Programación en Lógica Temporal Basada en los operadores Since y Until

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

Debido al creciente interés en diversas áreas de las Ciencias de la Computación en el estudio y manejo de nociones temporales, en los últimos años se han propuesto lenguajes de programación en lógica temporal. A pesar de esta diversidad de propuestas no hay implementaciones de las mismas accesibles públicamente. El objetivo del presente trabajo es explicar los aspectos salientes de un nuevo lenguaje de programación en lógica temporal, que utiliza los operadores *Since* y *Until*, tanto en sus fundamentos como en lo concerniente a su implementación. Este nuevo lenguaje de programación, denominado SU-TP, se basa en el aspecto lógico en una lógica temporal presentada en [BFG$^+$96] y en el aspecto algorítmico en una propuesta realizada por Dov Gabbay en [Gab87] para obtener una extensión a *Prolog* [SS86] denominada "*Temporal Prolog*".

La novedad de este trabajo con respecto a la presentación de Gabbay está en que se utilizan como operadores temporales básicos un conjunto de operadores más expresivos sobre estructuras lineales, que los que se tomaron como básicos en la mencionada propuesta. Además se ha demostrado que este conjunto de operadores es completo sobre la clase de flujos lineales isomorfos a los enteros o a los reales (ver [Kam68] y [GHR87]). Si bien por limitaciones de espacio no se incluirá aquí el código, el mismo puede obtenerse de [Cob98]. No obstante explicaremos detalladamente los fundamentos lógicos de la propuesta, el algoritmo utilzado como base en la implementación y se brindarán ejemplos de su uso. Finalmente se realizarán comentarios relativos a la implementación tanto el lo que respecta a problemas solucionados como a aquellos aspectos que podrían constituir futuras mejoras. También se demostrará que el algoritmo computa una función total (siempre se detiene). Otro punto que es importante resaltar es que si bien no se conocen implementaciones de este tipo de lenguajes, como se mencionó previamente, este utiliza los operadores *Since* y *Until* usualmente citados en determinadas clases de problemas. Algunas áreas en las que se han propuesto lógicas temporales basadas en estos operadores incluyen: análisis de correctitud de programas, bases de datos deductivas temporales y generación de documentos multimediales ([Gal87]).

[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).

Programación en Lógica Temporal Basada en los operadores Since y Until

1 Introducción

Para solucionar problemas en las diversas áreas de las Ciencias de la Computación donde el tiempo es necesario es conveniente disponer de un lenguaje de programación idóneo para representar cómo los sistemas cambian a través del tiempo En la última década han surgido una serie de propuestas destinadas a atenuar la falta de una "programación en lógica temporal" (Ver [OM94]).

Los operadores temporales más reconocidos del área son $\Diamond^+$ y $\Diamond^-$ que corresponden a los operadores F y P de Prior respectivamente, debido a que se han tomado como básicos para muchos de los desarrollos que se han hecho; uno de ellos fue el algoritmo presentado en [Gab87]. Por otro lado fue demostrado en [Kam68] que los operadores *Since* y *Until* son más expresivos que los operadores $\Diamond^+$ y $\Diamond^-$ sobre estructuras lineales, lo cual justifica la variante que los utiliza como operadores básicos. Consideraremos entonces un lenguaje de programación en lógica temporal basado en *Since* y *Until*, mencionando aspectos de su implementación . Este lenguaje toma las bases en la faz algorítmica del trabajo realizado por Gabbay, pero la base de su lógica esta inspirada en aquella utilizada en [BFG$^+$96].

Este trabajo está organizado como sigue. Se presenta la lógica temporal que sirvió como base de este lenguaje y la definición del mismo en la primer sección; en la siguiente sección se comentarán los aspectos más destacados de su implementación. Luego se brindarán ejemplos a fin de ilustrar su uso. En la útima sección brindaremos nuestras conclusiones y sugerimos futuras lineas de investigación. Por razones de espacio no brindamos aquí una exposición del código.

2 Una lógica basada en Since - Until

Se puede probar que los operadores *Since* y *Until* poseen mayor expresividad que los operadores $\Diamond^-$ y $\Diamond^+$ presentados por Prior y utilizados como base de "Temporal Prolog"por Gabbay. Esta diferencia en la expresividad fue demostrada por primera vez en [Kam68](un texto más moderno es [GHR94]).

En primer término brindaremos la lógica que constituirá la base de la herramienta a presentar. Es importante hacer notar que la misma está inspirada en la lógica presentada en [BFG$^+$96] más específicamente en la sección 1.4.5 del mismo. Luego se presenta el lenguaje a implementar y el algoritmo utilizado. Posteriormente se brindan consideraciones de la implementación y su documentación en un apéndice. Finalmente mostramos su comportamiento a través de ejemplos.

2.1 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, aquí será necesario considerar dos tipos:

> *Operadores reflexivos o irreflexivos*: 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.

Operadores débiles o fuertes: un operador fuerte 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")
- $\diamondsuit\,\phi$: "Alguna vez en el futuro ϕ" irreflexivo.
- $\diamondsuit\,\phi$: "Alguna vez en el pasado ϕ" irreflexivo.
- $\Diamond\,\phi$: $\diamondsuit\phi \vee \phi \vee \diamondsuit\phi$ ("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.

2.2 La lógica temporal y su algoritmo

Ahora que se han presentado 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.

Definición **1** La lógica temporal, que se utilizaremos, tiene los siguientes axiomas y reglas, en el lenguaje con *Since*, *Until*, $\oplus$ y $\ominus$. Veamos las reglas de inferencia y axiomas de la mencionada lógica.

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

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

$$\frac{\vdash X \to (B \vee (A \wedge \oplus X))}{\vdash X \to Until(A, B)} \quad ; \quad \frac{\vdash X \to (B \vee (A \wedge \ominus X))}{\vdash X \to Since(A, B)}$$

3. Los siguientes esquemas de axioma:

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

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

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

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

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

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

$$Until(A, B) \leftrightarrow (B \vee (A \wedge \oplus(Until(A, B)))$$

$$Since(A, B) \leftrightarrow (B \vee (A \wedge \ominus(Since(A, B)))$$

4. Los siguientes esquemas de axioma denotando el uso de una estructura temporal lineal:
   Nota: El símbolo $\mathbf{T}$ representa la constante "true".

$$Until(\mathbf{T}, A) \wedge Until(\mathbf{T}, B) \rightarrow$$
$$(Until(\mathbf{T}, A \wedge Until(\mathbf{T}, B)) \vee Until(\mathbf{T}, Until(\mathbf{T} \wedge B, A)) \vee Until(\mathbf{T}, A \wedge B))$$

$$Since(\mathbf{T}, A) \wedge Since(\mathbf{T}, B) \rightarrow$$
$$(Since(\mathbf{T}, A \wedge Since(\mathbf{T}, B)) \vee Since(\mathbf{T}, Since(\mathbf{T} \wedge B, A)) \vee Since(\mathbf{T}, A \wedge B))$$

■

Ahora se introducirá la definición del lenguaje de programación en lógica temporal que se basa en la lógica recien presentada.

DEFINICIÓN **2** Considere un lenguaje con átomos proposicionales, los conectivos booleanos: $\wedge$, $\vee$, $\rightarrow$, $\neg$ y los operadores temporales: $\Diamond$, $\diamondminus$, $\minusdiamond$, $\Box$, $\boxminus$, $\minusbox$, $\oplus$, $\ominus$, *Since*, *Until*. Definimos las nociones de *programa, cláusula, cabeza, cuerpo* y *meta*.

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

Una *cláusula* 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 $Since(A, B)$ o $Until(A, B)$ o $\oplus A$ o $\ominus A$, donde A y B son conjunciones de *cláusulas*.

Un *cuerpo* es o una fórmula atómica o una conjunción de cuerpos, $Since(A, B)$ o $Until(A, B)$ o $\oplus A$ o $\ominus A$ o $\neg A$, donde A y B son *cuerpos*.

Una *meta* es cualquier *cuerpo* o $\Diamond A$ o $\Box A$ o $\boxminus A$ o $\boxplus A$ o $\diamondminus A$ o $\minusdiamond A$, donde A es un *cuerpo*.

■

Para facilitar futuras referencias al lenguaje recién definido de aquí en adelante lo llamaremos SU-TP, abreviatura de "*Since-Until Temporal Prolog*". Se pueden observar a continuación las ideas básicas de la futura implementación, que se encuentran en la definición de árbol de computación etiquetado y el algoritmo detallado.

DEFINICIÓN **3** 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.

■

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 $X(t) = 1$, $G(t)$ es un átomo q y $q \in \mathbf{P}(t)$
3. si $t \in T$ es un punto de terminación y $X(t) = 0$ entonces
   (a) $G(t)$ es un átomo q y q no es la cabeza de una cláusula o
   (b) $G(t)$ 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 $G(s) \rightarrow q \in \mathbf{P}(t)$
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 $G(s) \rightarrow q \in \mathbf{P}(t)$
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 $\mathbf{P}(s_i) = \mathbf{P}(t)$, $X(s_i) = 0$ con $G(s_i) \rightarrow q$ para $0 \leq i \leq k$ y esas son exactamente todas las cláusulas de la forma mencionada 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 $G(s) = A$, $\mathbf{P}(s) = \mathbf{P}(t)$ y $X(s) = 1 - X(t)$
10. 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 $\mathbf{P}(s_i) = \mathbf{P}(t)$ y $G(s_i) = A_i$ y $X(s_i) = 0$
11. 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, $\mathbf{P}(s) = \mathbf{P}(t)$, $G(s)$ es o bien A_1 o A_2 y $X(s_i) = 1$
12. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \oplus A$, entonces t tiene un sucesor inmediato, s_1 y se da la siguiente condición:

    $\mathbf{P}(s_1) =$ {todas las *cláusulas always* de $\mathbf{P}(t)$} $\cup$
    {$\ominus C | C$ es una *cláusula* de $\mathbf{P}(t)$} $\cup$
    {$C | \oplus C$ es una *cláusula* de $\mathbf{P}(t)$},

    $X(s_1) = 1$ y $G(s_1) = A$ o
    tiene dos sucesores inmediatos,s_1,s_2, y se da la siguiente condición: $\mathbf{P}(s_1) = \mathbf{P}(t)$, $X(s_1) = 1$, 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 $G(s_2) = A$

13. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \oplus A$, entonces t tiene o un sucesor, s_1, o dos sucesores inmediatos,s_1 y s_2, y se da alguna de las condiciones del inciso 12 según corresponda, con $X(s) = 1$ reemplazado por $X(s) = 0$

14. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \ominus A$, entonces t tiene uno o dos sucesores inmediatos,s_1 o s_1,s_2, y se da alguna de las condiciones del inciso 12 aplicando la regla del espejo

15. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \ominus A$, entonces t tiene uno o dos sucesores inmediatos,s_0 o s_1, $s2$ y sucede alguna de las condiciones del inciso 13 aplicando la regla del espejo

16. 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 la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y $G(s) = B \vee (A \wedge \oplus(\mathit{Until}(A, B)))$

17. 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 la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ y $G(s) = B \vee (A \wedge \oplus(\mathit{Until}(A, B)))$

18. 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 la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y $G(s) = B \vee (A \wedge \ominus(\mathit{Since}(A, B)))$

19. 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 la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ y $G(s) = B \vee (A \wedge \ominus(\mathit{Since}(A, B)))$

20. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \Diamond\!\!\!\!+ A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y $G(s) = \oplus(\mathit{Until}(\mathit{true}, A))$

21. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \Diamond\!\!\!\!+ A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ y $G(s) = \oplus(\mathit{Until}(\mathit{true}, A))$

22. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \Diamond\!\!\!\!- A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y $G(s) = \ominus(\mathit{Since}(\mathit{true}, A))$

23. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \Diamond\!\!\!\!- A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ y $G(s) = \ominus(\mathit{Since}(\mathit{true}, A))$

24. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \Diamond A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y $G(s) = A \vee \Diamond\!\!\!\!+ A \vee \Diamond\!\!\!\!- A$

25. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \Diamond A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ y $G(s) = A \vee \Diamond\!\!\!\!+ A \vee \Diamond\!\!\!\!- A$

26. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \boxplus A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y $G(s) = \neg(\Diamond\!\!\!\!+ \neg A)$

27. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \boxplus A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ y $G(s) = \neg(\diamondsuit\neg A)$

28. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \boxminus A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y $G(s) = \neg(\diamondsuit\neg A)$

29. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \boxminus A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ y $G(s) = \neg(\diamondsuit\neg A)$

30. si t no es un punto de terminación, $X(t) = 1$ y $G(t) = \Box A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 1$ y $G(s) = A \vee \boxplus A \vee \boxminus A$

31. si t no es un punto de terminación, $X(t) = 0$ y $G(t) = \Box A$, entonces t tiene un sucesor inmediato, s, y la siguiente condición sucede: $\mathbf{P}(s) = \mathbf{P}(t)$, $X(s) = 0$ y $G(s) = A \vee \boxplus A \vee \boxminus A$

□

DEFINICIÓN **4** $\mathbf{P}?G$ *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.

■

Al igual que para ETP [CA98] demostraremos que el algoritmo considerado en la implementación de SU-TP también termina su computación. 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 $P(t)$. En primer lugar recordaremos el siguiente resultado

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

Demostración **1** Demostrado en [CA98] □

Luego, respecto de la construcción del árbol de computación etiquetado tenemos lo siguiente:

LEMA **2** 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) 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. 6) Si $G(t)$ es de la forma $\mathcal{U}A$ (el resultado es análogo para $\mathcal{S}A$), peor caso $x(t) = 1$ con dos sucesores inmediatos.

En cada caso la cantidad de hijos en cada nodo forma un conjunto cuya cardinalidad es menor o igual a la de $P(t)$. □

Debe recordarse que $\diamondsuit$, $\diamondsuit$, $\boxplus$ y $\boxminus$ son definidos utilizando los operadores básicos.

COROLARIO **1** Todo programa en Temporal Prolog termina. ■

Demostración **3** Para cada consulta el árbol consistirá de un número finito de nodos (por lema 2) en los cuales se tendrán en cuenta una cantidad finita de hechos y reglas (lema 1). □

3 Consideraciones de la implementación

Uno de los aspectos en que el algoritmo utilizado para la implementación del interprete de SU-TP también difiere del utilizado para implementar el intérprete de ETP es que fué necesario agregarle dos reglas que se utilizan para viajar por la recta temporal y cuya utilidad radica en el siguiente razonamiento: *"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"* . Esto es, si necesita posicionarse en el próximo instante de la recta temporal transforme la consulta A en $\ominus A$, de manera análoga si lo hace hacia el momento anterior, transfórmela en $\oplus A$.

La posibilidad de obtener razonamiento diferente radica en registrar en la base en el instante en el que se está tratando de resolver la consulta, al menos con una cláusula $\oplus$ para moverse un instante hacia el futuro o al menos con una $\ominus$ para hacerlo hacia el pasado.

Es importante notar que la performance puede ser notablemente mejorada implementando en forma más eficiente el predicado encargado de aumentar la base. Este predicado implementa las reglas de inferencia y parte de los axiomas, logrando que solo aumente lo que sea de utilidad a la meta que se está tratando de resolver. Tal como está implementado actualmente el predicado realiza la clausura deductiva respecto a la información asociada a un instante particular. Debe tenerse en cuenta que el mismo es un un problema clásico y abierto de los sistemas deductivos.

Se agregó una regla al algoritmo para tener la posibilidad de contar con una constante **T** que se interprete como una verdad eterna en el sistema. Este agregado resulta en una verdad eterna dado que en particular para cada instante considerado será verdadera cualquier tautología, por ejemplo $A \vee \neg A$. Dado que se considera una estructura temporal lineal ilimitada, la constante **T** estará en todo momento del pasado y del futuro, cualquiera sea el momento "presente" en el que el sistema se encuentre.

Al igual que en el algoritmo para la implementación de ETP [CA98] fué necesario implementar una estrategia para solucionar el caso particular de tener

$$\diamondsuit(\text{Cláusula 1} \wedge \text{ Cláusula 2})$$

y asi considerar un tipo de meta para la cual el algoritmo de Gabbay no provee una respuesta adecuada. Metas como la anterior presentaron el problema de que en general no se pueden demostrar con el esquema:

$$\diamondsuit\text{Cláusula 1} \wedge \diamondsuit \text{ Cláusula 2}$$

No obstante en determinados casos sí es deducible y es posiblemente la única manera de resolver la meta y es el caso en el cual la segunda cláusula no tiene una dependencia

temporal con la primera, es decir tiene una referencia temporal vaga. Un ejemplo donde se da esta situación es el siguiente. Si se tiene la siguiente base:

$Until(a, b)$,
$\neg b$,
$Since(a, b)$

Y la siguiente consulta: $\diamondminus(b \wedge \diamondplus b)$.
Se puede observar que la consulta se verifica en la base de conocimiento dada, sin embargo si no se agrega la regla especial que se mencionó con anterioridad, esta consulta será respondida en forma negativa por el algoritmo. La razón es que el algoritmo tiene previsto transformar este tipo de consulta en:

$$Since(true, b \wedge Until(true, b))$$

que no puede demostrarse a partir de la base de datos ya que se desconoce en que momento b se hace verdadero. En cambio de acuerdo a la estrategia sugerida existe una demostración para

$$Since(true, b)$$

y existe una para

$$Since(true, Until(true, b))$$

y por lo tanto existe una para la consulta total.

Se realizó un razonamiento análogo por aplicación de la regla del espejo para responder consultas del tipo:

$$\diamondminus(\text{Cáusula 1} \wedge \text{ Cáusula 2})$$

4 Ejemplos

Veamos algunos ejemplos de interés, a fin de mostrar el funcionamiento de la herramienta aquí presentada.

Ejemplo **1**

Base b: { a ,
$\ominus(a)$,
$\ominus(\ominus(b))$,
$\oplus(a)$,
$\oplus(\oplus(a))$,
$\oplus(\oplus(\oplus(c)))$,
$\oplus(\oplus(a)) \rightarrow \oplus(Until(d, e))$ }
Respuesta a la consulta $\diamondplus(Until(d, e) \wedge \diamondminus(b \wedge \oplus(Until(a, c))))$: *yes*. □

Ejemplo **2**

Base b: { a ,
$\ominus(a)$,
$\ominus(\ominus(b))$,
$\oplus(a)$,
$\oplus(\oplus(a))$,
$\oplus(\oplus(\oplus(c)))$,
$\oplus(\oplus(a)) \rightarrow \oplus(Until(d, e))$ }
Respuesta a la consulta $\diamondminus(b \wedge \oplus(Until(a, c)))$: *yes*. □

Ejemplo **3**

Base b: { $\oplus c$,
a ,
$\ominus a$,
$\ominus(\ominus b)$ }

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

Veamos algunos ejemplos de aplicaciones más concretas como algunas propiedades de programas en ejecución, las cuales son muy útiles en el testeo de sistemas. Por ejemplo podemos considerar si cierta propiedad se mantiene a lo largo de toda la computación, si alguna vez una propiedad dada se verifica o la persistencia de alguna característica. En [MP90] se brinda una clasificación de esquemas de fórmulas temporales que son de interés en la consideración del posible comportamiento de un programa. Los ejemplos que presentamos a continuación expresan condiciones relativas a la ejecución de programas con especificación de concurrencia.

Ejemplo **4** Sea la propiedad: "el programa eventualemente alcanza su punto de terminación."

En SU-TP si se tratara de una consulta la propiedad podría ser reescrita como:

$$\diamondsuit alfinal$$

en cambio si se requiere como una cláusula de un programa sería:

$$\oplus(\mathit{Until}(\mathbf{T}, alfinal))$$

Observación: El operador $\oplus$ es necesario debido a que nos estamos manejando con la versión irreflexiva del operador $\diamondsuit$ y como el operador *Until* que utilizamos es reflexivo no obtendriamos una reescritura apropiada de $\diamondsuit$ □

Ejemplo **5** Sea la propiedad: "los procesos Q1 y Q2 nunca estarán simultáneamente en la misma sección critica."

En SU-TP si se tratara de una consulta la propiedad podría ser reescrita como:

$$\boxplus\neg(Q1enseccioncritica \wedge Q2enseccioncritica)$$

en cambio si se requiere como una cláusula de un programa sería:

$$\neg(\oplus(\mathit{Until}(\mathbf{T}, (enseccioncritica1 \wedge enseccioncritica2))))$$

□

Ejemplo **6** Sea la propiedad: "si el sistema comienza en determinado estado relativo a las variables x e y, se alcanzará un estado de terminación en el cual se verifican ciertas condiciones, en este caso relativas a z."

En SU-TP como una cláusula de un programa en el lenguaje de programación dado, podría ser reescrita como:

$$inicio \wedge (x = y) \rightarrow \oplus(\mathit{Until}(true, alfinal \rightarrow (z = y)))$$

Observación: se debió escribir así debido a las restricciones del lenguaje para el usuario es equivalente a: $inicio \wedge (x = y) \rightarrow \diamondsuit(alfinal \rightarrow (z = y))$

□

5 Conclusiones y trabajo futuro

Se ha considerado un lenguaje lógico temporal, SU-TP, del cual hemos ofrecido sus fundamentos lógicos, su definición, un algoritmo con el cual se ha implementado su intérprete y ejemplos de su funcionamiento. Anteriormente hemos propuesto otro lenguaje de este tipo, llamado ETP (ver [CA98]). Ambos lenguajes permiten al usuario manejarse con el mismo conjunto de operadores. La diferencia fundamental entre ambos lenguajes es que en ETP se consideran como operadores básicos a $\diamondplus$, $\diamondminus$, $\oplus$ y $\ominus$, mientras que en el presentado en este trabajo, se toma un conjunto de operadores básicos diferente: *Since*, *Until*, $\oplus$ y $\ominus$. Un aspecto importante de esta diferencia radica en las distintas expresividades obtenidas, dado que fue demostrado en [Kam68] que los operadores *Since* y *Until* son más expresivos que $\diamondplus$ y $\diamondminus$, sobre estructuras lineales como las aquí consideradas.

La implementación del lenguaje considerado está basada en la noción de árbol de computación etiquetado. Dicha noción fue definida por Gabbay en [Gab87]. El algoritmo también está inspirado en el presentado en [Gab87] para "*Temporal Prolog*", pero aquí además incluímos una demostración de que el algoritmo que utilizamos termina. Es importante mencionar que la implementación constituye un recorte de la lógica temporal tomada como base, así como Prolog lo es del Cálculo de Predicados. La presentación de estos lenguajes de programación nacidos a través de una extensión al Prolog tradicional, presentan un avance al proveer un medio para resolver problemas influenciados por el tiempo de una manera más natural. Es destacable el hecho de que la implementación se encuentran a disposición, a pesar que aún quedan mejoras por hacer. Una 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 testeo sistemático para este tipo de programas.

Agradecimientos

Deseamos agradecer a los miembros del GIRIT, Marisa Sanchez y María Mercedes Vitturini por su contribución al mejoramiento de versiones preliminares de este artículo.

A Documentación de los predicados de la implementación

Definición operadores lógicos.

`:- op(194,xfy,^ ).`	Operador de conjunción($\wedge$).
`:- op(195,xfy,v).`	Operador de disyunción($\vee$).
`:- op(193,fx,no).`	Operador de negación($\neg$).
`:- op(193,yfx,- >).`	Operador de implicación($\rightarrow$).

Definición operadores temporales.

`:- op(280,fx,#).`	Operador "Siempre"($\Box$).
`:- op(180,fx,#>).`	Operador "Siempre en el Futuro"($\boxplus$).
`:- op(180,fx,<#).`	Operador "Siempre en el Pasado"($\boxminus$).
`:- op(180,fx,<>).`	Operador "Alguna Vez"($\Diamond$).
`:- op(180,fx,>>).`	Operador "Alguna Vez en el Futuro"($\diamondplus$).
`:- op(180,fx,<<).`	Operador "Alguna Vez en el Pasado"($\diamondminus$).
`:- op(180,fx,prev).`	Operador "Previous"($\ominus$).
`:- op(180,fx,next).`	Operador "Next"($\oplus$).
`:- op(180,xfy,since(A,B)).`	Operador "Since"(*Since*).
`:- op(180,xfy,until(A,B)).`	Operador "Until"(*Until*).

Es importante notar que los parámetros de entrada al predicado se distinguirán precediéndolos con el signo "+", por ejemplo: +(Nombre del parámetro) y los de salida con el signo "−", por ejemplo: −(Nombre del parámetro).

1. Predicado `resolver`:
   - Sintáxis: `resolver`(+Meta,+Objetivo).
   - Utilidad: Es el predicado principal, prepara el sistema para una consulta, obtiene la meta y el objetivo del usuario y se los da al predicado `algoritmo`, para luego darle la respuesta al usuario y dejar el sistema listo para otra consulta.
2. Predicado `algoritmo`:
   - Sintáxis: `algoritmo`(+Meta,+Objetivo,+Instante,+Nivel).
   - Utilidad: Es el predicado que implementa el algoritmo teórico con la extensiones que resultaron necesarias.
3. Predicado `actualizar`.
   - Sintáxis: `actualizar`(+Cláusula,+Nivel Actual,+Instante Actual, -Próximo Nivel,-Próximo Instante)
   - Utilidad: La finalidad de este predicado es obtener el próximo instante y nivel sea cual sea el sentido de la modificación temporal, esto es así tanto para aquellas computaciones que se mueven hacia el pasado o hacia el futuro. Además de obtener a través de la llamada a `cambiar_base` la situación de la base en ese nuevo instante.
4. Predicado `chequear`.
   - Sintáxis: `chequear`(+Meta,+Objetivo,+Nivel,+Vez)

- Utilidad: El predicado se encarga de controlar que no se intente resolver la misma meta mas de un número máximo establecido de veces.

5. Predicado `insertar`.
   - Sintáxis: `insertar`(+Meta,+Objetivo,+Instante,+Nivel)
   - Utilidad: Inserta en la base auxilar de metas el parámetro Meta con el resto de la información de la misma pasada como parámetros más la vez que se intenta resolver la misma.
6. Predicado `mayor_v`.
   - Sintáxis: `mayor_v`(+Meta,+Objetivo,+Vez,-Ultima Vez)
   - Utilidad: Obtiene la cantidad de veces que se intentó resolver el parámetro Meta, con el objetivo dado.
7. Predicado `no_demostrable`.
   - Sintáxis: `no_demostrable`(+Meta)
   - Utilidad: Decide si la Meta no puede demostrarse, debido a que no existe en la base nada que la involucre.
8. Predicado `descomp`.
   - Sintáxis: `descomp`(+Meta,-Lista)
   - Utilidad: Descompone la fórmula dada en las letras que la componen, devolviendo las mismas en una lista.
9. Predicado `no_pertenecen`.
   - Sintáxis: `no_pertenecen`(+Lista,+Letra).
   - Utilidad: Determina si una dada letra no pertenece a la lista dada.
10. Predicado `dem_trivial`.
    - Sintáxis: `dem_trivial`(+Meta,+Instante,+Nivel)
    - Utilidad: Determina el predicado si existe alguna demostración que surja trivialmente de la base para la meta dada.
11. Predicado `aument`.
    - Sintáxis: `aument`(+Pasado_o_Futuro,+Instante,+Nivel)
    - Utilidad: Su función es ir agregando cláusulas que se obtengan por aplicación de alguna regla de inferencia, lo realizará mientras se puedan agregar cláusulas diferentes a las que ya están en la base.
12. Predicado `aumentar_b`.
    - Sintáxis: `aumentar_b`(+Pasado_o_Futuro,+Instante,+Nivel)
    - Utilidad: Agrega a la base todas las cláusulas que surgen de la aplicación de Modus Ponens, desglosar conjunciones, eliminar operadores redundantes, etc.
13. Predicado `controlar_antecedente`.
    - Sintáxis: `controlar_antecedente`(+Cláusula,+Instante,+Nivel)

- Utilidad: El predicado se encarga de ver que la cláusula esté en forma efectiva en la base de datos.

14. Predicado `agregar`.

    - Sintáxis: `agregar`(+Cláusula,+Instante,+Nivel)
    - Utilidad: Verifica si la cláusula está en la base de conocimiento si no está, la agrega a la base.

15. Predicado `concatenar`.

    - Sintáxis: `concatenar`(+Lista1,+Lista2,-Lista)
    - Utilidad: Construye una lista que resulta de la concatenación de las dos listas pasadas como parámetro.

16. Predicado `miembro`.

    - Sintáxis: `miembro`(+Elemento,+Lista)
    - Utilidad: Verifica si el elemento pasado como parámetro forma parte de la lista dada.

17. Predicado `cambiar_base`.

    - Sintáxis: `cambiar_base`(+Pasado_o_Futuro,+Cláusula,+Instante, +Nivel)
    - Utilidad: Agrega la cláusula dada a la base y solicita la modificación de la base de manera acorde al parámetro Pasado_o_Futuro.

18. Predicado `cambiar`.

    - Sintáxis: `cambiar`(+Pasado_o_Futuro,+Instante,+Nivel)
    - Utilidad: Su función es cambiar la base mientras se agreguen cláusulas nuevas a la misma.

19. Predicado `c_base`.

    - Sintáxis: `c_base`(+Pasado_o_Futuro,+Instante,+Nivel)
    - Utilidad: Cambia todas las cláusulas del instante anterior al instante dado al pasado si el parámetro Pasado_o_Futuro es igual a pasado(<<), o las del instante siguiente al dado hacia el futuro si el parámetro tiene valor futuro(>>).

20. Predicado `crear_base_de_trabajo`.

    - Sintáxis: `crear_base_de_trabajo`
    - Utilidad: Crea la base bt sobre la que trabajará el algoritmo. Supone la existencia de una base de conocimiento dada. Se diferencia de esta última en el hecho que en bt cada cláusula tiene asociado el instante donde es válida y el nivel al que pertenece.

21. Predicado `borrar_base_de_trabajo`.

    - Sintáxis: `borrar_base_de_trabajo`
    - Utilidad: Destruye las bases utilizadas para resolver la meta propuesta.

Referencias

[BFG+96] H. Barringer, M. Fisher, D. Gabbay, R. Owens, and M. Reynolds. *The Imperative Future: Principles of Executable Temporal Logic*. Research Studies Press Ltd., 1996.

[CA98] María Laura Cobo and Juan Carlos Augusto. Fundamentos lógicos e implementación de una extensión a temporal prolog. Enviado para publicación al Congreso Argentino de Ciencias de la Computación, CACiC98, 1998.

[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.

[Gal87] Antony Galton. *Temporal Logics and their Applications*. Academic Press, San Diego, California, 1987.

[GHR87] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal Logic Mathematical Foundations and Computational Aspects*. Oxford University Press, 1987.

[GHR94] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal Logic Mathematical Foundations and Computational Aspects*. Oxford University Press, 1994.

[Kam68] J. A. W. Kamp. *On Tense Logic and the Theory of Order*. PhD thesis, University of California - Los Angeles, 1968.

[MP90] Z. Manna and A. Pnuelli. A hierarchy of temporal properties. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 377–408. ACM Press, 1990.

[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.