neg$ that does not use negation), then a bottom-up proof procedure can be easily defined using an immediate consequence operator (see for example [30]). For the more general case of branching-time logic programming [31], a resolution-based proof system can be defined (see [31, Section 5]). Additionally, a more sophisticated proof procedure for such languages is defined in [11]. However, for Branching-time Datalog$^\neg$ there does not exist at present an appropriate proof procedure. It is obvious that such a proof procedure would rely on the semantics that one adopts for negation. We believe that the temporal stratification notion that we define in this paper can form the basis for a proof procedure for Branching-time Datalog$^\neg$.

3. Local stratification in temporal deductive databases

In this section we formally define the notions of *stratification* and *local stratification* for Branching-time Datalog$^\neg$ programs (and therefore also for Linear-time Datalog$^\neg$ ones). These notions are actually easy extensions of the corresponding concepts in classical deductive databases [1,27].

Definition 5. Let P be a Branching-time Datalog$^\neg$ program. Then, P is called *stratified* if it is possible to partition the set of all predicate symbols in P into disjoint sets (called *strata*) $S_0, S_1, \ldots, S_r$, so that for every clause

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

in P such that the predicate symbol of H belongs to S_k with $0 \leqslant k \leqslant r$, the following hold:
- The predicate symbol of A_i belongs to $\bigcup_{j \leqslant k} S_j$, for $1 \leqslant i \leqslant n$.
- The predicate symbol of B_i belongs to $\bigcup_{j < k} S_j$, for $1 \leqslant i \leqslant m$.

Local stratification is defined in a similar way, in terms of the temporal Herbrand base instead of the set of predicate symbols:

Definition 6. Let P be a Branching-time Datalog$^{\neg}$ program. Then, P is called *locally stratified* if it is possible to partition its temporal Herbrand base B_P into disjoint sets (called *strata*) $S_0, S_1, \ldots, S_\alpha, \ldots$, where $\alpha < \gamma$ and γ is a countable ordinal so that for every canonical ground instance

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

of a clause in P such that $H \in S_\alpha$ with $\alpha < \gamma$, the following hold:

- A_i belongs to $\bigcup_{j \leqslant \alpha} S_j$, for $1 \leqslant i \leqslant n$.
- B_i belongs to $\bigcup_{j < \alpha} S_j$, for $1 \leqslant i \leqslant m$.

The following theorem is easy to prove:

Theorem 7. *If a Branching-time Datalog$^{\neg}$ program is stratified then it is locally stratified*.

In the rest of the paper, we will use alternative, more convenient, definitions of stratification and local stratification, given by Theorems 9, 12, and 14 that follow. These definitions use the graph-theoretic notions [13] of *directed walk* and *closed walk*. Recall that a directed walk in a graph G is a (finite or infinite) sequence of vertices and edges, $v_0 e_1 v_1 \cdots v_{k-1} e_k v_k \cdots$ in which e_i is an edge from v_{i-1} to v_i. A closed walk is a directed walk that has the same first and last vertices. In the following we will also need Definitions 8, 10, and 11 which are the Branching-time Datalog$^{\neg}$ analogs of the corresponding definitions that have been proposed [27,26] for classical deductive databases.

Definition 8. Let P be a Branching-time Datalog$^{\neg}$ program. The *predicate dependency graph* PDG_P of P is a graph whose vertex set is the set of predicate symbols in P and whose edges are determined as follows: there exists an edge from p to q iff there exists a clause C in P such that p is the predicate symbol in the head of C and q appears as a predicate symbol in the body of C. If there is a clause whose head predicate is p and its body contains a negated atom whose predicate is q then the edge from p to q is called *negative*.

Theorem 9. *A Branching-time Datalog$^{\neg}$ program P is stratified if and only if its predicate dependency graph does not contain any cycle that passes through negative edges*.

Proof. The proof is analogous to the one for the classical case [1]. $\square$

Definition 10. Let P be a Branching-time Datalog$^{\neg}$ program. The *atom dependency graph* ADG_P of P is a graph whose vertex set is the temporal Herbrand base B_P of P and whose edges are determined as follows: if A and B are two temporal atoms in B_P, there exists a directed edge from A to B if and only if there exists a canonical ground temporal instance of a clause in P whose head is A and whose body contains either B or $\neg B$. If there is a canonical ground temporal instance of a clause whose head is A and its body contains $\neg B$ then the edge from A to B is called *negative*.

Definition 11. Let P be a Branching-time Datalog$^\neg$ program. For any two canonical ground temporal atoms A and B in B_P we write $A < B$ if there exists a directed walk in the atom-dependency graph ADG_P leading from A to B and passing through at least one negative edge. We call the relation $<$ the *priority relation* between canonical ground temporal atoms.

Now, the following two theorems provide alternative definitions for local stratification in Branching-time Datalog$^\neg$ (actually the first one is a temporal analogue of Theorem 3, p. 206, of [27], and its proof is similar):

Theorem 12. *A Branching-time Datalog$^\neg$ program P is locally stratified if and only if every increasing sequence of canonical ground temporal atoms under $<$ is finite.*

Definition 13. Let P be a Branching-time Datalog$^\neg$ program. A walk in the atom dependency graph ADG_P of P is a *bad walk* if it contains infinitely many occurrences of negative edges.

Theorem 14. *A Branching-time Datalog$^\neg$ program P is locally stratified if and only if its atom dependency graph ADG_P does not contain any bad walk.*

Proof. Infinite increasing sequences under $<$ correspond to bad walks. $\square$

As in the classical case, every locally stratified Branching-time Datalog$^\neg$ program has a unique *perfect* (temporal Herbrand) model. This notion is precisely defined by the following:

Definition 15. Let M and N be two distinct temporal Herbrand models of a Branching-time Datalog$^\neg$ program P. Then, N is called *preferable* to M if for every canonical ground atom A in $N - M$, there exists a canonical ground atom B in $M - N$ such that $A < B$. A temporal Herbrand model M of P is called *perfect* if there are no temporal Herbrand models of P that are preferable to M.

Theorem 16. *Every locally stratified Branching-time Datalog$^\neg$ program P has a unique perfect temporal Herbrand model.*

Again, the proof of the above theorem is similar to the proof of Theorem 4, p. 208, of [27].

We conclude this section with a remark that is quite important for practical reasons because it demonstrates that the unit clauses (facts) of a given program can be ignored when one tests the program for local stratification. Actually, the following proposition (adapted from Proposition 3.3 of [24] concerning general logic programs) can be easily established:

Proposition 17. *Let P be a Branching-time Datalog$^\neg$ program. Then P is locally stratified if and only if the program consisting of the non-unit clauses of P is locally stratified.*

Notice that the above proposition does not simply suggest that the unit clauses are not used in the construction of the atom dependency graph of a program; it additionally suggests that the constant symbols that appear only in the unit clauses do not play any role with respect to local stratification.

4. Temporal stratification

The notions of stratification and local stratification presented in the last section are two possible candidates that one can consider when attempting to add negation to a temporal deductive database. However, (classical) stratification is too restrictive for such formalisms since it completely ignores their temporal nature. More specifically, applying the classical stratification test to a Branching-time Datalog$^{\neg}$ program (for example, the program in the introductory section) would in many cases result to the rejection of the program (although the program might appear to be meaningful).

Consider on the other hand the addition of locally stratified negation to Branching-time Datalog$^{\neg}$. It is well known that local stratification is undecidable for logic programs with function symbols [5]. For Branching-time Datalog$^{\neg}$ the problem of local stratification is decidable (as we show in Theorem 62). Therefore, at first sight local stratification seems to be a reasonable choice since it takes into consideration the temporal aspect of the language. However, as the following theorem demonstrates, local stratification for Datalog$^{\neg}$ is a co-NP hard problem, and this easily implies (Corollary 20) that local stratification for Branching-time Datalog$^{\neg}$ is also a co-NP hard problem. This result suggests that adding local stratification to the languages we consider is impractical: there does not exist a test that can decide *efficiently* whether a given Branching-time Datalog$^{\neg}$ program is locally stratified.

Theorem 18. *The Local Stratification of Datalog$^{\neg}$ programs is a co-NP hard problem.*

Proof. It is sufficient to reduce any NP-hard problem to the complement of Local Stratification of Datalog$^{\neg}$. It is well-known that deciding if a given graph contains a Hamilton cycle (that is a cycle that passes exactly once from every vertex) is an NP-hard problem. We will reduce this problem to the complement of the Local Stratification of Datalog$^{\neg}$. More specifically, given a graph $G(V, E)$, we will construct a Datalog$^{\neg}$ program P such that G contains a Hamilton cycle if and only if P is not locally stratified. Notice that for both problems the size of the input is the length of its representation as a string, which is the standard measure used in complexity theory.

Assume that G consists of n vertices labeled $1, 2, \ldots, n$ and m edges. Then P is constructed so as to consist of $m+1$ clauses, and contains a single predicate symbol p of arity $2n$. Ground atoms of the Herbrand base of P represent the states while traversing a Hamilton cycle of G. In any atom that may appear in the ground instantiation of a clause in P exactly one of the first n arguments in p has the value 1, indicating the current vertex. The last n arguments are used to mark the vertices of G visited so far.

For every edge $(i, j) \in E$, P contains a corresponding clause $C_{(i,j)}$:

$$p(s_1, s_2, \ldots, s_n, u_1, u_2, \ldots, u_n) \leftarrow p(t_1, t_2, \ldots, t_n, v_1, v_2, \ldots, v_n),$$

where

- $s_i = 1, s_k = 0$, for $1 \leqslant k \leqslant n,\ k \neq i$,
- $t_j = 1, t_k = 0$, for $1 \leqslant k \leqslant n,\ k \neq j$,
- $u_j = 0, v_j = 1, u_k = v_k = X_k$ for all k such that $1 \leqslant k \leqslant n,\ k \neq j$ (X_k's are distinct variables).

The intuitive meaning of clause $C_{(i,j)}$ is that traversing edge (i, j) while following a Hamilton cycle changes the current vertex from *i* to *j* and also adds *j* to the set of visited vertices.

Moreover, *P* contains the following clause $C_\neg$:

$$p(1, 0, \ldots, 0, 1, 1, \ldots, 1) \leftarrow \neg p(1, 0, \ldots, 0, 0, 0, \ldots, 0).$$

The head of $C_\neg$ is the state in which the current vertex is 1 and all the vertices are visited and the body of $C_\neg$ is the state in which the current vertex is 1 and no vertex is visited. In other words, traversing this edge of the atom dependency graph ADG_P, has the effect of resetting the set of all visited vertices when completing a Hamilton cycle.

The size of program *P* as well as the time required for its construction are polynomial to the size of *G*. We now prove that *G* contains a Hamilton cycle if and only if *P* is not locally stratified.

For the one direction assume that the graph *G* contains a Hamilton cycle $x_0, x_1, x_2, \ldots$, $x_n = x_0$. Without loss of generality, we assume that $x_0 = 1$. Let A_i be the atom representing the state after following *i* edges of the Hamilton cycle, starting from vertex 1 (notice that vertex 1 will be considered as visited only at the end of the Hamilton cycle). More specifically,

$$A_i = p(r_1, r_2, \ldots, r_n, w_1, w_2, \ldots, w_n),$$

where $r_j = 1$ if $j = x_i$ and $r_j = 0$ otherwise; $w_j = 1$ if $j = x_k$, for some *k* such that $1 \leqslant k \leqslant i$ and $w_j = 0$ otherwise.

Then, $A_i \leftarrow A_{i+1}$, $0 \leqslant i \leqslant n-1$, is a ground instance of the clause $C_{(x_i, x_{i+1})}$. Moreover, $A_n \leftarrow \neg A_0$ is exactly clause $C_\neg$.

Consequently, $A_0, A_1, A_2 \ldots A_n$ form a cycle in the atom dependency graph of *P*, which contains a negative edge. A bad walk can be constructed by repeating the above cycle infinitely many times. This implies that *P* is not locally stratified.

Conversely, assume that *P* is not locally stratified, that is it contains a bad walk. The only clause in *P* that contains negation is $C_\neg$. Hence, the only negative edge in the atom dependency graph of *P* is the one from the atom *H* in the head of $C_\neg$ to the atom *B* in the body of $C_\neg$. Since a bad walk passes through this edge infinitely many times, there must be a walk $w = A_0 A_1 \cdots A_k$ where $A_0 = B$ and $A_k = H$, that passes through edges corresponding to clauses other than $C_\neg$.

Let x_i be the current vertex in A_i, and suppose that we traverse w from A_0 to A_k. In A_0 no vertex is visited and the current vertex is $x_0 = 1$. According to the construction of *P*, when we move from A_i to A_{i+1}, the number of visited vertices increases by one, which implies that the current vertex of A_{i+1} was not visited in A_i. When we reach A_k the current vertex is 1 and all vertices have been visited exactly once. Consequently $k = n$, which implies that the sequence of the current vertices $x_0, x_1, \ldots, x_n$ is a Hamilton cycle in G. □

As the proof of the above theorem uses only one predicate symbol and two constants, the following is immediate:

Corollary 19. *The Local Stratification of Datalog$^\neg$ programs is a co-NP hard problem even for programs that use one predicate symbol and two constants in the non-unit clauses*.

Based on the above, the following can be easily derived:

Corollary 20. *The Local Stratification of Branching-time Datalog$^\neg$ programs is a co-NP hard problem even for programs that use one predicate symbol and two constants in the non-unit clauses*.

The above result is rather discouraging since it implies that there does not exist an efficient procedure for detecting whether a given Branching-time Datalog$^\neg$ program is locally stratified. It is therefore natural to wonder whether there exists an alternative notion of stratification which is intermediate between classical stratification and local stratification and which can be decided in an efficient way. The following definitions introduce *temporal stratification* which possesses the above properties.

Definition 21. Let P be a Branching-time Datalog$^\neg$ program. Then, the *skeleton* S of P is the propositional program that results after removing all the arguments of the predicates in P.

Notice that the skeleton of a given Branching-time Datalog$^\neg$ program is itself a (simpler in structure) Branching-time Datalog$^\neg$ program. Therefore, all the notions that we have defined so far for Branching-time Datalog$^\neg$ programs transfer directly to skeletons as well.

Example 22. Let P be the following program:

```
first next₂ p(X,Y) ← ¬ q(Y,X).
next₂ p(X,X) ← q(X,X).
q(X,Y) ← next₁ next₂ p(X,Z), ¬ next₁ p(Z,Y).
next₃ r(Z) ← ¬ r(Z).
```

Then, the skeleton S of P is the propositional program:

```
first next₂ p ← ¬ q.
next₂ p ← q.
q ← next₁ next₂ p, ¬ next₁ p.
next₃ r ← ¬ r.
```

Definition 23. A Branching-time Datalog$^\neg$ program P is said to be *temporally stratified* if the skeleton of P is locally stratified.

The theorems that follow establish the fact that for the languages we consider the idea of temporal stratification is an intermediate notion between stratification and local stratification.

Theorem 24. *If a Branching-time Datalog$^\neg$ program is stratified then it is temporally stratified.*

Proof. Let P be a given Branching-time Datalog$^\neg$ program and assume that it is stratified but not temporally stratified. Then, the atom dependency graph of the skeleton of P contains a bad walk. Notice now that for every edge of the atom dependency graph of the skeleton there exists a corresponding edge in the predicate dependency graph PDG_P of P (which has resulted from the same clause in P). This implies that by following the corresponding edges in PDG_P, we can find a walk that has infinitely many negative edges. Since PDG_P has finite size, this walk has to contain a negative cycle. Therefore, P is not stratified (contradiction). □

However, a temporally stratified program is not necessarily stratified as the following example illustrates:

Example 25. Consider the Branching-time Datalog$^\neg$ program:

$\texttt{first next}_1\ \texttt{r(a)} \leftarrow \neg\ \texttt{first r(a).}$

This program is temporally stratified since its skeleton is locally stratified. However, the program is not stratified due to the existence of a negative cycle in its predicate dependency graph.

Theorem 26. *If a Branching-time Datalog$^\neg$ program is temporally stratified then it is locally stratified.*

Proof. Let P be a given Branching-time Datalog$^\neg$ program and assume that it is temporally stratified but not locally stratified. This implies that there exists a bad walk in the atom dependency graph ADG_P of P. Observe now that for every edge of ADG_P there exists a corresponding edge in the atom dependency graph of the skeleton of P. This implies that the latter graph also contains a bad walk. Therefore the skeleton of P is not locally stratified and consequently P is not temporally stratified (contradiction). □

The converse of the above theorem is not true as the following example illustrates:

Example 27. Consider the Branching-time Datalog$^\neg$ program:

$\texttt{first r(a)} \leftarrow \neg\ \texttt{first r(b).}$

The atom dependency graph of the above program does not contain any bad walks and therefore the above program is locally stratified.

On the other hand, the skeleton of the above program is

$\texttt{first r} \leftarrow \neg\ \texttt{first r .}$

The atom dependency graph of this program contains a negative cycle and therefore the skeleton is not locally stratified.

We close this section with a theorem that together with Corollary 20 establish the borderline between tractable and intractable cases of local stratification for Branching-time Datalog$^\neg$ programs (with respect to their number of constants). As Corollary 20 suggests, local stratification for Branching-time Datalog$^\neg$ is co-NP hard even for programs that use one predicate symbol and two constants in the non-unit clauses. This leads to the question of what happens in the remaining cases (namely for programs that use at most one constant in the non-unit clauses and an arbitrary number of predicates). The following theorem demonstrates that in these cases local stratification is equivalent to temporal stratification which (as it will be later proved) is polynomial-time decidable.

Theorem 28. *Let P be a Branching-time Datalog$^\neg$ program that contains at most one constant symbol in the non-unit clauses. Then, P is locally stratified iff it is temporally stratified.*

Proof. The 'if' direction is Theorem 26. For the 'only if' direction, first observe that by Proposition 17, P is locally stratified if and only if the program (say P') consisting of the non-unit clauses of P is locally stratified. Now, since there exists only one constant symbol (say a) in P', from each clause of P' we get only one ground instance. Moreover, a predicate symbol p in the skeleton of P' always corresponds to the same atom in the instantiated program (and vice-versa). Therefore, the atom dependency graphs of P' and of its skeleton are isomorphic. $\square$

5. A temporal stratification test for linear-time deductive databases

In this section we propose a temporal stratification test for Linear-time Datalog$^\neg$. The new test builds on the *cycle-sum test* that was proposed in [29]. The test of [29] constructs the so-called *cycle-sum graph*, a weighted directed graph whose nodes are program predicates. A program passes the test if all the cycles in the graph have positive sums of weights. However, the construction of this graph does not take into consideration the negated atoms of the source program. It is therefore possible (as pointed out in [29]) that many programs will not pass the test although they appear to have a well-defined meaning. In particular, the test even rejects certain temporal programs that do not use negation (in case their cycle-sum graph contains cycles with non-positive sum of weights).

The test that we develop in this section broadens significantly the class of acceptable programs when compared to that of [29] (and of course it trivially accepts all positive programs). Moreover, the class of temporal programs accepted by the new test is strictly greater than the classes of programs accepted by the other existing approaches [34,19,12,15,18].

5.1. The extended cycle-sum test

The basic idea behind the proposed test is that one need not examine the whole cycle-sum graph but only those strongly connected components that contain at least one negatively signed edge. For these components, a careful inspection that takes into consideration the negatively signed edges of each component has to be performed.

As discussed in Section 4, a Linear-time Datalog$^\neg$ program is called temporally stratified when the skeleton of the program is locally stratified. For this reason, all the definitions that will be given below for the extended cycle-sum test will be based on the skeleton of the given program.

Definition 29. Let *P* be a Linear-time Datalog$^\neg$ program, *S* be the skeleton of *P* and *C* be a clause in *S*. Let *H* be the head of *C* and let *A* be an atom in the body of *C*. Then, $\mathit{dif}(H, A)$, is defined as follows:

$$\mathit{dif}(H, A) = \begin{cases} k - m, & \text{if } \mathit{time}(H) = \texttt{first next}^k \text{ and } \mathit{time}(A) = \texttt{first next}^m, \\ k - m, & \text{if } \mathit{time}(H) = \texttt{next}^k \text{ and } \mathit{time}(A) = \texttt{next}^m, \\ k - m, & \text{if } \mathit{time}(H) = \texttt{next}^k \text{ and } \mathit{time}(A) = \texttt{first next}^m, \\ -\infty, & \text{if } \mathit{time}(H) = \texttt{first next}^k \text{ and } \mathit{time}(A) = \texttt{next}^m. \end{cases}$$

The intuition behind the above definition is the following: $\mathit{dif}(H, A)$ is a lower bound for the temporal difference between the canonical atoms corresponding to *H* and *A* in any canonical instance of *C*. 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. The following definition formalizes the notion of the *extended cycle-sum graph* of the skeleton of a given program.

Definition 30. Let *P* be a Linear-time Datalog$^\neg$ program and *S* be its skeleton. The *extended cycle-sum graph* of *S* is a directed labeled multi-graph with self-loops $CG_S = (V, E)$. The set *V* of vertices of CG_S is the set of predicate symbols appearing in *S*. The set *E* of edges consists of triples (p, q, l), where $p, q \in V$ and $l \in (Z \cup \{-\infty\}) \times \{`+\text{'}, `-\text{'}\}$. An edge $(p, q, \langle w, s\rangle)$ belongs to *E* if there exists a clause in *S* with an atom *H* as its head and an atom *A* occurring in its body such that the predicate symbol of *H* is *p* and the predicate symbol of *A* is *q*; $w = \mathit{dif}(H, A)$; $s =$ '$-$' if *A* occurs negatively in the clause body and $s =$ '$+$' otherwise.

Definition 31. Let *P* be a Linear-time Datalog$^\neg$ program and *S* be its skeleton. Then, *P* passes the extended cycle-sum test if in any strongly connected component of CG_S that contains a negatively signed edge the following conditions both hold:
(1) The sum of weights across every cycle is non-negative.
(2) Every cycle which has a zero sum of weights does not contain a negatively signed edge.

Example 32. Consider the following skeleton *S* of a program *P* and its associated extended cycle-sum graph depicted in Fig. 1:

```
first p.
p ← q.
q ← p.
next p ← ¬ r.
next r ← q.
```

The cycle-sum graph CG_S consists of a single strongly connected component which contains a negatively signed edge. The sum of weights across every cycle of the graph is

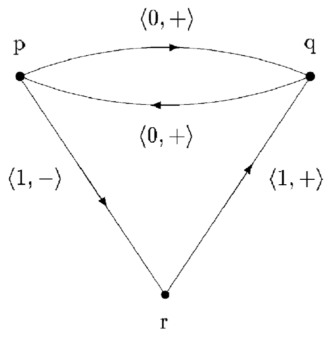

Fig. 1. The extended cycle-sum graph of the skeleton S in Example 32.

non-negative. The only cycle that has zero sum of weights does not contain any negatively signed edge. Therefore, program P passes the extended cycle-sum test.

One can easily verify that S is locally stratified, by taking stratum s_i to be equal to: $s_i = \{\texttt{first next}^i\ \texttt{p}, \texttt{first next}^i\ \texttt{q}, \texttt{first next}^i\ \texttt{r}\}$.

It is worth noting here that the extended cycle-sum test can be implemented efficiently using standard graph algorithms. More specifically, the existence of a cycle with non-positive sum of weights in a strongly connected component can be detected using an algorithm for shortest paths, that operates on graphs with negative weights, such as the Bellman–Ford algorithm, see [8], or the Gabow–Tarjan algorithm [9] which is the best known algorithm for the above problem. A technique working along these lines has been used in [16] to implement the cycle-sum test proposed in [29].

5.2. *Properties of the extended cycle-sum test*

In the following, we demonstrate that a Linear-time Datalog$^{\neg}$ program that passes the extended cycle-sum test is temporally stratified. Before stating the main theorem of this section, we need the following two simple lemmata:

Lemma 33. *Let P be a Linear-time Datalog$^{\neg}$ program and S be its skeleton. Assume that in ADG_S there exists an edge from vertex $\texttt{first next}^k\ p$ to vertex $\texttt{first next}^m\ q$. Then, there exists an edge in CG_S from vertex p to vertex q with weight at most $k - m$.*

Proof. Straightforward using the definition of *dif* and the construction of CG_S. □

Lemma 34. *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 $C_1, \ldots, C_k$.*

The proof of the above lemma is easy, and it was initially given in [29].

This leads us to the main theorem of this subsection:

Theorem 35. *If a Linear-time Datalog¬ program P passes the extended cycle-sum test then it is temporally stratified.*

Proof. Assume that P passes the extended cycle-sum test but it is not temporally stratified. Then, by Definition 23 the skeleton S of P is not locally stratified. By Theorem 12, this means that there exists an infinite increasing sequence $A_1 < A_2 < \cdots$ of canonical atoms of the temporal Herbrand base of S. Since the program contains a finite set of predicate names, there exists an infinite subsequence of the form $\texttt{first}\ \texttt{next}^{k_1}\ p < \texttt{first}\ \texttt{next}^{k_2}\ p < \cdots$, i.e., a subsequence in which all atoms have the same predicate name. There must be some i such that $k_{i+1} \geqslant k_i$ (because otherwise the subsequence would end). It is easy to check (using Lemma 33) that this implies the existence of a closed walk in CG_S which contains a negatively signed edge and has non-positive sum of weights (less than or equal to $k_i - k_{i+1}$). Moreover this closed walk is entirely contained in a single strongly connected component of the cycle-sum graph (because the subgraph that corresponds to the walk is itself strongly connected). This closed walk can be decomposed into a sequence of simple cycles that have the same sum of weights as the walk (Lemma 34). There are two cases:

- either there exists a cycle with negative sum of weights, or
- all cycles of the walk have zero sum of weights (and at least one of these cycles contains a negatively signed edge).

In both cases, the cycle-sum test will fail for P. □

We now examine an interesting class of programs for which the converse of the above theorem also holds:

Definition 36. The set of *open Linear-time Datalog¬ programs* consists of those Linear-time Datalog¬ programs in which all non-unit clauses contain only open temporal atoms.

The following graph-theoretic lemma will be necessary in the discussion that will follow:

Lemma 37. *Let G be a weighted directed multi-graph with self-loops. Let SC be a strongly connected component of G that contains a cycle with negative sum of weights across its edges. Then for every edge e of SC there exists a closed walk in SC that contains e and has negative sum of weights.*

Proof. Let C be a cycle in SC with negative sum of weights and let e be an edge in SC from vertex u to vertex v. Then for an arbitrary vertex x in C, there exists a path w_1 from x to u and a path w_2 from v to x, since these vertices belong to the same strongly connected component. Then starting from x, we can construct a closed walk $w_1 e w_2$ that contains e. If the sum of weights across this walk is positive, we can extend the walk with an appropriate number of repetitions of the cycle C, until the sum of the weights becomes negative. □

The following two lemmata will be used in the proof of Theorem 40.

Lemma 38. *Let P be an open Linear-time Datalog$^{\neg}$ program and let S be its skeleton. Let e be an edge in CG_S from p to q with weight w. Then, there exists a non-negative number denoted by k_e, such that for every $k \geqslant k_e$ there exists an edge in ADG_S from* `first next`^{k} *p to* `first next`$^{k-w}$ *q. Moreover, if e is negatively signed then the corresponding edges in ADG_S are also negatively signed.*

Proof. The existence of e in CG_S implies that there exists a clause in S with head `next`^{r} p and whose body contains `next`$^{r-w}$ q. Take $k_e = r$. The lemma follows directly according to the definition of ADG_S. □

Lemma 39. *Let P be an open Linear-time Datalog$^{\neg}$ program and let S be its skeleton. Let $W = p_0 e_1 p_1 \ldots e_m p_m$ be a walk in CG_S. Then, there exists an n_0 such that for all $n \geqslant n_0$, there exists a walk in ADG_S from* `first next`^{n} p_0 *to* `first next`$^{n-d}$ p_m*, where d is the sum of the weights across W. Moreover, if W contains a negatively signed edge, then each corresponding walk in CG_S also contains a negatively signed edge.*

Proof. For each edge e_i, $0 \leqslant i \leqslant m$, of the walk, let k_{e_i} be the number determined for edge e_i by Lemma 38. Let $k = max\{k_{e_i} \mid 0 \leqslant i \leqslant m\}$ and let s be the sum of all positive weights across W. Take $n_0 = k + s$. Let w_i be the weight of edge e_i and let $d_i = \sum_{j=1}^{i} w_i$. For any value $n \geqslant n_0$, consider the sequence of vertices $v_0, v_1, \ldots, v_m$ in ADG_S, where $v_i =$ `first next`$^{n-d_i}$ p_i, $0 \leqslant i \leqslant m$ (notice that $n - d_i \geqslant 0$ and therefore the above canonical temporal atoms are meaningful). Since $n = k + s \geqslant k_{e_i} + d_i$, we get that $n - d_i \geqslant k_{e_i}$. Thus, we can apply Lemma 38 for every edge e_i, $1 \leqslant i \leqslant m$, to prove that there exists an edge in ADG_S from v_{i-1} to v_i. This means that the sequence of vertices $v_0, v_1, \ldots, v_m$ form a walk in ADG_S, from $v_0 =$ `first next`^{n} p_0 to $v_m =$ `first next`$^{n-d}$ p_m (notice that $d_m = d$). To complete the proof observe that the number of negatively signed edges in both walks are equal, due to Lemma 38. □

Theorem 40. *Let P be an open Linear-time Datalog$^{\neg}$ program. Then, P passes the extended cycle-sum test if and only if P is temporally stratified.*

Proof. The one direction is Theorem 35. For the other direction assume that P is temporally stratified but it fails to pass the extended cycle-sum test. This means that there exists a strongly connected component G of CG_S (where S is the skeleton of P) containing a negatively signed edge such that either:

(1) there exists a cycle in G whose sum of weights is negative, or
(2) there exists a cycle in G, containing a negatively signed edge, that has a zero sum of weights.

Then, in both cases we can construct a closed walk $W = p_0 e_0 p_1 \ldots e_m p_m$ (where $p_m = p_0$) in CG_S, that contains a negatively signed edge and has a non-positive sum of weights equal to d. In particular, in the first case, if the negatively signed edge is not contained in a negative cycle we can apply Lemma 37 to get the desired closed walk. Then, by Lemma 39, for some sufficiently large k, there exists a walk in ADG_S from `first next`^{k} p_0 to `first next`$^{k-d}$ p_0, that contains a negatively signed edge. Thus, `first next`^{k} $p_0 <$ `first next`$^{k-d}$ p_0. Applying Lemma 39 repeatedly, we obtain an infinitely increasing sequence of canonical temporal atoms (of the form $\texttt{first}\,\texttt{next}^{k}\ p_0 < \texttt{first}\,\texttt{next}^{k-d}\ p_0 < \cdots < \texttt{first}\,\texttt{next}^{k-i\cdot d}\ p_0 < \cdots$). Therefore, S is not locally stratified and consequently P is not temporally stratified (contradiction). □

6. A temporal stratification test for branching-time deductive databases

In this section we develop a test for detecting whether a given Branching-time Datalog$^\neg$ program is temporally stratified. The test differs in a number of ways from the extended cycle-sum test of the previous section. One basic difference is that the test that will be described in the following covers the whole class of temporally stratified programs (but at the cost of a higher complexity). Moreover, the philosophy behind the two tests is different (but this issue will be further discussed in Section 7).

We can now explain at an informal level the basic idea behind the test that will follow. Given a Branching-time Datalog$^\neg$ program, we first obtain its skeleton. Then, we apply a series of transformations on the skeleton in such a way that at each step the information needed to verify local stratification is preserved. It is important to note that the programs obtained by applying the transformation steps are not necessarily semantically equivalent to the skeleton (but they preserve all information needed to decide local stratification). In general, the programs that result may contain much more clauses than the skeleton, but each clause is very simple in structure. We then demonstrate that the initial question of whether a given Branching-time Datalog$^\neg$ program is temporally stratified can be answered by examining the (much simpler) question of whether two programs that result from the transformation procedure, are stratified.

The three transformations that are applied on the skeleton of the initial program are *program normalization*, *walk normalization* and *subprogram extraction*, and are described in the next three subsections.

6.1. Program normalization

Let P be a Branching-time Datalog$^\neg$ program and let S be its skeleton. Then, program normalization consists of *clause normalization* and *temporal reference normalization* of S. Intuitively, clause normalization transforms each non-unit clause into a set of clauses that have exactly one atom in their body. Temporal reference normalization transforms each clause obtained after clause normalization into a set of clauses each one of which has a restricted number of temporal operators.

Step 1: Clause normalization.

The purpose of this step is to eliminate from S those clauses that contain more than one atoms in their bodies. We construct a new program S' from S by replacing every clause:

$$H \leftarrow A_1, \ldots A_m, \neg B_1, \ldots, \neg B_n.$$

in S, with $m + n > 1$, by the following $m + n$ clauses:

$$H \leftarrow A_1.$$
$$\ldots$$

$H \leftarrow A_m.$
$H \leftarrow \neg B_1.$
$\cdots$
$H \leftarrow \neg B_n.$

Lemma 41. *S is locally stratified iff S′ is locally stratified.*

Proof. The atom dependency graphs of the two programs are identical. □

Example 42. Consider the following skeleton S of a given Branching-time Datalog$^\neg$ program:
(**I1**) $\texttt{first}\,\texttt{next}_2\,\texttt{p} \leftarrow \neg\,\texttt{q}.$
(**I2**) $\texttt{next}_2\,\texttt{p} \leftarrow \texttt{q}.$
(**I3**) $\texttt{q} \leftarrow \texttt{next}_1\,\texttt{next}_2\,\texttt{p}, \neg\,\texttt{next}_1\,\texttt{p}.$
(**I4**) $\texttt{next}_3\,\texttt{r} \leftarrow \neg\,\texttt{r}.$
(**I5**) $\texttt{q} \leftarrow \neg\,\texttt{r}.$
Then, by applying the clause normalization step we get the program S':
(**J1**) $\texttt{first}\,\texttt{next}_2\,\texttt{p} \leftarrow \neg\,\texttt{q}.$
(**J2**) $\texttt{next}_2\,\texttt{p} \leftarrow \texttt{q}.$
(**J3**) $\texttt{q} \leftarrow \texttt{next}_1\,\texttt{next}_2\,\texttt{p}.$
(**J4**) $\texttt{q} \leftarrow \neg\,\texttt{next}_1\,\texttt{p}.$
(**J5**) $\texttt{next}_3\,\texttt{r} \leftarrow \neg\,\texttt{r}.$
(**J6**) $\texttt{q} \leftarrow \neg\,\texttt{r}.$
Notice that the clauses **J3** and **J4** have resulted from the transformation of clause **I3** in *S*.

Step 2: Temporal reference normalization.

The purpose of this step is to decrease the number of temporal operators that appear in a program clause. We construct a new program S'' from S' as follows. Every clause of the form:

$$[\texttt{first}]\,\texttt{next}_{i_1}\cdots\texttt{next}_{i_n} p \leftarrow [\neg]\,[\texttt{first}]\,\texttt{next}_{j_1}\cdots\texttt{next}_{j_m} q.$$

in S', with $n+m>0$, is replaced by the following $n+m+1$ clauses:

$$\begin{array}{lr} \texttt{next}_{i_k}\ p_k \leftarrow p_{k-1}. & \text{for } 1\leqslant k\leqslant n \\ [\texttt{first}]\ p_0 \leftarrow [\neg]\,[\texttt{first}]\,q_0. & (\textit{base clause}) \\ q_{r-1} \leftarrow \texttt{next}_{j_r}\ q_r. & \text{for } 1\leqslant r\leqslant m \end{array}$$

where $p_n=p$, $q_m=q$ and p_k, q_r, for $1\leqslant k\leqslant n-1$ and $1\leqslant r\leqslant m-1$, are new predicate symbols, used only for this clause.

If the operator `first` or the negation symbol appears in the original clause, then it is placed in the same position of the base clause: if the atom in the head (body) of the clause in S' is $\texttt{first}\ \texttt{next}_{i_1}\cdots\texttt{next}_{i_n}\ p$ ($\texttt{first}\ \texttt{next}_{j_1}\cdots\texttt{next}_{j_m}\ q$), then the head (body) of the base clause is $\texttt{first}\ p_0$ ($\texttt{first}\ q_0$). Moreover if the body of the original clause contains negation then the body of the base clause also contains negation. Notice that we could omit replacement in the case that the operator `first` does not appear in the clause and $m+n=1$.

Lemma 43. *S' is locally stratified iff S'' is locally stratified.*

Proof. Edges in the atom dependency graph of S' correspond to chains of edges in the atom dependency graph of S''. A bad walk in one graph, can be transformed into a bad walk in the other. $\square$

Example 44 (*Continued from Example 42*). Consider the program S' that has resulted in Example 42. Then, by applying the temporal reference normalization step we get the following program S'':

(1) $\texttt{next}_2\,\texttt{p} \leftarrow \texttt{t}.$
(2) $\texttt{first}\,\texttt{t} \leftarrow \neg\,\texttt{q}.$
(3) $\texttt{next}_2\,\texttt{p} \leftarrow \texttt{q}.$
(4) $\texttt{q} \leftarrow \texttt{u}.$
(5) $\texttt{u} \leftarrow \texttt{next}_1\,\texttt{s}.$
(6) $\texttt{s} \leftarrow \texttt{next}_2\,\texttt{p}.$
(7) $\texttt{q} \leftarrow \neg\,\texttt{next}_1\,\texttt{p}.$
(8) $\texttt{next}_3\,\texttt{r} \leftarrow \neg\,\texttt{r}.$
(9) $\texttt{q} \leftarrow \neg\,\texttt{r}.$

In the above program, clauses 1 and 2 have been obtained by transforming clause **J1** of S' while clauses 4, 5 and 6 are obtained from clause **J3**.

Notice now that there are six different types of clauses in S'', depending on the form of the temporal references:

- future clauses: $p \leftarrow [\neg]\,\texttt{next}_i\,q$.
- past clauses: $\texttt{next}_i\,p \leftarrow [\neg]q$.
- present clauses: $p \leftarrow [\neg]q$.
- canonical clauses: $\texttt{first}\,p \leftarrow [\neg]\,\texttt{first}\,q$.
- clauses with canonical head and open body: $\texttt{first}\,p \leftarrow [\neg]\,q$.
- clauses with open head and canonical body: $p \leftarrow [\neg]\,\texttt{first}\,q$.

The above different types of clauses will be used in the rest of the paper in order to formalize the proposed test.

6.2. Walk normalization

During walk normalization certain clauses are added to S'' in order to obtain a new program S^*. The goal of this transformation is that the endpoints of certain walks that exist in $ADG_{S''}$ will be directly connected by an edge in ADG_{S^*}. As a result, if there exists a bad walk in $ADG_{S''}$, then there exists a bad walk with a special form (more easily detectable) in ADG_{S^*}.

In the following, we define three different types of walks (namely *a-walk*, *b-walk* and *c-walk*) whose bypassing makes the detection of local stratification much easier. We start with the definition of a-walk bypassing and continue with b and c-walk bypassing.

Step 3: a-walk bypassing.

Consider the atom dependency graph $ADG_{S''}$ of the normalized program S''. Recall that $time(A)$ denotes the temporal reference of the atom A. Let $|time(A)|$ be the length (the number of temporal operators) of $time(A)$. Then:

Definition 45. A walk (of length $\geqslant 2$) in $ADG_{S''}$ from A to B is an *a-walk* if all the following conditions hold:

- $time(A) = time(B)$.
- For every intermediate node C in the walk, $|time(C)| > |time(A)|$.
- Every edge corresponds to a future, past or present clause.

Now, S''' is the least set of clauses that satisfies the following conditions:

- S''' contains all clauses in S''.
- S''' is transitive with respect to its present clauses, that is if
  $p \leftarrow [\neg]r.$
  $r \leftarrow [\neg]q.$
  belong to S''', then the clause
  $$p \leftarrow [\neg]q.$$
  also belongs to S'''.
- If a triple of clauses
  $p \leftarrow [\neg]\, \texttt{next}_i\ r.$
  $r \leftarrow [\neg]s.$
  $\texttt{next}_i\ s \leftarrow [\neg]q.$
  or a pair of clauses
  $p \leftarrow [\neg]\, \texttt{next}_i\ r.$
  $\texttt{next}_i\ r \leftarrow [\neg]q.$
  belongs to S''', then the clause
  $$p \leftarrow [\neg]q.$$
  also belongs to S'''.

In all cases, the atom in the body of the new clause is negated iff at least one of the original clauses contains negation.

Certain remarks concerning the consequences of the above transformation are in order. Since all a-walks in $ADG_{S'''}$ have been bypassed (as this will be demonstrated by the two lemmata that follow), then for any bad walk in $ADG_{S'''}$ whose edges correspond to future, past or present clauses of S''', there also exists a bad walk which is formed from edges that correspond to only future and present clauses. This is because the effect of past edges is to take us back to a point of time that we have already encountered, and therefore by doing a-walk bypassing we cancel entirely the need to consider past edges. Of course, present and future edges are still essential because their interplay can lead to bad walks. The temporal stratification test that will be proposed later in the paper will be based on the above remark (that past edges are inessential and therefore can be removed). The following two lemmata describe the consequences of a-walk bypassing:

Lemma 46. *S''' is locally stratified iff S'' is locally stratified.*

Proof. If S''' is locally stratified then obviously S'' is locally stratified, since the atom dependency graph of S''' is obtained by that of S'' by adding edges.

Conversely assume that S''' is not locally stratified, i.e. its atom dependency graph $ADG_{S'''}$ contains a walk w with infinitely many negative edges. But every edge in w either is contained in $ADG_{S''}$ or corresponds to a finite walk in $ADG_{S''}$. We can obtain an infinite walk in $ADG_{S''}$ by replacing each edge in w not in $ADG_{S''}$ with the corresponding walk. Notice that the number of negative edges is not decreased by this process. Thus S'' is not locally stratified. □

Lemma 47. *S''' is not locally stratified iff its atom dependency graph contains a bad walk without a-subwalks*.

Proof. The 'if' direction is straightforward. For the 'only if' direction, assume that S''' is not locally stratified, i.e., it contains a bad walk. We can show by induction on the number of past edges contained in an a-walk that its end points are also connected directly by an edge in the atom dependency graph of S''', which is negative iff the a-walk contains a negative edge. Following the bad walk, we can replace every maximal a-walk by the corresponding direct edge. The resulting walk also contains an infinite number of negative edges and does not contain a-subwalks. □

Example 48 (*Continued from Example 44*). By applying a-walk bypassing to the program S'' of Example 44, we get the following new clauses:

(10)	`s` $\leftarrow$ `t`.	(from clauses 6 and 1)
(11)	`s` $\leftarrow$ `q`.	(from clauses 6 and 3)
(12)	`s` $\leftarrow$ `u`.	(from clauses 11 and 4)
(13)	`s` $\leftarrow \neg$`r`.	(from clauses 11 and 9)

Clauses 1–13 constitute the program S'''.

Step 4: b- and c-walk bypassing.

The bypassing of a-walks is sufficient for programs that do not contain canonical atoms in the non-unit clauses. However, if the source program contains such canonical atoms then the test that we develop requires the bypassing of b-walks and c-walks, which are defined as follows:

Definition 49. A walk in $ADG_{S'''}$ from A to B is a *b-walk* if the following conditions are all satisfied:

- The first edge in the walk corresponds to a clause with canonical head and open body.
- The last edge in the walk corresponds to a past clause.
- Every intermediate edge corresponds either to a past clause or to a present clause.

Definition 50. A walk in $ADG_{S'''}$ from A to B is a *c-walk* if the following conditions are all satisfied:

- The first edge in the walk corresponds to a future clause.
- The last edge in the walk corresponds to a clause with open head and canonical body.
- Every intermediate edge corresponds either to a future clause or to a present clause.

Notice that the definitions of b-walk and c-walk are symmetric.

Now, S^* is the least set of clauses that satisfies the following conditions:

- S^* contains all clauses in S'''.
- If a triple of clauses

  $\texttt{first}\, p \leftarrow [\neg] r.$
  $r \leftarrow [\neg] s.$
  $\texttt{next}_i\ s \leftarrow [\neg] q.$

  or a pair of clauses

  $\texttt{first}\, p \leftarrow [\neg] r.$
  $\texttt{next}_i\ r \leftarrow [\neg] q.$

  belongs to S^*, then the clause
  $\texttt{first}\, p \leftarrow [\neg] q.$
  also belongs to S^*.
- If a triple of clauses

  $q \leftarrow [\neg]\texttt{next}_i\ r$
  $r \leftarrow [\neg] s.$
  $s \leftarrow [\neg]\ \texttt{first}\ p$

  or a pair of clauses
  $q \leftarrow [\neg]\texttt{next}_i\ r$
  $r \leftarrow [\neg]\ \texttt{first}\ p$
  belongs to S^*, then the clause
  $q \leftarrow [\neg]\ \texttt{first}\ p$
  also belongs to S^*.

In all cases, the atom in the body of the new clause is negated iff at least one of the original clauses contains negation.

Notice that although a b-walk or c-walk may contain consecutive present edges, considering at most triples of clauses is sufficient, since S''' is transitive with respect to its present clauses.

Intuitively, b-walk bypassing eliminates the necessity of using past edges after edges that correspond to clauses with canonical head and open body. Symmetrically, c-walk bypassing eliminates the necessity of using future edges before edges that correspond to clauses with open head and canonical body.

The following lemma is straightforward:

Lemma 51. *S''' is locally stratified iff S^* is locally stratified.*

Proof. The proof is similar to that of Lemma 46. Again additional edges correspond to walks in the original atom dependency graph. □

Example 52 (*Continued from Example 48*). By applying b-walk and c-walk bypassing to the program S''' of Example 48, we get an extra clause:
(14) $\texttt{first t} \leftarrow \neg\, \texttt{r}.$ (from clauses 2, 9 and 8)
Clauses 1–14 constitute the program S^*.

We can now define the notion of *normal walk*:

Definition 53. An infinite walk in ADG_{S^*} starting at A is a *normal walk* if the following conditions both hold:

- It does not contain any subwalk that is an a-walk, b-walk or c-walk.
- For every vertex C in the walk, $|time(C)| \geqslant |time(A)|$.

Lemma 54. *S^* is not locally stratified iff its atom dependency graph contains a normal bad walk.*

Proof. The 'if' direction is straightforward. For the 'only if' direction, we first observe that the endpoints of a b-walk or c-walk in the atom dependency graph of S^* are directly connected by an edge. Moreover, this direct edge is negative iff the corresponding b-walk or c-walk contains a negative edge. This can be easily proved by induction on the number of past (present) edges that are contained in the b-walk (c-walk).

Suppose that S^* is not locally stratified. Then, by Lemma 51, the same holds for S'''. From Lemma 47 there exists a bad walk without a-subwalks in the atom dependency graph of S'''. Since S^* is an extension of S''' the same bad walk also exists in the atom dependency graph of S^*.

Following this bad walk, we can replace every maximal b-walk or c-walk by the corresponding direct edge. Notice that this process does not introduce any new a-walks. Thus the resulting walk w does not contain a-walks, b-walks or c-walks. Moreover w contains infinitely many negative edges.

To complete the proof we will show that w contains a final part which is normal. We claim that there exists a unique temporal reference t with minimum length among all the canonical ground atoms in w. To prove this claim consider two different temporal references t_1 and t_2 of the same length. Now, any walk that connects two atoms with temporal references t_1 and t_2 must pass through an atom whose temporal reference is a common prefix of t_1 and t_2; this is because in order to get from t_1 to t_2 one must first remove from t_1 the longest possible suffix that makes it differ from t_2. Consequently t_1 and t_2 cannot have minimum length and the claim is true.

The final subwalk of w starting at the first vertex that has temporal reference t is normal and contains infinitely many negative edges. □

6.3. Subprogram extraction

In the last step of the transformation, two subprograms S_1^* and S_2^* of S^* are extracted. As it will be demonstrated, one can decide if the skeleton S is locally stratified, by deciding if both S_1^* and S_2^* are stratified.

Step 5: Subprogram extraction.

The Subprogram extraction step consists of the production of the following two subprograms of S^*:

- S_1^* is the program that contains only the present and future clauses of S^*.
- S_2^* is the program that results by deleting all future and past clauses of S^*.

The following lemma demonstrates the importance of the above two subprograms:

Lemma 55. *S^* is locally stratified iff both S_1^* and S_2^* are locally stratified.*

Proof. The 'only if' direction is straightforward. For the other direction, we will show that if S^* is not locally stratified then the atom dependency graph of at least one of S_1^* and S_2^* contains a bad walk.

Assume that S^* is not locally stratified. According to Lemma 54 the atom dependency graph of S^* contains a normal bad walk w . We consider two cases:

Case 1: Walk w passes through finitely many atoms with temporal reference equal to `first`. In that case there exists a final subwalk w' of w , which is normal and bad, that never passes through an atom with temporal reference `first`. This implies that w' does not contain any edge that corresponds to a clause in S^* that contains canonical atoms. Moreover, we claim that w' does not contain any edge corresponding to a past clause. To prove this fact, consider for the sake of contradiction the first edge e in w' that corresponds to a past clause. If all the edges before e in w' correspond to present clauses, then w' cannot be normal, since the second property of normality is violated. On the other hand if an edge that corresponds to a future clause appears before e in w' then an a-walk is formed, which also contradicts the normality of w'. Thus, w' contains edges that correspond only to present or future clauses, which implies that it is also contained in the atom dependency graph of S_1^*.

Case 2: Walk w passes through infinitely many atoms with temporal reference equal to `first`. We claim that w does not contain any edge that corresponds to a future or a past clause. To prove the claim notice that if w passes through an edge that corresponds to a future clause, then it must later pass through an edge that corresponds to either a past clause or a clause with open head and canonical body (because it will pass through an atom with temporal reference `first`). This is impossible since in the former case w would contain an a-subwalk and in the latter case a c-subwalk. Consequently w does not contain any future edge. Similarly, w cannot contain an edge corresponding to a past clause since in that case it would contain an a-subwalk or a b-subwalk. Thus the claim is true. Consequently w also exists in the atom dependency graph of S_2^*.

In any case at least one of S_1^* and S_2^* is not locally stratified. □

Example 56 (*Continued from Example 52*). Program S_1^* consists of the clauses 4–7 and 9–13 and S_2^* consists of the clauses 2,4 and 9–14.

The next two lemmata demonstrate that for the programs S_1^* and S_2^* local stratification coincides with stratification:

Lemma 57. *S_1^* is locally stratified iff it is stratified.*

Proof. If S_1^* is stratified, then it is also locally stratified from Theorem 7. For the other direction assume that S_1^* is not stratified and let $p_0, p_1, \ldots, p_{n-1}, p_0$ be a cycle of length n in its predicate dependency graph that contains a negative edge. We denote by T_i the temporal reference in the body of the clause corresponding to the edge $(p_i, p_{(i+1) \bmod n})$. Notice that T_i is null for edges corresponding to present clauses.

Consider the infinite sequence of temporal atoms $A_0, A_1, \ldots, A_i, \ldots$ such that the predicate symbol of A_i is $p_{i \bmod n}$ and the temporal reference R_i of A_i is defined recursively as follows: $R_0 = \texttt{first}$ and $R_{i+1} = R_i$ if $T_{i \bmod n} =$ null, otherwise $R_{i+1} = R_i T_{i \bmod n}$.

It is easy to check that A_i and A_{i+1} are adjacent in the atom dependency graph of S_1^*. Moreover there exists at least one negative edge in the subwalk with endpoints $A_{k \cdot n}$ and $A_{(k+1) \cdot n}$, for every $k \geqslant 0$. Consequently S_1^* is not locally stratified, since the A_i's form a bad walk in its atom dependency graph. □

Lemma 58. *S_2^* is locally stratified iff it is stratified.*

Proof. If S_2^* is stratified, then it is also locally stratified from Theorem 7. For the other direction assume that S_2^* is not stratified and let $p_0, p_1, \ldots, p_{n-1}, p_0$ be a cycle of length n in its predicate dependency graph that contains a negative edge. Then $\texttt{first}\, p_0$, $\texttt{first}\, p_1, \ldots, \texttt{first}\, p_{n-1}$, $\texttt{first}\, p_0$ is a cycle in the atom dependency graph of S_2^* that contains a negative edge. By repeating this cycle infinitely many times we construct a bad walk. Consequently S_2^* is not locally stratified. □

6.4. The temporal stratification test

Based on the results of the previous subsections we can now define the temporal stratification test for Branching-time Datalog$^\neg$ programs:

Definition 59. Let P be a Branching-time Datalog$^\neg$ program and S_1^*, S_2^* the programs obtained by applying Steps 1–5 to the skeleton S of P. Then, P passes the *temporal stratification test* if S_1^* and S_2^* are stratified.

Theorem 60. *A Branching-time Datalog$^\neg$ program P passes the temporal stratification test if and only if it is temporally stratified.*

Proof. It is an immediate consequence of Lemmata 41, 43, 46, 51, 55, 57, 58. □

Example 61 (*Continued from Example 56*). The predicate dependency graphs of programs S_1^* and S_2^* are shown in Fig. 2. Obviously neither graph contains a cycle with negative edge, thus S is locally stratified.

It is important to note that the temporal stratification test can be significantly simplified when the source program does not contain canonical temporal references in the non-unit clauses. In this case Step 4 can be omitted since it does not introduce any new clauses (because there do not exist any b-walks or c-walks). Moreover, in this case it can be easily seen that S_2^* is a subset of S_1^* and therefore the test need only examine S_1^* for stratification.

The temporal stratification test operates in polynomial time, however its complexity is higher than that of the extended cycle-sum test. The most expensive part of the test consists of steps 3 and 4. In particular step 3 requires to maintain the transitive closure of present clauses, while new present clauses are added to the program. This can be efficiently performed using the algorithm proposed by Italiano in [14], which is applied to an appropriate auxiliary graph.

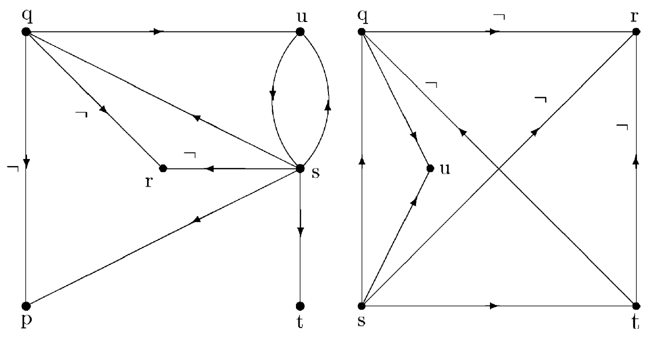

Fig. 2. The predicate dependency graphs of S_1^* and S_2^*.

Step 4 can be implemented in a similar way. However, we believe that a detailed description of the implementation of the test would be rather lengthy and is therefore beyond the scope of this paper.

We close this section with a theorem concerning the decidability of local stratification for Branching-time Datalog$^\neg$:

Theorem 62. *The Local Stratification problem for Branching-time Datalog*$^\neg$ *is decidable*.

Proof. Let P a Branching-time Datalog$^\neg$ program and let P' be the program consisting of all ground instances of the clauses in P. Then, P' is finite (since Branching-time Datalog$^\neg$ does not use function symbols). Moreover, the atom dependency graphs of P and P' are identical and therefore P is locally stratified if and only if P' is locally stratified. Now, since P' does not contain any variables one can replace every atom by a propositional symbol, getting a program P''. Obviously, P' is locally stratified if and only if P'' is locally stratified. But since P'' is propositional it coincides with its skeleton and therefore P'' is locally stratified if and only if it is temporally stratified. Consequently, we can decide if P is locally stratified by applying the temporal stratification test to P''. □

The above theorem is mainly of theoretical importance since the decision procedure presupposes the construction of the ground instantiation of the source program (whose size may be exponentially larger than the size of the initial program). However, the main idea of the theorem is interesting since it demonstrates that local stratification may be decidable for certain useful logic programming languages (even though their Herbrand universe may be infinite).

7. A comparison of the two tests

The two temporal stratification tests described in this paper have a different underlying philosophy and this fact gives to each one of them certain relative merit when compared to the other one.

First of all, the (extended) cycle-sum test is built on the notion of temporal difference between atoms (namely *dif*), which is a lower bound quantity. Therefore, there exist cases in which a program is temporally stratified but this cannot be detected by the extended cycle-sum test. The following example illustrates this state of affairs.

Example 63. Let *P* be the following program:

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

The skeleton *S* of *P* is

```
first p ← ¬ first next q.
first next next q ← ¬ first next p.
```

It is easy to see that although *P* is temporally stratified, it is rejected by the (extended) cycle-sum test: the cycle-sum graph of its skeleton contains a cycle with zero sum of weights and a negatively signed edge.

Consider now the application of the branching-time test. One can easily see that the program S^* that results is the following:

```
first p ← ¬ first r.
r ← next q.
next q ← s.
next s ← t.
first t ← ¬ first u.
u ← next p.
r ← s.
```

The program S_1^* is

```
r ← s.
r ← next q.
u ← next p.
```

The program S_2^* is

```
first p ← ¬ first r.
first t ← ¬ first u.
r ← s.
```

Both S_1^* and S_2^* are stratified and therefore *S* is locally stratified.

Therefore, although the extended cycle-sum test covers a significant subclass of Linear-time Datalog$^\neg$, it does not exhaust the whole class (while the branching-time test does). It is an open question for us whether there exists a simple test that is based on temporal differences and which covers the whole class of Linear-time Datalog$^\neg$ programs.

On the other hand however, the extended cycle-sum test does not alter in any way the skeleton of the input program. This is an important advantage because the branching-time test introduces during the program normalization steps a (possibly large) number of extra clauses based on the structure of the input program; additionally the test may introduce extra clauses due to the transitive closure procedure that it performs during the walk normalization steps.

8. Related work

To our knowledge, only a few other results exist regarding stratified negation in temporal logic programming. The pioneering work in this area appears to be the idea of *XY-stratification* proposed in [34] which applies to *XY-Datalog*, a language proposed for combining active and deductive databases. XY-Datalog clauses use a distinguished argument, called the *stage argument*, in the same way that Linear-time Datalog$^{\neg}$ possesses an implicit time argument. The idea of XY-stratification is applied to programs that have a restricted syntax when compared to that of Linear-time Datalog$^{\neg}$, and for this reason the extended cycle-sum test is more general than XY-stratification.

More recently, *state stratification* [19] was proposed, an approach which applies to the language *Statelog*. However, state stratification only applies to programs that are *progressive* (in Linear-time Datalog$^{\neg}$ terminology this means that the temporal reference of the head of a clause is greater than or equal to the temporal references of the atoms that appear in the clause body). This makes state stratification less generally applicable since it disallows clauses in which body atoms look “further into the future” than the head of the clause (and which are quite common in temporal logic programming). It should be noted, however, that the state stratification approach is based on the notion of *leap* which is similar to the notion of *dif* of the cycle-sum approach (the basic difference being that leaps are always non-negative).

Similar restrictions to the ones discussed above for Statelog also apply to the *temporal stratification* approach proposed for Starlog programs in [18]. More specifically, Starlog implicitly adds *causality* constraints to program clauses. As mentioned in [18], “causality means that no truth in the past is defined in terms of truth in the future” or equivalently “the timestamp of the head is no less than the timestamp of any literal in its body”. Clearly, the notion of causality is equivalent to the notion of progressiveness in Statelog.

In [15] the classes of ELS and EMS programs are proposed. Again, in these programs there exists a distinguished argument in predicates (the *strata-level argument*) on which certain conditions must be satisfied. For example, in ELS programs the authors of [15] impose certain restrictions, one of which is that “if s(N) occurs in a body literal, then the head atom must have a strata-level argument of s(N)”. EMS programs are more general than ELS ones but again the definitions given in [15] imply that even for this class the strata-level argument of a literal in the body of a clause cannot be greater than the strata-level argument in the head of the clause.

Last but not least we should mention the work in [24] which investigates sufficient conditions for local stratification of classical logic programs by taking into account the complexity of terms. Since the approach in [24] applies to arbitrary logic programs, it is natural to wonder whether this technique is weaker than our approach when one restricts attention to temporal programs. As the following example demonstrates, there exist classical logic programs for which the technique of [24] results to a “*Do not know*” output and whose local stratifiability can be decided by our temporal stratification test (provided that they have been properly encoded as Branching-time Datalog$^\neg$ programs). More specifically, consider the following program (given as Example 4.33, p. 228 of [24]):

```
p(X) ← ¬ q(f(f(X))).
q(f(Y)) ← r(g(Y)).
r(g(g(Z))) ← p(Z).
```

The above program when run through the algorithm in [24] for testing local stratification, results to a "*Do not know*" output (which signifies that the algorithm is unable to decide whether the program is locally stratified or not).

We can however, translate the above program in Branching-time Datalog$^\neg$ as follows:

$\texttt{p} \leftarrow \neg\ \texttt{next}_1\ \texttt{next}_1\ \texttt{q}.$
$\texttt{next}_1\ \texttt{q} \leftarrow \texttt{next}_2\ \texttt{r}.$
$\texttt{next}_2\ \texttt{next}_2\ \texttt{r} \leftarrow \texttt{p}.$

It is straightforward to show that the above two programs are equivalent from a local stratification point of view (actually, their atom dependency graphs are isomorphic). Now, by applying the proposed algorithm one can easily see that the above (second) program is a locally stratified one. We do not give the full transformation since it results to 13 clauses. From these clauses one need only examine the subset that corresponds to present and future clauses. The predicate dependency graph of this set is acyclic which implies that the program is locally stratified.

9. Discussion

Temporal deductive databases are promising formalisms whose properties and applications appear to require further research. We believe that the techniques developed in this paper contribute along this direction. There are, however, many aspects of this work that require further investigation. We briefly mention some of them:

- It would be interesting to embed the proposed tests in a practical system for temporal deductive databases. This would require an efficient implementation of the tests (an implementation of the linear-time test has already been undertaken [32] based on the ideas developed in [16]). An embedding of the tests in a temporal deductive database would give a feeling of how useful negation is in such a framework.
- The extended cycle-sum test developed in this paper covers a broad class of temporally stratified programs. It would be interesting, however, to investigate whether there exists a similar (i.e. temporal difference based) test that exhausts the whole class of temporally stratified Linear-time Datalog$^\neg$ programs.
- Linear time temporal logic programming is only an instance of the much more general paradigm of *intensional logic programming* [23]. Is it possible to develop a test that would apply to many different intensional languages, which, however, share some common semantic properties? The work in [23] which creates a language-independent semantic framework for intensional languages, might be a good starting point here.

We believe that answers to the above questions would offer a better understanding for the interplay between temporal deductive databases and negation.

References

[1] K.R. Apt, H.A. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann Publishers, Los Altos, CA, 1988, pp. 89–148.

[2] K.R. Apt, B.N. Bol, Logic programming and negation: a survey, J. Logic Programming 19/20 (1994) 9–71.

[3] F. Arni, K. Ong, S. Tsur, H. Wang, C. Zaniolo, The deductive database system LDL++, Theory Practice Logic Programming 3 (1) (2003) 61–94.

[4] M. Baudinet, J. Chomicki, P. Wolper, Temporal deductive databases, in: L. Fariñas del Cerro, M. Penttonen (Eds.), Temporal Databases: Theory, Design and Implementation, The Benjamin/Cummings, Menlo Park, CA, 1993, pp. 294–320.

[5] P. Cholak, H.A. Blair, The complexity of local stratification, Fundam. Inform. 21 (4) (1994) 333–344.

[6] J. Chomicki, Depth-bounded bottom-up evaluation of logic programs, J. Logic Programming 25 (1) (1995) 1–31.

[7] J. Chomicki, T. Imielinski, Finite representation of infinite query answers, ACM Trans. Database Systems 18 (2) (1993) 181–223.

[8] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms, MIT Press, McGraw-Hill, Cambridge, MA, 1990.

[9] H.N. Gabow, R.E. Tarjan, Faster scaling algorithms for network problems, SIAM J. Comput. 18 (5) (1989) 1013–1036.

[10] M. Gergatsoulis, Temporal and modal logic programming languages, in: A. Kent, J.G. Williams (Eds.), Encyclopedia of Microcomputers, Vol. 27, Suppl. 6, Marcel Dekker, New York, 2001, pp. 393–408.

[11] M. Gergatsoulis, C. Nomikos, A proof procedure for temporal logic programming, Internat. J. Found. Comput. Sci. 15 (2) (2004) 417–443.

[12] F. Giannotti, G. Manco, M. Nanni, D. Pedreschi, Nondeterministic, nonmonotonic logic databases, IEEE Trans. Knowledge Data Eng. 13 (5) (2001) 813–823.

[13] F. Harary, Graph Theory, Addison-Wesley, Reading, MA, 1972.

[14] G.F. Italiano, Amortized efficiency of a path retrieval data structure, Theoretical Computer Science 48 (1986) 273–281.

[15] D.B. Kemp, K. Ramamohanarao, Efficient recursive aggregation and negation in deductive databases, IEEE Trans. Knowledge Data Engineering 10 (5) (1998) 727–745.

[16] C.D. Koutras, C. Nomikos, On the computational complexity of stratified negation in linear-time temporal logic programming, in: M. Gergatsoulis, P. Rondogiannis (Eds.), Intensional Programming II, World Scientific, Singapore, 2000, pp. 106–116.

[17] J.W. Lloyd, Foundations of Logic Programming, Springer, Berlin, 1987.

[18] L. Lu, J.G. Cleary, An operational semantics of Starlog, in: Proc. PPDP'99, 1999, pp. 294–310.

[19] B. Ludäscher, Integration of active and deductive database rules, Ph.D. Thesis, Institut für Informatik, Universität Freiburg, 1998.

[20] C. Nomikos, P. Rondogiannis, M. Gergatsoulis, A stratification test for temporal logic programs, in: Proc. 3rd Panhellenic Logic Symp. 2001, Anogia, Greece, July 17–21.

[21] M.A. Orgun, On temporal deductive databases, Comput. Intell. 12 (2) (1996) 235–259.

[22] M.A. Orgun, W. Ma, An overview of temporal and modal logic programming, in: D.M. Gabbay, H.J. Ohlbach (Eds.), Proc. 1st Internat. Conf. on Temporal Logics (ICTL'94), Lecture Notes in Artificial Intelligence (LNAI), Vol. 827, Springer, Berlin, 1994, pp. 445–479.

[23] M.A. Orgun, W.W. Wadge, Towards a unified theory of intensional logic programming, J. Logic Programming 13 (4) (1992) 413–440.

[24] L. Palopoli, Testing logic programs for local stratification, Theoret. Comput. Sci. 103 (1992) 205–234.

[25] P. Potikas, P. Rondogiannis, M. Gergatsoulis, A transformation technique for datalog programs based on non-deterministic constructs, in: A. Pettorossi (Ed.), Logic Based Program Synthesis and Transformation,

11th Internat. Workshop, LOPSTR 2001, Paphos, Cyprus, November 28–30, Lecture Notes in Computer Science (LNCS), Vol. 2372, Springer, Berlin, 2002, pp. 25–45.

[26] H. Przymusinska, T. Przymusinski, Semantic issues in deductive databases and logic programming, in: R. Banerji (Ed.), Formal Techniques in Artificial Intelligence, North-Holland, Amsterdam, 1990, pp. 321–367.

[27] T.C. Przymusinski, On the declarative semantics of deductive databases and logic programs, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Menlo Park, CA, 1988, pp. 193–216.

[28] R. Ramakrishnan, J.D. Ullman, A survey of deductive database systems, The J. Logic Programming 23 (2) (1995) 125–149.

[29] P. Rondogiannis, Stratified negation in temporal logic programming and the cycle-sum test, Theoret. Comput. Sci. 254 (2001) 663–676.

[30] P. Rondogiannis, M. Gergatsoulis, The branching-time transformation technique for chain datalog programs, Intelligent Inform. Systems 17 (1) (2001) 71–94.

[31] P. Rondogiannis, M. Gergatsoulis, T. Panayiotopoulos, Branching-time logic programming: the language Cactus and its applications, Comput. Languages 24 (3) (1998) 155–178.

[32] I. Symeonidou, The implementation of the cycle-sum test, Diploma Thesis (in preparation), Department of Informatics & Telecommunications, University of Athens, 2003.

[33] W.W. Wadge, Tense logic programming: a respectable alternative, Proc. 1988 Internat. Symp. on Lucid and Intensional Programming, 1988, pp. 26–32.

[34] C. Zaniolo, N. Arni, K. Ong, Negation and aggregates in recursive rules: the LDL $++$ approach, in: Proc. DOOD-93, Phoenix, AZ, USA, December 6–8 1993.

[35] C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, R. Zicari, Advanced Database Systems, Morgan Kaufmann, San Francisco, CA, 1997.