Logical Foundations and Implementation of an Extension of Temporal Prolog

María Laura Cobo Juan Carlos Augusto
G.I.R.I.T. [1] - I.C.I.C. [2]
Departamento de Ciencias de la Computación [3]
Universidad Nacional del Sur
Bahía Blanca - Argentina
Fax +54 291 4595136
[mlcobo, ccaugust] @criba.edu.ar

Abstract

The need of counting with the appropriate management of situations involving time and the notion of change, has been recognized as an important aspect in many Computer Science areas. This fact has made evident the need of tools to handle this kind of information. As a reflection of this relevance many temporal logic programming languages have been proposed in the last few years. In particular, we will present here a temporal logic programming called ETP. This language is based on the proposal presented by Gabbay in [Gab87]. The present article concerns the explanation of its logical foundations and the most important aspects of its implementation. We are going to leave out the interpreter's implementation code of this new language for space reasons, although, the reader can find it in [Cob98].

Initially we will present an abstract of Gabbay's work in order to familiarize the reader with the language we are going to extend and we will explain the usefulness of such extension. As regards the extension, we will provide the extended logic in detail and the definition of the language that take it as its base. We will also present the algorithm which provides answers for the queries of this language. The algorithm will be accompanied by a demonstration to prove that it computes a total function. Finally some examples will be presented to show potential uses of the language, together with the analysis of the goals reached and the remaining tasks.

Key words: Temporal Logic, Logic Programming, Temporal Logic Programming.

1 Introduction

In Computer Science the study and computational treatment of temporal notions has aroused a growing interest in the last decades. Whenever we need to model a dynamic system, temporal notions will have to be considered through to show how the system evolves time.

There are many Computer Science areas where time is involved in a direct and almost unavoidable way and where the temporal logics are used. The logic programming area, which is the one of interest in this work, has had a constant and notable development in the last two decades, see [Llo95]. Nevertheless, the classical model of logic programs limited to Horn's clauses with fixed point semantics, see [vEK76], is not enough for several purposes. In case we need to represent change models associated with time, we will require an extended language and, as a consequence, a computational procedure in agreement with the new language.

[1]Temporal Information Representation Research Group.
[2]Institute of Computer Science and Engineering.
[3]Partially supported for the Secretary of Science and Technology (Universidad Nacional del Sur).

There are two main approaches to the treatment of temporal notions and therefore for presenting temporal logics. One is based on *First Order Logic* and the other on *Modal Logic*. Both focuses are very useful and it can not be said that one is better than the other. This depends on the problem we are trying to solve, and many other theoretical and practical considerations, such as the philosophical/temporal conception of the person trying to solve it.

As a natural consequence of the research done on logic programming in the last decade a number of proposals have arisen to attenuate this absence of "temporal logic programming". These proposals are based on different basic principles. The best known strategies are:

- *Interval Logic Programming* where the semantic of the formulas is given through a temporal interpretation based on intervals.
- *Temporal Logic Programming*, this strategy is based on linear or branched-time temporal logics. The underlying idea is the extension of the traditional logic programming with the addition of temporal operators.

An abstract of the most representative works of the above mentioned alternatives, can be found in [OM94]. We will assume that the reader is familiarized with the basic terminology related to logic programming ([Llo84] or [Llo95]). The main goal of this article is to develop a temporal logic programming language, ETP. This language allows us a simple handling of those situations that involve the notions of time and change. The best known strategies to face the development of this kind of languages has been presented above, the language we are going to present is based on the second strategy mentioned, *Temporal Logic Programming*. Some languages that are similar to the one presented here have been considered, in particular the one offered in [Gab87], called *Temporal Prolog*. The language used in this article is an extension of *Temporal Prolog*, because it considers the most commonly used temporal operators and gives a solution to some cases unspecified by *Temporal Prolog*.

In the first place and taking into consideration the fundamental contribution of Gabbay's proposal to our work, we will present an abstract of it in the next section. Then we will give the extension realized, section 3; after that we will mention some aspects of its implementation, section 4; and finally illustrate its work in section 5.

2 Temporal Prolog

Temporal Prolog is an extension to the traditional logic programming proposed by Gabbay in [Gab87] to obtain a temporal logic programming. In his proposal the following connectives, familiar in the modal temporal logic literature, are used:

Fq: "q will be true in the future"

Pq: "q was true in the past"

$Gq =_{def} \neg F \neg q$

$Hq =_{def} \neg P \neg q$

$\Diamond q =_{def} q \vee Fq \vee Pq$

$\Box q =_{def} q \wedge Gq \wedge Hq$

The proposal is, then, based on the following logic:

DEFINITION **1** (adapted from [Gab87], page 217) The underlying logic of the temporal logic programming proposal, called *Temporal Prolog* has the following axiom schemata and rules of inference:

1. All the theorems and rules of the predicate logic.
2. The following rules of inference:

$$\frac{\vdash A}{\vdash PA} \quad ; \quad \frac{\vdash A}{\vdash FA} \quad ; \quad \frac{\vdash A, \vdash A \rightarrow B}{\vdash B}$$

3. The following axiom schemata:

$$G(A \rightarrow B) \rightarrow (GA \rightarrow GB)$$

$$H(A \rightarrow B) \rightarrow (HA \rightarrow HB)$$

$$GA \rightarrow GGA$$

$$HA \rightarrow HHA$$

$$A \rightarrow GPA$$

$$A \rightarrow HFA$$

4. The following axiom schemata indicating the use of a linear temporal structure:

$$FA \wedge FB \rightarrow F(A \wedge FB) \vee F(B \wedge FA) \vee F(A \wedge B)$$

$$PA \wedge PB \rightarrow P(A \wedge PB) \vee P(B \wedge PA) \vee P(A \wedge B)$$

■

The syntax of *Temporal Prolog* can be resumed through the next definition:

Definition **2** ([Gab87], page 222)

A *program* (or Data Base) is a set of "clauses".

A *clause* is either an "Always clause" or an "Ordinary clause".

An *Always clause* is $\Box A$ where A is an "Ordinary clause".

An *Ordinary clause* is a "Head" or an $B \rightarrow H$, where H is a "Head" and B is a "Body".

A *Head* is either an atomic formula or FA or PA, where A is a conjunction of Ordinary clauses.

A *Body* is either an atomic formula, a conjunction of bodies, an FA or PA or $\neg A$ where A is a body.

A *goal* is any body. ■

Through the above definition the consideration of formulas like the following is allowed:

$$P[F(FA(x) \wedge PB(y) \wedge A(y)) \wedge A(y) \wedge B(x)] \rightarrow P[F(A(x) \rightarrow FP(Q(z) \rightarrow Q(y)))]$$

The reader interested in this language can find the description of the algorithm in [Gab87]. We do not include it for space reasons. Furthermore, it is implicitly considered in the algorithm given in the next section.

3 An extension to Gabbay's algorithm and its implementation

The main idea of this section is to obtain a tool for temporal logic programming, having as its base a modal temporal logic. Considering the intuitive the operators are for the management of temporal notions. Among the different proposals of this kind we choose Gabbay's one, called "Temporal Prolog", presented in the previous section. The reason for choosing this proposal is that although its algorithm is pretty well known, there is no implementation which can be publicly accessed. It is not included here for space restrictions, but could be accessible on request to the authors or from [Cob98]. Some aspects of it were also corrected and the use of more temporal operators to facilitate the queries is allowed.

The extension mentioned before can be observed below: first through the consideration of the logic, then in the definition of the language and finally in the algorithm. On the other hand, considering that the extended algorithm was implemented some aspects of that implementation are commented. Finally we show some uses of the new tool.

3.1 The temporal operators

In the rest of this article we are going to use a set of symbols for the representation of the different temporal operators considered here. The operators can be characterized by different properties, before mentioning them, we will indicate the meaning of these properties:

reflexivity: when an operator is reflexive its semantics should necessarily consider the moment of evaluation of the operator in question. For example if we consider that the operator "always in the past" has this property, the proposition affected with this operator must be true in the present and also in all the moments of the past. In the case of considering binary operator as *Since*, *Until* the reflexivity of them means that for a pair of propositions, A and B the truthness of B in the present verifies *Since* (A, B) only by the presence of B in the actual moment. On the other hand, if we ask the operator to be *irreflexive*, A must at least be true in the present for the verification of the proposition we are considering.

strong: this property secures the existence of a next instant where the proposition must be verified. The opposite, a *weak* operator, does not ensure the existence of a next moment of time. In the case of binary operators as *Since*(A, B) the strength of it implies that the B proposition must be true in some instant of time. In the case of the weak version of the operator B could never be true. This case could be possible if we always have A and not B in the data base.

In this article we will use the following operators:

$\boxplus\,\phi$: "Always in the future ϕ" irreflexive.

$\boxminus\,\phi$: "Always in the past ϕ" irreflexive.

$\Box\,\phi$: $\boxminus\phi \wedge \phi \wedge \boxplus\phi$ ("Always in the past, now and future")

$\diamondplus\,\phi$: "Sometime in the future ϕ" irreflexive.

$\diamondminus\,\phi$: "Sometime in the past ϕ" irreflexive.

$\Diamond\,\phi$: $\diamondminus\phi \vee \phi \vee \diamondplus\phi$ ("Sometime in the past, now or in the future")

$\oplus\,\phi$: "In the next moment ϕ" strong.

$\ominus\,\phi$: "In the previous moment ϕ" strong.

Until(ϕ, ψ): "Until ψ be true, ϕ will be true" strong and reflexive.

Since(ϕ, ψ): "Since ψ be true, ϕ has been true" strong and reflexive.

3.2 The new programming language

After introducing the operators that we will use, we will explain the logic which formalize their behavior and then the temporal logic programming language derived from it. Let us consider the following definitions, which are classic in the literature of the area:

$$\boxminus q = \neg \diamondminus \neg q$$

$$\boxplus q = \neg \diamondplus \neg q$$

$$\Diamond q = q \vee \diamondplus q \vee \diamondminus q$$

$$\Box q = \neg \Diamond \neg q$$

Let us see the logic's rules of inference and axiom schemata.

Definition **3** (extension from the one given in [Gab87], page 217)
The temporal logic, has the following axiom schemata and rules, in the language with the operators: $\diamondminus$, $\diamondplus$, $\oplus$ and $\ominus$.

1. All the theorems and rules from the predicate logic.
2. The following rules of inference:

$$\frac{\vdash A}{\vdash \boxminus A} \; ; \quad \frac{\vdash A}{\vdash \boxplus A} \; ; \quad \frac{\vdash A, \vdash A \to B}{\vdash B} \; ; \quad \frac{\vdash A}{\vdash \oplus A} \; ; \quad \frac{\vdash A}{\vdash \ominus A}$$

3. The following axiom schemata:

$$\boxplus(A \to B) \to (\boxplus A \to \boxplus B)$$

$$\boxminus(A \to B) \to (\boxminus A \to \boxminus B)$$

$$\boxplus A \to \boxplus\boxplus A$$

$$\boxminus A \to \boxminus\boxminus A$$

$$A \to \boxplus \diamondminus A$$

$$A \to \boxminus \diamondplus A$$

$$\oplus(A \to B) \to (\oplus A \to \oplus B$$

$$\ominus(A \to B) \to (\ominus A \to \ominus B)$$

$$\vdash \oplus \neg A \leftrightarrow \neg \oplus A$$

$$\vdash \ominus \neg A \leftrightarrow \neg \ominus A$$

$$\oplus \ominus A \leftrightarrow A$$

$$\ominus \oplus A \leftrightarrow A$$

4. The following axiom schemata indicating the use of a linear temporal structure:

$$\diamondplus A \wedge \diamondplus B \to \diamondplus(A \wedge \diamondplus B) \vee \diamondplus(B \wedge \diamondplus A) \vee \diamondplus(A \wedge B)$$

$$\diamondminus A \wedge \diamondminus B \to \diamondminus(A \wedge \diamondminus B) \vee \diamondminus(B \wedge \diamondminus A) \vee \diamondminus(A \wedge B)$$

■

Below we are going to use the recently defined logic as a base of a temporal logic programming language.

Definition 4 (extension from the one given in [Gab87], page 212)

Consider a language with propositional atoms, the logic connectives: $\wedge$, $\vee$, $\rightarrow$, $\neg$ and the temporal operators: $\Diamond$, $\diamondminus$, $\diamondplus$, $\Box$, $\boxminus$, $\boxplus$, $\oplus$, $\ominus$, *Since*, *Until*. We define the notions of *program, clause, ordinary clause, always clause, head, body* and *goal* as follows:

A *program* is a set of “clauses”.

A *clause* is either an “Always clause” or an “Ordinary clause”.

An *Always clause* is $\Box A$, $\boxminus A$ or $\boxplus A$, where A is an “Ordinary clause”.

An *Ordinary clause* is a “Head” or an $B \rightarrow H$, where H is a “Head” and B is a “Body”.

A *Head* is either an atomic formula or $\diamondplus A$ or $\diamondminus A$ or $\oplus A$ or $\ominus A$, where A is a conjunction of Ordinary clauses.

A *Body* is either an atomic formula, a conjunction of bodies, an $\diamondplus A$ or $\diamondminus A$ or $\oplus A$ or $\ominus A$ or $\neg A$ where A is a body. A *goal* is any body or $\Diamond A$ or $\Box A$ or $\boxminus A$ or $\boxplus A$ or *Since*(A, B) or *Until*(A, B), where A and B are *bodies.*

■

The operators $\boxplus$ and $\boxminus$ can be part of any body, although they are not explicit in the definition. This is because they can be defined through the use of the other operators allowed (see above in this section). It is important to point out the fact that *Since* and *Until* are not available to the programmer, they can only be used by the user for the simplification of some queries, or to make them more easily. Now, we will give an algorithm which answered the queries made to a logic program in the version of extended “Temporal Prolog”. By simplifying the way of referring to this new language, from now on we will call it ETP, abbreviation of “*Extended Temporal Prolog*”.

Definition 5 (extension to the one given in [Gab87], page 222)

Let $\mathbf{P}$ be a database and G a goal. We define the notion of a *labelled computation tree* for the success or failure of G from $\mathbf{P}$. If G succeeds we write $\mathbf{P}?G{=}1$ and if G finitely fails we write $\mathbf{P}?G{=}0$. Let $(T, \leq, 0, V)$ be a labelled computation tree, such that $0 \leq t$, for all $t \in T$. V is a labelling function giving each $t \in T$ a triple $(\mathbf{P}(t), G(t), X(t))$, where $\mathbf{P}(t)$ is a database $\mathbf{P}$ in the t-th moment, $G(t)$ is a goal in the t-th moment and $X(t)$ is the purpose for such query. If $X(t) = 0$ we want the goal to fail, if $X(t) = 1$ we want it to succeed. ■

The algorithm presented below has as input a goal, we called it G, the objective of the query, referenced through the variable X and $\mathbf{P}$ a temporal logic program. The algorithm's output is the affirmative or negative answer to the goal with the given objective respect to $\mathbf{P}$.

Algorithm

$(T, \leq, 0, V)$ is a labelled computation tree for $\mathbf{P}?G = x$ if and only if the following conditions are satisfied:

1. $V(0) = (P, G, x)$
2. If $t \in T$ is an endpoint and $X(t) = 1$ then $G(t)$ is an atom q then one of the following conditions holds:

   (a) $q \in \mathbf{P}(t)$

(b) $\Box q \in \mathbf{P}(t)$

3. If $t \in T$ is an endpoint and $X(t) = 0$ then one of the following conditions holds:
   (a) $G(t)$ is an atom q and q is neither the head of an ordinary clause, nor is there in $\mathbf{P}(t)$ any always clause with head q.
   (b) $G(t)$ has the form $\diamondplus A$ (or $\diamondminus A$) and there are not clauses with head of the form $\diamondplus D$ or $\oplus A$ ($\diamondminus D$ or $\ominus A$ respectively).
   (c) $G(t)$ has the form $\oplus A$ (or $\ominus A$) and there are not clauses with head of the form $\oplus A$ ($\ominus A$ respectively)
4. If t is not an endpoint, $X(t) = 1$ and $G(t)$ is an atom q then t has exactly one immediate successor s in the tree with $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ and one of the following conditions holds:
   (a) $G(s) \rightarrow q \in \mathbf{P}(t)$ or
   (b) $\Box(G(s) \rightarrow q) \in \mathbf{P}(t)$
5. If t is not an endpoint, $X(t) = 1$ and $G(t)$ is a body q then t has exactly one immediate successor s in the tree with $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ and one of the following conditions holds:
   (a) $G(s) \rightarrow q \in \mathbf{P}(t)$ or
   (b) $\Box(G(s) \rightarrow q) \in \mathbf{P}(t)$
6. If t is not an endpoint, $X(t) = 0$ and $G(t)$ is an atom q then t has $s_1, \ldots, s_k$ as immediate successors in the tree with $\mathbf{P}(s_i) = \mathbf{P}(t)$, $X(s_i) = 0$ and for some m such that $0 \leq m \leq k$, the clauses $G(s_i) \rightarrow q$ for $i \leq m$ and the clauses $\Box(G(s_i) \rightarrow q)$ for $m \leq i \leq k$ are exactly all the clauses with the above form in $\mathbf{P}(t)$
7. If t is not an endpoint, $X(t) = 1$ and $G(t) = A_1 \wedge A_2$, then t has two immediate successors, s_1 and s_2, with $\mathbf{P}(s_i) = \mathbf{P}(t)$, $X(s_i) = 1$ and $G(s_i) = A_i$
8. If t is not an endpoint, $X(t) = 0$ and $G(t) = A_1 \wedge A_2$, then t has exactly one immediate successor s, and for some $i \in \{1, 2\}$, $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s_i) = 0$ and $G(s_i) = A_i$
9. If t is not an endpoint and $G(t) = \neg A$, then t has exactly one immediate successor s, and $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1 - X(t)$ and $G(s) = A$
10. If t is not an endpoint, $X(t) = 1$ and $G(t) = \diamondplus A$, then t has exactly one immediate successor, s_1 and the following condition holds:

    $\mathbf{P}(s_1) = \mathbf{P}(t)$, $X(s_1) = 1$ and $G(s_2) = \oplus A$;

    or has exactly two immediate successors: s_1, s_2, and one of the following conditions holds:
    (a) $\mathbf{P}(s_1) = \mathbf{P}(t)$ and $X(s_1) = 1$ and for some H
       i. $G(s_1) \rightarrow \diamondplus H \in \mathbf{P}(t)$ and
       ii. $\mathbf{P}(s_2) =$ { all the *always clauses* in $\mathbf{P}$*(t)*} $\cup \{H\} \cup$
       $\{\diamondminus C | C$ is an *ordinary clause* in $\mathbf{P}$*(t)*} $\cup$
       $\{C | \oplus C$ is an *ordinary clause* in $\mathbf{P}$*(t)*} and
       iii. $X(s_2) = 1$

iv. $G(s_2) = A \vee \diamondplus A$

(b) idem to item 10a except that the condition $G(s_1) \rightarrow \diamondplus H \in \mathbf{P}(t)$ is replaced by $\Box(G(s_1) \rightarrow \diamondplus H) \in \mathbf{P}(t)$

(c) idem to item 10b except that the condition $\Box(G(s_1) \rightarrow \diamondplus H) \in \mathbf{P}(t)$ is replaced by $\Box(G(s_1) \rightarrow H) \in \mathbf{P}(t)$ and $G(s_2) = A \vee \diamondplus A$ by $G(s_2) = \diamondplus A$

(d) idem to item 10b except that the condition $\Box(G(s_1) \rightarrow \diamondplus H) \in \mathbf{P}(t)$ is replaced by $\boxplus(G(s_1) \rightarrow \diamondplus H) \in \mathbf{P}(t)$

(e) idem to item 10c except that the condition $\Box(G(s_1) \rightarrow H) \in \mathbf{P}(t)$ is replaced by $\boxplus(G(s_1) \rightarrow H) \in \mathbf{P}(t)$

11. If t is not an endpoint, $X(t) = 1$ and $G(t) = \diamondminus A$, then t has either one immediate successor s_1 or has exactly two immediate successors: s_1, s_2, and one of the conditions of the item 10 holds, applying the mirror image rule to the conditions described there.

12. If t is not an endpoint, $X(t) = 0$ and $G(t) = \diamondplus A$, then t has $s_1, \ldots, s_k$ as immediate successors and for m, n such that $1 \leq m \leq n \leq k$ and some well formed formulas $D_1, \ldots, D_k$, $D_i (i \leq m)$ are exactly all the clauses of the form $\Box(B_i \rightarrow H_i)$ in $\mathbf{P}(t)$, $D(m+1), \ldots, D(n)$ are exactly all the clauses of the form $\Box(B_i \rightarrow FH_i)$ in $\mathbf{P}(t)$, $D(n+1), \ldots, D(k)$ are exactly all the clauses of the form $B_i \rightarrow FH_i$ in $\mathbf{P}(t)$, and for each $1 \leq i \leq k$ one of the following conditions holds:

    (a) $\mathbf{P}(s_i) = \mathbf{P}(t)$, $X(s_i) = 0$ and $G(s_i) = B_i$

    (b) $\mathbf{P}(s_i) =$ {all the *always clauses* in $\mathbf{P}(t)$} $\cup \{H_i\} \cup$
    $\{\diamondminus C | C$ is an *ordinary clause* in $\mathbf{P}(t)\} \cup$
    $\{C | \oplus C$ is an *ordinary clause* in $\mathbf{P}(t)\}$,
    $X(s_i) = 0$,
    $G(s_i) = A \vee \diamondplus A$, for $1 \leq i \leq k$, except when $1 \leq i \leq m$, in which case $G(s_i) = \diamondplus A$

13. If t is not an endpoint, $X(t) = 0$ and $G(t) = \diamondminus A$, then conditions of termination are similar to the ones in item 12 applying the mirror image rule.

14. If t is not an endpoint, $X(t) = 0$ and $G(t) = A_1 \vee A_2$, then t has exactly two immediate successors: s_1 and s_2, with $\mathbf{P}(s_i) = \mathbf{P}(t)$, $X(s_i) = 0$ and $G(s_i) = A_i$

15. If t is not and endpoint, $X(t) = 1$ and $G(t) = A_1 \vee A_2$, then t has exactly one immediate successor s, $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ and $G(s)$ is either A_1 or A_2

16. If t is not an endpoint, $X(t) = 1$ and $G(t) = \oplus A$, then t has exactly two immediate successors, s_1 and s_2, and one of the following conditions holds:

    (a) $\mathbf{P}(s_1) =$ {all the *always clauses* in $\mathbf{P}(t)$} $\cup$
    $\{\diamondminus C | C$ is an *ordinary clause* in $\mathbf{P}(t)\} \cup$
    $\{C | \oplus C$ is an *ordinary clause* in $\mathbf{P}(t)\}$,
    $X(s_1) = 1$ and $G(s_1) = A$

    (b) $\mathbf{P}(s_1) = \mathbf{P}(t)$, $X(s_1) = 1$, $\boxplus(G(s_1) \rightarrow A) \in \mathbf{P}(t)$

    (c) $\mathbf{P}(s_1) = \mathbf{P}(t)$, $X(s_1) = 1$, and for some H, $G(s_1) \rightarrow \oplus H \in \mathbf{P}(t)$ $\mathbf{P}(s_2) = \mathbf{P}(t)$, $X(s_2) = 1$ and $G(s_2) = A$

    (d) idem to item 16c except that the condition $G(s_1) \rightarrow \oplus H \in \mathbf{P}(t)$ is replaced by $\Box(G(s_1) \rightarrow \oplus H) \in \mathbf{P}(t)$

    (e) idem to item 16c except that the condition $G(s_1) \rightarrow \oplus H \in \mathbf{P}(t)$ is replaced by $\boxplus(G(s_1) \rightarrow H) \in \mathbf{P}(t)$

17. If t is not an endpoint, $X(t) = 0$ and $G(t) = \oplus A$, then t has exactly one immediate successor, s, and one of the conditions of item 16 holds with $X(s) = 1$ replaced by $X(s) = 0$

18. If t is not an endpoint, $X(t) = 1$ and $G(t) = \ominus A$, then t has exactly two immediate successors, s_1, s_2, and one of the conditions of item 16 holds applying the mirror image rule.

19. If t is not an endpoint, $X(t) = 0$ and $G(t) = \ominus A$, then t has one immediate successor, s, and the condition of item 17 holds applying the mirror image rule.

20. If t is not an endpoint, $X(t) = 1$ and $G(t) = \mathit{Until}(A, B)$, then t has one immediate successor, s, and one of the following conditions holds:

    (a) $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ and $G(s) = A \wedge \boxplus A \wedge \diamondplus B$

    (b) $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ and $G(s) = B \vee (A \wedge \oplus(\mathit{Until}(A, B)))$

21. If t is not an endpoint, $X(t) = 0$ and $G(t) = \mathit{Until}(A, B)$, then t has one immediate successor, s, and one of the following conditions holds:

    (a) $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ and $G(s) = A \wedge \boxplus A \wedge \diamondplus B$

    (b) $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ and $G(s) = B \vee (A \wedge \oplus(\mathit{Until}(A, B)))$

22. If t is not an endpoint, $X(t) = 1$ and $G(t) = \mathit{Since}(A, B)$, then t has one immediate successor, s, and one of the following conditions holds:

    (a) $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ and $G(s) = A \wedge \boxminus A \wedge \diamondminus B$

    (b) $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ and $G(s) = B \vee (A \wedge \ominus(\mathit{Since}(A, B)))$

23. If t is not an endpoint, $X(t) = 0$ and $G(t) = \mathit{Since}(A, B)$, then t has one immediate successor, s, and one of the following conditions holds:

    (a) $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ and $G(s) = A \wedge \boxminus A \wedge \diamondminus B$

    (b) $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ and $G(s) = B \vee (A \wedge \ominus(\mathit{Since}(A, B)))$

□

Definition **6** ([Gab87], page 224) $\mathbf{P}?G$ *succeeds* if $\mathbf{P}?G = 1$ has a finite computation tree. $\mathbf{P}?G$ *finitely fails* if $\mathbf{P}?G = 0$ has a finite computation tree. ■

Another interesting aspect of the proposal developed in the article is the possibility of answering questions of importance easily through the use of logic programming. In the rest of this section we will demonstrate that every ETP program ends its computation. First we are going to demonstrate a similar result for “Temporal Prolog”; in this way we complete Gabbay's proposal, who did not publish such result.

We are going to symbolize the content of a program written in ETP as $P(t)$, $t \geq 0$. This must be understood as the content of the program after the addition of pieces of information. For example, the initial program will be denoted as $P(0)$. After the use of some rules of inference t times, t pieces of information can be added to it, and this will be denoted through $P(t)$.

In order to prove the following Theorems and Lemmas we have the following hypothesis:

- This result is over the algorithm, not considering a particular implementation. For example, if we use Prolog, it will be enough to consider a function or omit `occurs check` to obtain infinite computations.

- The algorithm responds to the queries checking the set of facts and applying the rules of inference that form the base P. It is going to do this for each case, where the goal may be a success or failure.
- Make sure that there are no cycles, introduced by the programmer. Note that the algorithm do not consider the tautologies.

LEMMA **1** Let $P(t)$, $t \geq 0$, a Temporal Prolog's program, $P(t)$ is always a finite set of clauses. ■

Proof **1** Let $n1$ be the amount of facts, $n2$ the amount of ordinary rules, $n3$ the amount of always rules, $k1$ the amount of inferred facts and $k2$ the amount of inferred rules. If $P(0)$ is the initial program in Temporal Prolog then $|P(0)| = n_1 + n_2 + n_3$. To calculate the cardinality of $P(t)$ an additional fact must be taken into account: the syntax of Temporal Prolog only allows the use of the $\Box$ operator in the most outside level of a rule. Then no always rule can be the consequent of an implication and can not be deduced by Modus Ponens either. Considering that only the amount of facts and ordinary clauses can grow, we have $|P(t)| = (n_1 + k_1) + (n_2 + k_2) + n_3$, $k_1, k_2 \geq 0$. The k_1 and k_2 constants have a superior cota, this is because each time we use an inference rule this one is applied to an only one rule and only once. □

THEOREM **1** Every Temporal Prolog labelled computation tree has a finite set of nodes. ■

Proof **2**
We know that each step is based on the facts and rules that are in $P(t)$:
1) $G(t)$ is an atom, in the cases analogous to the items 2, 3a or 4 in [Gab87], all the facts an rules from $P(t)$must be inspectioned.
2) $G(t)$ is of the form $A \wedge B$ worst case: $x(t) = 1$ with number of son nodes:2
3) $G(t)$ is of the form $A \vee B$ worst case: $x(t) = 0$ with number of son nodes:2
4) $G(t)$ is of the form $\neg A$ worst case a unique successor in the tree
5) $G(t)$ is of the form $\diamondplus A$ (analogous for $\diamondminus A$), worst case $x(t) = 0$ and must be verified that can not be deduced from the $P(t)$'s clauses, analogous cases to the items 2, 3a o 4 in [Gab87].
In each case the amount of sons of each node forms a set which cardinality is less or equal to the $P(t)$'s. For each query the tree will build a finite amount of nodes, each of them will have a finite amount of facts and rules (lemma 1). □

LEMA **1** Let $P(t)$, $t \geq 0$, an Extended Temporal Prolog's program, $P(t)$ is always a finite set of clauses. ■

Proof **3** Idem lemma 1. □

THEOREM **2** Each Extended Temporal Prolog labelled computation tree has a finite set of nodes. ■

Proof **4** All we have to do is the extension of the demonstration below for the consideration of the added operator:

> If $G(t)$ is of the form $\oplus A$ (the result is similar for $\ominus A$), worst case $x(t) = 1$ with two immediate successors, one of them implies the search through all the rules in $P(t)$.

We must recall that the restrictive queries made with the operators $\mathcal{S}$ o $\mathcal{U}$ are translated to their equivalent formula using the operators considered above. Again the tree will build a finite amount of nodes, each of them deal with a finite amount of facts and rules (lemma 1). □

However, it is important to point out that the results about the finishing of the algorithms are applicable only over the logical part. On the other hand, the results do not concern a particular implementation, as can be one in Prolog that we know could never finish. This result is useful because it let us know that if we use another tool for the implementation of the interpreter, for example, one without functions and breadth first search the algorithm is always going to find an answer.

4 Considerations about the implementation

Apart from presenting some considerations related to the implementation divergences between our proposal and Gabbay's one will be indicated. It is important to point out that the implementation of the inference rules and axioms plays a fundamental role in the efficiency of this kind of tools. But it is one of the most difficult aspects to reach. Knowing how critical this aspect is for these kinds of programs, we try to make this work more efficient adding some additional rules to avoid the process of inference in trivial cases or queries of direct answer.

An important aspect in which our proposal differs form Gabbay's one is the following: If we find a goal of the form $\diamondplus(A \wedge B)$ being A and B any ordinary clauses, it is intuitively clear that it is not equivalent to solve $\diamondplus A \wedge \diamondplus B$. In the first formula the temporal operator has more precedence than the logical one, that is why A and B must be true in the same moment. In the second one the temporal operators that affect A and B are independent between them so A and B do not need to be true in the same moment. Gabbay's algorithm does not provide a way to solve this kind of formulas appropriately, that is why rules were added to the extended algorithm. This leads us to the addition of an special rule in the implementation too. The introduction of this rule makes necessary the revision of all the items in rule 10, except the first one, the condition $G \neq F_1 \wedge F_2 \wedge \ldots \wedge F_n$, where the goal is $\diamondplus G$. We mentioned above that a new rule was introduced which solves queries of the form $\diamondplus(A \wedge B)$:

> If t is not an endpoint, $X(t) = 1$ and $G(t) = \diamondplus(A \wedge B)$, then t has exactly two immediate successors: s_1, s_2 and one of the following conditions holds:
>
> 1. $\mathbf{P}(s_1) = \mathbf{P}(t)$ y $X(s_1) = 1$ and for some H
>   - (a) $G(s_1) \rightarrow \diamondplus H \in \mathbf{P}(t)$ and
>   - (b) $\mathbf{P}(s_2) =$ all *always clauses* in $\mathbf{P}(t)$ $\cup \{H\} \cup$ $\{\diamondminus C | C$ is an *ordinary clause* in $\mathbf{P}(t)\} \cup$ $\{C | \oplus C$ is an *ordinary clause* in $\mathbf{P}(t)\}$ and
>   - (c) $X(s_2) = 1$
>   - (d) $G(s_2) = \diamondplus A$
> 2. idem item 1 except that the condition $G(s_1) \rightarrow \diamondplus H \in \mathbf{P}(t)$ is replaced by $\Box(G(s_1) \rightarrow \diamondplus H) \in \mathbf{P}(t)$
> 3. idem item 1 except that the condition $G(s_1) \rightarrow \diamondplus H \in \mathbf{P}(t)$ is replaced by $\Box(G(s_1) \rightarrow H) \in \mathbf{P}(t)$
> 4. idem item 1 except that the condition $G(s_1) \rightarrow \diamondplus H \in \mathbf{P}(t)$ is replaced by $\boxplus(G(s_1) \rightarrow \diamondplus H) \in \mathbf{P}(t)$
> 5. idem item 3 except that the condition $\Box(G(s_1) \rightarrow H) \in \mathbf{P}(t)$ is replaced by $\boxplus(G(s_1) \rightarrow H) \in \mathbf{P}(t)$

In addition to this new rule for goals of the form $\diamondplus(A \wedge B)$ we need some predicates. The reason for their existence was explained previously and their function is to solve trivial situations as the fact that the goal is part of some conjunction. As an example, let us suppose that we have the goal $\diamondplus(A \wedge B)$ and in the base we have the following clause $\diamondplus(A \wedge C \wedge \ldots \wedge B)$. If we do not add these predicates the algorithm has no way of finding $\diamondplus(A \wedge B)$ although it is pretty obvious that the query has a successful answer from the logical program. We have an analogous situation for $\diamondminus$ applying the mirror image rule.

This fact makes evident the necessity of predicates that realize the inverse work, i.e., if we deduce something from a $\diamondplus\phi$ clause, we know that the deduced formula and the one that makes possible the deduction happens in the same moment. This is very important if the clause that dares the deduction is affected by an $\diamondplus$ or $\diamondminus$ operator, because if we do not save the relation between these clauses, it will be lost because of the vagueness of the operator. On the other hand, with the other operators the relation does not need to be kept because they are more precise and the relation holds implicitly although is not expressed explicitly in the base. The following situation exemplifies the above idea, if we have $\Box A \rightarrow B$ and $\diamondplus A$, it is clear that we deduce $\diamondplus B$. We also know that A and B are true in the same moment of the future, but if we do not put $\diamondplus(A \wedge B)$ in the base, the relation that exists between A and B will be lost because the operator does not make any reference to a precise moment.

Another important divergence is pointed out by the addition of two rules, that are used to travel through the temporal line. These rules follow this strategy: "*If the algorithm can not solve a query in the present moment in none of the ways provided by the algorithm, then if some different reasoning is possible in the next moment or in the previous one, move to that instant and transform the query in the appropriate way*". In case it is useful to move through the temporal line to the next moment we transform the query A in $\ominus A$, in an analogous way if we move to the previous moment we transform it in $\oplus A$.
The possibility of obtaining a different reasoning lies in counting with at least one $\oplus A$ clause, in the base of the present moment, for moving the query one instant to the future or at least one $\ominus A$ clause for doing it to the past.

5 Examples

The examples we consider here include some of the examples considered by Gabbay for his algorithm besides our own examples which are used for the testing of special cases as well as basics queries.

Example **1** (Example 3.1, page 215 ([Gab87])) This is just a simple example where only basic operators are used. In ETP the example will be:

Base: { $a \rightarrow b$,
$\Box(b \rightarrow \diamondplus c)$,
a }

Answer to the query $\diamondplus c$: *yes.* □

Example **2** (Example 3.5, page 219 ([Gab87])) In this example we show the possibility of dealing with predicates, i.e., it allowed us to show that the tool was not build only for the management of propositions. In ETP the example will be:

Base: { $a1(a)$,
$a1(X) \rightarrow \diamondplus b1(Y)$ }

Answer to the query $\diamondplus(b1(a) \wedge b1(b))$: *no.*
Note: $a1$ and $b1$ are names for any predicate, X and Y represent variables. □

Example **3** This example shows how it is possible to answer a query that involves a *Since* operator, although this one can not be in the data base. In ETP the example will be:

Base: $\{\quad d\ ,$
$d \rightarrow a\ ,$
$a \rightarrow \ominus(\ominus c)\ ,$
$\ominus(\ominus(c \rightarrow \oplus(a))\ ,\ \boxminus(c \rightarrow a)\ ,$
$\Box(c \rightarrow \ominus b)\ \}$

Answer to the query $\mathit{Since}(a, b)$: *yes.* □

Prolog has been used as a query language for data bases. Analogously, a temporal logic programming language can be used as the base of some logical queries system over data bases with some temporal information, such us historic data bases ([TCG+93]). We will present now, an example of a real application. Let us see how some of the examples presented by Abadi y Manna in [AM89], page 288, look in this new system. Take the following example.

EXAMPLE 4 We have a data base with some facts related to the history of some company after some initial date. We have information such as who worked in which department and the salary earned by that employee. We would like to get a list of the employees, X, with their salaries, Y. Only those who worked in the toys department after the initial date. And also worked there while the manager was John. In temporal logic we simply write the query as:

$$\Diamond(\mathit{manager}(\mathit{John}) \wedge \mathit{in\_department}(X, \mathit{Toy}) \wedge \mathit{salary}(X, Y))$$

we suppose that the predicate *manager*, *in_department* and *salary* have been defined in an appropriate way, in accordance with the program. We obtain then the wished output for the query.

Taking the same suppositions exposed above and considering that the present is the initial date for the activities of the company and that the query is being made in that instant. The query in ETP will look like this:

$$(\mathit{manager}(\mathit{John}) \wedge \mathit{in\_department}(x, \mathit{Toy}) \wedge \mathit{salary}(x, y)) \vee$$

$$(\Diamond\!\!\!\!+\,(\mathit{manager}(\mathit{John}) \wedge \mathit{in\_department}(x, \mathit{Toy}) \wedge \mathit{salary}(x, y))$$

Observation: the disjunction is necessary because we assume $\Diamond\!\!\!\!+$ irreflexive (see section 3.1) □

EXAMPLE 5 In a similar way, we would like to know how much the salary of John has increased when he got the promotion from salesman to manager. Supposing that the program counts with the rule:

$$\mathit{increase}(X, Y) \leftarrow \mathit{salary}(X, Y_1), \oplus\mathit{salary}(X, Y_2), Y\ \mathit{is}\ (Y_2 - Y_1)$$

expressing that the increase in the salary of X from the moment n to the moment $n + 1$, is the difference between the salary of X in the moment n and the salary of X in the moment $n + 1$, and the query:

$$\Diamond(\mathit{salesman}(\mathit{John}) \wedge \oplus\mathit{manager}(\mathit{John}) \wedge \mathit{increase}(\mathit{John}, Y))$$

Taking the same set of suppositions that for the previous example, the program and query will look respectively as follows:

$$\mathit{increase}(X, Y) \leftarrow \mathit{salary}(X, Y_1) \wedge \oplus\mathit{salary}(X, Y_2) \wedge Y\ \mathit{is}\ (Y_2 - Y_1)$$

$$(\mathit{salesman}(\mathit{John}) \wedge \oplus\mathit{manager}(\mathit{John}) \wedge \mathit{increase}(\mathit{John}, Y)) \vee$$

$$(\Diamond\!\!\!\!+\,(\mathit{salesman}(\mathit{John}) \wedge \oplus\mathit{manager}(\mathit{John}) \wedge \mathit{increase}(\mathit{John}, Y)))$$

□

Example **6** Finally we would want to express that every employee has always been an employee:

$$\Box employee(x) \leftarrow employee(x)$$

In ETP we express it, in an equivalent way, as:

$$\Box(employee(X) \rightarrow \oplus(employee(X)))$$

$\Box$

6 Conclusions and future work

We have presented a temporal logic programming language, which is an extension of "Temporal Prolog". Turning our attention to the temporal logical language, its definition considers as basic operators the set formed by: $\diamondplus$, $\diamondminus$, $\oplus$ and $\ominus$, which gives us the idea of the extension mentioned above, because "Temporal Prolog" does not consider the operators $\oplus$ and $\ominus$ in its definition. It is important to notice that we allow the use of *Since* and *Until* operators in the user level to make the formulation of some queries easier although this does not add more expressive power. This is so because we only use them for a more confortable way of writing some queries that can also be written using the $\diamondplus$, $\diamondminus$, $\oplus$ and $\ominus$ operators. This extension has been implemented and we have commented some of their main characteristics. We also gave an algorithm for the reader and a demonstration that always stops. Besides the implementation is based on the notion of labelled computation tree. This notion had been defined by Gabbay in [Gab87]. It important to point out that the language we proposed is a cut of the temporal logic taken as its base, just as Prolog is of the predicate calculus. The presentation of this programming language born through an extension of Prolog, represents an advance in giving a way to solve the problems that are influenced by time in a more natural way.

We can say that the extension realized is very interesting not only for the new operators but also for the solution provided for some queries. This last point is important because the Temporal Prolog algorithm does not provide a way to solve them. We have already mentioned that Gabbay did not give an implementation of his algorithm or at least it is not accessible, we count with an implementation of ETP algorithm and can be obtained from [Cob98] or contacting the authors. The implementation can be improved in many aspects. One of them is to deal the control and handling of the levels used in the implementation to an external meta-predicate. This strategy was suggested by Sterling and Shapiro in [SS86]). This improvement makes the actual predicates to be simpler and smaller, because we get rid of the propositions that control the levels and change them. Another aspects of fundamental importance is to try to rise the efficiency of the deductive mechanism, i.e., of the implementation of the inference rules. It is also important to mention that a definition of systematic testing techniques for this kind of programs are in elaboration.

Acknowledgments

We would like to thank to the members of GIRIT, Marisa Sanchez and María Mercedes Vitturini for the collaboration given in the elaboration of this article.

References

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

[Cob98] María Laura Cobo. Dos propuestas de programación en lógica temporal. Tesis de Licenciatura. Departamento de Cs. de la Computación, Universidad Nacional del Sur, 1998.

[Gab87] D. Gabbay. Modal and temporal logic programming. In Antony Galton, editor, *Temporal Logic and their Applications*, pages 197–236. Academic Press, 1987.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Llo95] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, third edition edition, 1995.

[OM94] M. Orgun and W. Ma. An overview of temporal and modal logic programming. In *Proceedings of the FICTL (ICTL 94)*, pages 445–479, Bonn, Germany, 1994. Springer Verlag.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1986.

[TCG$^+$93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Data Bases (Theory, Design and Implementation).* The Benjamin Cummings Pub. Co., California, 1993.

[vEK76] M. van Emdem and Robert Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the Association for Computing Machinery*, 23(4):733–742, 1976.