Towards Temporal Reasoning in Constraint Contextual Logic Programming

Vitor Beires Nogueira[1], Salvador Abreu[1], and Gabriel David[2]

[1] Departamento de Informática, Universidade de Évora, Portugal
{vbn,spa}@di.uevora.pt
[2] Departamento de Engenharia Electrotécnica e de Computadores, Universidade do Porto, Portugal
gtd@fe.up.pt

Abstract There has been an increased interest in temporal reasoning from areas such as database systems and AI. In this paper we propose a temporal reasoning framework that bridges the paradigms of Constraint Logic Programming (CLP) and Contextual Logic Programming (CxLP). CLP is used to build a rather simple, yet powerful temporal algebra with points, intervals, durations and relations between these elements. To associate the time given by the algebraic system above and events or propositions, we build a framework that makes use of recent developments in CxLP and propose the notion of temporal contexts.

1 Introduction

Temporal representation and reasoning is a central part of many Artificial Intelligence areas such as planning, scheduling and natural language understanding. Also in the database community, the relevance of the temporal issue is widely acknowledged. Example of such relevance is the second–generation temporal query language TSQL2 [1]. Constraint–based frameworks are widely used to perform temporal reasoning [2] and there is even a temporal specialisation of the Constraint Satisfaction Problems. Constraint Logic Programming(CLP) in particular, has proven to be very suitable for such kind of reasoning [3,4].

The benefits of Logic Programming are well known: the rapid prototyping ability and the relative simplicity of program development and maintenance, just to name a few. Nevertheless, the Prolog language suffers from a serious scalability issue when addressing actual applications. There have been several efforts over the years to overcome this limitation. An interesting solution to this problem is that of Contextual Logic Programming (CxLP), a model introduced in the late 1980's [5]. Informally, the main point of CxLP is that programs are structured as sets of predicates, *units*, that can be dynamically combined in an execution attribute called a *context*. Recent developments [6] extended CxLP to attach *arguments* to units: these serve the dual purpose of acting as "unit-global" variables and as state placeholders in actual contexts.

In this article, we joined CLP and CxLP to build a temporal reasoning framework.

This paper is organised as follows. In Sects. 2 and 3 we briefly discuss some temporal constraint–based frameworks and the Contextual Logic Programming paradigm, respectively. Section 4 presents the formalisation and implementation of the temporal algebra adopted in our framework. The temporal reasoning system is detailed in Sect. 5. Conclusions and proposals for future work follow.

2 Modular Logic Programming

A modular extension to logic programming has been the subject of research during the last decades. Two different approaches have guided the research: one deals with what is called programming–in–the–large and another with programming–in–the–small. The former began with work by O'Keefe [7], and considers logic programs as elements of an algebra, whose operators are the operators for composing programs. Moreover, there was no need to extend the language of Horn clauses, since composition was a meta–linguistic mechanism. The programming–in–the–small approach is based on a work of Miller [8], and extends the Horn language with logic connectives for building and composing modules.

For a comprehensive reading related to both approaches and their comparison see for instance [9]. In the next subsection we shall present a brief overview of Contextual Logic Programming, a language that fits in the latter approach and is the basis for our temporal framework.

2.1 Overview of CxLP

Contextual Logic Programming [5] is a simple and powerful language that extends logic programming with mechanisms for modularisation.

In CxLP a finite set of Horn clauses with a given name is designated by *unit*. Recent work [6] presented a revised specification of CxLP, together with a new implementation for it, **GNU Prolog/CX**. Using the syntax of this implementation, consider a unit named `teacher` to represent some basic facts about the teachers of an University:

Example 1 (Unit Teacher).

```
:-unit(teacher).

teacher(john, computerScience, phd).
teacher(bill, computerScience, msc).

name(Name):-teacher(Name,  _, _).
department(Dep):-teacher(_, Dep, _).
degree(Deg):-teacher(_, _, Deg).
```

The only difference from a regular logic program is the first line that declares the unit name.

Consider also another unit to represent information about classes:

Example 2 (Unit Class).

```
:-unit(class).

class(ai, 6, 4).
class(logic, 6, 3.5).

name(Name):- class(Name, _, _).
ects(Ects):- class(_, Ects, _).
hours(Hours):- class(_, _, Hours).
```

A set of units is designated by a *contextual logic program*. With the units above we can build a program $P = \{teacher, class\}$.

Given a CxLP program, we can combine its units into sequences, leading to the notion of *context*. To build such sequences there is a special predicate $u :> G$ that extends the current context with unit u and resolves goal G in the new context. For example, the goal `"?- teacher :> class :> degree(D)."` starts by extending the initially empty (`[]`) context with unit `teacher`, obtaining context `[teacher]`. This context is extended with unit `class` obtaining the context `[class teacher]`, and it is in this last context that goal `degree(D)` is derived.

To derive an atomic goal G in a context $u_1 u_2 \dots u_n$ a search for the smallest $1 \leq i \leq n$, such that G can be derived with a clause of u_i, is made. The derivation of the body of that clause is considered in the reduced context $u_i \dots u_n$. In the example above a search is made for the first unit in the context `[class teacher]` where the goal `degree(D)` can be derived (unit `teacher` since unit `class` has no clauses for `degree/1`). Moreover, the body of the clause (`teacher(_, _, Deg)`) is derived in the reduced context `[teacher]`, and we obtain through backtracking `phd` and `msc`.

Units with Arguments. In [6] there is claim that *units arguments* are an essential addition to this programming model. A unit argument can be interpreted as a sort of "unit global" variable, i.e., which is shared by all clauses defined in the unit. This way, unit arguments help solving the annoying proliferation of predicate arguments whenever a global structure is to be passed around, but also give the context a more transparent form, allowing to be explicitly manipulated in the course of a program's regular computation.

As an illustration, for the unit `teacher` above we could have:

Example 3 (Unit teacher with arguments).

```
:-unit(teacher(NAME, DEPARTMENT, DEGREE))

name(NAME).
department(DEPARTMENT).
degree(DEGREE).
```

```
teacher(john, computerScience, phd).
teacher(bill, computerScience, msc).

item:- teacher(NAME, DEPARTMENT, DEGREE).
```

In this modified version, we have three unit arguments: `NAME`, `DEPARTMENT` and `DEGREE`. Facts for `teacher/3` remain equal, but predicates `name/1`, `department/1` and `degree/1` are simplified (they just access the corresponding arguments). A new predicate, `item/0`, that instantiates all the unit arguments using facts from the database, is added.

As mentioned above, units arguments can also be used to give contexts a more transparent form. For instance, to obtain Bill's department we can invoke the goal `"?- teacher(bill, D, _) :> item."` and variable `D` would be instantiated with `computerScience`.

3 Temporal Constraint Formalism

Constraint–based frameworks are widely used to perform temporal reasoning. There is even a temporal specialisation of the Constraint Satisfaction Problems [3], where variables represent time and constraints stand for sets of allowed temporal relations between the variables. Different types of variables such as time points, time intervals or durations define different temporal constraints frameworks.

In the following subsections we are going to summarise the most relevant aspects of the frameworks that inspired our proposal (for a general overview see for instance [2]).

3.1 Point Algebra

The Point Algebra was introduced by Vilain and Kautz [10]. In their proposal, the domain elements are the temporal points and define the three basic relations that can hold between temporal points, i.e., *before, equal* and *after*. Moreover, the Point Algebra defines two operations between point–point relations: *composition* and *intersection*.

3.2 Interval Algebra

The Interval Algebra was proposed by James F. Allen [11]. In his work, domain elements are temporal intervals, and constraints are built using the thirteen basic relations between intervals: *before, after, meets, met by, equal, overlaps, overlapped by, during, contained by, starts, started by, finishes, finished by.*

[3] A Constraint Satisfaction Problem is basically a tuple $< V, D, C >$, where V is the set of variables, D their domains and C the set of constraints to be satisfied.

4 Temporal Algebra

4.1 Formalisation

In this section we formalise the temporal algebra that is the basis of our framework. On one hand, we wanted our algebra to be capable of dealing with a large spectrum of temporal problems (common database applications, circuit analysis, etc.), and on the other hand, to be simple and with an efficient implementation.

Temporal Systems. With the goals above in mind, we defined a temporal system to be just a Cartesian product of finite subsets of $\mathbb{Z}^*$ [4], i.e.,

Definition 1 (Temporal System). *TS is a temporal system* $\Leftrightarrow$
$\exists\, k_1, \ldots, k_n \in \mathbb{N}\, \exists\, TU_1, \ldots, TU_n \subset \mathbb{Z}^* : TS = TU_1 \times \ldots \times TU_n$ *and* $|TU_i| = k_i$, $\forall i, 1 \leq i \leq n$.

To be more intuitive and comply with the standard for dates and time [12], in the Definition 1 we assume that TU_j stands for a greater temporal unit than TU_{j+1}, $\forall j, 1 \leq j \leq n-1$.

Example 4 (24–Hour). The *24–Hour* timekeeping system can be represented by $\textit{24–Hour} = Hour \times Minute \times Second$, where $Hour = \{0, \ldots, 23\}$ and $Minute = Second = \{0, \ldots, 59\}$.

Example 5 (Gregorian Calendar). The commonly used Gregorian calendar can be seen as $Gregorian = Year \times Month \times Day$, where $Day = \{1, \ldots, 31\}$, $Month = \{1, \ldots, 12\}$ and, assuming the standard four digit notation of year, $Year = \{0, \ldots, 9999\}$ [5].

However, not all elements of the Cartesian product above are valid dates: (2003,6,31) is an example of such. Therefore, a temporal system can have a set of constraints that a tuple must satisfy in order to be a valid element of the temporal system. As it should be clear, these constraints are defined in the finite domain. We leave the mathematical formalisation of this constraint domain for the interested reader [13] and present an illustrative example of these constraints for the case of the Gregorian calendar:

Example 5 (cont.) A tuple $(y, m, d) \in Gregorian$ is valid if it satisfies the following constraints:
$(m = 4 \vee m = 6 \vee m = 9 \vee m = 11) \Rightarrow d \leq 30 \wedge$
$(\neg(y \backslash 400 = 0 \vee (y \backslash 4 = 0 \wedge y \backslash 100 > 0))) \Rightarrow max\_feb = 28 \wedge$
$max\_feb \in [28, 29] \wedge m = 2 \Rightarrow d \leq max\_feb.$

Of course that the Gregorian calendar and the 24–Hour timekeeping system can be trivially concatenated into a larger one, that expresses both date and time.

[4] The nonnegative integers 0, 1, 2, ...
[5] For clarity reasons, we simplified the Gregorian calendar, allowing not only year 0 but also to represent dates after October 5, 1582.

Time Points. It is obvious that not all the applications need the full granularity/precision available in the temporal system. For instance, an application that deals with class schedule information just needs minutes precision, whereas in a stock exchange application the seconds precision might be crucial. Therefore, we decided that the basic elements of our algebra be tuples of varying size, that satisfy the constraints of the temporal system:

Definition 2 (Time Points). *Given a temporal system* $TS = TU_1 \times \ldots \times TU_n$, *a tuple* $x = (x_1, \ldots, x_i) \in TU_1 \times \ldots TU_i$ *is a time point with granularity* $i \Leftrightarrow x$ *is consistent with the constraints of* TS.

Moreover, if we don't specify the time point granularity, the maximum granularity of the temporal system is assumed. With the temporal system of Example 5, we represent the year 2004 with the tuple (2004) and for the "2^{nd} of January of 2003" we use $(2003, 1, 2)$.

Durations. To add the concept of duration to the temporal system presented, no changes have to be done, since our time points can be considered as durations. For instance, the tuple $(2003, 1, 1)$ could be interpreted as 2003 years, 1 month and 1 day (in this case there is no need for constraints).

Basic relations. In our temporal algebra we have two basic relations for time points (*equal* and *before*), and each one of these relations has a strong and a weak version. To see if a pair of time points belongs to one of these relations we check the satisfiability of a constraint:

Definition 3 (Strong Equality). *Let* $x = (x_1, x_2, \ldots, x_n)$ *and* $y = (y_1, y_2, \ldots, y_m)$ *be two time points* $x =_s y \Leftrightarrow m = n \wedge \ x_i = y_i, \forall i, 1 \leq i \leq n$.

Definition 4 (Weak Equality). *Let* $x = (x_1, x_2, \ldots, x_n)$ *and* $y = (y_1, y_2, \ldots, y_m)$ *be two time points and* $p = min(n, m)$, $x =_w y \Leftrightarrow x_i = y_i, \ \forall i, 1 \leq i \leq p$.

According to the definitions above, $(2004, 1, 10) \neq_s (2004, 1)$ and $(2004, 1, 10) =_w (2004, 1)$.

Definition 5 (Strong Before). *Let* $x = (x_1, x_2, \ldots, x_n)$ *and* $y = (y_1, y_2, \ldots, y_m)$ *be two time points,* $x <_s y \Leftrightarrow n = m \ \wedge \exists i, \ 1 \leq i \leq n : \forall j, \ j \leq i, x_j < y_j$.

Definition 6 (Weak Before). *Let* $x = (x_1, x_2, \ldots, x_n)$ *and* $y = (y_1, y_2, \ldots, y_m)$ *be two time points,* $x <_w y \Leftrightarrow \exists i \leq min(m, n) \forall j, 1 \leq j < i : x_j = y_j \wedge x_i < y_i$.

Although the constraint for the weak before may seem tricky, it stands for something quite simple: the time point x is before y if there is a temporal subunit of x that is smaller than the same subunit of y and all the other more significative subunits of x and y are equal. Since there is no imposition for the tuples to be of the same size, and assuming the Gregorian calendar, $(2004, 1, 1)$ is before $(2004, 2)$.

Actually, since the weak definitions include the strong ones, from here on we assume the weak ones as the default and use the symbols $=$ and $<$ for them.

Other relations. As we already saw, each basic relation has one equivalent *FD* constraint. Therefore, one can easily build all the other common relations ($>$, $\geq, \leq, \neq$) simply using logical negation, logical connectives $\wedge, \vee$ and the already defined constraints.

Intervals. Continuing with the discrete approach, intervals are defined by two time points: *start* and *end* of the interval, and stand for the set of time points between those bounds. Therefore, our intervals are just a *special* kind of sets of time points.

Definition 7 (Closed Interval). *Suppose that TS is a temporal system and $i_{start}, i_{end} \in TS$, with $i_{start} < i_{end}$. Then $[i_{start}, i_{end}] = \{x \in TS : x \geq i_{start} \wedge x \leq i_{end}\}$.*

Definition 8 (Open Interval). *Suppose that TS is a temporal system and $i_{start}, i_{end} \in TS$, with $i_{start} < i_{end}$. Then $]i_{start}, i_{end}[= \{x \in TS : x > i_{start} \wedge x < i_{end}\}$.*

Although it is possible to keep both definitions, for simplicity, from here on we shall use least restrictive definition, closed intervals.

Interval relations. The thirteen basic relations between intervals (*before, after, meets, met by, equal, overlaps, overlapped by, during, contained by, starts, started by, finishes, finished by*) are again defined using some constraint manipulation. For instance, $Constraint(overlaps([i_{1start}, i_{1end}], [i_{2start}, i_{2end}])) = Constraint(i_{1start} < i_{2start}) \wedge Constraint(i_{2start} < i_{1end})$.

4.2 Temporal Algebra Implementation

The implementation of the algebra formalised in Sect. 4.1 is divided in two parts. In one part we handle the issues that are common to every temporal system, namely the evaluation of temporal expressions and the definition of intervals. In the other part, we specify some temporal systems together with their time points.

Since we already formalised the Gregorian calendar and the 24–Hour timekeeping system, we will just say that our framework has three units for them: unit `date` for the Gregorian calendar, unit `time` for the 24–Hour and unit `date_time` for the combination of the previous two. For an illustration of their the usage, consider the following example:

Example 6 (Weekends of January and February, 2004).

```
date(D) :>* (year(2004), month(M), M < 3,
             weekday(WD), member(WD, [sat, sun])).
```

In this example, we extend the initially empty context with the unit `date`, and argument variable `D`. The context extension used, *typed context extension* (`:>*`), is an improvement of the regular one that besides extending the context, checks the arguments type. In this case it stands for constraining variable `D` to be a valid date, i.e., a tuple of CLP(FD) variables `(YEAR, MONTH, DAY)` that are consistent with the constraints of the Gregorian calendar. A more formal definition of this context extension is given in Definition 10.

Afterwards, we constrain `YEAR` to be equal to 2004 and `MONTH` to be equal to 1 or 2, i.e., less than 3. Finally we restrict `DAY` to be a Saturday or Sunday, i.e., a weekend.

To deal with intervals there is a unit `interval` with one argument (the interval) and two access predicates: `start/1` and `end/1`. For instance, to represent the recurring interval between 9 AM of the 1^{st} day and 11 AM of the 5^{th} day of every month in 2004:

```
date_time(S) :>* (year(2004), month(M), day(1), hour(9)),
date_time(E) :>* (year(2004), month(M), day(5), hour(11)),
interval(I) :>* (start(S), end(E)).
```

Finally, there is a unit called `temporal_expressions` to evaluate temporal expressions. This unit accepts expressions from the following grammar:

Definition 9 (Grammar for temporal expressions). *Assuming the definitions for (time) points and intervals given above, the grammar for temporal expressions is:*

$$\begin{aligned}
\mathit{EXPRESSION} &\rightarrow \mathit{POINT\ POINT_RELATION\ POINT} \\
&\quad \mid \mathit{INTERVAL\ INT_RELATION\ INTERVAL} \\
&\quad \mid \mathit{POINT\ POINT_INT_RELATION\ INTERVAL} \\
\mathit{POINT_RELATION} &\rightarrow \equiv \mid \leq \\
\mathit{INT_RELATION} &\rightarrow \underline{before} \mid \underline{meets} \mid \underline{equal} \mid \underline{overlaps} \\
&\quad \mid \underline{during} \mid \underline{starts} \mid \underline{finishes} \\
\mathit{POINT_INT_RELATION} &\rightarrow \underline{member} \mid \underline{not_member}
\end{aligned}$$

Example 7 (Temporal expressions).

```
?- date(DT1) :>* (year(2004), month(1)),
   date(DT2) :>*  true,
   temporal_expressions( before(DT1,DT2) ) :>* true.
```

In this example, `DT2` stands for every possible date after January, 2004.

5 Temporal Reasoning System

It is in the temporal reasoning system that we join the paradigms of Constraint Logic Programming and Contextual Logic Programming. As we already saw in Subsect. 4.2, CLP(FD) is used for implementing the temporal algebra, allowing for instance to talk about the period of time between year 2000 and *now*. To

associate time with events and propositions, we rely upon CxLP and its recent developments. Since CxLP is used not only in the temporal aspects of our reasoning system, but runs through the entire framework, we decided to give an uniform approach to all units in CxLP. More specifically, a common interface to all units (with arguments) was created. To see what that interface is, consider the unit `teacher` of Example 3:

`:-unit(teacher(NAME, DEPARTMENT, DEGREE))`

For this declaration, we define the following set of predicates:

- one "access" predicate for each unit argument. In this case we have the predicates: `name/1`, `department/1` and `degree/1`.
- one predicate `item/0` that returns through backtracking all instances of the unit, by instantiating their arguments.
  The query `"?- teacher(Name,Dep,Deg) :> item."` would answer

```
Name = john
Dep = computerScience
Deg = phd;

Name = bill
Dep = computerScience
Deg = msc
```

- one predicate `args/0` that enforces each unit argument to the set of values allowed for it. For the unit given, that predicate could be:

  Example 3 (cont.)[Predicate `args/0`]

```
args:- member(DEPARTMENT, [math, computerScience]),
       member(DEGREE, [bsc, msc, phd]).
```

  The goal `"?- teacher(NAME, DEPARTMENT, DEGREE) :> args."` would constrain variables `DEPARTMENT` and `DEGREE` to be a valid University department and academic degree, respectively.
  Due to the importance of this predicate and to the fact that he is so recurrent, we defined a new context extension, *typed context extension*, that verifies if the unit arguments are valid. This new extension is denoted by `:>*`, can be easily stated:

  Definition 10 (Typed context extension).
  `U :>* G :- U :> (args, G).`

  with this operator, the goal above is simplified to
  `"?- teacher(NAME, DEPARTMENT, DEGREE) :>* true."`
- two predicates `insert/0` and `delete/0` for inserting and deleting tuples from the relation represented by the unit. For instance, the goal
  `?- teacher(_, _, _) :> (degree('BSc'), delete).`
  removes, through backtracking, all teachers that hold a BSc.

5.1 Temporal Units

Temporal units are those that represent the propositions whose truth may vary with time. The difference from these units and the regular ones is the fact that at least one unit argument is a temporal timestamp. According to the semantics given to the temporal tags it is quite trivial to establish a parallelism to temporal database notions of *valid time* (the time when this fact is true in the modeled real world), *transaction time* (time of storage of the fact in the database) or even bitemporal time [14]. This way, we have valid time, transaction time and bitemporal units. Moreover, following our approach to units (with arguments), we also have a set of predicates (`valid_time/1`, `transaction_time/1`) for "accessing" those temporal arguments.

Definition 11 (Temporal Unit). *A unit u with parameter variables p is a temporal unit $\Leftrightarrow$ at least one parameter is a temporal timestamp.*

Moreover, due to the fact that valid time is the most commonly used notion, from here on we are going to consider only that notion of time, and name it simply by *time*. Please notice that we don't loose expressiveness since everything is still applicable to transaction time, etc.

It is assumed, that if the unit is not temporal, then its time is infinite (neutral element of intersection of times).

As an example, consider that we want to represent that a given lecturer, John, teaches Logic every Monday from 9 to 11, during the even semester of 2003/2004. Assuming that there is a unit called `semester` that stores, for each academic year, the start and end of each semester, the intended fact can expressed by:

```
1    semester(S) :>* (year(2003), type(even), time(I)),
2    date_time(D) :>* (weekday(mon), hour(H), H >= 9, H <= 11),
3    temporal_expression(member(D, I)) :> * true,
4    class_schedule(CS) :>* (lecturer(john),
5                            course(logic), time(D), insert).
```

We start by defining variable `I` as the temporal interval corresponding to the even semester of 2003/2004 (1). We then specify variable `D` to be the recurring interval between 9AM and 11AM, of every Monday in the calendar (2). After this we restrict `D` to be within the interval `I` (3). Finally, we insert the intended fact in the `class_schedule` database (4-5).

Now if we ask when John teaches

`"?- classe_schedule(C) :>* (lecturer(john), time(D))."`

then variable `D` should contain (at least) all the the Mondays of the even semester of 2003/2004, between 9AM and 11AM.

Finally, as the reader might have noticed, unit `teacher` is also a temporal one, since the teacher academic degree along with the department can change over time.

5.2 Temporal Contexts

In the previous section we saw that units can be regarded as the temporal relations in Contextual Logic Programming. However, during a CxLP computation we don't have isolated units but contexts made of units (or empty).

Therefore, if a context contains temporal units, we can consider such a context has some kind of temporal information. But what is the *time* of a context ? To build the answer to that question, let us start by considering a context with just one temporal unit:

```
"?- semester(_) :>* (year(2003), type(even), :> C)."
```

In this example, a context operator to obtain the current context, `:> C`, was used. Roughly speaking, `C` can be regarded as the context of the even semester of 2003/2004. Therefore it should be natural to say that the time of context `C` is the interval [(2004,2,23), (2004,6,18)]. Indeed, the goal:

`"?- C :< time(I)."`

instantiates the variable `I` with the interval above.

Now consider that we extend the previous context in the following way:

```
"?- semester(_) :>* (year(2003), type(even),
                     agenda(_) :>* (person(john), :> D))."
```

Again, we can say that the final context is the context of John's appointments *and* the even semester, 2003/2004. But what should the time of this new context be? The goal:

```
"?- D :< time(I)."
```

would instantiate variable `I` with the time of John's appointments, ignoring everything about the semester. Therefore, and since the standard context extension is time–independent, we decided to create a *temporal extension*:

Definition 12 (Temporal extension). *The operation of temporal extension of the current context with unit `U`, before attempting to reduce goal `G` is denoted by `U :>/ U` and this operator is defined as if by the Prolog clause:*

```
1   U :>/ G :-
2               time(TC),
3               [U] :< time(TU),
4               temporal_algebra(intersection(TC, TU, TC_U)) :>* true,
5               U :> time(TC_U) :> G.
```

Succinctly, to obtain `U :>/ G` we start by finding the time of the current context (2) an the time of unit `U` (3), afterwards, we intersect those times (4), and use this intersection as the time for the context after extending it with `U`. Finally, we call goal `G` in the new context.

To keep the coherence, also the non–temporal units define predicate `time` that gives always the same answer: *infinite* (that is, the time of a non-temporal

tuple is infinite). This way, the temporal extension with a non–temporal unit maintains the time of the context, as it should be.

Now we can give a more formal definition of a temporal context:

Definition 13 (Temporal Context). *Given a* CxLP *context* γ *we say that* γ *is a temporal context* $\Leftrightarrow$ *all the extensions to obtain* γ *were* temporal extensions.

This way, we built the notion of implicit time, that is, the time given by tuples currently valid in the context.

Finally, and to illustrate this concept consider that we want to find who taught Logic during John's sabbatical leave::

```
"?- classes_schedule(_) :>/ (lecturer(Name), class(logic),
                             agenda(_) :>/ (lecturer(john),
                                            desc(sabbatical)))."
```

The reader should notice that there is no explicit mention of time, but rather an implicit temporal synchronisation between the tuples of unit `classes_schedule` and `agenda`. This way, we built the notion of *implicit time*, that is, the time given by tuples currently valid in the context.

6 Conclusions and Future Work

We defined a temporal algebra with time points, durations and intervals and provided an efficient implementation for it based on CLP(FD). To associate the time given by this algebraic system and events or propositions, we built a framework that makes use of recent developments in CxLP, proposing the notion of temporal units and contexts. We illustrated the framework with simple, yet comprehensive examples.

We intend to prove the soundness and completeness of the computation schema with respect to the intended model. There is ongoing work to enrich the temporal algebra with operators like *add*, *subtract*, ...

Since there is already some work [15] in adding a temporal dimension to a logic language ISCO [16], future work will take that extension further, by incorporating the temporal mechanism described in this paper. Finally, it is our goal to apply the outcoming temporal language to a real system like SIIUE [17].

References

1. Ilsoo Ahn, G.A., Batory, D., Clifford, J., Dyreson, C.E., Elmasri, R., Grandi, F., Jensen, C.S., Käfer, W., Kline, N., Kulkarni, K., Leung, T.Y.C., Lorentzos, N., Roddick, J.F., Segev, A., Soo, M.D., Sripada, S.M.: The TSQL2 Temporal Query Language. Kluwer Academic Publishers (1995)
2. Schwalb, E., Vila, L.: Temporal constraints: A survey. Constraints **3** (1998) 129–149

3. Frühwirth, T.: Annotated constraint logic programming applied to temporal reasoning. In Hermenegildo, M., Penjam, J., eds.: Programming Language Implementation and Logic Programming: 6th International Symposium (PLILP'94). Springer, Berlin, Heidelberg (1994) 230–243
4. Lamma, E., Milano, M., Mello, P.: Extending constraint logic programming for temporal reasoning. Annals of Mathematics and Artificial Intelligence **22** (1998) 139–158
5. Porto, A., Monteiro, L.: Contextual logic programming. In Levi, G., Martelli, M., eds.: Proceedings 6th Intl. Conference on Logic Programming, Lisbon, Portugal, 19–23 June 1989. The MIT Press, Cambridge, MA (1989) 284–299
6. Abreu, S., Diaz, D.: Objective: in minimum context. In: Proc. Nineteenth International Conference on Logic Programming. (2003)
7. O'Keefe, R.: Towards an algebra for constructing logic programs. In Cohen, I.J., Conery, J., eds.: Proceedings of IEEE Symposium on Logic Programming, IEEE Computer Society Press (1985) 152–160
8. Miller, D.: A Logical Analysis of Modules in Logic Programming. Journal of Logic Programming **6** (1989) 79–108
9. Bugliesi, M., Lamma, E., Mello, P.: Modularity in logic programming. Journal of Logic Programming **19/20** (1994) 443–502
10. Vilain, M., Kautz, H.: Constraint Propagation Algorithms for Temporal Reasoning. In: Proc. Fifth National Conference on Artificial Intelligence, Philadelphia, PA, USA (1986) 377–382
11. Allen, J.F.: Maintaining Knowledge about Temporal Intervals. Communications of the ACM **26** (1983) 832–843 ACM.
12. International Organization for Standardization: ISO 8601:2000. Data elements and interchange formats — Information interchange — Representation of dates and times. International Organization for Standardization, Geneva, Switzerland (2000)
13. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. Journal of Logic Programming **19/20** (1994) 503–581
14. Clifford, J., Isakowitz, T.: On the semantics of (bi)temporal variable databases. In: Proceedings of the 4th international conference on extending database technology on Advances in database technology, Springer-Verlag New York, Inc. (1994) 215–230
15. Nogueira, V., Abreu, S., David, G.: Towards temporal reasoning on isco. In Juan José Moreno-Navarro, J.M.n.C., ed.: Proceedings of the Joint Conference on Declarative Programming APPIA-GULPE-PRODE, Madrid, Spain (2002) 311–324
16. Abreu, S.: Isco: A practical language for heterogeneous information system construction. In: Proceedings of INAP'01, Tokyo, Japan, INAP (2001)
17. Abreu, S.: A Logic-based Information System. In Pontelli, E., Santos-Costa, V., eds.: 2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000). Volume 1753 of Lecture Notes in Computer Science., Boston, MA, USA, Springer-Verlag (2000) 141–153