On the Representation of Infinite Temporal Data and Queries*
(Extended Abstract)

Marianne Baudinet†
Université Libre de Bruxelles

Marc Niézette‡
Université de Liège

Pierre Wolper§
Université de Liège

1 Introduction

Time is unbounded by nature. A temporal predicate (one that varies with time) will thus often have an infinite extension. To store such a predicate in a database, one can either artificially restrict its extension to a finite set or, more desirably, use a formalism that allows the finite representation of at least some infinite temporal extensions. Several such formalisms have been proposed in the past few years.

The formalism that extends traditional relational databases most directly is the *generalized databases* described in [KSW90]. There, database tuples are extended with *an arbitrary number* of additional columns carrying *linear repeating points*. These represent periodic sets of time points possibly constrained by linear inequalities. The query language proposed in [KSW90] is a multi-sorted first-order logic in which predicates have specific temporal parameters in addition to the usual data parameters. Queries are evaluated by computing algebraic operations on the relations of the database, and the answers are given in the form of relations with repeating point arguments. The answers to queries can thus be infinite, but always have a finite representation.

Approaching the problem from a different angle, Chomicki and Imieliński [CI88, Cho90] proposed a temporal language that extends Datalog by adding *one* temporal parameter to every Datalog predicate. This allows the definition of predicates with infinite extensions by stating for example that the predicate holds at time 0 and that, if it holds at time t, it also holds at time $t+5$. This extension of Datalog is also a natural way of querying the temporal data. A set of clauses in this language can be seen as an implicit representation of the infinite extension of temporal predicates. Chomicki and Imieliński have devoted much effort to obtaining more explicit representations of these extensions, for instance in the form of equivalence classes of congruence relations on the temporal domain [CI89, CI90].

An alternative to the language proposed by Chomicki and Imieliński is the language Templog of [AM89, Bau89a, Bau89b]. Templog is an extension of logic programming with the operators of temporal logic. Templog allows the use of $\bigcirc$ (*next*) anywhere in clauses, the use of $\Box$ (*always*) in the head of clauses or outside clauses, and the use of $\Diamond$ (*eventually*) in the body of clauses. Because of these restrictions on the use of temporal operators, Templog programs satisfy the model-intersection property and have a unique minimal model. Templog and the language of [CI88, Cho90] are actually very closely related and to a large extent notational variants of each other. This makes the comparison of the expressiveness results that have been established for Templog and for the language of Chomicki and Imieliński rather puzzling. Indeed, in [Bau89b, Bau90] the expressiveness of Templog is characterized as that of finitely regular ω-languages, whereas in [CI88] the expressiveness that is mentioned is that of periodic sets! These are certainly not identical.

The first contribution of this paper is to clarify the concepts needed to compare the expressiveness of various temporal database formalisms and to discuss the expressiveness of the formalisms we have just described. The key observation is that when considering infinite temporal databases, there are two distinct notions of expressiveness: *data expressiveness* and *query expressiveness*. The data expressiveness is the expressive power of the formalism for *storing* infinite temporal data. The

*The following text presents research results of the Belgian National incentive-program for fundamental research in artificial intelligence initiated by the Belgian State – Prime Minister's Office – Science Policy Programming. The scientific responsibility is assumed by its authors.

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

‡Address: Institut Montefiore B28, 4000 Liège Sart Tilman, Belgium. Email: niezette@montefiore.ulg.ac.be

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

To appear in *10th ACM Symposium on Principles of Database Systems*, Denver, May 1991.

query expressiveness is the expressive power of the language used for *querying* the temporal database. Although the latter notion is already familiar in classical database query languages, the former notion is never mentioned, because trivial, in the context of classical databases. Indeed, there is no discrepancy between the set of possible relations and the set of relations representable in a classical database. This is because one only considers finite relations, which are obviously all representable. In temporal databases, however, the number of possible temporal relations is uncountable, so they cannot all be finitely represented. Any choice of formalism hence imposes a restriction on the set of representable temporal relations, and it is important to be able to characterize and compare the various languages from this point of view.

With these concepts, the discrepancy between [Bau89b, Bau90] and [CI88] is simple to explain: in [Bau89b, Bau90] it is the query expressiveness that is considered, whereas [CI88] explores the data expressiveness. In fact, it is easy to show that both formalisms have the same data expressiveness (periodic sets) and the same query expressiveness (finitely regular ω-languages). Moreover, when extended with stratified negation, these languages have a query expressiveness that corresponds to the class of ω-regular languages [Bau89b, Bau90]. When limited to one temporal argument, the data expressiveness of the *generalized databases* of [KSW90] is also periodic sets. However, the expressiveness of the associated query language corresponds to the class of star-free ω-regular languages, which is incomparable to finitely regular ω-languages, but is strictly weaker than ω-regular languages. The intuitive reason for this is that, in the query language of [KSW90], negation is allowed but there is no recursion mechanism, whereas, in [CI88] and in Templog, negation is not allowed but queries can be recursive.

The situation can thus be summarized as follows. We have three equally expressive formalisms for representing infinite temporal data. They are thus all interchangeable. However, we would advocate using the formalism of [KSW90] since it is more explicit and since it allows a predicate to have an arbitrary number of temporal arguments as opposed to at most one in the other two frameworks. In addition, as is show in [CI89, CI90], any recursive definition of infinite temporal data can be converted into an explicit form and this sometimes expensive computation is better done once and for all rather than each time the data is queried.[1] As far as query languages, the situation is less straightforward. Indeed, we would like the query language to have a deductive capability as in [CI88] and in Templog, but also to be able to handle several temporal arguments as in [KSW90].

[1]Note that in this type of temporal databases, the deductive layer is used to define the temporal extension of all predicates, not just of derived predicates.

This leads us to the second contribution of our paper. We define a deductive query language that operates on the temporal databases of [KSW90]. This language allows the definition of predicates that operate on several temporal arguments. Unfortunately, the bottom-up evaluation of such predicates often leads to infinite executions. However, we show that, if some assumptions are satisfied, the queries of this language can be finitely evaluated when applied to infinite periodic data. Furthermore, their answers can be finitely represented as temporal databases (that is, in closed form). Finally, we characterize the expressiveness of this query language.

2 Existing Formalisms for Temporal Databases

This section briefly recalls the main features of the temporal database formalisms of [KSW90] and [CI88, Cho90], and of the language Templog [AM89, Bau89a, Bau89b].

2.1 Generalized Databases with Linear Repeating Points

The framework proposed in [KSW90] generalizes the notion of relational database by allowing 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 tuples of a relation can be constrained with linear inequalities.

Definition (Linear Repeating Point) A *linear repeating point* (*lrp*) is a set

$$\{x(n) \in \mathcal{Z} \mid x(n) = an + b, \text{ with } n \text{ ranging from } -\infty \text{ up to } +\infty \text{ in } \mathcal{Z}, \text{ and } a, b \text{ in } \mathcal{Z}\}$$

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

For instance, the lrp $5n+3$ denotes the infinite periodic set of integers $\{\ldots, -7, -2, 3, 8, 13, \ldots\}$.

Definition (Ground Generalized Tuple) A *ground generalized tuple* of temporal arity m and data arity ℓ is a ground tuple of the form

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

where

- each d_k ($1 \leq k \leq \ell$) is a data constant,
- each $a_i n_i + b_i$ ($1 \leq i \leq m$) is an lrp with non-zero period, that is, $a_i \neq 0$,
- *constraints*$(T_1, \ldots, T_m)$ denotes a finite set of constraints over the temporal attributes $T_1, \ldots, T_m$. Each constraint is of one of the following forms: $T_i < T_j + c$, $T_i < T_j - c$, $T_i = T_j + c$, $T_i = T_j - c$, $T_i < c$, $T_i = c$, or $c < T_i$, where c is any integer constant, $T_i, T_j \in \{T_1, \ldots, T_m\}$.

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

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

For instance, the generalized tuple $(2n_1 + 3, 2n_2 + 5)$ constrained by $T_2 = T_1 + 2$ represents the infinite set of tuples $\{\ldots, (-1, 1), (1, 3), (3, 5), \ldots\}$. Note that we impose here that all the lrp's in a generalized database have a non-zero period, an assumption which was not made in [KSW90]. This assumption will be useful when we discuss the evaluation of our deductive language in Section 4. It is not restrictive since an lrp with zero period is simply an integer constant, say c, which is nothing else than the lrp n with associated constraint $T = c$.

Definition (Generalized Database) A *generalized database* with relations $p_1, \ldots, p_r$ such that p_i ($1 \leq i \leq r$) is of temporal arity m_i and data arity ℓ_i consists, for each p_i, of a set of generalized tuples of temporal arity m_i and data arity ℓ_i.

Example 2.1 Let us consider a generalized database storing train schedules (relation *train* with temporal arity 2 and data arity 2). The following table stores the schedule of trains going from Liège to Brussels. Assuming that time 0 is at midnight some Monday morning and that the time unit is a minute, it states that there is a train leaving Liège for Brussels 5 minutes after time 0 and every 40 minutes thereafter, and arriving 60 minutes after having left.

train

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

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

The query language proposed in [KSW90] for generalized databases of lrp's is a partially interpreted first-order logic, that is, a logic in which predicates have temporal parameters interpreted over the integers in addition to the uninterpreted data parameters. The language is equipped with negation, but since it is first-order, it does not have a recursion mechanism.

2.2 The Temporal Formalism Proposed by Chomicki and Imieliński

The temporal language proposed in [CI88] and further studied in [Cho90] is exactly like Datalog [Ull88, Ull89] except that every predicate has one temporal parameter in addition to the usual uninterpreted parameters. A *temporal term* in this language is obtained from the constant 0 or from any temporal variable by applying the successor function any number of times (the temporal domain is the natural numbers, as opposed to the integers in [KSW90]).

Example 2.2 Let us consider again the train schedule of Example 2.1. The *train* relation cannot be represented as such in the language of Chomicki and Imieliński, which only allows one temporal parameter per predicate. But we can, for instance, represent the departure times and define the arrival times in terms of them.

$$\mathit{train\text{-}leaves}(5, \mathit{liège}, \mathit{brussels}) \leftarrow$$
$$\mathit{train\text{-}leaves}(t + 40, \mathit{liège}, \mathit{brussels}) \\ \leftarrow \quad \mathit{train\text{-}leaves}(t, \mathit{liège}, \mathit{brussels})$$

$$\mathit{train\text{-}arrives}(t + 60, \mathit{liège}, \mathit{brussels}) \\ \leftarrow \quad \mathit{train\text{-}leaves}(t, \mathit{liège}, \mathit{brussels})$$

■

In [CI89, CI90], the temporal language is generalized to allow *functional* terms rather than simply *temporal* terms. Functional terms are similar to temporal terms except that they are built using several function symbols. However, database predicates are still only allowed to have no more than one such functional parameter.

2.3 Templog

Templog extends logic programming to temporal logic (a version 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).

Example 2.3 The following clauses are the Templog translation of the temporal program given in Example 2.2. ($\bigcirc^i$ is an abbreviation for $\underbrace{\bigcirc \cdots \bigcirc}_{i}$.)

$$\bigcirc^5 \, \textit{train-leaves}(\textit{liège}, \textit{brussels}) \leftarrow$$

$$\Box\Big(\bigcirc^{40} \, \textit{train-leaves}(\textit{liège}, \textit{brussels}) \leftarrow \textit{train-leaves}(\textit{liège}, \textit{brussels})\Big)$$

$$\Box\Big(\bigcirc^{60} \, \textit{train-arrives}(\textit{liège}, \textit{brussels}) \leftarrow \textit{train-leaves}(\textit{liège}, \textit{brussels})\Big)$$

■

Example 2.3 illustrates the correspondence between Templog and the language of Chomicki and Imieliński. In fact, it has been shown in [Bau89b] that Templog is equivalent to a fragment of itself, namely TL1, where $\bigcirc$ is the only operator allowed within clauses, whereas $\Box$ is still allowed to appear outside entire clauses. It turns out that this fragment corresponds exactly to the language of [CI88], described in Section 2.2. This is why Templog and the language of [CI88] can essentially be seen as notational variants of each other.

3 Expressiveness Issues

A classical relational database consists of a finite number of finite relations defined by their extension. As any finite relation can be represented, in classical relational databases, there is no difference between the set of possible relations and those that are representable in a database.

In a temporal database, however, this is no longer the case. Indeed, there is a discrepancy between the set of possible temporal relations and those that are expressible in a given formalism. To see this, consider a temporal formalism in which relations have exactly one temporal attribute and let us assume that the temporal domain is the natural numbers. Then, a temporal relation consists of an ω-sequence of finite relations. So, 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.

Although in classical databases the data expressiveness is not an issue, the expressiveness of the language for extracting data from a database – the query language – is a very crucial feature (e.g. [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.

3.1 Data Expressiveness

Let us denote by $\mathcal{T}$ the temporal domain. In the case of generalized databases with linear repeating points over the integers, this temporal domain is the set of integers $\mathcal{Z}$, whereas, in the language of Chomicki and Imieliński and in Templog, this domain is the set of natural numbers $\mathcal{N}$. For the sake of clarity and simplicity, we consider databases consisting of a finite set of predicates $\wp$ that we take to be all of the same temporal arity m and of data arity 0. All the definitions given below extend directly to more general cases.

A temporal relation of temporal arity m is a subset of $\mathcal{T}^m$, namely the set of m-tuples of time instants at which the relation holds. A temporal database thus stores the m-tuples of time points at which its relations hold.

Definition (Temporal Database) A *temporal database* is a function mapping every predicate in $\wp$ into a subset of $\mathcal{T}^m$ (the set of m-tuples of time instants at which the predicate holds). It is thus a function in $\mathcal{B} = (\wp \rightarrow 2^{\mathcal{T}^m})$.

Notice that, even with one temporal predicate of arity 1, there are $2^{\aleph_0}$ temporal databases, all of which cannot be finitely represented. We thus introduce the following notion.

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

For deductive temporal databases, in which data is stored as temporal Horn clauses, the data expressiveness is characterized by the minimal models of the clausal rules of the language. Chomicki and Imieliński prove in [CI88] that their temporal database language is able to express eventually periodic sets of points[2]. More precisely, the result states that the minimal model of a set of temporal Horn rules is eventually periodic and it provides upper bounds on the offset and the periodicity. The same result holds for Templog since it can be translated into the language of [CI88].

Finally, the generalized relations with linear repeating points of [KSW90] naturally define eventually periodic sets of points, and hence, when restricted to just one temporal attribute, coincide in data expressiveness with the other two languages.

3.2 Query Expressiveness

Temporal queries are formally defined in the following way. (We use the same notational conventions as in Section 3.1).

Definition (Temporal Query)

- A *yes/no temporal query* is a function mapping any given database into an element of the set $\{0,1\}$. In other words, it is a function in $(\mathcal{B} \rightarrow \{0,1\}) = ((\wp \rightarrow 2^{\mathcal{T}^m}) \rightarrow \{0,1\})$, or equivalently, it defines a subset of the set of databases.
- An *all-answer temporal query* is a function mapping any given database into a temporal relation, that is, into a subset of $\mathcal{T}^m$ (the m-tuples of time instants at which the query holds for the given database). It is thus a function in $(\mathcal{B} \rightarrow 2^{\mathcal{T}^m}) = ((\wp \rightarrow 2^{\mathcal{T}^m}) \rightarrow 2^{\mathcal{T}^m})$.

Hence, there are $2^{|\mathcal{B}|} = 2^{2^{\aleph_0}}$ yes/no queries, "many" more than can be finitely represented. This leads us to the notion of query expressiveness.

Definition (Query Expressiveness) The *query expressiveness* of a temporal query language is the set of temporal queries that can be defined in this formalism. For yes/no queries, the query expressiveness is characterized by the class of subsets of the set of databases, that is, the class of subsets of $(\wp \rightarrow 2^{\mathcal{T}^m})$, that can be defined.

The query expressiveness of Templog without data arguments has been studied in [Bau89b, Bau90]. It is shown that the (yes/no) query expressiveness of such Templog predicates essentially corresponds to the class of finitely regular ω-languages. This result is obtained by viewing a Templog database not as a function in $(\wp \rightarrow 2^{\mathcal{T}})$ but equivalently as a function in $(\mathcal{T} \rightarrow 2^{\wp})$ and hence, when $\mathcal{T} = \mathcal{N}$, as an infinite word ($\omega$-word) over the alphabet $2^{\wp}$. An ω-language L is finitely regular if there is a regular language L' such that L can be obtained from L' by extending all the words of L' to infinite strings in all possible ways. These languages are exactly those accepted by finite-acceptance finite automata on infinite words, that is, automata that accept an infinite word if they accept a finite prefix of that word. Other results relate Templog queries and various logics such as μTL, a temporal logic extended with fixpoint quantifiers [Var88], and ETL$_f$, a temporal logic extended with automaton-operators [WVS83]. It is also shown that, when extended with stratified negation, Templog attains a query expressiveness that corresponds to the full class of ω-regular languages.

The query expressiveness results concerning Templog also apply directly to the temporal language of Chomicki and Imieliński. Indeed, these results are actually proved for the fragment TL1 of Templog, which is equivalent to the language of Chomicki and Imieliński.

When restricted to the case of one temporal parameter and to the natural numbers, the query language proposed in [KSW90] has an expressiveness that corresponds to the class of star-free ω-regular languages. Indeed, this query language is the first-order theory of one successor, which is expressively equivalent to the star-free ω-regular languages [Tho81]. It is also the expressiveness of temporal logic with the operators $\bigcirc$, $\Box$, $\Diamond$ and U (*until*) [GPSS80].

4 A Temporal Deductive Language

We consider a Horn clause deductive language where each predicate can have any number of uninterpreted (data) arguments as well as any number of temporal arguments interpreted over the integers (positive and negative). Moreover, we allow the use of the interpreted relations $<$ and $=$ (two temporal arguments), of the constant 0, and of the functions $+1$ and -1 applied to temporal arguments. On the other hand, no functions operate on data arguments. Our language is thus Datalog over integer order with the successor and the predecessor functions. It is essentially the extension of the language of [CI88] to an arbitrary number of temporal arguments.

This deductive language is used for defining the *intensional database* (IDB) relations. When we consider the evaluation of our deductive language, we will consider it in conjunction with the generalized database formal-

[2] which actually are the Presburger definable sets of points.

ism of [KSW90] (see Section 2.1) used for providing the *extensional database* (EDB) relations.

4.1 Definitions

The deductive language involves both temporal terms and data terms. There are thus two types of variables: data variables and temporal variables, which are used to construct respectively data terms and temporal terms.

Definitions (Data Term and Temporal Term)

- A *data term* is either an uninterpreted constant or a data variable.
- A *temporal term* is defined inductively as a temporal variable, the constant 0, the successor function $(+1)$ applied to a temporal term, or the predecessor function (-1) applied to a temporal term.

Note that we will usually write $\tau + c$ ($\tau - c$, respectively) as a shorthand for the temporal term obtained by c applications of the successor function (predecessor function, respectively) to τ.

We distinguish between intensional and extensional predicate symbols, which are used to construct respectively intensional and extensional atoms. Intensional atoms are the only ones that can appear both in the head and in the body of clauses. Extensional atoms appear only in the body of clauses. There is a third type of atomic formulas that can appear in the body of clauses, namely those constructed with the interpreted relational symbols $=$ and $<$ applied to temporal terms.

Definitions (Atom, Clause, Program)

- An *atom* is an intensional atom, an extensional atom, or a constraint atom.
- An *intensional atom* (*extensional atom*, resp.) is a formula of the form $p(\tau_1, \ldots, \tau_m, d_1, \ldots, d_\ell)$ where p is an intensional (extensional, resp.) predicate symbol, $\tau_1, \ldots, \tau_m$ are temporal terms, and $d_1, \ldots, d_\ell$ are data terms.
- A *constraint atom* is a formula of the form $\tau_1 = \tau_2$ or $\tau_1 < \tau_2$ where τ_1 and τ_2 are temporal terms. Notice, however, that atomic constraints can always be reduced to constraints of one of the following forms: $t_1 < t_2 + c$, $t_1 < t_2 - c$, $t_1 = t_2 + c$, $t_1 = t_2 - c$, $t < c$, $t = c$, $c < t$, where t, t_1, and t_2 are temporal variables and c is an integer.
- A *clause* of the deductive language is a formula of the form
$$A \leftarrow A_1, \ldots, A_r$$
where A is an intensional atom, and $A_1, \ldots, A_r$ are atoms (intensional, extensional, or constraint).
- A *program* is a finite set of clauses.

Notice that extensional relations are not defined in the deductive language, but rather are provided by generalized database relations. In other words, an extensional relation consists of a finite set of generalized tuples, which may correspond to an infinite set of ground tuples (see Section 2.1).

Example 4.1 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 $(t_1, t_2, database)$ 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*.

$$\begin{aligned} &problems(t_1 + 2, t_2 + 2, database) \\ &\quad \leftarrow \ course(t_1, t_2, database) \\ &problems(t_1 + 48, t_2 + 48, database) \\ &\quad \leftarrow \ problems(t_1, t_2, database) \end{aligned}$$

■

4.2 Semantics

The semantics of our deductive language is given with respect to two-sorted domains. Indeed, the temporal terms are interpreted over the set of integers, whereas the data terms are interpreted over a set of constants. Such interpretations have already been used in [CI88, JL87], for instance. It is possible to show that the declarative semantics of classical logic programs [vEK76] extend to the case of programs in our deductive language [JL87]. The declarative semantics of a deductive program considered together with an extensional database is captured by its minimal Herbrand model, which can be obtained by iterating a mapping operating over Herbrand interpretations.

A term or an atom is said to be *ground* if it is variable free. A ground temporal term is thus an integer constant c. A Herbrand interpretation is a set of ground atoms; in other words, for each extensional or intensional predicate symbol, it provides an extension, that is, a set of ground tuples for which the predicate is true. Let P denote a deductive program in our language. The semantic mapping T_P, operating over Herbrand interpretations, that is associated with P is defined in the following way.

Definition (Mapping T_P) Let H denote a Herbrand interpretation. Then $T_P(H)$ is the set of ground intensional atoms A such that $A \leftarrow A_1, \ldots, A_r$ is a ground instance of a clause of P, and for each ground atom A_i $(1 \leq i \leq r)$,

- either A_i is a ground intensional or extensional atom, in which case it must appear in H,
- or A_i is a ground constraint, in which case it must be true.

Given an extensional database EDB, the minimal model of a deductive program P considered in conjunction with EDB is obtained as the least fixpoint of the mapping $T_P + I$, where I is the identity function. We denote this minimal model by $M_{(P,EDB)}$.

$$M_{(P,EDB)} = \bigcup_{j=0}^{\infty} (T_P + I)^j(EDB)$$

where $(T_P + I)^0(EDB) = EBD$, and

$$\begin{aligned}
&(T_P + I)^{j+1}(EDB) \\
&\quad = \ (T_P + I)\big((T_P + I)^j(EDB)\big) \\
&\quad = \ T_P\big((T_P + I)^j(EDB)\big) \cup \big((T_P + I)^j(EDB)\big).
\end{aligned}$$

Notice that when EDB is a generalized database, it provides, for each extensional predicate, an extension in the form of a finite set of generalized tuples. Each such generalized tuple represents a possibly infinite set of ground tuples. More precisely, if the generalized tuple

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

appears in the extension of a predicate q in EDB, then this corresponds to having in EDB the possibly infinite set of ground atoms

$$\begin{aligned}
&\{q(t_1, \ldots, t_m, d_1, \ldots, d_\ell) \mid \\
&\qquad t_1 \in \{a_1 n_1 + b_1\}, \ldots, t_m \in \{a_m n_m + b_m\}, \\
&\qquad \text{and } \mathit{constraints}(t_1, \ldots, t_m) \text{ is satisfied}\}.
\end{aligned}$$

Applying T_P to such a generalized database EDB thus boils down to applying it (*one at a time*) to the possibly infinite set of ground extensional atoms that EDB represents.

4.3 Evaluating Predicates

As we have just seen, in our deductive language, the straightforward bottom-up evaluation of predicates by iterations of the mapping T_P is problematic since it corresponds to computing on a *tuple-at-a-time* basis on predicates with possibly infinite extensions. Moreover, these infinite extensions are not limited to being periodic as in the case of a unique temporal parameter [CI88]. For instance, our language allows the definition of the relation (i, i^2), with $i \in \mathcal{Z}$ (more on the expressiveness of this language in Section 4.4). So, bottom-up evaluation of such predicates might seem pretty hopeless. However, the situation can be very different if one operates directly on infinite periodic extensions as illustrated below.

Example 4.1 (continued) Let us consider the naive bottom-up evaluation of the predicate *problems*. 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,

$$\begin{aligned}
&(168n_1 + 346,\ 168n_2 + 348) \\
&\quad = \ \big(168(n_1 + 2) + 10,\ 168(n_2 + 2) + 12\big)
\end{aligned}$$

is a set of tuples of integers contained in a previously obtained set of tuples. The intuitive reason for which the computation terminates is that it starts with an infinite periodic set and can be seen as a computation in modulo-arithmetic, hence on a finite domain. ∎

As Example 4.1 illustrates, when the extensional relations are infinite and periodic, we can proceed to evaluate the predicates of our deductive language bottom-up, representing the successive extensions of each predicate by a generalized relation as in [KSW90]. This corresponds to computing on generalized tuples, which represent infinite periodic sets, rather than computing a finite number of tuples at a time, so every iteration may bring in an infinite number of tuples. This presents no particular problem using the operations on generalized relations defined in [KSW90]. Indeed, the intersection, the join, and the projection operations on generalized relations can be computed in PTIME (see [KSW90]), and applying the operation $+1$ (or -1) to a generalized relation is straightforward. We now define more precisely the evaluation procedure on generalized tuples.

Generalized Programs

We adopt a normalized form for extensional databases and we rewrite our deductive programs so that deductive rules can operate directly on generalized tuples. This requires a number of definitions.

Definitions

- The notion of *generalized tuple* is defined exactly as the notion of *ground generalized tuple* (see Section 2.1) except that temporal and data arguments may respectively be non-ground temporal or data terms.
- A *generalized intensional atom* (*generalized extensional atom*, resp.) is the result of applying an intensional (extensional, resp.) predicate to a generalized tuple. If the generalized tuple is ground, then the generalized atom is also said to be *ground*.
- A *generalized atom* is an intensional or an extensional generalized atom.
- A *generalized Herbrand interpretation* is a set of ground generalized atoms.

A generalized atom is thus a finite representation for a possibly infinite set of ground atoms, and a generalized Herbrand interpretation that is finite may actually represent an infinite Herbrand interpretation.

We make a few simplifying (but not restrictive) assumptions on the form of the programs and of the extensional database. First, we eliminate all integer constants from the programs. Indeed, we can replace every integer constant c in the ith position of a generalized tuple by the lrp n with associated constraint $T_i = c$. Any constraint atom in the deductive program can be seen as a generalized atom. For instance, the constraint $t_1 < t_2 + c$ can be seen as a special predicate symbol *constraint* applied to the generalized tuple (n_1, n_2) with $T_1 < T_2 + c$. So a deductive program can be transformed into an equivalent program, called a *generalized program*, which is a set of clauses constructed with generalized atoms. Moreover, an extensional database is a finite set of ground generalized atoms, that is, of ground generalized facts. This leads us to another view of the bottom-up iterations of our deductive programs.

Generalized Mapping

We associate with the generalized version GP of a deductive program P a mapping T_{GP} operating on generalized Herbrand interpretations. This mapping will serve as a basis for generalized-tuples-at-a-time computations.

One additional precaution has to be taken. The generalized clauses must be transformed in such a way that their heads are generalized atoms with all their temporal parameters being distinct temporal variables. This transformation may introduce additional constraints in the body of the clauses, but simplifies the evaluation.

Definition (Mapping T_{GP}) Let GH denote a generalized Herbrand interpretation. Then $T_{GP}(GH)$ is the set of ground generalized intensional atoms GA such that

- there exist a clause $A \leftarrow A_1, \ldots, A_r$ in P and ground generalized instances $GA_1, \ldots, GA_r$ respectively of $A_1, \ldots, A_r$ in GH, and
- GA is a ground generalized atom obtained by computing the join of $GA_1, \ldots, GA_r$, and projecting the result over the variables of A.

It is easy to see that computing with T_{GP} yields the same result as computing with T_P.

Lemma 4.1 *Let GH be a generalized Herbrand interpretation and let $extension(GH)$ be the corresponding Herbrand interpretation. We then have*

$$extension(T_{GP}(GH)) = T_P(extension(GH))$$

We can thus legitimately compute with T_{GP} on the ground generalized tuple representation of the extensional database EDB. The problem is to determine when this computation will terminate. It will terminate in many cases where the computation with T_P on the ground tuples is impossible (because the extension is infinite) or infinite, but it will not always terminate. We now establish conditions under which it does terminate. First some definitions.

Definition (Free Extension) The *free extension* of a ground generalized tuple

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

is the ground generalized tuple freed from its constraints (i.e., with constraint *true*), namely

$$(a_1 n_1 + b_1, \ldots, a_m n_m + b_m, d_1, \ldots, d_\ell) \\ \text{with } true.$$

The constraint *true* is usually simply omitted.

Definition (Free-Extension Safety) Let GH be a generalized Herbrand interpretation and let $free(GH)$ be its free extension. GH is *free-extension safe* for a program P if

$$T_{GP}(free(GH)) \subseteq free(GH).$$

A generalized Herbrand interpretation is thus free-extension safe if applying the mapping T_{GP} to this interpretation generates no ground generalized tuples with new free extensions (new tuples can be generated, but they will only differ from tuples in GP by their constraints). The interesting property is that when applying a mapping T_{GP} to an extensional database of generalized tuples, we eventually reach a generalized Herbrand interpretation that is free-extension safe.

Theorem 4.2 *Let EDB be an extensional database of ground generalized tuples, let T_{GP} be the generalized mapping associated with a deductive program P, and I the identity mapping. Then, there exists a k such that*

$$(T_{GP} + I)^k(EDB)$$

is free-extension safe.

Proof sketch: The theorem follows simply from the fact that there is a finite bound on the number of possible free extensions. Indeed, let $p = \Pi p_i$ be the product of the periods of the lrp's in EDB. Then, all lrp's appearing in the computation of $(T_{GP} + I)^j(EDB)$ are of period less than p and hence there is only a finite number of such lrp's. ■

Once a generalized Herbrand interpretation is free-extension safe for the mapping T_{GP}, applying T_{GP} can still lead to the modification of constraints. It is only when the new constraints are implied by existing ones that the evaluation can be stopped. We use the following definition.

Definition (Constraint Safety) Let GH be a generalized Herbrand interpretation and, for each ground generalized tuple gt, let $constraints(gt)$ be the constraints of that tuple. Then, GH is *constraint safe* for a program P if for every ground generalized tuple $gt' \in T_{GP}(GH)$ there are generalized tuples $gt_1, \ldots gt_n \in GH$ with the same free extension as gt' such that

$$\begin{aligned}&constraints(gt')\\ &\quad \Rightarrow \quad constraints(gt_1) \vee \ldots \vee constraints(gt_n).\end{aligned}$$

We can now give a sufficient criterion for the termination of the naive bottom-up generalized-tuple-at-a-time evaluation of a program P.

Theorem 4.3 *Let EDB be a generalized extensional database and let P be a program. Then, if for some k, $(T_{GP} + I)^k(EDB)$ is both free-extension safe and constraint safe, then the naive generalized-tuple-at-a-time bottom-up evaluation of P on EDB terminates after k iterations.*

Of course, Theorem 4.3 does not completely solve the problem since we might never reach a generalized Herbrand interpretation that is constraint safe. In practice, once the generalized Herbrand interpretation is free-extension safe (which is guaranteed to happen by Theorem 4.2), it is reasonable to give up on the computation if the interpretation does not become constraint safe after a few iterations.

4.4 Expressiveness

As in Section 3, we distinguish data and query expressiveness. As far as data expressiveness, our language is very powerful. Indeed, it is easy to show that it can express at least all the primitive recursive relations. On the other hand, notice that if the language is only used when the conditions of Theorem 4.3 are satisfied, its data expressiveness is the same as that of generalized databases with linear repeating points. This is of course the price to pay for being able to obtain a closed form for derived predicates. Indeed, one cannot expect a closed form for all primitive recursive relations that is much else than an algorithmic definition of the predicate.

Concerning query expressiveness, the situation is quite different. Indeed, we have already shown in Section 3 that in the case of a unique temporal argument, this type of deductive language can define queries that are not first-order definable. This result can be extended to the case of several temporal variables. The interesting point is that the increase in query expressiveness is meaningful even in cases where the conditions of Theorem 4.3 are satisfied.

5 Conclusions and Comparison with Other Work

Our contributions are

1. the clarification of the necessary concepts for comparing the expressiveness of databases in which predicates with infinite extensions can appear;
2. a closed form evaluation algorithm for a class of Datalog programs over the integers.

Our first result is not limited in scope to temporal databases. The same concepts are useful whenever databases can include predicates with infinite extensions, for instance as in [KKR90]. Our second result can be interpreted as saying that you can both have your cake and eat it. Indeed, it shows that you can have temporal databases with good data and query expressiveness and finite bottom-up evaluation. The only catch is that the bottom-up evaluation is not always possible. Nevertheless, we feel that the combination of an extensional database defined by extended relations and a deductive layer using predicates with multiple temporal variables is an interesting one.

It was already noticed in [CI88] that evaluating least fixpoints on infinite extensions could be easier than the same problem on finite extensions. However, this observation was limited to the case of deductive programs with one temporal argument and linear repeating points were not used as a representation formalism. Closed forms for Datalog over the integers naturally brings to mind [Rev90]. There are notable differences. In [Rev90], the closed form is not limited to some extensional databases and programs, but is only obtained for a restricted language. Indeed, this language does not allow the use of incrementation over recursion, and thus cannot express periodic sets of points. It can be made as expressive as (actually more expressive than) our language by the use of stratified negation. However, the closed form result is not obtained in this case.

References

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

[Bau89a] Marianne Baudinet. *Logic Programming Semantics: Techniques and Applications*. PhD thesis, Computer Science Department, Stanford University, Stanford, CA, February 1989.

[Bau89b] Marianne Baudinet. Temporal logic programming is complete and expressive. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 267–280, Austin, Texas, January 1989.

[Bau90] Marianne Baudinet. On the expressiveness of temporal logic programming. Submitted to a Technical Journal, July 1990.

[CH82] Ashok K. Chandra and David Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:98–128, 1982.

[CH85] Ashok K. Chandra and David Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[Cho90] Jan Chomicki. Polynomial time query processing in temporal deductive databases. In *Ninth ACM Symposium on Principles of Database Systems*, pages 379–391, Nashville, Tennessee, April 1990.

[CI88] Jan Chomicki and Tomasz Imieliński. Temporal deductive databases and infinite objects. In *Seventh ACM Symposium on Principles of Database Systems*, pages 61–73, Austin, Texas, March 1988.

[CI89] Jan Chomicki and Tomasz Imieliński. Relational specifications of infinite query answers. In *ACM-SIGMOD International Conference on Management of Data*, pages 174–183, Portland, Oregon, May 1989.

[CI90] Jan Chomicki and Tomasz Imieliński. Finite representation of infinite query answers. Technical Report TR-CS-90-10, Kansas State University, Manhattan, KS, August 1990.

[GPSS80] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Seventh ACM Symposium on Principles of Programming Languages*, pages 163–173, Las Vegas, NV, January 1980.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Fourteenth ACM Symposium on Principles of Programming Languages*, Munich, West Germany, January 1987.

[KKR90] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Revesz. Constraint query languages. In *Ninth ACM Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, April 1990.

[KSW90] F. Kabanza, J-M. Stevenne, and P. Wolper. Handling infinite temporal data. In *Ninth ACM Symposium on Principles of Database Systems*, pages 392–403, Nashville, Tennessee, April 1990.

[Rev90] Peter Revesz. A closed form for Datalog queries with integer order. In *Proceedings* *of the Third International Conference on Database Theory*, pages 187–201, Paris, December 1990. LNCS 470, Springer-Verlag.

[Tho81] Wolfgang Thomas. A combinatorial approach to the theory of ω-automata. *Information and Control*, 48(3):261–283, March 1981.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems – Volume I.* Computer Science Press, 1988.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems – Volume II: The New Technologies.* Computer Science Press, 1989.

[Var88] Moshe Y. Vardi. A temporal fixpoint calculus. In *Fifteenth ACM Symposium on Principles of Programming Languages*, pages 250–259, San Diego, CA, January 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, October 1976.

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *24th Symposium on Foundations of Computer Science*, pages 185–194, Tucson, Arizona, November 1983.