Temporal Deductive Databases *

Marianne Baudinet
Université Libre de Bruxelles[†]

Jan Chomicki
Kansas State University[‡]

Pierre Wolper
Université de Liège[§]

October 13, 1992

Keywords: query processing, logic programming, temporal logic, constraint logic programming, expressiveness, computational complexity.

1989 CR Classification: F.2.2, F.4.1, H.2.3, I.2.3.

Abstract

We survey a number of approaches to the problem of finite representation of infinite temporal extensions. Two of them, $Datalog_{1S}$ and *Templog*, are syntactical extensions of *Datalog*; the third is based on repetition and arithmetic constraints. We provide precise characterizations of the expressiveness and the computational complexity of these languages. We also describe query evaluation methods.

1 Introduction

In a historical database, the information that is stored includes temporal attributes. These attributes indicate *when* (at what time) the information is valid. Within this context, many options are available. The temporal information can be attached to small or large chunks of data, the underlying time model can be discrete or continuous, and the time model can be based on time points or on time intervals. However, whichever of these options are selected, the choice of a representation for the extension of the temporal attributes remains a fundamental issue.

*To appear in: A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases* (Benjamin/Cummings, 1993).

[†]Address: Informatique; 50 Avenue F.D. Roosevelt, C.P. 165; 1050 Brussels; Belgium. Email: mb@montefiore.ulg.ac.be

[‡]Address: Department of Computing and Information Sciences, 234 Nichols Hall, Manhattan, KS 66506-2302. Email: chomicki@cis.ksu.edu

[§]Address: Institut Montefiore, B28; 4000 Liège Sart-Tilman; Belgium. Email: pw@montefiore.ulg.ac.be

In the classical database approach, extensions are represented explicitly, that is, by the set of values that correspond to them. For instance, if some fact is true every day at 8 a.m., its extension includes as many time points as there are days on which the fact is true. Thus, this representation is only usable if the number of days is finite and, in practice, if it is not too large. An alternative is to use an implicit representation from which the extension can be computed. This chapter describes such implicit representations and their use in historical databases.

We consider two different implicit representations of temporal extensions: one based on deductive rules, and one based on repetition and constraints. To express deductively that some information is true every day at 8 a.m., one can say that it is true at some time t_0 corresponding to the first occurrence of 8 a.m. at which the information holds, and then specify that, if the information is true at some time t, it is true at time $t + 24$ hours. This form of representation has been studied since [CI88] and [Bau89b] and is the subject of Section 3.

An alternative representation of the fact that some information is true every day at 8 a.m. is to write that it is true at every time point of the form $t_0 + 24n$ where t_0 corresponds to the first occurrence of 8 a.m. at which the information holds and n is a variable ranging over the integers. Furthermore, one can limit the range of the variable n by introducing constraints. For instance, one can specify that the temporal extension being considered consists of all time points x such that x is of the form $t_0 + 24n$ and such that $0 \leq x \leq 673$. Section 4 is devoted to this representation, which was introduced in [KSW90] and further studied in [BNW91]. Related ideas about the use of constraints in databases were presented in [KKR90] and [Rev90].

The main advantage of implicit representations is expressiveness. They make the representation of infinite extensions possible and often allow a more compact representation of finite extensions. Expressiveness is thus one of our main concerns when surveying the formalisms we have just outlined. Actually, we are concerned with two distinct notions of expressiveness. The first concerns the extensions that can indeed be represented. Following [BNW91], we call it *data expressiveness*. Note that data expressiveness is not an issue when dealing with finite extensions since all finite extensions can be represented explicitly. However, infinite extensions form an uncountable set, thus not all sets of infinite extensions are finitely representable. Each particular formalism is able to represent a specific set of infinite extensions. This set constitutes the data expressiveness of the formalism. The second notion of expressiveness we consider is more akin to the notion of expressiveness usually considered in deductive databases: it is the expressiveness of the query language. We refer to this notion as *query expressiveness* [BNW91]. Section 2 gives precise definitions of these concepts.

Expressiveness is not the only issue. Representing information is only useful if it can be queried with reasonable efficiency. Therefore, our second concern when reviewing representation formalisms for infinite temporal extensions is the complexity of evaluating queries.

We consider various query evaluation methods and we mostly focus on *data complexity* [Var82], which appears to be the most important notion in database problems. Finally, whenever possible, we discuss work that is related to our main subject of concern either by its goals or by the techniques it uses.

2 Basic Model and Expressiveness

2.1 Basic Model

In what follows, we consider a temporal domain $\mathcal{T}$ that is either isomorphic to the set of natural numbers $\mathcal{N}$ (Section 3) or to the set of integers $\mathcal{Z}$ (Section 4). We then consider relations that have any number of non-temporal data attributes and either one (Section 3), or one or more (Section 4) temporal attributes ranging over the temporal domain $\mathcal{T}$. A *temporal database* is a finite collection of such relations. The extension of the non-temporal attributes is assumed to be represented explicitly. Our concern is the representation of the possibly infinite extensions of the temporal attributes.

Example 2.1 Consider the relation

$$train\_schedule(departure\_time, arrival\_time)$$

that represents a train schedule. Assuming a time granularity of one minute, an explicit representation of part of its extension could be as follows.

train_schedule

7:02	8:20
8:02	9:20
9:02	10:20
10:02	11:20
11:02	12:20
12:02	13:20

2.2 Expressiveness: definitions

Any finite temporal extension can be represented as in Example 2.1. However, if we also consider the possibility of infinite extensions, then the situation is entirely different. Indeed, it is possible to devise formalisms that can represent infinite temporal extensions, but it is not possible to finitely represent all infinite temporal extensions. For instance, let us consider a relation with exactly one temporal argument and let us assume that the temporal domain $\mathcal{T}$ is the natural numbers. Then, the extension of such a relation consists of an ω-sequence (infinite sequence whose members are indexed by natural numbers: $0, 1, 2, \ldots$) of finite relations. Therefore, the number of possible temporal relations is uncountable, and no language with finite expressions can represent all temporal relations. It is then useful to capture the expressive power of any particular formalism used for representing temporal data. We call this the *data expressiveness* of the formalism. We now give precise definitions.

The number of temporal attributes is called the *temporal arity* of a relation and the number of non-temporal attributes is called the *data arity* of a relation.

For the sake of clarity and simplicity, we consider databases consisting of a finite set $\wp$ of predicates that we take to be all of temporal arity 1 and of data arity 0. All the definitions given below extend directly to more general cases. A temporal relation of temporal arity 1 is a subset of $\mathcal{T}$, namely the set of time instants at which the relation holds.

Definition 2.1 A *temporal database* is a function mapping every predicate in $\wp$ to a subset of $\mathcal{T}$ (the set of time instants at which the predicate holds). It is thus a function in $\mathcal{B} = (\wp \rightarrow 2^{\mathcal{T}})$. Alternatively, it can be viewed as a function in $(\mathcal{T} \rightarrow 2^{\wp})$ which associates with each time instant the predicates that are true at that time instant.

Notice that, even with just one temporal predicate ($\wp$ being a singleton set), there are $2^{\aleph_0}$ temporal databases, all of which cannot be finitely represented. We thus introduce the following notion.

Definition 2.2 The *data expressiveness* of a temporal database formalism is the set of temporal databases (functions in $\mathcal{B} = (\wp \rightarrow 2^{\mathcal{T}})$) that can be defined in this formalism.

As we will see in the following sections, the data expressiveness of the formalisms we will consider is, in several cases, characterized in terms of ultimately periodic sets.

Although in classical databases the data expressiveness is not an issue (because all extensions are finite), the expressiveness of the language for extracting data from a database – the query language – is a very crucial feature [CH82, CH85]. For a temporal database formalism, the situation is identical, and we call this expressiveness the *query expressiveness*, to avoid any possible confusion with the data expressiveness.

A temporal (yes/no) query is a "filter" on temporal databases. It can be viewed as defining the set of databases for which it is true. Thus, each set of temporal databases corresponds to a query and, since there are $2^{\aleph_0}$ databases, there are $2^{2^{\aleph_0}}$ possible yes/no queries. All these queries cannot be finitely represented, which leads to the notion of query expressiveness.

Definition 2.3 The *query expressiveness* of a temporal query language is the class of sets of temporal databases that can be defined using expressions of this language, i.e., those sets of databases for which queries in this language are true.

When viewed as a function in $(\mathcal{T} \rightarrow 2^{\wp})$, a temporal database is isomorphic to an infinite sequence of sets of predicates or, equivalently, to an infinite word on the alphabet $2^{\wp}$. In what follows, we will thus use languages of infinite words to characterize the query expressiveness of various temporal query languages.

3 Deductive Formalisms

Horn clauses with no function symbols are a purely deductive framework in which relations can be defined and queried. This Horn-clause based language is a restriction of logic programming that is usually referred to as *Datalog* [GMN84, Ull88, Ull89, GM92]. In this section, we discuss two approaches to representing temporal phenomena using two different syntactic extensions of *Datalog*.

The first approach that we consider is $\textit{Datalog}_{1S}$ (*Datalog* with one successor, previously called *temporal deductive databases*) [CI88, CI89, Cho90b, Cho90a], a temporal language that extends *Datalog* by allowing every predicate to have (at most) one temporal parameter in addition to the usual data parameters. The temporal parameter can be constructed using a specific unary function symbol denoting the *successor* function, and can be viewed as being interpreted over the natural numbers.

The second temporal extension of *Datalog* that we consider is the language *Templog* [AM89, Bau89b, Bau89a, Bau92]. *Templog* is an extension of logic programming to temporal logic. It thus extends *Datalog* by allowing a restricted use of modal temporal operators in Horn clauses.

It has been shown that $\textit{Datalog}_{1S}$ and *Templog*, although syntactically different and based on different logics, are in fact equivalent (in the sense that one is a syntactical variant of the other: see subsection 3.3). Therefore, we begin by introducing both languages, but present expressiveness, query evaluation, and complexity results only in the context of $\textit{Datalog}_{1S}$, even though some of these results were first proved for *Templog*.

3.1 $\textit{Datalog}_{1S}$

Syntax

For the sake of completeness, let us begin by recalling some standard definitions and terminology used in logic programming and *Datalog*. For further detail, the reader is referred to [vEK76, Llo87, Ull88].

An *atom* is a formula $p(t_1, \ldots, t_n)$, where p is a predicate symbol and each t_i is a term (in the usual first-order logic sense). A term or an atom is said to be *ground* if it is variable free. A *fact* is a ground atom. A *database* is a finite set of facts. A *rule* is a formula written as $A \leftarrow B_1, \ldots, B_m$ (in clausal form: $A \vee \neg B_1 \vee \cdots \vee \neg B_m$), where $A, B_1, \ldots, B_m$ are atoms; A is called the *head* and $B_1, \ldots, B_m$ the *body* of the rule. A *logic program* is a finite set of rules together with a database. A *goal* is a formula written as $\leftarrow B_1, \ldots, B_m$ (in clausal form: $\neg B_1 \vee \cdots \vee \neg B_m$). A *query* is a finite set of rules together with a goal. All variables in rules and goals are implicitly universally quantified.

We distinguish *extensional database* (*EDB*) and *intensional database* (*IDB*) predicates. The EDB predicates correspond to the relations provided in the database, whereas the IDB predicates correspond to the derived relations, that is, those that are defined by rules. EDB predicates may thus appear in database facts, in bodies of rules and in goals, whereas IDB predicates can only appear in rules and goals, and not in database facts.

Datalog is the language of *function-free* logic programs, that is, logic programs in which the only terms are constants or variables (such terms are called *data terms*).

Datalog$_{1S}$ is an extension of *Datalog* in which predicates may have a single distinguished parameter (argument), which is said to be *temporal*, in addition to the usual *data* parameters. Temporal parameters are *temporal terms*, which are built from a distinguished constant 0, variables, and the unary function symbol $+1$ (written in postfix). For instance, if T is a variable, then $0, T, T+1$ are temporal terms. For the sake of clarity, we will write n instead of $(\cdots((0 \underbrace{+1)+1)\cdots+1)}_{n\ times}$ and $T+n$ instead of $(\cdots((T \underbrace{+1)+1)\cdots+1)}_{n\ times}$. The set of variables used to build temporal terms must be disjoint from the set of data variables. Furthermore, temporal terms cannot appear as data parameters of predicates. Predicates having a temporal parameter are said to be *temporal*.

Semantics

In the general case of logic programs, the semantics of a program P can be characterized by its *least Herbrand model* M_P, which is the intersection of all its Herbrand models [vEK76, Llo87]. M_P contains all the ground atoms that are logical consequences of P, and is usually infinite when the program contains temporal terms. M_P can also be obtained by the (possibly infinite) iteration of the mapping T_P defined as follows. Let I denote a Herbrand interpretation.

$$\begin{aligned} T_P(I) \quad = \quad \{A \mid\ & A \text{ is a fact in } P \text{ or } A \leftarrow A_1, A_2, \ldots, A_k \\ & \text{is a ground instance of a rule in } P \\ & \text{and } A_1 \in I, A_2 \in I, \ldots, A_k \in I\} \end{aligned}$$

The mapping T_P corresponds to one step of ground inference from I. The successive iterations of T_P are defined as follows.

$$T_P^0 = \emptyset$$
$$T_P^{k+1} = T_P(T_P^k)$$
$$T_P^\omega = \bigcup_{k \geq 0} T_P^k$$

T_P^ω is the least fixpoint of T_P, which we denote $\mathit{lfp}(T_P)$. It coincides with the least Herbrand model of P, that is $\mathit{lfp}(T_P) = M_P$ [vEK76].

For *Datalog*, the same semantics apply. However, since *Datalog* does not allow function symbols in programs, every Herbrand model of a program is finite and hence M_P is finite. The successive iterations of T_P thus also constitute an effective method for computing M_P. This method is known as *bottom-up evaluation*.

For $\mathit{Datalog}_{1S}$, the situation is similar, except for the presence of temporal terms. Since the set of ground temporal terms is isomorphic to the integers, the least model M_P is often infinite. Hence bottom-up evaluation is not guaranteed to terminate for $\mathit{Datalog}_{1S}$.

Example 3.1 The following rules schedule the meetings of a professor with his students.

$$\mathit{meets}(T, X) \leftarrow \mathit{meets\_first}(T, X)$$
$$\mathit{meets}(T+1, Y) \leftarrow \mathit{follows}(X, Y), \mathit{meets}(T, X)$$

Let us consider a database containing the following facts.

$\mathit{meets\_first}(0, \mathit{emma})$
$\mathit{follows}(\mathit{emma}, \mathit{kathy})$
$\mathit{follows}(\mathit{kathy}, \mathit{emma})$

The rules can be used to derive the following infinitely many facts from the database.

$\mathit{meets}(0, \mathit{emma})$
$\mathit{meets}(1, \mathit{kathy})$
$\mathit{meets}(2, \mathit{emma})$
$\mathit{meets}(3, \mathit{kathy})$
$\ldots$

Notice that these facts form a periodic set when viewed as a function of the integers, but that their period depends on the contents of the database.

Example 3.2 Consider the following rules governing the schedule of backups in a distributed system.

$$backup(T+24, X) \leftarrow backup(T, X)$$
$$backup(T, Y) \leftarrow dependent(X, Y), backup(T, X)$$

The first rule states that a backup on a machine should be taken *every 24 hours*. The second rule requires that the backups should be taken simultaneously on all *dependent* machines (e.g., sharing files). Notice that, whatever database these rules are applied to, the derived facts will again be periodic, but this time, the period will not depend on the contents of the database.

3.2 *Templog*

Syntax

Templog is a syntactic extension of logic programming based on temporal logic. It is built from a version of temporal logic that views time as isomorphic to the natural numbers [AM89, Bau89b]. In this language, predicates can vary with time, but the time point they refer to is defined implicitly by temporal operators rather than by an explicit temporal argument.

The three temporal operators used in *Templog* are $\bigcirc$ (*next*), which refers to the next time instant, $\Box$ (*always*), which refers to the present and all the future time instants, and $\Diamond$ (*eventually*), the dual of $\Box$, which refers to the present or to some future time instant. In *Templog*, $\bigcirc$ is allowed both in the head and in the body of clauses, $\Box$ is allowed only in the head of clauses or outside entire clauses, and $\Diamond$ is allowed only in the body of clauses (possibly nested with conjunction).

The abstract syntax of *Templog* clauses can be provided formally as follows. In these syntax rules, A stands for an atom, N for a *next-atom*, that is, an atom preceded by a finite number of $\bigcirc$'s, ε denotes an empty formula, and the symbols "::=" and "|" belong to the metalanguage.

Body: $B ::= \varepsilon \mid A \mid B_1,\ B_2 \mid \bigcirc B \mid \Diamond B$

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

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

Program clause: $C ::= IC \mid PC$

Goal clause: $G ::= \leftarrow B$

Notice that there are two types of clauses in *Templog*: *initial clauses*, which express a statement that holds at the initial time, and *permanent clauses*, which express a statement that holds at any time instant.

The semantics of a temporal logic formula is provided with respect to a temporal interpretation D that is an infinite sequence $D_0, D_1, \ldots, D_i, \ldots$ of classical first-order interpretations (one classical interpretation for each time instant). In *Templog*, only predicate symbols have time-varying meanings; constants (and function symbols) are assumed to be independent of time. The semantics of the temporal logic operators used in *Templog* is the following ("$\models_{D_i} F$" means "F is true in D_i"):

$\models_{D_i} \bigcirc F$ iff $\models_{D_{i+1}} F$
$\models_{D_i} \Box F$ iff for every $j \in \mathcal{N}, \models_{D_{i+j}} F$
$\models_{D_i} \Diamond F$ iff for some $j \in \mathcal{N}, \models_{D_{i+j}} F$

Example 3.3 The $Datalog_{1S}$ program of Example 3.1 is formulated in *Templog* as follows.

$\Box(meets(X) \leftarrow meets\_first(X))$
$\Box(\bigcirc\, meets(Y) \leftarrow follows(X, Y), meets(X))$

The facts are written as follows in *Templog*:

$meets\_first(emma)$
$follows(emma, kathy)$
$follows(kathy, emma)$

They are initial clauses thus expressing statements that hold at the initial time. Therefore, we need an additional rule to express the fact that the predicate *follows* does not vary with time (so what is true at the initial time instant remains true forever).

$\Box\, follows(X, Y) \leftarrow \Diamond\, follows(X, Y)$

Just like classical logic programs, *Templog* programs have semantics that can be expressed in terms of a least (temporal) Herbrand model, and in terms of the least-fixpoint of a mapping similar to T_P [Bau89b, Bau92].

3.3 $Datalog_{1S}$ and *Templog* related

The fragment TL1 of *Templog* that disallows the use of $\Diamond$ in the body of clauses and the use of initial clauses with $\Box$ in the head can easily be seen to be a syntactic variant of

Datalog$_{1S}$. (The role played in this fragment of *Templog* by the temporal operators is played in *Datalog*$_{1S}$ by the temporal parameters). It has also been shown that any *Templog* program can be transformed into an equivalent TL1 program [Bau89a, Cho90a].

The reverse translation, that is from *Datalog*$_{1S}$ to *Templog*, is also straightforward [Cho90a]. The only interesting point of this translation concerns time-independent predicates in *Datalog*$_{1S}$, which each require a specific *Templog* clause stating this time-independence, as illustrated in Example 3.3 above by the predicate *follows*.

In what follows, we will only consider *Datalog*$_{1S}$, and all results that we will provide also hold for *Templog* (or were actually initially established for *Templog*).

3.4 Expressiveness

As described in Section 2.2, there are two distinct aspects to the expressiveness of a database formalism: *data expressiveness*, which captures the class of temporal relations definable in the formalism, and *query expressiveness*, which captures the class of definable queries.

Let us recall from Section 2.2 that when discussing expressiveness, we limit our attention to databases consisting of a finite set $\wp$ of predicates of temporal arity 1 and data arity 0. We call *Monadic Datalog*$_{1S}$ the fragment of *Datalog*$_{1S}$ in which all predicates satisfy these conditions, that is, are monadic and temporal. Notice that, for *Templog*, this condition corresponds to predicates having no parameter at all. Such temporal predicates are also called *temporal propositions* and the corresponding fragment of *Templog* is called *propositional Templog*.

This restriction to parameters of temporal arity 1 and data arity 0 is not a real limitation, because every program can be reformulated in this way by instantiating its data parameters in all possible ways with the data constants appearing in the program. The size of the resulting program is at most polynomial in the number of such constants but exponential in the arity of its atoms.

Data expressiveness

The temporal database represented by a *Monadic Datalog*$_{1S}$ program P corresponds exactly to the least Herbrand model M_P of the program. As stated below, it turns out that every such model is ultimately periodic.

Definition 3.1 A function f defined on $\mathcal{N}$ is *ultimately periodic* with period (k, l), where $k \geq 0$, and $l > 0$, if for all $i \geq k$, we have $f(i + l) = f(i)$.

The least Herbrand model M_P of a *Monadic Datalog*$_{1S}$ program can be seen as a function that maps every time instant $i \geq 0$ to the set of predicates that are true at time i.

Theorem 3.1 *[CI88, Cho90a, CI92] The least Herbrand model M_P of a Monadic Datalog$_{1S}$ program P is ultimately periodic with period $(c+k, l)$ such that $k, l \leq 2^{|\wp|}$ and c is the maximum depth of a ground temporal term in P.*

Since the converse is straightforward, namely, every periodic function in $(\mathcal{N} \rightarrow 2^{\wp})$ can be represented as a *Datalog$_{1S}$* program, the ultimately periodic sets characterize exactly the data expressiveness of *Datalog$_{1S}$*. This result can be used to determine whether an infinite set of natural numbers can be represented as a *Datalog$_{1S}$* program. For example, the set $\{i^2 \mid i \in \mathcal{N}\}$ is aperiodic and thus cannot be represented in *Datalog$_{1S}$*.

Query expressiveness

At first thought, it might seem that since *Datalog$_{1S}$* and *Templog* are languages whose programs and goals are formulas of a logic, their expressivenes is simply the expressiveness of their underlying logic. This is not so. The intuitive reason for this is that the deduction hidden in the rules intervenes to add to the expressiveness of the underlying logics.

Such a phenomenon has been observed for standard logic programming. (Remember that even though a logic program is a logic formula, formulating its semantics requires a minimization operation – the least fixpoint). Indeed, it has been shown that, for instance, one could express in *Datalog* queries that are not expressible in first-order logic, for instance queries about the transitive closure of a given relation [AU79, CH85]. On the other hand, not all queries expressible in first-order logic can be expressed in *Datalog*. Chandra and Harel have shown in [CH85] that the query expressiveness of standard *Datalog* corresponds to the expressiveness of a positive fragment of fixpoint logic on finite structures, namely the fragment in which a formula consists of a least fixpoint operator applied to a positive existential formula.

The query expressiveness of (propositional) *Templog* was first studied in [Bau89a, Bau89b], using an approach similar to Chandra and Harel's. In a way, the results for *Templog* parallel those for *Datalog*. However, they relate *Templog* queries to fixpoint extensions of temporal logic and to finite automata on infinite words. For the sake of uniformity, we present these results using *Monadic Datalog$_{1S}$* rather than propositional *Templog*.

The query expressiveness of a language has been defined in Section 2.2 to be the class of sets of temporal databases that can be defined by queries in that language. Let us see what it means for a *Datalog$_{1S}$* query to define a set of databases. We consider a *Datalog$_{1S}$* query Q consisting of rules and of the ground goal $\leftarrow u(i)$ where u is a distinguished IDB predicate and i a natural number (denoting a time instant). Assuming that $\wp = \{p_1, \ldots, p_n\}$ is the set of EDB predicates occurring in Q, we can look at Q as being parameterized by $\wp$. Indeed, the query Q characterizes the set of temporal interpretations of $\wp$ that make $u(i)$ true. Notice that we consider here *infinite* databases.

Example 3.4 The query consisting of the rules

$$u(T) \leftarrow p(T)$$
$$u(T) \leftarrow u(T+1)$$

together with the goal $\leftarrow u(5)$ is parameterized by the set $\{p\}$. One can show that, for any ground $i \geq 0$, $u(i)$ will hold exactly when the database contains a fact expressing that p holds at time i or at some time instant later than i, or equivalently, when $\diamond p$ holds at time i. The predicate u can thus be seen as defining the temporal operator *eventually*.This simulation of *eventually* is actually used in the proof that *Templog* is equivalent to its fragment TL1 (c.f., Section 3.3).

Example 3.5 Let us consider the query consisting of the rules

$$u(T) \leftarrow q(T)$$
$$u(T) \leftarrow p(T), u(T+1)$$

together with the goal $\leftarrow u(i)$ for some ground $i \geq 0$. One can show that this query will hold exactly when the database is such that p holds from time i and on until q holds. This corresponds to the temporal logic formula $p\,U\,q$ holding at time i. (U denotes the operator *until* whose semantics is defined as follows:

$$\models_{D_i} F\,U\,G \text{ iff } \quad \text{for some } j \geq i, \models_{D_j} G \text{ and for all } k \text{ such that } i \leq k < j, \models_{D_k} F).$$

One can thus view the above query as defining the temporal operator *until*.

As Example 3.5 illustrates, *Datalog*$_{1S}$ (or *Templog*) is able to express a temporal logic operator *until*, which is known to be inexpressible in temporal logic with $\bigcirc, \Box$, and $\diamond$ [GPSS80]. This is analogous to the observation that *Datalog* is able to express non first-order concepts such as the transitive closure.

Example 3.6 [Bau89a] The predicate u defined by the rules below holds exactly when p holds an even number of time instants later.

$$u(T) \leftarrow p(T)$$
$$u(T) \leftarrow u(T+2)$$

One can show that the predicate u can be seen as the result of applying to p a temporal operator that is the dual of the operator *even*, which was shown by Wolper to be inexpressible in temporal logic [Wol82, Wol83]. It is expressible, however, in extensions of temporal logic with either fixpoint quantifiers or finite-automaton operators.

The expressiveness of *Monadic Datalog$_{1S}$* queries is related by the following result to the expressiveness of temporal logic with fixpoint quantifiers named μTL and defined by Vardi in [Var88].

Theorem 3.2 *[Bau89a, Bau91] The query expressiveness of Monadic Datalog$_{1S}$ is equivalent to the expressiveness of the fragment of the temporal fixpoint calculus μTL that contains only positive formulas and least-fixpoint operators.*

This result means that there are formulas of temporal logic that are not expressible in *Datalog$_{1S}$*. Expressing all of temporal logic in μTL can require the use of greatest fixpoints or the alternation of a greatest and a least fixpoint [Par81].

Example 3.7 [Bau89a] A consequence of this theorem is that the operator $\Box$ (*always*) cannot be defined in *Monadic Datalog$_{1S}$* because $\Box$ corresponds in μTL to a greatest-fixpoint formula. It is thus impossible to write a *Datalog$_{1S}$* query that will hold exactly when $\Box\, p$ holds. In particular, the obvious candidate

$$p(T)$$
$$q(T) \leftarrow p(T)$$

does not do the job because it makes q true whenever p is true and p always true. Moreover, no query can assert a fact about an EDB predicate as the above query does. Notice, however, that the dual of $\Box$, namely $\Diamond$, which corresponds in μTL to a positive least-fixpoint formula, is expressible in *Monadic Datalog$_{1S}$*, c.f., Example 3.4.

Because temporal logic is particularly suitable for defining infinite sequences, it is customary to relate its expressiveness to languages of infinite words or to finite automata on infinite words (for comprehensive recent reviews of those see [Eme90, Tho90]). Such a relation has also been established for propositional *Templog*, or equivalently, *Monadic Datalog$_{1S}$*.

Let us recall that a temporal database specifies, for each time instant, which predicates hold. It can thus be seen as a function from $\mathcal{N}$ (or ω) to $2^{\wp}$, or equivalently as an infinite word, an ω-*word*, over the alphabet $2^{\wp}$. The languages of interest here are the ω-regular languages defined as follows.

Definition 3.2 An ω-language L is *finitely regular* if there is a regular language L' such that L can be obtained by extending the words of L' to infinite words in all possible ways.

Theorem 3.3 *[Bau89a, Bau89b] Monadic Datalog$_{1S}$ queries correspond to the class of finitely regular ω-languages.*

The correspondence described in Theorem 3.3 is only completely accurate when *Datalog*$_{1S}$ programs allow EDB predicates to appear negated in the body of rules (but this is a technicality). Propositional temporal logic with the operators $\bigcirc$, $\Box$, $\Diamond$, and also U is known to have the same expressiveness as the class of *star-free* ω-regular languages [Tho81, Tho90]. The classes of finitely regular ω-languages and of star-free ω-languages are incomparable. Therefore, *Monadic Datalog*$_{1S}$ is in some respects more powerful and in some respects less powerful than propositional temporal logic (with $\bigcirc$, $\Box$, $\Diamond$, and U).

3.5 Query evaluation

We discuss here various theoretical and practical issues involved in the evaluation of *Datalog*$_{1S}$ queries. We consider here only finite *Datalog*$_{1S}$ programs, in particular the set of facts is finite.

We start by defining two distinct modes of query evaluation: recognition (providing a yes-no answer) and generation (providing all the answers). Thereafter, we turn to the complexity of query evaluation. We focus on data complexity and show that the data complexity of query evaluation for *Datalog*$_{1S}$ is PSPACE-complete. Therefore, unless PTIME=PSPACE, it is impossible to evaluate all *Datalog*$_{1S}$ queries in polynomial time.

Next, after reviewing various evaluation methods, including bottom-up and top-down, we present two approaches to dealing with infinite query answers. The first, more traditional, is concerned with checking *query safety*, that is, testing whether a query has a finite answer. The second, provides a framework for finitely representing the entire answer, as well as efficiently computing such a representation. Finally, we discuss polynomial time computable *Datalog*$_{1S}$ queries.

Modes of query evaluation

Assume $P = Z \cup D$ is a logic program where Z is a set of rules and D is a database, and $\leftarrow G$ a goal. Then $Q = Z \cup \{\leftarrow G\}$ is a query.

Definition 3.3 An *answer substitution* to the query Q on the database D is a mapping θ of variables in G to ground terms such that $P \models G\theta$ (where $G\theta$ is the result of applying θ to G). Alternatively, we could require that $\models_{M_P} G\theta$ where M_P is the least Herbrand model of P. An *answer* is the set of all answer substitutions.

If G is already ground, then the answer is either empty or contains a single substitution with an empty domain. We call the first case a *no* answer, and the second case a *yes* answer.

Definition 3.4 *Recognition* (yes-no query evaluation) is a computation of the yes-no answer to a query containing a ground goal. *Generation* (all-answers query evaluation) is a computation of the entire query answer (or a finite representation of the query answer)

.

Computational complexity

Following [CH82, Var82], we study *data* complexity, that is, complexity as a function of the number of tuples in the database, assuming that the query and the database schema are fixed.

Theorem 3.4 *[Pla84, CI88] Under the data complexity measure, recognition is PSPACE-complete for $Datalog_{1S}$ queries.*

The issue of generation is somewhat more complicated as a query answer may be infinite. We will come back to it later in this section, after having introduced formal tools for the finite representation of such answers.

Evaluation procedures

The ideal query evaluation procedure for $Datalog_{1S}$ should have the following properties:

- be applicable to both recognition and generation,
- terminate when the query answer is finite (in particular for yes/no answers),
- scale down: work in polynomial time for interesting subsets of $Datalog_{1S}$,
- scale up: work also for arbitrary logic programs (of course, in this case, termination of even yes-no query evaluation cannot be guaranteed, as logic programs can encode arbitrary Turing machine computations [AN78, Tar77, Llo87]).

With these criteria in mind, we will review various query evaluation techniques of logic programs. Unfortunately, to our knowledge no single method satisfies all these criteria.

Bottom-up. Bottom-up query evaluation is implemented by a repeated application of the T_P operator introduced in Section 3.1. It terminates when no new facts are computed. Thus, if the least Herbrand model of a program is infinite, as may very well happen for $Datalog_{1S}$ (see the examples), bottom-up evaluation does not terminate. It has though some redeeming characteristics: it works equally well for recognition and generation, and is applicable to an arbitrary logic program. It is interesting to note that various optimization methods developed for logic programs [Ull89] fail in general to guarantee the termination of $Datalog_{1S}$ queries and may even introduce additional sources of non-termination [Cho91]. Bottom-up evaluation fails to terminate even for the simplest $Datalog_{1S}$ programs that use the $+1$ function symbol in an essential way, thus it does not scale down.

In [CI88, Cho91], we introduced *depth-bounded* bottom-up evaluation: a variant of bottom-up evaluation in which facts containing "too large" terms (the bound on term size is calculated from the text of the program) are simply discarded without losing completeness. This method can be used for both recognition and generation, terminates for queries with finite answers, and scales down. Unfortunately, it does not scale up and is incomplete for arbitrary logic programs.

Top-down. Top-down evaluation (SLD-resolution [Llo87], or TSLD-resolution for *Templog* [Bau89b, Bau92]) in its practical incarnations, e.g., the Prolog evaluation strategy, introduces additional problems with termination. Guaranteeing termination through memoing [TS86, Vie89], breadth-first search or ground loop checking is possible for *Datalog* programs but not for programs with function symbols, in particular $Datalog_{1S}$ programs. Similarly to the bottom-up case, a test of whether the depth of terms does not exceed a certain bound can be added to top-down evaluation in order to obtain termination. Once this is achieved, top-down evaluation can be used for both recognition and generation. On the other hand, the presence of a test on term depth will make such a method incomplete for arbitrary logic programs.

Joyner's method. In [Joy76], Joyner proposed a resolution procedure R_2 that uses atom ordering and clause condensation (which is essentially tableau minimization [Ull89]). This procedure has several attractive properties. First, it is a complete recognition procedure for arbitrary logic programs. (In fact, it is also a refutation procedure for arbitrary first-order formulas). Second, it is terminating for $Datalog_{1S}$ queries with finite answers and does not require for termination that a bound (calculated from the text of the program) be supplied. However, if R_2 is combined with SLD-resolution, the resulting procedure is incomplete. For completeness, R_2 seems to require the ability to resolve rules among themselves. It is not clear how to efficiently implement such a facility.

Infinite answers

Assume that Z is a set of rules and $\leftarrow G$ a goal.

Definition 3.5 A query $Q = Z \cup \{\leftarrow G\}$ is *relatively safe* [Kif88] for a (finite) database D if the answer to Q on D is finite. A query is *universally safe* [Kif88] if it is relatively safe for any finite database.

Safe queries are well-behaved: they will not fail to terminate just trying to print all the answer substitutions. So it is important to be able to effectively test for both kinds of safety.

Theorem 3.5 *[CI88, Cho90a] Under the data complexity measure, testing relative safety of $Datalog_{1S}$ queries is PSPACE-complete.*

Theorem 3.6 *[CI88, Cho90a, Cho91] Testing universal safety of $Datalog_{1S}$ queries is decidable.*

How can we deal with unsafe queries? Most of the time, such queries make perfect sense. In Example 3.1, one could ask the query "when does Emma meet the professor?". This query has an infinite answer: $0, 2, 4, \ldots$.

Least Herbrand models of $Datalog_{1S}$ programs can be finitely represented using the notion of *relational specification* [CI88, Cho90a, CI92]. The same construct can be used to finitely represent infinite query answers.

Definition 3.6 A *relational specification* S is a triple (T, W, B) where T is a finite set of ground temporal terms, W is a finite set of ground rewrite rules, both sides of which are ground temporal terms, and B is a finite $Datalog_{1S}$ database.

The terms in T are said to be *representative* and B is called the *primary database*. We write $t \stackrel{W}{\leadsto} t_0$ to indicate that the ground term t can be rewritten to t_0 using the rules in W and no more rewritings are applicable (t_0 is a canonical form of t).

Definition 3.7 A relational specification $S = (T, W, B)$ *represents* the least Herbrand model M_P of a $Datalog_{1S}$ program P if the three following conditions hold.

- For every temporal fact $p(t, \bar{a}) \in M_P$, there is a term $t_0 \in T$ such that $t \stackrel{W}{\leadsto} t_0$ and $p(t_0, \bar{a}) \in B$.
- For every term $t_0 \in T$ such that $p(t_0, \bar{a}) \in B$ and every fact $p(t, \bar{a})$ such that $t \stackrel{W}{\leadsto} t_0$, $p(t, \bar{a}) \in M_P$.
- The sets of non-temporal facts in M_P and B are identical.

Example 3.8 A relational specification $S = (T, W, B)$ representing the least Herbrand model of Example 3.1 may look as follows.

$$
\begin{aligned}
T &= \{0, 1\} \\
W &= \{2 \rightarrow 0\} \\
B &= \{\mathit{follows}(\mathit{emma}, \mathit{kathy}),\ \mathit{follows}(\mathit{kathy}, \mathit{emma}),\ \mathit{meets\_first}(0, \mathit{emma}), \\
&\quad\ \mathit{meets}(0, \mathit{emma}),\ \mathit{meets}(1, \mathit{kathy})\}
\end{aligned}
$$

Let us assume that $Q = Z \cup \{\leftarrow G\}$ is a query, D is a database, and $S = (T, W, B)$ is a relational specification representing $M_{Z \cup D}$. Then Q can be evaluated just by evaluating $\leftarrow G$ in S, without any reference to the program $Z \cup D$. If G is a ground atom, then it is rewritten using the rules in W until no more rewrites are possible and then it is checked whether the resulting atom is in B. If G is not ground, the query $\leftarrow G$ is evaluated in S, which is just a finite relational database. There will be finitely many answer substitutions, each representing possibly infinitely many original answer substitutions. The correspondence between those two types of substitutions are captured by the rewrite rules, so the rewrite rules themselves should be a part of the query answer.

In the case of $Datalog_{1S}$, a relational specification representing M_P has a particularly simple form. The set W contains exactly one rewrite rule, namely

$$k + l + c \rightarrow k + c$$

where (k, l) is a period of M_P (see subsection 3.4) and c is the maximum depth of a ground temporal term in a fact from P.

Theorem 3.7 *[Cho90a, CI92] A relational specification representing M_P for a $Datalog_{1S}$ program P can be constructed in polynomial space (under the data complexity measure).*

Thus in terms of the complexity class, constructing a finite representation of the entire, possibly infinite, answer to a $Datalog_{1S}$ query is not harder than providing a yes/no answer. The size of a relational specification may, however, be exponential.

Tractable $Datalog_{1S}$

In [Cho90b], we showed how to use the notion of a relational specification to define fragments of $Datalog_{1S}$ for which queries can be evaluated in polynomial time (under the data complexity assumption).

Theorem 3.8 *[Cho90b] The size of a relational specification representing M_P is polynomial if and only if the specification can be computed in polynomial time.*

Once we have a relational specification representing M_P, we can use it to answer queries. Therefore, if we can prove that, for a fragment of $Datalog_{1S}$, the size of relational specifications is polynomial, then polynomial time evaluation of queries will also be guaranteed. In [Cho90b], we identified several interesting fragments of $Datalog_{1S}$ with this property.

3.6 Negation

Horn clause languages such as *Datalog*$_{1S}$ are said to be *monotonic* because they can express only *monotonic* inferences. In the area of temporal reasoning, this means, for instance, that *frame axioms*, which state that everything remains the same unless explicitly modified, cannot be expressed. A simple extension of *Datalog*$_{1S}$, namely *Stratified Datalog*$_{1S}$, remedies this problem.

A stratified program can be seen as divided into layers or strata. It allows the use of negation in the body of rules, but in such a way that there never be any recursion through negation. In other words, the predicates that may appear negated in the body of a rule in a layer ℓ are those that are defined in layers lower than ℓ. For a complete discussion of stratification and formal semantics of programs with stratified negation, we refer the reader to [ABW88, Apt90]. Stratified programs have a distinguished Herbrand model, the *perfect model*, and query answers can be defined with respect to this model.

Example 3.9 Assuming a database of facts that define the relation *promoted*, the following *Datalog*$_{1S}$ program with negation is stratified.

$$\begin{array}{l} position(T+1, X, P') \leftarrow position(T, X, P), promoted(T, X, P') \\ position(T+1, X, P) \leftarrow position(T, X, P), \neg promoted(T, X, P') \end{array}$$

Adding stratified negation to *Datalog*$_{1S}$ results in a significant increase in query expressiveness, while keeping data expressiveness and complexity of query evaluation unchanged.

Data Expressiveness

Intuitively, stratified negation does not enhance the data expressiveness because the complement of a periodic temporal database is also a periodic temporal database.

Theorem 3.9 *[Cho90a] For a Monadic Stratified Datalog*$_{1S}$ *program P, the perfect model M_P is ultimately periodic.*

Query Expressiveness

Having proven that the query expressiveness of *Datalog* corresponds to the expressiveness of the positive existential fragment of fixpoint logic on finite structures, Chandra and Harel suggested that extending *Datalog* with stratified negation would lift its expressiveness to the level of the full fixpoint logic on finite structures [CH85]. However, Kolaitis [Kol91] showed, using of result of Dahlhaus's [Dah87], that *Datalog* enhanced with stratified negation is still strictly weaker than fixpoint logic on finite structures.

We have pointed out similarities between the query expressiveness results of *Datalog* and those of *Monadic* $Datalog_{1S}$. However, for the extensions with stratified negation, the similarity does not carry over. Indeed, the expressiveness of Propositional *Templog* or *Monadic* $Datalog_{1S}$ queries was shown to correspond to the positive fragment of the temporal fixpoint calculus μTL; but extending $Datalog_{1S}$ with stratified negation does lift the expressiveness of $Datalog_{1S}$ queries up to the level of the full temporal fixpoint calculus [Bau89a, Bau91]. This calculus is also known to correspond exactly to the full class of ω-regular languages.

Theorem 3.10 *[Bau89a, Bau91] Monadic Stratified* $Datalog_{1S}$ *queries correspond to the class of all* ω*-regular languages, and to the full temporal fixpoint calculus* μTL.

Example 3.10 For instance, $\Box p$ can be expressed using double negation in *Stratified* $Datalog_{1S}$ as follows:

$$np(T) \leftarrow \neg p(T)$$
$$np(T) \leftarrow np(T+1)$$
$$u(T) \leftarrow \neg np(T)$$

Complexity of Query Evaluation

Theorem 3.11 *[Cho90a] Under the data complexity measure, the recognition problem is PSPACE-complete for Stratified Datalogqueries. A relational specification representing a perfect model of a Stratified* $Datalog_{1S}$ *program can be constructed in polynomial space.*

Relatively little is known about query evaluation methods for Stratified $Datalog_{1S}$. Clearly, a general method for evaluating stratified logic programs like SLDNF-resolution [Llo87] can be used, but it will not guarantee termination. The method used in the proof of Theorem 3.11 is rather impractical, although it achieves the upper complexity bound.

4 Constraint-Based Formalisms

A database tuple t of schema $R = (A_1, \ldots, A_n)$ assigns values $(a_1, \ldots, a_n)$ to the attributes of R. Alternatively, it can be viewed as imposing equality *constraints* on the values of these attributes:

$$A_1 = a_1 \wedge A_2 = a_2 \wedge \cdots \wedge A_n = a_n.$$

Recently, the idea has emerged that this view of tuples as constraints could be generalized to other than equality constraints and could be used to finitely represent data with an infinite extension. In this section, we discuss two major lines of work based on this idea. First, we present the work of [KSW90, BNW91] which is specifically oriented towards the representation of temporal data, and more specifically, of periodic temporal data. Next we turn to [KKR90] and [Rev90] which study a variety of applications of tuples viewed as constraints.

4.1 Linear Repeating Points and Constraints

4.1.1 Definition

The framework proposed in [KSW90] generalizes the notion of relational database by allowing the tuples to contain an arbitrary number of temporal attributes in addition to the usual data attributes. The temporal attributes represent periodic sets of integers, namely, *linear repeating points*. Moreover, the repeating points appearing in the tuples of a relation can be constrained with linear inequalities.

Definition 4.1 A *linear repeating point* (*lrp*) is a set

$$\begin{aligned}\{x(n) \in \mathcal{Z} \quad | \quad & x(n) = an + b, \text{ with } a, b \text{ in } \mathcal{Z}, \\ & \text{and } n \text{ ranging from } -\infty \text{ up to } +\infty \text{ in } \mathcal{Z}\},\end{aligned}$$

where $\mathcal{Z}$ denotes the set of integers. Such a set is simply denoted by $an + b$.

Notice that an lrp is either an infinite periodic set of integers or a singleton (if $a = 0$). For instance, the lrp $5n+3$ denotes the infinite periodic set of integers $\{\ldots, -7, -2, 3, 8, 13, \ldots\}$.

Constraints on the temporal attributes are conjunctions of atomic constraints. We distinguish between general and restricted constraints. *General constraints* are built from atomic constraints that are arbitrary linear equalities or inequalities between at most two temporal attributes. *Restricted constraints* are built from equalities and inequalities, also between at most two temporal attributes, but in which the coefficients of temporal attributes are 1 (this restriction will be useful for the computation of operations on relations). More specifically, if T_i and T_j are temporal attributes, restricted atomic constraints are of the form

$$T_i \leq T_j + a, \quad T_i = T_j + a, \quad T_i \leq (\geq) a, \quad \text{or} \quad T_i = a.$$

Definition 4.2 Let $\mathcal{T}$ denote the collection of lrps and $\mathcal{D}$ a collection of nontemporal data values. A *generalized tuple* of temporal arity m and data arity ℓ is an element of $\mathcal{T}^m \times \mathcal{D}^\ell$, together with (general or restricted) constraints on the temporal components. In other words, a *general tuple* of temporal arity m and data arity ℓ is a ground tuple of the form

$$(a_1 n_1 + b_1, \ldots, a_m n_m + b_m, d_1, \ldots, d_\ell) \text{ with } \mathit{constraints}(T_1, \ldots, T_m)$$

where

- each $a_i n_i + b_i$ $(1 \leq i \leq m)$ is an lrp (we assume that the variables n_i are all distinct),
- each d_k $(1 \leq k \leq \ell)$ is a data constant from $\mathcal{D}$,
- $constraints(T_1, \ldots, T_m)$ denotes a finite set of constraints over the temporal attributes $T_1, \ldots, T_m$.

Such a generalized tuple is in fact a finite representation of a possibly infinite set of ground tuples, namely the set

$$\{(t_1, \ldots, t_m, d_1, \ldots, d_\ell) \;\mid\; t_1 \in (a_1 n_1 + b_1), \ldots, t_m \in (a_m n_m + b_m), \text{ and } constraints(t_1, \ldots, t_m) \text{ is satisfied } \}.$$

The constraints $constraints(t_1, \ldots, t_m)$ in a generalized tuple thus define a subset of the cartesian product of the lrp's $(a_1 n_1 + b_1) \times \ldots \times (a_m n_m + b_m)$, namely the subset of tuples that are actually in the database.

Example 4.1 For instance, the generalized tuple $(2n_1 + 3,\, 2n_2 + 5)$ constrained by the equality $T_2 = T_1 + 2$ represents the infinite set of tuples

$$\{\ldots, (-1, 1), (1, 3), (3, 5), \ldots\},$$

which is a subset of the cartesian product $(2n_1 + 3) \times (2n_2 + 5)$.

Example 4.2 The generalized tuple of the relation *train_schedule*

train_schedule			
$40n_1 + 5$	$40n_2 + 65$	*liège*	*brussels*

with $T_1 \geq 0 \wedge T_2 = T_1 + 60$

represents the following infinite set of "point-tuples" (ground tuples).

$$\{(t_1, t_2, \textit{liège}, \textit{brussels}) \;\mid\; t_1 \in (40n_1 + 5),\; t_2 \in (40n_2 + 65), \; t_1 \geq 0 \text{ and } t_2 = t_1 + 60\}$$

Definition 4.3 A *generalized relation* is a finite set of generalized tuples of a given schema.

Example 4.3 The generalized relation *action* below describes the timing of some activities of two robots.

action

0	1	*robot1*	*task1*	
$10n_1 + 2$	$10n_2 + 6$	*robot1*	*task2*	$T_1 = T_2 - 4 \wedge T_1 \geq 0$
$8n_1 + 5$	$8n_2 + 7$	*robot2*	*task1*	$T_1 = T_2 - 2 \wedge T_1 \geq 8$
$10n_1$	$10n_2 + 3$	*robot2*	*task2*	$T_1 = T_2 - 3$

It must be noted that generalized relations with an arbitrary number of temporal attributes are allowed. Generalized relations with one temporal attribute correspond to point-based temporal data (a tuple holds at specific time points). Generalized relations with two temporal attributes correspond to interval-based temporal data (a tuple holds on an interval specified by two time points). Generalized relations with more than two temporal attributes have no immediate intuitive meaning but are useful as intermediate steps in computations. For instance, when concatenating temporal intervals, the mid-point at which the intervals are joined appears as a third temporal attribute before being projected out.

4.1.2 Data Expressiveness

The concept of data expressiveness we have defined in Section 2 applies directly to generalized databases. It is quite interesting that the data expressiveness of a generalized relation with just one temporal attribute is exactly that of $Datalog_{1S}$, that is, ultimately periodic sets (see Section 3). This result can be established by relating generalized relations to Presburger arithmetic which, in the monadic case, also defines eventually periodic sets.

We consider purely temporal generalized relations (no data attributes). Also, it is necessary to distinguish between generalized relations with restricted and with general constraints, since they lead to distinct expressiveness results.

Definition 4.4

- A relation of temporal arity m (and data arity 0) is *weak-lrp-definable* if it can be defined by a generalized relation with m temporal attributes using only restricted constraints.
- A relation of temporal arity m (and data arity 0) is *lrp-definable* if it can be defined by a generalized relation with m temporal attributes using general constraints.

Definition 4.5 An m-ary relation on the integers is said to be *Presburger-definable* if it can be defined by a Presburger arithmetic formula with m free variables.

Note the following relation between Presburger-definability and eventually periodic sets (which can be found in e.g. [End72]).

Proposition 4.1 *A unary relation, that is, a set of integers, is Presburger-definable if and only if it is ultimately periodic.*

The data expressiveness results for generalized relations are then the following.

Theorem 4.1 *[KSW90]*

- *A unary relation on the integers is weak-lrp-definable if and only if it is Presburger-definable.*
- *A binary predicate on the integers is lrp-definable if and only if it is Presburger-definable.*

The result cannot be generalized to relations of temporal arity greater than 2 since the constraints used in generalized databases involve at most two temporal attributes. However, for temporal data, only relations of temporal arity 1 or 2 are of practical significance.

4.1.3 Query Languages and Their Evaluation

In [KSW90], the problem of querying generalized databases is addressed by first showing that all the usual relational algebra operations can be computed on generalized relations and then by applying these operations to the evaluation of first-order queries.

[KSW90] gives a detailed description of how the various relational algebra operations are computed on generalized databases with restricted constraints. Of all these operations, the trickiest and the one requiring that only restricted constraints be used is projection. In a classical relational database, the projection of a relation on some of its columns simply requires forsaking the other columns. For generalized databases, before forsaking a temporal column, its constraints have to be projected. This means that variables have to be eliminated from an integer constraint system. For real constraints, this could be done by a simple algorithm that eliminates variables by computing linear combinations of the constraints. But, this is not adequate for integer constraints. The solution described in [KSW90] is that, before computing a projection, database tuples be normalized, that is, that they be converted to tuples in which all the lrp's occurring as temporal attributes have the same periodicity (and all the constraints on the temporal attributes of the tuple have the coefficient 1 since all constraints are restricted). Only thereafter can the projection be carried out with algorithms that are appropriate for the real numbers.

The complexity of computing the relational algebra operations is also studied in [KSW90]. The general result is that, with a few restrictions, everything can be done in polynomial time. The first restriction is that only normalized relations be considered (that is, relations where all lrp's have the same period). The second is that the complexity be measured for a family of databases in which the schema of the relations is fixed and only the number of tuples can vary. This measure is called *fixed-schema* complexity, the case in which the schema can also vary being called *general* complexity. The exact results are the following.

Theorem 4.2 *[KSW90] The projection, selection, union, intersection, difference and join of (a) normalized generalized relation(s) can be computed in PTIME under the fixed-schema complexity measure.*

Theorem 4.3 *[KSW90] Under the general complexity measure, the projection, selection, union, intersection, and join of (a) normalized generalized relation(s) can be computed in PTIME. The difference of two normalized generalized relations can be computed in EXPTIME.*

Note that the exponential complexity of computing the difference of two generalized relations is only due to the complexity of taking the negation of a boolean combination of constraints and writing the result in disjunctive normal form.

Once algorithms for computing the relational algebra operations are known, one can consider the evaluation of first-order queries. The natural first-order language to use with generalized databases is a two-sorted first-order logic. One sort is temporal points (interpreted over the integers), the other is generic. We refer to the number of temporal parameters of a predicate as its *temporal arity* and to its number of nontemporal arguments of a predicate as its *data arity*.

The language includes one interpreted predicate of temporal arity 2 and data arity 0, namely $\leq$, and any number of uninterpreted predicates. In the spirit of what is done in relational databases, function symbols are not used on the generic sort. On the temporal sort, one interpreted function can be used, namely the successor function. Arbitrary quantification is allowed on both temporal and nontemporal variables.

Example 4.4 The following formula of this two-sorted first-order query language refers to the data of Example 4.3. It expresses the fact that there are robots x and y and time instants t_1 and t_2 such that, if x performs *task2* over a time interval $[t_1, t_2]$ of length at least 5, then y does not perform any task over any subinterval of that time interval.

$$\exists X \exists Y \exists T_1 \exists T_2 \forall T_3 \forall T_4 \forall Z \\ [\,(action(T_1, T_2, X, \text{task2}) \;\wedge\; T_1 \leq T_3 \leq T_4 \leq T_2 \;\wedge\; T_1 + 5 \leq T_2) \\ \supset \neg action(T_3, T_4, Y, Z)\,]$$

It turns out that the data-complexity [Var82] of evaluating such queries on a database is PTIME. Indeed, if the query is fixed, the schema of the database must also be fixed and one can use the complexity results obtained for fixed-schema complexity. Since all operations are polynomial under that measure, one has the following.

Theorem 4.4 *[KSW90] Determining the truth of yes/no queries on a normalized generalized database can be done in PTIME under the data complexity measure.*

Another interesting question to consider is the query expressiveness of the language we have just outlined and its relation to the query expressiveness of $Datalog_{1S}$. It turns out that, in the monadic case (one temporal argument), this language corresponds exactly to temporal logic with the four temporal operators $\bigcirc$, $\Box$, $\Diamond$, U [GPSS80]. The query expressiveness of this language is that of star-free ω-regular languages [Tho90] which is incomparable to the expressiveness of $Datalog_{1S}$ (finitely regular ω-languages) and strictly weaker than that of *Stratified* $Datalog_{1S}$ (the whole class of ω-regular languages).

4.1.4 Linear Repeating Points and Deductive Languages

The fact that, for generalized databases, the query expressiveness of a first-order language is incomparable to that of a deductive language such as $Datalog_{1S}$ suggests that studying the combination of generalized databases and deductive languages could be fruitful. Such a study was undertaken in [BNW91].

The deductive language that is considered there is similar to $Datalog_{1S}$ with the following differences.

1. Each predicate can have more than one temporal parameter.

2. Constraint atoms, that is, atoms of the form $\tau_1 = \tau_2$ or $\tau_1 < \tau_2$ where τ_1 and τ_2 are temporal terms, can appear in the body of clauses.

Notice also that temporal terms are constructed from temporal variables, the constant 0, and the successor function, as in $Datalog_{1S}$, but also the predecessor function. This is simply because, in generalized databases, the temporal domain has been taken to be the set of all integers, whereas in $Datalog_{1S}$, the temporal domain is the set of natural numbers.

Example 4.5 Let us consider the following extensional relation *course* stating that the database course is taught every Monday morning from 8 until 10. We assume that time 0 is at midnight some Monday morning and that the time unit is one hour (so one week is 168 time units).

course			
$168n_1 + 8$	$168n_2 + 10$	*database*	$T_2 = T_1 + 2$

The extension of the *course* relation is thus the infinite set of ground tuples of the form $(t_1, t_2, database)$ such that $t_1 \in (168n_1 + 8)$, $t_2 \in (168n_2 + 10)$, and $t_2 = t_1 + 2$.

The fact that database problem sessions are given right after the course and every other day thereafter can be represented as follows in our deductive language, by the derived (intensional) predicate *problems*.

$$problems(T_1 + 2, T_2 + 2, database) \leftarrow course(T_1, T_2, database)$$
$$problems(T_1 + 48, T_2 + 48, database) \leftarrow problems(T_1, T_2, database)$$

The (model-theoretic and fixpoint) semantics of the language are defined analogously to that of $Datalog_{1S}$.

This language is in fact very expressive since it can express at least all primitive recursive predicates. A consequence of this is that, in general, there is no hope of efficiently evaluating queries expressed in this language. However, as is shown in [BNW91], the situation can be different when the query is evaluated on a generalized database. The basic idea is that when the intensional database on which the query is evaluated is a generalized database, the evaluation can be done a generalized-tuple-at-a-time, rather than a tuple-at-a-time. The consequence of this is that the evaluation can terminate even when computing an infinite extension.

In the absence of constraints, evaluating queries a generalized-tuple-at-a-time will always terminate. This is easy to understand since in that case, one is only dealing with repeating points and the evaluation can be viewed as a computation in modulo arithmetic and hence on a finite domain. In the presence of constraints, termination depends on whether the constraints are strengthened or weakened by the application of the deductive rules. In the first case, termination will occur, in the latter case nothing can be said in general.

Example 4.6 Let us consider the naive bottom-up evaluation of the predicate *problems* defined in Example 4.5. It can be done by operating directly on generalized tuples (representing possibly infinite sets of ground tuples) rather than operating a tuple at a time. One obtains the following sequence of generalized tuples (we omit the data argument *database*)

$$\begin{array}{lll}
(168n_1 + 10, & 168n_2 + 12) & T_2 = T_1 + 2 \\
(168n_1 + 58, & 168n_2 + 60) & T_2 = T_1 + 2 \\
(168n_1 + 106, & 168n_2 + 108) & T_2 = T_1 + 2 \\
(168n_1 + 154, & 168n_2 + 154) & T_2 = T_1 + 2 \\
(168n_1 + 202, & 168n_2 + 204) & T_2 = T_1 + 2 \\
(168n_1 + 250, & 168n_2 + 252) & T_2 = T_1 + 2 \\
(168n_1 + 298, & 168n_2 + 300) & T_2 = T_1 + 2 \\
(168n_1 + 346, & 168n_2 + 348) & T_2 = T_1 + 2
\end{array}$$

after which the evaluation stops since no new points are added to the extension of the predicate. Indeed,

$$(168n_1 + 346,\ 168n_2 + 348) \quad = \quad (168(n_1 + 2) + 10,\ 168(n_2 + 2) + 12)$$

is a set of tuples of integers contained in a previously obtained set of tuples.

The deductive language we have just discussed is also studied in [Rev90]. However, there the study is without any connection to generalized databases of linear repeating points. The main result that is established is a closed form for *Datalog* queries over integer order. However, this closed form is only obtained in a restricted case which does not allow the use of incrementation over recursion. It should be noted that this restriction implies that periodic sets are not definable. Finally, [CMT90] studies the complexity of query evaluation for various constraint programming languages among which is the language described in this section. Not surprisingly, the result is that this problem is undecidable (in fact r.e.-complete).

4.2 Constraint Databases

[KKR90] presents a general study of constraints in databases. The general idea is similar to the one used in the generalized databases of [KSW90]: a generalized database is a set of generalized tuples, each of which is a set of constraints. The query language is then a traditional query language augmented with constraints. This general framework is then applied to several specific forms of constraints and of query languages (both first-order and deductive).

The families of constraints that are considered are:

1. *Polynomial constraints on the reals* (constraints $p(x_1, \ldots, x_n) = 0$ or $p(x_1, \ldots, x_n) < 0$ where p is a polynomial in $x_1, \ldots, x_n$);
2. *Order constraints on the rationals*, that is, constraints of the form $u = v$, $u \neq v$, $u \leq v$, $u < v$, where u and v are variables or constants;

3. *Equality constraints on the integers*, that is, constraints of the form $u = v$ or $u \neq v$;
4. *Boolean constraints*, that is, constraints of the form $s = t$, where s and t are terms of a boolean algebra.

The query languages that are considered are the relational calculus, *Datalog*, and inflationary *Datalog* with negation, all extended with constraints, but without the +1 operator. The paper then shows that for various combinations of constraint formalisms and query languages, the bottom-up evaluation of queries is of reasonable data-complexity. More precisely, depending on the chosen constraint and query languages, the complexity ranges from NC to PTIME.

5 Related Work

In [Brz91], *Templog* is seen as an instance of the Constraint Logic Programming schema [JL87]. This paper gives an elegant description of TSLD-resolution as resolution enhanced with linear equation solving.

In [TK89], a different semantics for *Datalog* is proposed with a view towards temporal applications. Every fact that belongs to the least Herbrand model of a *Datalog* program P is labelled with the numbers of the iterations of the T_P operator in which it can be derived. The totality of such facts corresponds in a natural way to a temporal interpretation, thus temporal logic is suggested as a query language. If the set of facts derived in the iteration i is denoted by D_i, then clearly $D_i \subset D_{i+1}$ for all $i \geq 0$. This restricts considerably the range of infinite temporal databases that can be represented using this framework. In fact, only databases with period $(n, 1)$ (where a single state repeats itself indefinitely) can be represented. None of the examples in our paper fits this pattern. There is, however, a clear correspondence between the above approach and *inflationary temporal deductive databases* [Cho90b], a fragment of $\mathit{Datalog}_{1S}$ admitting polynomial time query evaluation. Moreover, many temporal logic queries can be directly expressed in $\mathit{Datalog}_{1S}$ (see Section 3.4) and all such queries can be expressed in *Stratified* $\mathit{Datalog}_{1S}$ (Section 3.6).

In [TK89], *Datalog* with negation is also discussed. However, the semantics of negation is *inflationary*, which is different from the semantics of stratified negation (Section 3.6). It is not difficult to see though that fixpoint queries, which have the same expressive power as *Datalog* with inflationary negation [AV88], can be expressed in *Stratified* $\mathit{Datalog}_{1S}$.

Chen and Hsiang [CH91] have proposed an extension of logic programming that is related to temporal deductive languages. Indeed, in [CH91], they generalize the kinds of terms that can appear in the atoms of Horn clauses to what they call ω-terms, which finitely denote (potentially infinite) regular sets of terms. They develop a sound and complete unification algorithm for ω-terms, and show that logic programs with ω-terms can be translated into finite standard logic programs. However, they point out that programming with ω-terms can avoid some non-termination problems encountered with standard logic programs. This phenomenon is similar to the one we have described in Section 4.1.4 for deductive languages over databases of lrp's. It also appears in [CI88] where logic programs over temporal terms ($Datalog_{1S}$ programs) are transformed into logic programs over sets of terms in order to obtain finitely computable least models.

Formalisms for the definition of infinite periodic sets were proposed in [SS89] (many-sorted logic with term declarations) and in [TNF91] (algebraic term representations). None of these approaches, however, made a connection to logic programming.

We should also mention that $Datalog_{1S}$, as well as *Stratified* $Datalog_{1S}$, has been generalized to the case of multiple unary function symbols in [CI89, Cho90a, CI92]. These languages have applications to planning. It is interesting to note that many techniques carry over from $Datalog_{1S}$ to these languages, in particular the framework for finite representation of infinite query answers.

For the sake of completeness, let us mention that a number of other programming languages based on temporal logic have been proposed in the literature [Mos86, FKTMo86, Gab87, Gab89, BFG+89, Wad88, OW88]. These language are quite diverse. They are based on different subsets of temporal logic and use a variety of execution mechanisms. In fact, one can distinguish two major types of execution mechanisms defining two major families: either executing a formula amounts to imperatively constructing a model for this formula (the *imperative* languages), or the execution of a program is a refutation proof allowing the inference of logical consequences of the program (languages based on the logic programming paradigm, like *Templog*).

The early Tempura language of [Mos86, Hal87] belongs to the first (imperative) family. It is based on a subset of interval temporal logic whose formulas can be interpreted as traditional imperative programs. In logical terms, executing a Tempura formula (program) amounts to building a model for that formula. The languages of [Gab89, BFG+89] also belong to the imperative family, although they are based on a fragment of linear-time temporal logic in which programs are formulas stating conditions to be satisfied in the future with respect to conditions on the past. For these imperative languages, the expressiveness is directly related to the expressiveness of the underlying fragment of temporal logic. Indeed, since executing a program is just building a model of this program, no deduction engine intervenes to enhance the expressive power.

In [OW88], modal extensions of (Horn clause) logic programming are considered. One such extension is the THLP language, also described in [Wad88], which is equivalent to TL1, a fragment of *Templog*. The expressiveness of THLP is thus identical to that of TL1 and hence to that of *Templog*. In [Gab87], an extension of classical logic programming distinct from *Templog* is proposed. This language is based on a different subset of temporal logic: $\Box$ can only be applied to entire clauses and the only operators allowed in the body and in the head of clauses are $\Diamond$ and the corresponding operator for the past. A proof method is sketched for this language, but it is unclear how it could be used as the basis of an execution mechanism and of operational semantics for the language. The only semantics defined for this language is its logical semantics, which does not provide a sufficient basis for the formal study of the language's expressiveness.

6 Conclusions

The languages described in this survey show that expressive temporal databases are possible within simple and elegant frameworks. A natural question is whether these frameworks can be made practical.

A first issue is the ease of use. It is unlikely that users can be convinced to express data or queries in the precise languages that have been described here. However, these languages are a good theoretical basis and reference for more user-oriented languages. Such user-oriented languages would probably combine features of the deductive and constraint oriented formalisms.

A second consideration is the ease of implementation. $Datalog_{1S}$ which is a syntactic subset of Prolog seems easier to implement than the other languages. However, an implementation using an existing Prolog system could be too inefficient for for database purposes. Moroever, the termination of query evaluation could not be guaranteed. *Templog* requires at least the implementation of TSLD-resolution and linear repeating points require the implementation of a complete set of new relational algebra operators. This means unfortunately that implementation on top of an existing database system is not really feasible.

In conclusion, we view deductive and constraint temporal databases as developments that have fundamentally modified what can be expressed in temporal databases. However, building systems that implement these ideas in a usable way is still, at present, a challenging research program.

References

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

[AM89] M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8(3), September 1989.

[AN78] H. Andreka and I. Nemeti. The Generalised Completeness of Horn Predicate Logic as a Programming Language. *Acta Cybernetica*, 4(1):3–10, 1978.

[Apt90] K.R. Apt. Logic Programming. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 10, pages 493–574. Elsevier/MIT Press, 1990.

[AU79] A.V. Aho and J.D. Ullman. Universality of Data Retrieval Languages. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1979.

[AV88] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Languages. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1988.

[Bau89a] M. Baudinet. *Logic Programming Semantics: Techniques and Applications.* PhD thesis, Stanford University, February 1989.

[Bau89b] M. Baudinet. Temporal Logic Programming is Complete and Expressive. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1989.

[Bau91] M. Baudinet. On the Expressiveness of Temporal Logic Programming. Submitted, September 1991.

[Bau92] M. Baudinet. A Simple Proof of the Completeness of Temporal Logic Programming. In L. Fariñas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming.* Oxford University Press, 1992.

[BFG+89] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. MetateM: a Framework for Programming in Temporal Logic. Technical report, Dept. of Computer Science, University of Manchester, 1989.

[BNW91] M. Baudinet, M. Niézette, and P. Wolper. On the Representation of Infinite Temporal Data and Queries. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1991.

[Brz91] Ch. Brzoska. Temporal Logic Programming and its Relation to Constraint Logic Programming. In *International Logic Programming Symposium*, 1991.

[CH82] A.K. Chandra and D. Harel. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[CH85] A.K. Chandra and D. Harel. Horn Clause Queries and Generalizations. *Journal of Logic Programming*, 2(1):1–16, April 1985.

[CH91] H. Chen and J. Hsiang. Logic Programming With Recurrence Domains. In *International Colloquium on Automata, Languages and Programming.* Springer-Verlag, LNCS 510, 1991.

[Cho90a] J. Chomicki. *Functional Deductive Databases: Query Processing in the Presence of Limited Function Symbols.* PhD thesis, Rutgers University, New Brunswick, New Jersey, January 1990. Also Laboratory for Computer Science Research Technical Report LCSR-TR-142.

[Cho90b] J. Chomicki. Polynomial-Time Computable Queries in Temporal Deductive Databases. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Nashville, Tennessee, April 1990.

[Cho91] J. Chomicki. Depth-Bounded Bottom-Up Evaluation of Logic Programs. Submitted, June 1991.

[CI88] J. Chomicki and T. Imieliński. Temporal Deductive Databases and Infinite Objects. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Austin, Texas, March 1988.

[CI89] J. Chomicki and T. Imieliński. Relational Specifications of Infinite Query Answers. In *ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, May 1989.

[CI92] J. Chomicki and T. Imieliński. Finite Representation of Infinite Query Answers. *ACM Transactions on Database Systems*, 1992. To appear.

[CMT90] J. Cox, K. McAloon, and C. Tretkoff. Computational Complexity and Constraint Logic Programming Languages. Technical report, Department of Computer and Information Science, Brooklyn College of C.U.N.Y, 1990. To appear in *Annals of Mathematics and Artificial Intelligence.*

[Dah87] E. Dahlhaus. Skolem Normal Forms concerning the Least Fixpoint. In E. Börger, editor, *Computation Theory and Logic*. Springer-Verlag, LNCS 270, 1987.

[Eme90] E.A. Emerson. Temporal and Modal Logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier/MIT Press, 1990.

[End72] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[FKTMo86] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Tokio: Logic Programming Language based on Temporal Logic and its Compilation to Prolog. In *International Conference on Logic Programming.* Springer-Verlag LNCS 225, 1986.

[Gab87] D. Gabbay. Modal and Temporal Logic Programming. In A. Galton, editor, *Temporal Logics and Their Applications.* Academic Press, 1987.

[Gab89] D. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In B. Banieqbal, B. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398, pages 409–448. Springer-Verlag, LNCS 398, 1989.

[GM92] J. Grant and J. Minker. The Impact of Logic Programming on Databases. *Communications of the ACM*, 35(3):66–81, March 1992.

[GMN84] H. Gallaire, J. Minker, and J. Nicolas. Logic and Databases: a Deductive Approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and S. Stavi. On the Temporal Analysis of Fairness. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1980.

[Hal87] R. Hale. Temporal Logic Programming. In A. Galton, editor, *Temporal Logics and their Applications*, pages 91–119. Academic Press, 1987.

[JL87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1987.

[Joy76] W.H. Joyner Jr. Resolution Strategies as Decision Procedures. *Journal of the ACM*, 23(3):398–417, July 1976.

[Kif88] M. Kifer. On Safety, Domain Independence and Capturability of Database Queries. In *Data and Knowledge Base Conference*, Jerusalem, Israel, 1988.

[KKR90] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint Query Languages. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, April 1990.

[Kol91] P. Kolitis. The Expressive Power of Stratified Logic Programs. *Information and Computation*, 90:50–66, 1991.

[KSW90] F. Kabanza, J-M. Stevenne, and P. Wolper. Handling Infinite Temporal Data. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 392–403, Nashville, Tennessee, April 1990.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[Mos86] B.C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.

[OW88] M.A. Orgun and W.W. Wadge. A Theoretical Basis for Intensional Logic Programming. In *Symposium on Lucid and Intensional Programming*, 1988.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In P. Deussen, editor, *Theoretical Computer Science*. Springer-Verlag, LNCS 104, 1981.

[Pla84] D.A. Plaisted. Complete Problems in the First-Order Predicate Calculus. *Journal of Computer and System Sciences*, 29:8–35, 1984.

[Rev90] P.Z. Revesz. A Closed Form for Datalog Queries with Integer Order. In *International Conference on Database Theory*, pages 187–201. Springer-Verlag, LNCS 470, 1990.

[SS89] M. Schmidt-Schauss. *Computational Aspects of an Order-Sorted Logic with Term Declarations*. Lecture Notes in Artificial Intelligence 395. Springer-Verlag, 1989.

[Tar77] S-A. Tarnlund. Horn Clause Computability. *BIT*, 17:215–226, 1977.

[Tho81] W. Thomas. A Combinatorial Approach to the Theory of ω-automata. *Information and Control*, 48(3):261–283, 1981.

[Tho90] W. Thomas. Automata on Infinite Objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 133–164. Elsevier/MIT Press, 1990.

[TK89] A. Tuzhilin and Z.M. Kedem. Using Temporal Logic and Datalog to Query Databases Evolving in Time. Technical Report 484, New York University, December 1989.

[TNF91] M. Tsukamoto, S. Nishio, and M. Fujio. DOT: a Term Representation using DOT Algebra for Knowledge-bases. In *International Conference on Deductive and Object-Oriented Databases*, pages 391–410. Springer-Verlag, Lecture Notes in Compuetr Science 566, 1991.

[TS86] S. Tamaki and T. Sato. OLD Resolution with Tabulation. In *International Conference on Logic Programming*, pages 84–98. Springer-Verlag, 1986.

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

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[Var82] M.Y. Vardi. The Complexity of Relational Query Languages. In *ACM SIGACT Symposium on Theory of Computing*, pages 137–146, 1982.

[Var88] M. Y. Vardi. A Temporal Fixpoint Calculus. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1988.

[vEK76] M.H. van Emden and R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

[Vie89] L. Vieille. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.

[Wad88] W. W. Wadge. Tense Logic Programming: A Respectable Alternative. In *Symposium on Lucid and Intensional Programming*, 1988.

[Wol82] P. Wolper. *Synthesis of Communicating Processes from Temporal Logic specifications*. PhD thesis, Stanford University, 1982.

[Wol83] P. Wolper. Temporal Logic Can Be More Expressive. *Information and Control*, 56:72–99, 1983.