ON THE EXPRESSIVE POWER OF INFINITE TEMPORAL DATABASES

by

Gabriel M. Kuper
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

and

Alex Tuzhilin
New York University
Leonard N. Stern School of Business
Information Systems Department
40 West 4th Street, Tisch Hall Rm. 621
New York, NY 10003

April 1992

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-92-16

This paper also appears as IBM Technical Report RC17881.

On the Expressive Power of Infinite Temporal Databases

Gabriel M. Kuper* Alexander Tuzhilin†

Abstract

We discuss different techniques for representing infinite temporal data. There are two basic approaches: A procedural description, as used in production systems, and represented, in this paper, by a version of Datalog. The second approach is a more declarative method, using some form of temporal logic programming. We examine several versions of each approach, and compare their expressive power, i.e., what temporal data each formalism can capture.

1 Introduction

There has been a substantial amount of research done recently on studying finite temporal databases. A few representative examples of this work are [Ari86, CW83, CC87, Gad88, NA88, Sno87, Tan86][1]. Most research in this area has assumed that all the temporal data is stored explicitly in the database. However, there have been some studies that try to extend finite temporal databases to support infinite time horizons [CI88, KT89, KSW90, TK91, BNW91] and to support infinite sequences of database states [Via87, GT86]. Since it is impossible to store infinitely many tuples in a database, there is clearly a need for some finite "encoding" of these tuples so that they can be actually stored in the database.

There are several reasons why the study of infinite temporal databases is important. First, it is often difficult to set an a priori time limit on the time period over which a temporal database is defined [KSW90]. Secondly, the data in the sequence may not be

*IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY. 10598; e-mail: kuper@watson.ibm.com

†New York University, Stern School of Business, Information Systems Department, 40 West 4th Street, Rm. 621, New York, NY 10003; e-mail: atuzhilin@stern.nyu.edu

[1]This list is not exhaustive; for an overview of the area of time and databases see [Sno90] and [TCG+].

materialized yet, i.e. the data might be stored not in an explicit form but in the form of constraints [KKR90, KSW90], as can be the case with future data in temporal databases [TK89]. Various mechanisms for materializing the data, such as production systems [BFK86], temporal logic programs [AM89, Bau89, Gab89, BFG$^+$89, Tuz91], linear repeating points [KSW90], handle both finite and infinite data. A third reason for studying infinite temporal databases is that they lead to more compact and tractable representations than provided by existing methods for finite sequences.

An important issue in studying infinite temporal databases is how to describe them in *finite* terms. There have been several methods presented in the literature for describing infinite temporal databases. For example, [KT89] proposes the use of production systems and recurrence equations, [TK91] considers Predicate Transition Networks in addition to production systems and recurrence equations. Other proposals are linear repeating points [KSW90], temporal logic programming [BNW91], and logic programming with explicit references to time [CI88].

Among the various formalisms for defining infinite temporal databases, we are especially interested in temporal logic programming and in production systems. On one hand, temporal logic programming provides a declarative method to defining the semantics of infinite (and finite) temporal databases [Bau89]. On the other hand, production systems represent a practical knowledge representation method that has recently became widely used for the specification of active databases (for example,see [dMS88, MD89, WF90, SJGP90, GJ91]). They can also be used for defining infinite temporal databases assuming that each recognize-act cycle generates a new state of a temporal database. We thus have two approaches, a declarative and a procedural method for defining infinite temporal databases. The purpose of this paper is to study the relationship between them.

One of the most important measures for comparing the two approaches is their *expressive power*, i.e., what temporal databases can they represent. We ask whether or not there is an infinite temporal database generated by one of these methods that cannot be generated by the other method. Some problems of this nature were addressed in [KT89, TK91] where expressive powers of production systems, recurrence equations and Predicate Transition Networks were compared. Also, [BNW91] compared expressive powers of linear repeating points, Templog, and the formalism of Chomicki and Imielinski.

A key idea in our approach is that Datalog (with negation) can be viewed as an appropriate mathematical abstraction of some aspects of production systems [dMS88]. In order to do this, we look at a Datalog program as specifying a *sequence* of states, namely the sequence we obtain by applying all the rules in the program in parallel at each step. This contrasts with the more usual interpretation of Datalog, where we are only interested in the fixpoint of the rules.

Of course, production systems also allow deletion of facts, whereas Datalog does not. However, [AV89] has recently proposed a language called "doubly negated" Datalog, i.e., Datalog$^{\neg *}$ in which negations are allowed both in the body and in the head of a rule. In this language, a negation in the head corresponds to a deletion of a fact.

This is the general problem we are interested in: What is the relation between declarative temporal logic programming languages and doubly negated Datalog. In this paper, as an initial approach to this problem, we study the relative expressive power of negated Datalog and the temporal programming language Templog [AM89, Bau89].

The rest of the paper is organized as follows. In Section 2, we overview the language Templog and define a new meaning of Datalog and Datalog$^{\neg}$ programs. In Section 3, we define the concepts of bounded and unbounded simulations of one program by another and of relative expressive powers of different formalisms. We compare expressive powers of Datalog and Templog formalisms for unbounded simulations in Section 4 and for the bounded simulations in Section 5.

2 Preliminaries

2.1 Overview of Templog

The temporal logic programming language Templog is described in [AM89]. To make the paper self-contained, we review the key points in this section.

Templog is based on a clausal subset of first-order temporal logic with a discrete linear model of time extending infinitely into the future but not into the past. The temporal operators used in Templog are *next* $\circ$, *necessity (always)* $\Box$, and *possibility (eventually, sometimes)* $\Diamond$.

The syntax of Templog is defined as follows [Bau89], where A denotes an atom, and N a *next-atom* (i.e., an atom preceded by a finite number of $\circ$'s).

Body: $B ::= \epsilon | A | B_1 B_2 | \circ B | \diamond B$

where ϵ denotes the empty body

Initial clause: $IC ::= N \leftarrow B | \Box N \leftarrow B$

Permanent clause: $PC ::= \Box(N \leftarrow B)$

Program clause: $C ::= IC | PC$

Example 1 The following example is the modification of the "backup" example from [AM89]. Assume that we maintain various computer systems on a weekly basis. But before maintaining them, we do backups. Let predicate *maintenance(x)* specify that the system x should be maintained (at some time), and *backup(x)* specify that the system x should be backed up (also at some time). Then the Templog program

$$\Box(\circ^7 maintenance(x) \leftarrow maintenance(x))$$
$$\Box(backup(x) \leftarrow \circ maintenance(x))$$

says that the maintenance is performed on the weekly basis and that a system is backed up before it is maintained.

∎

The semantics of a Templog program P is defined in terms of its least temporal Herbrand model [Bau89]. The temporal Herbrand base of P is the set of all the ground next-atoms (i.e. ground atoms preceded by a finite number of the *next* operators $\circ$) constructed from the predicates of the program P and the ground terms of the Herbrand universe. A *temporal Herbrand model* of P is a subset of the temporal Herbrand base that makes all the formulas in P true at *all* moments of time.

An alternative way of defining the semantics of a Templog program is in terms of the fixpoint of the mapping T_P [Bau89]. To do this, let a strictly ground instance (SGI) of a clause C be a clause obtained from C by replacing its $\diamond$ and $\Box$ operators by arbitrary next-atoms and by replacing variables in C by arbitrary constants (thus making the clause ground). Then

$$T_P(I) = \{N \mid N \leftarrow N_1, \ldots, N_m \text{ is an SGI of a clause in P and } \{N_1, \ldots, N_m\} \in I\}$$

Baudinet [Bau89] shows that the fixpoint semantics coincides with the least temporal Herbrand model semantics.

In order to bring Templog to the database domain, we make a few additional assumptions. First of all, we assume that Templog has no function symbols. Secondly, we separate facts from rules. This is similar to the way that facts (EDB predicates) are separated from rules (that compute IDB predicates) in Datalog programs. Our third assumption states that all the facts are specified only at the initial moment of time. This assumption is needed in order to bring Templog closer to Datalog and to production systems, since in these languages all the facts are specified at the initial stage of the computation. Finally, we do not allow Templog programs to contain initial clauses, i.e., a Templog program is divided into facts that are true at time 0 and rules that hold at all the moments of time. If no confusion arises, we will also omit the necessity operator $\Box$ in front of permanent rules implicitly assuming that it is there.

These assumptions imply that the only facts allowed in Templog clauses have the form $p \leftarrow$, where p are ground atoms. These facts form the *temporal EDB predicates.*

There are two types of monotonicity that can be applicable to Templog programs: they can be monotone in the EDB predicates, or monotone in time. Formally,

Definition 2.1 *A program P is* monotone in the EDB predicates *if $EDB_1 \subseteq EDB_2$ implies that the least model of P applied to EDB_1 is contained in the least model of P applied to EDB_2.*

Definition 2.2 *A program P is* temporally monotone *if the least model of P at time k is contained in the least model of P at time $k+1$, for all EDB instances.*

It is easy to see that every Templog program is monotone in EDB predicates. On the other hand, not all Templog programs are temporally monotone. For example, consider the program consisting of the single fact $p \leftarrow$. Predicate p is clearly true at time 0 but not at time 1.

2.2 Semantics of Datalog and Datalog$^\neg$ Programs

Let P be a Datalog program and let E be a set of EDB predicates. Consider the sequence of database states D_0, D_1, $\ldots$, D_n, $\ldots$, where $D_{i+1} = EVAL(D_i)$ and $D_0 = E$. The mapping $EVAL$ computes new facts from the facts in D_i by applying all the Datalog rules in P simultaneously[2].

Traditionally, the meaning of Datalog program is associated with the least fixpoint of the mapping $EVAL$, i.e., with the first value D_i in the sequence above for which $D_i = D_{i+1}$ [Ull88]. Similarly, the meaning of a Datalog$^\neg$ program under *inflationary semantics* [AV88b, KP88], is defined as the fixpoint of the mapping

$$D_{i+1} = D_i \cup EVAL(D_i) \tag{1}$$

As we explained above, we shall be interested in the sequence $D_0, D_1, \ldots, D_n, \ldots$, rather than just its fixpoint in this paper. Furthermore, this sequence will be obtained with equation (1) for Datalog$^\neg$ programs.

3 Simulations

So far, we have described two general approaches to defining semantics of infinite temporal databases, as fixpoints of Templog programs, and as sequences of states generated by Datalog. The next question we address is how to compare their expressive power.

Consider a Datalog program P_D. We could define a Templog program P_T by replacing each rule $\alpha \leftarrow \beta$ in P_D by the rule $\circ\, \alpha \leftarrow \beta$ and adding the rule $\circ\, p \leftarrow p$ for each predicate p in P_D. It follows immediately that the trajectory of database states generated by P_D is equal to the trajectory corresponding to the least model of P_T, for all values of the EDB predicates.

This motivates the concept of *exact simulation* of one program by another. Let P be a program (either some variant of Datalog or Templog) and let E be the initial values of the EDB predicates, at time $t = 0$. Program P then generates the trajectory $P(E)$ in the manner described in Section 2. We use $P_i(E)$ to denote the i-th state in this trajectory. Exact simulation is then defined as follows.

[2]For precise definition of $EVAL$ see [Ull88, p. 115]

Definition 3.1 *Program P* exactly simulates *program P' if for any initial instance of EDB predicates E, trajectories $P(E)$ and $P'(E)$ coincide.*

"Exactly simulates" is, clearly, a commutative relationship, i.e. if P exactly simulates P' then P' exactly simulates P. If two programs exactly simulate each other then they are isomorphic in terms of trajectories they generate. The concept of exact simulation was defined and studied in [KT89, TK91], but for our purposes it can be too restrictive. Consider the following example.

Example 2 Let P be the Templog program

$$\begin{aligned} \circ\, r(x,y) &\leftarrow p(x,y) \\ q(x,y) &\leftarrow \circ\, r(x,y) \\ q(x,z) &\leftarrow q(x,y) \wedge q(y,z) \end{aligned}$$

with the EDB predicate p and the IDB predicates q and r. The Datalog program P'

$$\begin{aligned} p'(x,y) &\leftarrow p(x,y) \\ q(x,y) &\leftarrow p'(x,y) \\ p''(x,y) &\leftarrow p'(x,y) \\ r(x,y) &\leftarrow p''(x,y) \\ q(x,z) &\leftarrow q(x,y) \wedge q(y,z) \end{aligned}$$

(with two auxiliary predicates p' and p'' that are not part of the trajectory) does not simulate P exactly. This is because only the second step in the trajectory of P' matches the initial zero step in the trajectory of P.

However, P' is "close enough" to P, in the sense that any trajectory generated by P contains a subtrajectory generated by P'. Furthermore, this subtrajectory has the property that any of its two consecutive steps are no more than two places apart in the trajectory of P (steps 0 and 1 in P correspond to steps 0 and 2 in P', and all other adjacent steps in P are also adjacent in P').

∎

This motivates the following definitions.

Definition 3.2 *Program P* unboundedly simulates *program P' if for any initial instance of EDB predicates E, the trajectory $P'(E)$ forms a subtrajectory of $P(E)$.*

In Example 2, program P' unboundedly simulates program P. In this case, however, we can make a stronger statement about the relationship between the two programs. We first need some preliminary definitions.

We say that a trajectory TR is *n-congruent* to a trajectory TR' if TR is a subsequence of TR' and for all steps i, TR_i and TR_{i+1} are never more than n steps apart in TR'. This means that any two subsequent steps in TR cannot be arbitrarily far away in TR'.

Definition 3.3 *Program P* boundedly simulates *program P' if there is a number n such that for any initial instance of EDB predicates E, the trajectory $P'(E)$ is n-congruent to the trajectory $P(E)$.*

The program P' in Example 2 is an example of a program that boundedly simulates the program P. Furthermore, the value of constant n in Definition 3.3 in this case is $n = 2$.

If program P boundedly simulates program P' then P simulates P' unboundedly. Also, if program P simulates program P' exactly then P simulates P' boundedly. Exact simulation is the strongest type of simulation, while unbounded simulation is the weakest, with bounded simulation in between.

We can now formally define our goal of comparing the expressive power of two different formalisms. Let $\mathcal{F}$ and $\mathcal{F}'$ be two formalisms (variants of Datalog or Templog).

Definition 3.4 *We say that $\mathcal{F}$ has at least the same* expressive power *as $\mathcal{F}'$ (denoted as $\mathcal{F} \subseteq \mathcal{F}'$) if any program P' from $\mathcal{F}'$ can be simulated with some program P from $\mathcal{F}$. If $\mathcal{F}'$ has strictly more expressive power than $\mathcal{F}$ we denote it as $\mathcal{F} \subset \mathcal{F}'$.*

There are three different types of simulation, and therefore three different ways to compare the expressive power of the languages. In Section 4 we look at unbounded simulation and in Section 5 we look at bounded simulation. Exact simulation is too restrictive to be of much interest, and so we do not discuss it further.

4 Expressive Power of Datalog and Templog for Unbounded Simulations

Throughout this section we will use the term "simulation" to refer to unbounded simulation. To begin, we show that Templog is strictly more expressive than Datalog.

Proposition 4.1 *Datalog $\subset$ Templog*

Proof: The inclusion follows from our description in Section 3 of how a Datalog program can be simulated with a Templog program. The fact that the inclusion is proper follows from the fact that any Datalog trajectory is temporally monotone, whereas there are Templog programs that are not. ∎

Since Inflationary Datalog$^{\neg}$ is temporally monotone, it follows by the same argument that there are Templog programs that cannot be simulated even by Inflationary Datalog$^{\neg}$.

These results follow from the fact that Templog, like doubly negated Datalog, allows "deletion", i.e., it allows facts to be true at some point of time, and false later. Since, for the purposes of this paper, we are focusing our attention on Datalog (with negation), we shall look at Templog programs that are restricted to be temporally monotone. We can do this by considering only Templog programs that contain the rule $\circ p \leftarrow p$, for every predicate in the program. We call this restriction of Templog *Monotone Templog*, or MTemplog.

Definition 4.2 *A MTemplog program P is a Templog program with the property that, for each predicate p in P, P contains the rule*

$$\circ p \leftarrow p$$

We shall now show that Datalog is properly contained even in the restricted version of Templog.

Theorem 4.3

$$Datalog \subset MTemplog$$

Proof: The containment follows immediately from the fact that the simulation in Section 3 describes a monotone Templog program. To show that the containment is proper, consider the following MTemplog program P, with EDB p, and IDB q, r and s.

$$\begin{aligned} q(x,y) &\leftarrow p(x,y) \\ q(x,z) &\leftarrow q(x,y) \wedge q(y,z) \\ \circ\, r(x,y) &\leftarrow q(x,y) \\ \circ\, s(x,y) &\leftarrow r(x,y) \end{aligned}$$

The trajectory described by P is as follows[3]

time	q	r	s
0	$TC(p)$	$\emptyset$	$\emptyset$
1	$TC(p)$	$TC(p)$	$\emptyset$
2	$TC(p)$	$TC(p)$	$TC(p)$

Assume that P' is a Datalog program that simulates P. Consider the computation of P' on some fixed nonempty EDB E. By the definition of unbounded simulation, there must be some point l in the computation that corresponds to the result of P at time 2. This means that in $P'_l(E)$, $r = s = TC(p)$.

Since transitive closure cannot be expressed in first-order logic [AU79] and since bounded queries can be expressed in first-order terms [Cos89], this means that transitive closure is unbounded. Therefore, there must be some EDB E' for which the computation of $q = TC(p)$ takes $m > l$ steps. By renaming of the elements of p, we may assume that E and E' have no elements in common, and therefore if we merge the contents of p in E and E', we get a new EDB E'' for which (a) the computation of $q = TC(p)$ takes $m > l$ steps, and (b) E'' is a superset of the value of p in E.

However, a simple induction shows for all i, $P'_i(E)$ is monotone in E, and therefore, whenever $i \geq l$, the value of s in $P'_i(E'')$ is nonempty.

On the other hand, because P' simulates P, there must be some i for which $P(E'') = P'_i(E'')$. Since the value of q at this point is equal to $TC(p)$, it follows that $i \geq m > l$. But the value of s at this point must be empty, a contradiction. ∎

The next question we want to address is the relationship between Datalog$^\neg$and MTemplog. This is a much harder problem, whose difficulty comes from the fact that a Datalog$^\neg$

[3]$TC(p)$ is the transitive closure of p.

program generates a trajectory of its states in *one* forward movement in time, whereas (intuitively) a MTemplog program generates a trajectory as a result of *multiple* forward and backward movements. This would happen if, for example, a MTemplog program contained clauses of the form $\circ p \leftarrow q$ and $q \leftarrow \circ p$. Therefore, the challenge is to simulate multiple forward and backward movements of an MTemplog program with a single forward movement of a Datalog$^\neg$ program.

In this paper, we prove a partial result in this direction: We show that Datalog$^\neg$ *with counters* is strictly more expressive than MTemplog. Following Chomicki [CI88], we consider a 2-sorted first order logic, the domain of the second sort consisting of natural numbers with the total order imposed on them and with the *successor* function defined for that sort. At most one parameter in a predicate can belong to this sort. Datalog$^\neg$ programs defined over this logic will be called *Datalog$^\neg$ programs with counters.* We start the proof of the result with a technical lemma.

Let P be a Datalog, Datalog$^\neg$, or MTemplog program. Note that in all the three cases, program P is temporally monotone. Therefore, for each choice of EDB predicates E, there is some time t for which $P_t(E) = P_{t+1}(E)$. The mapping from E to $P_t(E)$, denoted by $F(P)$ (for fixpoint), is the standard definition of the meaning of a Datalog or Datalog$^\neg$ program. Note, however, that the definition above enables us to talk about the fixpoint of a MTemplog program, which we will call the *IDB fixpoint* to distinguish it from the fixpoint defined in Section 2.1.

There is in fact a simple relation between the fixpoint of Datalog and MTemplog programs. Let P be a MTemplog program, and let P' be the Datalog program obtained from P by deleting all the temporal operators from P. It is easy to see that the two programs have the same fixpoint, i.e.,

Lemma 4.4 $F(P) = F(P')$. ∎

Using this lemma, we can show:

Theorem 4.5 *For any MTemplog program P there exists a Datalog$^\neg$ program with counters that unboundedly simulates P.*

Proof. As shown by Baudinet [Bau89], Templog is equivalent to its fragment TL1 that allows only the *next* temporal operator in its rules (except the case when the necessity operator makes a rule permanent). We can therefore, without loss of generality, restrict our attention to TL1 programs that have atoms optionally preceded by at most one *next* operator. Let P be a TL1 program. We construct a program P''' in Datalog$^\neg$ with counters that simulates P.

The first step is to construct an intermediate Datalog$^\neg$ program P'. P' is obtained from P by replacing each Templog rule $q \leftarrow p$ in P with the Datalog$^\neg$ rule $q' \leftarrow p'$. This rule is obtained from the rule $q \leftarrow p$ by replacing each atom $A(x_1, \ldots, x_n)$ in P by the two-sorted predicate $A'(x_1, \ldots, x_n, t)$ and by replacing each next-atom $\circ A(x_1, \ldots, x_n)$ by $A'(x_1, \ldots, x_n, t+1)$. In these predicates t is a variable over the separate sort "time".

We next construct the program P'' by adding rules to P' to detect the IDB fixpoint of P'. The idea here is that P'' detects the IDB fixpoint of program P' and computes the values of all of its predicates $A'_i(x_1, \ldots, x_n, t)$ for the finite set of values t before the IDB fixpoint of P' is reached.

We now describe the construction of P''. As an initial step, replace program P with the program P^* obtained from P by removing all the temporal operators from its rules as was done in Lemma 4.4. Then P^* is a Datalog program with fixpoint $\{A_1^*, A_2^*, \ldots, A_n^*\}$, where A_i^* is an IDB predicate in P^*. By Lemma 4.4, this fixpoint is equal to the IDB fixpoint of P.

We next construct P'' from P' and P^* as follows. First, compute the fixpoint of P^* and detect when this computation has terminated using the technique from [AV88a]. For each predicate A_i in P^*, maintain the predicate $\mathit{previous\_unless\_last}_i$ that trails A_i by one step and is equal to A_i until the time the fixpoint of P^* is reached. At that time, $\mathit{previous\_unless\_last}_i$ differs from A_i. The difference between the two predicates at the last moment is the point when the fixpoint of P^* has been reached.

After the fixpoint of P^* is computed, start execution of P' (its execution can be held off until the fixpoint of P^* is reached with a "trigger" that becomes true at that time). At each computation step of P' check if there exists t such that the predicates A'_i at that time are equal to predicates A_i^*, i.e., for all i taken over the IDB predicates of P and for all $x_1, \ldots, x_n$, $A'_i(x_1, \ldots, x_n, t) \Leftrightarrow A_i^*(x_1, \ldots, x_n)$. If such t exists, then it means that the IDB fixpoint of P'

is reached. Set a flag FP to be true at that time. The rules that implement all the actions described above form the program P''.

Finally, the program P''' that simulates P is constructed from P'' by introducing an additional predicate $C(t)$ and adding the following rules:

$$\begin{array}{c} C(0) \leftarrow FP \\ A_i(x_1, \ldots, x_n) \leftarrow A'_i(x_1, \ldots, x_n, t) \wedge C(t) \\ C(t+1) \leftarrow C(t) \end{array}$$

for each IDB predicate A_i in P''. The reason that P''' unboundedly simulates program P is that the trajectories of P and P''' coincide from the first time point when $C(0)$ is true. ∎

We next show that the converse is false: There are Datalog$^{\neg}$ programs (even without counters) that cannot be simulated in Templog.

Theorem 4.6 *There are Datalog$^{\neg}$ programs that cannot be simulated in MTemplog.*

Proof: This theorem follows from Lemma 4.4 and from the fact that inflationary Datalog$^{\neg}$ queries have the expressive power of fixpoints [AV88b, KP88], and, therefore are more expressive than Datalog queries. We are interested in trajectories, rather than fixpoints. However, if there is a Datalog$^{\neg}$ program whose fixpoint cannot be computed by a Templog program, then clearly its trajectory cannot be simulated either. ∎

Corollary 4.7 *MTemplog $\subset$ Datalog$^{\neg}$ (with counters)*

Proof: The inclusion follows from Theorem 4.5, and the proper inclusion from Theorem 4.6. ∎

We proved the result for Datalog$^{\neg}$ programs with counters. The question whether MTemplog programs can be unboundedly simulated by Datalog$^{\neg}$ programs without counters is still open.

This completes our results on unbounded simulation. We showed that Datalog$^{\neg}$ and Templog programs are incomparable under unbounded simulations. We also showed that Datalog programs are strictly less expressive than MTemplog programs and that MTemplog programs are strictly less expressive than Datalog$^{\neg}$ programs with counters.

5 Expressive Power of Datalog and Templog for Bounded Simulations

We now turn our attention to bounded simulation. Since bounded simulation implies unbounded simulation, this means that the negative results from Section 4 carry over to bounded simulations. In particular, Datalog$^\neg$ and Templog classes of programs are incomparable for the bounded simulations as well. Furthermore, Theorem 4.3 still holds.

In contrast to the unbounded case, we show that there are MTemplog programs that cannot be boundedly simulated by any Datalog$^\neg$ program.

We use the notion of a *bounded* Datalog program.[4] A query q on a Datalog$^\neg$ program P is *bounded* [Cos89] if there is a constant bound on the number of iterations it takes to compute the fixpoint of q, and this bound is independent of the EDB predicates in P. A query q on a Datalog (or Datalog$^\neg$) program P is *first-order expressible* if there is a first-order formula defined on EDB predicates of P that computes the same answer as the fixpoint of q for all the values of the EDB predicates of P.

We first need two preliminary lemmas. The first extends the result due to Cosmadakis [Cos89] that every bounded Datalog query is first-order expressible to Datalog$^\neg$ queries. The proof uses a similar argument to [Cos89].

Lemma 5.1 *Every bounded Datalog$^\neg$ query is first-order expressible.*

The second lemma is due to Ajtai and Gurevich [AG89]

Lemma 5.2 [Ajtai & Gurevich] *Every first-order expressible Datalog query is bounded.*

We can now show:

Theorem 5.3 *There is an MTemplog program that cannot be boundedly simulated with any Datalog$^\neg$ program.*

Proof: Consider the following Templog program P (together with the monotonicity rules required by the definition of MTemplog):

[4]This has no relation with bounded simulation.

$$\begin{aligned} \circ\, q(x,z) &\leftarrow q(x,y) \wedge q(y,z) \\ r(x,y) &\leftarrow q(x,y) \\ r(x,y) &\leftarrow \circ\, r(x,y) \end{aligned}$$

Assume that P' is a Datalog$^\neg$ program that boundedly simulates P. The definition of bounded simulation implies that the predicate r, considered as a Datalog$^\neg$ query, is bounded. Lemma 5.1 then implies that r is first-order expressible. As a Datalog$^\neg$ query, r computes the transitive closure of q. Since q, as a Datalog query, is unbounded, q is not first-order expressible (Lemma 5.2) and hence neither is r, a contradiction. ∎

The intuitive reason why Theorem 5.3 holds is because the trajectory of a MTemplog program is obtained by moving forward and backward in time, and there is no a priori bound on the number of times we must do this to reach the fixpoint. Theorem 5.3 says that there is no way to simulate this in a bounded number of steps by a Datalog$^\neg$ program.

6 Conclusions and Future Work

We have compared the expressive power of two families of languages, those based on Templog and on Datalog in terms of trajectories these formalisms can generate. For unbounded simulations, we showed that Datalog$^\neg$ and Templog programs are incomparable, that Datalog programs are strictly less expressive than MTemplog programs, and that MTemplog programs are strictly less expressive than Datalog$^\neg$ programs with counters.

For bounded simulations, Datalog programs can be simulated with MTemplog programs, but some MTemplog programs cannot be boundedly simulated with Datalog$^\neg$ programs. The question whether MTemplog programs can be unboundedly simulated in Datalog$^\neg$ remains open. Future work includes extending our results to languages with deletion, such as Datalog$^{\neg *}$ and full Templog.

References

[AG89] M. Ajtai and Y. Gurevich. Datalog vs. first-order logic. In *FOCS*, 1989.

[AM89] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8:277–295, 1989.

[Ari86] G. Ariav. A temporally oriented data model. *TODS*, 11(4):499–527, 1986.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of 6th ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[AV88a] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. Technical Report 900, I.N.R.I.A., 1988.

[AV88b] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proceedings of PODS Symposium*, pages 240–250, 1988.

[AV89] S. Abiteboul and V. Vianu. Fixpoint extensions of first-order logic and Datalog-like languages. In *IEEE Symposium on Logic in Computer Science*, 1989.

[Bau89] M. Baudinet. Temporal logic programming is complete and expressive. In *Symp. on Principles of Programming Languages*, pages 267–280, 1989.

[BFG$^+$89] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A framework for programming in temporal logic. In *Stepwise Refinement of Distributed Systems*, pages 94–129. Springer-Verlag, 1989. LNCS 430.

[BFK86] L. Brownston, R. Farrell, and E. Kant. *Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming.* Addison-Wesley, 1986.

[BNW91] M. Baudinet, M. Niezette, and P. Wolper. On the representation of infinite temporal data and queries. In *Proceedings of PODS Symposium*, pages 280–290, 1991.

[CC87] J. Clifford and A. Croker. The historical data model (HRDM) and algebra based on lifespans. In *Proceedings of the International Conference on Data Engineering*, 1987. IEEE Computer Society.

[CI88] J. Chomicki and T. Imielinski. Temporal deductive databases and infinite objects. In *Proceedings of PODS Symposium*, pages 61–73, 1988.

[Cos89] S. S. Cosmadakis. On the first-order expressibility of recursive queries. In *Proceedings of PODS Symposium*, pages 311–323, 1989.

[CW83] J. Clifford and D. S. Warren. Formal semantics for time in databases. *TODS*, 8(2):214–254, 1983.

[dMS88] C. de Maindreville and E. Simon. Modelling non deterministic queries and updates in deductive databases. In *International Conference on Very Large Databases*, pages 395–406, 1988.

[Gab89] D. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of Colloquium on Temporal Logic in Specification*, pages 402–450. Springer-Verlag, 1989. LNCS 398.

[Gad88] S. K. Gadia. A homogeneous relational model and query languages for temporal databases. *TODS*, 13(4):418–448, 1988.

[GJ91] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *International Conference on Very Large Databases*, 1991.

[GT86] S. Ginsburg and K. Tanaka. Computation-tuple sequences and object histories. *TODS*, 11(2):186–212, 1986.

[KKR90] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Proceedings of PODS Symposium*, pages 299–313, 1990.

[KP88] P. G. Kolaitis and C. H. Papadimitriou. Why not negation by fixpoint? In *Proceedings of PODS Symposium*, pages 231–239, 1988.

[KSW90] F. Kabanza, J.-M. Stevenne, and P. Wolper. Handling infinite temporal data. In *Proceedings of PODS Symposium*, pages 392–403, 1990.

[KT89] Z. M. Kedem and A. Tuzhilin. Relational database behavior: Utilizing relational discrete event systems and models. In *Proceedings of PODS Symposium*, 1989.

[MD89] D. McCarthy and U. Dayal. The architecture of an active, object-oriented database system. In *Proceedings of ACM SIGMOD Conference*, 1989.

[NA88] S. B. Navathe and R. Ahmed. TSQL – a language interface for history databases. In C. Rolland, F. Bodart, and M. Leonard, editors, *Temporal Aspects in Information Systems*, pages 109–122. North-Holland, 1988.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, cashing and views in database systems. In *Proceedings of ACM SIGMOD Conference*, pages 281 – 290, 1990.

[Sno87] R. Snodgrass. The temporal query language TQuel. *TODS*, 12(2):247–298, 1987.

[Sno90] R. Snodgrass. Temporal databases: Status and research directions. *ACM SIGMOD Record*, 19(4):83–89, December 1990.

[Tan86] A. U. Tansel. Adding time dimension to relational model and extending relational algebra. *Information Systems*, 11:343–355, 1986.

[TCG$^+$] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases.* Benjamin/Cummings. Forthcoming.

[TK89] A. S. Tuzhilin and Z. M. Kedem. Querying and controlling the future behavior of complex objects. In *Proceedings of International Conference on Data Engineering*, February 1989.

[TK91] A. Tuzhilin and Z. M. Kedem. Modeling dynamics of databases with relational discrete event systems and models. Working Paper IS-91-5, Stern School of Business, NYU, 1991.

[Tuz91] A. Tuzhilin. Temporal logic as a simulation language. In *Proceedings of the International Conference on Artificial Intelligence and Simulation*, New Orleans, Louisiana, April 1991.

[Ull88] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Via87] V. Vianu. Dynamic functional dependencies and database aging. *JACM*, 34(1):28–59, 1987.

[WF90] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of ACM SIGMOD Conference*, pages 259 – 270, 1990.