Towards Temporal Reasoning in *ISCO*

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. . We propose a temporal ontology for a logic programming language geared towards the development and maintenance of organisational information systems, called *ISCO* [1]. The temporal representation follows the first–order approach, i.e. a temporal argument (point or interval) is added to *ISCO* classes. The syntax and the operational semantics of this temporal extension is presented, altogether with an illustrative time–related applicational example - scheduling meetings. We claim that this extension increases the expressive power of the language, by allowing to represent temporal information and to reason about it, and we compare it to a few other similar systems.

1 Introduction and Motivation

Temporal representation and reasoning is a central part of many Artificial Intelligence areas such as planning, scheduling and natural language understanding.

There have been two radically different approaches to a logical treatment of time:

- modal–logic type of approach, introduced by Arthur Prior under the name *Tense Logic* [2];
- first–order terms to denote times and define change by the difference in truth–value between propositions differing only in their temporal arguments;

Soon after the original *Tense Logic* was presented, its syntax was extended in various ways, and such extensions continue to exist.

Formalisms such as Allen's interval calculus [3], Vilain and Kautz's point algebra [4], allow us to do temporal representation and reasoning in a first–order way. Recently, the widely spread calculus–based query language SQL(-92) had a consensual temporal extension named TSQL2 [5].[3]

[3] Some of the constructs of TSQL2 were transfered to SQL3, leading to what was termed SQL3/Temporal. These proposals were accepted by ANSI and were forwarded to ISO.

Arguments run in favour of both sides, there is even work [6,7] that bridges those two approaches, but since first–order approaches have the advantage of being directly implementable and it is our goal to encompass time information in our first–order framework, it becomes obvious the approach that we follow.

ISCO (Information System COnstruction language) is already a state–of–the–art logical framework for constructing organisational information systems. Although this language has a simple *date/time* data type, all the temporal reasoning and representation has to be done by the programmer in an *ad–hoc* way. *ISCO* is compiled into a GNU Prolog [8] executable with the native–code executable version of all *ISCO* predicates. Since GNU Prolog has a finite domain constraint solver, *CLP(FD)*, and it is widely accepted that a language based on constraint propagation can be a suitable tool for expressing and reasoning about temporal information [9], we propose to extend *ISCO* a few steps further, to encompass explicit/implicit time information, by taking advantage of Constraint Logic Programming.

This paper is organised as follows. In section 2 we briefly review the Constraint Logic Programming paradigm. Section 3 presents some notions of temporal constraint–based frameworks, that serve as basis for our proposal. *ISCO* language and concepts are detailed in section 4. The temporal extension, called *ISTO*, is formalised in section 5. Here, besides explaining the new syntactical constructions, we also give its semantic definition. An illustrative example is shown in section 6. Conclusions and proposals for future work follow.

2 On the use of *CLP(FD)*

Constraint Logic Programming (*CLP*) [10], can be considered as a generalisation of Logic Programming (*LP*), that replaces *LP* unification by a more powerful constraint solving mechanism. There is quite a variety of domains to which *CLP* scheme can be particularized such as Booleans, Pseudo–Booleans, Finite Domains, Reals, etc.

In this paper we are going to focus on the Finite Domains, *CLP(FD)*, i. e., the domain of the variables used to build the primitive constraints is a finite set of (ordered) values. Solvers for this domain, instead of guaranteeing that the set of constraints is consistent (since there is no efficient way of doing that), reduce the domain of the variables using a mechanism called *local propagation*. Therefore, solvers for *CLP(FD)* are *incomplete* and have a complementary *enumeration* procedure to obtain the possible solutions. These notions are made clear in the next example:

Example 1. Consider the constraints $X > Y$ and $Y > Z$, where X, Y and Z are variables in the finite domain [0..3], i.e., $D(X) = D(Y) = D(Z) = [0..3]$. From the constraint $X > Y$ and using local propagation, we get $D(X) = [1..3]$ and $D(Y) = [0..2]$. The constraint $Y > Z$ causes the narrowing of $D(Y)$ to [1..2] and $D(Z)$ to [0..1]. Since $D(Y)$ changed, by local propagation we get $D(X) = [2..3]$

$D(Y) = [1..2]$ and $D(Z) = [0..1]$. Using enumeration, we *could* get the solutions: $\{X = 3, Y = 2, Z = 1\}$, $\{X = 3, Y = 2, Z = 0\}$ and $\{X = 2, Y = 1, Z = 0\}$.

3 Temporal Constraints

Constraint–based frameworks are widely used to perform temporal reasoning. There is even a temporal specialisation of the Constraint Satisfaction Problems [4], 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 [11,12]).

3.1 Point Algebra

Point Algebra was introduced by Vilain and Kautz [4]. In their proposal, the domain elements of the constraint problems range over the set of (rational) real numbers and stand for temporal points, and constraints are disjunctions of the three basic relations between time points: *before, equal* and *after* (or, $<, =$, and $>$).

3.2 Interval Algebra

Interval Algebra was proposed by James F. Allen [3]. In his proposal, domain elements are temporal intervals, and constraints are built using the thirteen basic relations between intervals: *before, after, meets, meet by, equal, overlaps, overlapped by, during, contained by, starts, started by, finishes, finished by.*

4 *ISCO*: Language and Concepts

In this section we outline the aspects of the *ISCO* language needed for understanding the temporal extension proposed, namely predicates and goals. For a more in depth reading of the language and architecture see for instance [1].

An ISCO program is a regular Prolog program augmented with a syntax for describing classes, ISCO predicates and goals, sequences, integrity constraints and access control rules. That is, besides regular Prolog code (either clauses or directives), an *ISCO* program is a sequence of:

- A *class* definition, which is further discussed in Section 4.1.

[4] 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.

- A *data* definition, in which specific values for a class may be provided.
- A *sequence* definition, which can be used to implement global counters in an application.
- An *external* declaration which provides ISCO with the necessary information to access an external data source or sink, such as an ODBC-accessed database or an LDAP directory.

4.1 ISCO Classes

An ISCO predicate (class) is similar to a Prolog predicate. However, its actual implementation may differ because some predicates may rely upon an external storage mechanism such as that provided by an RDBMS, which provides persistency for facts and the ability to efficiently process some queries. Syntactically, ISCO predicates (classes) are different from regular predicates in that:

- Arguments are explicitly named and typed.
- There may be constraints that specific arguments are automatically required to satisfy.
- Classes are organised hierarchically, forming an inheritance relation so that whatever is true for a member of a given class, will also be true for any member of any of its subclasses.
- There's a built-in notion of update for ISCO predicates, which may take on several forms (see Section 4.2).

Formally, a class definition is made up of three parts: the *head*, the *argument list* and the *body*.

```
HEAD → CLASS_ATTRs class CLASS_NAME .
     | CLASS_ATTRs class CLASS_NAME : SUPERCLASS_NAME .
```

Fig. 1. ISCO class head syntax

Head A class is introduced by its HEAD (Figure 1). It may be tagged with an optional sequence of class attributes given by CLASS_ATTRs, which characterises the class being defined. This is a sequence of comma-separated terms enumerated in Figure 2.


```
CLASS_ATTR → static | mutable | computed | external ( ID , EXT_ID ) | ε
```

Fig. 2. ISCO class attributes

```
ARG_DEF → NAME : TYPE . ARG_ATTRS
        | NAME : CLASS_NAME . ARG_NAME . ARG_ATTRS
```

Fig. 3. ISCO argument syntax

Argument list The argument list in a class declaration given by Figure 3 is made up of a (possibly empty) sequence of argument definitions. Each argument declaration can come in one of the forms shown in Figure 3:

1. The first form is for regular arguments, which are explicitly typed at the time they are declared. Data types in ISCO can be simple types (integer, numbers, floating point numbers, booleans, text, date/time and serial) that map to the back-end types or compound types which constitute a re-use of *class* definitions as a data type.
2. The second form is for arguments which are implicitly typed by constraining them to take values only in the set of values specified by some argument in another class. In this case the type is inferred to be the same as that of the target argument.

The ARG_ATTRS symbol stands for a possibly empty sequence of Prolog terms. These describe attributes specific to the argument (such as not_null, unique, key, etc) which immediately precedes them and generally represent constraints that the argument in question must satisfy for a tuple to be acceptable for the class.

Body The computed classes are expected to contain one or more *rules*, which come after the body of the class. These rules are regular Prolog clauses, with the following differences:

- The clause head is always "`rule :-`", with no arguments. There may be more than one rule for any given class.
- The clause body may access the implicit head variables, which are named after the arguments in the class definition, rewritten to be all in upper-case in order to comply with the Prolog notation for variables.
- ISCO predicate calls are subject to the preprocessing discussed in 4.2, namely the non-positional argument syntax is automatically translated to regular Prolog calls.

Because their contents is fixed, static classes must have their declaration followed by their tuples. This form of declaration translates to Prolog "database" predicates, in which clauses have an empty body.

In Figure 4 we illustrate some of the notions described above, i.e. creation of class entity, with two arguments id and name, and class organizational_unit, sub-class of the previous one.

```
class entity.
  id: serial. key.
  name: text.

class organizational_unit: entity.
  parent: entity.id.
```

Fig. 4. Class creation example

4.2 Goal syntax

A program is made up of class and predicate definitions. Goals which occur in clause bodies can be categorized as regular Prolog goals or ISCO goals. The latter are intended to map to queries to the underlying database or other back-end engine and may be classified in the following categories:

- **Simple queries**. These correspond to interrogations and conform to the syntax:

$$\mathsf{NAME}_{rel}\ \underline{(}\ \mathsf{TUPLE}_{query}\ \underline{)}$$

Where NAME_{rel} is the name of an ISCO class. TUPLE_{query} is a constrained query-tuple (see [1]for details.) For example, the query:

```
20 #< P, P #< 30,
organizational_unit(parent=P, name=N)
```

This query is equivalent to the conjunction of constraints and goals:

```
20 #< P, P #< 30,
organizational_unit(_, N, P)
```

but does not require knowledge of the argument positions [5]. Its meaning could be "what are the names of the organizational units whose parent identifier lies in the interval 21..29?"
Simple queries behave like regular Prolog goals, in that their arguments are either bound, free or constrained on input, and become all-ground on completion of the query. Simple queries are non-deterministic and may therefore produce several solutions upon backtracking.
Arguments may be suffixed with an *ordering operator*, which indicates whether and how the corresponding argument is to be ordered when producing solutions.

[5] In fact, the non-positional query is compiled into the regular Prolog goal syntax

- **Insertion update queries**. Queries which add tuples to the relation use the syntax:

$$\mathsf{NAME}_{rel}\ \underline{:=}\ \underline{(}\ \mathsf{TUPLE}_{new}\ \underline{)}$$

Arguments which are omitted from the tuple are assigned the default value, should it exist. For example:

```
:- organizational_unit :=
     (name='Computing Services')
```

Insertion queries are deterministic and require all arguments to be either ground or omitted. It is an error to omit an argument for which there is no default.
For arguments of *evaluable types*, their value is evaluated before being inserted.

- **Modification update queries**. These queries replace existing tuples with new values. The syntax is:

$$\mathsf{NAME}_{rel}\ \underline{(}\ \mathsf{TUPLE}_{query}\ \underline{)}\ \underline{:=}\ \underline{(}\ \mathsf{TUPLE}_{new}\ \underline{)}$$

An update query includes two tuples, which are interpreted respectively as the *selection* and the *modified* values. For example:

```
organizational_unit(
  name='Academic Services') :=
   (parent=12)
```

would cause the `organizational_unit` whose name is `Academic Services` to change its parent unit identifier to 12.
Modification queries are non-deterministic and, as opposed to an SQL `update` query, alter tuples one-at-a-time. In order to change all tuples that satisfy the selection constraints, the modification update query must have its search space exhausted, ie. it must be made to backtrack over all solutions. This approach has a reading more consistent with the usual Prolog operational semantics and allows for certain useful programming dialects to be used.
Similarly to insertion update arguments, arguments of *evaluable types* occurring in the modified tuple are evaluated before being inserted.

- **Removal queries**. These queries remove existing tuples from a relation and may be expressed with the syntax:

$$\mathsf{NAME}_{rel}\ \underline{(}\ \mathsf{TUPLE}_{query}\ \underline{)}\ \underline{:\backslash}$$

For example, the query:

```
ID #> 1000, entity(id=ID) :\
```

Removes all tuples for the `entity` class (and all its subclasses) for which the `id` argument takes values greater than 10000.
Removal queries are non-deterministic in the same way as modification update queries: the tuples are deleted one-at-a-time.

5 Time in *ISCO*

As mentioned above, our proposal to incorporate temporal representation and reasoning in *ISCO*, follows the first–order approach, i. e. an extra temporal argument is added to our classes. This new argument can be either a time point or a temporal interval (described by their starting and ending points).

5.1 Temporal Elements

We consider that time points are elements located on a discrete time line, and that intervals, described by a start and ending point, are just a way to represent the set of all time points between (and including) those limits.

The time points used in our proposal instead of being simple terms (that are usually mapped to integers), are composed terms according to Figure 5, where Year, Month ... are constants or *CLP(FD)* variables.

TIME_PT → time_pt (Year, Month, Day, Hour, Minute, Second)

Fig. 5. Time point syntax

The reason for this has to due not only with the greater expressive power of such composed terms, but also for implementation efficiency reasons.

Considering that TP_1 and TP_2 are time points (see 5) of the form:

$$time\_pt(, Y_{(1/2)}, M_{(1/2)}, D_{(1/2)}, H_{(1/2)}, Mi_{(1/2)}, S_{(1/2)})$$

and using the symbols $\#=$, $\#\neq$ and $\#>$ to express the obvious constraints between *CLP(FD)* variables, we define the relations according to table 1.

The first line in the table can be read as time_pts TP_1 and TP_2 are equal if the year of the first one is (constraint) equal to the year of the second, ... In a similar way, time_pt TP_1 is strictly greater then time_pt TP_2 if the year of the first is (constraint) greater then the year of the second, or if the years are equal, then the month of TP_1 must be (constraint) greater then the month of TP_2 ....

$TP_1 = TP_2$	if $Y_1 \#= Y_2 \wedge M_1 \#= M_2 \wedge \ldots,$
$TP_1 > T2$	if $Y_1 \#> Y_2 \vee (Y_1 \#= Y_2 \wedge (M_1 \#> M_2 \vee (M_1 \#= M_2 \ldots)))$
$TP_1 \neq TP_2$	if $Y_1 \#\neq Y_2 \vee M_1 \#\neq M_2 \wedge \ldots,$

Table 1. Time–Point Relations

Using the relations just defined and logical connectives (such as $\wedge$, $\vee$) it is obvious to define the relations $\leq$ and $\geq$.

In what concerns intervals, and remembering that they represent the set of all elements between two points in a discrete time line, it is rather straightforward to define Allen's thirteen basic relations between intervals that we saw on 3.2. Consider for instance the relation of one interval being strictly contained in another:

$$[TP_1, TP_2] \textbf{ during } [TP_3, TP_4] \text{ if it is true that } TP_1 > TP_3 \wedge TP_2 < TP_4$$

5.2 Class definition

Along with the original *ISCO* classes mentioned in 4.1, a new type of classes, *Temporal* classes, will be available. Since there are two types of temporal elements, we extend the class definition with two new temporal attributes:

temporal(point): each class tuple has an corresponding temporal point that indicates when he is valid;
temporal(interval): in a similar way, each class tuple has a corresponding (discrete) temporal interval, that indicates when he is valid.

To illustrate the class definition, suppose that an organisation wants to keep track of the personal appointments of their employees. In a simplistic way, that can be done using the following class:

Example 2 (Personal Agenda).

```
temporal(interval) class personal_agenda.
    name: text
    appointment: text.
```

5.3 Goal syntax

In pursuing simplicity, the goal syntax is just:

$$GOAL \quad @ \quad TIME\_EXPR \tag{1}$$

where $GOAL$ is any goal from *ISCO* language and $TIME\_EXPR$ is given by:

$$TIME\_EXPR \rightarrow [TIME\_PT, TIME\_PT] \mid TIME\_PT \tag{2}$$

and $TIME\_PT$ (time point) is defined in 5.

Continuing example 2, suppose that *John Smith* works for such organisation and has take care of his kids, every *Monday* of *June*, between 9 and 11 AM. This fact could be insert into the personal personal agenda as follows:

```
:- personal_agenda := (name = 'John Smith',
                       appointment = 'Take care of kids')
@ [time_pt(2, Day, 6, Year, 9, 0, 0), time_pt(2, Day, 6, Year, 11, 0, 0)].
```

Some remarks must be made with respect to the time_pts above: first, instead of *Monday* [6] and *June* we used the trivial translation to integers, respectively, 2 and 6; second, hours are in a *0 ... 24* format; and, last *Day* and *Year* are finite domains variables that have their domains constrained to [1..30] and [0..$MAX\_INTEGER$], respectively.

5.4 Operational Semantics

In defining he operational semantics for the temporal extension presented above we follow the usual Logic Programming approach, i.e. we define derivations in a declarative way, by considering a derivation relation and introducing a set of inference rules for it. A tuple in the derivation relation is written as:

$$TIME\_EXPR \vdash GOAL$$

The inference rules have the form:

$$\frac{Consequent}{Antecedent}\{Conditions\}$$

where *Consequent* and *Antecedent* are derivation tuples, and *Conditions* are a (possibly empty) set of temporal relations defined in 5.1. In a declarative way, the inference rule says that *Consequent* holds if the *Antecedent* holds and *Conditions* are true. In a procedural way, to establish the *Consequent*, if the *Conditions* are true, establish the *Antecedent*.

Due to the fact that the temporal relations are already defined, the inference rules get rather simple and intuitive. Therefore, we will illustrate only one inference rule with time points and another with temporal intervals, respectively:

$$\frac{\vdash G > TP_1}{TP_2 \vdash G}\{TP_2 > TP_1\}$$

$$\frac{\vdash G \textbf{ during } [TP_3, TP_4]}{[TP_1, TP_2] \vdash G}\{[TP_3, TP_4] \textbf{ during } [TP_1, TP_2]\}$$

[6] The map chosen for days of the week, *Sunday* → 1, *Monday* → 2, *etc*, has to do with the translation of those names into the Portuguese language ...

The first rules says that we can establish that goal G is true after time point TP_1, if we can derive G at time TP_2, and TP_1 precedes TP_2. Similar reasoning applies to the second rule.

6 Scheduling meetings in a University: an applicational example

Its common sense how difficult it can be to schedule a meeting. In most cases, the reasons for that have to do with trying to satisfy the constraints imposed by the professional/personal agenda of the meeting participants.

Using the knowledge acquired from our professional experience in implementing Information Systems and as teachers in Universities, we propose a formalisation for this problematic based on the programming language presented in this paper. Moreover, we will demonstrate some of power of this language with illustrative temporal queries. We should notice that in this formalisation, most of the non–temporal elements where simplified or even removed, so that we could focus on the temporal issues.

6.1 Problem formalisation

Since we are talking about scheduling meetings in a (Portuguese) University, we start by presenting the class that contains, for each year, data about the periods of exams, holidays, classes ... , that is, we start by defining the *School Calendar*:

```
temporal( interval ) class school_calendar.
        period: text.
```

The reason for a temporal interval class, can be easily seen as we insert some data into it:

```
:- school_calendar := (period = 'Odd Semester 01/02')
@ [time_pt(1, 2001, 10, 1, 8, 0, 0), time_pt(7, 2002, 2, 2, 20, 0, 0)].

:- school_calendar := (period = 'Even Semester 01/02')
@ [time_pt(1, 2002, 3, 4, 8, 0, 0), time_pt(7, 2002, 6, 22, 20, 0, 0)].
```

Next, we are going to represent information relative to the teacher, that is going to be essentially, the *Personal Agenda* from example 2 and the *Classes Timetable*:

```
temporal( interval ) class classes_timetable.
        period: text.
        discipline: text
        teacher: text
        room: text
```

Again, lets insert some information into the defined class:

```
:- classes_timetable := (
        period = 'Even Semester 01/02'.
        discipline = 'Logic Programming'
        teacher = 'John Smith'
        room = 'ABC.10')
@ (time_pt(2, YY, MM, DD, 14, 0, 0), time_pt(2, YY, MM, DD, 16, 0, 0)).
```

The reader should notice that this entry shows information that is *cyclic*, i. e., *John Smith* teaches *Logic Programming* all *Mondays* (remember that 2 stands for *Monday*), between 14 and 16. For that purpose we used variables *YY*, *MM* and *DD* [7] that unify with any valid tuple of *Year*, *Month* and *Day*, respectively.

Last, but not least, we have the class that stores information about meetings:

```
temporal( interval ) class meeting.
        description: text.
        person: text.

:- meeting := (
        description = 'AGP paper'
        person = 'John Smith').
@ [time_pt(3, 2002, 6, 25, 9, 00), time_pt(3, 2002, 6, 25, 12, 00).
```

6.2 Temporal Queries

Our goal is to find out if, during a given time period, a teacher is available. One possible way of doing it is by negation, i. e. asking if he has no commitments for that period. So, our first interrogation is if the teacher *T* gives any classes during time period *[Start, End]*:

```
:- school_calendar( period = S ) 'contains' [Start, End],
classes_timetable( period = S, teacher = T ) 'overlaps' [Start, End]
```

In this query, we start by finding the semester, *S*, that contains the time period given. Using that semester we discover if the teacher as any classes that clash with that period. For instance, if we replace *T* by *John Smith* and and *[Start, End]* by *[time_pt(2, 2002, 6, 23, 10, 0, 0), time_pt(2, 2002, 6, 23, 15, 0, 0)]*, the answer will be yes, since the class of *Logic Programming* collides with this time period.

Next, we ask for any if he has any personal commitments:

```
:- personal_agenda( name = T ) 'overlaps' [Start, End]
```

[7] That in this case, must be the same in both limits of the interval

Again, with the same teacher and period we should get yes as answer (remember that he has to take care of his kids).

Using a similar reasoning, we can find if he has any meetings:

```
:- meeting( person = T ) 'overlaps' [Start, End]
```

Then, if we call the those three queries *personal*, *classes* and *meetings* all we have to do to see if he is available is: `:- not personal, not classes, not meetings.`

7 Comparison with other work

Both *TSQL2* [5] and the language proposed here have time points (or instants) and intervals. Time in our language is unidimensional whereas in *TSQL2* is multidimensional, having valid–time relations, transaction–time relations and bitemporal relations[8]. Neverthless, since our language is based on Constraint Logic Programming, it presents greater expressive power, namely in building temporal information and in querying that information.

There is much work that also uses *CLP* for temporal reasoning. From that we decided to compare with the one that is closer to our, namely using annotated constraint logic programming for temporal reasoning [13] and extending *CLP* for temporal reasoning [9]. The basis of our proposal is *ISCO* language whereas the other two works are based in (first–order) annotated constraint logic (*ACL*) and *CLP*, respectively. All three have in common time points and temporal intervals, but in ours lacks the temporal element *duration* (see section 8). In our work and in [9], since there is a concern with efficiency, there is a commitment to *CLP(FD)* and the other work leaves that issue open. Last, our time point is a composed term, whose structure is used for obtaining better efficiency and expressivity.

8 Conclusions and Future Work

We have presented a temporal extension of *ISCO* language. The extension allow us to do temporal representation and reasoning, and is based on temporal constraints between the language elements. We have shown its operational semantics and one application example that deals with scheduling meetings in the context of an University.

There is ongoing work to add a new temporal element present in many other approaches, *duration*.

Since there is one extension of GNU Prolog that incorporates Contextual Logic Programming (as described in [14]), future work will lead this framework a few steps further, by taking advantage of such paradigma and carrying temporal information to the context where the goal is evaluated.

[8] Although the bitemporal conceptual data model could be considered as an extension to our language.

Acknowledgments

Thanks João for the *working place!*

References

1. Abreu, S.: Isco: A practical language for heterogeneous information system construction. In: Proceedings of INAP'01, Tokyo, Japan, INAP (2001)
2. Prior, A.N.: Past, present and future. The Clarendon Press, Oxford (1967)
3. Allen, J.F.: Maintaining Knowledge about Temporal Intervals. Communications of the ACM **26** (1983) 832–843 ACM.
4. Vilain, M., Kautz, H.: Constraint Propagation Algorithms for Temporal Reasoning. In: Proc. Fifth National Conference on Artificial Intelligence, Philadelphia, PA, USA (1986) 377–382
5. Snodgrass, R.T., ed.: The TSQL2 Temporal Query Language. Kluwer Academic Publishers (1995)
6. Boehlen, M.H., Chomicki, J., Snodgrass, R.T., Toman, D.: Querying *TSQL2* databases with temporal logic. Lecture Notes in Computer Science **1057** (1996)
7. Chomicki, J., Toman, D.: Temporal logic in information systems. In Chomicki, J., Saake, G., eds.: Logics for Databases and Information Systems, Kluwer (1998) 31–70
8. Diaz, D., Codognet, P.: Design and implementation of the gnu prolog system. Journal of Functional and Logic Programming **2001** (2001)
9. Lamma, E., Milano, M., Mello, P.: Extending constraint logic programming for temporal reasoning. Annals of Mathematics and Artificial Intelligence **22** (1998) 139–158
10. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. Journal of Logic Programming **19/20** (1994) 503–581
11. Schwalb, E., Vila, L.: Temporal constraints: A survey. Constraints **3** (1998) 129–149
12. Gennari, R.: Temporal reasoning and constraint programming. Master's thesis, ILLC MoL-1998-1 (1998)
13. Fr"uhwirth, 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
14. 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 (1991) 284–299

A

João A

[1] CE
[2] Compu
[3]

Ab
dea
to
sta
tim
pos
Up
sta
se
fra
of
(e
D
tc
(I
M
o
n
b

1 Ir

Inspire
paradi
is give
logic p
and he
The s
so lon
they
DL
time.

[4] Sin