Fundamentos Lógicos e Implementación de una Extensión a 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
[mlcobo, ccaugust] @criba.edu.ar

Palabras Clave: Lógica Temporal, Programación en Lógica, Programación en Lógica Temporal.

Resumen

La necesidad de contar con un manejo adecuado de situaciones que involucren el tiempo y la noción de cambio, ha sido reconocido como un aspecto importante en muchas áreas de ciencias de la computación. Esto ha hecho evidente la necesidad de herramientas que manejen este tipo de información. Como reflejo de esta relevancia, en los últimos años se han propuesto diversos lenguajes de programación en lógica temporal. En particular, presentaremos aquí un lenguaje de programación en lógica temporal denominado ETP, abreviatura de "*Extended Temporal Prolog*". Este lenguaje está basado en una propuesta presentada por Gabbay en [Gab87]. Dedicaremos el artículo a explicar sus fundamentos lógicos y los aspectos más destacados de su implementación. Omitiremos dar aquí el código de la implementación de interprete del nuevo lenguaje por razones de espacio pudiendo este ser obtenido de [Cob98]. Cabe destacar que no se conoce una implementación de uso público de la propuesta de Gabbay por lo tanto la disponibilidad de la implementación de ETP es parte del aporte del trabajo que resumimos en este artículo.

Inicialmente se presentará un resumen del trabajo de Gabbay a fin de que el lector conozca el lenguaje que extenderemos y explicaremos la utilidad de dicha extensión. El conjunto de operadores considerados en este lenguaje contiene a los operadores más comunmente utilizados en este tipo de programación. En lo que respecta a la extensión brindaremos un detalle de la lógica extendida y la definición del lenguaje que la toma como base. También presentaremos el algoritmo que permite responder consultas de dicho lenguaje. Acompañaremos el algoritmo con una demostración de que el mismo computa una función total (siempre se detiene). En lo que respecta a la implementación de un lenguaje de este tipo, se puede decir que es una novedad, ya que Gabbay no presentó una implementación con su propuesta. Finalmente se presentarán ejemplos a fin de ilustrar potenciales usos del lenguaje para concluir con el análisis de los objetivos alcanzados y las tareas que aún quedan pendientes.

[1] Grupo de Investigación en Representación de Información Temporal.
[2] Instituto de Ciencias e Ingeniería de la Computación.
[3] Parcialmente financiado por la Secretaría de Ciencia y Técnología (Universidad Nacional del Sur).

Fundamentos Lógicos e Implementación de una Extensión a Temporal Prolog

1 Introducción

En Ciencias de la computación el estudio y tratamiento computacional de nociones temporales es un área que ha despertado un interés creciente en las últimas décadas. Siempre que se trate de modelar un sistema dinámico este involucrará la consideración de nociones temporales para representar como el sistema se va modificando a través del tiempo.

Hay varias áreas de las Ciencias de la Computación donde el tiempo está involucrado en una forma directa y casi ineludible y donde las lógicas temporales son usadas. El área de programación en lógica, que es la de interés a este trabajo, ha tenido un desarrollo constante y notable en las últimas dos décadas, ver [Llo95]. Sin embargo, el modelo clásico de programas lógicos limitados a cláusulas de Horn con su semántica de punto fijo, ver [vEK76], es insuficiente para muchos propósitos. Por ejemplo, en el caso que necesitemos representar modelos de cambio como los asociados al tiempo requeriremos un lenguaje extendido y en consecuencia un procedimiento de computación acorde al nuevo lenguaje.

Existen dos enfoques principales para el tratamiento de nociones temporales y por lo tanto para presentar lógicas temporales, ellos son: el enfoque apoyado en la *Lógica de Primer Orden* y el enfoque basado en la *Lógica Modal*. Ambos enfoques resultan de mucha utilidad y no se puede decir que uno sea mejor que otro, eso depende del problema que se esté intentando resolver, y de muchas otras consideraciones, incluyendo la concepción filosófica y/o temporal de la persona que intente resolverlo.

Como consecuencia natural de estas investigaciones realizadas sobre la programación en lógica en la última década han surgido una serie de propuestas destinadas a atenuar esta falta de una "programación lógica temporal". Estas propuestas están basadas en diferentes principios básicos. Las estrategias más conocidas son:

- *Programación en lógica de intervalos* (o *Interval Logic Programming*) donde la semántica de las fórmulas está dada a través de una interpretación temporal basada en intervalos.
- *Programación en lógica temporal* (o *Temporal Logic Programming*), esta estrategia utiliza como base lógicas temporales basadas en tiempo lineal o ramificado. La idea básica subyacente es extender la programación en lógica tradicional con operadores temporales.

Un resumen de los trabajos más representativos de las alternativas antes mencionadas, puede encontrarse en [OM94]. En lo que resta del presente trabajo se asumirá que el lector conoce la terminología básica relacionada a la programación en lógica ([Llo84] o [Llo95]). Los objetivos principales de este trabajo son desarrollar un lenguaje de programación en lógica temporal, a fin de contar con una herramienta que permita un manejo sencillo de todas aquellas situaciones que involucran el manejo de tiempo o la noción de cambio. Se han presentado previamente las estrategias más conocidas para enfrentar el desarrollo de lenguajes de este tipo, el lenguaje que se presentará aquí está fundamentado en la segunda estrategia que se ha mencionado, *Temporal Logic Programing*. Se han considerado lenguajes similares al que se presentará, en particular el lenguaje propuesto en [Gab87], denominado *Temporal Prolog*. No obstante el lenguaje utilizado en este artículo es una extensión de *Temporal Prolog*, ya que considera los operadores temporales más comunmente utilizados y soluciona casos para los cuales *Temporal Prolog* no establece como proceder.

En primer lugar y teniendo en cuenta el aporte fundamental de la propuesta de Gabbay a este trabajo, se presentará un resumen de la misma en la próxima sección. Luego

se presentará la extensión realizada, sección 3; para luego mencionar aspectos de su implementación, sección 4; e ilustrar su funcionamiento con ejemplos en la sección 5.

2 Temporal Prolog

Temporal Prolog es una extensión propuesta por Gabbay en [Gab87] a la programación lógica tradicional para obtener una programación en lógica temporal. En la propuesta se consideran los siguientes conectivos familiares en la literatura de las lógicas modales temporales:

Fq: "q será verdadera en el futuro"

Pq: "q fue verdadera en el pasado"

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

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

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

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

La propuesta está basada, entonces, en la siguiente lógica:

Definición **1** (adaptada de [Gab87], página 217) La lógica temporal subyacente a la propuesta de programación en lógica temporal denominada *Temporal Prolog* tiene los siguientes axiomas y reglas de inferencia:

1. Todos los teoremas y reglas de la lógica de predicados.
2. Las siguientes reglas de inferencia:

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

3. Los siguientes esquemas de axioma:

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

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

$$GA \rightarrow GGA$$

$$HA \rightarrow HHA$$

$$A \rightarrow GPA$$

$$A \rightarrow HFA$$

4. Los siguientes esquemas de axioma denotando el uso de una estructura temporal lineal:

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

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

■

La sintáxis de *Temporal Prolog* puede ser resumida a través de la siguiente definición:

Definición **2** ([Gab87], página 212)

Un *programa* (o Base de Datos) es un conjunto de *cláusulas*

Una cláusula es o una cláusula *Always* o una cláusula *ordinaria*

Una cláusula *Always* es $\Box A$ donde A es una cláusula ordinaria

Una cláusula *ordinaria* es una *cabeza* o $H \rightarrow B$, donde H es la *cabeza* de la cláusula y B es el *cuerpo* de la cláusula.

Una cabeza de cláusula es una fórmula atómica, FA o PA, donde A es una conjunción de cláusulas ordinarias

Un cuerpo de cláusula es una fórmula atómica, una conjunción de cuerpos, FA, PA o $\neg A$, donde A es un cuerpo

Una *meta* es un cuerpo de cláusula. ■

Mediante esta definición se permite la consideración de fórmulas como:

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

El lector interesado puede encontrar la descripcion del algoritmo en [Gab87]. El mismo no fue incluído por razones de espacio y por estar presente de manera implícita en el algoritmo que se da en la siguiente sección.

3 Una extensión al algoritmo de Gabbay y su implementación

La idea central de esta sección es obtener una herramienta para programación en lógica temporal, tomando como base una lógica modal temporal, debido a lo intuitivo que resultan los operadores de este tipo de lógicas para el manejo de nociones temporales. De entre las propuestas de este tipo se escogió la propuesta de Gabbay, conocida como "Temporal Prolog", que se presentó en la sección anterior. La razón para tomar como base esta propuesta es que a pesar de ser conocido su algoritmo, no se conoce alguna implementación que pueda ser accedida públicamente. En este caso no la incluimos por razones de espacio pero la implementación sería accesible bajo requerimiento a los autores o a través de [Cob98]. También se corrigieron algunos aspectos del mismo y se permite el uso de más operadores temporales para facilitar las consultas.

La extensión a la que se hace mención se puede observar a continuación: primero a través de la lógica, luego en la definición del lenguaje y finalmente en el algoritmo. Por otro lado como este algoritmo extendido fue implementado se comentan aspectos de esa implementación, para finalmente mostrar ejemplos del uso de la nueva herramienta.

3.1 Los operadores temporales

A través de este trabajo se utilizarán una serie de símbolos para denotar los distintos operadores temporales a considerar. Los operadores pueden ser caracterizados por distintas propiedades, antes de mencionarlos se indicará el significado de las mismas:

> *reflexividad*: cuando un operador es reflexivo su semántica necesariamente debe considerar el momento de evaluación de dicho operador. Por ejemplo si se considera que el operador "siempre en el pasado"cumple con esta propiedad, la proposición afectada debe darse ahora, en el presente, además de en todos los momentos del pasado. En el caso de los operadores binarios como *Since*, *Until* la *reflexividad* del operador implica que para un par de proposiciones, A y B puede darse que B sea verdadera ahora provocando que $Since(A, B)$ se verifique ante la sola presencia de B en el momento actual. En cambio si se

pide que verifique *irreflexividad*, debe al menos ser verdadera A en el presente, para que verifique la proposición en cuestión.

fuerte: esta propiedad asegura la existencia de un próximo instante donde la proposición debe verificarse. El contrapuesto, un operador *débil*, no asegura siempre la existencia de un próximo instante de tiempo. El pedir que un operador binario como $Since(A, B)$ sea *fuerte* implica que la proposición B debe ser verdadera en algún momento, mientras que el pedir que sea *débil* implica que puede darse el caso de que B no sea verdadera en ningún momento. Esto podría darse si siempre se da A y no B en la base de datos.

En este trabajo utilizaremos los siguientes operadores:

$\boxplus\,\phi$: "Siempre en el futuro ϕ" irreflexivo.

$\boxminus\,\phi$: "Siempre en el pasado ϕ" irreflexivo.

$\Box\,\phi$: $\boxminus\phi \wedge \phi \wedge \boxplus\phi$ ("Siempre en el pasado, ahora y futuro")

$\diamondplus\,\phi$: "Alguna vez en el futuro ϕ" irreflexivo.

$\diamondminus\,\phi$: "Alguna vez en el pasado ϕ" irreflexivo.

$\Diamond\,\phi$: $\diamondminus\phi \vee \phi \vee \diamondplus\phi$ ("Alguna vez el pasado, en el futuro o ahora")

$\oplus\,\phi$: "En el próximo instante ϕ" fuerte.

$\ominus\,\phi$: "En el instante previo ϕ" fuerte.

$Until(\phi, \psi)$: "Hasta que ψ sea veradero, ϕ será verdadero" fuerte y reflexivo.

$Since(\phi, \psi)$: "Desde que ψ fue verdadero, ϕ ha sido verdadero" fuerte y reflexivo.

3.2 El nuevo lenguaje de programación

Ahora que hemos introducido los operadores temporales que se utilizarán, pasaremos a explicar la lógica en la que se formaliza su comportamiento y el lenguaje de programación en lógica temporal derivado de esta. Consideraremos las siguientes definiciones, clasicas en la literatura del área:

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

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

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

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

Veamos ahora las reglas de inferencia y axiomas de la lógica.

Definición **3** (extensión de la dada en [Gab87], página 217)
La lógica temporal, tiene los siguientes axiomas y reglas, en el lenguaje con $\diamondminus$, $\diamondplus$, $\oplus$ y $\ominus$.

1. Todos los teoremas y reglas de la lógica de predicados.
2. Las siguientes reglas de inferencia:

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

3. Los siguientes esquemas de axioma:

$$\boxplus(A \rightarrow B) \rightarrow (\boxplus A \rightarrow \boxplus B)$$

$$\boxminus(A \rightarrow B) \rightarrow (\boxminus A \rightarrow \boxminus B)$$

$$\boxplus A \rightarrow \boxplus\boxplus A$$

$$\boxminus A \rightarrow \boxminus\boxminus A$$

$$A \rightarrow \boxplus\diamondsuit A$$

$$A \rightarrow \boxminus\diamondsuit A$$

$$\oplus(A \rightarrow B) \rightarrow (\oplus A \rightarrow \oplus B$$

$$\ominus(A \rightarrow B) \rightarrow (\ominus A \rightarrow \ominus B)$$

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

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

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

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

4. Los siguientes esquemas de axioma denotando el uso de una estructura temporal lineal:

$$\diamondsuit A \wedge \diamondsuit B \rightarrow \diamondsuit(A \wedge \diamondsuit B) \vee \diamondsuit(B \wedge \diamondsuit A) \vee \diamondsuit(A \wedge B)$$

$$\diamondsuit A \wedge \diamondsuit B \rightarrow \diamondsuit(A \wedge \diamondsuit B) \vee \diamondsuit(B \wedge \diamondsuit A) \vee \diamondsuit(A \wedge B)$$

■

A continuación utilizaremos la lógica recién definida como base de un lenguaje de programación en lógica temporal.

DEFINICIÓN **4** (extensión de la dada en [Gab87], página 212) Considere un lenguaje con átomos proposicionales, los conectivos lógicos: $\wedge$, $\vee$, $\rightarrow$, $\neg$ y los operadores temporales: $\diamondsuit$, $\diamondsuit$, $\diamondsuit$, $\Box$, $\boxminus$, $\boxplus$, $\oplus$, $\ominus$, *Since*, *Until*. Definimos las nociones de *programa, cláusula ordinaria, cláusula always, cabeza, cuerpo* y *meta* como sigue:

Un *programa* es un conjunto de *cláusulas*.

Una *cláusula* es o una *cláusula ordinaria* o una *cláusula always*.

Una *cláusula always* es $\Box A$, $\boxminus A$ o $\boxplus A$, donde A es una *cláusula ordinaria*.

Una *cláusula ordinaria* es una *cabeza* o $A \rightarrow H$, donde A es un *cuerpo* y H es una *cabeza*.

Una *cabeza* es o una fórmula atómica o $\diamondsuit A$ o $\diamondsuit A$ o $\oplus A$ o $\ominus A$, donde A es una conjunción de *cláusulas ordinarias*.

Un *cuerpo* es o una fórmula atómica o una conjunción de cuerpos, $\diamondsuit A$ o $\diamondsuit A$ o $\oplus A$ o $\ominus A$ o $\neg A$, donde A es un *cuerpo*.

Una *meta* es cualquier *cuerpo* o $\diamondsuit A$ o $\Box A$ o $\boxminus A$ o $\boxplus A$ o $Since(A, B)$ o $Until(A, B)$, donde A y B son *cuerpos*.

■

Es importante notar que *Since* y *Until* no están disponibles para el programador, solo pueden ser utilizados por el usuario para realizar algunas consultas con más facilidad. Seguidamente se brindará un algoritmo para responder consultas a un programa lógico en esta versión de "Temporal Prolog" extendido. Para facilitar futuras alusiones al lenguaje recién definido, utilizaremos de aquí en adelante la sigla ETP, abreviatura de "*Extended Temporal Prolog*".

DEFINICIÓN **5** (extensión de la dada en [Gab87], página 222)

Sea $\mathbf{P}$ una base de conocimiento y G una meta. Definimos la noción de un *árbol de computación etiquetado* para el éxito o el fracaso finito de G a partir de $\mathbf{P}$. Un árbol de computación etiquetado es una cuádrupla $(T, \leq, 0, V)$ tal que $0 \leq t$ para todo $t \in T$. V es una "función de etiquetado" que asigna a cada $t \in T$ una terna $(\mathbf{P}(t), G(t), X(t))$, donde $\mathbf{P}(t)$ es una base de datos $\mathbf{P}$ en el momento t-ésimo, $G(t)$ es una consulta en el momento t-ésimo y $X(t)$ es un propósito para tal consulta. Si $X(t) = 0$ el propósito es que la consulta sea respondida negativamente, si $X(t) = 1$ el propósito es que la computación tenga éxito. ■

El algoritmo que se presenta a continuación tiene como datos de entrada una meta, que llamaremos G, el objetivo de la consulta, que se denota a través de la variable X y $\mathbf{P}$ una programa en lógica temporal. La salida del mismo es la respuesta afirmativa o negativa de la meta dada con el objetivo dado respecto de $\mathbf{P}$.

ALGORITMO

$(T, \leq, 0, V)$ es un árbol de computación etiquetado para $\mathbf{P}?G = x$ si y sólo si las siguientes condiciones son satisfechas:

1. $V(0) = (P, G, x)$
2. si $t \in T$ es un punto de terminación con $X(t) = 1$ y $G(t)$ es un átomo q entonces alguna de las siguientes condiciones es satisfecha:
   (a) $q \in \mathbf{P}(t)$
   (b) $\Box q \in \mathbf{P}(t)$
3. si $t \in T$ es un punto de terminación y $X(t) = 0$ entonces alguna de las siguientes condiciones es satisfecha:
   (a) $G(t)$ es un átomo q y q no es ni la cabeza de una cláusula ordinaria, ni hay en $\mathbf{P}(t)$ alguna cláusula always con cabeza q
   (b) $G(t)$ tiene la forma $\oplus A$ (o $\diamond A$) y no hay cláusulas con cabezas de la forma $\oplus D$ o $\oplus A$ (respectivamente $\diamond D$ o $\ominus A$)
   (c) $G(t)$ tiene la forma $\oplus A$ (o $\ominus A$) y no hay cláusulas con cabezas de la forma $\oplus A$ (respectivamente $\ominus A$)
4. si t no es un punto de terminación, $X(t) = 1$ y $G(t)$ es un átomo q entonces t tiene exactamente un sucesor inmediato s en el árbol con $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y alguna de las siguientes condiciones es satisfecha:
   (a) $G(s) \rightarrow q \in \mathbf{P}(t)$ o
   (b) $\Box(G(s) \rightarrow q) \in \mathbf{P}(t)$

5. si t no es un punto de terminación, $X(t) = 1$ y $G(t)$ es un cuerpo q entonces t tiene exactamente un sucesor inmediato s en el árbol con $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y alguna de las siguientes condiciones es satisfecha:

   (a) $G(s) \rightarrow q \in \mathbf{P}(t)$ o

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

6. si t no es un punto de terminación, $X(t) = 0$ y $G(t)$ es un átomo q entonces t tiene $s_1, \ldots, s_k$ sucesores inmediatos en el árbol con $\mathbf{P}(s_i) = \mathbf{P}(t)$, $X(s_i) = 0$ y para algún m tal que $0 \leq m \leq k$ las cláusulas $G(s_i) \rightarrow q$ para $i \leq m$ y las cláusulas $\Box(G(s_i) \rightarrow q)$ para $m \leq i \leq k$ son exactamente todas las cláusulas de las formas mencionadas que pertenecen a $\mathbf{P}(t)$

7. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = A_1 \wedge A_2$, entonces t tiene exactamente dos sucesores inmediatos s_1 y s_2 con $\mathbf{P}(s_i) = \mathbf{P}(t)$, $X(s_i) = 1$ y $G(s_i) = A_i$

8. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = A_1 \wedge A_2$, entonces t tiene exactamente un sucesor inmediato s, y para algún $i \in \{1, 2\}$, $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s_i) = 0$ y $G(s_i) = A_i$

9. si t no es un punto de terminación y $G(t) = \neg A$, entonces t tiene exactamente un sucesor inmediato s, y $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1 - X(t)$ y $G(s) = A$

10. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \Diamond\!\!\!\!+ A$, entonces t tiene un único sucesor, s_1 y se da la siguiente condición:

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

    o bien tiene exactamente dos sucesores inmediatos: s_1, s_2 y alguna de las siguientes condiciones sucede:

    (a) $\mathbf{P}(s_1) = \mathbf{P}(t)$ y $X(s_1) = 1$ y para algún H

    i. $G(s_1) \rightarrow \Diamond\!\!\!\!+ H \in \mathbf{P}(t)$ y

    ii. $\mathbf{P}(s_2) = \{$ todas las *cláusulas always* de $\mathbf{P}(t)\} \cup \{H\} \cup \{\Diamond\!\!\!\!+ C | C$ es una *cláusula ordinaria* de $\mathbf{P}(t)\} \cup \{C | \oplus C$ es una *cláusula ordinaria* de $\mathbf{P}(t)\}$ y

    iii. $X(s_2) = 1$

    iv. $G(s_2) = A \vee \Diamond\!\!\!\!+ A$

    (b) ídem inciso 10a excepto que la condición $G(s_1) \rightarrow \Diamond\!\!\!\!+ H \in \mathbf{P}(t)$ es reemplazada por $\Box(G(s_1) \rightarrow \Diamond\!\!\!\!+ H) \in \mathbf{P}(t)$

    (c) ídem inciso 10b excepto que la condición $\Box(G(s_1) \rightarrow \Diamond\!\!\!\!+ H) \in \mathbf{P}(t)$ es reemplazada por $\Box(G(s_1) \rightarrow H) \in \mathbf{P}(t)$ y $G(s_2) = A \vee \Diamond\!\!\!\!+ A$ lo es por $G(s_2) = \Diamond\!\!\!\!+ A$

    (d) ídem inciso 10b excepto que la condición $\Box(G(s_1) \rightarrow \Diamond\!\!\!\!+ H) \in \mathbf{P}(t)$ es reemplazada por $\boxplus(G(s_1) \rightarrow \Diamond\!\!\!\!+ H) \in \mathbf{P}(t)$

    (e) ídem inciso 10c excepto que la condición $\Box(G(s_1) \rightarrow H) \in \mathbf{P}(t)$ es reemplazada por $\boxplus(G(s_1) \rightarrow H) \in \mathbf{P}(t)$

11. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \Diamond A$, entonces t tiene o un sucesor inmediato s_1 o tiene exactamente dos sucesores inmediatos: s_1, s_2 y sucede alguna de las condiciones del inciso 10, según la situación, a las cuales se les aplica la regla del espejo

12. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \Diamond A$, entonces t tiene $s_1, \ldots, s_k$ como sus sucesores inmediatos y para m, n tales que $1 \leq m \leq n \leq k$ y algunas fórmulas bien formadas $D_1, \ldots, D_k$, $D_i (i \leq m)$ son exactamente todas las cláusulas de la forma $\Box(B_i \rightarrow H_i)$ en $\mathbf{P}(t)$, $D(m+1), \ldots, D(n)$ son exactamente todas las cláusulas de la forma $\Box(B_i \rightarrow FH_i)$ en $\mathbf{P}(t)$, $D(n+1), \ldots, D(k)$ son exactamente todas las cláusulas de la forma $B_i \rightarrow FH_i$ en $\mathbf{P}(t)$, y para cada $1 \leq i \leq k$ alguna de las siguientes condiciones se cumple:

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

    (b) $\mathbf{P}(s_i) =$ {todas las *cláusulas always* de $\mathbf{P}(t)$} $\cup$ $\{H_i\} \cup$ {$\Diamond C | C$ es una *cláusula ordinaria* de $\mathbf{P}(t)$} $\cup$ {$C | \oplus C$ es una *cláusula ordinaria* de $\mathbf{P}(t)$},
    $X(s_i) = 0$,
    $G(s_i) = A \vee FA$, para $1 \leq i \leq k$, excepto cuando $1 \leq i \leq m$, en cuyo caso $G(s_i) = FA$

13. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \Diamond A$, entonces ocurren condiciones similares de terminación a las del inciso 12 aplicando la regla del espejo

14. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = A_1 \vee A_2$, entonces t tiene exactamente dos sucesores: s_1 y s_2 con $\mathbf{P}(s_i) = \mathbf{P}(t)$, $X(s_i) = 0$ y $G(s_i) = A_i$

15. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = A_1 \vee A_2$, entonces t tiene exactamente un sucesor s, $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y $G(s)$ es o bien A_1 o A_2

16. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \oplus A$, entonces t tiene dos sucesores inmediatos, s_1 y s_2, y se da alguna de las siguientes condiciones:

    (a) $\mathbf{P}(s_1) =$ {todas las *cláusulas always* de $\mathbf{P}(t)$} $\cup$ {$\Diamond C | C$ es una *cláusula ordinaria* de $\mathbf{P}(t)$} $\cup$ {$C | \oplus C$ es una *cláusula ordinaria* de $\mathbf{P}(t)$},
    $X(s_1) = 1$ y $G(s_1) = A$

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

    (c) $\mathbf{P}(s_1) = \mathbf{P}(t)$, $X(s_1) = 1$, y para algún H, $G(s_1) \rightarrow \oplus H \in \mathbf{P}(t)$ $\mathbf{P}(s_2) = \mathbf{P}(t)$, $X(s_2) = 1$ y $G(s_2) = A$

    (d) ídem inciso 16c excepto que la condición $G(s_1) \rightarrow \oplus H \in \mathbf{P}(t)$ es reemplazada por $\Box(G(s_1) \rightarrow \oplus H) \in \mathbf{P}(t)$

    (e) ídem inciso 16c excepto que la condición $G(s_1) \rightarrow \oplus H \in \mathbf{P}(t)$ es reemplazada por $\boxplus(G(s_1) \rightarrow H) \in \mathbf{P}(t)$

17. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \oplus A$, entonces t tiene un sucesor inmediato, s, y se da alguna de las condiciones del inciso 16 con $X(s) = 1$ reemplazado por $X(s) = 0$

18. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \ominus A$, entonces t tiene dos sucesores inmediatos,s_1,s_2, y se da alguna de las condiciones del inciso 16 aplicando la regla del espejo

19. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \ominus A$, entonces t tiene un sucesor inmediato, s, y sucede la condición del inciso 17 aplicando la regla del espejo

20. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \mathit{Until}(A, B)$, entonces t tiene un sucesor inmediato, s, y una de las siguientes condiciones sucede:

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

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

21. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \mathit{Until}(A, B)$, entonces t tiene un sucesor inmediato, s, y una de las siguientes condiciones sucede:

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

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

22. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \mathit{Since}(A, B)$, entonces t tiene un sucesor inmediato, s, y una de las siguientes condiciones sucede:

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

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

23. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \mathit{Since}(A, B)$, entonces t tiene un sucesor inmediato, s, y una de las siguientes condiciones sucede:

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

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

□

DEFINICIÓN **6** ([Gab87], página 224) $\mathbf{P}?G$ *tiene éxito* si $\mathbf{P}?G = 1$ tiene un árbol de computación finito. $\mathbf{P}?G$ *falla finitamente* si $\mathbf{P}?G = 0$ tiene un árbol de computación finito. ■

Otro aspecto interesante de la propuesta desarrollada en este artículo es que la utlización de la programación en lógica nos permite responder ciertas preguntas de importancia con facilidad. En lo que resta de la sección demostraremos que todo programa en ETP termina su computación. Inicialmente comenzaremos demostrando un resultado similar para Temporal Prolog complementando de esta manera la propuesta de Gabbay, quien no ha publicádo tal resultado.

En lo que resta de la sección utilizaremos la convención de denotar mediante $P(t)$, $t \geq 0$, el contenido de un programa escrito utilizando el lenguaje ETP. Esto debe entenderse como el contenido del programa luego de habersele agregado piezas de información. Por ejemplo, el programa dado inicialmente sería denotado mediante $P(0)$. Luego de utilizar las reglas de inferencia t veces se le puede haber adicionado t piezas de información y esto será indicado mediante $P(t)$.

LEMA **1** Sea $P(t)$, $t \geq 0$, un programa en Temporal Prolog, $P(t)$ siempre es un conjunto finito de clausulas. ■

Demostración **1** Sean $n1$ la cantidad de hechos, $n2$ la cantidad de Reglas Ordinarias, $n3$ la cantidad de Reglas Always, $k1$ la cantidad de hechos inferidos y $k2$ la cantidad de reglas inferidas. Si $P(0)$ es el programa inicial en Temporal Prolog entonces $|P(0)| = n_1 + n_2 + n_3$. Para calcular la cardinalidad de $P(t)$ se debe tener en cuenta un hecho adicional: la sintáxis de Temporal Prolog solo permite el uso del símbolo $\Box$ en el nivel más externo de una regla. Por lo tanto ninguna Regla Always puede ser el consecuente de una implicación y entonces no pueden ser deducidas por Modus Ponens. Como solo pueden aumentar la cantidad de hechos y de Reglas Ordinarias tenemos que $|P(t)| = (n_1 + k_1) + (n_2 + k_2) + n_3$, $k_1, k_2 \geq 0$. Las constantes k_1 y k_2 tienen una cota superior finita ya que cada vez que se utiliza una regla de inferencia es aplicada a una sola regla y una sola vez. □

TEOREMA **1** Todo árbol de computación etiquetado para Temporal Prolog tiene un conjunto finito de nodos. ■

Demostración **2**
Sabemos que cada paso se basa en los hechos y reglas que conforman $P(t)$:
1) $G(t)$ es un átomo, los casos análogos de los items 2, 3a o 4 en [Gab87], en cuyo caso deben inspeccionarse todos los hechos o todas las reglas de $|P(t)|$.
2) $G(t)$ es de la forma $A \wedge B$ peor caso: $x(t) = 1$ con número de hijos:2
3) $G(t)$ es de la forma $A \vee B$ peor caso: $x(t) = 0$ con número de hijos:2
4) $G(t)$ es de la forma $\neg A$ peor caso un único sucesor en el árbol
5) $G(t)$ es de la forma $\oplus A$ (analogamente para $\ominus A$), peor caso $x(t) = 0$ y debe verificarse que no puede deducirse de las clausulas de $P(t)$, casos análogos de los items 2, 3a o 4 en [Gab87].
En cada caso la cantidad de hijos en cada nodo forma un conjunto cuya cardinalidad es menor o igual a la de $P(t)$. Para cada consulta el árbol consistirá de un número finito de nodos en los cuales se tendrán en cuenta una cantidad finita de hechos y reglas (lema 1). □

LEMA **2** Sea $P(t)$, $t \geq 0$, un programa en Extended Temporal Prolog, $P(t)$ siempre es un conjunto finito de clausulas. ■

Demostración **3** Idem lema 1. □

TEOREMA **2** Todo árbol de computación etiquetado para Extended Temporal Prolog tiene un conjunto finito de nodos. ■

Demostración **4** Basta con extender la demostración anterior para considerar el operador agregado:

> Si $G(t)$ es de la forma $\oplus A$ (el resultado es análogo para $\ominus A$), peor caso $x(t) = 1$ con dos sucesores inmediatos uno de los cuales implica una búsqueda entre las reglas de $P(t)$.

Debe tenerse en cuenta que el tipo de consultas restringidas que utiliza los operadores $\mathcal{S}$ o $\mathcal{U}$ son traducidas a sus equivalentes utilizando los operadores antes considerados. Nuevamente el árbol consistirá de un número finito de nodos en los cuales se tendrán en cuenta una cantidad finita de hechos y reglas (lema 2). □

4 Consideraciones sobre la implementación

En esta sección además de presentar ciertas consideraciones relativas a la implementación, se indicarán las divergencias que existen entre nuestra propuesta y la de Gabbay. Es importante destacar que la implementación de las reglas de inferencia y los axiomas contituyen un aspecto clave en la eficiencia de este tipo de herramienta. Pero también se sabe que es uno de los aspectos más difíciles de lograr. Debido a lo crítico que es esta característica en este tipo de programas, se intentó ayudar a que esta tarea se desarrollara con más eficiencia. Para tal efecto se definieron reglas extras para que se encargaran de evitar los procesos de inferencia en casos trivales o de respuesta directa.

Un aspecto importante en el que nuestra popuesta difiere de la realizada por Gabbay es el siguiente. Si se cuenta con una meta de la forma $\oplus(A \wedge B)$ siendo A y B cláusulas ordinarias cualesquiera resulta intuitivamente claro que no es lo mismo que resolver $\oplus A \wedge \oplus B$. En la primera fórmula como el operador afecta a la fórmula conjunción se requiere

que A y B sean verdaderas en el mismo instante. En la segunda A y B estan afectadas por operadores distintos y entonces podrían no ser verdaderas en el mismo instante. El algoritmo brindado por Gabbay no tiene en cuenta este tipo de problemas, razón por la cual se agregó al algoritmo de la extensión propuesta y a la implementación, una regla especial para trabajar con ellos. La introducción de esta regla provocó que debiera agregarse a todos los incisos de la regla 10, excepto el primero, la condición $G \neq F_1 \wedge F_2 \wedge \ldots \wedge F_n$, donde la meta es $\diamondsuit G$. Como se mencionó antes se introdujo una nueva regla que anexada al algoritmo que hemos brindado permite responder consultas de la forma $\diamondsuit(A \wedge B)$:

> Si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \diamondsuit(A \wedge B)$, entonces t tiene exactamente dos sucesores inmediatos: s_1, s_2 y alguna de las siguientes condiciones sucede:
>
> 1. $\mathbf{P}(s_1) = \mathbf{P}(t)$ y $X(s_1) = 1$ y para algún H
>   (a) $G(s_1) \rightarrow \diamondsuit H \in \mathbf{P}(t)$ y
>   (b) $\mathbf{P}(s_2) =$ todas las *cláusulas always* de $\mathbf{P}(t) \cup \{H\} \cup \{\diamondsuit C | C$ es una *cláusula ordinaria* de $\mathbf{P}(t)\} \cup \{C | \oplus C$ es una *cláusula ordinaria* de $\mathbf{P}(t)\}$ y
>   (c) $X(s_2) = 1$
>   (d) $G(s_2) = \diamondsuit A$
> 2. ídem inciso 1 excepto que la condición $G(s_1) \rightarrow \diamondsuit H \in \mathbf{P}(t)$ es reemplazada por $\Box(G(s_1) \rightarrow \diamondsuit H) \in \mathbf{P}(t)$
> 3. ídem inciso 1 excepto que la condición $G(s_1) \rightarrow \diamondsuit H \in \mathbf{P}(t)$ es reemplazada por $\Box(G(s_1) \rightarrow H) \in \mathbf{P}(t)$
> 4. ídem inciso 1 excepto que la condición $G(s_1) \rightarrow \diamondsuit H \in \mathbf{P}(t)$ es reemplazada por $\boxplus(G(s_1) \rightarrow \diamondsuit H) \in \mathbf{P}(t)$
> 5. ídem inciso 3 excepto que la condición $\Box(G(s_1) \rightarrow H) \in \mathbf{P}(t)$ es reemplazada por $\boxplus(G(s_1) \rightarrow H) \in \mathbf{P}(t)$

Para metas de tipo $\diamondsuit(A \wedge B)$ además de la introducción de una nueva regla fue necesario agregar varios predicados. La razón de su existencia es la misma que fue expuesta anteriormente y su función es resolver situaciones triviales como lo es el hecho de que la meta forme parte de una conjunción. Por ejemplo, supongamos que la meta es $\diamondsuit(A \wedge B)$ y que en la base se tiene la siguiente cláusula $\diamondsuit(A \wedge C \wedge \ldots \wedge B)$. Sin estos agregados el algoritmo no tiene forma de encontrar $\diamondsuit(A \wedge B)$ aunque del programa lógico resulta evidente que la meta tiene una respuesta exitosa. Se puede hacer un razonamiento análogo al anterior para el operador $\diamondsuit$ utilizando la regla del espejo.

Relacionado a este hecho se hizo evidente la necesidad de predicados que hicieran el trabajo inverso, es decir, si se deduce algo a partir de una cláusula $\diamondsuit\phi$ se sabe que lo deducido y la cláusula que provocó su deducción están en el mismo momento de tiempo, esto es importante sobre todo si el operador que provoca la deducción es $\diamondsuit$ o $\diamondsuit$ debido a que si no se almacena la relación esta se pierde debido a la vaguedad del operador, en cambio con los otros operadores no hace falta almacenarlo ya que al ser más precisos la relación se mantiene a pesar de no figurar explícitamente en la base. La siguiente situación ejemplifica lo antes expuesto, si tenemos $\Box A \rightarrow B$ y $\diamondsuit A$, resulta claro que se deduce $\diamondsuit B$. Se sabe que A y B son verdaderas en el mismo momento del futuro, pero si no almacenamos $\diamondsuit(A \wedge B)$ en la base, la relación que existe entre A y B se pierde ya que el operador no hace referencia a un instante de tiempo presiso.

Otra divergencia de importancia radica en que se agregaron dos reglas que se utilizan para viajar por la recta temporal y cuya utilidad radica en permitir la siguiente estrategia: "*Si el algoritmo no puede resolver una consulta en el instante actual de ninguna de las*

formas previstas, entonces de ser posible un razonamiento diferente en el siguiente instante de tiempo o en el anterior, muévase a dicho instante de tiempo y transforme la consulta de manera acorde". Por ejemplo en casa de ser útil desplazarnos en la recta temporal al siguiente instante transformaremos la consulta A en $\ominus A$, de manera análoga si lo hacemos hacia el momento anterior la transfomaremos en $\oplus A$.
La posibilidad de obtener razonamiento diferente radica en contar en la base en el instante en el que se está tratando de resolver la consulta al menos una cláusula $\oplus A$ para desplazarse un instante hacia el futuro en la recta temporal o al menos una cláusula $\ominus A$ para hacerlo hacia el pasado.

5 Ejemplos

Los ejemplos que consideraremos a continuación incluyen algún ejemplo de los que Gabbay consideró para su algoritmo, además de ejemplos propios que se utilizaron para probar casos especiales como así también consultas básicas.

EJEMPLO **1** (Ejemplo 3.1, página 215 ([Gab87])) Este es un ejemplo simple donde solo se utilizan los operadores básicos. En ETP el ejemplo sería:

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

Respuesta a la consulta $\diamondsuit c$: *yes*. □

EJEMPLO **2** (Ejemplo 3.5, página 219 ([Gab87])) Este es un ejemplo donde se muestra la posibilidad de utilizar predicados, es decir, permite mostrar que la herramienta no fue construída para manejar proposiciones solamente. En ETP el ejemplo sería:

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

Respuesta a la consulta $\diamondsuit(b1(a) \wedge b1(b))$: *no*.
Nota: $a1$ y $b1$ indican predicados cualesquiera, X e Y representan variables. □

EJEMPLO **3** Este ejemplo muestra como es posible responder una consulta que involucra al operador *Since*, a pesar de que este no puede estar en la base de datos. En ETP el ejemplo sería:

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

Respuesta a la consulta *Since*(a, b): *yes*. □

Prolog ha sido utilizado como lenguaje de consulta para bases de datos. Análogamente, un lenguaje de programación en lógica temporal puede ser la base de un sistema lógico de consultas sobre bases de datos con alguna información temporal, tal como una base de datos histórica ([TCG+93]). Presentaremos ahora un ejemplo de aplicación real. Veamos como se verían en este nuevo sistema algunos de los ejemplos presentados por Abadi y Manna en [AM89], página 288. Tomemos el siguiente ejemplo.

EJEMPLO **4** Tenemos una base de datos con algunos hechos sobre la historia de una cierta empresa despues de cierto día inicial, tal como quién trabajó en que departamento y con qué salario. Nos gustaría obtener una lista de los empleados, X, con sus salarios, Y, que trabajaron en el departamento de juguetes despues del día inicial. Y que además hayan

trabajado allí mientras John fue el gerente. En lógica temporal simplemente escribimos la consulta:

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

suponiendo que los predicados *manager*, *in_department* y *salary* han sido definidos de manera acorde al programa. Se obtiene la salida deseada de la consulta expuesta.

Tomando las mismas suposiciones del enunciado además de considerar que el presente es el momento de inicio de actividades de la empresa y que la consulta se está realizando en ese instante de tiempo, en ETP la consulta se vería de la siguiente manera:

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

$$(\diamondplus(manager(John) \wedge in\_department(x, Toy) \wedge salary(x, y))$$

Observación: La disyunción es necesaria ya que asumimos $\diamondplus$ irreflexivo (ver 3.1) □

Ejemplo **5** De manera similar, podríamos querer saber en cuanto se ha incrementado el salario de John cuando el ascendió de vendedor a gerente. Suponiendo que el programa cuenta con la regla:

$$increase(X, Y) \leftarrow salary(X, Y_1), \oplus salary(X, Y_2), Yis(Y_2 - Y_1)$$

expresando que el incremento del salario de X del momento n al momento $n + 1$, es la diferencia entre el salario de X en momento n y el salario de X en momento $n + 1$, y la consulta:

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

Tomando las mismas suposiciones que para el ejemplo previo, el programa y la consulta se verían de la siguiente manera respectivamente:

$$increase(X, Y) \leftarrow salary(X, Y_1) \wedge \oplus salary(X, Y_2) \wedge Yis(Y_2 - Y_1)$$

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

$$(\diamondplus(salesman(John) \wedge \oplus manager(John) \wedge increase(John, Y)))$$

□

Ejemplo **6** Finalmente podríamos querer expresar que todos los empleados siempre fueron empleados:

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

Lo que en ETP se reexpresaría de manera equivalente como:

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

□

6 Conclusiones y trabajo futuro

Se ha presentado un lenguaje para programación en lógica temporal, el cual constituye una extensión a "Temporal Prolog". Dicha extensión ha sido implementado y aquí se han comentado algunas de sus características mas destacables. También se ha provisto al lector de un algoritmo y de una demostración de que el mismo siempre termina. Volviendo nuestra atención al lenguaje lógico temporal, su definición considera como operadores básicos a $\diamondplus$, $\diamondminus$, $\oplus$ y $\ominus$. Es importante destacar que se permite el uso de los operadores *Since* y *Until*

al usuario para facilitar la realización de ciertas consultas aunque esto no agrega poder expresivo ya que solo se permite su uso para reescribir en forma más cómoda consultas que también podrían escribirse con los operadores $\diamondsuit$, $\diamondsuit$, $\oplus$ y $\ominus$. Además la implementación está basada en la noción de árbol de computación etiquetado. Dicha noción fue definida por Gabbay en [Gab87]. Es importante mencionar que el lenguaje propuesto para realizar programación en lógica temporal constituye un recorte de la lógica temporal tomada como base, así como Prolog lo es del Cálculo de Predicados. La presentación de este lenguaje de programación nacido a través de una extensión a Prolog, presenta un avance al proveer un medio para resolver problemas influenciados por el tiempo de una manera más natural.

La extensión realizada es interesante no solo por la introducción de nuevos operadores, sino también por la solución a metas que el algoritmo original no consideraba. Es destacable el hecho de que la implementación puede ser obtenida requiriéndola a los autores o a través de [Cob98]. Una de las posibles mejoras a realizar es trasladar el manejo y control de los niveles utilizados a un meta-predicado externo (ver [SS86]). Esta mejora permitiría acortar varios predicados de la implementacion actual. Otra mejora posible consistiría en aumentar la eficiencia del mecanismo deductivo, es decir, de la implementación de las reglas de inferencia. Es, además importante mencionar que se encuentra en elaboración la definición de técnicas de testing sistemático para este tipo de programas.

Agradecimientos

Deseamos agradecer a los miembros del GIRIT, Marisa Sanchez y María Mercedes Vitturini por la colaboración brindada en la elaboración de este artículo.

Referencias

[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.