Facultad de Ingeniería – Instituto de Computación

COMPILADORES DE GRAFOS TEMPORIZADOS

Informe

Sebastián Bello

Tutor: Luis Sierra

Proyecto de Taller V - 2000

INDICE

INDICE DE FIGURAS

Nota: este documento está incluido en un CD que contiene

- Compiladores de Grafos Temporizados – Informe (este documento)
- Compiladores de Grafos Temporizados – Manual del Sistema
- Compiladores de Grafos Temporizados – Manual del Usuario
- GraflexedH – Manual del Usuario
- GraflexedK – Manual del Usuario
- GraflexedU – Manual del Usuario
- Compiladores de Grafos Temporizados – Fuentes y binarios
- Editores de Grafos Temporizados – Fuentes y binarios
- Ejemplos

Proyecto de Taller V
Instituto de Computación
Facultad de Ingeniería
Marzo 2000

Compiladores de grafos temporizados

Resumen

Los sistemas de tiempo real son sistemas que toman en cuenta la evolución del tiempo, y permiten expresar propiedades temporales cuantitativas. En este tipo de sistemas es posible formular expresiones del tipo "toda ejecución de la acción A ocurre al menos 20 segundos después que la ejecución de una acción B". Distintos lenguajes de especificación para estos sistemas han sido propuestos. El objetivo de este taller es la construcción de una serie de compiladores que conviertan especificaciones -tanto de sistemas como de propiedades a verificar- entre un conjunto de formalismos elegido. Ellos serán los usados por las herramientas Kronos, Uppaal y HyTech. Se han desarrollado además interfases gráficas para el uso de dichos formalismos –todos ellos se basan en autómatas-.

1. Introducción

Los **sistemas de tiempo real** son sistemas que toman en cuenta la evolución del tiempo, y permiten expresar propiedades temporales **cuantitativas**[Hei96]. En este tipo de sistemas es posible formular expresiones del tipo "toda ejecución de la acción A ocurre al menos 20 segundos después que la ejecución de una acción B". Distintos lenguajes de especificación para estos sistemas han sido propuestos, distinguiéndose claramente al menos tres enfoques: extensiones de autómatas[Alu94], extensiones de lenguajes imperativos y álgebras de procesos[Bae91].

El problema alrededor del cual se desarrolla este trabajo es el de la verificación de ciertas propiedades en sistemas de tiempo real. La verificación de un sistema de tiempo real provee confiabilidad al software resultante. Esto es particularmente relevante en la producción de software usado en aplicaciones médicas y sistemas de control industriales.

En la última década se han implementado herramientas que automatizan total o parcialmente la verificación de sistemas de tiempo real, especificados como autómatas extendidos[Kro, Upp, HyT]. Distintas herramientas implementan distintas variaciones de sistemas, y no se ha desarrollado un modelo único sobre el cual trabajar.

2. Proyecto

2.1. Descripción del Proyecto

La verificación automática de sistemas de tiempo real se encuentra en un proceso de integración cada vez más definido en la producción de software. La diversidad de herramientas disponibles para la especificación de estos sistemas, y que aspiran a ser consideradas seriamente en el desarrollo de software correcto, nos obliga a encontrar una herramienta con la cual poder comparar sus distintos formalismos subyacentes.
El objetivo de este taller es la construcción de una serie de compiladores que conviertan especificaciones -tanto de sistemas como de propiedades a verificar- entre un conjunto de formalismos elegido.
Dichos formalismos serán los usados por las herramientas Kronos[Kro], Uppaal[Upp] y HyTech[HyT].
Se incluye la traducción de un ejemplo clásico, y se han desarrollado interfases gráficas para el uso de los formalismos basados en autómatas. Las mismas son aplicaciones de la herramienta **Graflexed**[Par+98], desarrollada en un taller anterior.

En resumen, el presente trabajo de Taller V comprende:

- estudio de los lenguajes de especificación de tiempo real
- construcción de los compiladores
- estudio y uso de *Graflexed* para la construcción de interfases gráficas
- integración de los compiladores a la interfaz gráfica
- traducción de un ejemplo escogido

La sección 3 presenta los grafos temporizados y algunos lenguajes de especificación de propiedades sobre ellos. La sección 4 propone los compiladores de grafos temporizados como una solución a la diversidad de formalismos existente. Incluye los sublenguajes contemplados por las traducciones para cada uno de los lenguajes elegidos. La sección 5 comprende todo lo relativo a la construcción de las interfaces gráficas. La sección 6 presenta un ejemplo clásico y la sección 7 las conclusiones de este trabajo.

2.2. Hardware, sistemas operativos, lenguajes, utilitarios

El producto resultante estará codificado en C y C++, y deberá ser portable tanto a ambiente Unix como Windows.

3. Especificación de Sistemas de Tiempo Real

Existen distintos enfoques para el modelado de los sistemas de tiempo real. Algunos son las extensiones de autómatas, las extensiones de lenguajes imperativos y las álgebras de procesos. Este taller se centra en el primero de ellos.
Dado un problema de tiempo real -en general asociado a un problema "del mundo real"- se busca un modelo que permita expresar las características del mismo sobre las que se quiere trabajar. Vale la pena recalcar que existe un salto entre el problema real y el modelo construido. Un modelo mal construido puede describir comportamientos que no se correspondan con los del problema real; una propiedad sobre el modelo no necesariamente tendrá los mismos valores de verdad que la propiedad correspondiente del sistema real.
Nuestro ejemplo guía consistirá en dos puertas que se abren al presionar un botón y se cierran automáticamente. Dado este sistema, el primer paso consistirá en construir un autómata que lo modele; ese será el tema de las próximas secciones.

Construido el autómata interesa saber si un conjunto de propiedades dado se verifica. "Sólo una puerta puede estar abierta en un momento dado" o "ninguna puerta estará abierta más de 15 segundos" son dos ejemplo de propiedades.
Existen varias formas de expresar las propiedades; una consiste en usar una lógica de primer orden. Si $P_i(t)$ representa el estado de la puerta *i* en el instante *t* (*abierta*, *cerrando*, *cerrada* y *abriendo*), la primera propiedad puede expresarse como

$$(\neg\exists(t > 0))(P_1(t) = abierta \wedge P_2(t) = abierta)$$

La desventaja de este enfoque consiste en que resulta difícil construir y leer predicados que expresen la propiedad. En general se alejan del lenguaje natural.
Las *lógicas temporales* son otra forma de hacerlo. Se trata de lógicas especializadas en expresiones que impliquen un ordenamiento en el tiempo. Sus construcciones son más sencillas que las que mencionamos antes, y se acercan más al lenguaje natural. En las secciones 3.4 y 3.5 se describen *CTL** y *TCTL* respectivamente, dos lógicas temporales usadas en herramientas de model-checking.

Una vez expresada la propiedad, el paso final consiste en determinar si es verificada por el modelo. Existen algoritmos que permiten hacerlo. Una opción consiste en "correr manualmente" los algoritmos sobre el modelo construido. Este enfoque sin duda presenta problemas. Las ejecuciones pueden tener errores e insumir mucho tiempo. En la medida en que los sistemas crecen este enfoque se hace inviable.
La automatización de algoritmos que permiten verificar propiedades ha cambiado sustancialmente la situación. Ella ha dado lugar a la creación de herramientas informáticas de verificación, los *model-checkers*.
Uno de los principales problemas enfrentados por las herramientas de model-checking es el de la *explosión del número de estados*[Sch+99]. En general la composición paralela de un conjunto de autómatas temporizados crece en forma exponencial con el número de estados. El tamaño de la memoria de los ordenadores se ha convertido en uno de los principales cuellos de botella para la verificación de sistemas grandes.

Se han desarrollado numerosas técnicas que permiten atenuar el problema, entre ellas el *model-checking simbólico*.[Sch+99]

No es el interés de este trabajo profundizar en la implementación de los model-checkers.

3.1. Autómatas clásicos

Los autómatas modelan una máquina que evoluciona de *estado* en estado ejecutando *transiciones*. Por ejemplo la puerta de un ascensor podría ser representada por un autómata con los estados *AA, AO, CA, CO*, representando los estados de la puerta *abierta*, *abriendo*, *cerrada* y *cerrando* respectivamente. Presionar el botón para abrirla implicaría una transición del estado CA al estado AO (suponemos que la respuesta de la puerta es inmediata).
Uno de los atractivos de este enfoque es que tiene una representación gráfica directa y clara; como se puede ver en la figura 3-1, cada estado es representado por un círculo y cada transición por una flecha entre círculos. El estado inicial del autómata se representa con una flecha entrante.

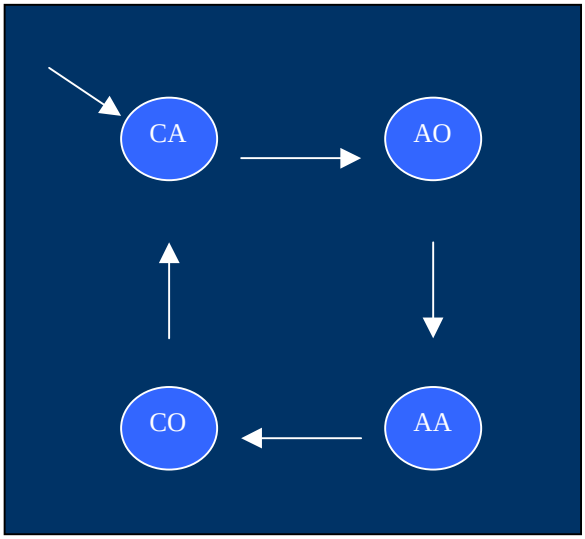


Figura 3-1. Representación gráfica de un autómata que modela una puerta.

¿Qué pasaría si quisiéramos contar la cantidad de veces que se abre la puerta? Una posibilidad consiste en aumentar el número de estados del autómata; podríamos tener los estados AA1, AA2, etc, representando abierta por primera vez, abierta por segunda vez, etc, respectivamente. El problema de este enfoque, como se muestra en la figura 3-2, consiste en que la estructura del autómata depende de la cantidad de veces que se podrá abrir la puerta, y que al aumentar, el autómata crecerá tornándose más complicado. Notar que nuestra puerta real deberá poder abrirse una cantidad cualquiera de veces, hecho que en el autómata implica una cantidad infinita de estados. Nos restringiremos aquí a los autómatas

con una cantidad finita de estados.

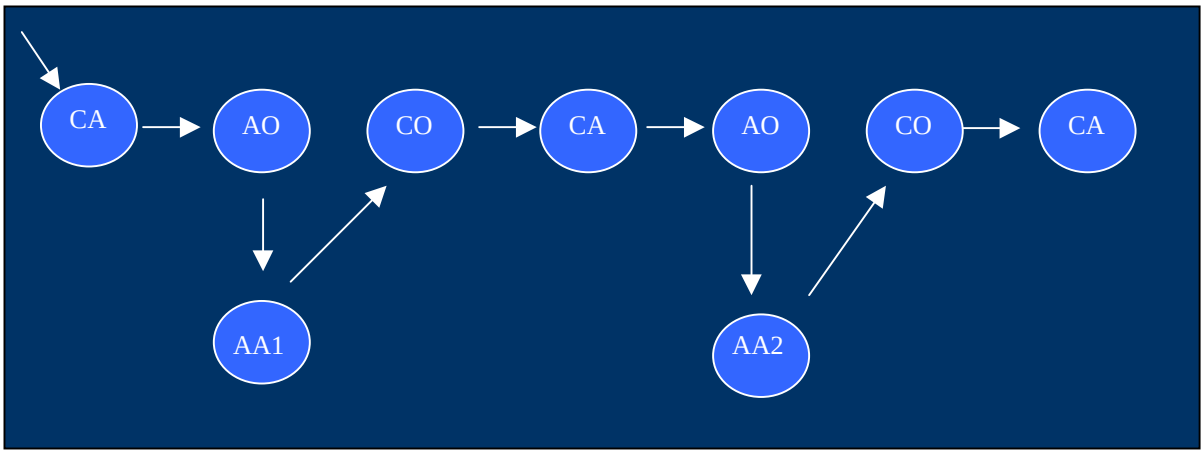


Figura 3-2. ¿Cuántas veces se ha abierto la puerta?

Una solución consiste en agregar *variables*. Una variable *veces* contaría la cantidad de veces que la puerta es abierta. Las asignaciones a las variables ocurren en las transiciones con el cambio de estado, como se puede ver en la figura 3-3. La estructura del autómata no varía. De esta forma se hace la distinción clásica entre datos –la cantidad de veces que se abre la puerta- y control –representado por estados y transiciones-.

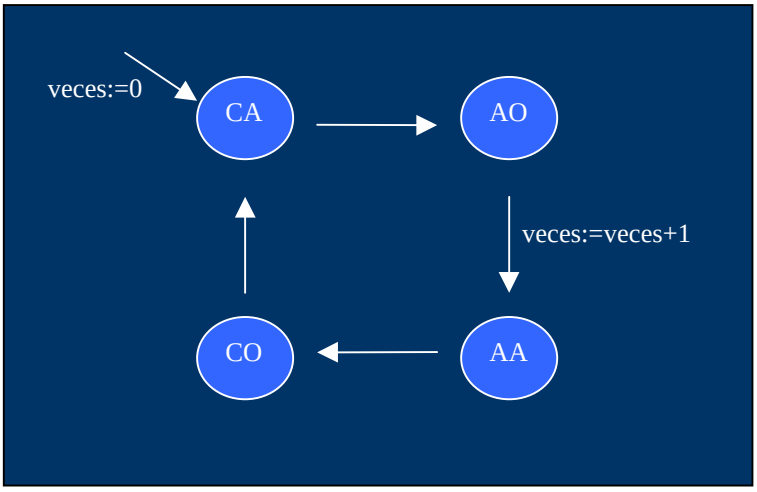


Figura 3-3. Representación gráfica del autómata con variables.

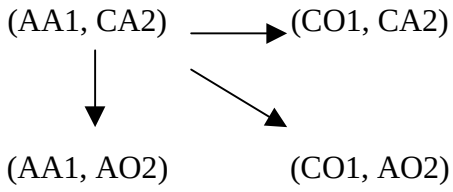
Supongamos ahora que tenemos dos puertas, y nos interesa que ambas se abran a la vez. Un enfoque consiste en construir un nuevo autómata, donde cada estado represente una combinación autorizada del estado de ambas puertas, por ejemplo AA1_AA2, AA1_CO2, etc. Otra posibilidad consiste en resolver el problema por partes, modelando las puertas por

separado como lo hemos hecho hasta ahora. Lo que está faltando es una forma de comunicar ambos autómatas. Existen numerosas formas de efectuar la cooperación –también conocida como *sincronización*- entre autómatas. En todos los casos el resultado final es un único autómata que modela el sistema como una unidad.

Veamos primero un ejemplo de sistema formado por componentes que no se sincronizan. Sería el caso de las dos puertas sin restricciones que las vinculen. El conjunto de estados del autómata global es el *producto cartesiano* de los estados de ambos autómatas, es decir, cada uno de sus estados sería un vector formado por un estado de cada autómata, como por ejemplo (AA1, AA2), (CA1, AO2), etc.
Tendremos 4x4=16 estados, que para este caso de dos autómatas podríamos representar como una matriz

(CA1, CA2)	(CA1, AO2)	(CA1, AA2)	(CA1, CO2)
(AO1, CA2)	(AO1, AO2)	(AO1, AA2)	(AO1, CO2)
(AA1, CA2)	(AA1, AO2)	(AA1, AA2)	(AA1, CO2)
(CO1, CA2)	(CO1, AO2)	(CO1, AA2)	(CO1, CO2)

Las transiciones del nuevo autómata están dadas por las transiciones de los autómatas componentes. En este caso sin sincronización se considerarán todas ellas. Tendremos una transición del autómata global por cada transición correspondiente al primer autómata, una por cada una correspondiente al segundo y una por cada una de las combinaciones posibles de ambos; por ejemplo



Veamos ahora el caso de las puertas que deben abrirse en forma simultánea.
El autómata global correspondiente a los autómatas sincronizados se obtiene eliminando transiciones a aquel. Partiendo del estado inicial en que ambas puertas están cerradas (formado por el estado inicial de cada autómata) permitimos que se ejecute sólo la transición que comienza a abrir ambas puertas a la vez:

$$(CA1, CA2) \rightarrow (AO1, AO2)$$

Hemos eliminado las transiciones (CA1, CA2) $\rightarrow$ (AO1, CA2), (CA1, CA2) $\rightarrow$ (CA1, AO2), entre otras. La figura 3-4 muestra el autómata global.

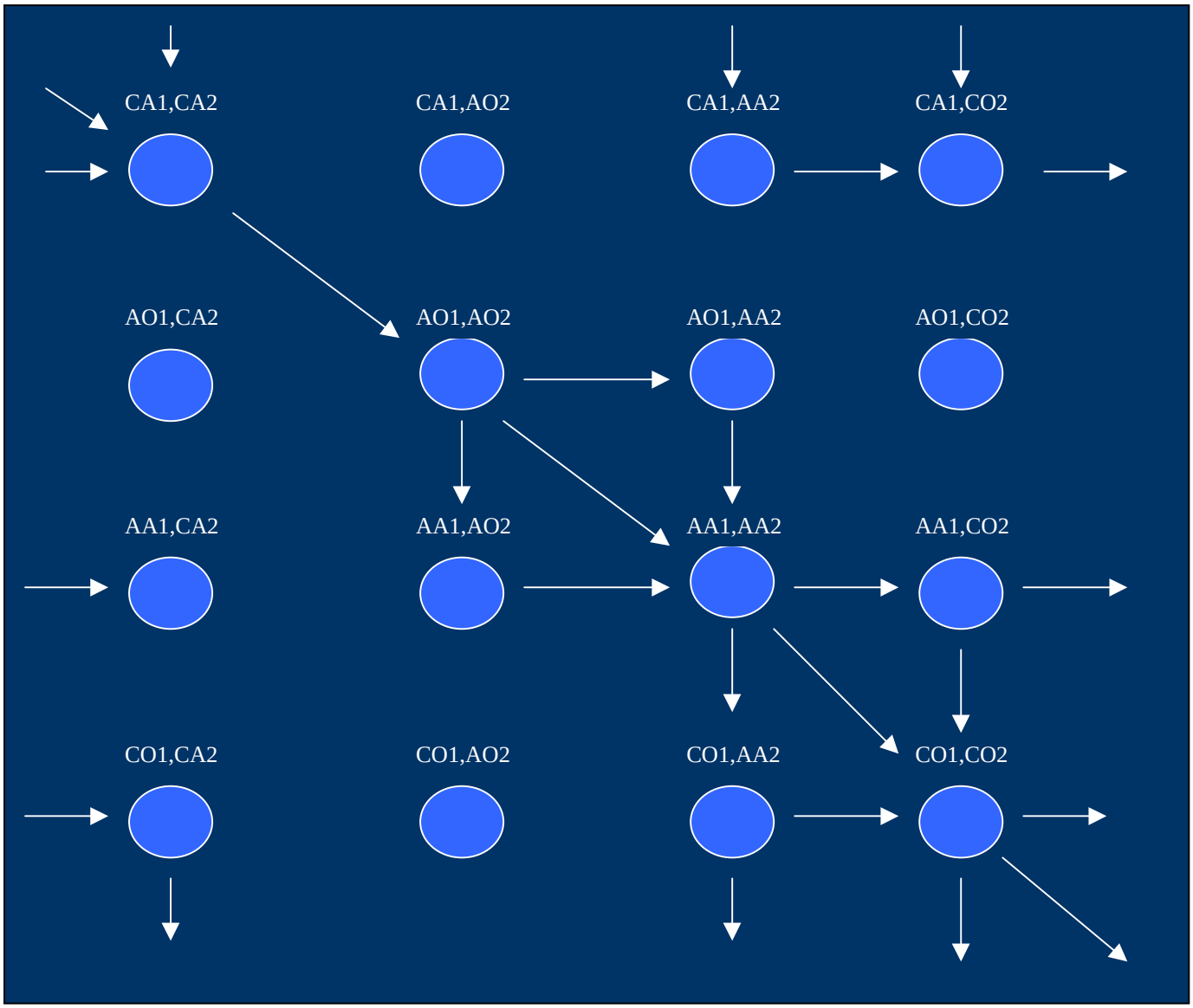


Figura 3-4. El autómata global con sincronización.

El problema de la sincronización consiste en brindar al diseñador del modelo primitivas que permitan definir el conjunto de transiciones válidas del autómata global en una forma sencilla e intuitiva.
Una opción consiste en adornar las aristas con *etiquetas de sincronización*, de forma que las transiciones con la misma etiqueta deban ser tomadas **sólo en forma simultánea**.
Para nuestro ejemplo agregamos la etiqueta de sincronización *juntas*, como se puede ver en la figura 3-5.
Otras formas de sincronización pueden verse en [Sch+99].

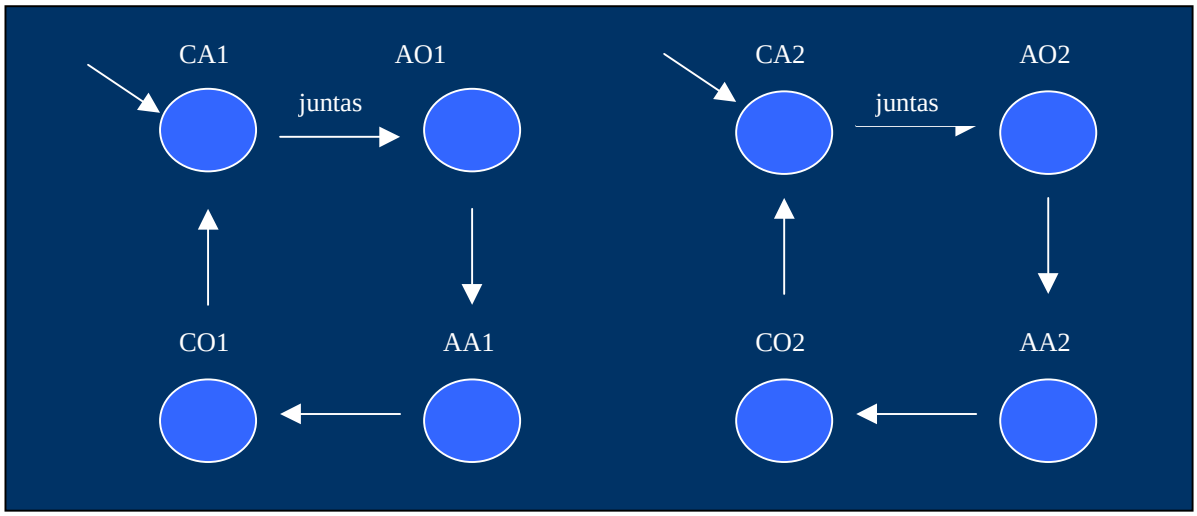


Figura 3-5. Puertas 1 y 2 sincronizadas.

3.2. Autómatas temporizados

Los autómatas clásicos permiten modelar el ordenamiento de eventos en el tiempo, como por ejemplo que “la puerta debe estar cerrada antes de poder abrirse”. Sin embargo no permiten expresar información cuantitativa acerca del intervalo de tiempo que separa dos acciones, como “la puerta deberá permanecer abierta 15 segundos antes de poderse cerrar”. Este último tipo de condiciones caracteriza a los sistemas de tiempo real.
Como forma de resolver el problema se introduce la idea de *autómata temporizado*[Alu94]. Se han propuesto distintas definiciones en la literatura[Hei96]. Aquí nos concentraremos en aspectos generales y comunes a todas ellas.
Los autómatas son extendidos con un conjunto de variables reales llamadas relojes, cuyos valores varían uniformemente con el tiempo. Es posible llevar un reloj a cero, en la misma forma en que en un autómata clásico se asignaba un valor a una variable sobre una transición. De esta forma es posible medir el tiempo de permanencia en un estado –para, por ejemplo, medir la cantidad de tiempo que permaneció abierta la puerta-.
Para representar comportamientos del tipo “la puerta debe permanecer abierta al menos 15 segundos” se introduce la idea de *guarda*. Una guarda es una condición expresada en función de los relojes del autómata que **debe** cumplirse para poder efectuar una transición. Las guardas son atributos de las transiciones, al igual que el volver los relojes a cero. En el ejemplo la guarda sería “Tiempo>=15”, y correspondería a la transición *abierta* $\rightarrow$ *cerrando*.
La sincronización funciona de la misma forma que en los autómatas clásicos.
Otra idea que extiende los autómatas clásicos es la de *invariante*. Se asocia un invariante a cada estado del autómata; al igual que las guardas los invariantes son condiciones sobre los relojes del sistema. Expresan una idea de ‘necesidad’: el sistema podrá permanecer en un estado sólo mientras se cumpla su invariante; al dejar de cumplirse esta condición se **deberá** abandonar el estado, pasando a cualquier otro válido. “La puerta demorará a lo

sumo 30 segundos en cerrarse" se modela a través del invariante "Tiempo<=30". La figura 3-6 representa un autómata temporizado. Observar que una guarda "Tiempo<=30" no tiene el mismo efecto: las guardas expresan una posibilidad. La puerta podría permanecer abierta 35 segundos –ya que nada obliga al autómata a abandonar el estado *abierta*. En ese caso la guarda no se verificaría –no estaría habilitada- y el sistema no podría cambiar de estado(a través de esa transición).
Los invariantes permiten por lo tanto limitar los comportamientos posibles del autómata, descartando esperas indefinidas no deseadas.

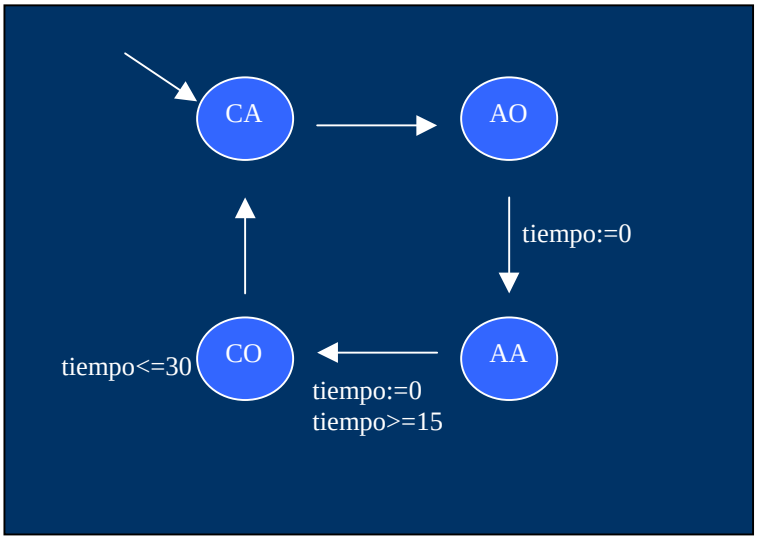


Figura 3-6. La puerta como autómata temporizado.

Notar que los grafos temporizados tal como los hemos definido permiten crear sistemas con comportamientos *no deseados*. Veamos la figura 3-7: si la puerta permaneciera abierta más de 15 segundos el autómata no podrá cambiar de estado, hecho que no queremos modelar. Se dice que el autómata no permite *la divergencia del tiempo,* es decir, que no modela el hecho de que el tiempo no se detiene.

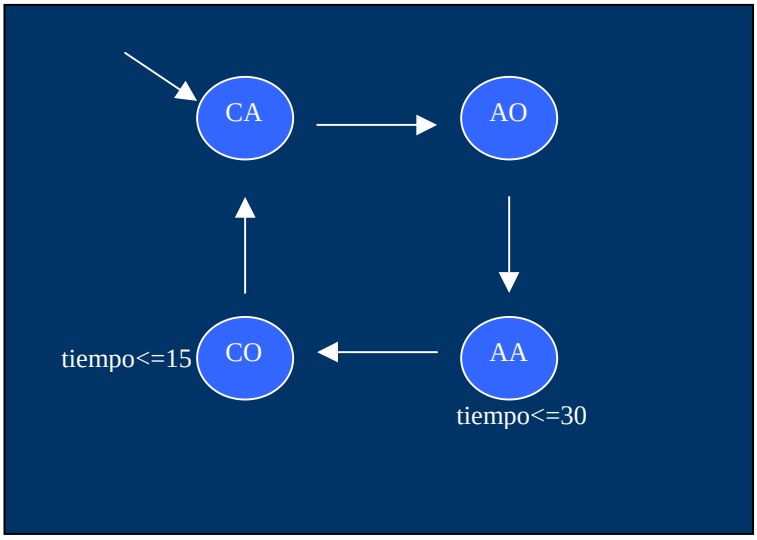


Figura 3-7. Un autómata temporizado que no permite la divergencia del tiempo.

3.3. Extensiones

Urgencia

Algunos formalismos manejan la idea de *transición urgente*. Si alguna de las transiciones salientes del estado actual de un autómata es urgente, el estado deberá ser abandonado tan pronto como sea posible. En ese caso cualquier transición habilitada podrá ser tomada, no necesariamente la urgente. Un caso en el que podría ocurrir una demora podría ser aquel en el que existe sólo una transición saliente (la urgente) y su guarda no está habilitada.
Si bien la urgencia puede ser modelada a través de invariantes y relojes, las transiciones urgentes brindan una forma más natural de expresarla.
Una variante a la transición urgente es la *commited location.* Un estado marcado como commited location deberá ser abandonado inmediatamente; si ninguna transición es posible (sea por problemas de sincronización o porque no existen guardas habilitadas) el sistema entra en *deadlock,* es decir, no puede cambiar de estado.

Sistemas híbridos

En los autómatas temporizados todos los relojes evolucionan a igual velocidad, modelan relojes ideales que no atrasan ni adelantan.
Una extensión natural de los autómatas temporizados consiste en permitir que los relojes tengan distintas velocidades, incluso velocidades variables, como forma de modelar temperaturas, alturas, etc. Para cada estado y reloj se especifica una ley que describe la derivada del reloj con respecto al tiempo; dicha ley podrá tratarse, por ejemplo, de una ecuación diferencial. Estos son los llamados autómatas temporizados híbridos (ATH).
Existen dos casos particulares de los ATH que por sus propiedades y uso frecuente vale la pena mencionar. Uno de ellos es el de los relojes que atrasan o adelantan. Nos referimos a

los autómatas en los que los relojes tienen pendiente fija no necesariamente 1. Por ejemplo un reloj de pendiente 2 es un reloj exactamente dos veces más rápido que uno ideal. El otro caso es el caso de los relojes cuyas derivadas pueden variar en un intervalo cerrado dado. Para el caso del intervalo [1.5, 2] tendremos un reloj que adelantará, sin embargo su velocidad es desconocida; podrá variar, siempre que se mantenga dentro del intervalo.
En la bibliografía no existe un acuerdo respecto a la nomenclatura; los términos *autómatas híbridos lineales* o *multirate timed automata* son usados para referirse a los dos tipos de autómatas mencionados.

3.4. CTL*: una lógica temporal

Describimos aquí la lógica CTL*[Eme90] como ejemplo de una lógica temporal y porque muchas herramientas de model-checking se inspiran en ella (ya sea con subconjuntos o extensiones de ella).

CTL* permite enunciar propiedades de las ejecuciones de un sistema descrito por un autómata **no** temporizado. La componen los siguientes elementos:

- *Proposiciones atómicas.* CTL* usa la noción de *proposición atómica* para referirse a una propiedad que tiene un valor de verdad bien definido **para cada estado**, y que no depende de la ejecución. En nuestro ejemplo agregaríamos al estado AA la proposición atómica *abierta,* etc.
- *Los operadores booleanos clásicos.* Se trata de las constantes *true* y *false*, de la negación $\neg$, de la conjunción $\wedge$, la disyunción, la implicación $\Rightarrow$ y la doble implicación $\Leftrightarrow$. Permiten construir enunciados complejos a partir de los elementales, como por ejemplo $\neg(abierta1 \wedge abierta2)$.
- *Los operadores temporales.* Permiten hacer referencia al encadenamiento de estados **a lo largo de UNA ejecución**, y no a los estados considerados individualmente.
  Los más simples son **X**, **F** y **G**, y expresan propiedades de estados. Si *p* es una propiedad que se cumple en el estado *e*, *X(p)* indica que "el estado siguiente a *e* verificará *p*". *F(p)* indica que "existe un estado futuro en el que se verifica *p*", sin especificar cual. *G(p)* indica que "todos los estados futuros verificarán *p*".
  Finalmente *(p1)U(p2)* indica que *p2* será verdadera para algún estado futuro, y que todos los anteriores verificarán *p1*.
  Resta ahora mencionar los cuantificadores **A** y **E**; permiten expresar propiedades de **CONJUNTOS de ejecuciones**, por lo que son denominados también cuantificadores de caminos. La fórmula *A(p)* expresa que **todas las ejecuciones** que parten del estado actual satisfacen la propiedad *p,* mientras que *E(p)* expresa que **existe una ejecución** que parte del estado actual y que satisface *p*.
  Es importante no confundir *A* y *G*: *A* expresa una propiedad respecto al conjunto de todas las ejecuciones que parten del estado corriente, mientras que *G* expresa una propiedad de los estados de una ejecución dada.
- *La posibilidad de combinar en forma arbitraria los operadores temporales.* Es esto lo que da todo su poder expresivo a la lógica temporal. Para nuestro ejemplo,

$AG(\neg(abierta1 \wedge abierta2))$, que puede leerse "todos los estados de todas las ejecuciones posibles verifican que no pueden valer *abierta1* y *abierta2* a la vez".

3.5. TCTL: una lógica temporal temporizada

Además de las propiedades simplemente temporales –que expresan solamente una relación de ordenamiento en el tiempo- existen propiedades de *tiempo real*, que expresan información cuantitativa sobre el intervalo de tiempo entre acciones. "La puerta deberá demorar a lo sumo 15 segundos en abrirse" es un ejemplo.
Las lógicas temporales sólo pueden expresar un conjunto restringido de propiedades de tiempo real. Un enfoque distinto consiste en usar *lógicas temporales temporizadas*, una extensión de las lógicas temporales.
TCTL (*Timed CTL*) es la versión temporizada de CTL[1]. Permite expresar propiedades de tiempo real sobre autómatas temporizados.
TCTL extiende los operadores temporales de CTL con información cuantitativa sobre los intervalos de tiempo. Por ejemplo la propiedad *E(p)U(<=3)(q)* expresa que existe una ejecución a partir del estado actual a lo largo de la cual se verifica *p* hasta el momento en que se verifique *q* y que esto último ocurrirá **en a lo sumo 3 unidades de tiempo.** Los demás operadores temporales se extienden en forma similar. Una propiedad interesante de estos operadores es que permiten expresar intervalos de tiempo **sin hacer referencia a los relojes del sistema.**
Las propiedades atómicas –a diferencia de CTL, en la que no existe la idea de reloj- pueden hacer referencias a relojes. Por ejemplo *AG(t<5)* indica que a lo largo de todas las ejecuciones y en todo momento el reloj *t* tendrá un valor menor que 5.
Kronos y HyTech son ejemplos de herramientas de model-checking que expresan sus propiedades como subconjuntos de TCTL.

[1] CTL (*Computation Tree Logic*) es el sublenguaje de CTL* que resulta de imponer que cada operador temporal (X, F, U, etc) deba estar precedido inmediatamente por un cuantificador A o E. Esta restricción simplifica el lenguaje pero también su poder expresivo. Por más detalles ver [Sch+99].

4. Un punto de vista unificador: los compiladores de grafos temporizados

Dentro del enfoque de especificación de sistemas a través de los autómatas temporizados han surgido una gran cantidad de propuestas.
Distintos model-checkers han propuesto definiciones –aunque similares, distintas- de extensiones de autómatas, así como también distintos lenguajes de especificación de propiedades.
Cada enfoque e implementación implica sus propias restricciones, ventajas, desventajas y posibilidades. Esto enfrenta al usuario de las herramientas de model-checking al problema de limitarse a una de ellas o, en el mejor de los casos, a traducir en forma manual las especificaciones. Como se mencionara antes, este trabajo propone automatizar la traducción de especificaciones.

4.1. La solución propuesta

El problema a resolver consiste en construir compiladores que, dado un conjunto de lenguajes de especificación de sistemas de tiempo real, permita traducir especificaciones escritas en cualquiera de ellos a una equivalente en cualquiera de los demás.
Los lenguajes elegidos son HyTech, Kronos y Uppaal, por lo cual se pretende el esquema de la figura 4-1.
Una posible implementación consiste en escribir un compilador por cada arista del dibujo, por ejemplo *k2u*, *u2k*, etc. Sus ventajas son la sencillez y la posibilidad de explotar al máximo la expresividad **de cada par** de lenguajes: cada par de lenguajes fijará las restricciones que más le convenga, de forma que en general se lograrán mejores resultados.
El inconveniente de este enfoque consiste en los problemas de mantenimiento que genera. Un cambio en la gramática (cambios en el lenguaje o simplemente el levantamiento de ciertas restricciones iniciales en nuestros compiladores) de alguno de los lenguajes implica cambios en dos fuentes. Por otra parte, dados *N* lenguajes en el esquema, el agregado de uno nuevo implica la escritura de *2N* compiladores.

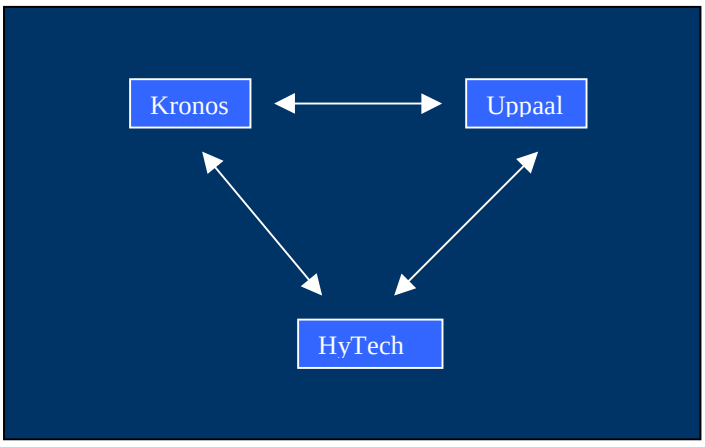


Figura 4-1. La función de los compiladores de grafos temporizados.

Una segunda posibilidad consiste en crear un lenguaje *puente*, como se muestra en la figura 4-2.

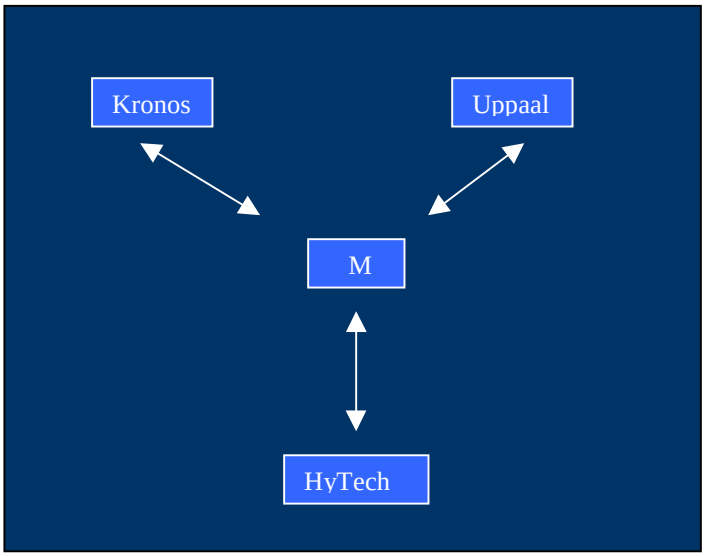


Figura 4-2. La representación intermedia.

El código de entrada se lleva a una forma intermedia, desde la cual se traduce al lenguaje objeto. M deberá poder expresar todas las construcciones que se vayan a querer traducir; la traducción desde M no dependerá del lenguaje origen; dada por ejemplo una traducción Kronos $\rightarrow$ M $\rightarrow$ HyTech y otra Uppaal $\rightarrow$ M $\rightarrow$ HyTech, el algoritmo que correrá sobre M será el mismo. De esta forma se hace más sencillo agregar nuevos lenguajes al esquema: sólo será necesario agregar las rutinas que generen código en M a partir de la nueva gramática, y el código que dado M traduzca al nuevo lenguaje.
Para la presente implementación se ha optado por el segundo enfoque. De esta forma cada compilador recibe como entrada una construcción en un lenguaje origen (formato textual de los lenguajes Kronos, HyTech y Uppaal) y genera a partir de él un parse-tree en memoria. M representa por lo tanto más que un lenguaje una estructura de datos en memoria. Finalmente un algoritmo recorre M generando código en el lenguaje objeto. El siguiente pseudocódigo resume el proceso:

```
Parsear-programa-de-entrada
Construir-parse-tree-en-memoria
Recorrer-parse-tree-y-traducir-al-lenguaje-objeto
```

Por más detalles consultar “Compiladores de Grafos Temporizados – Manual del Sistema”.

4.2. La representación intermedia M

Esta sección describe las principales características de la representación intermedia M. Por detalles de implementación consultar “Compiladores de Grafos Temporizados – Manual del Sistema”.

Especificación del sistema

M describe un conjunto de autómatas.
Cada autómata está formado por un conjunto de estados y aristas. A los estados se asocian un nombre (y un número para traducciones que involucran a Kronos) y un invariante. A las aristas una guarda, una etiqueta de sincronización y una lista de asignaciones. Guardas, invariantes y asignaciones pueden hacer referencia a relojes y variables enteras; ambos tipos de variable podrán ser locales a un autómata o globales a todos ellos.
La sincronización en M, que llamaremos *sincronización trivial* (ver sección 4.5.1), se basa en etiquetas de sincronización. A cada arista se podrá asociar a lo sumo una etiqueta de sincronización, y podrán sincronizarse sólo dos autómatas a la vez.
No es posible inicializar relojes ni variables en M. Se asumen todas inicializadas en cero.
No es posible representar autómatas híbridos en M (ver sección 4.5.2.).

Especificación de propiedades

M toma como lenguaje de especificación de propiedades un subconjunto de TCTL.
No incluye los operadores temporales temporizados del tipo $EF_{(\sim k)}\phi$ o $A\phi U_{(\sim k)}\psi$, sólo *AG* y *EF* sin anidar. No se podrá hacer referencia a propiedades atómicas, sólo a estados de los autómatas. Los operadores lógicos están incluidos.

4.3. Sublenguajes a traducir

Dada la sintaxis de cada uno de los tres lenguajes elegidos se ha construido un subconjunto válido para cada uno de ellos. Sólo dichos subconjuntos serán traducidos.
En las próximas secciones se presentan las características básicas de cada uno de ellos. Por detalles ver Anexo A.
Se usará el esquema de la figura 4-3 para describir las gramáticas. Cada fila describe el conjunto de todas las producciones de igual lado izquierdo. La primera columna da en cada caso la variable correspondiente a las producciones en cuestión, la segunda sus lados derechos –uno por renglón- y la tercera una breve descripción.

4.3.1. Kronos

Especificación del sistema

La figura 4-3 muestra los aspectos más importantes de la subgramática de Kronos reconocida por el compilador. La gramática completa se anexa en A.2.

graph	NUMERAL LOCS NAT NUMERAL TRANS NAT NUMERAL CLOCKS listofclocks NUMERAL SYNC listoflabels Listofstates	*Descripción del grafo formada por la cantidad de estados y transiciones, la cantidad y lista de relojes, la declaración de las etiquetas de sincronización y la lista de estados*
	LOC DOSPTOS NAT PROP DOSPTOS listofprops INVAR DOSPTOS clockconstraint TRANS DOSPTOS listoftransitions	
Trans	clockconstraint IGUAL MAYOR NAME PTOYCOMA RESET ILLAVE listofclocks DLLAVE PTOYCOMA GOTO NAT	*Descripción de una transición formada por su guarda, la lista de asignaciones y el estado destino*

Figura 4-3. Subgramática Kronos del lenguaje de especificación de sistemas aceptada por el compilador de grafos temporizados.

Especificación de propiedades

La figura 4-4 muestra los aspectos más importantes de la subgramática de Kronos reconocida por el compilador. La gramática completa se anexa en A.2.

Formula	TRUE	*Las propiedades TCTL aceptadas son del tipo AG, EF, expresiones parentizadas, lógicas y restricciones sobre los relojes.*
	Clockconstraint IPAR formula DPAR NAME	
	NOT formula	

	formula AND formula	
	formula OR formula	
	Formula IMPL formula	
	ED bound formula	
	AB bound formula	
	IPAR formula DPAR	
	NAME	

Figura 4-4. Subgramática Kronos del lenguaje de especificación de propiedades aceptada por el compilador de grafos temporizados.

El lenguaje aceptado permite expresar propiedades del tipo AG y EF de CTL* (es decir, **no** temporizadas), expresiones lógicas y restricciones sobre los relojes del sistema.

4.3.2. Uppaal

Especificación del sistema

La figura 4-5 muestra los aspectos más importantes de la subgramática de Uppaal reconocida por el compilador. La gramática completa se anexa en A.3.

Ita	declGlob ProcList Globals	*Sistema formado por declaraciones globales, lista de autómatas definidos y lista de autómatas habilitados*
DeclGlob ...	Channel declGlob	*Declaración de canales y variables globales*
	VarGlob declGlob	
DeclLoc	(vacío)	*Declaración de variables locales*
	VarLoc declLoc	
ProcList	Proc	*Lista de autómatas definidos*
	Proc ProcList	
Globals	SYSTEM IdList PTOYCOMA OpHide	*Lista de autómatas habilitados*
Channel	CHAN IdList PTOYCOMA	*Declaración de canales*
VarGlob	Type IdList PTOYCOMA	*Declaración de variables globales*

VarLoc	Type IdListPref PTOYCOMA	*Declaración de variables locales*
Proc	PROCESS ID ILLAVE declLoc StateDecls TransDecls DLLAVE	*Definición de un autómata*
StateDecls	STATE StateList PTOYCOMA INIT ID PTOYCOMA	*Declaración de estados del autómata*
TransDecls	TRANS TransList PTOYCOMA	*Declaración de las transiciones del autómata*
TransList	Trans	*Lista de transiciones*
	Trans COMA TransList	
Trans	ID RESTA MAYOR ID ILLAVE OpGuard OpSync OpAssign DLLAVE	*Transición formada por estados origen y destino, guarda, etiqueta de sincronización y lista de asignaciones*

Figura 4-5. Subgramática Uppaal del lenguaje de especificación de sistemas aceptada por el compilador de grafos temporizados.

Especificación de propiedades

La figura 4-6 muestra los aspectos más importantes de la subgramática de Uppaal reconocida por el compilador. La gramática completa se anexa en A.3.

TopProp	Prop	*La propiedad es simple o una negación de ella*
	NOT Prop	
Prop	E MENOR MAYOR StateProp	*Las propiedades simples son del tipo EF o AG*
	A ICORCH DCORCH StateProp	
StateProp	AtomicProp	*Propiedades de estado*
	NOT StateProp	
	IPAR StateProp DPAR	
	StateProp OR StateProp	
	StateProp AND StateProp	
	StateProp IMPLY StateProp	

Figura 4-6. Subgramática Uppaal del lenguaje de especificación de propiedades aceptada por el compilador de grafos temporizados.

En general tienen la forma *A[]P* o *E<>P* (también es válida la negación de ellas), donde *P* representa un "state predicate". *A[]P* representa "todas las configuraciones alcanzables desde el estado inicial verifican *P*", o dicho de otra forma, "*P* se verifica en todo instante para todas las ejecuciones posibles que parten del estado inicial". Se dice que *P* es un invariante del sistema. La construcción es similar al cuantificador temporal *AG* de CTL.
En forma similar, *E<>P* expresa una posibilidad. Existe un estado de una ejecución que comienza en el estado inicial que verifica *P*. Nuevamente, es similar a *EF* de CTL.
El resto de las construcciones expresan propiedades de estado.

4.3.3. HyTech

Especificación del sistema

La figura 4-7 muestra los aspectos más importantes de la subgramática de HyTech reconocida por el compilador. La gramática completa se anexa en A.4.

automata_descriptions	declarations automata	*Descripción del sistema formada por las declaraciones y los autómatas*
Declarations	VAR var_lists	*Palabra clave más una lista de declaraciones.*
Var_lists	(vacía) Var_list DOSPTOS var_type PTOYCOMA var_lists	*Listas de variables acompañadas de su tipo*
Automata	Automaton automata Automaton	*Lista de autómatas*
Automaton	AUTOMATON NAME prolog locations END	*Cada autómata está formado por un prólogo y por su lista de estados*
Prolog	initialization sync_labels	*Formado por las inicializaciones del autómata y la declaración de las etiquetas de sincronización*
Location	LOC NAME DOSPTOS WHILE convex_predicate WAIT ILLAVE rate_info_list DLLAVE transitions	*Cada estado está formado por su invariante y su lista de transiciones salientes*

Transition	WHEN GuardList update_synchronization GOTO NAME PTOYCOMA	*Las transiciones están formadas por su lista de guardas, su etiqueta de sincronización, su lista de asignaciones y su estado destino*

Figura 4-7. Subgramática HyTech del lenguaje de especificación de sistemas aceptada por el compilador de grafos temporizados.

Especificación de propiedades

La figura 4-8 muestra los aspectos más importantes de la subgramática de HyTech reconocida por el compilador. La gramática completa se anexa en A.4.

Commands	region_declaration statements	*La propiedad está compuesta por las declaraciones de las variables de región y los comandos*
Region_declaration	VAR region_declaration_list DOSPTOS REGION PTOYCOMA	*Declaración de las variables de región*
Statements	Statement PTOYCOMA statements	*Lista de comandos separados por punto y coma*
Statement	NAME ASIG re	*Los comandos pueden ser solamente asignaciones*
Re	IPAR re DPAR	*Construcción de regiones.*
	COMPL re	
	Re AMP re	
	Re UNION re	
	Re SUMA re	
	Re RESTA re	
	Re IGUAL re	
	Re MENOR re	
	Re MAYOR re	
	Re MAYORIGUAL re	
	Re MENORIGUAL re	
	REACH FORWARD FROM re ENDREACH	

REACH BACKWARD FROM re ENDREACH
NAME
TRUE
FALSE
LOC ICORCH NAME DCORCH IGUAL NAME

Figura 4-8. Subgramática HyTech del lenguaje de especificación de propiedades aceptada por el compilador de grafos temporizados.

4.4. Sublenguajes descartados

Las secciones que siguen describen en forma general los lenguajes descartados. Por una descripción completa de los mismos consultar Anexo B.

4.4.1. Sincronización

La presente implementación contemplará solamente un conjunto restringido de formas de sincronización que de aquí en más llamaremos *sincronización trivial*. En el Anexo C se detalla la complejidad de la traducción de otros casos más generales.
Se permitirá solamente una etiqueta de sincronización por transición, y solamente dos autómatas podrán sincronizarse a la vez. Esto significa que una etiqueta de sincronización podrá estar definida o ser referenciada en a lo sumo dos autómatas distintos. Todas las demás formas de sincronización serán descartadas.

4.4.2. Autómatas temporizados híbridos

Kronos no da soporte nativo a los autómatas temporizados híbridos.
La situación de Uppaal es similar a la de Kronos; disponía en una versión anterior de un preprocesador llamado *hs2ta* que implementaba la traducción de los ATHL a AT [OSY94]. Sin embargo dicha herramienta fue descontinuada en Uppaal2k, pues introducía problemas en los “traces” de diagnóstico.
Finalmente HyTech sí da soporte a los ATHL y ATH.

No existe un método sencillo general para traducir ATH a AT. Por otra parte el caso de los ATHL sí es manejable, pero se ha optado, por una cuestión de complejidad, por dejarlo de lado en esta implementación. Queda pendiente como una mejora futura al producto. La propuesta consiste en seguir [OSY94], planteando

- Kronos no da soporte a los ATHL, deberán simularse según [OSY94]
- en H se pueden dar *N* condiciones del tipo combinaciones lineales(con

coeficientes racionales) de relojes vinculados por un operador relacional (como descripción de la derivada de los relojes), o nombres en un intervalo de racionales.
- Uppaal2k tampoco da soporte a los ATHL, deberán simularse según [OSY94]

Se propone tomar Uppaal, esto es, la condición implicará que la derivada de un reloj pertenezca a un intervalo **cerrado** dado. Los extremos del intervalo serán enteros.

4.4.3. Kronos

El lenguaje de especificación de propiedades de Uppaal no incluye operadores temporales temporizados; por lo tanto expresiones Kronos del tipo *ab{~k}(p)* no podrán ser traducidas a Uppaal y deberán ser descartadas.
Si bien los operadores *A<>* y *E[]* están anunciados para nuevas versiones de Uppaal, no están incluidos en la actual; por lo tanto las fórmulas Kronos que incluyan *ad* y *eb* serán descartadas.
No existen construcciones del lenguaje de especificación de sistemas de Kronos que deban ser descartadas.
Por una lista completa de las construcciones descartadas consultar B.1.

4.4.4. Uppaal

A excepción de las primitivas que expresan transiciones urgentes, todas las construcciones de Uppaal serán traducidas.

4.4.5. HyTech

Existe una gran cantidad de construcciones del lenguaje de especificación de propiedades de HyTech que serán descartadas en las traducciones a HyTech y Kronos.
Dado su enfoque algorítmico, HyTech dispone de primitivas de impresión en pantalla; ellas no tienen sentido en una lógica como TCTL, por lo que construcciones del tipo *prints cadena* y *print omit locList* deberán ser descartadas.
Lo mismo ocurre con la mayor parte de las sentencias de control. Salvo algunas excepciones, las construcciones que incluyan *while* e *if* serán descartadas.
Los *reach forward* no pueden ser traducidos a TCTL, ya que no es posible expresar "referencias al pasado".
Del iterador *iterate* sólo se traducirá una versión recortada.
Las construcciones *pre (regExp)* y *post(regExp)* no pueden ser traducidas, al igual que *hull(regExp)*.
Si bien los templates simplifican muchas especificaciones, no se traducirán; se dejarán para versiones futuras de la herramienta. Es este también el caso de los arrays y rangos, ya que ni Kronos ni Uppaal cuentan con construcciones similares.

Otro de los atractivos de HyTech lo constituyen los análisis paramétricos. Nuevamente, los verificadores de Kronos y Uppaal no cuentan con facilidades similares, por lo tanto no se traducirán.
Las transiciones urgentes también serán descartadas.
Por una lista completa de las construcciones descartadas consultar B.3.

4.5. Traducciones

En las secciones que siguen se da una descripción general de las traducciones, tanto para los lenguajes de especificación de sistemas como para los de propiedades.

4.5.1. Kronos $\rightarrow$ M

Sistema

Los estados quedan identificados en Kronos por un natural, y en HyTech y Uppaal por nombres; las traducciones asociarán a un estado *i* el nombre *Statei*.
Las propiedades atómicas de los estados Kronos no tendrán correspondientes en M.
Invariantes y guardas se traducirán sin pérdidas.
Sólo se permitirá **una** etiqueta de sincronización por transición, y su traducción no implica cambios.

Propiedades

La traducción de las fórmulas Kronos a M es directa, ya que ambos toman un subconjunto de TCTL. Sin embargo existen dos consideraciones a tener en cuenta. Por un lado las restricciones señaladas en la sección 4.4.3 respecto a las fórmulas temporizadas descartadas. Y por otra parte que las referencias a propiedades atómicas en fórmulas TCTL son sustituidas por un predicado que hace referencia a los estados en los que se verifica la propiedad atómica.

4.5.2. Uppaal $\rightarrow$ M

Sistema

Los estados se traducen en forma directa, sin cambios.
Uppaal define los canales como mecanismo de sincronización, basado en el modelo de transmisión de mensajes. Dado un canal 'a' es posible etiquetar las transiciones de los autómatas con 'a!' -para el envío- y 'a?' -para la recepción-. Ambas etiquetas, verificada la condición de sincronización trivial, serán traducidas a 'a'.
Asignaciones, guardas e invariantes deben respetar un conjunto de restricciones sintácticas que no detallaremos aquí.

Propiedades

En general las propiedades tienen en Uppaal la forma *A[]P* o *E<>P* (también es válida la negación de ellas), donde *P* representa un "state predicate".
A[]P representa "todas las configuraciones alcanzables desde el estado inicial verifican *P*", o dicho de otra forma, "*P* se verifica en todo instante para todas las ejecuciones posibles que parten del estado inicial" (se dice que *P* es un invariante del sistema).
Esta construcción es similar al cuantificador temporal *AG* de CTL*. Sin embargo la semántica de *AG(P)* indica que "*P* se verifica en todo instante para todas las ejecuciones del sistema que parten del estado actual".
Por su parte *E<>P* expresa una posibilidad: "existe un estado de una ejecución que comienza en el estado inicial que verifica *P"*. Nuevamente, es similar a *EF* de CTL*, salvo por la condición sobre el/los estados iniciales.
Dado que M respeta la semántica de CTL*, las traducciones deberán tener en cuenta los estados iniciales. *A[]prop* y *E<>prop* de Uppaal serán traducidos a $init \Rightarrow AG(prop)$ y $init \Rightarrow EF(prop)$ de M, donde *init* representa los estados iniciales de todos los autómatas.

4.5.3. HyTech $\rightarrow$ M

Sistema

Los estados se traducen en forma directa, sin cambios.
El modelo de sincronización de HyTech es similar al de M, se traduce sin cambios.
Asignaciones, guardas e invariantes deben respetar un conjunto de restricciones sintácticas que no detallaremos aquí.

Propiedades

Las propiedades se especifican en HyTech a través de código escrito en un lenguaje de comandos. Dicho lenguaje incluye iteradores y la noción de alcanzabilidad, y permite la construcción de regiones. El código aceptado por los compiladores construidos se restringe a una lista de asignaciones de regiones a variables de región.
La idea de las traducciones $H \rightarrow M$ es asociar una región a cada fórmula TCTL.
Dicha región (que será la última en referenciarse en el fuente H) podrá haber sido definida en función de otras.
Las regiones auxiliares -que sirven para construir la región que define la propiedad- podrán darse explícitamente en la región final, o podrán almacenarse en variables de región, y más tarde ser referenciadas. Dichas variables de región serán mantenidas en una tabla de símbolos; por más detalles de implementación ver "Compiladores de Grafos Temporizados - Manual del Sistema ".

Existen básicamente tres tipos de regiones válidas, que pueden ser asignadas a variables de región.
Uno de ellos es el que define una *propiedad de estado*:

var_region := state_predicate;

Estas regiones podrán estar formadas por referencias a estados y relojes de algún autómata, y sumas y restas de variables; son ejemplos de este tipo de región las expresiones *loc[train]=far* y *x<=15*. No expresan una propiedad, sirven de construcciones intermedias.
Un segundo tipo de región es el definido por las construcciones *reach forward* y *reach backward*. Para el primero de ellos tendremos

var_region := reach forward from state_predicate endreach;

Asigna una región (definida como el conjunto de los estados alcanzables desde una región inicial definida a través de un SP) a una variable de región. No expresa una propiedad, sirve de construcción intermedia.

Y finalmente el tercer tipo de regiones es el que define propiedades. El lenguaje permitirá expresar solamente propiedades del tipo *AG* y *EF*. Para las primeras tendremos construcciones de la forma

var_region := [reach_forw | var_RF | iterate | var_I] [< | <= | =] [reach_back | var_RB];

Asigna una región a una variable de región; define una propiedad del tipo $init \Rightarrow AG(final)$, es decir, permite representar invariantes del sistema.
Para las propiedades del tipo *EF* tendremos

var_region := [reach backward from safety_SP endreach | var_RB] & inicial_SP;

El *reach backward* puede estar dado explícitamente o por una variable. Lo mismo ocurre con *inicial_SP*, que debe ser un *state_predicate*.
Representa una propiedad del tipo $init \Rightarrow EF(safety)$, es decir, una propiedad de tipo *safety*.

Las propiedades de tipo *bounded response* son representadas en HyTech por *procesos monitores*[Hen+96]; resulta al menos muy complejo reconocer este tipo de construcciones, por lo que quedarán pendientes para mejoras futuras al producto. Una forma de encarar el problema consiste en definir restricciones sintácticas que permitan reconocer los procesos monitores sintácticamente: una construcción determinada podría estar representando solamente un proceso monitor. De todas formas sería necesario evaluar la viabilidad y utilidad de este enfoque.
Las propiedades de tipo *minimal event separation* tienen consideraciones similares a las anteriores[Hen+96].
Por una descripción completa de las construcciones válidas consultar A.4.

4.5.4. Kronos←M

Sistema

Los nombres de estados serán traducidos a naturales consecutivos comenzando por el cero.
A cada estado del autómata destino de la traducción se le asignará una propiedad atómica igual al nombre del estado en el autómata origen, como recurso mnemotécnico.
Si la construcción en M incluye variables enteras, la traducción no será posible.
Se chequeará sintaxis de invariantes, guardas y asignaciones.
La declaración y referencia a variables locales será traducida a nuevas variables que tendrán el nombre del autómata que las declara como prefijo. Todas las variables serán declaradas en todos los archivos de salida Kronos.

Propiedades

La traducción de propiedades es directa ya que Kronos y M implementan subconjuntos de CTL*.

4.5.5. Uppaal←M

Sistema

La traducción de los nombres de los estados es directa.
En general invariantes, guardas y asignaciones no presentarán problemas sintácticos.

Propiedades

La traducción deberá tener en cuenta la consideración descrita en 4.5.2. respecto a los estados iniciales. La propiedad a traducir deberá ser una implicación y su antecedente deberá ser un predicado que “cubra” el estado inicial de todos los autómatas del sistema.

4.5.6. HyTech←M

Dada la complejidad de la traducción se ha dejado para futuras versiones de la herramienta.

4.6. Una propuesta alternativa: AuTe

Este trabajo se ha centrado en los compiladores como solución al problema de la diversidad de formalismos y herramientas existentes. Dada una especificación en un lenguaje determinado, los compiladores permiten obtener una equivalente en un lenguaje distinto, como se muestra en la figura 4-9. Este enfoque permite al diseñador de sistemas de tiempo real cambiar la herramienta **sobre la que especifica**, lo que tiene dos consecuencias directas. Por un lado las traducciones complican las especificaciones finales, haciéndolas más difíciles de comprender. Es este el caso de los esquemas de sincronización: una

sincronización de tipo *broadcast* en Kronos será traducida a una equivalente más compleja en Uppaal. Por otro lado el usuario de los compiladores deberá conocer **cada una** de las herramientas contempladas por los compiladores, cada una de ellas basada probablemente en un formalismo distinto.

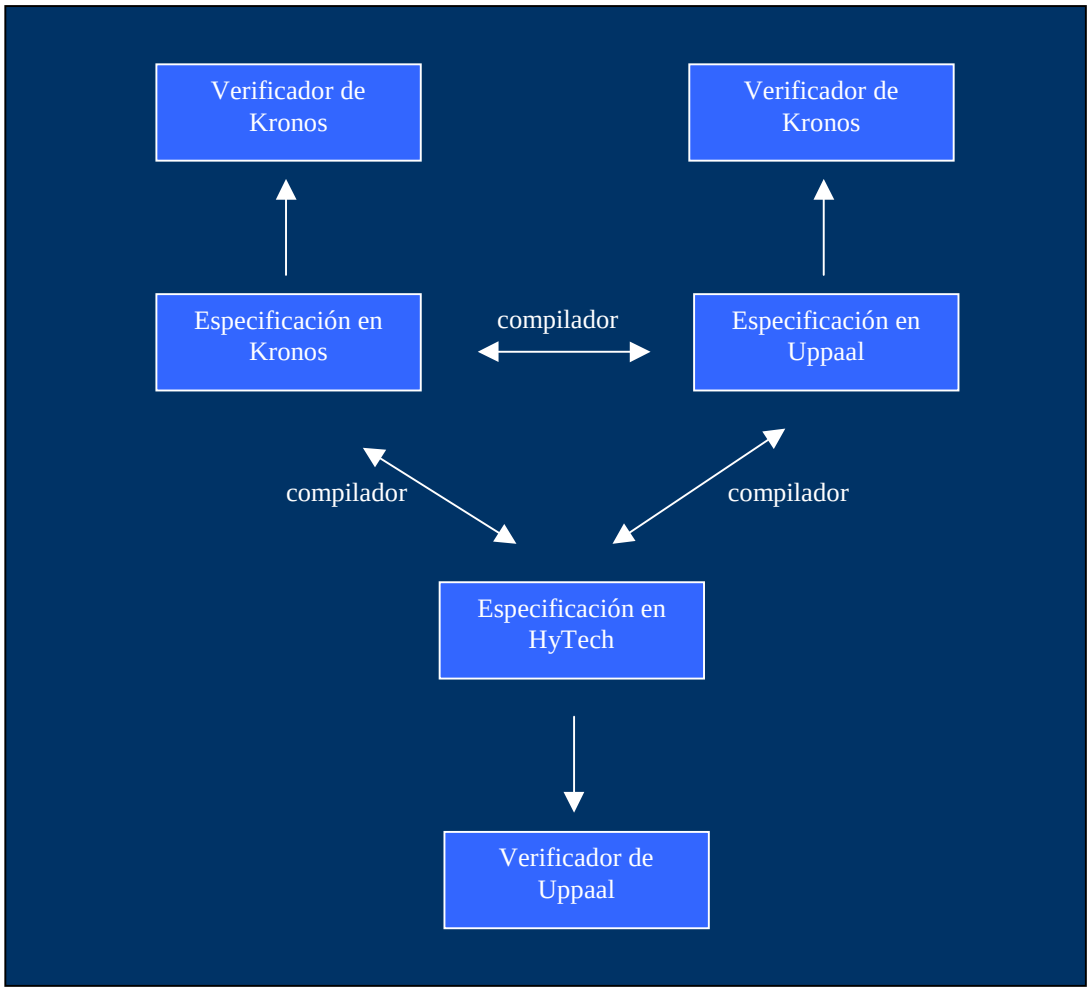


Figura 4-9. El enfoque de los compiladores de grafos temporizados.

Dadas estas dificultades se propone un enfoque distinto. Consideraremos un nuevo formalismo que abarcará tanto la especificación de sistemas (que en adelante llamaremos *AuTeS*, por "Autómatas Temporizados-Sistema") como de propiedades (*AuTeP*, por "Autómatas Temporizados- Propiedades").

AuTeS será un lenguaje de alto nivel: dispondrá de primitivas que permitan expresar las construcciones más comunes de los sistemas de tiempo real. En AuTe será posible especificar esquemas de sincronización tanto del tipo broadcast como del tipo de envío de mensajes, pudiendo usarse ambos en un mismo autómata. De esta forma resultará más clara

la especificación, dejándose de lado las simulaciones con committed locations, etc, que complican la interpretación. Algo similar ocurrirá con la especificación de la idea de urgencia.
El tipo de lenguaje de AuTeP todavía no ha sido definido, pero podría tratarse de una lógica temporal. El enfoque sería similar al de AuTeS: primitivas que permitan expresar los cuatro o cinco tipos clásicos de propiedades (invariantes, bounded response, etc) que reúnen el gran porcentaje de las popiedades que en la práctica interesa expresar.
El diseñador trabajará solamente sobre AuTeS y AuTeP; como se puede ver la en figura 4-10, *AuTe* (AuTeS y AuTeP) no dispondría de un verificador propio, sino que usaría otros existentes. Consideremos Kronos, HyTech y Uppaal. Existirá un compilador -*AuTecC*- que convierta una entrada AuTe en una salida Kronos, HyTech o Uppaal. De acuerdo al tipo de construcciones de la especificación AuTe, AuTeC determinará el lenguaje objeto. Por ejemplo, para autómatas híbridos generará salida HyTech. Es importante notar que esta salida –posiblemente complicada- será empleada por el model-checker correspondiente solamente, **no** por el usuario final.

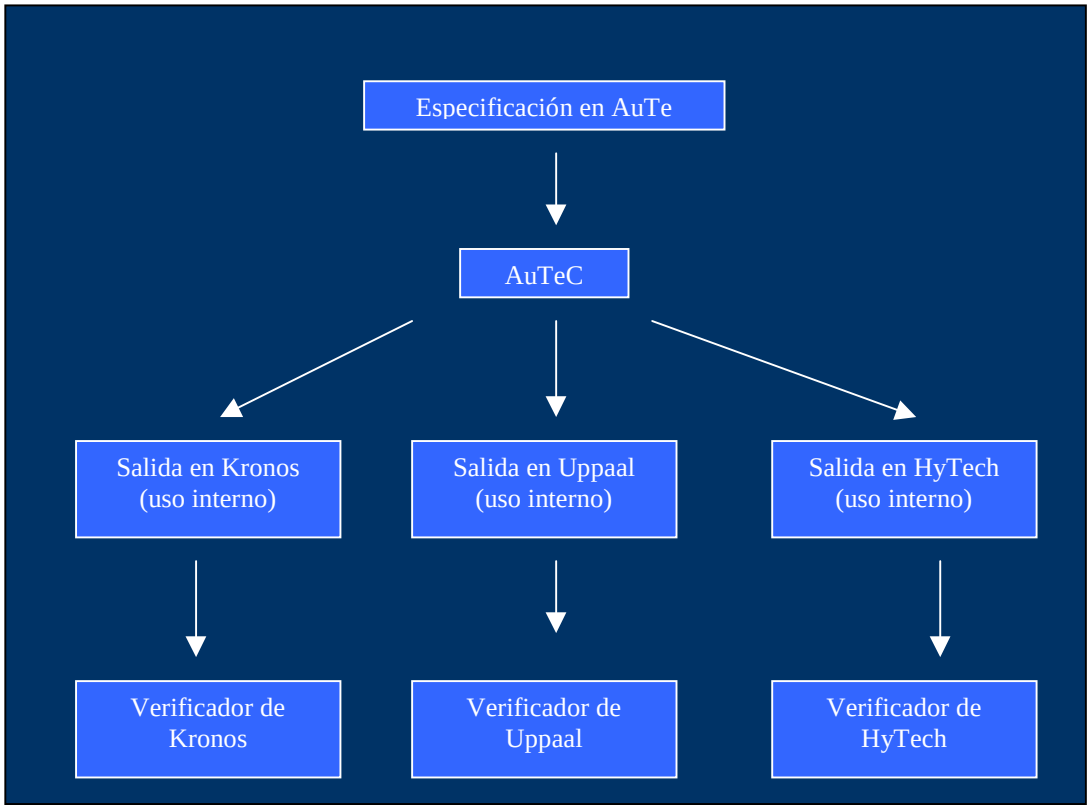


Figura 4-10. Un enfoque distinto: AuTe.

De esta forma el diseñador trabaja sobre un único lenguaje de especificación y elige las construcciones que mejor se adapten a su problema.
Algunos aspectos de AuTe que podrían determinar el lenguaje objeto son el tipo de propiedad a verificar, autómatas híbridos o no híbridos y análisis paramétricos.
Para una propiedad a verificar una respuesta *TRUE* será suficiente; sin embargo una respuesta *FALSE* es acompañada en general de información adicional de diagnóstico, como

por ejemplo una ejecución que viola la propiedad. Dicha ejecución hace referencia al sistema leído por el model-checker, y NO al especificado en AuTe. Por lo tanto AuTeC será capaz de interpretar dicha salida y expresarla en términos de AuTe.
AuTe no permite reutilizar especificaciones existentes. Sin embargo se podrían crear compiladores adicionales para traducciones a AuTe; este enfoque puede preservar en AuTe especificaciones "sucias", como por ejemplo en el caso de una especificación Uppaal en la que se ha simulado una sincronización tipo *broadcast;* la especificación puede ser traducida en forma directa, a pesar de que AuTeS probablemente cuente con primitivas de sincronización similares a las de Kronos que permitan expresar el mismo sistema en una forma más sencilla. En todo caso de trata de una limitación de Uppaal y no de AuTe.
AuTeC permitirá traducir tanto parejas <sistema, propiedad> como solamente sistemas, para traducir y evaluar luego un conjunto de propiedades sobre ellos.
En resumen, AuTe permite explotar la gran cantidad de verificadores existentes, sin para ello ser necesario conocer los formalismos sobre los que trabajan. El diseñador trabaja sobre un único lenguaje de especificación, de "alto nivel", que permite simplificar las especificaciones.

5. Visualización

Esta sección abarca lo relativo a la creación de las interfases gráficas.

5.1. Descripción del problema

El problema planteado consiste en construir herramientas que permitan representar gráficamente los grafos temporizados manejados por las herramientas de verificación de sistemas de tiempo real Kronos, HyTech y Uppaal. La solución debe ser una aplicación de la herramienta de generación de editores de grafos *Graflexed*.

La idea original consistía en manipular las representaciones gráficas en forma independiente del lenguaje que los describía. Una vez obtenido el grafo deseado se debía poder generar un archivo que describiera el sistema en el lenguaje textual -de descripción de sistemas- de cualquiera de los lenguajes, de forma de poder usar el verificador correspondiente. Los compiladores, por su parte, permitirían comunicar los lenguajes textuales.

5.2. Herramientas usadas

5.2.1. Graflexed

Graflexed simplifica la construcción de editores de grafos en muchos sentidos.
Resuelve el manejo de algunos eventos básicos como la creación ,borrado, selección y desplazamiento de nodos y aristas. No es necesario ocuparse de detalles de bajo nivel como por ejemplo si el evento “click” ocurrió en un nodo o en una arista.
Brinda la flexibilidad de poder asociar estructuras de datos cualquiera a nodos y aristas, lo que permite contemplar una gran cantidad de tipos de grafos (grafos temporizados, cadenas de Markov, etc).
Otra ventaja fundamental es que permite la creación de editores que pueden interactuar; esto hace posible ejecutar operaciones sobre conjuntos de grafos, como la unión, intersección o la composición paralela, y la creación de nuevas ventanas de edición con el grafo resultado.
Finalmente los editores se independizan de la interfaz gráfica. Esto es logrado a través de la definición de una interfaz editor$\leftrightarrow$visualizador llamada *EditLayoutInterface*. Para desplegar un grafo en pantalla los editores invocan métodos de la interfaz en lugar de funciones de la biblioteca gráfica elegida.

5.2.2. LEDA

LEDA (Library for Efficient Data Types and Algorithms) [LED] es una biblioteca escrita en C++. Fue usada por Graflexed para implementar la EditLayoutInterface que provee. Además de su biblioteca gráfica implementa una completísima colección de tipos de datos y algoritmos como ser listas, conjuntos, diccionarios, grafos y algoritmos sobre ellos.

5.3. Solución construida

Finalmente, dadas algunas diferencias entre los lenguajes difíciles de salvar, se optó por construir un editor para cada lenguaje. De esta forma cada uno maneja un formato propio, y es capaz de generar archivos textuales para el lenguaje que representa. Además, cada editor respeta la sintaxis del lenguaje que representa (para guardas, invariantes, etc).
Los editores construidos son *GraflexedK*, *GraflexedH* y *GraflexedU*, para Kronos, HyTech y Uppaal respectivamente.

5.4. Otros detalles

Por más detalles del producto final consultar “GraflexedX – Manual del Usuario”.

5.5. Limitaciones

- los editores manejan sintaxis similares para sus guardas, invariantes, etc. Sin embargo los elementos de las listas de invariantes están separados en GraflexedH por ‘&’ y en GraflexedU por ‘,’ (con las asignaciones sobre las aristas ocurre algo similar con ‘:=’ y ‘=’). Esta es una limitación desde el punto de vista del usuario, ya que deberá recordar la sintaxis propia de cada lenguaje
- las traducciones a los lenguajes textuales abren una nueva ventana gráfica (a pesar de que no la usan)
- la implementación actual no chequea la condición de sincronización trivial impuesta por los compiladores (ver la primera parte de este documento)
- no es posible hacer bajas y modificaciones de símbolos
- no es posible borrar ‘bend-points’ de las aristas

Limitaciones heredadas de Graflexed

- problemas de performance
- redibujo de nodos y aristas al cargar un grafo de archivo
- algunas etiquetas de nodos y aristas se superponen

- por una decisión de diseño de Graflexed los menús se desacoplaron de las ventanas de dibujo, de forma que resulta poco práctico tener que buscar el menú asociado a una ventana dada –entre muchos de ellos-.
- las etiquetas de los lazos en general se superponen con otras
- los lazos de un mismo nodo se superponen
- Graflexed fue construido con la biblioteca LEDA 3.7.1 que requiere la versión 2.7.2.1 del compilador *g++*
- el tamaño de las ventanas de edición es fijo
- no es posible minimizar, maximizar ni cerrar las ventanas de Graflexed desde los íconos correspondientes ubicados en la esquina superior derecha de las mismas

5.6. Mejoras futuras

- sintaxis única en invariantes, etiquetas de sincronización (recordar canales de Uppaal) y guardas
- mejorar performance
- mejorar superposición de etiquetas de nodos y aristas
- ayuda sintáctica en línea (recordar que los lenguajes, más allá de la forma de representar elementos de las listas y asignaciones tienen diferencias sintácticas en guardas, invariantes, etc)
- agregar bajas y modificaciones de símbolos
- menús acoplados a las ventanas
- parpadeo al leer archivo gráfico
- entorno único de edición para los tres lenguajes
- portabilidad a ambiente Windows
- agregar borrado de bend-points

5.7. Modificaciones a Graflexed

Se utilizó la herramienta *Graflexed* para generar los editores y multieditores; sobre el propio Graflexed se hicieron los siguientes cambios en '/GRAFLEXED/base/base_editor/BaseGraphEditor.h':

- 'void quit();'se cambio de 'protected' a 'public'
- 'TT& get_edited();' análogo
- 'virtual const graph& get_graph() const;' análogo para 'dar_abiertos'
- cambio en el tamaño de las ventanas de los editores(fijo)
- se eliminó el redibujo del grafo en edición al crearse una arista

6. Un ejemplo: el pasaje de nivel

En esta sección damos un ejemplo de problema de tiempo real, su modelado, su especificación en los lenguajes Kronos, HyTech y Uppaal y la salida de las traducciones de los compiladores. Se incluyen también algunas sesiones de edición de los editores gráficos construidos.

Se trata del modelado de un pasaje de nivel, un ejemplo clásico en la bibliografía de sistemas de tiempo real; en este caso fue tomado de “Comparing verification with HyTech, Kronos and Uppaal on the railroad crossing example”, Research Report LSV-00-2, enero de 2000 de B. Bérard y L. Sierra[BS00].El sistema está compuesto por *N* trenes, una barrera y un controlador. El controlador gobierna el comportamiento de la barrera, de forma de que ella se encuentre cerrada cuando algún tren atraviesa el pasaje de nivel. Contemplaremos el caso de dos trenes.

6.1. Modelización del sistema

El modelo propuesto es el representado en las figuras 6-1, 6-2 y 6-3.

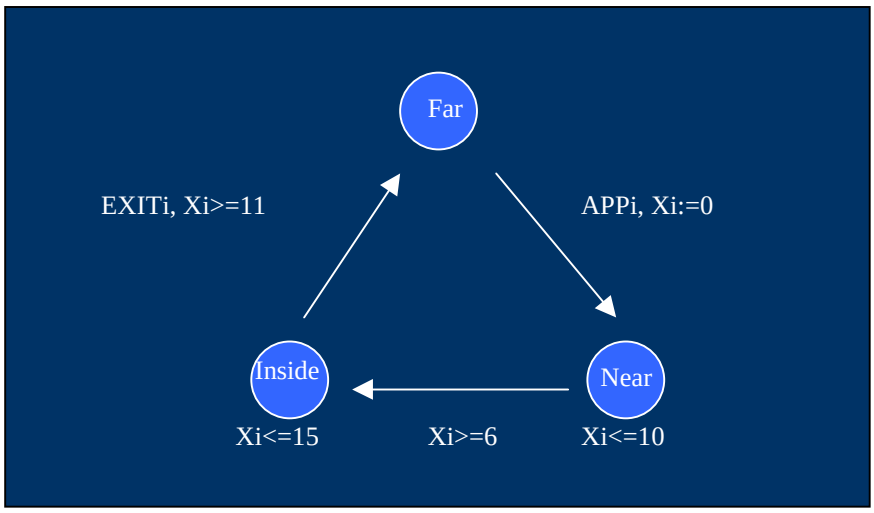


Figura 6-1.Un tren genérico.

El tren se encuentra lejos de la barrera, y se mueve en dirección a ella. Al acercarse se sincroniza con el controlador a través de la etiqueta app_i. Al abandonar el cruce el tren vuelve a sincronizar a través de la $exit_i$. Al recibir estas señales el controlador se sincroniza con la barrera enviando las señales *GoDown* y *GoUp* correspondientes. No será posible usar una variable que implemente un contador, ya que el ejemplo deberá ser representable

en Kronos (que no soporta variables). Por lo tanto, el número de trenes que atraviesan el cruce en un momento dado será representado por los estados *idle*, $C_1,...,C_n$ del controlador; existirá un estado adicional *error*. Los autómatas temporizados serán llamados *train1*, *train2*,..., *gate y control*, y para *n=2* el sistema completo estará dado por el producto sincronizado *train*1Θ*train*2Θ*gate*Θ*control*

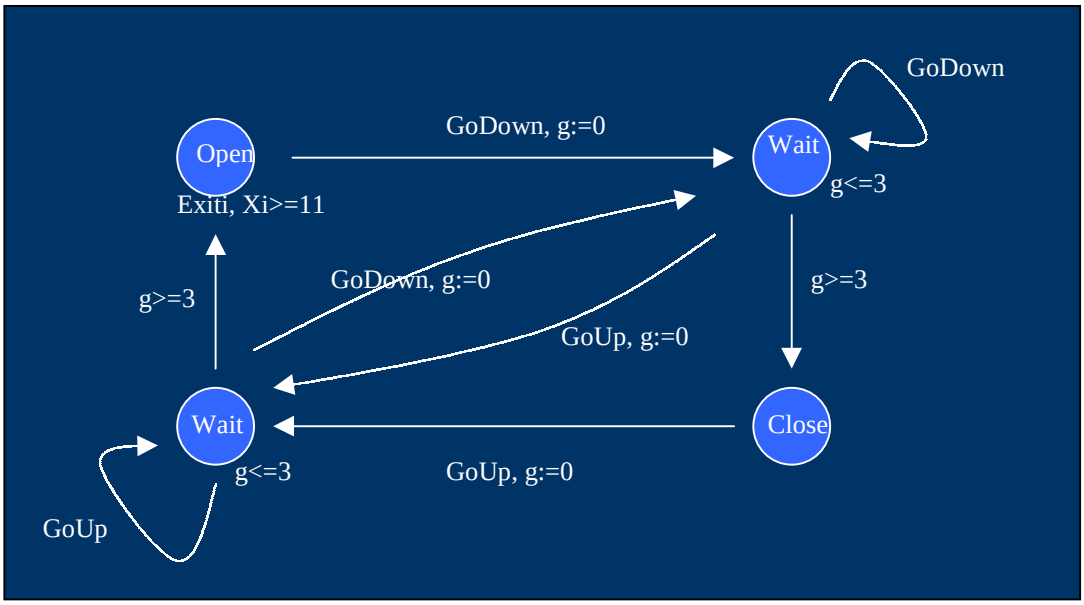


Figura 6-2. La barrera.

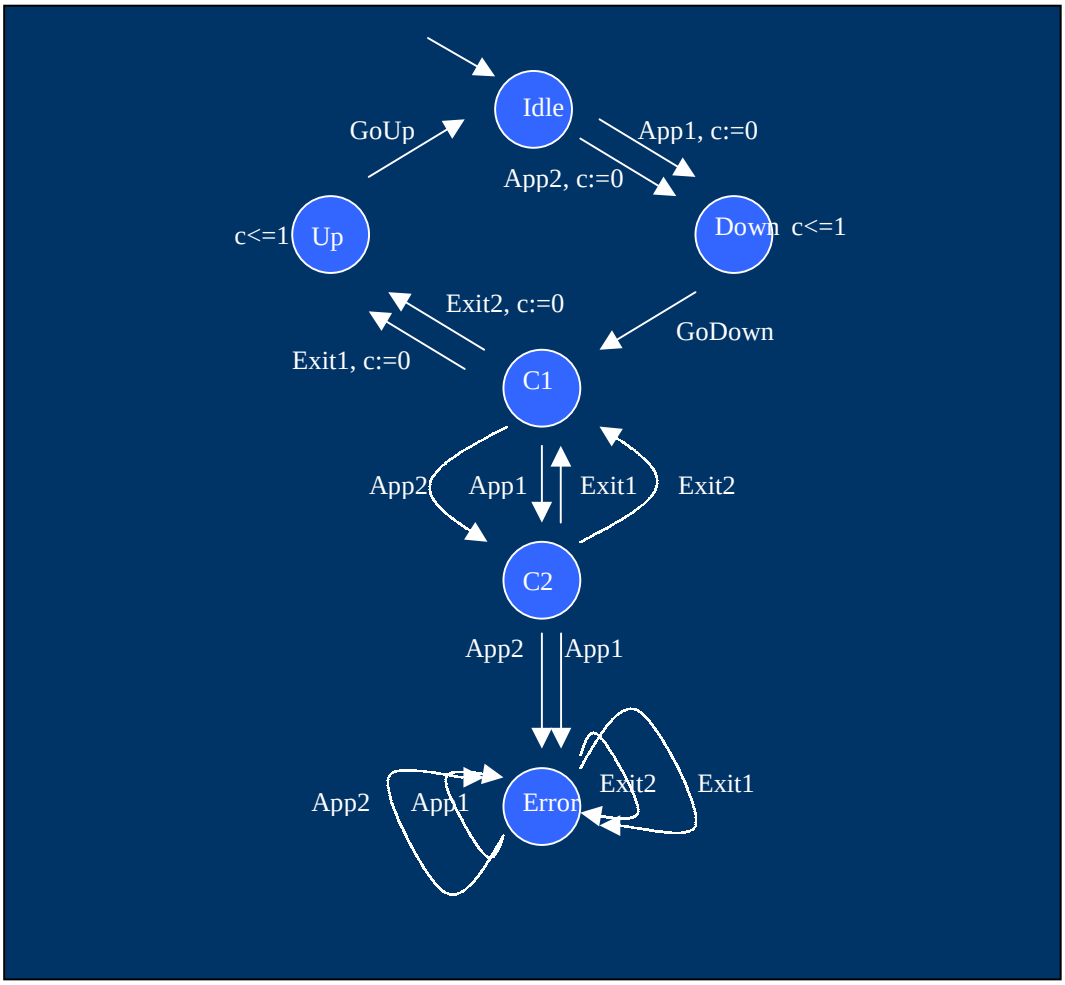


Figura 6-3. El controlador para *n*=2

6.2. Especificación de una propiedad

La especificación será segura si las restricciones temporales impiden que algún tren se encuentre sobre el cruce mientras la barrera no esté cerrada.

6.3. Especificación en HyTech

En este documento incluiremos solamente la especificación del sistema en HyTech, y daremos sus equivalentes en Kronos y Uppaal construidos por los compiladores (las especificaciones en Kronos y Uppaal se incluyen en el CD). Para los tres casos se incluirá una captura de pantalla de los editores de grafos correspondientes. La figura 6-4 muestra una sesión de edición de GraflexedH en la que se puede ver la especificación del autómata que modela el tren número uno. Los casos del otro tren, el controlador y la barrera no se incluyen en este documento (sí en el CD).

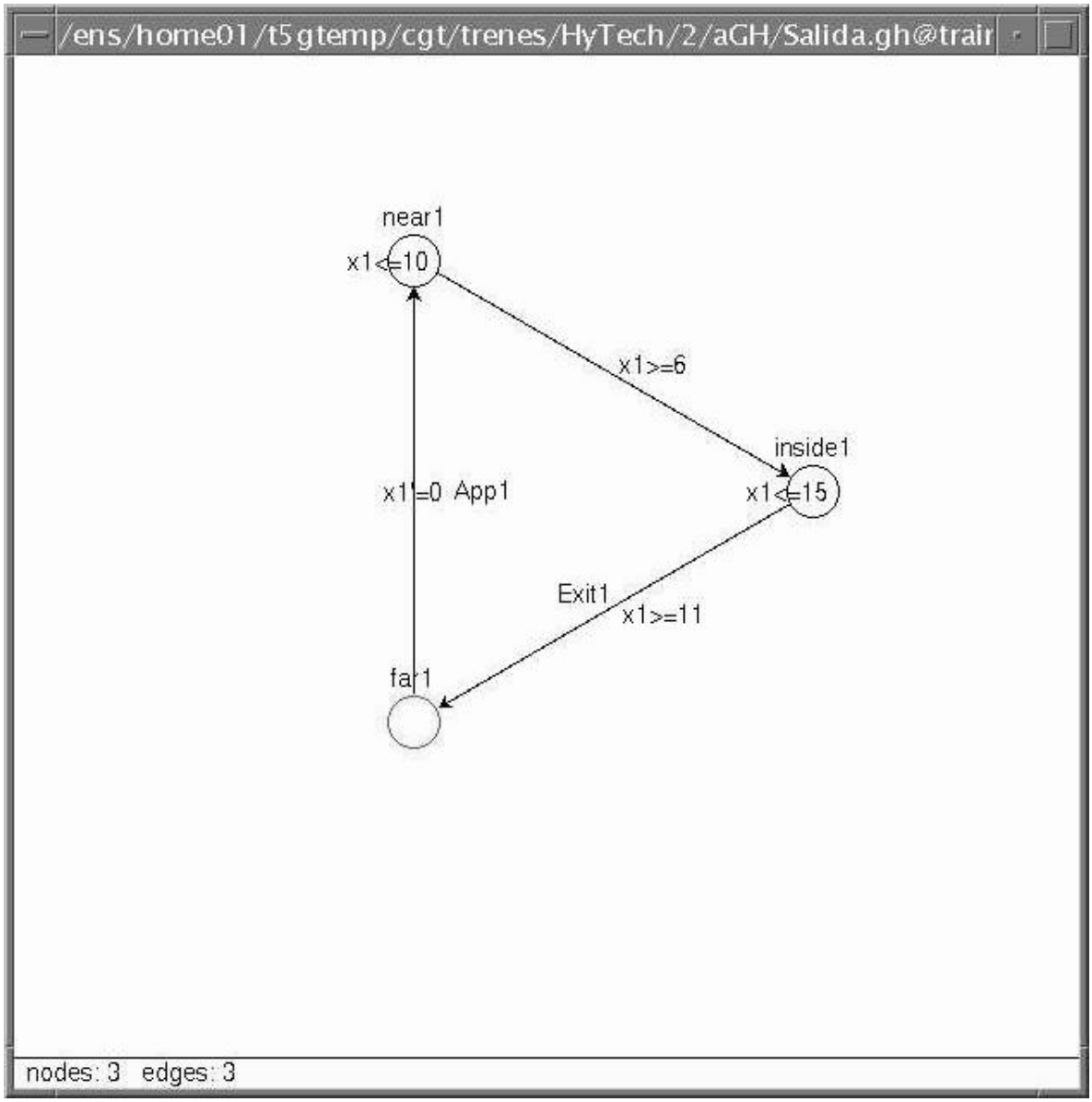


Figura 6-4. El tren en GraflexedH.

La especificación de la propiedad a verificar expresada en HyTech se da en la figura 6-5.

```
var init, cur, sol, bad, x: region;

init := loc[gate] = open & loc [control] = idle & loc [train1] = far1 & loc [train2] = far2;
bad  := reach backward from (loc [train1] = inside1 | loc [train2] = inside2)  & ~loc [gate] = close endreach;
sol  := bad & init;
if empty(sol) then prints "OK, la propidad se verifica";
else prints "NOK, la propiedad NO se verifica";
endif;
```

Figura 6-5. La propiedad a verificar especificada en HyTech

Como se puede ver en la figura 6-6, la salida del verificador de HyTech indica que la propiedad propuesta se verifica.

```
Command: /usr/local/opt/Hytech/hytech sys.hy
================================================================
HyTech: symbolic model checker for embedded systems
Version 1.04a 12/6/96
For more info:
email: hytech@eecs.berkeley.edu
http://www.eecs.berkeley.edu/~tah/HyTech
Warning: Input has changed from version 1.00(a). Use -i for more
info
================================================================

Number of iterations required for reachability: 4
OK, la propiedad se verifica


================================================================
Max memory used =     0 pages =        0 bytes =   0.00 MB
Time spent      =       0.33u +      0.06s =       0.39 sec total
================================================================
```

Figura 6-6. La salida del verificador de HyTech

6.4. Traducciones

Las figuras 6-7 y 6-8 muestran las traducciones del tren a Kronos y Uppaal respectivamente. Para este caso sencillo se puede ver que las especificaciones son muy similares en los tres lengujes. Las diferencias residen en las propiedades atómicas y números de location de Kronos y en la sincronización.

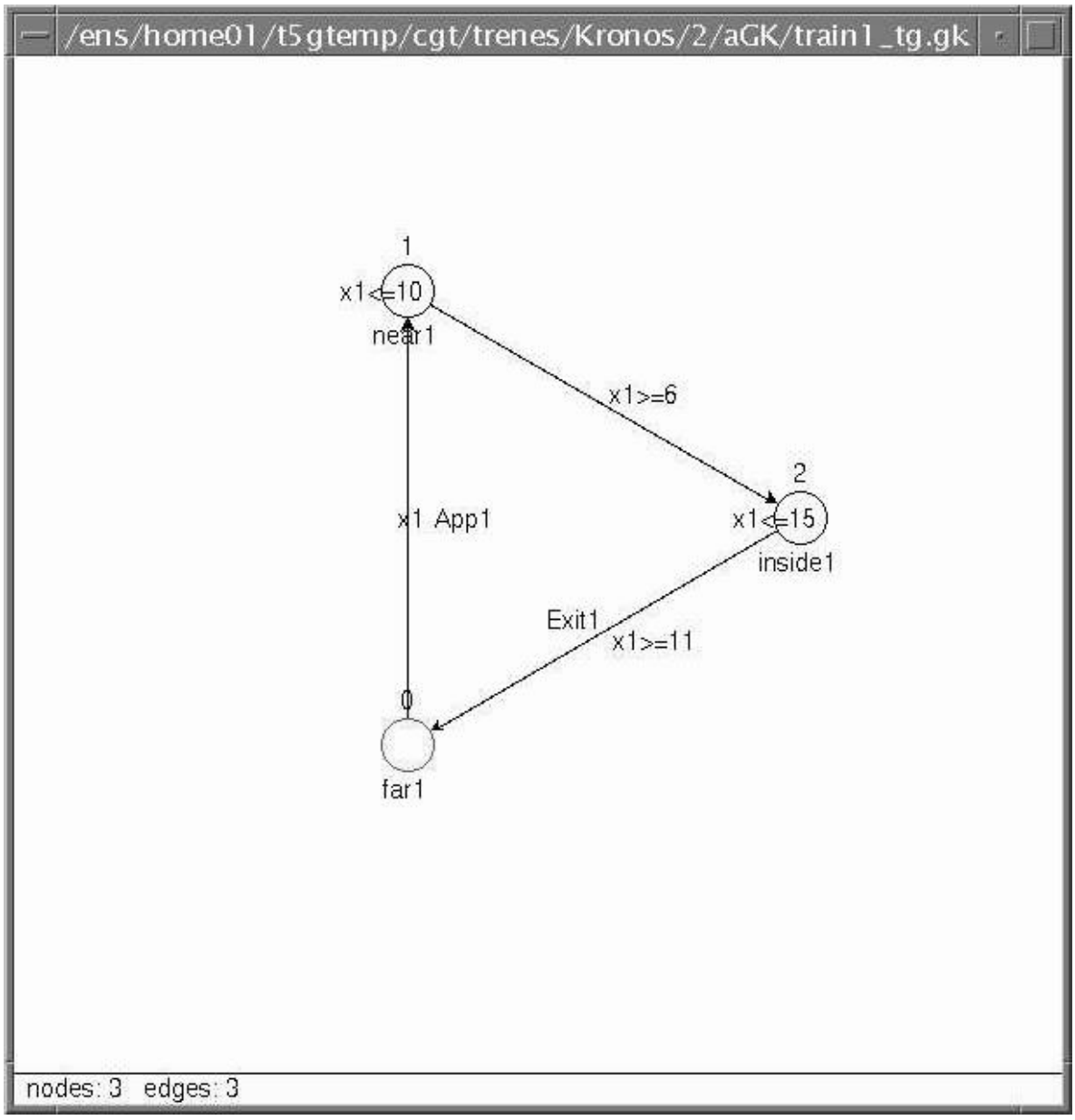


Figura 6-7. El tren en Kronos traducido por el compilador

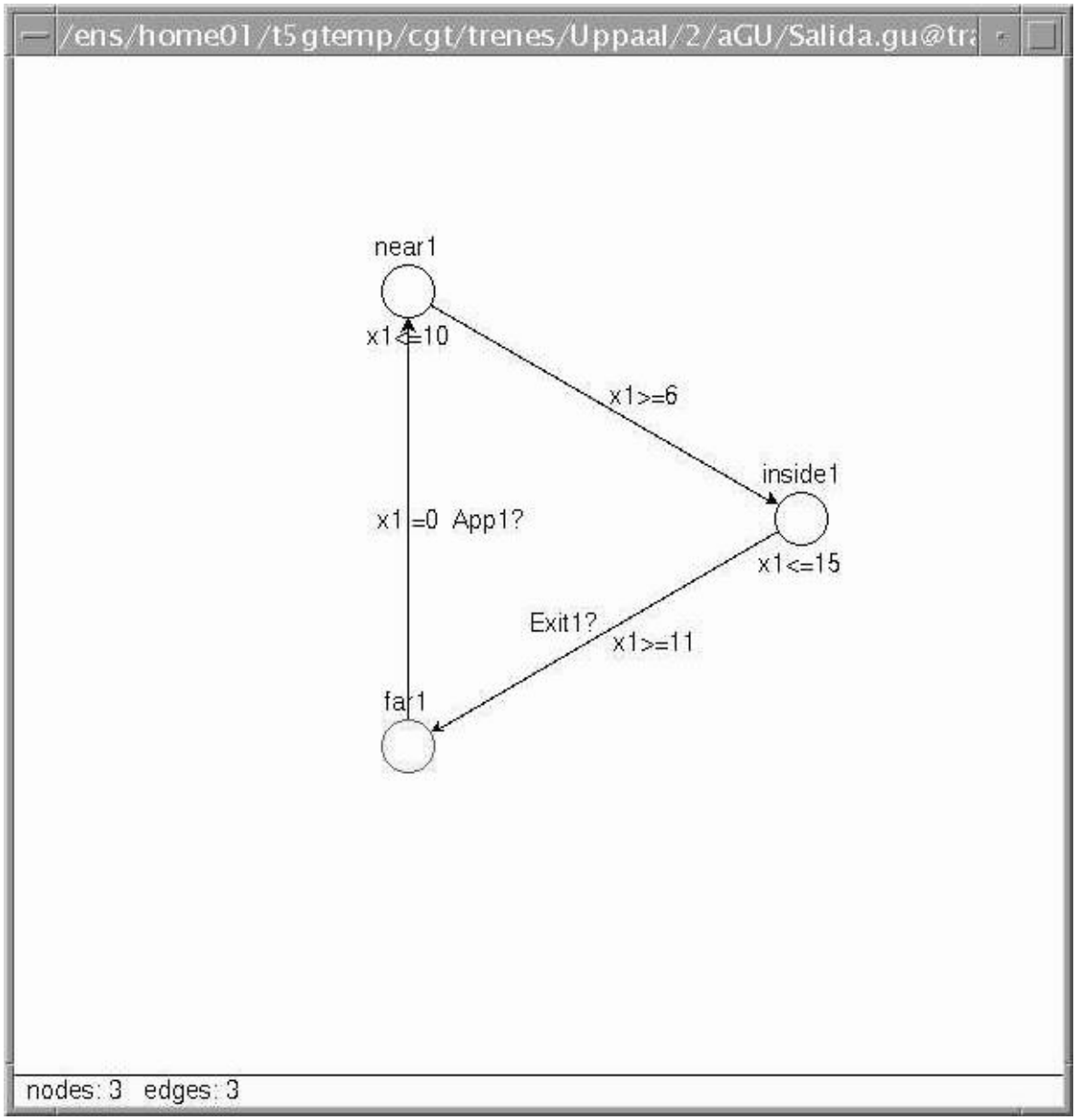


Figura 6-8. El tren en Uppaal traducido por el compilador

La traducción de la propiedad de seguridad se puede ver en las figuras 6-9 y 6-10. Son similares en Kronos y Uppaal, ya que ambos recurren a subconjuntos de TCTL para expresarla. Notar la diferencia con respecto a la especificación en HyTech.

```
(gate_open and control_idle and train1_far1 and train2_far2) impl
ed((train1_inside1 or train2_inside2) and not gate_close)
```

Figura 6-9. La traducción de la propiedad a Kronos

```
E<>((train1.inside1 or train2.inside2) and not gate.close)
```

Figura 6-10. La traducción de la propiedad a Uppaal

```
99 t5gtemp@lulu ~/cgt/trenes/HyTech/2/aK > more Salida.eval
state(1) and (
X1<=10 and C<=1)
or
state(2) and (
X2<=10 and C<=1)
or
state(3) and (
X1<=15 and C<=1)
...
or
state(27) and (
X2<=15 and G<=3 and C<=1)
```

Figura 6-11. Salida del verificador de Kronos para la traducción de la propiedad

El compilador ha traducido la propiedad a una del tipo *EF* en Kronos. La figura 6-11 muestra una porción de la salida del verificador de Kronos sobre la traducción de la propiedad. Se puede ver que la propiedad se verifica para todos los estados del autómata compuesto salvo para el *0*; este último es el representado por los estados iniciales de todos los autómtas miembros. Por lo tanto el verificador expresa que el conjunto de los estados iniciales NO cumple que a partir de ellos existe una ejecución desde la cual es posible llegar a un estado del sistema en que la barrera no esté cerrada y que halla al menos un tren cruzando el pasaje de nivel: se verifica nuestra condición de seguridad.

El caso del verificador de Uppaal es similar. Como se puede ver en la figura 6-12, la propiedad de existencia de una configuración “prohibida” NO se verifica.

```
106 t5gtemp@lulu ~/cgt/trenes/HyTech/2/aU > verifyta Salida.xta Salida.q
UPPAAL version 2.17, Mar 1998 -- verifyta.
Copyright (c) 1995 - 1998, Uppsala University and Aalborg University.
All rights reserved.
Property 1 (line 1) is NOT satisfied.
107 t5gtemp@lulu ~/cgt/trenes/HyTech/2/aU > more Salida.q
E<>((train1.inside1 or train2.inside2) and not gate.close)
108 t5gtemp@lulu ~/cgt/trenes/HyTech/2/aU >
```

Figura 6-12. Salida del verificador de Uppaal para la traducción de la propiedad

7. Conclusiones

La construcción de los compiladores exige un fuerte conocimiento de los lenguajes de especificación de sistemas de tiempo real. Su estudio y la construcción de las traducciones ha abarcado un gran porcentaje del tiempo dedicado a la primera parte de este trabajo.
Los lenguajes de especificación de sistemas de tiempo real estudiados –todos ellos basados en autómatas- están formados por un lenguaje de especificación de sistemas y por un lenguaje de especificación de propiedades. Los lenguajes de especificación de sistemas de las tres herramientas estudiadas –Kronos, HyTech y Uppaal- son similares. Las principales diferencias residen en el manejo de las ideas de urgencia y sincronización. Queda pendiente definir con precisión la semántica de las *urgent locations*, *urgent channels* y *commited locations* de Uppaal, la de las guardas *asap* de HyTech, y proponer un esquema de traducción. La traducción de los esquemas de sincronización es por su complejidad un tema en sí mismo. La semántica propuesta por Uppaal es bien distinta a la propuesta por Kronos y HyTech; la traducción corre el riesgo de caer en la pérdida de claridad, ya que puede implicar la modificación e incluso el agregado de autómatas. Los autómatas híbridos no fueron contemplados por este trabajo, pero se propuso un enfoque para los lineales.
Con respecto a los lenguajes de especificación de propiedades se presentan dos situaciones distintas. Por una parte, Kronos y Uppaal son similares en este sentido ya que ambos implementan subconjuntos de TCTL. Kronos tiene un poder expresivo mayor, que hace que muchas propiedades en la traducción Kronos $\rightarrow$ Uppaal deban ser descartadas. Por otra parte HyTech propone un punto de vista totalmente diferente. En lugar de recurrir a una lógica temporal se basa en un enfoque algorítmico; provee un conjunto de comandos con los que el usuario deberá construir programas basados en la idea de alcanzabilidad para expresar las propiedades. La traducción HyTech $\rightarrow$ Kronos-Uppaal asocia ciertas construcciones predefinidas de dicho lenguaje (especificadas en la sección A.4.) a fórmulas TCTL. El sentido inverso de la traducción requiere un estudio aparte que no es abarcado por este trabajo.
Dadas las dificultades mencionadas, en la sección 4.6 se propuso un enfoque alternativo: la introducción de un nuevo formalismo de poder expresivo mayor, desde el cual el usuario puede acceder a los verificadores de las herramientas existentes soportadas.
Sin embargo, un gran porcentaje de los ejemplos existentes en la literatura (en particular los incluidos en las distribuciones de Kronos, HyTech y Uppaal) son soportados por los compiladores construidos.
Los compiladores permiten crear archivos de entrada para las herramientas GraflexedU, GraflexedH y GraflexedK a partir de especificaciones Uppaal, HyTech y Kronos respectivamente.

La segunda parte de este trabajo se ha basado en el uso de la herramienta Graflexed, lo que ha permitido hacer una serie de apreciaciones sobre la misma.
Graflexed permite construir editores de grafos independientes de la interfaz gráfica usada.
Sin embargo esto implica algunas desventajas. Una de ellas es la pérdida de performance, no menor para grafos grandes. Otra tiene que ver con el manejo de los menús. Cada editor está formado por una ventana donde se despliega el grafo editado, y por otra, **independiente**, que implementa un menú. Esto hace desagradable trabajar con más de un editor a la vez.

El manejo de las etiquetas de nodos y aristas presenta algunos problemas. La posición relativa de las etiquetas respecto a los nodos y aristas es fija y en muchos casos hace que las etiquetas se superpongan, con lo que se pierde claridad.
El redibujo de los grafos genera "parpadeos" desagradables; deben hacerlo los editores en forma explícita, en lugar de ser resuelto en forma automática como se sugiere en la documentación de Graflexed.

Hablemos ahora de la implementación de la EditLayoutInterface provista por Graflexed, *LedaELI*, construida sobre la biblioteca gráfica de LEDA.
Las aristas son representadas por rectas o poligonales; la existencia de curvas mejoraría el aspecto de los grafos, sobre todo en grafos grandes.
LEDA presenta algunos problemas. Algunos *bugs* hacen que las ventanas de los grafos deban refrescarse una gran cantidad de veces. A su vez nuevas versiones de la biblioteca cambian la sintaxis de algunas funciones, lo que hace que se deba modificar código de Graflexed. Además, la versión de LEDA usada por Graflexed es incompatible con nuevas versiones del compilador g++. Esto ha implicado trabajar bajo Linux 2.0.35. Sin duda es necesario adaptar Graflexed a nuevas versiones de LEDA y por supuesto de g++.

A pesar de las facilidades provistas por Graflexed la implementación de los editores no ha sido sencilla. Ha exigido volver a los manuales una gran cantidad de veces, recurrir a ejemplos.

LEDA merece un comentario extra. Además de la interfaz gráfica mencionada provee una completísima biblioteca de tipos de datos y algoritmos, como ser listas, conjuntos, diccionarios, grafos y algoritmos sobre ellos que sin duda han facilitado la implementación.

Los editores construidos cumplen el objetivo planteado. Permiten editar un subconjunto importante de los grafos temporizados definidos por Kronos, HyTech y Uppaal. Hacen posible salvar los grafos en edición tanto en el lenguaje textual de la herramienta sobre la que se está trabajando como de los de cualquiera de las otras dos.
Algunos aspectos que deben mejorarse son la performance, los menús desacoplados, el manejo de las tablas de símbolos y el redibujo.

8. Referencias

[Alu94] R. Alur, C. y D.L. Dill. A theory of timed automata. TCS, 126:183-235, 1994.

[Bae91] J.C.M. Baeten y J.A. Bergstra. Real time process algebra. Formal Aspects of Computing 3(2), 1991, 142-188.

[BS00] B. Bérard y L. Sierra . "Comparing verification with HyTech, Kronos and Uppaal on the railroad crossing example", Research Report LSV-00-2, enero de 2000.

[Daw95] C. Daws y S. Yovine. Two examples of verification of multirate timed automata with Kronos. In Proc. 16th IEEE Real-Time Systems Symposium (RTSS '95), Pisa, Italy, Dec. 1995, páginas 66-75. IEEE Comp. Soc. Press, 1995.

[Eme90] E.A. Emerson. Temporal and modal logic. En J. Van Leeuwen, Handbook of theoretical Computer Science, vol. B, capítulo 16, 995-1072. Elsevier Science Publishers, 1990.

[Hei96] Constance Heitmeyer y Dino Mandrioli. Formal Methods for Real-Time Computing. Wiley, 1996.

[Hen+96] T. A. Henzinger, Pei-Hsin Ho, y H Wong-Toi. A User Guide to HyTech, disponible en http://www.cad.eecs.berkeley.edu/~tah/Publications/a_user_guide_to_hytech.html, University of California, Berkeley, USA, octubre 1996.

[HyT] http://www-cad.eecs.berkeley.edu/~tah/HyTech/. University of California, Berkeley, USA.

[Kro] http://www-verimag.imag.fr/TEMPORISE/kronos/kronos.html. Verimag, Grenoble, Francia.

[LED] http://www.mpi-sb.mpg.de/LEDA/leda.html, Algorithmic Solutions Software GmbH (AS).

[Oli94] A. Olivero. Modélisation et Analyse de Systemes Temporisés et Hybrides. Tesis de doctorado del Instituto Nacional Politécnico de Grenoble, 1994.

[OSY94] A. Olivero, J. Sifakis y S. Yovine. Using abstractions for the verification of linear hybrid systems". In Proc. 6th Int. Conf. on Computer Aided Verification, number 818 in Lecture Notes in Computer Science, Springer Verlag, pages 81-94. Springer Verlag, 1994.

[Par+98] B. Paroli y F. Purtscher. Proyecto de Taller V 1998: Ambiente gráfico para STA, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay, 1998.

[Sch+99] P. Schnoebelen, B. Bérard, M. Didoit, F. Laroussinie, A. Petit. Vérification de logiciel, Techniques et outils du model-checking Vuibert Informatique, 1999.

[Upp] http://www.docs.uu.se/docs/rtmv/uppaal/start.html. Department of Computer Science, Uppsala Universitet, Suecia.

Anexo A: Sublenguajes aceptados

Este anexo da una descripción completa de los lenguajes aceptados por los compiladores.

A.1. La representación intermedia M

La presente sección describe la representación intermedia M. Su estructura quedó determinada por la de los lenguajes de especificación de las herramientas estudiadas; por lo tanto se da, para cada decisión a tomar, un breve resumen del enfoque adoptado por cada herramienta.

Nombres de los estados de control

- K: los estados de control son identificados por naturales
- U: los estados de control son identificados por nombres
- H: los estados de control son identificados por nombres

- M: los estados de control serán identificados por nombres

Propiedades atómicas en Kronos

Kronos permite la declaración de propiedades atómicas, que se verifican en un conjunto de nodos. Así, una declaración del tipo *prop=lejos* indica que la propiedad atómica *lejos* se verifica en el estado en el que aparece la declaración. Estas sentencias son útiles al momento de especificar propiedades a verificar.
Son traducidas en HyTech y Uppaal como un predicado que indica el autómata y el estado en que se verifican (una disyunción si la misma propiedad se verifica en más de un estado). Supongamos, por ejemplo, que la propiedad *lejos* se verifica en el estado *0* (que fue traducido como *est_lejos*) del autómata *Tren*. Así en Uppaal la propiedad será traducida como *Tren.est_lejos* y en HyTech como *loc[Tren]=est_lejos*.

Variables y relojes

▪ K:	clock	derivada siempre 1
▪	variables	no se definen
▪ U:	clock	derivada siempre1
	variables int	derivada 0 (no se soportan racionales ni reales, sólo enteros)
▪ H:	clock	derivada siempre 1
	stopwatch	derivada 0 o 1 (se comportan como constantes o relojes)
	analog	sin restricciones sintácticas (reales)

integrator	asociadas a los autómatas híbridos
discrete	derivada 0 (sólo cambia el valor a través de las asignaciones, no es un reloj; puede tomar valores reales)
parameter	derivada 0 siempre, no se les puede asignar valores (usada en análisis paramétricos)

- M: sólo relojes y variables enteras (ni racionales ni reales).

Valores a asignar

- K: el valor a asignar a los relojes es el cero solamente ("resets"; por otra parte, **no** se definen variables)
- H: los valores a asignar a los relojes y variables son combinaciones lineales (con coeficientes racionales) de variables.
- U: los valores a asignar a los relojes son naturales, y a variables enteras, expresiones enteras (en las que pueden intervenir otras variables), como se puede ver en la figura A-1.

- M: aceptará las asignaciones que se muestran en la figura A-2.
  Notar que la sintaxis descrita por M es soportada por HyTech y Uppaal; sin embargo las traducciones M $\rightarrow$ Kronos deberán verificar que las asignaciones de M sean sólo de la forma 'clock=0'; esto permite conservar poder expresivo para HyTech y Uppaal sin restringirse a Kronos.

```
AList  :    (vacía)
       |    Assign , Alist
       ;

Assign :    Iassign
       |    Cassign
       ;

CAssign:    ID := NAT
       ;

IAssign:    ID := Iexpr
       ;

IExpr  :    ID
       |    ID[IExpr]
       |    NAT
       |    -Iexpr
       |    (IExpr)
       |    IExpr OP Iexpr
```

```
        |       (IGuard ? IExpr : IExpr)
        ;

OP      :       + | - | * | /
        ;
```

Figura A-1. Sintaxis de las asignaciones en Uppaal.

```
AsigHU:     NAME IGUAL NAT
        |   NAME IGUAL intExpr
        ;

intExpr:    integer MULT  NAME SUMA  NAT
        |   integer MULT  NAME RESTA NAT
        |   NAME    SUMA  NAT
        |   NAME    RESTA NAT
        |   NAME
        |   integer
        ;

integer :   NAT
        |   - NAT
        ;
```

Figura A-2. Sintaxis de las asignaciones en M.

Inicialización de variables

- K: no existen variables, por lo que no tiene sentido
- H: no es posible inicializar variables al declararlas
- U: sí es posible inicializar variables al declararlas

- M: se descartará la inicialización. Se supondrá inicialización implícita a cero.

Cantidad de asignaciones por transición

- K: se puede dar una lista de *N* relojes a "resetear", lo que se puede ver como una lista de *N* asignaciones del tipo 'clock=0'
- U: soporta asignaciones a *N* relojes o variables
- H: soporta asignaciones a *N* relojes o variables

- M: lista de *N* asignaciones a relojes o variables. La traducción M $\rightarrow$ Kronos descartará las asignaciones a variables, ya que Kronos soporta solamente relojes.

Scope de variables y relojes

- K: las declaraciones de relojes en Kronos se hacen a nivel de cada autómata temporizado. El scope de los relojes es por lo tanto el archivo '.tg' en que se declaran, y el/los archivos '.tctl' que expresen propiedades del sistema. En este sentido se puede decir que no existe la idea de variable local a un autómata. Sin embargo, si más de un autómata declara un reloj con el mismo nombre, todos estarán haciendo referencia al mismo reloj.
- U: Uppaal soporta dos tipos de declaraciones de variables y relojes. Las variables y relojes locales tienen por scope el autómata que las declara; referencias a estas variables dentro del autómata ignoran la existencia de variables globales del mismo nombre.
  Las variables y relojes globales tienen por scope todo el archivo '.xta' en que se declaran (es decir, todos los autómatas del mismo) y el/los archivos '.q' que expresen propiedades del sistema.
- H: La declaración de variables y relojes en HyTech es común a todos los grafos descritos en un mismo archivo '.hy', y su scope es todo el archivo (es decir, todos los autómatas). No existe la noción de variable o reloj local a un autómata.

- M: contemplará variables y relojes locales y globales.

Inicialización de los autómatas

- K: el estado inicial de cada autómata es aquel que tiene 'prop:init', o 'loc:0' en caso de que ninguno verifique la condición anterior. Si hay más de una el sistema elige una en forma no determinística; si no hay ninguna, se elige la cero.
- U: se declara el estado inicial como 'init ID', no se declaran condiciones iniciales de las variables. También se puede declarar 'final ID', el que no será tenido en cuenta.
- H: se especifica explícitamente el estado inicial de cada autómata, y además se pueden inicializar los relojes y variables según una serie de predicados convexos como en el caso de las asignaciones en las transiciones.

- M: contemplará solamente el estado inicial de cada autómata. Se podrían simular las asignaciones a variables en U con la introducción de un nuevo estado inicial con invariante y guarda vacíos y que en la transición se hicieran las inicializaciones mencionadas. No tiene sentido agregar complejidad si el propio U no brinda esta facilidad.

Invariantes

- K: un invariante es un par de relojes (a lo sumo) vinculados por un operador relacional, según la gramática de la figura A-3.
- U: un invariante es una lista de guardas simples sobre relojes de la forma ID [<, <=]NAT, como se muestra en la figura A-5. Los invariantes pueden ser también vacíos, en cuyo caso se los supone verdaderos.
- H: un invariante es una lista de pares de combinaciones lineales de relojes y variables (con coeficientes racionales) vinculadas por un operador relacional (un predicado convexo), como se describe en la figura A-4.

- M: aceptará una lista de invariantes al estilo de K; de esta forma se contempla el caso de U (aunque se deberán descartar algunos casos), K se restringirá sólo al primero y H los aceptará.
  Se podrá dar el caso de que una traducción K $\rightarrow$ H sea posible sintácticamente, y sin embargo que una K $\rightarrow$ U (para el mismo sistema Kronos) no; en lugar de construir un lenguaje intermedio que garantice que funcionarán ambas traduciones o ninguna (como forma de facilitar el pasaje entre lenguajes sin sucesivas correcciones sintácticas) se optó por no restringir la expresividad de los invariantes de K y U.

```
ClockConstraint:    clock  rel    nat
                  |  clock  rel    clock
                  |  clock  rel    clock  +    nat
                  |  clock  rel    nat    +    clock
                  |  nat    rel    clock
                  |  nat    +      clock  rel  clock
                  |  clock  -      clock  rel  nat
                  |  TRUE
                  ;

rel               :  < | <= | = | > | >=
                  ;
```

Figura A-3. Sintaxis de invariantes y guardas en Kronos.

```
ConvexPredicate:      linear_constraint
                    |  linear_constraint & linear_constraint
                    ;

linear_constraint:    linear_expression relop linear_expression
                    |  True
                    |  False
                    |  Asap
```

```
                ;

linar_expression:   linear_term
                |   linear_expression + linear_term
                |   linear_expression - linear_term
                ;

linear_term     :   rational
                |   rational name
                |   name
                ;

relop           :   < | <= | = | > | >=
                ;
```

Figura A-4. Sintaxis de los invariantes en HyTech.

```
InvList         :   Inv
                |   Inv , InvList
                ;

Inv             :   name rel nat
                ;
rel             :   < | <=
                ;
```

Figura A-5. Sintaxis de los invariantes en Uppaal.

Guardas

- K: invariantes y guardas tienen igual sintaxis
- U: se pueden dar *N* guardas según se describe en la figura A-6.
- H: invariantes y guardas tienen igual sintaxis

- M: aceptará guardas según la figura A-7. Nuevamente, con esta gramática se preserva expresividad para las traducciones H<->U, en lugar de restringirnos a la de K, más pobre.

```
GList   :   (vacía)
        |   CGuard , GList
        |   IGuard , Glist
        ;
```

```
CGuard:     ID REL NAT
        |   ID REL ID
        |   ID REL ID + NAT
        |   ID REL ID - NAT
        ;

REL     :   < | <= | >= | > | ==
        ;

IGuard  :   IExpr  REL   IExpr
        |   IExpr  !=    Iexpr
        ;

IExpr   :   ID
        |   ID[IExpr]
        |   NAT
        |   - Iexpr
        |   (IExpr)
        |   IExpr OP Iexpr
        |   (IGuard ? IExpr: IExpr)
        ;

OP      :   + | - | * | /
        ;
```

Figura A-6. Sintaxis de las guardas en Uppaal.

```
GuardaHU:   (vacía o TRUE)
        |   Expr Relop Expr
        ;

Expr    :   NAME
        |   INT
        |   INT * NAME
        |   Expr + Expr
        |   Expr - Expr
        ;

Relop   :   < | <= | = | > | >=
        ;
```

Figura A-7. Sintaxis de las guardas para Uppaal y HyTech en M.

Sincronización

- K: Las aristas en Kronos pueden ser decoradas con *N* etiquetas de sincronización.
  Un conjunto de transiciones de sincronización pueden ser ejecutadas simultáneamente si cada una de sus etiquetas de sincronización es ejecutada también por cada uno de los autómatas que la declaró. En este sentido se dice que el conjunto de transiciones de sincronización es maximal, ya que sincronizarán todos los autómatas que declararon la etiqueta.
  De esta forma autómatas que no declararon una etiqueta de sincronización no participarán en la sincronización.
- U: Uppaal define los canales como mecanismo de sincronización, basado en el modelo de transmisión de mensajes.
  Dado un canal 'a' es posible etiquetar las transiciones de los autómatas con 'a!' para el envío y 'a?' para la recepción. Toda transición puede tener a lo sumo una etiqueta de sincronización. Este esquema de sincronización es binario: una transición con la etiqueta 'a?' sincronizará con solamente una transición etiquetada con 'a!' y viceversa. Si más de una transición 'a?' espera por una 'a!', solamente una de ellas -elegida en forma no determinística- sincronizará, la otra continuará bloqueada en espera de la ejecución de una nueva etiqueta de sincronización 'a!'.
- H: el esquema de sincronización de HyTech es similar al de Kronos.
  Las transiciones también se decoran con etiquetas de sincronización (declaraciones de tipo 'synclab') con la diferencia de que se admite sólo **una** por transición.
  En forma similar a Kronos un conjunto de transiciones asociado a una etiqueta de sincronización ejecutará en forma simultánea sólo si cada uno de los autómatas que la declara ejecuta también una transición decorada con la misma etiqueta de sincronización. En este caso sincronizan también todos los autómatas a la vez.

- M: soporta un esquema de sincronización que llamaremos **sincronización trivial.**
  La sincronización trivial contempla solamente un conjunto restringido de formas de sincronización. Más abajo se detalla la complejidad de la traducción de otros casos más generales.
  Se permitirá solamente una etiqueta de sincronización por transición, y solamente dos autómatas podrán sincronizarse a la vez. Esto significa que una etiqueta de sincronización podrá estar definida en a lo sumo dos autómatas distintos.

A.2. Kronos

Especificación del sistema

La figura A-8 muestra la subgramática de Kronos reconocida por el compilador.

graph	: NUMERAL	LOCS	NAT
	NUMERAL	TRANS	NAT

```
                    NUMERAL  CLOCKS     listofclocks
                    NUMERAL  SYNC       listoflabels
                    listofstates
                  ;

listoflabels      :
                  | NAME listoflabels
                  ;

listofclocks      :
                  | NAME listofclocks
                  ;


listofstates      :
                  | state listofstates
                  ;

state             : LOC       DOSPTOS   NAT
                    PROP      DOSPTOS   listofprops
                    INVAR     DOSPTOS   clockconstraint
                    TRANS     DOSPTOS   listoftransitions
                  ;

listofprops       :
                  | NAME listofprops
                  ;

listoftransitions :
                  | trans listoftransitions
                  ;

trans             : clockconstraint IGUAL MAYOR NAME PTOYCOMA RESET ILLAVE
                  listofclocks DLLAVE PTOYCOMA GOTO NAT
                ;
```

Figura A-8. Subgramática Kronos del lenguaje de especificación de sistemas aceptada por el compilador de grafos temporizados.

Especificación de propiedades

La gramática aceptada por el compilador es la dada en la figura A-9.

```
Formula          : TRUE
                 | clockconstraint
                 | NOT formula
                 | formula      AND   formula
                 | formula      OR    formula
                 | formula      IMPL  formula
                 | AD bound formula
                 | AB bound formula
                 | IPAR formula DPAR
                 | NAME
                 ;

clockconstraint  : NAME                              rel   NAT
                 | NAME                              rel   NAME
                 | NAME                              rel   NAME  SUMA  NAT
                 | NAME                              rel   NAT   SUMA
                 | NAT                               rel   NAME
                 | NAT          SUMA        NAME     rel   NAME
                 | NAME         SUMA        NAT      rel   NAME
                 | NAME         RESTA       NAME     rel   NAT
                 | NAT                               rel   NAME RESTA NAME
                 | TRUE
                 | TRUEP
                 ;

bound            :
                 | interval
                 | ILLAVE rel NAT DLLAVE
                 ;

interval         : DCORCH   NAT COMA NAT   ICORCH
                 | DCORCH   NAT COMA NAT   DCORCH
                 | ICORCH   NAT COMA NAT   ICORCH
                 | ICORCH   NAT COMA NAT   DCORCH
                 ;

rel              : MENOR
                 | MENORIGUAL
                 | MAYOR
                 | MAYORIGUAL
```

```
                    | IGUAL
                    ;
```

Figura A-9. Subgramática Kronos del lenguaje de especificación de propiedades aceptada por el compilador de grafos temporizados.

A.3. Uppaal

Especificación del sistema

La figura A-10 muestra la subgramática de Uppaal aceptada por el compilador.

```
Ita             : declGlob ProcList Globals
                ;

declGlob        :
                | Channel declGlob
                | VarGlob declGlob
                ;

declLoc         :
                | VarLoc declLoc
                ;

ProcList        : Proc
                | Proc ProcList
                ;

Globals         : SYSTEM IdList PTOYCOMA OpHide
                | OpHide SYSTEM IdList PTOYCOMA
                ;

Channel         : CHAN IdList     PTOYCOMA
                ;

VarGlob         : Type IdList     PTOYCOMA
                ;

VarLoc          : Type IdListPref PTOYCOMA
                ;

 Proc           : PROCESS ID ILLAVE declLoc StateDecls TransDecls DLLAVE
                ;
```

```
IdList          : ID
                | ID COMA IdList
                ;

IdListPref      : ID
                | ID COMA IdList
                ;

OpHide          :
                | HIDE IdList PTOYCOMA
                ;

StateList       : ID
                | ID ILLAVE IList DLLAVE
                | ID COMA StateList
                | ID ILLAVE IList DLLAVE COMA StateList
                ;

IList           : Inv
                | Inv COMA IList
                ;

Inv             : ID MENOR          NAT
                | ID MENORIGUAL  NAT
                ;


StateDecls      : STATE StateList PTOYCOMA INIT ID PTOYCOMA
                | STATE StateList PTOYCOMA INIT PTOYCOMA FINAL    PTOYCOMA
                ;

TransDecls      : TRANS TransList PTOYCOMA
                ;

TransList       : Trans
                | Trans COMA TransList
                ;

trans           : ID RESTA MAYOR ID ILLAVE OpGuard OpSync OpAssign DLLAVE
                ;

OpGuard         :
                | GUARD GuardList PTOYCOMA
                ;

OpSync          :
                | SYNC ID SEND PTOYCOMA
```

```
                | SYNC ID RECV PTOYCOMA
                ;

GuardList       : Guard
                | Guard COMA GuardList
                ;

OpAssign        :
                | ASSIGN AssignList PTOYCOMA
                ;

AssignList      : Assign
                | Assign COMA AssignList
                ;

Assign          : ID DOSPTOS IGUAL NAT
                | ID DOSPTOS IGUAL intExpr
                ;

intExpr         : int MULT   ID SUMA    NAT
                | int MULT   ID RESTA   NAT
                |            ID SUMA    NAT
                |            ID RESTA   NAT
                |            ID
                |            int
                ;

Type            : CLOCK
                | INT
                ;

Guard           : GuardTerm RelTodos GuardTerm
                ;

GuardTerm       : ID
                | int
                | int  MULT  ID
                | GuardTerm SUMA  GuardTerm
                | GuardTerm RESTA GuardTerm
                ;

RelOp           : MENORIGUAL
                | MAYORIGUAL
                | IGUALRELOP
                ;

RelTodos        : MENORIGUAL
```

```
                    | MENOR
                    | MAYORIGUAL
                    | MAYOR
                    | IGUALRELOP
                    ;

OP                  : SUMA
                    | RESTA
                    | MULT
                    | DIV
                    ;

 int                : NAT
                    | RESTA NAT
                    ;
```

Figura A-10. Subgramática Uppaal del lenguaje de especificación de sistemas aceptada por el compilador de grafos temporizados.

Especificación de propiedades

Las fórmulas Uppaal traducidas son las reconocidas por la gramática descrita por la figura A-11.

```
TopProp     : Prop
            |  NOT Prop
            ;

Prop        : E MENOR  MAYOR   StateProp
            | A ICORCH DCORCH  StateProp
            ;

 StateProp  : AtomicProp
            | NOT StateProp
            | IPAR StateProp DPAR
            | StateProp OR    StateProp
            | StateProp AND   StateProp
            | StateProp IMPLY StateProp
            ;

 AtomicProp : ID RelOp NAT
            | ID PUNTO ID RelOp NAT
            | ID PUNTO ID
```

```
                ;

RelOp           : MENORIGUAL |  MAYORIGUAL |  IGUALRELOP
                ;
```

Figura A-11. Subgramática Uppaal del lenguaje de especificación de propiedades aceptada por el compilador de grafos temporizados.

A.4. HyTech

Especificación del sistema

La figura A-12 da la subgramática de HyTech reconocida por el compilador.

```
Automata_descriptions: declarations automata
                ;

declarations    : VAR var_lists
                ;

var_lists       :
                | var_list DOSPTOS var_type PTOYCOMA var_lists
                ;

var_list        : NAME
                | NAME COMA var_list
                ;

var_type        : CLOCK
                | DISCRETE
                ;

automata        : automaton automata
                | automaton
                ;

automaton       : AUTOMATON NAME prolog locations END
                ;

prolog          : initialization sync_labels
                | sync_labels initialization
                ;
```

```
initialization      : INITIALLY NAME PTOYCOMA
                    ;

sync_labels         : SYNCLABS DOSPTOS sync_var_list PTOYCOMA
                    ;


sync_var_list       :
                    | sync_var_nonempty_list
                    ;

sync_var_nonempty_list: NAME COMA sync_var_nonempty_list
                    | NAME
                    ;

locations           :
                    | location locations
                    ;

location            : LOC NAME DOSPTOS WHILE convex_predicate WAIT ILLAVE
                    rate_info_list DLLAVE transitions
                    ;

rate_info_list      :
                    | rate_info_nonempty_list
                    ;

rate_info_nonempty_list: rate_info
                    ;

rate_info           : NAME IN ICORCH integer COMA integer DCORCH
                    ;

transitions         :
                    | transition transitions
                    ;

transition          : WHEN GuardList update_synchronization
                      GOTO NAME PTOYCOMA
                    ;

update_synchronization: updates sync_label
                    | sync_label updates
                    ;

updates             :
                    | DO ILLAVE update_list DLLAVE
```

```
                    ;

update_list         :
                    | update COMA update_list
                    | update
                    ;

update              : NAME PRIMA      IGUAL NAT
                    | NAME            IGUAL NAT
                    | NAME PRIMA      IGUAL intExpr
                    | NAME IGUAL
                    ;

intExpr             : integer MULT    NAME SUMA  NAT
                    | integer MULT    NAME RESTA NAT
                    |                 NAME SUMA  NAT
                    |                 NAME RESTA NAT
                    |                 NAME
                    |                 integer
                    ;

GuardList           : TRUE
                    | Guard
                    | Guard AMP GuardList
                    ;

Guard               : GuardTerm relop GuardTerm
                    ;

GuardTerm           : NAME
                    | integer
                    | integer   MULT  NAME
                    | GuardTerm SUMA  GuardTerm
                    | GuardTerm RESTA GuardTerm
                    ;

sync_label          :
                    | SYNC NAME
                    ;

convex_predicate    : linear_constraint AMP convex_predicate
                    | linear_constraint
                    ;

linear_constraint   : linear_expression relop linear_expression
                    | TRUE
                    | FALSE
```

```
                    | ASAP
                    ;

relop               : MENOR
                    | MENORIGUAL
                    | IGUAL
                    | MAYORIGUAL
                    | MAYOR
                    ;

linear_expression   : linear_term
                    | linear_expression SUMA linear_expression
                    | linear_expression RESTA linear_expression
                    ;

linear_term         : rational
                    | NAME
                    ;

rational            : integer
                    | integer DIV NAT
                    ;

integer             : NAT
                    ;
```

Figura A-12. Subgramática HyTech del lenguaje de especificación de sistemas aceptada por el compilador de grafos temporizados.

Especificación de propiedades

Se traducirán a M sólo las tiras de la gramática de la figura A-13.

```
Commands            : region_declaration statements
                    ;

region_declaration  :
                    | VAR region_declaration_list DOSPTOS REGION
                      PTOYCOMA
                    ;

region_declaration_list: NAME rest_of_decl_list
                    ;

 rest_of_decl_list  :
```

```
                    | COMA region_declaration_list
                    ;


statements          :
                    | statement PTOYCOMA statements
                    ;

statement           : NAME ASIG re
                    ;

re                  : IPAR re DPAR
                    |       COMPL           re
                    | re     AMP            re
                    | re     UNION          re
                    | re     SUMA           re
                    | re     RESTA          re
                    | re     IGUAL          re
                    | re     MENOR          re
                    | re     MAYOR          re
                    | re     MAYORIGUAL     re
                    | re     MENORIGUAL     re
                    | REACH FORWARD         FROM re ENDREACH
                    | REACH BACKWARD        FROM re ENDREACH
                    | NAME
                    | NAT
                    | TRUE
                    | FALSE
                    | LOC ICORCH NAME DCORCH IGUAL NAME
                    ;
```

Figura A-13. Subgramática HyTech del lenguaje de especificación de propiedades aceptada por el compilador de grafos temporizados.

Las únicas sentencias válidas serán las asignaciones a variables de región.
La idea de las traducciones $H \rightarrow M$ es asociar una región a cada fórmula TCTL. Dicha región (que será la última en referenciarse en el fuente H) podrá haber sido definida en función de otras.
Las regiones auxiliares -que sirven para construir la región que define la propiedad- podrán darse explícitamente en la región final, o podrán almacenarse en variables de región, y más tarde ser referenciadas. Dichas variables de región serán mantenidas en una tabla de símbolos; por más detalles de implementación ver "Compiladores de Grafos Temporizados - Manual del Sistema ".

La siguiente es la lista de construcciones válidas:

var_region := state_predicate;

Asigna una región (definida por un *state_predicate*) a una variable de región. No expresa una propiedad, sirve de construcción intermedia.

var_region := var_SP & state_predicate;

Asigna una región (definida por la intersección de otras dos, una definida por el nombre de otra –de tipo *state_predicate*- y otra definida explícitamente por un 'state_predicate'.) a una variable de región. No expresa una propiedad, sirve de construcción intermedia.

var_region := reach forward from state_predicate endreach;

Asigna una región (definida como el conjunto de los estados alcanzables desde una región inicial definida a través de un SP) a una variable de región. No expresa una propiedad, sirve de construcción intermedia.

var_region := iterate var_reg FROM state_predicate USING { var_reg := POST(var_reg)}

Otra forma de expresar un reach forward.

var_region := reach backward from state_predicate endreach;

Asigna una región (definida como el conjunto de los estados desde los cuales se puede alcanzar una región definida a través de un SP) a una variable de región. No expresa una propiedad, sirve de construcción intermedia.

var_region := [reach_forw | var_RF | iterate | var_I] [< | <= | =] [reach_back | var_RB];

Asigna una región a una variable de región; define una propiedad del tipo $initial \Rightarrow AG(final)$, es decir, permite representar invariantes del sistema.

var_region := [reach_back | var_RF] [> | >= | =] [reach_for | var_RB | iterate | var_I];

Análoga a la anterior.

var_region := [reach backward from safety_SP endreach | var_RB] & inicial_SP;

El reach backward puede estar dado explícitamente o por una variable. Lo mismo ocurre con inicial_SP, que debe ser un state_predicate.
Representa una propiedad del tipo $initial \Rightarrow EF(safety)$, es decir, una propiedad de tipo safety.

var_region := inicial_SP & [reach backward from safety_SP endreach | var_RB];

Idem a la anterior.

var_region := [reach forward from inicial_SP endreach | var_RB | iterate | var_I] & safety_SP;

Idem a la anterior.

var_region := safety_SP & [reach forward from inicial_SP endreach | var_RB | iterate | var_I];

Idem a la anterior.

Las propiedades de tipo bounded response son representadas en HyTech por procesos monitores [Hen+96]; resulta al menos muy complejo reconocer este tipo de construcciones, por lo que quedarán pendientes para mejoras futuras al producto. Una forma de encarar el problema consiste en definir restricciones sintácticas que permitan reconocer los procesos monitores sintácticamente: una construcción determinada podría estar representando solamente un proceso monitor. De todas formas sería necesario evaluar la viabilidad y utilidad de este enfoque.

Las propiedades de tipo minimal event separation son análogas a la anterior [Hen+96].

Anexo B: Sublenguajes descartados

B.1. Kronos

ab{~k}(p)	no tiene sentido implementarla en U.
ed{~k}(p)	no tiene sentido implementarla en U.
eb{~k}(p)	no tiene sentido implementarla en U.
ad{~k}(p)	no tiene sentido implementarla en U.
(p1)eu(p2)	no tiene sentido implementarla en U.
(p1)eu{~k}(p2)	no tiene sentido implementarla en U.
(p1)au(p2)	no tiene sentido implementarla en U.
(p1)au{~k}(p2)	no tiene sentido implementarla en U.

No existen otras restricciones sobre el sistema ni sobre las fórmulas TCTL.

B.2. Uppaal

A excepción de las primitivas que expresan transiciones urgentes, todas las construcciones de Uppaal serán traducidas.

B.3. HyTech

pre (regExp)	no es posible traducir
post(regExp)	no es posible traducir
hull(regExp)	no es posible traducir
reach forward from regExp endreach	no se pueden traducir “referencias al pasado”
print omit locList locations regExp print regExp	despliega por pantalla, no contribuye a generar una propiedad a verificar
print omit locList	idem
locations regExp	idem
prints cadena	idem
printsize	idem

while	se traduce solamente una versión recortada de esta construcción
print trace to regName using regExp	no se traduce
weakdiff(regExp1, regExp2)	no interesa traducir
free(regName)	no tiene sentido traducir
iterate regName from regExp using commands	sólo una forma reducida será traducida (para expresar alcanzabilidad)
templates	en esta versión no se traducirán
arrays	en esta versión no se traducirán
rangos	en esta versión no se traducirán
análisis paramétrico	no se puede traducir
transiciones urgentes	serán descartadas
variables analog	no se traducen
variables integrator	usada por los AH, no se traducen
variables parameter	no se traducen
variables stopwatch	no se traducen (en tiempo de compilación no se pude saber si 'int' o 'clock')
constantes	en esta versión no se traducirán

Anexo C: Algunos detalles sobre las traducciones

Esta sección incluye algunos detalles de las traducciones que por su extensión no tenía sentido incluir en la sección 4.5. NO es una descripción exhaustiva de las traducciones.

Scope de variables y relojes

M soporta variables y relojes y locales y globales. Dadas las particularidades de cada una de las herramientas estudiadas, las traducciones seguirán el siguiente esquema:

$K \rightarrow M$ Kronos no implementa la distinción entre scopes de relojes locales y globales; además, variables de igual nombre en distintos archivos '.tg' son la misma. Por lo tanto la traducción declarará en M todas las variables (sin duplicarlas) de todos los autómatas Kronos , y dichas declaraciones serán globales.

- $H \rightarrow M$ En forma similar al caso anterior, como HyTech maneja la idea de scope global solamente, todos los relojes y variables serán serán definidos en M, y las declaraciones serán globales.

- $U \leftrightarrow M$ Como ambos soportan variables locales y globales, variables locales serán traducidas a variables locales, y variables globales, a globales.

- $K \leftarrow M$ $H \leftarrow M$ Tanto Uppaal como Kronos no soportan variables locales. Las variables locales de M se traducirán agregándoseles el nombre del autómata que las declara como prefijo.
  Las variables globales serán traducidas a variables globales.
  En Kronos todas las variables serán definidas en todos los archivos '.tg'.

Sincronización

Al levantar las restricciones de sincronización trivial aparece un conjunto de problemas a resolver cuya complejidad hace que sean dejados para futuras mejoras de la herramienta. A continuación se describen algunos de ellos.

Supongamos en Uppaal el caso en que el autómata A ejecuta la etiqueta de sincronización 'f!' y los autómatas B y C la etiqueta 'f?', como se muestra en la figura B-1.

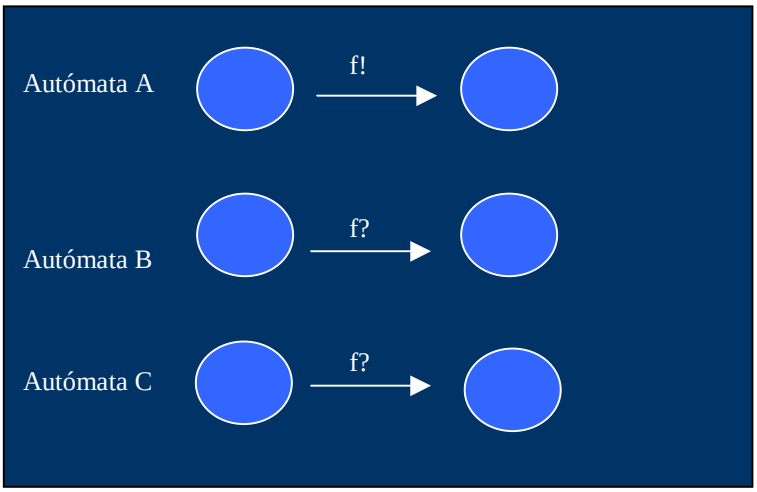


Figura B-1. Sincronización doble en Uppaal.

Dada la semántica de la sincronización en Uppaal, podrá ocurrir que sincronicen A y B o A y C. En ambos casos o bien B o bien C no sincronizarán, bloqueándose a la espera de una nueva transición 'f!'.
Una posible traducción de esta situación (a Kronos o Uppaal) consiste en traducir la arista etiquetada con 'f!' como dos aristas, una con la etiqueta de sincronización 'fAB' y otra con 'fAC'. La traducción de B contendrá una transición con la etiqueta 'fAB', y la de C una con 'fAC', como en la figura B-2.

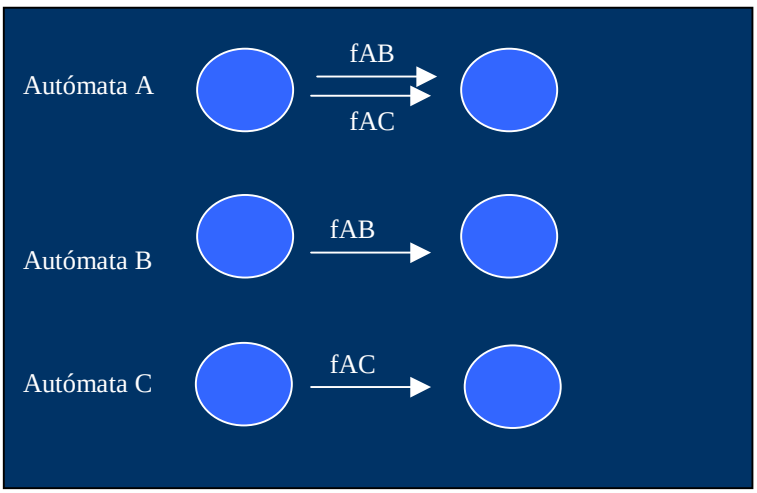


Figura B-2. Traducción de la sincronización doble.

Este método hace los grafos menos legibles. Y recordemos que este razonamiento se aplicaría a **cada** etiqueta de sincronización para la cual exista un emisor y múltiples receptores.

Consideremos ahora el caso de una sincronización triple en HyTech o Kronos. A, B y C se sincronizan según la etiqueta 'f'.
Una forma de simular esta situación en Uppaal (el hecho de que no estamos sincronizando pares de autómatas sino más de dos de ellos) consiste en recurrir a las *commited locations*.
En este enfoque se deberá generar un autómata adicional S, que regulará la sincronización. A se sincronizará según una nueva etiqueta 'f1!', B según 'f2!', C según 'f3!'; por otra parte S estará formado por cuatro nodos y tres aristas. Las aristas estarán etiquetadas por 'f1?', 'f2?' y 'f3?' (en cualquier orden) y los nodos que las unen serán *commited locations*, es decir, el sistema permanecerá tiempo cero en dichos estados.

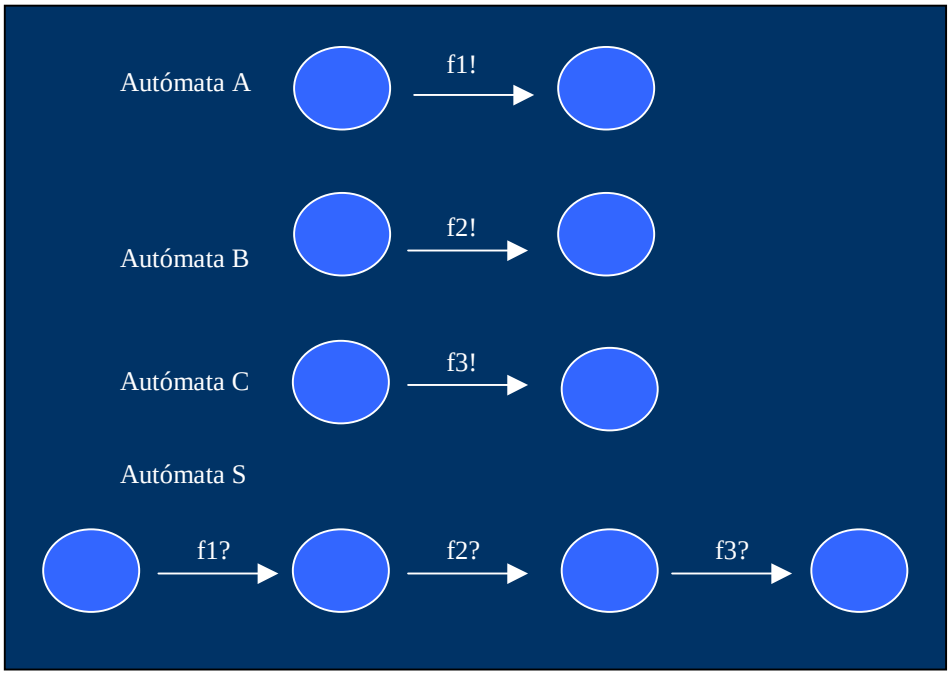


Figura B-3. Traducción de la sincronización triple.

Una desventaja obvia de este método es que por cada sincronización *N*-aria, con *N>2*, (para traducciones $K \rightarrow U$ y $H \rightarrow U$) se deberá crear un nuevo autómata. Y la traducción inversa -al menos no trivialemente- conservará el/los nuevo/s autómatas; la interpretación y manipulación del nuevo sistema es más complicada. Ambos enfoques preservan la semántica de la sincronización al precio de la pérdida de claridad.