Un lenguaje visual para la especificación y verificación automática de requerimientos de tiempo real complejos

Alejandra Alfonso
Directores: Dr. Alfredo Olivero, Dr. Victor Braberman

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

5 de mayo de 2003

Resumen

La construcción de sistemas de tiempo real libres de fallas es un objetivo perseguido por muchas actividades industriales debido al alto costo monetario y en vidas humanas que pueden provocar los desperfectos en los sistemas que éstas utilizan. Si bien existen numerosas herramientas para modelización de sistemas y verificación automática de propiedades sobre los mismos, la transferencia al ámbito industrial es lenta. Una de las razones fundamentales para este fenómeno es la poca *usabilidad* de los lenguajes de especificación existentes, basados en formalismos lógicos difíciles de entender y de usar por diseñadores no familiarizados con ese tipo de herramientas. Nuestra propuesta consiste en brindar una notación gráfica y de alto nivel para especificar requerimientos de sistemas concurrentes y de tiempo real, de forma tal de abstraer a los diseñadores y desarrolladores de los formalismos matemáticos subyacentes y aun así brindarles el poder de las herramientas de verificación formal automática.

Este trabajo sienta las bases para un lenguaje gráfico de especificación de propiedades basado en *patrones de eventos*, que permiten describir de forma simple y abstracta ciertos aspectos de las ejecuciones de un sistema. Por ejemplo, el orden en que deben ocurrir los eventos, la separación en el tiempo entre dos eventos, el conjunto de eventos que no deben ocurrir entre dos puntos de la ejecución, etc. El lenguaje de nuestros patrones está basado en órdenes parciales de eventos y su semántica está inspirada en el concepto de *pattern matching*.

patrones de mal comportamiento, es decir, en la descripción de comportamientos que invalidarían requerimientos críticos del sistema. Demostramos que contando con un modelo del sistema expresado con autómatas temporizados, el problema de "model-checking" de patrones de mal comportamiento puede ser reducido al problema clásico de verificación de autómatas temporizados. Dado que este último problema se sabe decidible, concluímos que el chequeo de patrones de mal comportamiento también lo es. Por otro lado, utilizamos esta reducción para construir un algoritmo de verificación automática de patrones de eventos basada en las herramientas existentes para model-checking de autómatas temporizados. Este algoritmo constituye el corazón de un prototipo de verificador de patrones de eventos que desarrollamos en Java.

Agradecimientos

Quiero agradecer a la Universidad y especialmente a la gente que mediante su trabajo hizo posible que nos formaramos varias generaciones de licenciados. Espero poder realizar mi aporte con mi trabajo como docente.

A mis directores, Victor y Alfredo, por todo el apoyo y la paciencia que tuvieron a lo largo de los muchos meses que duró el desarrollo de este trabajo.

Al jurado por haber aceptado el compromiso de revisar y corregir este trabajo.

A Daniel, mi amor y compañero de vida, por su ayuda tan valiosa durante todos estos meses y por todo su aguante especialmente en las últimas semanas.

A Sergio, Rula, Agustín, Hernán, Analía, Andran, Nacho, Manu y Tobe, gente genial con la cual tuve la suerte de compartir toda la carrera y a quienes en gran parte debo el haber llegado hasta este punto.

A mi familia por la confianza que tenían en que iba a poder terminar este ciclo.

A Nicolás, Diego, a todos mis compañeros de trabajo y a la cátedra de Paradigmas que me bancaron y apoyaron durante este último tiempo.

Gracias... *totales!*

Índice General

Capítulo 1

Introducción

El objetivo de las técnicas de *verificación* es mostrar que un determinado sistema de software o de hardware se adecúa a su especificación, es decir, que no puede comportarse en forma contraria a su especificación conduciendo a situaciones inesperadas, indeseadas o riesgosas.

Actualmente las técnicas más difundidas de verificación se basan en simulación y testing. Sin embargo, ambos enfoques sólo pueden analizar un conjunto relativamente pequeño de comportamientos de un sistema y tienden a ser inadecuados cuando el número de estados posibles del sistema es muy grande. Muchos sistemas son por naturaleza concurrentes y el comportamiento no determinístico introducido por la concurrencia puede conducir en distintos momentos a distintos comportamientos frente al mismo estímulo. En estos casos es especialmente inadecuado el uso de testing como método de verificación. En el caso de los sistemas de tiempo real, la correctitud del sistema no depende sólo del resultado lógico del cómputo sino también del cumplimiento de ciertos requerimientos temporales. Por ejemplo, en el desarrollo de aplicaciones críticas como control de tráfico aéreo o monitoreo de pacientes, es importante verificar que los sistemas realicen los cálculos adecuados, en el momento adecuado.

Para solucionar las falencias de las técnicas "empíricas" (basadas en simulación y testing), los métodos de verificación *formal* intentan "demostrar" formalmente que los comportamientos indeseados son imposibles en el sistema. Una de las principales características de las técnicas formales de verificación es que trabajan sobre un *modelo* del sistema, posiblemente abstrayendo los detalles no relevantes desde el punto de vista de las propiedades a verificar. Esto hace posible, por ejemplo, que se verifique el diseño de un circuito antes de que éste sea construido (es decir, en una etapa temprana de desarrollo), permitiendo ahorrar el enorme costo que implica descubrir un fallo lógico en el circuito una vez que está construido. A pesar de que los métodos formales de verificación ofrecen a los desarrolladores y diseñadores de sistemas numerosas ventajas con respecto a los métodos empíricos, todavía no han sido adoptados en forma masiva fuera del ámbito académico[1].

1.1 Métodos formales y verificación algorítmica

Existen básicamente dos enfoques distintos en cuanto a verificación formal. Uno está basado en métodos *deductivos* mientras que el otro está basado en métodos *algorítmicos*. Por razones históricas, a los métodos deductivos se los conoce habitualmente con el nombre de "*theorem-proving*" mientras que a los métodos del segundo grupo se los denomina en forma genérica "*model-checking*".

El problema de la verificación formal puede ser expresado de la siguiente manera: dado un sistema S, un requerimiento R y una relación de satisfacción $\models$, ¿vale que $S \models R$? Lo que diferencia a las distintas técnicas entre sí es la forma en que interpretan S, R y $\models$.

[1] Sin embargo, existen varias experiencias exitosas en la aplicación de técnicas formales a la industria del hardware y del software. Ver por ejemplo los casos mencionados en [CK96, CWA+96]

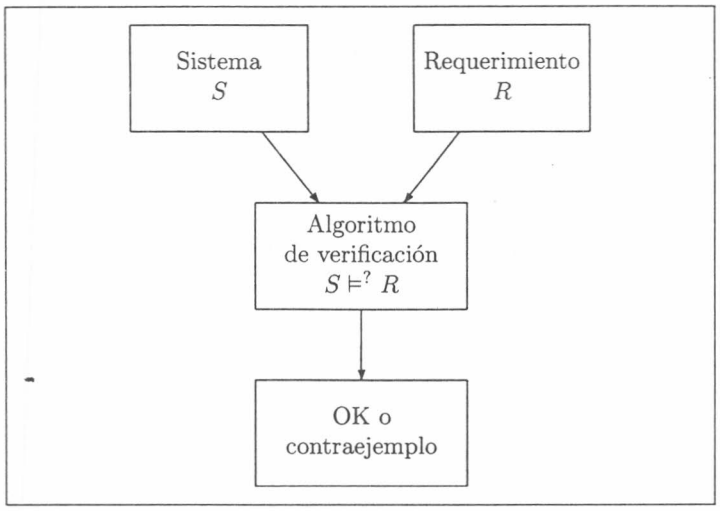


Figura 1.1: Esquema general del problema de verificación algorítmica

1.1.1 Métodos deductivos o *Theorem proving*

Theorem proving caracteriza al conjunto de técnicas donde tanto S como R están expresados cómo fórmulas de una cierta lógica. Esta lógica está dada por una teoría que define un conjunto de axiomas y de reglas de inferencia. *Theorem proving* consiste en demostrar formalmente que R se desprende como teorema de S en dicha teoría. Las principales desventajas de estas técnicas son que hasta el momento no existen herramientas totalmente automatizadas de verificación y que para cada sistema a analizar es necesario desarrollar una nueva teoría lógica.

1.1.2 Verificación algorítmica o *Model-checking*

Las técnicas que más interés despertaron entre los investigadores durante las dos últimas décadas son las pertenecientes al segundo grupo, es decir, las técnicas basadas en "model-checking" o verificación algorítmica. La principal diferencia con el grupo anterior es que, como su nombre lo indica, éstas técnicas proveen un algoritmo para decidir en forma automatizada el problema $S \models R$. Más aún, la mayoría de estas técnicas devuelven, en el caso en que el sistema[2] no satisfaga el requerimiento, un "contraejemplo" (generalmente la descripción de un comportamiento o conjunto de comportamientos) que muestra por qué el sistema no verifica la propiedad. La Figura 1.1 muestra en forma esquemática el problema de la verificación algorítmica.

Dentro de las técnicas denominadas "model-checking" existen numerosas clases de problemas de verificación y todas se distinguen por los formalismos que utilizan para expresar S y R y por el algoritmo que utilizan para decidir $S \models R$. Por ejemplo, el enfoque que dio el nombre a todo este grupo de técnicas [EC81] consiste en modelar al sistema S como una estructura de Kripke donde los nodos representaban los *estados* posibles del sistema y los ejes las *transiciones* o posibles cambios de estado. En ese contexto, R se expresa como una fórmula de la lógica temporal CTL y la relación de satisfacción se interpreta como "es modelo de" (Figura 1.2 (a)). Otras técnicas de verificación algorítmica se basan en la teoría de autómatas finitos. En estos enfoques una *ejecución* del sistema se modela como una secuencia (generalmente infinita) de *eventos* o de *estados* y el *lenguaje* del sistema $\mathcal{L}(S)$ está dado por el conjunto de ejecuciones del mismo. A su vez, los requerimientos son interpretados sobre ejecuciones individuales y el *lenguaje* del requerimiento $\mathcal{L}(R)$ es el conjunto de ejecuciones que lo satisfacen. Finalmente, $S \models R$ se traduce en $\mathcal{L}(S) \subseteq \mathcal{L}(R)$ (Figura 1.2 (b) y (c)).

[2]Cuando no se preste a confusión, no haremos distinción entre el "modelo de sistema" y el "sistema" en sí.

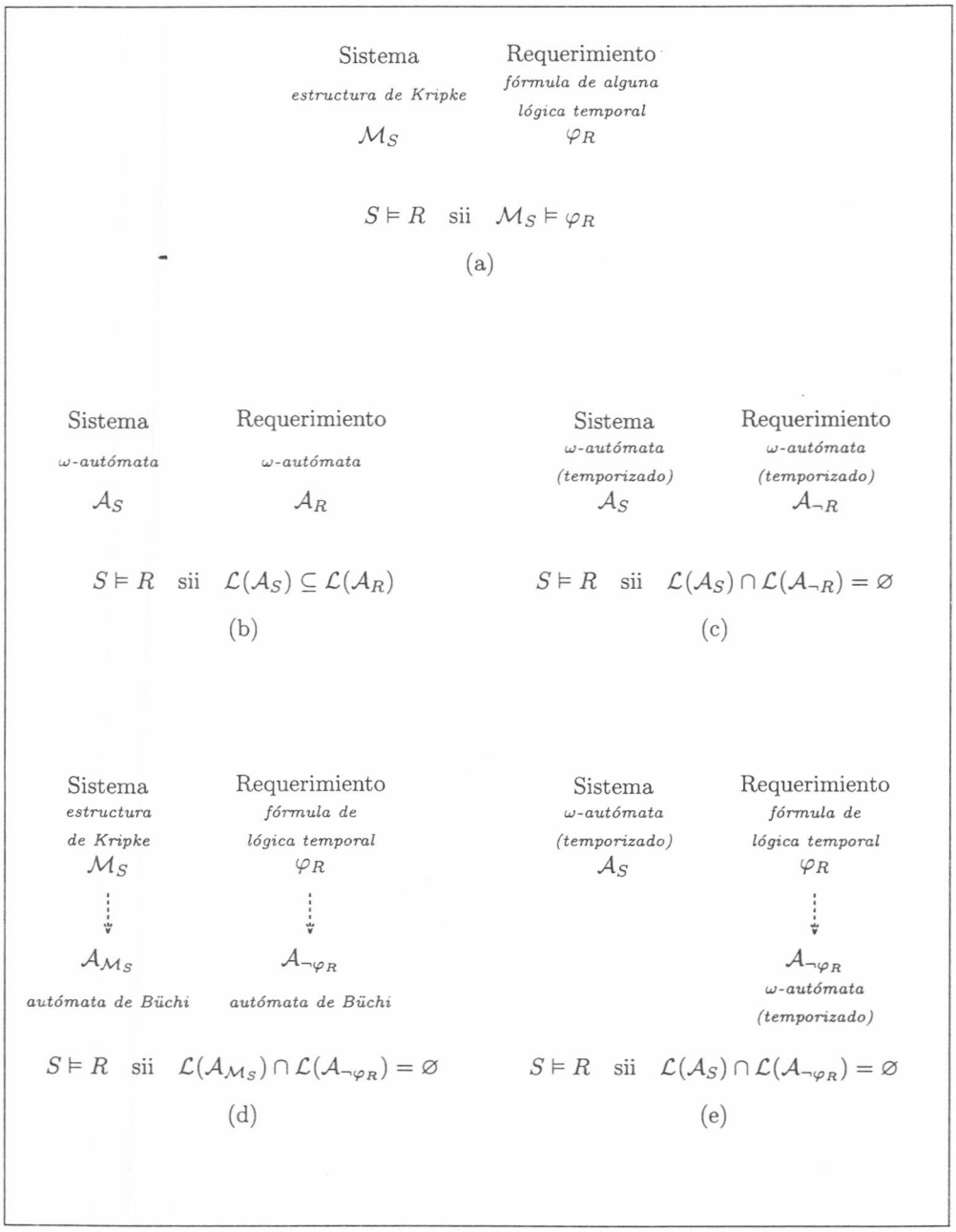


Figura 1.2: Model-checking, técnicas *automata-theoretic* y mixtas. (a) model-checking puro (ejemplo: [EC81]); (b),(c) técnicas automata-theoretic (ejemplo: [AD94]); (d),(e) técnicas mixtas (ejemplo: [VW86])

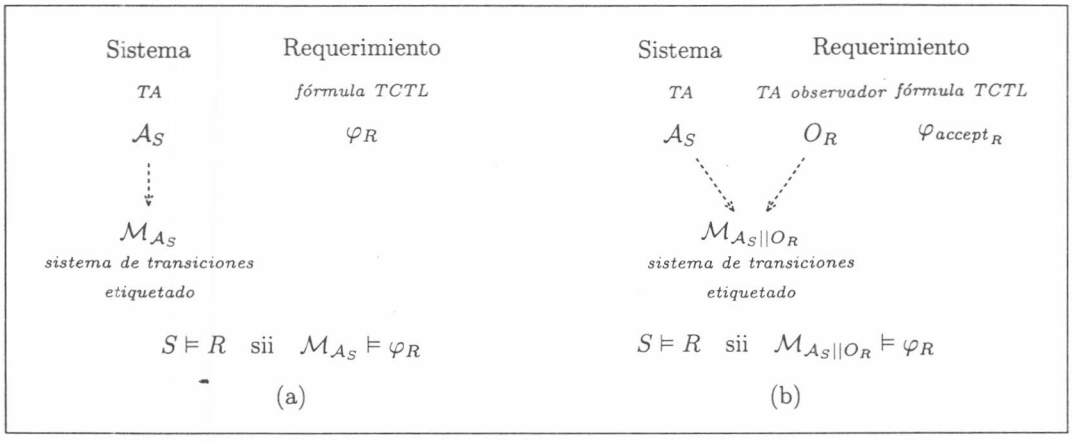


Figura 1.3: Verificación basada en autómatas temporizados. (a) enfoque de Kronos [DOTY95]; (b) enfoque usado en [Bra00]

En [VW86], Vardi y Wolper mostraron que estos dos enfoques pueden combinarse planteando el problema de model-checking de una fórmula LTL en función de un problema de inclusión de lenguajes (Figura 1.2 (d)). Los ω-autómatas son autómatas finitos con una condición de aceptación particular para reconocer palabras infinitas. Una de las condiciones de aceptación más usada es la de Büchi. Informalmente, la condición de aceptación de Büchi pide que se visiten infinitas veces ciertas locaciones distinguidas del autómata (en el próximo capítulo daremos una definición formal de la versión temporizada de los autómatas de Büchi). Otros enfoques parten de un sistema modelado con un ω-autómata y un requerimiento expresado en alguna lógica temporal (Figura 1.2 (e)) y reducen el problema a verificar que la intersección del lenguaje del sistema y del complemento del lenguaje del requerimiento sea vacía. En el caso de los ω-autómatas no temporizados, este problema es equivalente al de verificar que la inclusión de $\mathcal{L}(S)$ en $\mathcal{L}(R)$. Sin embargo, los ω-autómatas temporizados no son cerrados con respecto a la operación de complementación [AD94] y por lo tanto se debe proveer directamente un autómata que reconozca la negación del requerimiento.

Verificación basada en autómatas temporizados

La Figura 1.3 muestra dos variantes de la técnica de verificación basada en *autómatas temporizados* ([AD94, Yov96, DOTY95], entre otros). Existen numerosas herramientas que implementan la verificación basada en autómatas temporizados ([DOTY95, BLL+96, TC96, HHWT95], etc) y constituye la técnica más usada para modelización y verificación formal de sistemas concurrentes de tiempo real.

La verificación basada en autómatas temporizados permite modelar restricciones temporales explícitas: no sólo cuestiones temporales *cualitativas* como *liveness*, *fairness* y no determinismo sino aspectos *cuantitativos* como periodicidad, *bounded response*, *delays*, *deadlines*, etc. En este contexto, el problema básico de verificación se plantea en los siguientes términos: el sistema está modelado por un autómata temporizado[3] y los requerimientos se expresan en la lógica temporizada TCTL[4]. La semántica del autómata se expresa en función de un *Sistema de transiciones etiquetadas* (STE), que consiste en un grafo decorado donde los nodos son los estados del sistema (potencialmente no enumerables) y los ejes corresponden a la ocurrencia de un evento o al paso de una determinada cantidad de tiempo. Sobre este grafo se interpretan las fórmulas TCTL para decidir si el sistema cumple o no el requerimiento.

En la práctica, una variante posible es el uso de autómatas *observadores* para describir los requerimientos del sistema. En este contexto, se utilizan autómatas que capturen la negación del requerimiento (es decir,

[3] En general, resultado de la composición paralela de autómatas de menor tamaño llamados *componentes* o *módulos*. La composición paralela de autómatas temporizados se define formalmente en la sección 2.2.4.

[4] Sección 2.4 en el próximo capítulo

todos los comportamientos que *violan* el requerimiento) y una fórmula TCTL (en general, mucho más simple que las usadas en el enfoque clásico), que expresa la alcanzabiliad de un conjunto de estados considerados erróneos. El problema de verificación se traduce, entonces, en un problema de *alcanzabilidad* sobre el STE generado por la composición del modelo del sistema y el autómata observador. Esta variante, utilizada por ejemplo en [Bra00], constituye la base teórica para este trabajo.

1.1.3 Transferencia de tecnología hacia la industria

A pesar del entusiasmo que han despertado estas técnicas de verificación formal entre los investigadores durante las últimas dos décadas, ese entusiasmo no ha podido ser transmitido aún a la industria del desarrollo de sistemas.

Nuestra visión, compartida por otros grupos de investigación ([DKM+94, DAC98, AY99], etc), es que para que ocurra la transferencia de tecnología hacia la industria es necesario dar a los desarrolladores la posibilidad de escribir sus especificaciones de la forma más natural posible y brindarles herramientas de alto nivel para generar, analizar y verificar sus especificaciones. En particular, ninguna de las técnicas mencionadas arriba cumple en forma satisfactoria con el primer requisito. Todas se basan en formalismos matemáticos difíciles de escribir y entender, aun por personas muy entrenadas. Las lógicas temporales usadas por muchas de estas técnicas se tornan inadecuadas a medida que la complejidad de los sistemas crece. El siguiente ejemplo, tomado de [DAC98], ilustra esta situación. Dado el siguiente requerimiento para un ascensor:

"*Entre el momento en que el ascensor es llamado desde un piso hasta el momento en que el ascensor abre sus puertas en dicho piso, el ascensor puede pasar por ese piso como máximo dos veces*"

se escribe en LTL con la siguiente fórmula:

$$
\begin{array}{l}
\Box((call \wedge \Diamond open) \rightarrow \\
\quad ((\neg atfloor \wedge \neg open)\,\mathcal{U} \\
\quad\quad (open \vee ((atfloor \wedge \neg open)\,\mathcal{U} \\
\quad\quad\quad (open \vee ((\neg atfloor \wedge \neg open)\,\mathcal{U} \\
\quad\quad\quad\quad (open \vee ((atfloor \wedge \neg open)\,\mathcal{U} \\
\quad\quad\quad\quad\quad (open \vee (\neg atfloor\,\mathcal{U}\, open)))))))))
\end{array}
$$

Si bien los autómatas finitos permiten en general describir lenguajes en forma más o menos simple, las variedades más expresivas de autómatas (ω-autómatas (temporizados), autómatas temporizados) usadas para verificación formal, agregan poder expresivo a cambio de un incremento en la complejidad y dificultad de escritura. Por ejemplo, no es una tarea trivial garantizar que un autómata temporizado de complejidad media o grande esté correctamente temporizado (en el sentido de no permitir situaciones en las cuales el sistema se "bloquearía"). La complejidad y susceptibilidad a errores de estos formalismos los hacen particularmente inadecuados para ser utilizados por desarrolladores y diseñadores fuera del ámbito académico.

Por otro lado, en muchos de estos formalismos los requerimientos se expresan como una propiedad que deben cumplir *todas* las ejecuciones del sistema. Muchas veces, sin embargo, los requerimientos expresan una propiedad *safety* ("*nada malo pasa*") y, en estos casos, resulta más fácil y más natural expresar formalmente qué es lo que *no* se quiere que ocurra antes que hacerlo en forma indirecta describiendo todos los casos en los que no se produce la condición errónea.

Nuestra propuesta

Nuestra propuesta consiste en brindar una notación gráfica y de alto nivel para especificar requerimientos de sistemas concurrentes y de tiempo real, de forma tal de abstraer a los diseñadores y desarrolladores de los formalismos matemáticos subyacentes y aun así brindarles el poder de las herramientas de verificación formal. Nuestro trabajo sienta las bases para un lenguaje gráfico de especificación basado en *patrones de eventos*. Los patrones de eventos permiten describir de forma simple y abstracta ciertos aspectos de *una*

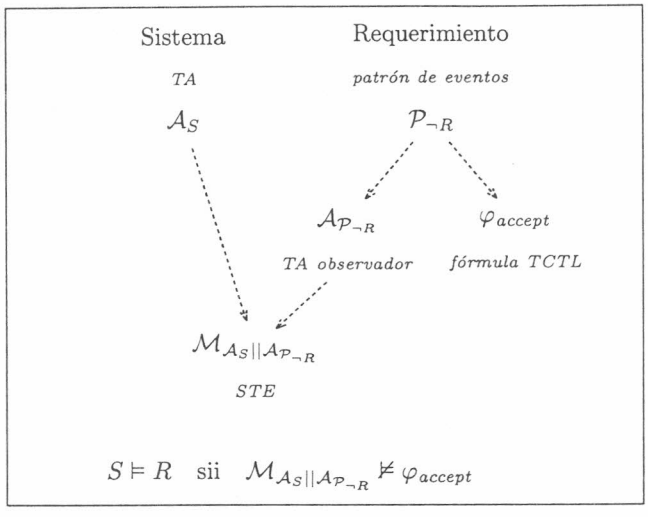
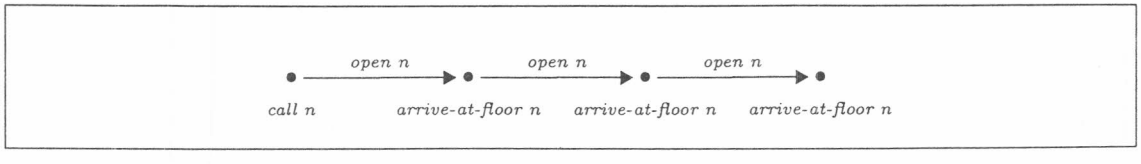


Figura 1.4: Enfoque basado en patrones de eventos

ejecución del sistema. Por ejemplo, el orden en que deben ocurrir los eventos, la separación en el tiempo entre dos eventos, el conjunto de eventos que no deben ocurrir entre dos puntos de la ejecución, etc.

Como veremos más adelante, los patrones de eventos podrían ser aplicados en varios contextos y ofrecen muchas posibilidades de extensiones futuras. En este trabajo aplicaremos los patrones de eventos a la descripción de *comportamientos no deseados* de un sistema y traduciremos el problema de "model-checking" de patrones de eventos al problema clásico de verificación basada en autómatas temporizados. Como muestra la Figura 1.4, usaremos los patrones de eventos como una forma abstracta de describir un comportamiento no deseado en el sistema (es decir, un caso en el que se violaría un requerimiento del sistema), y a partir de esos patrones construiremos, en forma automática y transparente para el diseñador, autómatas temporizados que capturen dichos comportamientos y que serán usados como observadores al estilo de la Figura 1.3 (b). Llegado este punto, tendremos a nuestra disposición todas las herramientas clásicas para verificación basada en autómatas temporizados.

Volviendo al ejemplo del ascensor mencionado anteriormente, parte de la complejidad de la fórmula residía en que describía todas las posibles combinaciones de sucesos en los cuales se cumplía el requerimiento. Sin embargo, alcanza con encontrar un caso en el cual se viole este requerimiento para concluir que el ascensor no cumple con su especificación. Quiere decir que si el ascensor admite *un* comportamiento en el cual el ascensor llegue (al menos) tres veces al piso donde lo llamaron antes de abrir sus puertas, podremos concluir que el ascensor no es correcto con respecto a su especificación. El Patrón 1.1 muestra a modo de ejemplo cómo se describiría este comportamiento no deseado para el ascensor usando patrones de eventos. El punto de más a la izquierda corresponde al llamado del ascensor desde el piso n y los tres puntos restantes corresponden a tres momentos distintos en los cuales el ascensor llegó a ese piso. Las flechas describen el orden en que ocurrieron los eventos y la decoración de las flechas indica el conjunto de eventos que *no* ocurrieron entre los



Patrón 1.1: Violación del requerimiento para el ascensor expresada como patrón de eventos

dos correspondientes a los extremos de la flecha. En este caso particular, se pide que en ningún momento se hayan abierto las puertas en el piso n.

1.2 Características de los formalismos de especificación

La mayoría de los lenguajes formales de especificación comparten ciertas características comunes. Sin embargo, existen varias diferencias en cuanto a la forma en que modelan ciertos elementos comunes a todos los sistemas, el poder expresivo, la facilidad para describir ciertos aspectos particulares de un sistema, etc. Dado que gran parte de este trabajo estará dedicado a la definición de un nuevo formalismo, debemos definir los elementos que utilizaremos para caracterizar y comparar este nuevo lenguaje con los existentes.

El primer elemento común a todos los formalismos de especificación es el concepto de *comportamiento* de un sistema. Debido a que la operatoria de un sistema puede variar en función de la entrada y/o del contexto en el cual está corriendo, el sistema se puede comportar de muchas maneras distintas. En el contexto de la verificación formal, a cada una de estas "formas" distintas en las que se puede comportar el sistema se la denomina un *comportamiento* del sistema. En general, el modelo construido a partir del sistema captura *todas* estas posibilidades, de forma tal que cualquier comportamiento observado en el sistema pueda ser representado como una ejecución del modelo. Según el propósito para el cual fue concebido cada formalismo, varía la forma en que se representa una ejecución. Las ejecuciones pueden estar representadas como secuencias, árboles, etc... finitos o infinitos; pueden hablar únicamente de los *estados* por los que pasa el sistema, de las transiciones[5] o de los *eventos* que ocurren en el sistema o de todos a la vez. Pueden hacer referencia explícita o no al momento en el tiempo en que ocurrió cada evento, etc.

El segundo aspecto está relacionado con el poder expresivo que tienen los distintos formalismos para especificar cuestiones relacionadas con la *precedencia* o relación de *causalidad* y con el *paralelismo* o *concurrencia* entre los eventos o estados de un sistema. Algunos formalismos fueron diseñados explícitamente para expresar este tipo de propiedades y dan soporte tanto a nivel sintáctico como semántico o sólo a nivel sintáctico. La mayoría de los formalismos pensados para modelar sistemas secuenciales no brindan soporte a ningún nivel para este tipo de propiedades.
Cuando el soporte es a nivel sintáctico, el formalismo incluye una notación para decir que dos eventos / estados pueden ocurrir en paralelo (o secuencialmente pero en cualquier orden) o que deben ocurrir en un orden determinado. Cuando el soporte es también a nivel semántico, las ejecuciones están representadas como órdenes parciales entre eventos / estados del sistema, donde el orden refleja las dependencias de causalidad entre ellos. A este tipo de semánticas se las conoce con el nombre de *partial-order semantics*. Sin embargo, las semánticas más utilizadas son las denominadas *interleaving semantics*, en las cuales las ejecuciones son palabras finitas o infinitas de eventos / estados. En estos modelos, un comportamiento de un sistema concurrente puede corresponderse con varias ejecuciones que representen todas las formas posibles de reordenar los eventos / estados que no tienen dependencias de causalidad entre sí.

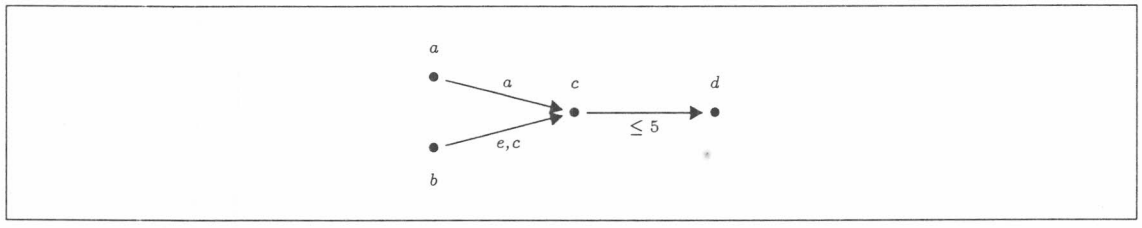
Otro elemento importante de los formalismos de especificación es la forma en que representan el tiempo. Los formalismos no temporizados no permiten hacer referencia al tiempo en forma explícita. En estos formalismos sólo se pueden expresar nociones de orden relativo entre eventos. Algunos formalismos utilizan un modelo de tiempo *discreto* en donde se asume que todos los eventos ocurren sincrónicamente cuando se produce un *tick* del reloj del sistema. Finalmente, el modelo de tiempo *denso* o *continuo*, utilizado por los autómatas temporizados, permite expresar restricciones temporales explícitas como *deadlines*, duración de una acción, máxima y mínima separación en el tiempo entre dos eventos, etc... lo cual lo hace especialmente adecuado para especificación de sistemas de tiempo real. Los autómatas temporizados incluyen la noción de *relojes* y de *restricciones temporales* sobre los estados (invariantes) y las transiciones del sistema (guardas). Las redes de Petri[6], las lógicas como LTL, CTL ([EC81]) y GIL ([DKM+94]) son ejemplos de formalismos no temporizados.

[5] En forma general, llamaremos *transiciones* a los cambios de estado de un sistema.
[6] En su versión básica, dado que existe una extensión temporizada de las mismas

1.3 Patrones de eventos

En este trabajo definimos y presentamos el formalismo basado en *patrones de eventos*. Este formalismo permite expresar en forma simple y abstracta las dependencias de causalidad entre los eventos del sistema y la ausencia de ellas. Además, permiten expresar restricciones temporales explícitas.
La semántica de los patrones de eventos estará dada por un modelo de *interleaving* de eventos junto con un modelo de tiempo *denso*. Los patrones serán interpretados sobre *ejecuciones temporizadas*. Una ejecución temporizada estará formada por una secuencia infinita de eventos, cada uno de ellos apareado con un real no negativo o *timestamp* que representa el momento de ocurrencia.

Básicamente, un patrón de eventos es un grafo dirigido y acíclico donde los nodos o *puntos* representan la ocurrencia de un evento en un sistema (puntos llenos) o instantes en el tiempo (puntos huecos) y los ejes representan dependencias de causalidad entre los puntos. Los puntos llenos deben estar *etiquetados* por uno o más símbolos de un alfabeto Σ que representan los eventos posibles en el sistema. Los puntos huecos no están etiquetados. La ausencia de restricciones de causalidad entre dos puntos permite modelar *no determinismo* (entre eventos de un mismo proceso) y *concurrencia* (entre eventos de procesos distintos).



Patrón 1.2: Ejemplo de patrón de eventos

Ejemplo. *Patrón de eventos.*
El Patrón 1.2 se refiere a *una* ocurrencia de cada uno de los eventos a, b, c y d en una ejecución. Cualesquiera sean esas ocurrencias, se debe cumplir que:

- la ocurrencia de a debe ser anterior a la de c (indicado por la flecha entre a y c)
- la ocurrencia de b debe ser anterior a la de c (indicado por la flecha entre b y c)
- la ocurrencia de c debe ser anterior a la de d (indicado por la flecha entre c y d)
- a y b pueden ocurrir en forma concurrente o bien en forma secuencial sin importar el orden (indicado por la ausencia de un camino de flechas entre a y b)
- entre la ocurrencia de a y la de c no puede haber otra a, quiere decir que la ocurrencia de a es la *última* antes de la de c (indicado por la 'a' sobre la flecha entre a y c)
- análogamente, entre la ocurrencia de b y c no debe haber otra c ni una e, quiere decir que la ocurrencia de c es la *próxima* después de la de b (indicado por la lista 'e, c' sobre la flecha entre b y c)
- finalmente, el tiempo transcurrido entre la ocurrencia de c y la de d no debe superar las 5 u.t.

Si existe al menos una forma de identificar las ocurrencias de los cuatro eventos sobre una ejecución verificando todas las restricciones enunciadas arriba, entonces la ejecución *satisface* el patrón. En otro caso, la ejecución no satisface el patrón.

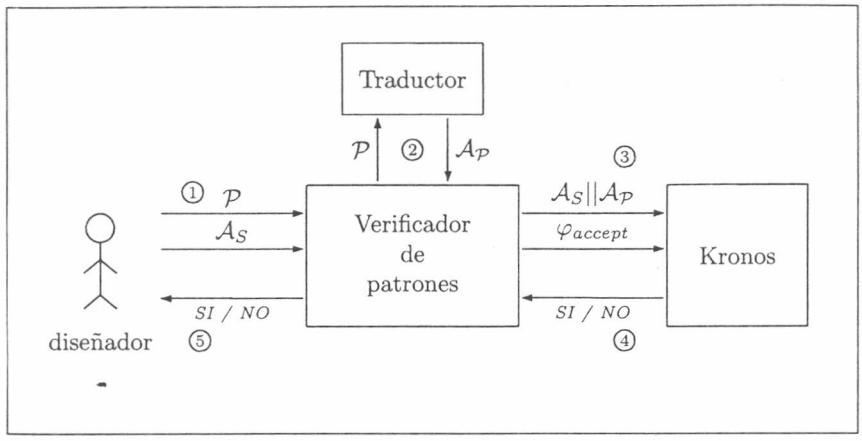


Figura 1.5: Verificación de patrones

1.3.1 Model-checking de patrones de *mal comportamiento*

Como mencionamos anteriormente, en este trabajo usaremos los patrones de eventos para expresar *comportamientos no deseados* de un sistema (es decir, la *negación* o el *complemento* de un requerimiento). En ese contexto, nos referiremos a los patrones de eventos como *patrones de mal comportamiento.*

En general, los requerimientos de un sistema representan una propiedad que deben cumplir *todas* las ejecuciones de dicho sistema. Quiere decir que alcanza con encontrar *una* ejecución que *no* cumpla esa propiedad para concluir que el sistema *viola* el requerimiento. Dado que usaremos los patrones de eventos para describir los casos en los que se viola un requerimiento, alcanza con que una sola de las ejecuciones del sistema satisfaga el patrón para concluir que el sistema es incorrecto (con respecto a ese requerimiento). Quiere decir que dado un sistema S y un patrón de mal comportamiento $\mathcal{P}$, diremos que $S \models \mathcal{P}$ si y sólo si existe una ejecución de S que satisface $\mathcal{P}$. Entonces, $S \models R$ si y sólo si $S \not\models \mathcal{P}_{\neg R}$. Podemos pensar al conjunto de ejecuciones que satisfacen $\mathcal{P}_{\neg R}$ como el *lenguaje* $\mathcal{L}(\mathcal{P}_{\neg R})$ de $\mathcal{P}_{\neg R}$. Más adelante veremos que el ω-autómata $\langle \mathcal{A}_{\mathcal{P}_{\neg R}}, \mathcal{F} \rangle$, donde $\mathcal{F}$ es una condición de aceptación de Büchi, reconoce el lenguaje $\mathcal{L}(\mathcal{P}_{\neg R})$. Veremos también que dada la morfología de los autómatas reconocedores, la condición de aceptación puede ser expresada como una fórmula TCTL φ_{accept} y en consecuencia $S \models \mathcal{P}$ si y sólo si $\mathcal{A}_S||\mathcal{A}_{\mathcal{P}_{\neg R}} \models \varphi_{accept}$. Dado que éste último problema puede ser decidido por herramientas como Kronos, el problema de la verificación de patrones resulta ser decidible.

Dado un sistema S, $\mathcal{A}_S$ será un modelo de S expresado como autómata temporizado. Dado un requerimiento R sobre S, el diseñador deberá construir un patrón de eventos $\mathcal{P}_{\neg R}$ que represente los comportamientos que violan R[7]. Este patrón y el modelo del sistema serán el input para el *Verificador de patrones.* La Figura 1.5 muestra esquemáticamente el proceso de verificación.

A partir del patrón $\mathcal{P}$, la herramienta genera un autómata temporizado $\mathcal{A}_\mathcal{P}$ que acepta exactamente las ejecuciones que satisfacen el patrón, usando para esto un *Traductor.* El autómata $\mathcal{A}_\mathcal{P}$ cumple el rol de *componente observadora* de $\mathcal{A}_S$. El verificador utiliza la herramienta Kronos para decidir si $\mathcal{A}_S||\mathcal{A}_\mathcal{P} \models \varphi_{accept}$, donde φ_{accept} es la codificación en TCTL de la condición de aceptación mencionada más arriba. Kronos responde *OK* si vale φ_{accept} en $\mathcal{A}_S||\mathcal{A}_\mathcal{P}$ o un contraejemplo en caso contrario. Finalmente, esta información es transmitida por el verificador al usuario.

[7]La idea es que estos patrones sean generados directa o indirectamente, utilizando para esto una herramienta gráfica.

1.4 Estructura del trabajo

El Capítulo 2 presenta los conceptos y definiciones básicas necesarios para el resto del trabajo. En ese capítulo se incluye la sintaxis y semántica de los autómatas temporizados y de los autómatas de Büchi temporizados. También se incluye la sintaxis y semántica de la lógica temporizada TCTL.

El Capítulo 3 define la sintaxis y la semántica de los patrones de eventos. La presentación se realiza en tres etapas: se introduce primero una versión básica de los patrones en la cual no se incluye la parte temporal, luego se presenta una versión temporizada de estos patrones y en una última etapa se extienden los patrones para soportar los conceptos de *principio* y *final* de una ejecución. Este esquema de presentación incremental tiene el objetivo de facilitar la introducción de los distintos conceptos así como también facilitar las demostraciones posteriores. De todas formas, debe quedar claro que excepto en el contexto de los capítulos 2 y 3, cuando nos refiramos a *patrones de eventos* nos estaremos refiriendo a la versión más expresiva de los mismos.

En el Capítulo 4 definimos el concepto de *autómata reconocedor* de un patrón de eventos y la construcción de dicho autómata sigue el esquema del capítulo 3: primero se realiza la construcción para los patrones básicos no temporizados, después se agrega la parte temporal y finalmente se agregan los conceptos de principio y final de ejecución. En ese mismo capítulo se demuestra que dado un patrón $\mathcal{P}$, su autómata reconocedor $\mathcal{A}_\mathcal{P}$ aumentado con una condición de aceptación de Büchi acepta exactamente las ejecuciones que satisfacen $\mathcal{P}$. Dada la morfología particular de los autómatas reconocedores, la condición de aceptación puede ser expresada como una fórmula TCTL y se demuestra que un sistema S satisface P si y sólo si $\mathcal{A}_S||\mathcal{A}_\mathcal{P} \vDash init \Rightarrow \exists\Diamond(\exists\Box accept)$ y por lo tanto el problema de la verificación de patrones es decidible. La fórmula $init \Rightarrow \exists\Diamond(\exists\Box accept)$, cuyo significado analizaremos más adelante, corresponde a φ_{accept} mencionada en la Figura 1.5.

En el Capítulo 5 mostramos varios ejemplos aplicados a casos de estudio presentados en la bibliografía.

En el Capítulo 6 se incluye el pseudocódigo de los algoritmos para la generación de autómatas reconocedores a partir de patrones.

El Capítulo 7 presenta las conclusiones y el trabajo futuro relacionado con este trabajo. En ese capítulo se enumeran también los trabajos relacionados con el nuestro, resaltando las principales similitudes y diferencias entre ellos.

Finalmente, en el Apéndice A se incluyen las demostraciones de todas las propiedades enunciadas a lo largo de la tesis y en el Apéndice B se comentan los detalles de la implementación del *Verificador de patrones* (Figura 1.5) realizada en Java y se incluyen varios listados relacionados con su aplicación a uno de los casos de estudio.

Capítulo 2

Definiciones preliminares

En este capítulo presentaremos los conceptos y definiciones básicas que utilizaremos a lo largo de todo el trabajo. Empezaremos por formalizar las nociones de *secuencias*, *ejecuciones* y *ejecuciones temporizadas* que ya introdujimos informalmente en el capítulo anterior.

A continuación, formalizaremos los conceptos básicos de la Teoría de Sistemas Temporizados, también mencionados en la Introducción: sintaxis y semántica de los autómatas temporizados, los autómatas de Büchi temporizados y de la lógica temporizada TCTL. En los próximos capítulos usaremos la teoría de autómatas temporizados con dos propósitos: por un lado, para demostrar desde el punto de vista teórico la *decidibilidad* de la verificación de patrones de mal comportamiento y por el otro, desde el punto de vista práctico, para construir un verificador de patrones basado en herramientas existentes para la verificación de autómatas.

2.1 Secuencias y Ejecuciones

Secuencias

Empezaremos por introducir la notación básica que utilizaremos para trabajar con secuencias.

Dado un conjunto C, una *secuencia sobre* C es una secuencia de elementos de C. Dada una secuencia s, $|s|$ será la longitud de la secuencia. Cuando s sea una secuencia infinita, diremos que $|s| \stackrel{def}{=} \infty$. Llamaremos $\Pi_s \stackrel{def}{=} \{i \in \mathbf{N} \mid 0 \leq i < |s|\}$ al subconjunto de los números naturales correspondientes a las posiciones de la secuencia s.

Para $i, j \in \Pi_s$, s_i denotará al i-ésimo elemento de la secuencia s, $s_{i]}$ será el prefijo de s que termina con el i-ésimo elemento inclusive, $s_{[i}$ será el sufijo de s que empieza desde el i-ésimo elemento inclusive y $s_{[i,j]}$ la subsecuencia que empieza en el i-ésimo elemento y termina en el j-ésimo inclusive (si $i > j$, $s_{[i,j]}$ será la secuencia vacía). En cualquier caso, usar '(' o ')' en lugar de '[' o ']' indicará que la subsecuencia no incluye el borde en cuestión. Llamaremos $prim(s)$ al primer elemento de s, es decir, $prim(s) \stackrel{def}{=} s_0$. Análogamente, si s fuera finita, llamaremos $ult(s)$ al último elemento de s, es decir, $ult(s) \stackrel{def}{=} s_{|s|-1}$.

La concatenación de dos secuencias s y s', donde la primera es finita, será denotada por ss'. Las secuencias con un único elemento serán identificadas por ese elemento.

Dado un conjunto E y una secuencia s sobre E, $elems(s)$ denotará al conjunto de los elementos que componen s. Para simplificar la notación en las definiciones de los próximos capítulos, sobrecargaremos el operador $\cap$ de forma tal que dado un subconjunto X de E, $s \cap X$ será equivalente a $elems(s) \cap X$.

Definición 2.1.1. Secuencias temporales.
Una *secuencia temporal* es una secuencia débilmente creciente de *timestamps*, es decir, de números reales no negativos.

Dada una secuencia temporal τ, definimos la función Δ que determina el *tiempo transcurrido* para esa secuencia como:

$$\Delta(\tau) = \begin{cases} 0 & \text{si } |\tau| \leq 1 \\ ult(\tau) - prim(\tau) & \text{si } 1 < |\tau| < \infty \\ lim_{i \longrightarrow \infty} \tau_i - prim(\tau) & \text{en otro caso} \end{cases}$$

Definimos *desplazamiento* de una secuencia temporal τ por un real ϵ (notado $\tau + \epsilon$) de forma tal que $\forall i \in \Pi_\tau,\ (\tau + \epsilon)_i = \tau_i + \epsilon$.

Definimos también la operación $\lhd$ entre secuencias temporales como sigue. Dada una secuencia temporal finita τ y otra secuencia temporal τ', $\tau \lhd \tau' \stackrel{def}{=} \tau(\tau' + ult(\tau))$. Por ejemplo: $(0\ 2\ 3\ 5.5) \lhd (1\ 5.3) = (0\ 2\ 3\ 5.5\ 6.5\ 10.8)$

Ejecuciones

Usaremos el concepto de *ejecución* para representar formalmente el comportamiento de un sistema. Como vimos en el capítulo anterior, usaremos un modelo orientado a *eventos*, con lo cual nuestras ejecuciones modelarán la sucesión de eventos ocurridos en una corrida del sistema. Utilizaremos dos tipos de ejecuciones: *temporizadas* y *no temporizadas*.

Definición 2.1.2. Ejecución (no temporizada).
Dado un conjunto de eventos Σ, una *ejecución (no temporizada)* sobre Σ es una secuencia finita o infinita sobre $\Sigma \cup \{\lambda\}$.

Ejemplo. Dado el conjunto de eventos $\Sigma = \{read,\ write\}$, los siguientes son ejemplos de ejecuciones:

read write read write read write read write ...
read read read read
read λ read λ λ read read λ λ λ λ ...

La primera muestra una ejecución infinita formada por infinitos *read*'s seguidos de su respectivos *write*'s. La segunda muestra un ejemplo de ejecución finita formada por únicamente cuatro *read*'s. Finalmente, la tercera muestra un ejemplo de ejecución donde aparece el símbolo distinguido λ. Usaremos ese símbolo para indicar explícitamente que no ha ocurrido ningún evento nuevo en el sistema. En ese sentido, la segunda y la tercera ejecuciones representan formas sintácticamente distintas de describir el *mismo* comportamiento del sistema. Definimos el concepto de *equivalencia* entre ejecuciones para capturar esta relación entre las distintas formas de representar un mismo comportamiento abstracto.

Definición 2.1.3. Equivalencia de ejecuciones.
Dado un conjunto de eventos Σ, dos ejecuciones ς y ς' sobre Σ son *equivalentes*, notado $\varsigma \equiv \varsigma'$, si al eliminar todas las apariciones de λ en ς y ς' las secuencias resultantes son iguales.

Es decir, $\varsigma \equiv \varsigma'$ si y sólo si tienen los mismos eventos, en el mismo orden.

Ejemplo. Como vimos antes, *read read read read* $\equiv$ *read λ read λ λ read read λ λ λ λ ...*

Debería resultar más o menos evidente que para toda ejecución ς finita, se puede construir una ejecución ς' *infinita* y equivalente a ς son solo agregar infinitos λ's al final de ς. Esto resultará importante más adelante cuando hablemos de *lenguaje* de un autómata temporizado.

Definición 2.1.4. Ejecución filtrada por un conjunto de eventos.
Dada una ejecución $\varsigma = a_0 a_1 \ldots a_i \ldots$ sobre Σ y un conjunto de eventos $\mathcal{S}$, definimos $\varsigma\lceil_{\mathcal{S}}$ como la secuencia que se obtiene de reemplazar en ς todas los eventos que no pertenezcan a $\mathcal{S}$ por λ. Formalmente:

$$\varsigma\lceil_{\mathcal{S}} = a'_0 a'_1 \ldots a'_i \ldots$$

donde:

$$a'_i = \begin{cases} a_i & \text{si } a_i \in \mathcal{S} \\ \lambda & \text{en otro caso} \end{cases}$$

Ejemplo. Dado el conjunto de eventos $\Sigma = \{a,\ b,\ c\}$ y el conjunto $\mathcal{S} = \{a,\ b\}$

$$(a\ b\ c\ a\ c\ c\ b)\lceil_{\mathcal{S}} \quad = \quad a\ b\ \lambda\ a\ \lambda\ \lambda\ b$$

En general, $\mathcal{S}$ será un subconjunto de Σ y usaremos la operación $\lceil_{\mathcal{S}}$ para abstraernos de ciertos eventos que no resulten importantes en un determinado contexto. Por ejemplo, en un sistema formado por varios componentes independientes, una ejecución del sistema incluirá eventos de *todos* los componentes. Sin embargo, si quisiéramos analizar el comportamiento de un componente individualmente (analizando para ello una ejecución del sistema), podríamos filtrar todos los eventos que no sean relevantes para ese componente (es decir, todos los eventos "internos" de los demás componentes).

Definición 2.1.5. Ejecuciones temporizadas.
Dado un conjunto de eventos Σ, una *ejecución temporizada* sobre Σ es un par $\sigma = \langle \varsigma, \tau \rangle$, donde ς es una secuencia finita o infinita sobre $\Sigma \cup \{\lambda\}$ y τ es una secuencia temporal de la misma longitud.

Extendemos las operaciones de secuencias a las ejecuciones temporizadas de la siguiente manera: dada una ejecución temporizada $\sigma = \langle \varsigma, \tau \rangle$, $|\sigma| \stackrel{def}{=} |\varsigma| = |\tau|$, las posiciones de σ serán las mismas que para ς y τ, para todo $i, j \in \Pi_\sigma$, $\sigma_i \stackrel{def}{=} (\varsigma_i, \tau_i)$, $\sigma_{i]} \stackrel{def}{=} \langle \varsigma_{i]}, \tau_{i]} \rangle$, $\sigma_{[i} \stackrel{def}{=} \langle \varsigma_{[i}, \tau_{[i} \rangle$ y $\sigma_{[i,j]} \stackrel{def}{=} \langle \varsigma_{[i,j]}, \tau_{[i,j]} \rangle$. En este caso también, usar '(' o ')' en lugar de '[' o ']' para delimitar subsecuencias indicará que no se incluye el borde en cuestión. Llamaremos $prim(\sigma)$ al par $(prim(\varsigma), prim(\tau))$ y, si σ fuera finita, $ult(\sigma)$ al par $(ult(\varsigma), ult(\tau))$. Finalmente, dada $\sigma' = \langle \varsigma', \tau' \rangle$ finita, la concatenación de σ con σ' se define como $\sigma\sigma' \stackrel{def}{=} \langle \varsigma\varsigma', \tau \triangleleft \tau' \rangle$.

Ejemplo. El siguiente es un ejemplo de ejecución temporizada:

$$\left\langle \begin{matrix} read & write & \lambda & read & \lambda & write & \\ 0.5 & 1.1 & 1.2 & 5 & 5 & 8.6 & \cdots \end{matrix} \right\rangle$$

donde el primer *read* ocurrió a las 0.5 u.t. desde que comenzó la corrida, el primer *write* ocurrió a las 1.1 u.t., etc.. Como en el caso de las ejecuciones no temporizadas, los λ indican que no ocurrió ningún evento. Dado que la secuencia temporal que acompaña a la secuencia de eventos debe ser creciente aunque en forma no estricta, los λ's pueden estar acompañados de un timestamp igual al del evento anterior, igual al del evento siguiente o con un valor intermedio (modelo de tiempo *denso*). En este caso como en el anterior, definimos la noción de *equivalencia* entre ejecuciones temporizadas que representan el mismo comportamiento del sistema.

Definición 2.1.6. Equivalencia de ejecuciones temporizadas.
Dado un conjunto de eventos Σ, dos ejecuciones temporizadas $\sigma = \langle \varsigma, \tau \rangle$ y $\sigma' = \langle \varsigma', \tau' \rangle$ sobre Σ son *equivalentes*, notado $\sigma \equiv \sigma'$, si y sólo si $\varsigma \equiv \varsigma'$ y todos los eventos tienen el mismo timestamp asociado en σ y en σ'.

Es decir, $\sigma \equiv \sigma'$ si y sólo si tienen los mismos eventos, en el mismo orden y cada evento tiene el mismo timestamp asociado en las dos ejecuciones.

Ejemplo.

$$\left\langle \begin{matrix} a & \lambda & b & c & d \\ 1 & 1.5 & 2.6 & 3 & 4.8 \end{matrix} \right\rangle \equiv \left\langle \begin{matrix} a & b & c & d \\ 1 & 2.6 & 3 & 4.8 \end{matrix} \right\rangle \equiv \left\langle \begin{matrix} \lambda & \lambda & a & b & \lambda & c & d & \lambda & \lambda & \lambda & \cdots \\ 0.5 & 0.8 & 1 & 2.6 & 3 & 3 & 4.8 & 5 & 7.5 & 8.1 & \end{matrix} \right\rangle$$

En este caso, como en caso de las ejecuciones no temporizadas, siempre será posible construir una ejecución temporizada infinita equivalente a una finita dada. La diferencia es que en este caso existen infinitas formas de *extender* la ejecución original: tantas como sucesiones débilmente crecientes de números reales.
En general, nos interesarán las ejecuciones infinitas dado que permiten representar el comportamiento de sistemas *reactivos* cuyas corridas podrían no terminar nunca (por ejemplo: sistemas operativos, microprocesadores, etc). En los casos en que un sistema admita corridas finitas, usaremos el mecanismo explicado en el párrafo anterior para construir una representación infinita de la misma corrida (conceptualmente estaríamos modelando el hecho de que el sistema deja pasar el tiempo permaneciendo en el estado final alcanzado después de la corrida).
A su vez, dentro de las ejecuciones infinitas, preferiremos las ejecuciones *divergentes*, es decir, aquellas en las cuales la secuencia temporal es divergente. Esto nos permite modelar el *progreso* del tiempo más allá de cualquier constante real. Por otro lado, la decisión de usar secuencias temporales débilmente crecientes nos permite modelar la ocurrencia *simultánea* de eventos.

Definición 2.1.7. Ejecución divergente.
Diremos que una ejecución infinita $\sigma = \langle \varsigma, \tau \rangle$ es *divergente* sii:

$$\lim_{i \to \infty} \tau_i = \infty$$

Los filtros sobre ejecuciones temporizadas son una extensión natural de los filtros sobre ejecuciones no temporizadas:

Definición 2.1.8. Ejecución temporizada filtrada por un conjunto de eventos.
Dada una ejecución temporizada $\sigma = \langle \varsigma, \tau \rangle$ y un conjunto de eventos $\mathcal{S}$, definimos $\sigma \lceil_{\mathcal{S}} \stackrel{def}{=} \langle \varsigma \lceil_{\mathcal{S}}, \tau \rangle$.

Ejemplo. Dado $\mathcal{S} = \{a,\ b,\ e\}$:

$$\left\langle \begin{matrix} a & \lambda & b & c & d \\ 1 & 1.5 & 2.6 & 3 & 4.8 \end{matrix} \right\rangle \lceil_{\mathcal{S}} = \left\langle \begin{matrix} a & \lambda & b & \lambda & \lambda \\ 1 & 1.5 & 2.6 & 3 & 4.8 \end{matrix} \right\rangle$$

2.2 Autómatas temporizados

Los autómatas temporizados constituyen uno de los formalismos más usados para modelización y verificación de sistemas concurrentes de tiempo real. La verificación basada en autómatas temporizados está soportada por varias herramientas ([DOTY95, BLL+96, TC96, HHWT95], etc) que han sido aplicadas con éxito al chequeo de protocolos de comunicación y circuitos, y han sido usadas por varios grupos de investigación, tanto en el ámbito académico como en la industria.

Los autómatas temporizados son básicamente autómatas finitos en los cuales el *tiempo* se incorpora mediante el uso de *relojes*. Como los autómatas finitos, los autómatas temporizados están compuestos por un conjunto finito de nodos (llamados *locaciones* en la bibliografía) y un conjunto de aristas etiquetadas. Debido a que se asume que las corridas del sistema son infinitas, no existe el concepto de *estado final*. Las aristas modelan la ocurrencia de eventos. Todos los relojes avanzan al mismo ritmo y tienen un comportamiento similar al de un cronómetro: miden el tiempo transcurrido desde que fueron iniciados (o reiniciados). La ocurrencia de un evento puede provocar que ciertos relojes se reinicien. Esto se modela asociando a cada arista el conjunto

de relojes que se reinician. Cada arista tiene asociada, además, una *guarda* o condición de habilitación. Dicha guarda impone restricciones sobre los valores que deben tener los relojes al momento de producirse el evento. Las aristas se atraviesan en forma instantánea y el tiempo transcurre en las locaciones del autómata. Además, cada locación tiene asociada una restricción temporal o *invariante* que determina las combinaciones de valores de los relojes que son válidas en esa locación. Estos invariantes pueden ser usados para indicar, por ejemplo, que el control del sistema no puede permanecer más de cierta cantidad de tiempo en una locación (*deadline*).

2.2.1 Definiciones preliminares

- $X = \{x_1, x_2, \ldots, x_n\}$ es un conjunto de *relojes* (variables reales no negativas)
- Dado un conjunto de relojes X, una *valuación* es una función total $v : X \xrightarrow{tot} \Re^+$ donde $v(x_i)$ es el valor asociado al reloj x_i.

  Llamaremos $\mathcal{V}_X$ al conjunto de todas las valuaciones sobre X. Es decir, $\mathcal{V}_X$ es el conjunto de todas las funciones totales $[X \xrightarrow{tot} \Re^+]$.

  Sea $\rho \subseteq X$ y v una valuación sobre los relojes de X. Definimos $Reset_\rho(v)$ como:

$$Reset_\rho(v)(x) = \begin{cases} 0 & \text{si } x \in \rho, \\ v(x) & \text{en caso contrario.} \end{cases}$$

- Dado un conjunto de relojes X, una valuación $v \in \mathcal{V}_X$ y un real $t \in \Re^+$, la valuación $v + t$ asigna a cada reloj $x \in X$ el valor $v(x) + t$.
- Dado un conjunto de relojes X y un real $t \in \Re^+$, llamamos $\overline{\mathbf{t}}$ a la valuación que asigna a cada reloj $x \in X$ el valor t.
- Dado el conjunto de relojes X, definimos el conjunto de restricciones sobre relojes Ψ_X según la siguiente gramática:

$$\psi ::= \top \mid x \sim c \mid x - x' \sim c \mid \psi \wedge \psi \mid \neg\psi$$

  donde $x, x' \in X$, $\sim \in \{<, \leq\}$ y $c \in \mathbf{N}$.

  Definimos inductivamente la relación $\vDash$ incluida en $\mathcal{V}_X \times \Psi_X$ como:

$v \vDash \top$		siempre
$v \vDash x \sim c$	sii	$v(x) \sim c$
$v \vDash x - x' \sim c$	sii	$v(x) - v(x') \sim c$
$v \vDash \psi \wedge \psi'$	sii	$v \vDash \psi$ y $v \vDash \psi'$
$v \vDash \neg\psi$	sii	$v \nvDash \psi$

  Diremos que una valuación $v \in \mathcal{V}_X$ *satisface* una restricción $\psi \in \Psi_X$ si y sólo si $v \vDash \psi$.

  Llamaremos $[\psi]$ al conjunto de las valuaciones que satisfacen ψ, o sea:

$$[\psi] = \{v \in \mathcal{V}_X \mid v \vDash \psi\}$$

2.2.2 Autómatas temporizados

Definición 2.2.1. Autómata temporizado. Un *autómata temporizado* es una tupla $\mathcal{A} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$ donde:

- S es un conjunto finito de *locaciones*
- X es un conjunto finito de *relojes*
- Σ es un conjunto finito de *eventos*
- A es un conjunto finito de *aristas*. Cada arista en A es una tupla $\langle s, a, \psi, \rho, s' \rangle$ donde:
  - $s \in S$ es la *locación de origen*
  - $s' \in S$ es la *locación de destino*
  - $a \in \Sigma \cup \{\lambda\}$ es la *etiqueta* de la arista
  - $\psi \in \Psi_X$ es la *restricción* o *guarda* de la arista
  - $\rho \subseteq X$ determina el conjunto de relojes que se *resetean* al atravesar la arista.
- $\mathcal{I} : S \xrightarrow{tot} \Psi_X$ es una función total que asocia a cada locación una restricción sobre los relojes. Dada una locación $s \in S$, diremos que $\mathcal{I}(s)$ es el *invariante de la locación* s.
- $s_0 \in S$ es la *locación inicial*.

Como los autómatas finitos, los autómatas temporizados admiten una representación gráfica. El siguiente ejemplo muestra un autómata sencillo que modela el funcionamiento de una máquina expendedora de café.

Ejemplo. *Máquina expendedora de café.*
La Figura 2.1 muestra un autómata temporizado que modela el funcionamiento de una expendedora de café. Los nodos corresponden a los estados discretos de la máquina: libre, esperando opción, sirviendo bebida. Los ejes corresponden a los eventos que afectan el funcionamiento de la máquina: introducir una moneda en la máquina, elegir una bebida, etc. La máquina comienza estando libre y puede permanecer así indefinidamente, por esta razón el invariante de la locación es $\top$ y para simplificar el gráfico puede omitirse. Estando en esa locación, en cualquier momento un usuario puede introducir una moneda. Quiere decir que la arista *moneda* tiene como guarda $\top$. Como en el caso de los invariantes, cuando una guarda es $\top$ puede omitirse en el gráfico. Al pasar a la locación *eleccion*, donde la máquina aguardará la opción del usuario, se resetea el reloj x. Este reloj medirá el tiempo transcurrido desde que se introdujo la moneda. La máquina aguardará a lo sumo 10 u.t. en *eleccion* (invariante $x \leq 10$) y si no se produce una elección antes de ese momento, devolverá el dinero recibido y volverá a quedar libre. Si, en cambio, el usuario realiza su elección antes de las 10 u.t., la máquina procederá a servir la bebida correspondiente. Las distintas elecciones posibles están modeladas con dos aristas salientes desde *eleccion*. Dado que la máquina requiere 5 u.t. para servir la bebida, se resetea nuevamente x cuando el usuario realiza su elección y se fuerza a que la máquina deje la locación *sirviendo* recién al cumplirse las 5 u.t. Aunque simple, el ejemplo muestra dos posibles usos de los invariantes: para establecer *deadlines*, como en el caso de *eleccion* o para modelar duración de una acción, como en el caso de *sirviendo*.

Semántica

La semántica de los autómatas temporizados está dada en función de un sistema de transiciones etiquetadas (STE) formado por un conjunto, generalmente no numerable, de *estados* y una relación $\rightarrow$ de *transición*. Cada estado se compone de una locación del autómata y una valuación sobre los relojes. Estando en cualquier estado, $\mathcal{A}$ puede evolucionar atravesando una de las aristas en A (*transición discreta*) o dejando que el tiempo transcurra en la locación actual (*transición temporal*).

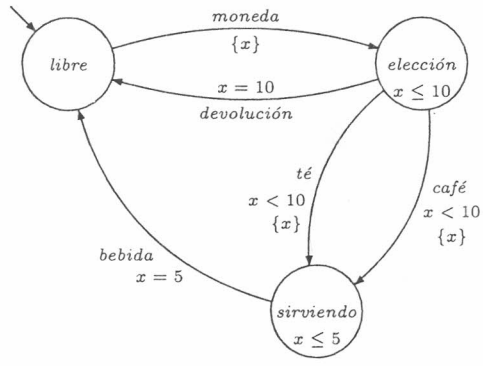


Figura 2.1: Expendedora de café

Estado de un autómata temporizado. Un *estado* q de un autómata $\mathcal{A} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$ es un par $q = (s, v) \in S \times \mathcal{V}_X$ tal que $v \vDash \mathcal{I}(s)$. El estado inicial de $\mathcal{A}$ será $q_{init} = (s_0, \overline{0})$.

Llamamos $Q_{\mathcal{A}}$ (o simplemente Q si queda claro por contexto) al conjunto de todos los estados del autómata $\mathcal{A}$.

Notación. Dado $q = (s, v) \in Q$ definimos:

- $q + t \stackrel{def}{=} (s, v + t)$
- $q^{@} \stackrel{def}{=} s$

Evolución de un autómata temporizado. Sea $\mathcal{A} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$ un autómata temporizado y (s, v) y (s', v') estados de $\mathcal{A}$.

- **Transiciones discretas.** Sea $a \in \Sigma \cup \{\lambda\}$. Existe una *transición discreta* desde el estado (s, v) hacia el estado (s', v') por a, notado $(s, v) \rightarrow^{a} (s', v')$, si para alguna arista $\langle s, a, \psi, \rho, s' \rangle \in A$:

  - $v \vDash \psi$
  - $v' = Reset_{\rho}(v)$

  o bien, $a = \lambda$ y $(s, v) = (s', v')$.

- **Transiciones temporales.** Sea $t \in \Re^{+}$. Existe una *transición temporal* desde el estado (s, v) hacia el estado $(s, v + t)$ por t, notado $(s, v) \rightarrow^{t} (s, v + t)$, si:

  - para todo $t' < t$, $v + t' \vDash \mathcal{I}(s)$

Dado que estamos pidiendo que las transiciones sean únicamente entre estados válidos del autómata, en todo momento el autómata debe respetar los invariantes de las locaciones por las que pasa.

Las transiciones discretas corresponden a un cambio de locación (aunque una arista puede ser un *loop* sobre la misma locación y la locación de destino sería igual a la de partida). Estas transiciones ocurren en forma *instantánea*, provocan que se reinicien los relojes correspondientes, y sólo pueden suceder si el valor de los relojes satisface la guarda de la arista. Generalmente, las transiciones discretas representan la ocurrencia de

un evento en el sistema (caso $a \in \Sigma$), pero también pueden darse transiciones discretas en forma "espontánea" (caso $a = \lambda$). Finalmente, estando en cualquier estado, el autómata puede realizar una transición discreta trivial por λ permaneciendo en el mismo estado.

Las transiciones temporales corresponden al paso del tiempo. Estas transiciones no modifican la parte discreta de los estados (es decir, la locación del estado). Estando en cualquier estado el autómata puede realizar una transición temporal trivial por 0 permaneciendo en el mismo estado. Los autómatas utilizan un modelo de tiempo denso con lo cual, si existe la transición $(s, v) \rightarrow^t (s, v + t)$ y $t = t_1 + t_2$, entonces existen las transiciones $(s, v) \rightarrow^{t_1} (s, v + t_1)$ y $(s, v + t_1) \rightarrow^{t_2} (s, v + t)$.

Definición 2.2.2. Sistema de transiciones etiquetadas (STE).
Dado un autómata temporizado $\mathcal{A}$, el grafo $\langle Q, \rightarrow \rangle$, con $\rightarrow \subseteq Q \times (\Re^+ \cup \Sigma \cup \{\lambda\}) \times Q$ denotará el *sistema de transiciones etiquetadas* generado por $\mathcal{A}$.

La representación clásica de las corridas de los autómatas temporizados se define en función de secuencias infinitas de estados y transiciones de la forma: $q_0 \rightarrow^{a_0} q_1 \rightarrow^{a_1} \ldots q_i \rightarrow^{a_i} \ldots$, donde $\forall i \in \mathbf{N}$, $q_i \in Q$, $a_i \in \Re^+ \cup \Sigma \cup \{\lambda\}$ llamadas *runs* ([Yov96, HNSY92, ACD93, Bra00]). En este trabajo representaremos las corridas de un autómata usando *evoluciones*.

Evolución en un paso. Diremos que un autómata temporizado $\mathcal{A} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$ *puede evolucionar en un paso* desde el estado q hacia el estado q' por $a \in \Sigma \cup \{\lambda\}$ y $t \in \Re^+$, notado $q \Rightarrow_t^a q'$, sii:

$$q \rightarrow^t q + t \quad \text{y} \quad q + t \rightarrow^a q'$$

Una evolución en un paso corresponde a dejar pasar cierto tiempo en una locación y luego realizar una transición discreta. Se puede ver que toda corrida de un autómata puede ser representada como una sucesión de estas evoluciones en un paso.

Definición 2.2.3. Evolución.
Dado un autómata temporizado $\mathcal{A} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$ una *evolución* r es una secuencia (posiblemente infinita) de evoluciones en un paso de la forma:

$$r = q_0 \Rightarrow_{t_0}^{a_0} q_1 \Rightarrow_{t_1}^{a_1} \ldots q_i \Rightarrow_{t_i}^{a_i} \ldots$$

donde $\forall i \in \mathbf{N}$, $q_i \in Q$, $a_i \in \Sigma \cup \{\lambda\}$ y $t_i \in \Re^+$.

Dado un estado q de $\mathcal{A}$, llamaremos $\mathcal{R}_q(\mathcal{A})$ al conjunto de todas las evoluciones de $\mathcal{A}$ que comiencen en el estado q. Llamaremos $\mathcal{R}(\mathcal{A})$ al conjunto $\mathcal{R}_{q_{init}}(\mathcal{A})$, es decir, al conjunto de todas las evoluciones de $\mathcal{A}$ que comiencen en el estado inicial q_{init}.

Dada una secuencia finita ς sobre $\Sigma \cup \{\lambda\}$ y una secuencia temporal τ de la misma longitud, definimos la relación $q \Rightarrow_\tau^\varsigma q'$ (que se lee "es posible evolucionar desde q hacia q' por $\langle \varsigma, \tau \rangle$") de la siguiente manera: $q \Rightarrow_\tau^\varsigma q'$ sii

$$\begin{cases} q = q' & \text{si } |\varsigma| = |\tau| = 0 \\ \exists q'',\ q \Rightarrow_{\tau'}^{\varsigma'} q'' \ \text{y} \ q'' \Rightarrow_t^a q' & \text{si } \varsigma = \varsigma' a \text{ y } \tau = \tau' \lhd t \end{cases}$$

Diremos también que un autómata puede *evolucionar* desde el estado q hacia el estado q' (notado $q \Rightarrow q'$) si existe un par de secuencias $\langle \varsigma, \tau \rangle$ tales que $q \Rightarrow_\tau^\varsigma q'$.

Observación. Todo *run* de un autómata es también una *evolución*. Por otro lado, toda evolución tiene un *run* equivalente que se obtiene reemplazando las evoluciones en un paso por el par de transiciones que representan.

Por ejemplo, el *run*:

$$q_0 \rightarrow^{a} q_1 \rightarrow^{b} q_2 \rightarrow^{1.5} q_3 \rightarrow^{3.14} q_4$$

puede ser visto como la evolución:

$$q_0 \Rightarrow_0^{a} q_1 \Rightarrow_0^{b} q_2 \Rightarrow_{1.5}^{\lambda} q_3 \Rightarrow_{3.14}^{\lambda} q_4$$

Por otro lado, la evolución:

$$q_0 \Rightarrow_{4.5}^{a} q_1 \Rightarrow_3^{b} q_2 \Rightarrow_{1.5}^{\lambda} q_3 \Rightarrow_0^{c} q_4$$

puede ser transformada en *run* "desdoblando" las evoluciones en un paso:

$$q_0 \rightarrow^{4.5} (q_0 + 4.5) \rightarrow^{a} q_1 \rightarrow^{3} (q_1 + 3) \rightarrow^{b} q_2 \rightarrow^{1.5} (q_2 + 1.5) \rightarrow^{\lambda} q_3 \rightarrow^{0} q_3 \rightarrow^{c} q_4$$

Como se puede ver, la diferencia entre *runs* y evoluciones es muy sutil y ésta es la justificación para apartarnos de la semántica clásica en favor de la semántica basada en evoluciones.

Notación. Dado un autómata $\mathcal{A}$, un par de estados q, q' en $Q_{\mathcal{A}}$ y un par de locaciones s, s' en S, definimos el siguiente conjunto de abreviaturas:

$q \Rightarrow_\tau^\varsigma s$	$\exists v \in \mathcal{V}_X,\ q \Rightarrow_\tau^\varsigma (s, v)$
$q \Rightarrow_\tau^\varsigma s$	$\exists \tau,\ q \Rightarrow_\tau^\varsigma s$
$q \Rightarrow s$	$\exists \varsigma,\ q \Rightarrow^\varsigma s$
$s \Rightarrow_\tau^\varsigma q$	$\exists v \in \mathcal{V}_X,\ (s, v) \Rightarrow_\tau^\varsigma q$
$s \Rightarrow_\tau^\varsigma q$	$\exists \tau,\ s \Rightarrow_\tau^\varsigma q$
$s \Rightarrow q$	$\exists \varsigma,\ s \Rightarrow^\varsigma q$
$s \Rightarrow_\tau^\varsigma s'$	$\exists v, v' \in \mathcal{V}_X,\ (s, v) \Rightarrow_\tau^\varsigma (s', v')$
$s \Rightarrow_\tau^\varsigma s'$	$\exists \tau,\ s \Rightarrow_\tau^\varsigma s'$
$s \Rightarrow s'$	$\exists \varsigma,\ s \Rightarrow^\varsigma s'$

Tiempo transcurrido en una evolución. Dada una evolución $r = q_0 \Rightarrow_{t_0}^{a_0} q_1 \Rightarrow_{t_1}^{a_1} q_2 \Rightarrow_{t_2}^{a_2} \cdots$ de $\mathcal{A}$, el *tiempo transcurrido en r hasta la iésima evolución en un paso*, denominado $\tau_r(i)$, se define como:

$$\tau_r(i) = \sum_{k=0}^{i} t_k$$

Las evoluciones de los autómatas temporizados siguen un modelo similar al de las ejecuciones temporizadas mencionadas en la sección anterior: las corridas de un autómata estarán dadas por evoluciones infinitas y divergentes, con un progreso del tiempo débilmente monótono para permitir eventos simultáneos.

Evolución divergente. Dado un autómata temporizado $\mathcal{A}$, diremos que una evolución (infinita) r es *divergente* sii:

$$\lim_{i \to \infty} \tau_r(i) = \infty$$

Dado un autómata temporizado $\mathcal{A}$ y un estado q de $\mathcal{A}$, llamaremos $\mathcal{R}_q^\infty(\mathcal{A})$ al conjunto de todas las evoluciones *divergentes* de $\mathcal{A}$ que comiencen en el estado q.

Dado un autómata temporizado $\mathcal{A}$, llamaremos $\mathcal{R}^\infty(\mathcal{A})$ al conjunto $\mathcal{R}_{q_{init}}^\infty(\mathcal{A})$, es decir, al conjunto de todas las evoluciones divergentes de $\mathcal{A}$ que comiencen en el estado inicial q_{init}.

Diremos que un autómata $\mathcal{A}$ es *non-zeno* si toda evolución finita que comience en q_{init} es prefijo de alguna evolución en $\mathcal{R}^{\infty}(\mathcal{A})$. En general, pediremos que un autómata que modele el comportamiento de un sistema tenga la propiedad de ser non-zeno. Esta propiedad garantiza que cualquier secuencia finita de transiciones, que parta desde el estado inicial, forma parte de por lo menos una evolución infinita y divergente. Si esto no fuera así, el autómata estaría permitiendo operaciones donde llegado un punto no se podría dejar que el tiempo diverja, cosa que no es posible en el "mundo real" que se está modelando (donde el tiempo avanza inexorablemente).

Extenderemos la notación presentada para *secuencias* a evoluciones. Dada una evolución $r \in \mathcal{R}_q^{\infty}(\mathcal{A})$, una **posición** de r es un par $(i,t) \in \mathbf{N} \times \Re^+$ tal que $t \leq t_i$. Llamaremos Π_r al conjunto de todas las posiciones de r. Dada una posición $(i,t) \in \Pi_r$, el *estado* en dicha posición se define como:

$$r_{(i,t)} = q_i + t$$

y el *tiempo* transcurrido hasta dicha posición se define como:

$$\tau_r(i,t) = \tau_r(i) + t$$

Definimos el orden total $\ll$ sobre las posiciones de una evolución de la siguiente manera:

$$(i,t) \ll (j,t') \qquad \text{sii} \qquad i < j \quad \text{o} \quad i = j \text{ y } t \leq t'$$

Dadas dos posiciones $p, p' \in \Pi_r$, $r_{[p}$ será la porción de r que comienza desde el estado r_p, $r_{p]}$ será la porción de r desde el comienzo hasta el estado r_p y $r_{[p,p']}$ será la porción de r desde r_p hasta $r_{p'}$. Como en los casos anteriores, usar '(' o ')' en lugar de '[' y ']' significará que los bordes correspondientes no están incluidos en la subevolución.

2.2.3 Lenguaje de un autómata temporizado

Como vimos, clásicamente se utiliza un modelo basado en estados y transiciones para representar el comportamiento de los autómatas temporizados. Sin embargo, es posible representar el comportamiento de un autómata temporizado utilizando las *ejecuciones* temporizadas y no temporizadas que definimos en la sección 2.1. Dado que la semántica de nuestros patrones de eventos estará dada en base a estas ejecuciones, resulta conveniente poder definir cuándo un autómata temporizado *acepta* una ejecución y cuándo no.

Definición 2.2.4. Ejecución expuesta por una evolución.
Dada una evolución

$$r = q_0 \Rightarrow_{t_0}^{a_0} q_1 \Rightarrow_{t_1}^{a_1} \dots q_i \Rightarrow_{t_i}^{a_i} \dots$$

donde $\forall i \in \mathbf{N}$, $q_i \in Q$, $a_i \in \Sigma \cup \{\lambda\}$ y $t_i \in \Re^+$, llamaremos $\overline{r}$ a la ejecución temporizada $\langle \varsigma, \tau \rangle$, donde:

- $\varsigma = a_0 a_1 \dots a_i \dots$
- $\tau = t_0 \lhd t_1 \lhd \dots \lhd t_i \lhd \dots$

$\overline{r}$ es la ejecución temporizada obtenida de abstraer los estados intermedios de r.

Ejemplo. Dada la evolución:

$$r = q_0 \Rightarrow_{4.5}^{a} q_1 \Rightarrow_{3}^{b} q_2 \Rightarrow_{1.5}^{\lambda} q_3 \Rightarrow_{0}^{c} q_4$$

la ejecución expuesta por r será:

$$\overline{r} = \left\langle \begin{array}{cccc} a & b & \lambda & c \\ 4.5 & 7.5 & 9 & 9 \end{array} \right\rangle$$

Definición 2.2.5. Aceptación de ejecuciones temporizadas.
Dado un autómata temporizado $\mathcal{A}$, diremos que $\mathcal{A}$ *acepta* una ejecución temporizada $\sigma = \langle \varsigma, \tau \rangle$ sii existe una evolución r en $\mathcal{R}^\infty(\mathcal{A})$, tal que $\sigma = \overline{r}$.

Diremos que $\mathcal{A}$ *acepta* una ejecución (no temporizada) ς sii $\mathcal{A}$ acepta alguna ejecución temporizada de la forma $\langle \varsigma, \tau \rangle$.

Definición 2.2.6. Lenguaje de autómatas temporizados.
Dado un autómata $\mathcal{A}$, el *lenguaje* de $\mathcal{A}$ estará dado por las ejecuciones aceptadas por $\mathcal{A}$.

Formalmente, definiremos *lenguaje de un autómata temporizado* $\mathcal{A}$ (notado $\mathcal{L}(\mathcal{A})$), de la siguiente manera:

$$\mathcal{L}(\mathcal{A}) = \{\sigma \mid \sigma \text{ es } \textit{aceptada} \text{ por } \mathcal{A}\} = \{\sigma \mid \exists r \in \mathcal{R}^\infty(\mathcal{A}),\ \overline{r} = \sigma\}$$

Análogamente, definiremos *lenguaje no temporizado de un autómata temporizado* $\mathcal{A}$ (notado $\mathcal{L}^*(\mathcal{A})$), de la siguiente manera:

$$\mathcal{L}^*(\mathcal{A}) = \{\varsigma \mid \varsigma \text{ es } \textit{aceptada} \text{ por } \mathcal{A}\} = \{\varsigma \mid \exists r \in \mathcal{R}^\infty(\mathcal{A}),\ \exists \tau,\ \overline{r} = \langle \varsigma, \tau \rangle\}$$

2.2.4 Composición de Autómatas Temporizados

El formalismo de autómatas temporizados permite modelar individualmente los distintos componentes de un sistema. La integración de esos componentes dentro del sistema está dada por la *composición paralela* de los mismos.

Dado un par de autómatas temporizados $\mathcal{A}_1$ y $\mathcal{A}_2$ con conjuntos disjuntos de relojes, la composición paralela de ambos ($\mathcal{A}_1 || \mathcal{A}_2$) se construye a partir del producto cartesiano de sus locaciones, la unión de los relojes y la sincronización de las aristas con eventos en común. El invariante de una locación compuesta será la conjunción de los invariantes de sus componentes. Para una arista de sincronización, la guarda será la conjunción de las guardas originales y el conjunto de relojes que se reinician será la unión de los conjuntos locales. Formalmente:

Definición 2.2.7. Composición paralela.
Dados dos autómatas temporizados $\mathcal{A}_1 = \langle S_1, X_1, \Sigma_1, A_1, \mathcal{I}_1, s_{01} \rangle$, y $\mathcal{A}_2 = \langle S_2, X_2, \Sigma_2, A_2, \mathcal{I}_2, s_{02} \rangle$ donde $X_1 \cap X_2 = \varnothing$, y $\Sigma_1 \cap \Sigma_2$ constituye el conjunto de *eventos de sincronización*, llamaremos $\mathcal{A}_1 || \mathcal{A}_2 = \langle S_1 \times S_2, X_1 \cup X_2, \Sigma_1 \cup \Sigma_2, A, \mathcal{I}, (s_{01}, s_{02}) \rangle$ a la *composición paralela* de $\mathcal{A}_1$ con $\mathcal{A}_2$ donde[1]:

- $\langle (s_1, s_2), l, \psi, \rho, (s_1', s_2') \rangle \in A$ sii
  - $\langle s_1, l, \psi, \rho, s_1' \rangle \in A_1$, $l \notin \Sigma_2$ y $s_2 = s_2'$
  - $\langle s_2, l, \psi, \rho, s_2' \rangle \in A_2$, $l \notin \Sigma_1$ y $s_1 = s_1'$
  - $\langle s_1, l, \psi_1, \rho_1, s_1' \rangle \in A_1$, $\langle s_2, \lambda, \psi_2, \rho_2, s_2' \rangle \in A_2$, $l \notin \Sigma_2$, $\psi = \psi_1 \wedge \psi_2$ y $\rho = \rho_1 \cup \rho_2$
  - $\langle s_2, l, \psi_2, \rho_2, s_2' \rangle \in A_2$, $\langle s_1, \lambda, \psi_1, \rho_1, s_1' \rangle \in A_1$, $l \notin \Sigma_1$, $\psi = \psi_1 \wedge \psi_2$ y $\rho = \rho_1 \cup \rho_2$
  - $\langle s_i, l, \psi_i, \rho_i, s_i' \rangle \in A_i$, con $i = 1, 2$, $l \in \Sigma_1 \cap \Sigma_2$, $\psi = \psi_1 \wedge \psi_2$ y $\rho = \rho_1 \cup \rho_2$

  En los primeros dos casos, decimos que sólo una de las componentes *participa* de la arista, por el contrario, en los últimos tres casos decimos que ambas componentes participan.

- $\mathcal{I}((s_1, s_2)) = \mathcal{I}(s_1) \wedge \mathcal{I}(s_2)$

[1] En realidad, alcanza con tomar la componente conexa que contenga al par (s_{01}, s_{02})

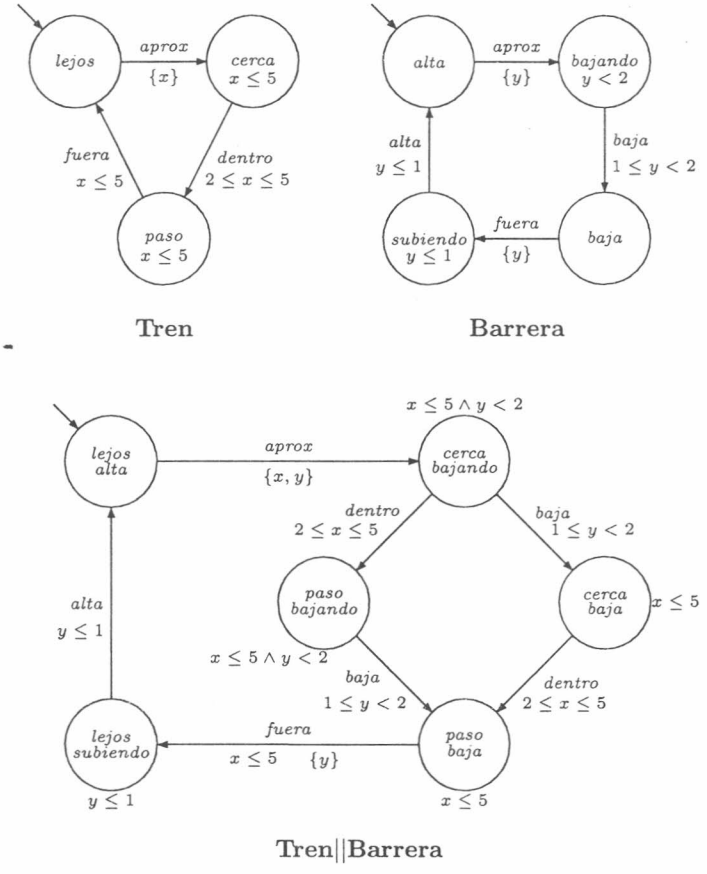


Figura 2.2: Cruce ferroviario

Notación. Sea un autómata temporizado $\mathcal{A}$ construido como la composición paralela de $\mathcal{A}_1||\mathcal{A}_2$. Dada una locación s de $\mathcal{A}$, para $i = 1, 2$ definimos $\pi_i(s) = s_i$ como la locación correspondiente a la iésima componente. Dada la transición $\langle s, l, \psi, \rho, s' \rangle$ en $\mathcal{A}$, escribimos $\pi_i(\psi) = \psi_i$, la condición de la arista correspondiente a la iésima componente. Por definición, si la componente i no participa en la transición diremos que $\pi_i(\psi) = \top$. Llamaremos $\pi_i(\rho) = \rho_i$ al conjunto de relojes reseteados en la arista correspondiente a la iésima componente. Si la componente i no participa en la relación diremos que $\pi_i(\rho) = \varnothing$. Finalmente, dado un estado q de $\mathcal{A}$, diremos que $\pi_i(q)$ es el estado local de $\mathcal{A}_i$.

La noción de composición paralela puede ser generalizada naturalmente n componentes $\mathcal{A}_1, \ldots, \mathcal{A}_n$ y decimos que $\mathcal{A}_1|| \cdots ||\mathcal{A}_n$ es el resultado de asociar los términos de izquierda a derecha: $(\cdots((\mathcal{A}_1||\mathcal{A}_2)||\mathcal{A}_3)\cdots)||\mathcal{A}_n$.

Ejemplo. *Cruce de ferrocarril.*
La Figura 2.2 muestra un ejemplo de sistema modelado con dos componentes. El ejemplo constituye un caso de estudio clásico mencionado en la bibliografía. Consiste en un cruce de ferrocarril modelado con una componente que representa la *barrera* y otra componente que representa el *tren*. Los eventos que sincronizan el funcionamiento de la barrera y el tren son *aprox* y *fuera*. Cuando el tren se acerca al cruce ferroviario, envía una señal *aprox* a la barrera y entra al cruce al menos 2 u.t. después. El tren se aleja del cruce, enviando una señal *fuera* a la barrera, dentro de las 5 u.t. desde que señalizó que se estaba aproximando. La barrera requiere entre 1 y 2 u.t. para bajar completamente y a lo sumo 1 u.t para subir completamente. La Figura 2.2 también muestra la composición paralela de ambas componentes.

Observaciones sobre la semántica de la composición paralela

Dado $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2$, se puede ver que existirá una **transición discreta** $q \rightarrow^l q'$ en $\mathcal{A}$ si y sólo si para $i = 1, 2$:

$$\begin{cases} \pi_i(q) \rightarrow^l \pi_i(q') & \text{si } l \in \Sigma_i \\ \pi_i(q) \rightarrow^\lambda \pi_i(q') & \text{en otro caso} \end{cases}$$

También se puede ver que existirá una **transición temporal** $q \rightarrow^t q'$ en $\mathcal{A}$ si y sólo si para $i = 1, 2$:

$$\pi_i(q) \rightarrow^t \pi_i(q')$$

Proposición 2.1. *Dados dos autómatas temporizados* $\mathcal{A}_i = \langle S_i, X_i, \Sigma_i, A_i, \mathcal{I}_i, s_{0i} \rangle$ *con* $i = 1, 2$ *y una ejecución* σ *sobre* $\Sigma_1 \cup \Sigma_2$,

$$\sigma \in \mathcal{L}(\mathcal{A}_1||\mathcal{A}_2) \quad sii \quad \sigma\lceil_{\Sigma_1} \in \mathcal{L}(\mathcal{A}_1) \text{ y } \sigma\lceil_{\Sigma_2} \in \mathcal{L}(\mathcal{A}_2)$$

Demostración. La demostración de esta proposición es más o menos directa a partir de la semántica de las transiciones de un autómata compuesto. La demostración completa se encuentra en el Apéndice A, página 92. □

2.3 Autómatas de Büchi temporizados

Los autómatas de Büchi son un tipo particular de ω-autómatas. A su vez, los ω-autómatas son extensiones de los autómatas finitos que incluyen una *condición de aceptación* para palabras infinitas (denominadas ω-palabras). Una de las condiciones de aceptación para ω-palabras más usada es la condición de aceptación de Büchi, que justamente caracteriza a los autómatas de Büchi. Informalmente, la condición de aceptación de Büchi pide que se visiten infinitas veces ciertas locaciones distinguidas del autómata.

Los autómatas de Büchi *temporizados* son autómatas temporizados a los cuales se les agrega una condición de aceptación de Büchi.

Definición 2.3.1. Autómata de Büchi temporizado.
Un *autómata de Büchi temporizado* es una tupla $\mathcal{B} = \langle \mathcal{A}, \mathcal{F} \rangle$ donde $\mathcal{A} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$ es un autómata temporizado y $\mathcal{F} \subseteq S$ es el conjunto de locaciones *de aceptación*.

Semántica

Dado un autómata de Büchi temporizado $\mathcal{B} = \langle \mathcal{A}, \mathcal{F} \rangle$ y una evolución $r = q_0 \Rightarrow^{a_0}_{t_0} q_1 \Rightarrow^{a_1}_{t_1} \dots q_i \Rightarrow^{a_i}_{t_i} \dots$ de $\mathcal{A}$, llamaremos $inf(r)$ al conjunto de locaciones $s \in S$ tales que $s = q_i^{@}$ para una cantidad infinita de i.

Definición 2.3.2. Aceptación de evoluciones.
Dado un autómata de Büchi temporizado $\mathcal{B} = \langle \mathcal{A}, \mathcal{F} \rangle$, diremos que $\mathcal{B}$ *acepta* una evolución $r \in \mathcal{R}^\infty(\mathcal{A})$ si y sólo si $inf(r) \cap \mathcal{F} \neq \varnothing$.

Llamaremos $\mathcal{R}^\infty(\mathcal{B})$ a las evoluciones divergentes aceptadas por $\mathcal{B}$.

Informalmente podemos decir que un autómata de Büchi temporizado *aceptará* una evolución si algunas de las locaciones en $\mathcal{F}$ son visitadas un número infinito de veces en el transcurso de dicha evolución.

Es fácil ver que si $\mathcal{F} = S$, la condición de aceptación se vuelve trivialmente verdadera y $\mathcal{B}$ aceptará todas las evoluciones de $\mathcal{A}$. En cambio, si $\mathcal{F} = \varnothing$, la condición de aceptación será siempre falsa y $\mathcal{B}$ no aceptará ninguna evolución.

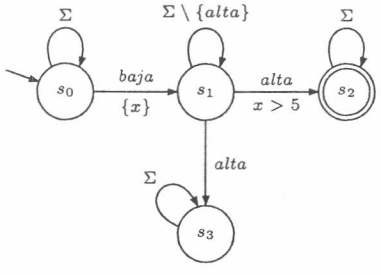


Figura 2.3: Ejemplo de autómata de Büchi

Definición 2.3.3. Aceptación de ejecuciones temporizadas. Dado un autómata de Büchi temporizado $\mathcal{B} = \langle \mathcal{A}, \mathcal{F} \rangle$, diremos que $\mathcal{B}$ *acepta* una ejecución temporizada σ sii existe una evolución r en $\mathcal{R}^{\infty}(\mathcal{B})$, tal que $\sigma = \overline{r}$.

Diremos que $\mathcal{B}$ *acepta* una ejecución (no temporizada) ς sii $\mathcal{B}$ acepta alguna ejecución temporizada de la forma $\langle \varsigma, \tau \rangle$.

Definición 2.3.4. Lenguaje de autómatas de Büchi temporizados.
Dado un autómata de Büchi temporizado $\mathcal{B}$, el *lenguaje* de $\mathcal{B}$ (notado $\mathcal{L}(\mathcal{B})$) estará dado por las ejecuciones aceptadas por $\mathcal{B}$.

Formalmente:

$$\mathcal{L}(\mathcal{B}) = \{\sigma \mid \sigma \text{ es } \textit{aceptada} \text{ por } \mathcal{B}\} = \{\sigma \mid \exists r \in \mathcal{R}^{\infty}(\mathcal{B}),\ \overline{r} = \sigma\}$$

Análogamente, definiremos *lenguaje no temporizado* de $\mathcal{B}$ (notado $\mathcal{L}^{\star}(\mathcal{B})$), de la siguiente manera:

$$\mathcal{L}^{\star}(\mathcal{B}) = \{\varsigma \mid \varsigma \text{ es } \textit{aceptada} \text{ por } \mathcal{B}\} = \{\varsigma \mid \exists r \in \mathcal{R}^{\infty}(\mathcal{B}),\ \exists \tau,\ \overline{r} = \langle \varsigma, \tau \rangle\}$$

Ejemplo. La Figura 2.3 muestra un ejemplo de autómata de Büchi. La representación gráfica es similar a la de los autómatas temporizados y las locaciones de aceptación se marcan con un doble borde. El autómata del ejemplo acepta todas las ejecuciones del Cruce ferroviario en las cuales la barrera tarda más de 5 u.t. en subir completamente. Esta es una propiedad *no deseada* para el cruce ferroviario.
Estado en la locación inicial s_0, ante la ocurrencia del evento *baja* (es decir, "la barrera terminó de bajar") se puede decidir entre permanecer en s_0 o atravesar la arista hacia la locación s_1. Al atravesar la arista se está tomando la decisión de medir el tiempo desde que la barrera bajó hasta que la barrera vuelva a estar alta. Si el siguiente evento *alta* (es decir, "la barrera terminó de subir") ocurre a lo sumo 5 u.t. después, entonces la arista que va de s_1 a s_2 no estará habilitada y sólo se podrá tomar la arista hacia s_3. La locación s_3 no tiene ninguna arista saliente, solo *stutters*[2] por todos los eventos. A este tipo de locaciones se las denomina *trampa*. Si, por el contrario, estando en s_1 el siguiente evento *alta* ocurre después de las 5 u.t., se podrá decidir entre pasar a s_2 o a s_3. La locación de aceptación s_2 también es una locación trampa. Quiere decir que una vez que una evolución (infinita) del autómata alcanzó s_2, pasará infinitas veces por esa locación, cumpliendo de esa forma la condición de aceptación de Büchi.
Dado que un autómata de Büchi acepta una ejecución si es la abstracción de al menos una evolución que satisface la condición de aceptación, alcanza con que exista *una* forma de evolucionar sobre el autómata desde el estado inicial, siguiendo la ejecución y visitando infinitas veces alguna de las locaciones de aceptación para

[2]Aristas cuyo origen y destino son la misma locación.

que el autómata la acepte. Quiere decir que si en una ejecución existe una ocurrencia de *baja* separada de la siguiente ocurrencia de *alta* por más de 5 u.t., habrá muchas formas de evolucionar sobre el autómata siguiendo la ejecución y sin llegar a s_2, pero lo importante es que existirá al menos una manera de hacerlo llegando a esa locación.

Producto entre autómatas temporizados y autómatas de Büchi temporizados

Definimos la operación *producto* entre un autómata temporizado $\mathcal{A}$ y un autómata de Büchi temporizado $\mathcal{B}$. Este producto estará basado en la composición paralela de autómatas temporizados pero tendrá la particularidad de ser en sí mismo un autómata de Büchi.

La operación $\otimes$ entre un autómata temporizado $\mathcal{A}_1$ y un autómata de Büchi temporizado $\mathcal{B} = \langle \mathcal{A}_2, \mathcal{F} \rangle$ dará como resultado un nuevo autómata de Büchi donde la estructura subyacente estará dada por la composición paralela de $\mathcal{A}_1$ y $\mathcal{A}_2$ y la condición de aceptación estará dada por las locaciones compuestas que contengan locaciones de $\mathcal{F}$.

Cabe aclarar que es posible definir una noción más general de "composición" (o "intersección") de autómatas de Büchi temporizados (dada por ejemplo en [AD94]). Sin embargo, la construcción de la intersección de autómatas de Büchi es poco natural cuando se utiliza un único conjunto de locaciones de aceptación, como se hizo en la definición 2.3.1 (que sigue la línea de [AD94], [Bra00]). Para definir la composición de autómatas de Büchi es más conveniente utilizar la versión *generalizada* de los mismos, que incluye un conjunto de condiciones de aceptación. En esta versión, $\mathcal{F} \subseteq 2^S$ es un conjunto de conjuntos de locaciones y la condición de aceptación está dada por visitar infinitas veces locaciones que pertenezcan a cada uno de estos conjuntos. Dado que se ha demostrado que la versión generalizada de los autómatas de Büchi tiene el mismo poder expresivo que la versión presentada en la sección anterior (por ejemplo, [CGP99]), usaremos esta última por conveniencia en cuanto al uso que le daremos en este trabajo.

Definición 2.3.5. Producto entre autómatas temporizados y autómatas de Büchi temporizados. Dado un autómata temporizado $\mathcal{A}_1$ y un autómata de Büchi temporizado $\mathcal{B} = \langle \mathcal{A}_2, \mathcal{F} \rangle$, el *producto* entre $\mathcal{A}_1$ y $\mathcal{B}$ (notado $\mathcal{A}_1 \otimes \mathcal{B}$) será un autómata de Büchi temporizado $\mathcal{B}' = \langle \mathcal{A}', \mathcal{F}' \rangle$ tal que:

- $\mathcal{A}' = \mathcal{A}_1 || \mathcal{A}_2$
- $\mathcal{F}' = S_1 \times \mathcal{F}$

Proposición 2.2. *Dados dos autómatas temporizados $\mathcal{A}_i = \langle S_i, X_i, \Sigma_i, A_i, \mathcal{I}_i, s_{0i} \rangle$, con $i = 1, 2$, un autómata de Büchi temporizado $\mathcal{B} = \langle \mathcal{A}_2, \mathcal{F} \rangle$, con $\mathcal{F} \subseteq S_2$ y una ejecución σ sobre $\Sigma_1 \cup \Sigma_2$,*

$$\sigma \in \mathcal{L}(\mathcal{A}_1 \otimes \mathcal{B}) \quad sii \quad \sigma \restriction_{\Sigma_1} \in \mathcal{L}(\mathcal{A}_1) \ y \ \sigma \restriction_{\Sigma_2} \in \mathcal{L}(\mathcal{B})$$

Demostración. La demostración es bastante directa usando la Proposición 2.1. La demostración completa se puede ver en el Apéndice A, página 93. □

2.4 Lógica TCTL

TCTL (*Timed Computational Tree Logic*, presentada en [ACD93]), es una extensión de la lógica CTL introducida por [EC81] que permite expresar propiedades temporales *cuantitativas*.

Sintaxis

Definición 2.4.1. Intervalos.
Llamaremos $\mathfrak{I}_{\mathbf{N}}$ al conjunto de todos los intervalos θ delimitados por naturales construidos de la siguiente manera:

$$\theta ::= (a,b) \mid (a,b] \mid [a,b) \mid [a,b] \mid [a,\infty) \mid (a,\infty)$$

donde $a, b \in \mathbf{N}$.

Dado un conjunto finito de proposiciones booleanas PROPS, un *fórmula* TCTL se define según las siguientes reglas sintácticas:

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \exists\mathcal{U}_\theta \phi \mid \phi \forall\mathcal{U}_\theta \phi$$

Donde $p \in$ PROPS y $\theta \in \mathfrak{I}_{\mathbf{N}}$.

Abreviaturas

Para facilitar la escritura de fórmulas TCTL se definen las siguientes abreviaturas de uso común:

$\bot$	$\neg\top$
$\phi_1 \vee \phi_2$	$\neg(\neg\phi_1 \wedge \neg\phi_2)$
$\phi_1 \Rightarrow \phi_2$	$\neg\phi_1 \vee \phi_2$
$\phi_1 \exists\mathcal{U} \phi_2$	$\phi_1 \exists\mathcal{U}_{[0,\infty)} \phi_2$
$\phi_1 \forall\mathcal{U} \phi_2$	$\phi_1 \forall\mathcal{U}_{[0,\infty)} \phi_2$
$\exists\Diamond_\theta \phi$	$\top \exists\mathcal{U}_\theta \phi$
$\forall\Diamond_\theta \phi$	$\top \forall\mathcal{U}_\theta \phi$
$\exists\Diamond \phi$	$\exists\Diamond_{[0,\infty)} \phi$
$\forall\Diamond \phi$	$\forall\Diamond_{[0,\infty)} \phi$
$\forall\Box \phi$	$\neg\exists\Diamond\neg\phi$
$\exists\Box \phi$	$\neg\forall\Diamond\neg\phi$

Semántica de TCTL

Las fórmulas TCTL se interpretan sobre el sistema de transiciones etiquetadas $\langle Q_{\mathcal{A}}, \rightarrow \rangle$, generado por un autómata temporizado $\mathcal{A} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, y una función $\mathbf{P} :$ PROPS $\longrightarrow 2^S$, que asocia a cada proposición un conjunto de locaciones de $\mathcal{A}$. Diremos que un *modelo* TCTL es una tupla $\langle Q_{\mathcal{A}}, \rightarrow, \mathbf{P} \rangle$.

Intuitivamente, las proposiciones se usan para nombrar conjuntos de locaciones del autómata (por ejemplo: el conjunto de las locaciones donde "el sistema está inestable").

El operador $\Diamond$ indica *posibilidad*, es decir, estando en un estado q, $\exists\Diamond\phi$ significa que hay alguna evolución (que parte desde q) en la que eventualmente vale ϕ, $\forall\Diamond\phi$ significa que para toda evolución desde q, inevitablemente vale ϕ. Al agregar a la fórmula un intervalo θ se expresa *posibilidad acotada*: es posible pero además dentro de cierto intervalo.

El operador $\Box$ indica *necesidad*, es decir, estando en un estado q, $\exists\Box\phi$ dice que hay alguna evolución que parte de q en que siempre vale ϕ, $\forall\Box\phi$ dice que en todas las evoluciones que parten de q, siempre vale ϕ.

La fórmula $\phi_1 \exists\mathcal{U} \phi_2$ dice que existe una evolución tal que en algún momento vale ϕ_2 y, hasta ese momento, vale siempre ϕ_1. Análogamente, $\phi_1 \forall\mathcal{U} \phi_2$ dice que para *toda* evolución vale la propiedad anterior. Nuevamente, al agregar un intervalo θ a la fórmula se restringe el momento en el tiempo en el cual puede valer ϕ_2. Por

ejemplo, $\phi_1 \exists\mathcal{U}_\theta \phi_2$ indica que para alguna evolución, existe un prefijo finito cuya duración respeta θ, tal que al final del mismo vale ϕ_2 y ϕ_1 vale en todos los estados intermedios. A continuación definimos formalmente la semántica de TCTL.

Dado un modelo $\mathcal{M} = \langle Q_{\mathcal{A}}, \rightarrow, \mathbf{P} \rangle$, una fórmula TCTL ϕ, y un estado $q \in Q_{\mathcal{A}}$, definimos inductivamente la relación $q \models_{\mathcal{M}} \phi$ de la siguiente manera:

$q \models_{\mathcal{M}} \top$ siempre

$q \models_{\mathcal{M}} p$ si $q^{@} \in \mathbf{P}(p)$

$q \models_{\mathcal{M}} \neg\phi$ si $q \not\models_{\mathcal{M}} \phi$

$q \models_{\mathcal{M}} \phi_1 \wedge \phi_2$ si $q \models_{\mathcal{M}} \phi_1$ y $q \models_{\mathcal{M}} \phi_2$

$q \models_{\mathcal{M}} \phi_1 \exists\mathcal{U}_\theta \phi_2$ si existe $r \in \mathcal{R}_q(\mathcal{A})$ tal que
$\exists k \in \Pi_r,\ r_k \models_{\mathcal{M}} \phi_2$ y $\tau_r(k) \in \theta$ y
$\forall k' \in \Pi_r,\ k' \ll k,\ r_{k'} \models_{\mathcal{M}} \phi_1$ o $(r_{k'} \models_{\mathcal{M}} \phi_2$ y $\tau_r(k') \in \theta)$

$q \models_{\mathcal{M}} \phi_1 \forall\mathcal{U}_\theta \phi_2$ si para todo $r \in \mathcal{R}_q^{\infty}(\mathcal{A})$,
$\exists k \in \Pi_r,\ r_k \models_{\mathcal{M}} \phi_2$ y $\tau_r(k) \in \theta$ y
$\forall k' \in \Pi_r,\ k' \ll k,\ r_{k'} \models_{\mathcal{M}} \phi_1$ o $(r_{k'} \models_{\mathcal{M}} \phi_2$ y $\tau_r(k') \in \theta)$

Diremos que un estado q *satisface* una fórmula TCTL ϕ (con respecto a un modelo $\mathcal{M}$) si y sólo si $q \models_{\mathcal{M}} \phi$. Usaremos directamente la notación $q \models \phi$ cuando quede claro por contexto a qué modelo nos estamos refiriendo.

Dada una fórmula TCTL ϕ y un modelo $\mathcal{M}$, llamaremos $[|\phi|]_{\mathcal{M}}$ al *conjunto característico* de ϕ, es decir, al conjunto de estados del modelo que satisfacen ϕ. Diremos que un autómata temporizado $\mathcal{A}$ *satisface* una fórmula ϕ cuando todos sus estados lo hacen y lo notaremos $\mathcal{A} \models_{\mathcal{M}} \phi$. Nuevamente, usaremos las versiones abreviadas $[|\phi|]$ y $\mathcal{A} \models \phi$ cuando quede claro por contexto a qué modelo nos estamos refiriendo.

Capítulo 3

Patrones de eventos

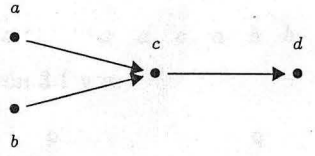
En este capítulo presentamos los *patrones de eventos*. Empezaremos por describir informalmente la notación gráfica de los patrones y después formalizaremos la sintaxis y semántica de los mismos.

La presentación estará dividida en tres partes: primero introduciremos los conceptos básicos referentes a patrones de eventos y definiremos formalmente los patrones de eventos *no temporizados*. En segundo lugar, presentaremos una versión temporizada de los patrones básicos. Finalmente, extenderemos los patrones temporizados para soportar los conceptos de *principio* y *final* de ejecución de los cuales hablaremos más adelante. Esta presentación incremental tiene por objetivo la simplificación de las definiciones y demostraciones. Sin embargo, en los próximos capítulos cuando nos refiramos a "patrones de eventos" nos estaremos refiriendo a la versión que incluye los conceptos básicos y todas las extensiones.

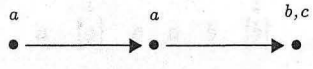
3.1 Patrones básicos

Los patrones básicos de eventos son grafos dirigidos y acíclicos de *puntos*. Los puntos representan la ocurrencia de algún evento y deben estar etiquetados por uno o más símbolos de un alfabeto Σ que representa a los eventos posibles en el sistema. Las flechas que unen los puntos representan las restricciones de *causalidad* entre los puntos (en el sentido de "debe ocurrir antes que").

Por ejemplo:



representa una ocurrencia del evento d, precedida por una ocurrencia del evento c y ésta última a su vez precedida por las ocurrencias de a y b. Dado que a y b no están unidos por ninguna flecha, dichos eventos pueden ocurrir en forma concurrente o secuencial pero sin importar el orden.
Cuando un punto está etiquetado por más de un evento, el punto puede representar la ocurrencia de cualquiera de esos eventos. Dos puntos etiquetados con el mismo evento corresponden a dos ocurrencias distintas del mismo evento. Por ejemplo:



corresponde a dos ocurrencias del evento a en la misma ejecución, seguidas por una ocurrencia de b o de c.
Las flechas simples, como las usadas hasta el momento, permiten expresar que *alguna* ocurrencia de un determinado evento debe estar seguida de *alguna* ocurrencia de otro evento. El patrón:

representa *alguna* ocurrencia de a seguida de *alguna* ocurrencia de b. Si se quiere forzar a que la ocurrencia de b sea la **primera** después de alguna ocurrencia de a, entonces se puede agregar una marca de *primera ocurrencia* en el extremo más próximo a la cabeza de la flecha. El patrón:

representa *alguna* ocurrencia de a seguida por la *primera* ocurrencia de b. Si se quiere restringir aún más el patrón forzando a que la ocurrencia de a que precede a b sea la **última**, entonces se puede agregar una marca de *última ocurrencia* en el extremo más próximo a la cola de la flecha. El patrón:

representa *alguna* ocurrencia de a seguida por la *primera* ocurrencia de b, de forma tal que entre ambos momentos de la ejecución no haya ninguna otra ocurrencia de a. Llamaremos *marcas de consecutividad* a las marcas de *primera ocurrencia* y *última ocurrencia*. Cuando sea necesario identificar unívocamente a alguno de los puntos de un patrón, se les podrá asociar un *nombre* o *identificador* usando el operador $\triangle$:

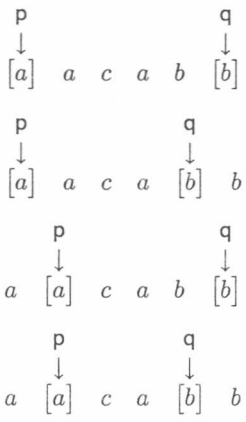
identifica al punto de la izquierda con el nombre 'p' y al punto de la derecha con el nombre 'q'.

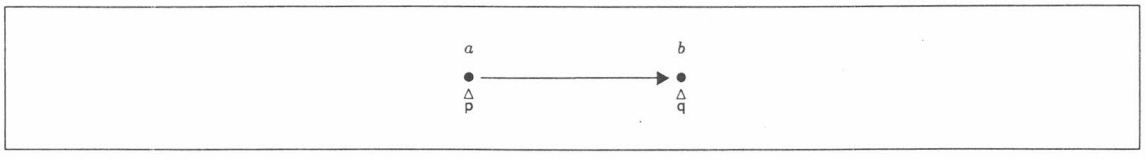
Los patrones se interpretan sobre una ejecución asociando cada punto del patrón con una *posición* de la ejecución. En el caso de los patrones básicos, las ejecuciones estarán modeladas como secuencias de eventos. Cuando asociemos un punto a una posición de la ejecución diremos que estamos *marcando* dicha posición. Los puntos sólo puede ser asociados con posiciones que contengan uno de los eventos que los etiquetan. Al marcar los puntos sobre la ejecución se deberá respetar la precedencia de los puntos, así como también las restricciones asociadas a los distintos pares de puntos del patrón. Diremos que una ejecución *satisface* un patrón si existe al menos una manera de marcar todos los puntos del patrón sobre la secuencia. En general, cuando una ejecución satisfaga un patrón, existirán muchas formas distintas de marcar los puntos sobre la ejecución. A cada una de estas formas la llamaremos *matching*.

Por ejemplo, dada la ejecución:

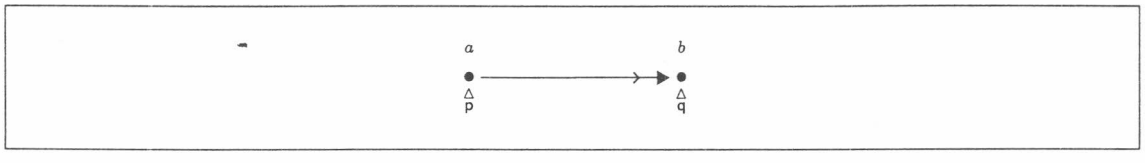
$$\sigma: \quad a \; a \; c \; a \; b \; b$$

existen 6 matchings distintos entre Patrón 3.1 y σ:

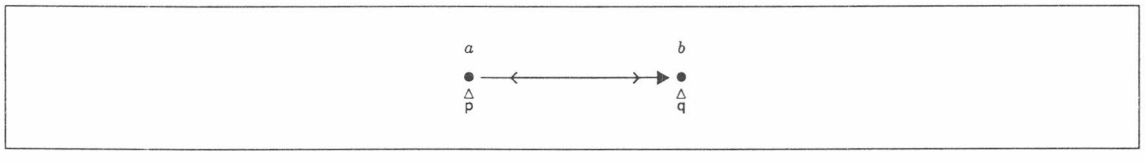




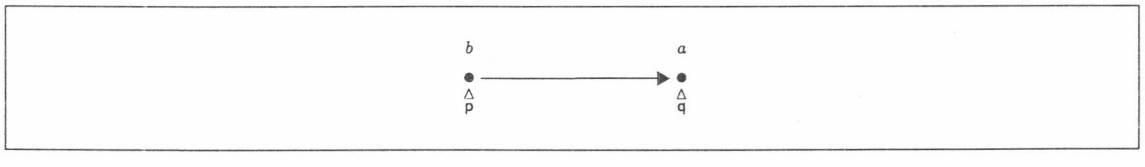
Patrón 3.1: Evento a seguido por el evento b



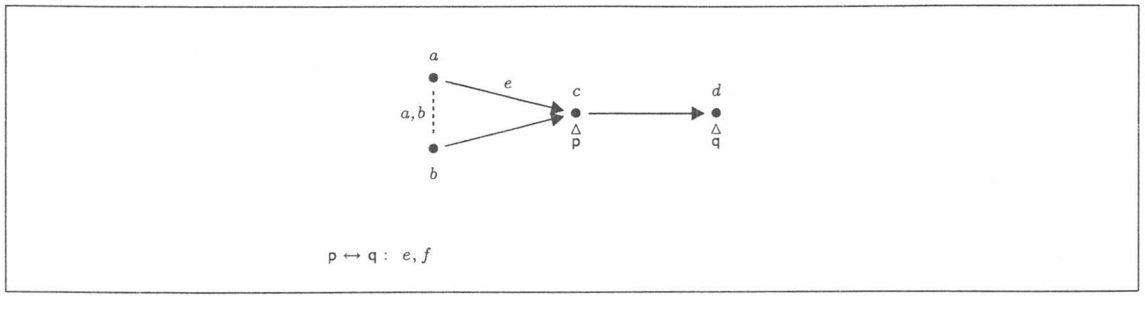
Patrón 3.2: Evento a seguido por el primer evento b



Patrón 3.3: Último evento a antes del primer evento b



Patrón 3.4: Evento b seguido por el evento a



Patrón 3.5: Ejemplos de las distintas formas de escribir restricciones

$$\begin{array}{cccccccc} & & & & \mathsf{p} & & \mathsf{q} & \\ & & & & \downarrow & & \downarrow & \\ a & a & c & [a] & & b & [b] & \end{array}$$

$$\begin{array}{cccccccc} & & & & \mathsf{p} & \mathsf{q} & & \\ & & & & \downarrow & \downarrow & & \\ a & a & c & [a] & & [b] & & b \end{array}$$

Por otro lado, para el Patrón 3.2, más restrictivo que el anterior, sólo existen 3 matchings:

$$\begin{array}{cccccccc} \mathsf{p} & & & & \mathsf{q} & & & \\ \downarrow & & & & \downarrow & & & \\ [a] & a & c & a & [b] & & b & \end{array}$$

$$\begin{array}{cccccccc} & \mathsf{p} & & & \mathsf{q} & & & \\ & \downarrow & & & \downarrow & & & \\ a & [a] & c & a & [b] & & b & \end{array}$$

$$\begin{array}{cccccccc} & & & & \mathsf{p} & \mathsf{q} & & \\ & & & & \downarrow & \downarrow & & \\ a & a & c & [a] & & [b] & & b \end{array}$$

Para el Patrón 3.3 sólo existe un matching:

$$\begin{array}{cccccccc} & & & & \mathsf{p} & \mathsf{q} & & \\ & & & & \downarrow & \downarrow & & \\ a & a & c & [a] & & [b] & & b \end{array}$$

Y para el Patrón 3.4 no existe ninguno.

Los patrones de eventos permiten asociar *restricciones* a cada par de puntos del patrón. Al marcar un par de puntos de un patrón sobre una ejecución, queda determinado unívocamente un *segmento* de ejecución o *subejecución*. Es sobre dicho segmento sobre el cual se interpretan las restricciones asociadas al par. Los patrones básicos admiten un único tipo de restricciones: las restricciones de *eventos*. Estas restricciones limitan los eventos que pueden ocurrir en el tramo de ejecución correspondiente. Las restricciones pueden anotarse directamente sobre los ejes del patrón o pueden documentarse usando los identificadores de puntos. Si el diagrama no resulta muy confuso, es posible anotar una restricción entre eventos no relacionados causalmente uniéndolos con una línea punteada y asociando a esa línea la restricción. Por ejemplo, el Patrón 3.5 muestra las tres formas de escribir las restricciones de eventos. Entre a y c se prohibe el evento e; entre a y b se prohiben otras ocurrencias de a y b y entre c y d se prohiben los eventos e y f, tal como indica la restricción documentada debajo del patrón.

Los patrones 3.2 y 3.3 muestran casos particulares de restricciones de eventos donde los eventos *prohibidos* en la subejecución son los asociados a los extremos de las flechas. Es decir, las marcas de consecutividad son abreviaturas de restricciones de eventos. La Figura 3.1 muestra la relación entre las marcas de consecutividad y la versión equivalente escrita usando restricciones de eventos.

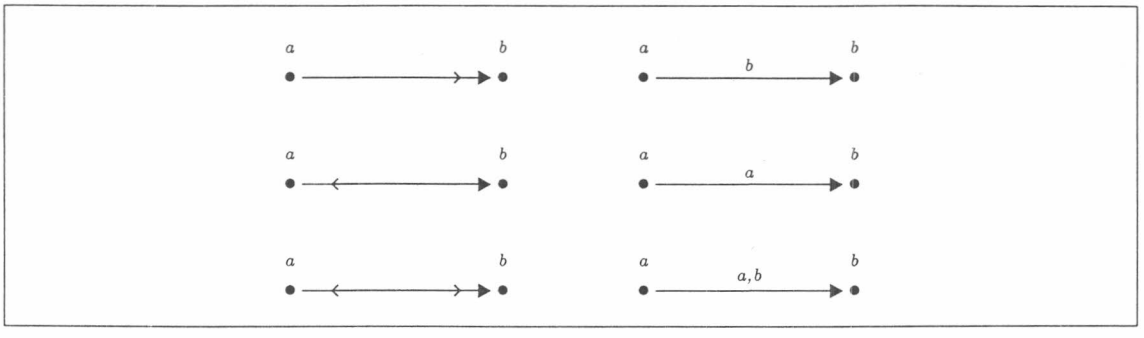


Figura 3.1: Relación entre las marcas de consecutividad (columna izquierda) y las restricciones de eventos (columna derecha)

3.1.1 Sintaxis formal

Definición 3.1.1. Patrones de eventos básicos.
Un *patrón de eventos básico* $\mathcal{P}$ es una tupla $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ donde:

- Σ es un conjunto finito de eventos
- $P = E \uplus I$ es un conjunto finito de *puntos*, particionado en un conjunto $E = \{\mathsf{e}_1, \mathsf{e}_2, \ldots, \mathsf{e}_n\}$ y un conjunto $I = \varnothing$[1]
- $\ell : E \longrightarrow 2^{\Sigma}$ es una función que *etiqueta* los puntos de E asignándoles un conjunto no vacío de eventos.
- $\rightarrow$ es una relación de *causalidad* incluida en $P \times P$.

  La clausura transitiva de $\rightarrow$ debe definir una relación de orden parcial $\prec_{\mathcal{P}}$ entre los elementos de P. Es decir,

  $$\prec_{\mathcal{P}} = (\rightarrow)^{+}$$

  Dados dos puntos $\mathsf{x}, \mathsf{y} \in P$, diremos que x y y están *causalmente relacionados* (con respecto a un patrón de eventos $\mathcal{P}$) si $\mathsf{x} \prec_{\mathcal{P}} \mathsf{y}$.

  Cuando quede claro por contexto a qué patrón nos estamos refiriendo, usaremos $\prec$ en lugar de $\prec_{\mathcal{P}}$.

- γ es una función:

  $$\gamma : P \times P \longrightarrow 2^{\Sigma}$$

  tal que para todo par x, y, $\gamma(\mathsf{x}, \mathsf{y}) = \gamma(\mathsf{y}, \mathsf{x})$, que asocia a cada par de puntos una restricción de eventos.

El conjunto P representa el conjunto de puntos de un patrón básico. Según la definición, este conjunto está particionado en E e I. Los puntos que mostramos en la representación gráfica de los patrones básicos pertenecen exclusivamente a la clase E y decimos que representan *ocurrencias de eventos*. Los puntos de la clase I no tienen representación gráfica para los patrones básicos y de hecho tampoco les asignaremos ningún significado: este tipo de puntos tendrán sentido cuando presentemos los patrones temporizados y se incluyen en la sintaxis formal de los patrones básicos sólo para facilitar las definiciones y demostraciones posteriores. Dado que no tienen representación gráfica ni semántica para los patrones básicos, asumiremos que para todo patrón básico, $I = \varnothing$.

[1] Ver aclaración a continuación.

Ejemplo. El Patrón 3.5 corresponde a la siguiente tupla:
$\mathcal{P} = \langle \Sigma, E \cup I, \ell, \rightarrow, \gamma \rangle$, donde:

$$\Sigma = \{a, b, c, d, e, f\}$$
$$E = \{\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}\}$$
$$I = \varnothing$$
$$\ell = \{\mathsf{p} \mapsto \{c\}, \mathsf{q} \mapsto \{d\}, \mathsf{r} \mapsto \{a\}, \mathsf{s} \mapsto \{b\}\}$$
$$\rightarrow = \{\mathsf{p} \rightarrow \mathsf{q}, \mathsf{r} \rightarrow \mathsf{p}, \mathsf{s} \rightarrow \mathsf{p}\}$$
$$\gamma = \left\{ \begin{array}{llll} (\mathsf{p},\mathsf{p}) \mapsto \varnothing & (\mathsf{p},\mathsf{q}) \mapsto \{e,f\} & (\mathsf{p},\mathsf{r}) \mapsto \{e\} & (\mathsf{p},\mathsf{s}) \mapsto \varnothing \\ (\mathsf{q},\mathsf{p}) \mapsto \{e,f\} & (\mathsf{q},\mathsf{q}) \mapsto \varnothing & (\mathsf{q},\mathsf{r}) \mapsto \varnothing & (\mathsf{q},\mathsf{s}) \mapsto \varnothing \\ (\mathsf{r},\mathsf{p}) \mapsto \{e\} & (\mathsf{r},\mathsf{q}) \mapsto \varnothing & (\mathsf{r},\mathsf{r}) \mapsto \varnothing & (\mathsf{r},\mathsf{s}) \mapsto \{a,b\} \\ (\mathsf{s},\mathsf{p}) \mapsto \varnothing & (\mathsf{s},\mathsf{q}) \mapsto \varnothing & (\mathsf{s},\mathsf{r}) \mapsto \{a,b\} & (\mathsf{s},\mathsf{s}) \mapsto \varnothing \end{array} \right\}$$

Se puede ver que p y q conservan el nombre explícitamente mencionado en el patrón. Por otro lado, aunque no es necesario identificar todos los puntos en la versión gráfica de los patrones, sí lo es en la versión textual. Como dijimos antes, todos los puntos de la representación gráfica del patrón corresponden a puntos en E e I es vacío. Cuando no se explicite una restricción de eventos entre un par de puntos se asumirá que no existe tal y por lo tanto el conjunto de eventos "prohibidos" es vacío.

3.1.2 Semántica

Como vimos anteriormente, la semántica de los patrones básicos se define sobre una ejecución y utilizando el concepto de *matching*. Intuitivamente, un matching representa una forma de asociar los puntos de un patrón a posiciones de una ejecución de forma tal de respetar el orden parcial entre los puntos, la función de etiquetación y las restricciones de eventos. Además, cada punto debe ser asociado a una posición distinta de la ejecución.

Definición 3.1.2. Matching básico.
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$, una ejecución ς sobre Σ y un mapeo[2] $\hat{\cdot} : P \mapsto \Pi_\varsigma$, decimos que $\hat{\cdot}$ es un *matching básico* entre ς y $\mathcal{P}$ sii verifica las siguientes *condiciones de validez*:

$$\begin{array}{lll} \mathbf{M1} & \forall \mathsf{x} \in E, \quad \varsigma_{\hat{\mathsf{x}}} \in \ell(\mathsf{x}) & \\ \mathbf{M2} & \forall \mathsf{x},\mathsf{y} \in P, \quad \mathsf{x} \prec_{\mathcal{P}} \mathsf{y} \Rightarrow \hat{\mathsf{x}} < \hat{\mathsf{y}} & (\hat{\cdot} \textit{ monótono creciente}) \\ \mathbf{M3} & \forall \mathsf{x},\mathsf{y} \in P, \quad \hat{\mathsf{x}} < \hat{\mathsf{y}} \Rightarrow \varsigma_{(\hat{\mathsf{x}},\hat{\mathsf{y}})} \cap \gamma(\mathsf{x},\mathsf{y}) = \varnothing & \end{array}$$

La condición **M1** exige que los puntos sean marcados respetando la función de etiquetación ℓ. Como dijimos antes, los puntos de E corresponden a ocurrencias de *eventos*. La condición **M2** pide que se respete el orden de causalidad o de precedencia de los puntos, que en caso de los patrones básicos corresponden únicamente a ocurrencias de eventos. Dado que la condición está expresada como una implicación, si dos puntos no están causalmente relacionados, entonces no habrá ninguna restricción en cuanto al orden en que deben ser marcados. Finalmente, la condición **M3** pide que se cumplan las restricciones de eventos asociadas a todos los pares de puntos del patrón. Notar que la condición **M3** no incluye el caso de un punto x con sí mismo. Esto es porque sólo tienen sentido las restricción de eventos definidas entre puntos distintos. Por otro lado, dado que la función γ debe ser simétrica ($\gamma(\mathsf{x},\mathsf{y}) = \gamma(\mathsf{y},\mathsf{x})$) y $\hat{\cdot}$ es una función inyectiva, la condición **M3** alcanza para garantizar que se cumplan todas las restricciones de eventos.

Definición 3.1.3. Satisfacción de patrones básicos.
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$, una ejecución ς sobre Σ *verifica* o *satisface* $\mathcal{P}$ (notado $\varsigma \vDash \mathcal{P}$) sii existe al menos un matching básico entre ς y $\mathcal{P}$.

Como ya vimos anteriormente, si una ejecución satisface un patrón, en general hay más de un matching entre los puntos del patron y la ejecución.

[2] Diremos que una función $f : A \mapsto B$ es un *mapeo* si es una función **total** e **inyectiva**.

Lenguaje de un patrón básico

Definición 3.1.4. Lenguaje de un patrón básico
Dado un patrón básico $\mathcal{P}$, llamaremos *lenguaje de* $\mathcal{P}$ al conjunto

$$\mathcal{L}(\mathcal{P}) = \{\varsigma \mid \varsigma \vDash \mathcal{P} \text{ y } |\varsigma| = \infty\}$$

Como adelantamos cuando introdujimos las ejecuciones, preferiremos las ejecuciones infinitas dado que permiten modelar sistemas cuyas corridas nunca terminan (sistemas *reactivos*) y permiten simular el comportamiento de sistemas cuyas corridas sí terminan, extendiendo las ejecuciones con infinitos λ's al final. Si una ejecución satisface un patrón, esa ejecución o alguna equivalente pertenecerá al lenguaje del patrón.

Muchas veces, un patrón de eventos hará referencia explícita a un número relativamente pequeño de eventos de un sistema. Es decir, el alfabeto de eventos del patrón podría ser un subconjunto propio del alfabeto de eventos del sistema. Sin embargo, desde el punto de vista formal, para decidir si una ejecución satisface un patrón, ambos deben estar definidos sobre el mismo alfabeto. Ahora bien, resulta más o menos claro que si un patrón no menciona a un determinado evento (es decir, no figura como etiqueta de ningún punto ni como restricción de eventos entre ningún par de puntos), entonces ese evento no es relevante para el patrón. Da lo mismo que haya ocurrido o que no lo haya hecho. Quiere decir que podríamos filtrar todos los eventos "desconocidos" para el patrón reemplazándolos por λ's y eso no afectaría la satisfacción del patrón. Usando esta estrategia definimos el concepto de *lenguaje ampliado* de un patrón de eventos que permite trabajar con un alfabeto más rico que el del patrón.

Definición 3.1.5. Lenguaje ampliado.
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$, y un conjunto Σ' tal que $\Sigma \subseteq \Sigma'$, definimos:

$$\mathcal{L}_{\Sigma'}(\mathcal{P}) = \{\ \varsigma \text{ sobre } \Sigma' \mid \varsigma\lceil_{\Sigma} \vDash \mathcal{P}\ \}$$

Propiedades de la satisfacción de patrones básicos

A continuación enunciamos dos propiedades de la satisfacción de patrones que resultarán de suma importancia para la generación de autómatas temporizados que acepten el lenguaje de un patrón básico.

La primera propiedad dice que si una ejecución finita satisface un patrón básico, entonces cualquier extensión de la misma, sea finita o infinita, también lo hace. Esto quiere decir que si un prefijo finito de una ejecución satisface un patrón básico, es condición *suficiente* para generalizar el resultado a toda la ejecución.

La segunda propiedad es, de alguna manera, complementaria con la anterior. Esta propiedad dice que si una ejecución (finita o infinita) satisface un patrón, entonces existe un prefijo finito de la misma que también lo hace. Es decir, que un prefijo finito de una ejecución satisfaga un patrón básico es condición *necesaria* para que la ejecución completa lo haga.

En ambos casos, las demostraciones se basan en el hecho de que los patrones predican sobre finitos puntos de una ejecución y sobre los segmentos de ejecución entre estos. Quiere decir que siempre va a existir un "último" punto (es decir, un punto que está marcado en una posición mayor a las de todo el resto) y los eventos más allá de ese punto no serán relevantes para el patrón. Es por eso que la decisión de la satisfacción de un patrón básico siempre se basa en prefijos finitos de la ejecución. Las demostraciones completas están en el Apéndice A, página 93.

Propiedad 3.1. *Clausura por extensiones. Dado un patrón básico* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ *y una ejecución finita* ς *sobre* Σ*, si* $\varsigma \vDash \mathcal{P}$ *entonces para cualquier ejecución* ς' *sobre* Σ*,* $\varsigma\varsigma' \vDash \mathcal{P}$*.*

Propiedad 3.2. *Satisfacción finita. Dado un patrón básico* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ *y una ejecución* ς *sobre* Σ*, si* $\varsigma \vDash \mathcal{P}$ *entonces existe una posición* $i \in \Pi_\varsigma$ *tal que* $\varsigma_{\lceil i \rceil} \vDash \mathcal{P}$*.*

3.2 Extensión: tiempo

La primera extensión sobre los patrones básicos que presentaremos agrega el concepto de *tiempo* a los patrones y a las ejecuciones. Los *patrones temporizados* nos permitirán definir restricciones temporales explícitas entre puntos de un patrón y no sólo relaciones implícitas dadas por el orden en que deben ocurrir los eventos. Para poder delimitar intervalos de tiempo arbitrarios, extenderemos el conjunto de puntos para agregar a los puntos que representan *eventos* otros puntos que representen *instantes* en el tiempo.

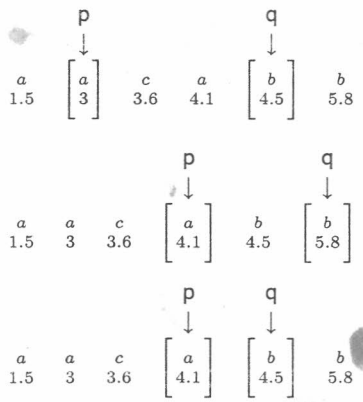
Básicamente, un patrón temporizado tiene dos tipos de puntos: los puntos "llenos" que representan la ocurrencia de un evento (corresponden a los puntos de los patrones básicos) y los puntos "huecos" que representan instantes en el tiempo entre dos eventos. Además de las restricciones de eventos que permitían los patrones básicos, los patrones temporizados permiten otro tipo de restricciones: las *restricciones temporales*. Estas restricciones se definen asociando un intervalo de números reales no negativos con extremos enteros (ej: $[0,6)$, $(8,\infty)$, $[7,9]$) o como el complemento de uno de estos intervalos (ej: $\neg[0,6)$, $\neg(8,\infty)$, $\neg[7,9]$) a un par de puntos. Intuitivamente, la separación en el tiempo entre los dos puntos (al ser marcados sobre una ejecución temporizada) debe pertenecer al intervalo especificado. En los casos donde sólo se quiere expresar una cota superior o inferior, se pueden usar las abreviaturas $\geq n$ y $\leq m$, respectivamente. Como en el caso de las restricciones de eventos, las restricciones temporales pueden ser anotadas directamente sobre el patrón o documentadas usando los identificadores de puntos.

Los patrones temporizados se interpretan sobre ejecuciones temporizadas (definición 2.1.5) y se extiende el concepto de *matching básico* para incluir a los puntos huecos y a las restricciones temporales. Los puntos llenos se marcan sobre la ejecución de la misma manera que los puntos de un patrón básico. Los puntos huecos sólo pueden ser mapeados sobre posiciones que no representen ocurrencias de eventos, es decir, sobre posiciones de la ejecución que contengan λ. La duración de las subejecuciones determinadas por todo par de puntos deben pertenecer al intervalo asociado como restricción a ese par. Si no hay ningún intervalo asociado, se asume $[0,\infty)$, es decir, cualquier duración.

Dada la ejecución temporizada:

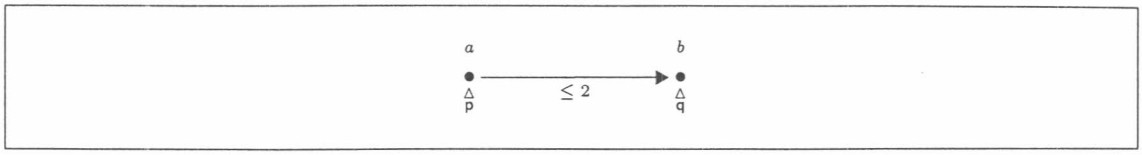
$$\sigma = \left\langle \begin{array}{ccccccc} a & a & c & a & b & b \\ 1.5 & 3 & 3.6 & 4.1 & 4.5 & 5.8 \end{array} \right\rangle$$

El Patrón 3.6 determina los siguientes matchings:

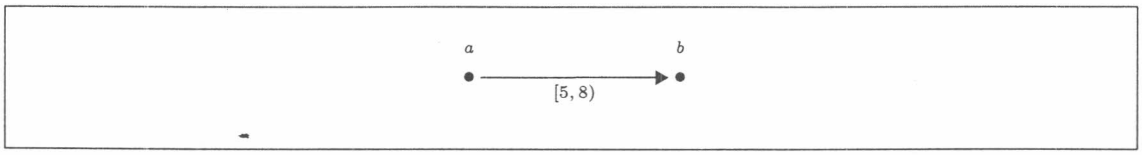


debido a que son los únicos en las cuales la ocurrencia de a asociada p y la ocurrencia de b asociada q están separadas por menos de 2 unidades de tiempo.
Por otro lado, para el Patrón 3.7 no existe ningún matching sobre esa ejecución dado que la máxima separación entre ocurrencias de a y b es de 4.3 u.t.

Como en el caso de las restricciones de los patrones básicos, es posible asociar restricciones temporales entre puntos no relacionados causalmente. Las restricciones se aplican sobre el segmento de ejecución correspondiente, sin importar en qué orden se marquen los puntos sobre la ejecución. Para asociar restricciones a

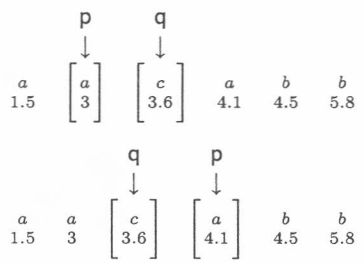


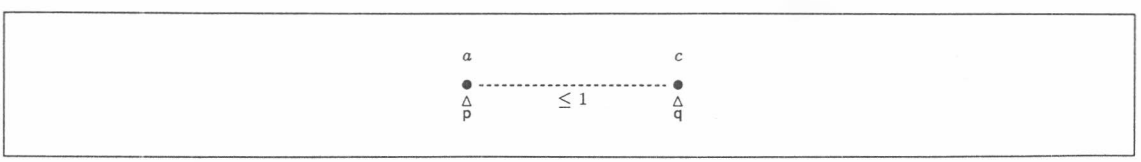
Patrón 3.6: Evento a separado del evento b por menos de 2 u.t.



Patrón 3.7: Evento a separado del evento b por mas de 5 u.t. y por menos de 8 u.t.

eventos no relacionados, se puede dibujar una línea punteada entre ambos y decorar esa línea con las restricciones. Por ejemplo, el Patrón 3.8 indica que la ocurrencia de a y de c (sin importar cual ocurra primero y cual después) no deben estar separadas por más de 1 unidad de tiempo. Este patrón, interpretado sobre nuestra ejecución ejemplo admite los siguientes matchings:





Patrón 3.8: Ejemplo de restricciones temporales entre puntos no relacionados causalmente

3.2.1 Sintaxis formal

Una *restricción temporal* es una fórmula φ con la siguiente forma:

$$\varphi ::= \theta \mid \neg\theta$$

donde $\theta \in \mathfrak{I}_{\mathbf{N}}$ es un intervalo de números reales positivos con extremos enteros (definición 2.4.1). Son ejemplos de restricciones temporales: $[0,6)$, $(8,\infty)$, $[7,9]$, $\neg[0,6)$, $\neg(8,\infty)$, $\neg[7,9]$.

Llamamos Φ al conjunto de todas las restricciones temporales construidas de esta forma.

Dado $t \in \Re^+$ y una restricción temporal, definimos inductivamente la relación $\models$ como:

$$\begin{array}{lll} t \models \theta & \text{sii} & t \in \theta \\ t \models \neg\theta & \text{sii} & t \not\models \theta \end{array}$$

donde $\theta \in \mathfrak{I}_{\mathbf{N}}$. Diremos que t *satisface* una restricción temporal φ sii $t \models \varphi$. Por ejemplo: dado el intervalo $\theta = [7,9]$; $4.5 \models \theta$, $7 \models \theta$, $9.1 \not\models \theta$, y $3 \models \neg\theta$ y $100 \models \neg\theta$.

Definición 3.2.1. Patrones de eventos temporizados.
Un *patrón de eventos temporizado* es una tupla $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ donde:

- Σ es un conjunto finito de eventos
- P es un conjunto finito de puntos particionado en los conjuntos $E = \{\mathsf{e}_1, \mathsf{e}_2, \ldots, \mathsf{e}_n\}$ e $I = \{\mathsf{i}_1, \mathsf{i}_2, \ldots, \mathsf{i}_m\}$
- $\ell : E \longrightarrow 2^{\Sigma}$ es una función que *etiqueta* los puntos de E asignándoles un conjunto no vacío de eventos.
- $\rightarrow$ es una relación de *causalidad* incluida en $P \times P$.
  La clausura transitiva de $\rightarrow$ debe definir una relación de orden parcial $\prec_{\mathcal{P}}$ entre los elementos de P. Es decir,

$$\prec_{\mathcal{P}} = (\rightarrow)^+$$

- γ es una función:

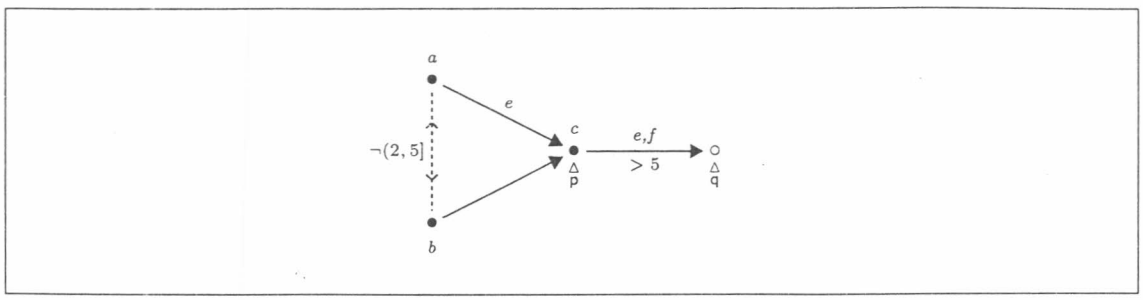
$$\gamma : P \times P \longrightarrow 2^{\Sigma}$$

  tal que para todo par x, y, $\gamma(\mathsf{x},\mathsf{y}) = \gamma(\mathsf{y},\mathsf{x})$, que asocia a cada par de puntos una restricción de eventos.
- δ es una función que a cada par de puntos le asocia una restricción temporal.
  Formalmente, δ es una función $\delta : P \times P \longrightarrow \Phi$, donde para todo x, y, $\delta(\mathsf{x},\mathsf{y}) = \delta(\mathsf{y},\mathsf{x})$.

Como podemos ver, las diferencia entre patrones básicos y temporizados son que el conjunto I es no vacío y el agregado de la función δ. Esta última asocia a pares de puntos una restricción temporal. El conjunto de puntos E corresponde a los puntos "llenos" y el conjunto I corresponde a los puntos "huecos".

Ejemplo. El Patrón 3.9 corresponde a la siguiente tupla: $\mathcal{P} = \langle \Sigma, E \cup I, \ell, \rightarrow, \gamma, \delta \rangle$, donde:

$$\begin{array}{l}
\Sigma = \{a, b, c, e, f\} \\
E = \{\mathsf{p}, \mathsf{r}, \mathsf{s}\} \\
I = \{\mathsf{q}\} \\
\ell = \{\mathsf{p} \mapsto \{c\}, \mathsf{r} \mapsto \{a\}, \mathsf{s} \mapsto \{b\}\} \\
\rightarrow = \{\mathsf{p} \rightarrow \mathsf{q}, \mathsf{r} \rightarrow \mathsf{p}, \mathsf{s} \rightarrow \mathsf{p}\} \\
\gamma = \left\{ \begin{array}{llll} (\mathsf{p},\mathsf{p}) \mapsto \varnothing & (\mathsf{p},\mathsf{q}) \mapsto \{e,f\} & (\mathsf{p},\mathsf{r}) \mapsto \{e\} & (\mathsf{p},\mathsf{s}) \mapsto \varnothing \\ (\mathsf{q},\mathsf{p}) \mapsto \{e,f\} & (\mathsf{q},\mathsf{q}) \mapsto \varnothing & (\mathsf{q},\mathsf{r}) \mapsto \varnothing & (\mathsf{q},\mathsf{s}) \mapsto \varnothing \\ (\mathsf{r},\mathsf{p}) \mapsto \{e\} & (\mathsf{r},\mathsf{q}) \mapsto \varnothing & (\mathsf{r},\mathsf{r}) \mapsto \varnothing & (\mathsf{r},\mathsf{s}) \mapsto \{a,b\} \\ (\mathsf{s},\mathsf{p}) \mapsto \varnothing & (\mathsf{s},\mathsf{q}) \mapsto \varnothing & (\mathsf{s},\mathsf{r}) \mapsto \{a,b\} & (\mathsf{s},\mathsf{s}) \mapsto \varnothing \end{array} \right\} \\
\delta = \left\{ \begin{array}{llll} (\mathsf{p},\mathsf{p}) \mapsto [0,\infty) & (\mathsf{p},\mathsf{q}) \mapsto (5,\infty) & (\mathsf{p},\mathsf{r}) \mapsto [0,\infty) & (\mathsf{p},\mathsf{s}) \mapsto [0,\infty) \\ (\mathsf{q},\mathsf{p}) \mapsto (5,\infty) & (\mathsf{q},\mathsf{q}) \mapsto [0,\infty) & (\mathsf{q},\mathsf{r}) \mapsto [0,\infty) & (\mathsf{q},\mathsf{s}) \mapsto [0,\infty) \\ (\mathsf{r},\mathsf{p}) \mapsto [0,\infty) & (\mathsf{r},\mathsf{q}) \mapsto [0,\infty) & (\mathsf{r},\mathsf{r}) \mapsto [0,\infty) & (\mathsf{r},\mathsf{s}) \mapsto \neg(2,5] \\ (\mathsf{s},\mathsf{p}) \mapsto [0,\infty) & (\mathsf{s},\mathsf{q}) \mapsto [0,\infty) & (\mathsf{s},\mathsf{r}) \mapsto \neg(2,5] & (\mathsf{s},\mathsf{s}) \mapsto [0,\infty) \end{array} \right\}
\end{array}$$



Patrón 3.9: Ejemplo de patrón temporizado

Como mencionamos antes, los patrones básicos son un caso particular de patrones temporizados. Todo patrón básico de la forma $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$, es equivalente al patrón temporizado $\mathcal{P}' = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$, donde $\delta(\mathsf{x}, \mathsf{y}) = [0, \infty)$ para todo par de puntos $\mathsf{x}, \mathsf{y} \in P$.

3.2.2 Semántica

La semántica de los patrones temporizados se define en función de ejecuciones temporizadas. Las condiciones para que un mapeo sea un *matching* en el caso temporizado incluyen a las de los patrones básicos más dos nuevas.

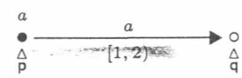
Definición 3.2.2. Matching temporizado.
Dado un patrón de eventos temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$, una ejecución temporizada $\sigma = \langle \varsigma, \tau \rangle$ y un mapeo $\hat{\cdot} : P \mapsto \Pi_\sigma$, decimos que $\hat{\cdot}$ es un *matching temporizado* entre σ y $\mathcal{P}$ sii verifica las condiciones **M1-M3** además:

$$\begin{array}{ll} \textbf{MT1} & \forall \mathsf{x} \in I, \quad \varsigma_{\hat{\mathsf{x}}} = \lambda \\ \textbf{MT2} & \forall \mathsf{x}, \mathsf{y} \in P, \quad \hat{\mathsf{x}} < \hat{\mathsf{y}} \;\Rightarrow\; \Delta(\tau_{[\hat{\mathsf{x}}, \hat{\mathsf{y}}]}) \vDash \delta(\mathsf{x}, \mathsf{y}) \end{array}$$

La condición **MT1** indica que los puntos huecos sólo pueden mapearse en posiciones de la ejecución que correspondan a instantes en el tiempo: es decir, a posiciones que contengan λ. La condición **MT2** exige que se cumplan todas las restricciones temporales entre todo par de puntos del patrón. Como era de esperarse, no tiene ningún significado una restricción temporal definida entre un punto x y sí mismo. Además, dado que δ es simétrica ($\delta(\mathsf{x}, \mathsf{y}) = \delta(\mathsf{y}, \mathsf{x})$) y $\hat{\cdot}$ es inyectiva, la condición **MT2** alcanza para garantizar que valen todas las restricciones entre todos los pares de puntos.

Definición 3.2.3. Satisfacción de patrones de eventos temporizados.
Una ejecución temporizada σ *satisface* un patrón de eventos temporizado $\mathcal{P}$ (notado como $\sigma \vDash \mathcal{P}$), sii existe un matching temporizado entre σ y $\mathcal{P}$.

Como antes, si una ejecución temporizada satisface un patrón, en general existirá más de un matching temporizado entre el patrón y la ejecución.
En este caso, a diferencia de lo que ocurría con las ejecuciones de los patrones básicos, dos ejecuciones temporizadas *equivalentes* podrían diferir en cuanto a la satisfacción de un patrón temporizado. Observemos el siguiente ejemplo.



Patrón 3.10: Ejemplo de patrón temporizado con instantes

Ejemplo. Sea $\mathcal{P}$ el Patrón 3.10, que pide que desde alguna ocurrencia de a haya intervalo de por lo menos 1 u.t. y a lo sumo 2 u.t. de duración sin más ocurrencias de a. Dadas las ejecuciones:

$$\sigma_1 = \left\langle \begin{matrix} a & a & c & a & b & b \\ 1.5 & 3 & 3.6 & 4.1 & 4.5 & 5.8 \end{matrix} \right\rangle$$

$$\sigma_2 = \left\langle \begin{matrix} \lambda & a & \lambda & a & c & \lambda & a & b & \lambda & b & \lambda \\ 0.2 & 2 & 3 & 3 & 3.6 & 4 & 4.1 & 4.5 & 5.7 & 5.8 & 6.099 \end{matrix} \right\rangle$$

claramente $\sigma_1 \equiv \sigma_2$ dado que tienen los mismo eventos, ocurridos en el mismo instante de tiempo. Sin embargo $\sigma_1 \not\models \mathcal{P}$ porque el punto q no puede ser asociado con ninguna posición de la ejecución y $\mathcal{P}$ admite al menos 4 matchings sobre σ_2:

	p ↓	q ↓								
λ	[a]	[λ]	a	c	λ	a	b	λ	b	λ
0.2	[2]	[3]	3	3.6	4	4.1	4.5	5.7	5.8	6.099

			p ↓		q ↓					
λ	a	λ	[a]	c	[λ]	a	b	λ	b	λ
0.2	2	3	[3]	3.6	[4]	4.1	4.5	5.7	5.8	6.099

						p ↓		q ↓		
λ	a	λ	a	c	λ	[a]	b	[λ]	b	λ
0.2	2	3	3	3.6	4	[4.1]	4.5	[5.7]	5.8	6.099

						p ↓				q ↓
λ	a	λ	a	c	λ	[a]	b	λ	b	[λ]
0.2	2	3	3	3.6	4	[4.1]	4.5	5.7	5.8	[6.099]

Es fácil ver que si un comportamiento del sistema cumple una propiedad expresada por un patrón de eventos, entonces existe una forma de intercalar λ en cualquier ejecución temporizada que lo represente de forma tal de que exista al menos un matching entre el patrón y la ejecución. Quiere decir que si necesitáramos decidir para un comportamiento puntual si verifica o no una propiedad dada por un patrón temporizado, deberíamos analizar las infinitas ejecuciones equivalentes que representan a dicho comportamiento. Sin embargo, en el contexto de este trabajo sólo nos interesará resolver problemas expresados como "vacuidad" de lenguajes, y por lo tanto nos resulta suficiente la noción de satisfacción a nivel ejecuciones.

Lenguaje de un patrón temporizado

En el caso de los patrones básicos, exigimos que las ejecuciones que formaran parte del lenguaje del patrón fueran infinitas. En el caso de los patrones temporizados vamos a pedir, además, que sean ejecuciones *divergentes* (definición 2.1.7). Esto nos permite modelar el progreso del tiempo más allá de cualquier constante real.

Definición 3.2.4. Lenguaje de un patrón temporizado
Dado un patrón temporizado $\mathcal{P}$, llamaremos *lenguaje de* $\mathcal{P}$ al conjunto

$$\mathcal{L}(\mathcal{P}) = \{\sigma \mid \sigma \vDash \mathcal{P} \text{ y } \sigma \text{ es divergente}\}$$

Adaptamos de forma natural el concepto de *lenguaje ampliado*, presentado para patrones básicos, a patrones temporizados:

Definición 3.2.5. Lenguaje ampliado.
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$, y un conjunto Σ' tal que $\Sigma \subseteq \Sigma'$, definimos:

$$\mathcal{L}_{\Sigma'}(\mathcal{P}) = \{\ \sigma \text{ sobre } \Sigma' \mid \sigma\lceil_{\Sigma} \vDash \mathcal{P}\ \}$$

Propiedades de la satisfacción de patrones temporizados

Las propiedades de la satisfacción de patrones básicos también valen para los patrones temporizados. Quiere decir que siempre alcanza con analizar los prefijos finitos de una ejecución temporizada para decidir si la misma satisface o no un patrón temporizado.

Propiedad 3.3. *Clausura por extensiones. Dado un patrón* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ *y una ejecución finita* σ *sobre* Σ, *si* $\sigma \vDash \mathcal{P}$ *entonces para cualquier ejecución* σ' *sobre* Σ, $\sigma\sigma' \vDash \mathcal{P}$.

Propiedad 3.4. *Satisfacción finita. Dado un patrón temporizado* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ *y una ejecución temporizada* σ *sobre* Σ, *si* $\sigma \vDash \mathcal{P}$ *entonces existe una posición* $i \in \Pi_\sigma$ *tal que* $\sigma_{\text{i}} \vDash \mathcal{P}$.

3.3 Extensión: *principio* y *final*

La segunda y última extensión que presentaremos consiste en permitir mencionar en forma explícita el *comienzo* y el *final* de una ejecución. Sin esta extensión no sería posible expresar propiedades de la forma:

"*Desde el comienzo de la ejecución y hasta la ocurrencia del evento a no ocurre el evento b*"

Se puede hacer referencia explícita al *comienzo* de la ejecución utilizando el punto distinguido:

Por otro lado, se puede hacer referencia explícita al *final* de la ejecución utilizando el punto distinguido:

Tanto el punto inicial como el punto final no son considerados puntos *propios* de una ejecución. Es decir, el punto inicial representa el "momento anterior" a que comience la ejecución y el punto final representa el "momento después" de que termine la ejecución. Dado que trabajaremos mayormente con ejecuciones infinitas, en esos casos el punto final debe ser interpretado como una especie de *límite*. Todos los puntos tienen una relación de precedencia implícita con respecto al punto inicial y al punto final.

Con el agregado del punto inicial y el punto final se pueden expresar cuatro de los *scopes* más comunes según [DAC98]: **global**, **before**, **after** y **between**. La Figura 3.2 muestra una descripción de cada uno de estos

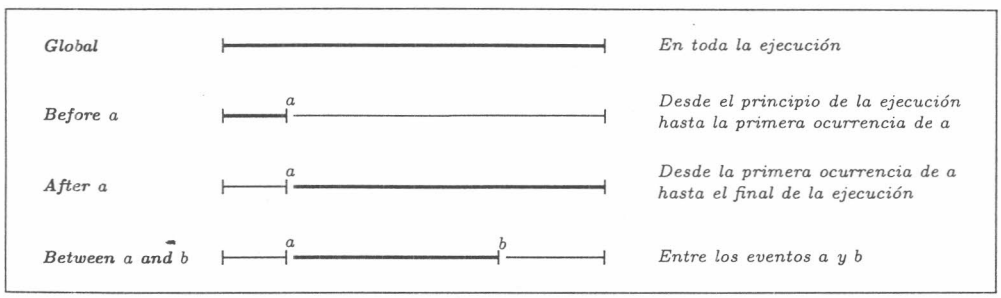


Figura 3.2: Scopes mas comunes

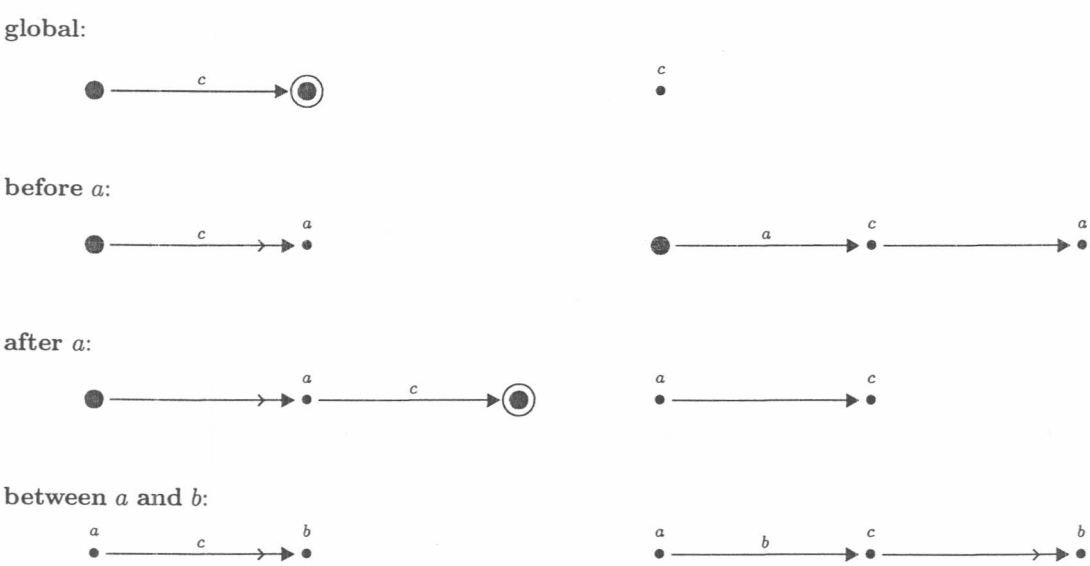
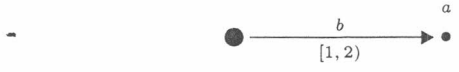


Figura 3.3: Patrones de eventos representando la ausencia (columna de la izquierda) o la ocurrencia (columna de la derecha) de un evento c en cada uno de los distintos scopes.

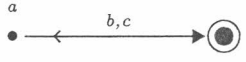
scopes y la Figura 3.3 muestra ocho patrones que expresan la ausencia y la ocurrencia de un evento c en cada uno de los distintos scopes.

Estos nuevos patrones permiten asociar restricciones entre el punto de comienzo y el resto de los puntos y entre cualquier punto y el punto de final. El único caso que no está permitido es definir restricciones temporales entre un punto y el punto final. Esto es así porque en general trabajaremos con ejecuciones temporizadas infinitas y divergentes y en esos casos no tendría sentido acotar la duración de un sufijo de la ejecución.

Se pueden componer todos los conceptos presentados hasta el momento para expresar patrones más complejos como por ejemplo:



indica que debe ocurrir a después de transcurrida 1 unidad de tiempo y antes de que transcurran 2u.t. desde el comienzo de la ejecución y, además, no debe haber ninguna ocurrencia de b antes de dicha a.



indica que a partir de la última ocurrencia de a en la ejecución, no debe haber ninguna ocurrencia de b ni de c.

Como en el caso de los patrones temporizados, la semántica de estos patrones se definirá sobre ejecuciones temporizadas usando una extensión apropiada del concepto de *matching*.

Con esta última extensión completamos la presentación de los *patrones de eventos* y en adelante usaremos ese nombre para referirnos a los patrones básicos temporizados con punto inicial y punto final.

3.3.1 Sintaxis formal

Definición 3.3.1. Patrones de eventos.
Un *patrón de eventos* es una tupla $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ donde:

- Σ es un conjunto finito de eventos
- $P = E \uplus I$ es un conjunto finito de *puntos*, particionado en un conjunto $E = \{e_1, e_2, \ldots, e_n\}$ y un conjunto $I = \{i_1, i_2, \ldots, i_m\}$.
- $\mathbf{0}$ es un punto distinguido que representa el *comienzo* de una ejecución
- ∞ es un punto distinguido que representa el *final* de una ejecución
- $\ell : E \longrightarrow 2^{\Sigma}$ es una función que *etiqueta* los puntos de E asignándoles un conjunto no vacío de eventos incluido en Σ
- $\rightarrow$ es una relación de *causalidad* incluida en $P \cup \{\mathbf{0}, \infty\} \times P \cup \{\mathbf{0}, \infty\}$. Como en el caso de los patrones temporizados, definimos:

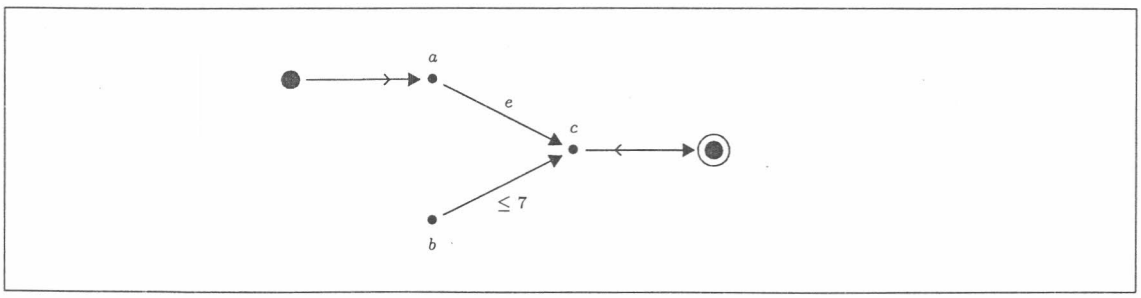
  $$\prec_{\mathcal{P}} = (\rightarrow)^+$$

  $\prec_{\mathcal{P}}$ debe ser un orden parcial sobre los elementos de $P \cup \{\mathbf{0}, \infty\}$ y además, $\mathbf{0}$ debe ser el ínfimo de la relación, es decir:

  $$\forall\, x \in P,\ \mathbf{0} \prec_{\mathcal{P}} x$$

  y ∞ debe ser el supremo de la relación, es decir:

  $$\forall\, x \in P,\ x \prec \infty$$



Patrón 3.11: Ejemplo de patrón de eventos

- γ es una función:

$$\gamma : P \cup \{\mathbf{0}, \infty\} \times P \cup \{\mathbf{0}, \infty\} \longrightarrow 2^{\Sigma}$$

donde para todo $\mathsf{x}, \mathsf{y} \in P \cup \{\mathbf{0}, \infty\}$, $\gamma(\mathsf{x}, \mathsf{y}) = \gamma(\mathsf{y}, \mathsf{x})$, que asocia a cada par de puntos una restricción de eventos.

- δ es una función que a cada par de puntos le asocia una restricción temporal.

Formalmente, δ es una función $\delta : P \cup \{\mathbf{0}\} \times P \cup \{\mathbf{0}\} \longrightarrow \Phi$, donde para todo $\mathsf{x}, \mathsf{y} \in P \cup \{\mathbf{0}\}$, $\delta(\mathsf{x}, \mathsf{y}) = \delta(\mathsf{y}, \mathsf{x})$.

Como se puede ver, los puntos de comienzo y fin de ejecución se representan formalmente con los puntos distinguidos $\mathbf{0}$ y ∞, respectivamente. La relación de causalidad y las restricciones de eventos se extienden de forma tal de incluir a estos dos nuevos puntos. Como dijimos antes, no se permiten restricciones temporales entre los puntos comunes y el ∞ y además todos los puntos están relacionados causalmente con $\mathbf{0}$ y ∞, aunque no se explicite gráficamente.

Ejemplo. El Patrón 3.11 corresponde a la siguiente tupla: $\mathcal{P} = \langle \Sigma, E \cup I, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, donde:

$$\Sigma = \{a, b, c, e\}$$
$$E = \{\mathsf{p}, \mathsf{q}, \mathsf{r}\}$$
$$I = \varnothing$$
$$\ell = \{\mathsf{p} \mapsto \{a\}, \mathsf{q} \mapsto \{b\}, \mathsf{r} \mapsto \{c\}\}$$
$$\rightarrow = \{\mathbf{0} \rightarrow \mathsf{p}, \mathbf{0} \rightarrow \mathsf{q}, \mathsf{p} \rightarrow \mathsf{r}, \mathsf{q} \rightarrow \mathsf{r}, \mathsf{r} \rightarrow \infty\}$$

$$\gamma = \left\{ \begin{array}{lllll} (\mathsf{p},\mathsf{p}) \mapsto \varnothing & (\mathsf{p},\mathsf{q}) \mapsto \varnothing & (\mathsf{p},\mathsf{r}) \mapsto \{e\} & (\mathsf{p},\mathbf{0}) \mapsto \{a\} & (\mathsf{p},\infty) \mapsto \varnothing \\ (\mathsf{q},\mathsf{p}) \mapsto \varnothing & (\mathsf{q},\mathsf{q}) \mapsto \varnothing & (\mathsf{q},\mathsf{r}) \mapsto \varnothing & (\mathsf{q},\mathbf{0}) \mapsto \varnothing & (\mathsf{q},\infty) \mapsto \varnothing \\ (\mathsf{r},\mathsf{p}) \mapsto \{e\} & (\mathsf{r},\mathsf{q}) \mapsto \varnothing & (\mathsf{r},\mathsf{r}) \mapsto \varnothing & (\mathsf{r},\mathbf{0}) \mapsto \varnothing & (\mathsf{r},\infty) \mapsto \{c\} \\ (\mathbf{0},\mathsf{p}) \mapsto \{a\} & (\mathbf{0},\mathsf{q}) \mapsto \varnothing & (\mathbf{0},\mathsf{r}) \mapsto \varnothing & (\mathbf{0},\mathbf{0}) \mapsto \varnothing & (\mathbf{0},\infty) \mapsto \varnothing \\ (\infty,\mathsf{p}) \mapsto \varnothing & (\infty,\mathsf{q}) \mapsto \varnothing & (\infty,\mathsf{r}) \mapsto \{c\} & (\infty,\mathbf{0}) \mapsto \varnothing & (\infty,\infty) \mapsto \varnothing \end{array} \right\}$$

$$\delta = \left\{ \begin{array}{llll} (\mathsf{p},\mathsf{p}) \mapsto [0,\infty) & (\mathsf{p},\mathsf{q}) \mapsto [0,\infty) & (\mathsf{p},\mathsf{r}) \mapsto [0,\infty) & (\mathsf{p},\mathbf{0}) \mapsto [0,\infty) \\ (\mathsf{q},\mathsf{p}) \mapsto [0,\infty) & (\mathsf{q},\mathsf{q}) \mapsto [0,\infty) & (\mathsf{q},\mathsf{r}) \mapsto [0,7] & (\mathsf{q},\mathbf{0}) \mapsto [0,\infty) \\ (\mathsf{r},\mathsf{p}) \mapsto [0,\infty) & (\mathsf{r},\mathsf{q}) \mapsto [0,7] & (\mathsf{r},\mathsf{r}) \mapsto [0,\infty) & (\mathsf{r},\mathbf{0}) \mapsto [0,\infty) \\ (\mathbf{0},\mathsf{p}) \mapsto [0,\infty) & (\mathbf{0},\mathsf{q}) \mapsto [0,\infty) & (\mathbf{0},\mathsf{r}) \mapsto [0,\infty) & (\mathbf{0},\mathbf{0}) \mapsto [0,\infty) \end{array} \right\}$$

Es importante notar que la relación de causalidad debe incluir suficientes pares de la forma $\mathbf{0} \rightarrow \mathsf{x}$ y $\mathsf{x} \rightarrow \infty$ como para garantizar que $\mathbf{0}$ sea el elemento ínfimo de la relación de orden parcial $\prec$ y ∞ sea el elemento supremo de dicha relación. Siempre es posible garantizar esto incluyendo todos los pares de esa forma.

Tal como buscábamos, los patrones de eventos son una generalización de los patrones básicos temporizados. Todo patrón básico temporizado de la forma $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$, es equivalente al patrón de eventos

$\mathcal{P}' = \langle \Sigma, P, \ell, \rightarrow', \gamma', \delta', \mathbf{0}, \infty \rangle$, donde:

$$\begin{aligned} \rightarrow' &= \rightarrow \cup \{\mathbf{0} \rightarrow x \mid x \in P\} \cup \{x \rightarrow \infty \mid x \in P\} \\ \gamma' &= \begin{cases} \gamma(x,y) & \text{si } x,y \in P \\ \varnothing & \text{en otro caso} \end{cases} \\ \delta' &= \begin{cases} \delta(x,y) & \text{si } x,y \in P \\ [0,\infty) & \text{en otro caso} \end{cases} \end{aligned}$$

3.3.2 Semántica

Una vez más, adaptamos el concepto de *matching* a los patrones de eventos.

Definición 3.3.2. Matching.
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, una ejecución $\sigma = \langle \varsigma, \tau \rangle$ sobre Σ y un mapeo $\hat{\cdot} : P \mapsto \Pi_\sigma$, decimos que $\hat{\cdot}$ es un *matching* entre σ y $\mathcal{P}$ sii verifica las condiciones **M1-M3 + MT1-MT2** y además:

$$\begin{array}{ll} \mathbf{MI} & \forall x \in P, \quad \sigma_{\hat{x})} \cap \gamma(\mathbf{0}, x) = \varnothing \\ \mathbf{MS} & \forall x \in P, \quad \sigma_{(\hat{x}} \cap \gamma(x, \infty) = \varnothing \\ \mathbf{MTI} & \forall x \in P, \quad \Delta(\tau_{\hat{x}]}) \vDash \delta(\mathbf{0}, x) \end{array}$$

Dado que los patrones de eventos son una extensión de los patrones básicos temporizados que, a su vez, son una extensión de los patrones básicos, las condiciones que debe cumplir un mapeo para ser un matching entre un patrón de eventos y una ejecución temporizada incluyen a todas las dadas anteriormente más tres nuevas. La primera y la segunda de las condiciones nuevas exigen que se respeten las restricciones de eventos entre todos los puntos comunes y $\mathbf{0}$ y ∞. De la misma manera que la condición **M3** predica sobre un segmento de ejecución delimitado por dos puntos, las condiciones **MI** y **MS** predican sobre segmentos de ejecución que constituyen prefijos o sufijos (respectivamente) de la ejecución. Como era de esperarse, la última condición exige que se cumplan las restricciones temporales especificadas para los distintos prefijos de ejecución.

Definición 3.3.3. Satisfacción de patrones de eventos.
Una ejecución temporizada σ *satisface* un patrón de eventos $\mathcal{P}$ (notado como $\sigma \vDash \mathcal{P}$) si y sólo si existe algún matching entre σ y $\mathcal{P}$.

Lenguaje de un patrón de eventos

Definición 3.3.4. Lenguaje de un patrón de eventos
Dado un patrón de eventos $\mathcal{P}$, llamaremos *lenguaje de* $\mathcal{P}$ al conjunto

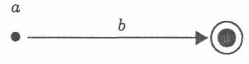
$$\mathcal{L}(\mathcal{P}) = \{\sigma \mid \sigma \vDash \mathcal{P} \text{ y } \sigma \text{ es divergente}\}$$

Definición 3.3.5. Lenguaje ampliado.
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, y un conjunto Σ' tal que $\Sigma \subseteq \Sigma'$, definimos:

$$\mathcal{L}_{\Sigma'}(\mathcal{P}) = \{ \sigma \text{ sobre } \Sigma' \mid \sigma\lceil_\Sigma \vDash \mathcal{P} \}$$

Propiedades de la satisfacción de patrones de eventos

Los patrones de eventos no son cerrados para extensiones como los patrones básicos y los patrones temporizados. Quiere decir que no es suficiente con que un prefijo finito de una ejecución satisfaga un patrón de eventos para concluir que toda la ejecución lo hace. Observemos por ejemplo el siguiente caso:



Dada una ejecución temporizada σ, supongamos que existe un prefijo de σ en el cual hay alguna ocurrencia de a y ninguna de b. Dicho prefijo satisface el patrón de arriba. Sin embargo, el patrón exige que no haya *ninguna* ocurrencia de b después de a en toda la duración de la ejecución. Es decir, no alcanza con analizar ningún prefijo finito de la ejecución. Si tuviéramos mas información sobre σ y supiéramos que el prefijo hasta cierta posición satisface el patrón y en el segmento restante de ejecución no hay ninguna ocurrencia de b entonces podríamos concluir que toda la ejecución satisface el patrón. Básicamente, todas las condiciones que debe cumplir un matching excepto **MS** predican sobre posiciones que pueden ser abarcadas por algún prefijo finito de la ejecución. Y dado que no se puede decidir si vale **MS** mirando ningún prefijo finito, es necesario contar con más información para poder hacerlo.

Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, llamaremos $\mathbb{L}_{\mathcal{P}}$ al conjunto:

$$\mathbb{L}_{\mathcal{P}} = \bigcup_{x \in P} \gamma(x, \infty)$$

Es decir, $\mathbb{L}_{\mathcal{P}}$ representa el conjunto de eventos involucrados en restricciones de eventos que deben valer hasta el final de la ejecución. En particular, es necesario que ninguno de los eventos de ese conjunto ocurra en la ejecución más allá del último punto marcado para que la ejecución satisfaga el patrón. Usando este conjunto, enunciamos una versión más débil de la propiedad de clausura por extensiones:

Propiedad 3.5. *Clausura débil por extensiones. Dado un patrón de eventos* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ *y una ejecución temporizada finita* σ *sobre* Σ*, si* $\sigma \vDash \mathcal{P}$ *entonces para cualquier ejecución* $\sigma' = \langle \varsigma, \tau \rangle$ *tal que* $\varsigma \cap \mathbb{L}_{\mathcal{P}} = \varnothing$*,* $\sigma\sigma' \vDash \mathcal{P}$*.*

Por otro lado, si supiéramos que una ejecución temporizada satisface un patrón de eventos, es más o menos evidente que existirá un prefijo finito de la misma que también satisface el patrón. El razonamiento es similar al usado para los patrones básicos y los patrones temporizados. Sin embargo, sabiendo que la ejecución satisface el patrón es posible afirmar algo aún más fuerte: el complemento de dicho prefijo seguro no contiene ningún evento del conjunto $\mathbb{L}_{\mathcal{P}}$.

Propiedad 3.6. *Satisfacción finita. Dado un patrón* $\mathcal{P} = \langle \Sigma, E, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ *y una ejecución temporizada* σ *sobre* Σ*, si* $\sigma \vDash \mathcal{P}$ *entonces existe una posición* $i \in \Pi_\sigma$ *tal que* $\sigma_{i]} \vDash \mathcal{P}$ *y, además,* $\sigma_{(i} \cap \mathbb{L}_{\mathcal{P}} = \varnothing$*.*

Como en los casos anteriores, las demostraciones completas de ambas propiedades se encuentran en el Apéndice A.

Capítulo 4

Model-checking de patrones de mal comportamiento

Habiendo presentado los patrones de eventos, vamos a enfocarnos ahora hacia el problema de verificación de propiedades en un sistema de tiempo real. Dado un sistema S modelado con un autómata temporizado $\mathcal{A}_S$, nuestro enfoque consistirá en expresar casos en los que se viole un determinado requerimiento R utilizando para ello patrones de eventos. Dado un requerimiento R, llamaremos $\mathcal{P}_{\neg R}$ al patrón de eventos que captura los comportamientos que violan R. Cuando usemos patrones de eventos exclusivamente para expresar comportamientos no deseados en el sistema, nos referiremos a ellos como *patrones de mal comportamiento.* Por otro lado, cuando usemos el termino *patrones de eventos* nos estaremos refiriendo al formalismo en general, sin denotar ningún uso particular para los mismos.

En general, los requerimientos de un sistema representan una propiedad que deben cumplir *todas* las ejecuciones de dicho sistema. Quiere decir que alcanza con encontrar *una* ejecución que *no* cumpla esa propiedad para concluir que el sistema *viola* el requerimiento. Dado que usaremos los patrones de eventos para describir los casos en los que se viola un requerimiento, alcanza con que una sola de las ejecuciones del sistema satisfaga el patrón para concluir que el sistema es incorrecto (con respecto a ese requerimiento). Dado un sistema S y un patrón de mal comportamiento $\mathcal{P}$, diremos que $S \models \mathcal{P}$ si y sólo si algún comportamiento de S verifica el patrón. Dado que el sistema estará representado por $\mathcal{A}_S$ y los comportamientos del sistema por ejecuciones temporizadas divergentes, diremos que $S \models \mathcal{P}$ si y sólo si $\mathcal{A}_S \models \mathcal{P}$, y a su vez, diremos que $\mathcal{A}_S \models \mathcal{P}$ si existe una ejecución en el lenguaje de $\mathcal{A}_S$ que pertenece al lenguaje de $\mathcal{P}$. Llamaremos *model-checking* de patrones de mal comportamiento al problema de determinar si dado un autómata temporizado $\mathcal{A}$ y un patrón de mal comportamiento $\mathcal{P}$, $\mathcal{A} \models \mathcal{P}$.

En este capítulo demostraremos que el problema de *model-checking* de patrones de mal comportamiento es decidible. Para esto, mostraremos que puede ser reducido a model-checking de autómatas temporizados, que se sabe decidible [HNSY92].

En primer lugar, expresaremos el problema de model-checking de patrones de mal comportamiento en función de la vacuidad del lenguaje de un autómata de Büchi temporizado. Definiremos el concepto de *autómatas reconocedores* para cada uno de los tres tipos de patrones que introdujimos en el capítulo anterior. Los autómatas reconocedores serán, básicamente, autómatas temporizados que capturen el lenguaje de un patrón de eventos. Dado un patrón de eventos $\mathcal{P}$, su autómata reconocedor $\mathcal{A}_\mathcal{P}$ más una condición de aceptación de Büchi particular serán un *tableau* $\mathcal{T}_\mathcal{P}$ para el patrón de mal comportamiento. Mostraremos que dado un autómata temporizado $\mathcal{A}_S$, que modela un sistema S, y un patrón de mal comportamiento $\mathcal{P}$, decidir si vale $\mathcal{A}_S \models \mathcal{P}$ se reduce a decidir si $\mathcal{L}(\mathcal{A}_S \otimes \mathcal{T}_\mathcal{P}) \neq \varnothing$.

En una segunda etapa, mostraremos que decidir si $\mathcal{L}(\mathcal{A}_S \otimes \mathcal{T}_\mathcal{P}) \neq \varnothing$ se puede reducir a verificar si el autómata compuesto $\mathcal{A}_S || \mathcal{A}_\mathcal{P}$ satisface la fórmula TCTL $\varphi_{accept} = init \Rightarrow \exists\Diamond(\exists\Box accept)$. Dado que este último problema es decidible, la verificación de patrones de mal comportamiento también lo es.

Para simplificar la presentación, repetiremos el esquema incremental del capítulo anterior para presentar los autómatas reconocedores y demostrar la corrección y completitud de la construcción de los distintos tableaux. Sin embargo, pasado ese punto trataremos únicamente la decidibilidad de model-checking de patrones de mal comportamiento en su versión más expresiva (es decir: patrones básicos + tiempo + punto inicial y final). Dado que los patrones básicos y los patrones básicos temporizados son casos particulares de patrones de eventos, los resultados presentados en este capítulo serán igualmente aplicables a esos casos.

Más adelante veremos que los autómatas reconocedores son más que una construcción teórica y que tienen además una utilidad práctica. Presentaremos el pseudocódigo y la implementación en Java de un algoritmo para la construcción de autómatas reconocedores para patrones de eventos. Este algoritmo será la pieza clave para la construcción del verificador de patrones diagramado en la Figura 1.5 del capítulo 1.

4.1 Autómatas reconocedores

Empezaremos por definir el concepto de *autómatas reconocedores* para cada uno de los tres tipos de patrones presentados en el capítulo anterior. La construcción de los autómatas reconocedores se basa fuertemente en las propiedades de clausura por extensiones y de satisfacción finita de los patrones de eventos.

Informalmente, la idea detrás de la construcción de los autómatas reconocedores consiste en representar *todas* las posibles formas de construir un matching para un patrón de eventos. Dada una ejecución, el autómata comienza a consumirla y para cada posición, si pudiera ser matcheada con algún punto del patrón[1], decide en forma no determinística si lo hace o si saltea la posición. El no determinismo de la decisión permite capturar tanto el caso en que el autómata marca la posición y como el caso en que no lo hace y dado que esto es así para todas las posiciones de la ejecución, el autómata capturará todas las posibles formas de marcar los puntos de un patrón. Si en algún momento el autómata descubre que no se puede completar el matching (porque no puede ser marcado ningún otro punto más del patrón sin violar alguna restricción), entonces se aborta el proceso.

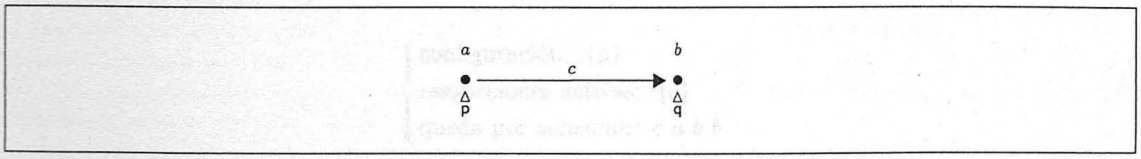
Las locaciones del autómata representarán cuantos (y cuales) de los puntos del patrón fueron marcados hasta el momento. La condición mínima que debe darse para poder pasar de una locación a otra es que la segunda corresponda a marcar exactamente un punto más que la primera. Es más o menos evidente que los puntos deben ser marcados en algún orden que respete el orden parcial determinado por el patrón. Si habiendo consumido un prefijo finito de la ejecución se viola esta propiedad, no habrá forma de continuar marcando puntos del patrón y que el mapeo obtenido pueda ser considerado un matching (recordemos que **M2** exige que los puntos sean marcados en un orden consistente con $\prec$). Para que un punto pueda ser marcado, deben haber sido marcados con anterioridad todos sus predecesores. Utilizaremos el concepto de *configuración* para representar los puntos que han sido marcados hasta un cierto momento.

Dado un patrón de eventos (de cualquier tipo), la relación de precedencia $\rightarrow$ del patrón determina (por definición) un orden parcial entre los puntos del patrón. Dado un punto x, diremos que otro punto y es *sucesor directo* del primero si $\mathsf{x} \rightarrow \mathsf{y}$. Análogamente, diremos que y es *predecesor directo* de x si $\mathsf{y} \rightarrow \mathsf{x}$. Llamaremos $suc(\mathsf{x})$ al conjunto de todos los sucesores directos de x y $pred(\mathsf{x})$ al conjunto de todos los predecesores directos de x.

Definición 4.1.1. Configuración. Dado un patrón de eventos básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$, una *configuración* es un conjunto de puntos $\Theta \in P$ cerrado a izquierda bajo $\prec_{\mathcal{P}}$. Es decir, Θ es una configuración sii $\forall \mathsf{x} \in \Theta$, $pred(\mathsf{x}) \subset \Theta$. Llamaremos $\Theta_{\mathcal{P}}$ al conjunto de todas las configuraciones de puntos de $\mathcal{P}$.

Cada locación del autómata tendrá asociada una configuración y los puntos de la configuración serán los que fueron marcados hasta el momento. La locación inicial del autómata corresponderá a la configuración $\varnothing$, es decir, a no haber marcado ningún punto. Por otro lado, la locación asociada a P corresponderá a haber marcado todos los puntos del patrón. Dada una configuración actual, marcar un nuevo punto del patrón corresponderá a *extender* la configuración.

[1] Más adelante veremos en qué casos pueden ser matcheados un punto y una posición.



Patrón 4.1: Patrón básico

Definición 4.1.2. Extensión de configuraciones.
Diremos que una configuración Θ puede ser *extendida por* x si $\Theta \uplus \{x\}$ es una configuración. En ese caso diremos, también, que $\Theta \uplus \{x\}$ es una *extensión* de Θ.
Diremos que una configuración puede ser *extendida* si existe un x tal que dicha configuración pueda ser extendida por x.

Es fácil ver que sólo se puede extender una configuración con un punto x si todos los predecesores de x ya estaban en la configuración.

Dada una cierta configuración actual y habiéndose consumido un cierto prefijo finito de una ejecución, ¿cuándo puede ser marcado un punto sobre la posición actual de la ejecución? En principio, deben cumplirse las reglas que determinan con que tipo de posiciones puede mapearse un punto. Por ejemplo, los puntos llenos de un patrón sólo pueden ser mapeados en posiciones que contengan alguno de los eventos que tienen asociados (condición **M1**) y los puntos huecos sólo pueden ser mapeados en posiciones que contengan λ (condición **MT1**). El problema es cómo garantizar que al ir construyendo un matching de a un punto por vez se respeten todas las restricciones. Analicemos el caso de las restricciones de eventos. Supongamos que estamos trabajando con el Patrón 4.1 y que estamos analizando la ejecución $\varsigma = a\ a\ c\ a\ b\ b$. El autómata comienza en el estado:

$$\begin{cases} \text{configuración: } \varnothing \\ \text{queda por consumir: } a\ a\ c\ a\ b\ b \end{cases}$$

El el primer paso, el autómata debe decidir si marca p sobre la primera posición de la ejecución o no. Supongamos que elige marcarla. El autómata pasa al siguiente estado:

$$\begin{cases} \text{configuración: } \{p\} \\ \text{queda por consumir: } a\ c\ a\ b\ b \end{cases}$$

Claramente, q no puede ser marcado sobre la segunda posición porque q está etiquetado con el evento b en el patrón y la segunda posición corresponde a evento a. Con lo cual, el autómata simplemente avanza esa posición de la ejecución y pasa al estado:

$$\begin{cases} \text{configuración: } \{p\} \\ \text{queda por consumir: } c\ a\ b\ b \end{cases}$$

Nuevamente, q no puede ser marcado sobre la posición actual. Más aún, podemos afirmar que q no podrá ser marcado nunca porque entre p y q ya sabemos que al menos hubo una ocurrencia del evento prohibido c. Quiere decir que el autómata debe abortar la construcción actual e intentar otra forma de marcar p. Para saber si en un determinado momento es *seguro* saltear una posición o no (con respecto al cumplimiento de todas las restricciones de eventos), alcanza con que el autómata tenga en cuenta las restricciones que existan entre puntos marcados y puntos que todavía no han sido marcados. Las restricciones entre puntos ya marcados no son relevantes al analizar la próxima posición de la ejecución. Lo mismo ocurre con las restricciones entre puntos que todavía no han sido marcados. Informalmente, diremos que una restricción se *activa* cuando uno de los puntos involucrados es marcado y se *desactiva* cuando se marca el segundo punto. Si hubiéramos enriquecido el estado de nuestra "máquina generadora de matchings" con los eventos correspondientes a restricciones activas, hubiéramos llegado al punto anterior en el siguiente estado:

$$\begin{cases} \text{configuración: } \{\mathsf{p}\} \\ \text{restricciones activas: } \{c\} \\ \text{queda por consumir: } c\ a\ b\ b \end{cases}$$

y como el próximo evento en la ejecución es uno de los del conjunto de restricciones activas, hubiéramos concluido que no había forma de extender el matching. Después de varios reintentos, eventualmente la máquina hubiera llegado al estado:

$$\begin{cases} \text{configuración: } \varnothing \\ \text{restricciones activas: } \varnothing \\ \text{queda por consumir: } a\ b\ b \end{cases}$$

y hubiera decidido marcar p sobre la posición actual, pasando al estado:

$$\begin{cases} \text{configuración: } \{\mathsf{p}\} \\ \text{restricciones activas: } \{c\} \\ \text{queda por consumir: } b\ b \end{cases}$$

y en este punto hubiera sido seguro marcar q sobre la siguiente posición y hubiera dejado de haber restricciones activas:

$$\begin{cases} \text{configuración: } \{\mathsf{p}, \mathsf{q}\} \\ \text{restricciones activas: } \varnothing \\ \text{queda por consumir: } b \end{cases}$$

(También hubiera sido posible que el autómata salteará la primera b y marcara q sobre la segunda. Estas dos posibilidades constituyen los únicos matchings posibles entre el patrón y la ejecución).

Dada una configuración Θ, llamaremos $\Gamma(\Theta)$ al conjunto de eventos correspondientes a restricciones activas de los puntos de Θ. Formalmente:

Definición 4.1.3. Función Γ.
Dado un patrón de eventos $\mathcal{P}$ (de cualquier tipo), donde γ es la función que determina las restricciones de eventos entre los puntos del patrón, definimos la función Γ sobre configuraciones de puntos de la siguiente manera:

$$\Gamma(\Theta) = \bigcup_{\substack{\mathsf{x} \in \Theta \\ \mathsf{y} \notin \Theta}} \gamma(\mathsf{x}, \mathsf{y})$$

Como dijimos antes, $\Gamma(\Theta)$ es la unión de las restricciones de eventos para las cuales uno de los puntos ha sido marcado (es decir, pertenece a Θ) y el otro todavía no.

Supongamos que además de estar prohibido c entre p y q, q tenía que ser la primera ocurrencia de b después de p. Al marcar p sobre la cuarta posición de la ejecución, el autómata hubiera quedado en el siguiente estado:

$$\begin{cases} \text{configuración: } \{\mathsf{p}\} \\ \text{restricciones activas: } \{b, c\} \\ \text{queda por consumir: } b\ b \end{cases}$$

y hubiera llegado a la conclusión incorrecta de que q no podía ser marcado. Sin embargo, hubiera sido correcto que se marcara q sobre la siguiente posición. Por otro lado, no hubiera sido correcto que se salteara la primera b y se marcara q sobre la segunda. Quiere decir que es distinto el conjunto de restricciones

que hay que analizar para decidir si es seguro saltear un evento que para decidir si es seguro marcarlo. Fundamentalmente, es necesario tener una noción de a quién corresponden las restricciones que están activas. Dada una configuración Θ, definimos el conjunto $\Gamma_{\triangleright \mathsf{p}}(\Theta)$ como el conjunto de eventos correspondientes a restricciones activas, excepto las correspondientes al punto p. Formalmente:

$$\forall \mathsf{p} \in P,\ \Gamma_{\triangleright \mathsf{p}}(\Theta) = \bigcup_{\substack{\mathsf{x} \in \Theta \\ \mathsf{y} \notin \Theta \\ \mathsf{y} \neq \mathsf{p}}} \gamma(\mathsf{x}, \mathsf{y})$$

Una vez más, podríamos enriquecer el estado de nuestro autómata aclarando qué puntos son los que determinan que cierto evento esté prohibido. En el ejemplo que venimos analizando, el autómata eventualmente llegaría al estado:

$$\begin{cases} \text{configuración: } \{\mathsf{p}\} \\ \text{restricciones activas: } \{b : \{\mathsf{q}\},\ c : \{\mathsf{q}\}\} \\ \text{queda por consumir: } b\ b \end{cases}$$

Donde se ve claramente que b está prohibido únicamente por q y por lo tanto puede ser marcado, aunque no salteado. En este ejemplo, $\Gamma(\{\mathsf{p}\}) = \{b, c\}$ y $\Gamma_{\triangleright \mathsf{q}}(\{\mathsf{p}\}) = \varnothing$.

4.1.1 Autómata reconocedor para patrones básicos

Un autómata reconocedor para un patrón básico tendrá una locación por cada configuración del patrón más una locación trampa distinta de las demás. Dada una configuración Θ, $[\Theta]$ representará a la locación asociada a Θ. Estando en cualquier configuración, el autómata puede decidir *marcar* un punto sobre el siguiente evento de la ejecución y cambiar de configuración, *saltear* el siguiente evento y permanecer en la misma configuración o *abortar* la construcción del matching y pasar a la locación trampa. Estando en la locación trampa, el autómata solo puede ciclar sobre esa misma locación. Estas cuatro posibilidades serán representadas con cuatro conjuntos de aristas disjuntos dos a dos: $Mark$, $Skip$, $Fail$ y $Trap$, respectivamente. El autómata no tendrá ningún reloj dado que no hay restricciones temporales que chequear. Por otro lado, los invariantes de todas las locaciones serán trivialmente $\top$. La locación inicial será $[\varnothing]$, es decir la locación asociada a la configuración vacía.

Definición 4.1.4. Autómata reconocedor para patrones básicos.
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$, $\mathcal{A}_\mathcal{P}$ es un autómata temporizado de la forma $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$ donde:

- (*Locaciones*) $S = \{[\Theta] \mid \Theta \in \Theta_\mathcal{P}\} \cup \{s_{trap}\}$, donde s_{trap} es una locación distinta de todas las demás.
  Notación. Llamaremos $s_{accept} = [P]$.
- (*Relojes*) $X = \varnothing$
- (*Eventos*) $\Sigma_{\mathcal{A}_\mathcal{P}} = \Sigma_\mathcal{P}$
- (*Aristas*) $A = Mark \cup Skip \cup Fail \cup Trap$, donde:

$$\begin{array}{rcl} Mark & = & \{\ \langle [\Theta], l, \top, \varnothing, [\Theta \uplus \{\mathsf{e}\}] \rangle \mid l \in \ell(\mathsf{e})\ \text{y}\ l \notin \Gamma_{\triangleright \mathsf{e}}(\Theta)\ \} \\ Skip & = & \{\ \langle [\Theta], l, \top, \varnothing, [\Theta] \rangle \mid l \in \Sigma\ \text{y}\ l \notin \Gamma(\Theta)\ \} \\ Fail & = & \{\ \langle [\Theta], l, \top, \varnothing, s_{trap} \rangle \mid l \in \Gamma(\Theta)\ \} \\ Trap & = & \{\ \langle s_{trap}, l, \top, \varnothing, s_{trap} \rangle \mid l \in \Sigma\ \} \end{array}$$

- (*Invariantes*) $\forall s \in S,\ \mathcal{I}(s) = \top$
- (*Locación inicial*) $s_0 = [\varnothing]$

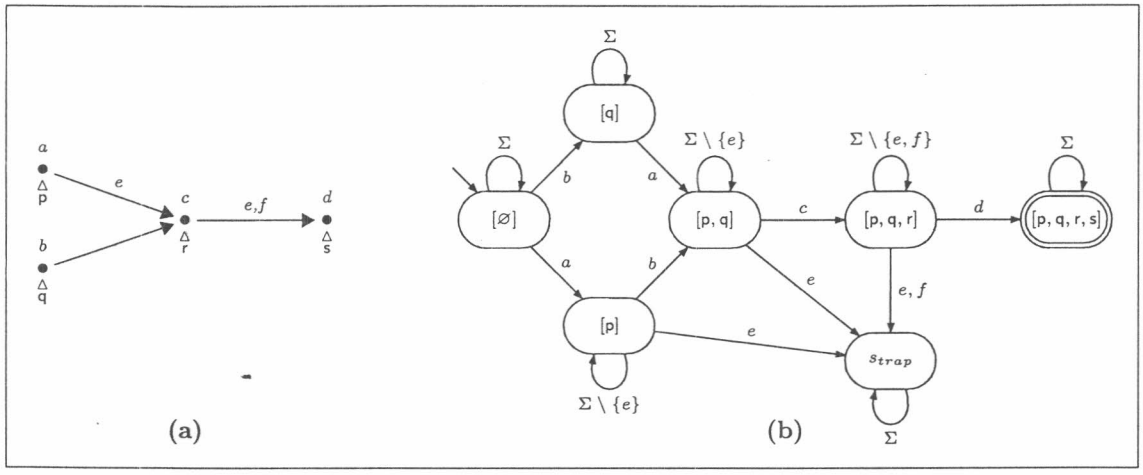


Figura 4.1: Patrón básico (a) y su autómata reconocedor (b)

Como vimos antes, para poder marcar un punto, el evento que ocupa la siguiente posición de la ejecución debe ser uno de los que etiqueta al punto en el patrón. Además, para que sea seguro marcar el punto es necesario que el próximo evento no corresponda a ninguna restricción activa, más allá de las referidas al punto que se quiere marcar. Por otro lado, siempre será seguro saltear un evento que no viole ninguna restricción activa y siempre será seguro abortar la construcción del matching al encontrar un evento que viole alguna restricción activa. Es claro que el autómata así construido no es determinístico.

Ejemplo. *Autómata reconocedor para un patrón básico.*
La Figura 4.1 muestra un patrón básico y su autómata reconocedor. La locación s_{accept} se indica con un doble borde. La arista que va de [q] a [p, q] por a es un ejemplo de arista $Mark$. La arista que va de [p, q] a s_{trap} por e es un ejemplo de arista $Fail$. Los loops sobre [p] por todos los eventos menos e son ejemplos de aristas $Skip$. Finalmente, los loops sobre s_{trap} son ejemplos de aristas $Trap$.

Propiedades de los autómatas reconocedores para patrones básicos

Analizaremos las propiedades de los autómatas reconocedores para patrones básicos.

Observación. En todo autómata reconocedor para un patrón básico los conjuntos de aristas $Mark$, $Skip$, $Fail$ y $Trap$ son disjuntos dos a dos.

Las aristas de cada uno de estos tipos se diferencian entre sí con solo mirar sus extremos: las aristas $Mark$ van de una configuración a otra que es una extensión de la primera, las aristas $Skip$ son loops sobre configuraciones, las aristas $Fail$ van de una configuración a la locación trampa y las aristas $Trap$ son loops sobre la locación trampa.

Propiedad 4.1. *Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, sean Θ y Θ' configuraciones de $\mathcal{P}$. Si $[\Theta] \Rightarrow [\Theta']$, entonces toda evolución entre $[\Theta]$ y $[\Theta']$ tiene la forma:*

$$([\Theta_0], v_0) \Rightarrow_{t_0}^{a_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{a_1} \ldots ([\Theta_i], v_i) \Rightarrow_{t_i}^{a_i} \ldots \Rightarrow_{t_{n-1}}^{a_{n-1}} ([\Theta_n], v_n)$$

*donde $\Theta_0 = \Theta$, $\Theta_n = \Theta'$ y para todo i, $0 \leq i < n$, $v_i \in \mathcal{V}_X$, $a_i \in \Sigma \cup \{\lambda\}$, $t_i \in \Re^+$ y Θ_{i+1} **es igual a** Θ_i **o es una extensión de** Θ_i.*

La demostración completa de esta propiedad se encuentra en el Apéndice A. Intuitivamente, esta propiedad dice que las configuraciones no decrecen a lo largo de una evolución sobre el autómata. Esto quiere decir que o bien se marca un punto sobre la ejecución extendiendo la configuración actual o se saltea el siguiente evento y se permanece en la misma configuración. Nunca se "desmarca" un punto marcado. Las distintas posibilidades que existen para marcar los puntos de un patrón sobre una ejecución se capturan a través del no determinismo del autómata reconocedor y no mediante un mecanismo de *backtracking*.

Como dijimos antes, el autómata reconocedor intenta construir un matching a medida que consume una ejecución. En todo momento el conjunto de puntos ya marcados está dado por la configuración asociada a la locación actual. El mapeo entre los puntos de la configuración y la porción de ejecución ya consumida es una especie de "matching a medio construir". Formalizaremos este concepto definiendo la noción de *matching básico parcial*.

Definición 4.1.5. Matching básico parcial.
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$, un conjunto de puntos $C \subseteq P$, una ejecución ς sobre Σ y un mapeo $\hat{\cdot} : C \mapsto \Pi_\varsigma$, decimos que $\hat{\cdot}$ es un *matching básico parcial* entre ς y $\mathcal{P}$ restringido a C sii verifica las siguientes condiciones:

MP1	$\forall x \in E \cap C,$	$\varsigma_{\hat{x}} \in \ell(x)$
MP2	$\forall x, y \in C,$	$x \prec y \Rightarrow \hat{x} < \hat{y}$
MP3	$\forall x, y \in C,$	$\hat{x} < \hat{y} \Rightarrow \varsigma_{(\hat{x},\hat{y})} \cap \gamma(x, y) = \varnothing$
MP4	$\forall x \in C,\ y \in P \setminus C,$	$\varsigma_{(\hat{x}} \cap \gamma(x, y) = \varnothing$

Como era de esperarse, las condiciones **MP1**-**MP3** piden que el mapeo $\hat{\cdot}$ verifique las condiciones de validez restringidas al conjunto de puntos ya marcados. Esto es fundamental para que $\hat{\cdot}$ pueda ser extendido eventualmente hasta ser un matching básico. La condición **MP4** pide, además que ninguna posición de las consumidas viole una restricción activa. A partir del concepto de matching básico parcial podemos definir *satisfacción parcial*.

Definición 4.1.6. Satisfacción parcial de patrones básicos.
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ y un conjunto de puntos $C \subseteq P$, una ejecución ς *verifica* o *satisface parcialmente* $\mathcal{P}$ *restringido a* C (notado $\varsigma \vDash_C \mathcal{P}$) sii existe un matching básico parcial entre ς y $\mathcal{P}$ restringido a C.

Notar que cuando $C = P$, las condiciones **MP1** - **MP3** son idénticas a **M1** - **M3** y, además, la condición **MP4** es trivialmente verdadera, con lo cual para cualquier patrón básico $\mathcal{P}$ y cualquier ejecución ς, $\varsigma \vDash \mathcal{P}$ sii $\varsigma \vDash_P \mathcal{P}$.

Si nuestro autómata reconocedor estuviera correctamente construido, quisiéramos que en todo momento la porción de ejecución consumida satisfaga parcialmente el patrón, restringido a los puntos de la configuración actual. Si esto fuera así, al marcar el último punto del patrón podríamos garantizar que hemos construido un matching. Llamaremos a esta propiedad *Corrección del autómata reconocedor*.

Propiedad 4.2. *Corrección del autómata reconocedor para patrones básicos.*
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ *y su autómata reconocedor* $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, *para toda configuración* $\Theta \in \Theta_\mathcal{P}$ *y toda ejecución finita* ς *sobre* Σ,

$$si\ q_{init} \Rightarrow^\varsigma [\Theta]\ entonces\ \varsigma \vDash_\Theta \mathcal{P}$$

La demostración de esta propiedad se basa en que para alcanzar la configuración $[\Theta]$, el autómata tuvo que atravesar un cierto camino desde q_{init} hasta dicha locación. Dicho camino estará formado por aristas $Skip$ o $Mark$ y es posible construir un matching parcial "imitando" la forma en que el autómata fue marcando los puntos (es decir, las posiciones para las cuales atravesó una arista $Mark$). La demostración detallada se encuentra en el Apéndice A.

Por otro lado, nos gustaría poder garantizar que el autómata es capaz de construir todo posible matching. Quiere decir que para una porción finita de una ejecución y una determinada configuración de puntos, si existe algún matching parcial entre el patrón y la ejecución restringido a esa configuración, entonces el autómata tiene que poder reproducirlo. Es decir, el autómata tiene que poder consumir esa porción de ejecución quedando en esa configuración. Llamaremos a esta propiedad *Completitud del autómata reconocedor.*

Propiedad 4.3. *Completitud del autómata reconocedor para patrones básicos.*
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ *y su autómata reconocedor* $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, *para toda configuración* $\Theta \in \Theta_\mathcal{P}$ *y toda ejecución finita* ς *sobre* Σ,

$$si\ \varsigma \vDash_\Theta \mathcal{P}\ entonces\ q_{init} \Rightarrow^\varsigma [\Theta]$$

La demostración de esta propiedad utiliza la estrategia inversa a la usada para el la propiedad de Corrección: a partir de un matching parcial construiremos un camino sobre el autómata entre q_{init} y $[\Theta]$ que consuma ς. Como en el caso anterior, los momentos en los cuales el autómata atravesará una arista $Mark$ corresponderán con una posición marcada según el matching parcial. El resto de las aristas serán $Skip$. La demostración completa se encuentra en el Apéndice A.

Sabiendo que valen estas dos propiedades, podemos garantizar que el autómata es capaz de construir todos los matchings posibles para un patrón y una ejecución y, además, sólo alcanza la locación s_{accept} si fue capaz de construir un matching para la porción de ejecución que consumió.

Tableau para patrones básicos

Dado un autómata reconocedor para un patrón básico, nos interesa quedarnos sólo con las ejecuciones que se *estabilizan* en la locación s_{accept}[2], es decir, aquellas para las cuales pudo ser construido al menos un matching básico. Quiere decir que s_{accept}, como su nombre lo sugiere, es una locación de *aceptación* para nosotros. Veremos a continuación que el autómata reconocedor más la condición de aceptación de Büchi dada por el conjunto unitario $\{s_{accept}\}$ acepta exactamente el mismo lenguaje que el patrón.

Teorema 4.4. *Tableau para patrones básicos. Dado un patrón básico* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ *y su autómata reconocedor* $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, *el autómata de Büchi temporizado* $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$ *reconoce el lenguaje* $\mathcal{L}(\mathcal{P})$.

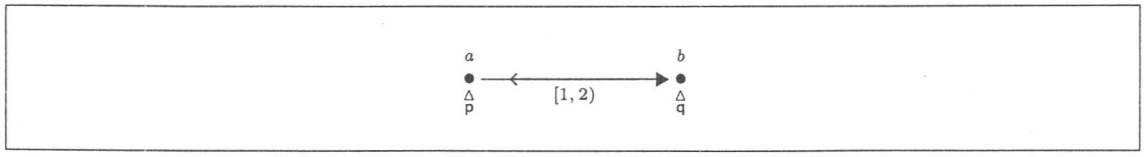
La demostración de este teorema está dividida en dos partes: se demuestra primero que $\mathcal{L}^\star(\mathcal{T}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P})$ y luego que $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}^\star(\mathcal{T}_\mathcal{P})$. La demostración completa se encuentra en el Apéndice A.

4.1.2 Autómata reconocedor para patrones básicos temporizados

Hasta este punto, hemos mostrado como construir un autómata temporizado que construya matchings para un patrón básico y hemos mostrado que usando las funciones Γ y $\Gamma_{\triangleright x}$ se puede garantizar que se cumplan las restricciones de eventos.

En esta sección veremos además como garantizar que se cumplan las restricciones temporales de los patrones temporizados. Naturalmente, usaremos los relojes del autómata reconocedor para medir el tiempo

[2] Diremos que una ejecución se *estabiliza* en una locación si la alcanza y además permanece indefinidamente en ella



Patrón 4.2: Patrón temporizado

transcurrido desde que se marcó un punto (sea un punto lleno o un punto hueco). Dado que definimos el concepto de *configuración* para cualquier tipo de punto, esto también se aplica a los puntos huecos. Lo mismo ocurre con las funciones Γ y $\Gamma_{\triangleright\times}$. Como pasaba con las restricciones de eventos, diremos que las restricciones temporales se *activan* cuando se marca uno de los puntos involucrados y se *desactivan* cuando se marca el segundo. Para que sea seguro marcar el segundo punto, es necesario garantizar que el tiempo transcurrido desde que se marcó verifica la restricción temporal asociada a estos puntos. Para determinar si eso es cierto, utilizaremos un reloj por cada punto y cada reloj será reseteado únicamente cuando se marque el punto correspondiente. Conceptualmente estamos extendiendo el estado de nuestra "maquina generadora de matchings" para que mantenga el tiempo transcurrido desde que se marcó cada punto. Veamos el siguiente ejemplo. Supongamos que estamos trabajando con el Patrón 4.2 y queremos encontrar los markings entre dicho patrón y la ejecución temporizada

$$\left\langle \begin{matrix} a & a & c & b & \cdots \\ 2 & 3 & 3.6 & 5.5 & \end{matrix} \right\rangle$$

El autómata comenzará en el estado:

$$\begin{cases} \text{configuración: } \varnothing \\ \text{tiempo ocurrencia: } \varnothing \\ \text{restricciones activas: } \varnothing \\ \text{queda por consumir: } \begin{matrix} a & a & c & b & \cdots \\ 2 & 3 & 3.6 & 5.5 & \end{matrix} \end{cases}$$

Supongamos que ahora el autómata decide marcar p sobre la primera posición y pasa al estado:

$$\begin{cases} \text{configuración: } \{\mathsf{p}\} \\ \text{tiempo ocurrencia: } \{\mathsf{p} : 2\} \\ \text{restricciones activas: } \{a : \{\mathsf{q}\}\} \\ \text{queda por consumir: } \begin{matrix} a & c & b & \cdots \\ 3 & 3.6 & 5.5 & \end{matrix} \end{cases}$$

En este momento, la siguiente posición de la ejecución no puede ser salteada dado que contiene un evento a que está en conflicto con las restricciones activas referentes a q. Quiere decir que el autómata debe abortar la construcción del matching e intentar otra forma de marcar p. Supongamos que el autómata hubiera decidido marcar p sobre la segunda a:

$$\begin{cases} \text{configuración: } \{\mathsf{p}\} \\ \text{tiempo ocurrencia: } \{\mathsf{p} : 3\} \\ \text{restricciones activas: } \{a : \{\mathsf{q}\}\} \\ \text{queda por consumir: } \begin{matrix} c & b & \cdots \\ 3.6 & 5.5 & \end{matrix} \end{cases}$$

Es seguro saltear c y nuestra máquina pasa al estado:

$$\begin{cases} \text{configuración: } \{\mathsf{p}\} \\ \text{tiempo ocurrencia: } \{\mathsf{p} : 3\} \\ \text{restricciones activas: } \{a : \{\mathsf{q}\}\} \\ \text{queda por consumir: } \begin{matrix} b & \cdots \\ 5.5 & \end{matrix} \end{cases}$$

Ahora, el autómata debe decidir si marca q sobre la siguiente posición. Sin embargo, el tiempo transcurrido desde que se marcó p es 2.5 y $2.5 \not\models [1, 2)$. Con lo cual, lo único que puede hacer el autómata es abortar la construcción.

En definitiva, la estrategia para verificar que se cumplan las restricciones temporales será la siguiente: a cada punto p se le asociará un reloj z_{p} que será reseteado en la arista que marque dicho punto. A su vez, para poder marcar un punto q, se deberá garantizar por medio de una guarda que el tiempo transcurrido desde el resto de los puntos ya marcados hasta el instante actual verifique las correspondientes restricciones temporales.

Dada una restricción temporal φ, $\psi_x(\varphi)$ será una restricción sobre relojes que verifique φ sobre el valor del reloj x. Formalmente, definimos $\psi_x(\varphi)$ de la siguiente manera:

$$\begin{array}{lcl}
\psi_x((a,b)) & = & a < x \wedge x < b \\
\psi_x((a,b]) & = & a < x \wedge x \leq b \\
\psi_x([a,b)) & = & a \leq x \wedge x < b \\
\psi_x([a,b]) & = & a \leq x \wedge x \leq b \\
\psi_x((a,\infty)) & = & a < x \\
\psi_x([a,\infty)) & = & a \leq x \\
\psi_x(\neg\theta) & = & \neg\psi_x(\theta)
\end{array}$$

Naturalmente, queremos que una restricción $\psi_x(\varphi)$ sea verdadera cuando $v(x) = t$ si y sólo si $t \models \varphi$.

Proposición 4.5. *(Preservación de verdad) Dada una restricción temporal φ, para todo real no negativo t,*

$$\psi_x(\varphi)[x|t] \text{ es verdadero sii } t \models \varphi$$

La demostración de esta propiedad es bastante directa y puede encontrarse en el Apéndice A. Por otro lado, dado que las restricción sobre relojes construidas con la definición anterior siempre involucran un único reloj, es fácil ver que para determinar si una valuación satisface una restricción $\psi_x(\varphi)$, alcanza con ver si el valor del reloj x según dicha valuación satisface φ.

Proposición 4.6. *Dada una restricción temporal φ y un reloj x,*

$$v \models \psi_x(\varphi) \text{ sii } v(x) \models \varphi$$

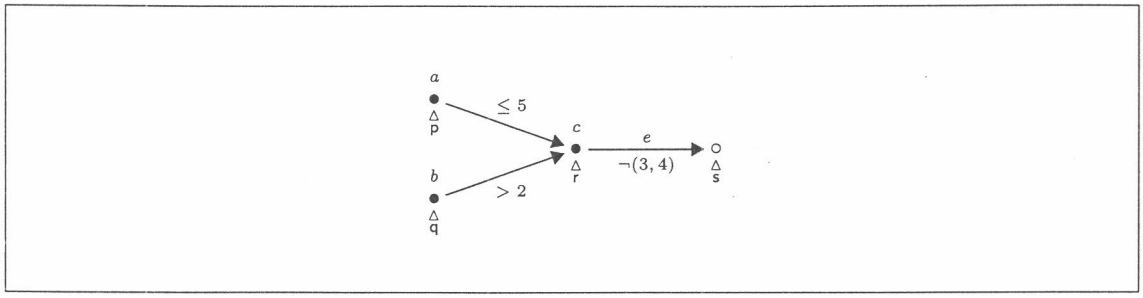
Nuevamente, la demostración es muy directa y puede encontrarse en el Apéndice A. Teniendo la representación de cada restricción temporal asociada a un punto escrita como restricción de relojes, se puede construir la guarda de la arista que marcan dicho punto como una conjunción de estas fórmulas atómicas:

Definición 4.1.7. Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$, el conjunto de relojes $X = \{z_{\mathsf{x}} \mid \mathsf{x} \in P\}$ y un conjunto de puntos $\Theta \subseteq P$, para cada punto $\mathsf{p} \in P$ definimos:

$$\psi^{\mathsf{p}}_{\Theta} \stackrel{def}{=} \bigwedge_{\mathsf{x} \in \Theta} \psi_{z_{\mathsf{x}}}(\delta(\mathsf{x}, \mathsf{p}))$$

Tomemos por ejemplo el Patrón 4.3. Estando en la configuración $\Theta = \{\mathsf{p}, \mathsf{q}\}$, la guarda para poder marcar r, es decir $\psi^{\mathsf{r}}_{\Theta}$ es igual a:

$$\psi_{z_{\mathsf{p}}}([0,5]) \wedge \psi_{z_{\mathsf{q}}}((2,\infty)) \quad = \quad z_{\mathsf{p}} \leq 5 \wedge z_{\mathsf{q}} > 2$$



Patrón 4.3: Patrón temporizado complejo

Quiere decir, en el momento de marcar r, deben haber pasado 5 u.t. o menos desde que se marcó p y más de 2 u.t. desde que se marcó q.
Por otro lado, estando en la configuración $\Theta' = \{\mathsf{p}, \mathsf{q}, \mathsf{r}\}$, la guarda para la arista que marque s será:

$$\psi_{z_\mathsf{p}}([0,\infty)) \wedge \psi_{z_\mathsf{q}}([0,\infty)) \wedge \psi_{z_\mathsf{r}}(\neg(3,4)) \quad = \quad z_\mathsf{r} < 3 \vee z_\mathsf{r} > 4$$

Dado que no hay ninguna restricción temporal no trivial entre p, q y s, las restricciones asociadas son equivalentes a $\top$ ($\psi_{z_\mathsf{q}}([0,\infty)) = x \geq 0 \wedge x < \infty$, lo cual es trivialmente verdadero para cualquier reloj, y por eso las omitimos.

Un autómata reconocedor para un patrón temporizado tendrá la misma estructura básica de los autómatas reconocedores para patrones básicos: una locación por configuración más una locación trampa distinta de las demás y cuatro conjuntos disjuntos dos a dos de aristas *Mark*, *Skip*, *Fail* y *Trap*. Además, tendrán un conjunto adicional de aristas llamado *Instant* que corresponderán a marcar puntos huecos. El autómata tendrá un reloj por cada punto del patrón y las aristas *Mark* e *Instant* estarán protegidas por guardas como las descritas anteriormente que garanticen el cumplimiento de las restricciones temporales. Finalmente, los invariantes de todas las locaciones seguirán siendo $\top$ y la locación inicial será $[\varnothing]$.

Definición 4.1.8. Autómata reconocedor para patrones temporizados
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P = E \cup I, \ell, \rightarrow, \gamma, \delta \rangle$, $\mathcal{A}_\mathcal{P}$ es un autómata temporizado de la forma $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$ donde:

- (*Locaciones*) $S = \{[\Theta] \mid \Theta \in \Theta_\mathcal{P}\} \cup \{s_{trap}\}$, donde s_{trap} es una locación distinta de todas las demás.
  Notación. Llamaremos $s_{accept} = [P]$.
- (*Relojes*) $X = \{z_\mathsf{x} \mid \mathsf{x} \in P\}$
- (*Eventos*) $\Sigma_{\mathcal{A}_\mathcal{P}} = \Sigma_\mathcal{P}$
- (*Aristas*) $A = Mark \cup Instant \cup Skip \cup Fail \cup Trap$, donde:

$$\begin{array}{rcll}
Mark & = & \{ \langle [\Theta], l, \psi^{\mathsf{e}}_\Theta, \{z_\mathsf{e}\}, [\Theta \uplus \{\mathsf{e}\}] \rangle & \mid \mathsf{e} \in E,\ l \in \ell(\mathsf{e}) \text{ y } l \notin \Gamma_{\triangleright \mathsf{e}}(\Theta) \} \\
Instant & = & \{ \langle [\Theta], \lambda, \psi^{\mathsf{i}}_\Theta, \{z_\mathsf{i}\}, [\Theta \uplus \{\mathsf{i}\}] \rangle & \mid \mathsf{i} \in I \} \\
Skip & = & \{ \langle [\Theta], l, \top, \varnothing, [\Theta] \rangle & \mid l \in \Sigma \text{ y } l \notin \Gamma(\Theta) \} \\
Fail & = & \{ \langle [\Theta], l, \top, \varnothing, s_{trap} \rangle & \mid l \in \Gamma(\Theta) \} \\
Trap & = & \{ \langle s_{trap}, l, \top, \varnothing, s_{trap} \rangle & \mid l \in \Sigma \}
\end{array}$$

- (*Invariantes*) $\forall s \in S,\ \mathcal{I}(s) = \top$
- (*Locación inicial*) $s_0 = [\varnothing]$

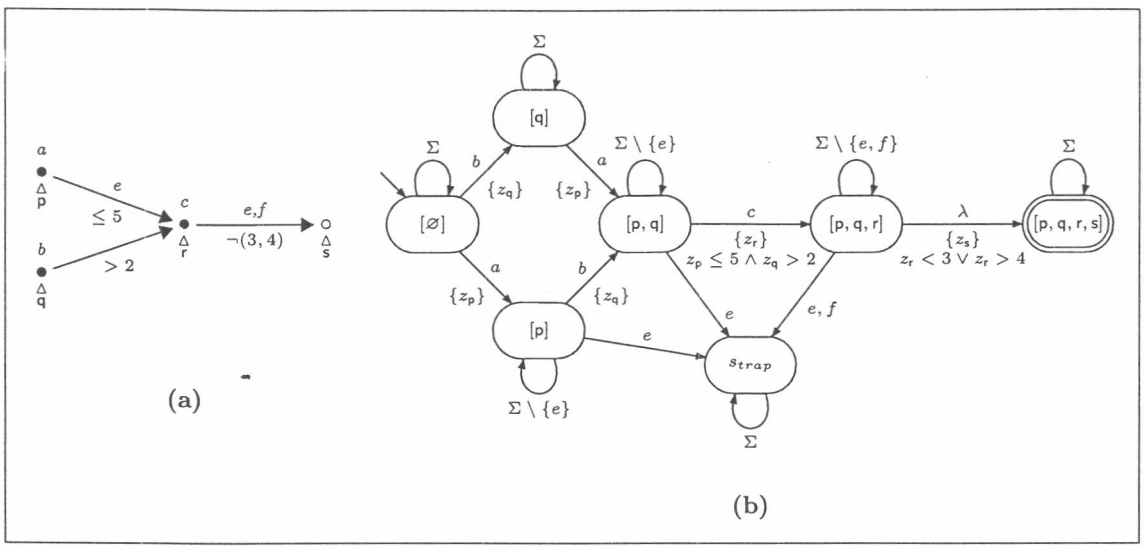


Figura 4.2: Patrón temporizado (a) y su autómata reconocedor (b)

Las aristas *Mark* y las aristas *Instant* son estructuralmente muy similares. La única diferencia es que las primeras siempre están etiquetadas con un evento y corresponden a marcar un punto de la clase E y las segundas siempre están etiquetadas con λ y corresponden a marcar un punto de la clase I. En ambos casos será seguro marcar un punto (desde el punto de vista de las restricciones temporales) si vale la guarda correspondiente a la configuración actual y a ese punto. Como vimos antes, esta guarda contiene una representación de todas las restricciones temporales sobre ese punto en función de inecuaciones sobre relojes del autómata. Cuando se marca un punto, se resetea el reloj asociado a ese punto para comenzar a medir el tiempo transcurrido desde ese instante.

Es importante notar que en la práctica no siempre se necesitan tantos relojes como puntos tenga el patrón. En principio, sólo se necesitan relojes para aquellos puntos que estén involucrados en restricciones temporales no triviales (es decir, distintas de $[0, \infty)$). Además, podría darse el caso de que el mismo reloj pudiera ser usado para medir el tiempo de ocurrencia de dos puntos distintos (porque, por ejemplo, nunca se da que restricciones temporales de esos puntos estén activas a la vez). Esta observación resulta de importancia dado que la complejidad de los algoritmos de model-checking de autómatas que usaremos más adelante depende del número de relojes del autómata. Preferimos mantener la construcción formal de los autómatas reconocedores tan simple como fuera posible y por eso no hicimos ningún intento por reducir en esta presentación la cantidad de relojes del autómata. Sin embargo, cuando presentemos el algoritmo que construye autómatas reconocedores a partir de patrones de eventos descartaremos los relojes correspondientes a puntos que sólo están involucrados en restricciones triviales. Si fuera necesario una optimización mayor del conjunto de relojes, podría aplicarse alguna herramienta de minimización de relojes (ej: [DY96]).

Ejemplo. *Autómata reconocedor para un patrón temporizado.*
La Figura 4.2 muestra un patrón temporizado y su autómata reconocedor. La arista que va de $[\mathsf{p}, \mathsf{q}, \mathsf{r}]$ a $[\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}]$ por λ es un ejemplo de arista *Instant*. Todas las aristas *Mark* e *Instant* resetean el reloj asociado al punto que están marcando. La arista que va de $[\mathsf{p}, \mathsf{q}]$ a $[\mathsf{p}, \mathsf{q}, \mathsf{r}]$ muestra un ejemplo de guarda no trivial sobre una arista *Mark* y la que va de $[\mathsf{p}, \mathsf{q}, \mathsf{r}]$ a $[\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}]$ muestra lo mismo para una arista *Instant*. En el ejemplo resulta claro que z_s nunca se utiliza en ninguna restricción y que, por lo tanto podría eliminarse.

Propiedades de los autómatas reconocedores para patrones temporizados

Analizaremos las propiedades de los autómatas reconocedores de patrones temporizados. La primera observación que haremos está relacionada con los tipos de aristas del autómata:

Observación. En todo autómata reconocedor para un patrón temporizado los conjuntos de aristas $Mark$, $Instant$, $Skip$, $Fail$ y $Trap$ son disjuntos dos a dos.

Ya sabemos que $Skip$, $Fail$ y $Trap$ pueden distinguirse entre ellos y con respecto a $Mark$ e $Instant$ analizando sólo los extremos de las aristas. Pero además, como $Mark$ sólo admite aristas etiquetadas con eventos e $Instant$ sólo admite aristas etiquetadas con λ, queda claro que esos conjuntos también son disjuntos entre sí.

Finalmente, también vale que en todas las evoluciones del autómata la configuración actual va extendiéndose o permanece igual, pero nunca decrece.

Propiedad 4.7. *Dado un patrón temporizado* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ *y su autómata reconocedor* $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, *sean* Θ *y* Θ' *configuraciones de* $\mathcal{P}$. *Si* $[\Theta] \Rightarrow [\Theta']$, *entonces toda evolución entre* $[\Theta]$ *y* $[\Theta']$ *tiene la forma:*

$$([\Theta_0], v_0) \Rightarrow_{t_0}^{a_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{a_1} \dots ([\Theta_i], v_i) \Rightarrow_{t_i}^{a_i} \dots \Rightarrow_{t_{n-1}}^{a_{n-1}} ([\Theta_n], v_n)$$

donde $\Theta_0 = \Theta$, $\Theta_n = \Theta'$ *y para todo* i, $0 \leq i < n$, $v_i \in \mathcal{V}_X$, $a_i \in \Sigma \cup \{\lambda\}$, $t_i \in \Re^+$ *y* Θ_{i+1} ***es igual a*** Θ_i ***o es una extensión de*** Θ_i.

La demostración de esta propiedad para el caso temporizado se encuentra en el Apéndice A.

Como en el caso de los autómatas reconocedores para patrones básicos, definimos el concepto de *matching temporizado parcial* para caracterizar el estado interno de nuestra "maquina generadora de matchings":

Definición 4.1.9. Matching temporizado parcial.
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$, un conjunto de puntos $C \subseteq P$, una ejecución temporizada $\sigma = \langle \varsigma, \tau \rangle$ sobre Σ y un mapeo $\hat{\cdot} : C \mapsto \Pi_\sigma$, decimos que $\hat{\cdot}$ es un *matching temporizado parcial* entre σ y $\mathcal{P}$ restringido a C sii verifica las condiciones **MP1-MP4** y además verifica que:

$$\begin{array}{ll} \textbf{MPT1} & \forall x \in I \cap C, \quad \varsigma_{\hat{x}} = \lambda \\ \textbf{MPT2} & \forall x, y \in C, \quad \hat{x} < \hat{y} \;\Rightarrow\; \Delta(\tau_{[\hat{x},\hat{y}]}) \vDash \delta(x, y) \end{array}$$

Quiere decir que en todo momento, los puntos ya marcados sobre la porción de ejecución consumida respetan todas las restricciones de eventos (**MP1-MP3**) y temporales (**MPT1** - **MPT2**) y, además, los eventos consumidos hasta el momento no violan ninguna restricción de eventos activa (**MP4**).
Adaptamos la noción de *satisfacción parcial* a los patrones temporizados.

Definición 4.1.10. Satisfacción parcial de patrones temporizados.
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ y un conjunto de puntos $C \subseteq P$, una ejecución temporizada σ sobre Σ *verifica* o *satisface parcialmente* $\mathcal{P}$ *restringido a* C (notado $\sigma \vDash_C \mathcal{P}$) sii existe un matching temporizado parcial entre σ y $\mathcal{P}$ restringido a C.

Nuevamente, para cualquier patrón temporizado $\mathcal{P}$ y cualquier ejecución temporizada σ, $\sigma \vDash \mathcal{P}$ si y sólo si $\sigma \vDash_P \mathcal{P}$.

Para terminar de caracterizar el comportamiento de los autómatas reconocedores de patrones temporizados, adaptaremos las propiedades de *Completitud* y *Corrección* de los autómatas reconocedores de patrones básicos al caso temporizado.

Propiedad 4.8. *Corrección del autómata reconocedor para patrones temporizados.*
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ *y su autómata reconocedor* $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, *para toda configuración* $\Theta \in \Theta_\mathcal{P}$ *y toda ejecución finita* $\sigma = \langle \varsigma, \tau \rangle$ *sobre* Σ,

$$si\ q_{init} \Rightarrow_\tau^\varsigma [\Theta]\ entonces\ \sigma \vDash_\Theta \mathcal{P}$$

Como antes, esta propiedad garantiza que toda porción de ejecución consumida por el autómata satisface parcialmente el patrón restringido a los puntos de la configuración actual. La demostración repite la estrategia del caso no temporizado: construir un matching temporizado parcial emulando una evolución del autómata desde q_{init} hasta $[\Theta]$. De esta forma, las posiciones consumidas por aristas *Mark* o *Instant* serán marcadas con el punto correspondiente. La demostración completa puede encontrarse en el Apéndice A

Propiedad 4.9. *Completitud del autómata reconocedor para patrones temporizados.*
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ *y su autómata reconocedor* $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, *para toda configuración* $\Theta \in \Theta_\mathcal{P}$ *y toda ejecución finita* σ *sobre* Σ,

$$si\ \sigma \vDash_\Theta \mathcal{P}\ entonces\ q_{init} \Rightarrow^\sigma [\Theta]$$

La propiedad de Completitud garantiza que si existe algún matching temporizado parcial entre un prefijo de una ejecución y el patrón (restringido a alguna configuración Θ), el autómata será capaz de generarlo. Dado que esto vale también para la configuración P y que los patrones temporizados tienen la propiedad de satisfacción finita, podremos garantizar que el autómata será capaz de generar todos los matchings temporizados que existan entre una ejecución y un patrón. La demostración de que los autómatas reconocedores para el caso temporizado cumplen esta propiedad se basa en mostrar que se puede construir una evolución sobre el autómata a partir de un matching parcial para un prefijo de ejecución y una determinada configuración. La demostración completa está incluida en el Apéndice A.

Tableau para patrones temporizados

Como en el caso no temporizado, un autómata reconocedor para un cierto patrón temporizado aumentado con la condición de aceptación de Büchi $\{s_{accept}\}$ acepta exactamente el lenguaje de dicho patrón.

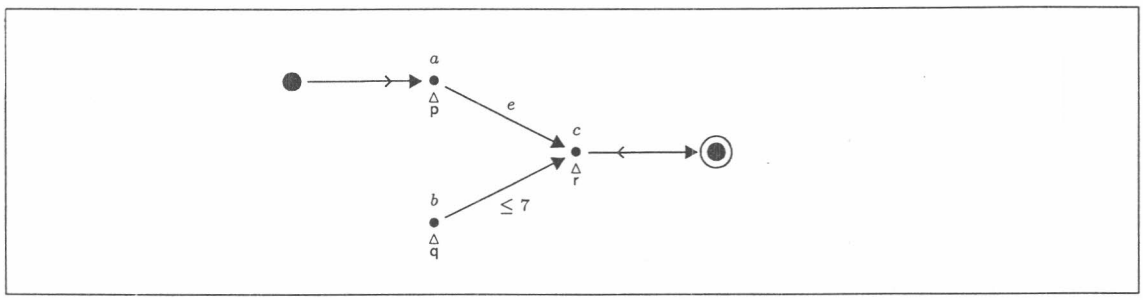
Teorema 4.10. *Tableau para patrones temporizados. Dado un patrón temporizado* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ *y su autómata reconocedor* $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, *el autómata de Büchi temporizado* $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$ *reconoce el lenguaje* $\mathcal{L}(\mathcal{P})$.

La demostración de este teorema se puede encontrar en el Apéndice A y está basada en demostrar la doble inclusión entre los lenguajes.

4.1.3 Autómata reconocedor para patrones de eventos

El autómata reconocedor para un patrón de eventos debe garantizar que además de cumplirse las restricciones de eventos y las restricciones temporales entre los puntos comunes del patrón también se cumplen las restricciones que hacen referencia al punto inicial y al punto final.

Para contemplar las restricciones de eventos con respecto al punto inicial alcanza con notar que estas restricciones comienzan estando activas y recién son desactivadas cuando se marca el otro punto en cuestión. Por el otro lado, las restricciones en las que está involucrado el punto final se activan cuando se marca el otro punto en cuestión y no se desactivan nunca más. De alguna manera, esto es consistente con el sentido que le damos a los puntos inicial y final: en el primer caso, el punto inicial se considera ya "marcado" cuando



Patrón 4.4: Restricciones activas para un patrón de eventos

comienza la ejecución y el segundo se considera que el punto final será "marcado" recién después de finalizada la ejecución.

Para capturar esta extensión de las restricciones de eventos activas en una determinada configuración definimos las funciones $\Gamma^\star$ y $\Gamma^\star_{\triangleright \mathsf{p}}$.

Definición 4.1.11. Función $\Gamma^\star$.
Dado un patrón de eventos $\mathcal{P}$, definimos la función $\Gamma^\star$ sobre configuraciones de puntos de la siguiente manera:

$$\Gamma^\star(\Theta) = \Gamma(\Theta) \cup \bigcup_{\mathsf{x} \notin \Theta} \gamma(\mathbf{0}, \mathsf{x}) \cup \bigcup_{\mathsf{x} \in \Theta} \gamma(\mathsf{x}, \infty)$$

$$\forall \mathsf{p} \in P,\ \Gamma^\star_{\triangleright \mathsf{p}}(\Theta) = \Gamma_{\triangleright \mathsf{p}}(\Theta) \cup \bigcup_{\substack{\mathsf{x} \notin \Theta \\ \mathsf{x} \neq \mathsf{p}}} \gamma(\mathbf{0}, \mathsf{x}) \cup \bigcup_{\mathsf{x} \in \Theta} \gamma(\mathsf{x}, \infty)$$

De alguna manera, $\Gamma^\star$ es una extensión de Γ. Para toda configuración Θ, $\Gamma^\star(\Theta)$ contiene todas las restricciones activas comunes (es decir, $\Gamma(\Theta)$) y además las restricciones referentes al punto inicial que todavía no han sido desactivadas y todas las restricciones referentes al punto final que ya han sido activadas.
Análogamente, $\Gamma^\star_{\triangleright \mathsf{p}}$ contiene todas las restricciones activas comunes que no serían desactivadas por p, más las restricciones referentes al punto inicial que todavía no fueron desactivadas (excepto las que serían desactivadas por p) y todas las restricciones activas referentes al punto final.

Ejemplo. *Restricciones activas para un patrón de eventos.*
Dado el Patrón 4.4, estando en la configuración $\varnothing$, $\Gamma^\star(\varnothing) = \{a\}$, $\Gamma^\star_{\triangleright \mathsf{p}}(\varnothing) = \varnothing$ y $\Gamma^\star_{\triangleright \mathsf{q}}(\varnothing) = \{a\}$. Por otro lado, estando en la configuración $\{\mathsf{p}, \mathsf{q}, \mathsf{r}\}$, $\Gamma^\star(\{\mathsf{p}, \mathsf{q}, \mathsf{r}\}) = \{c\}$. En este ejemplo se puede ver que cuando la configuración es igual al conjunto de puntos del patrón, $\Gamma^\star$ será igual al conjunto $\mathbb{L}$ del patrón.

Usaremos $\Gamma^\star$ y $\Gamma^\star_{\triangleright \mathsf{p}}$ de la misma forma que antes usábamos Γ y $\Gamma_{\triangleright \mathsf{p}}$ para determinar si es seguro saltear un punto un punto o si es seguro marcarlo, respectivamente.

El otro tema que nos queda por resolver es garantizar que se cumplan las restricciones temporales entre los puntos comunes del patrón (como hacíamos para los patrones temporizados) y también las restricciones entre el punto inicial y otros puntos del patrón. Nuevamente, adaptaremos la estrategia que usamos en los patrones temporizados contemplando este último caso. Usaremos los relojes del autómata para medir el tiempo transcurrido desde que se marcó un punto y también para medir el tiempo transcurrido desde que comenzó la ejecución. Una vez más, es como si estuviéramos midiendo el tiempo desde que se "marcó" el punto inicial del patrón. Para cada punto común p del patrón usaremos el reloj z_p para medir el tiempo desde que fue marcado. Usaremos el reloj z_0 para medir el tiempo desde el comienzo de la ejecución.

Para decidir si es seguro marcar un punto desde el punto de vista de las restricciones temporales que se apliquen a ese punto, usaremos unas guardas similares a las del caso temporizado. La única diferencia será que agregaremos un término más a cada una que corresponderá a la restricción entre ese punto y el punto inicial:

Definición 4.1.12. Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, el conjunto de relojes $X = \{z_{\mathsf{x}} \mid \mathsf{x} \in P\}$ y un conjunto de puntos $\Theta \subseteq P$, para cada punto $\mathsf{p} \in P$ definimos:

$$\psi_{\Theta}^{\star \mathsf{p}} \stackrel{def}{=} \bigwedge_{\mathsf{x} \in \Theta \cup \{\mathbf{0}\}} \psi_{z_{\mathsf{x}}}(\delta(\mathsf{x}, \mathsf{p}))$$

Un autómata reconocedor para patrones de eventos tendrá exactamente la misma estructura que uno para patrones temporizados, excepto que las condiciones para poder marcar un punto serán ampliadas para contemplar las restricciones referentes al comienzo y al final de la ejecución. Por otro lado, las condiciones para poder saltear un punto también serán ampliadas en forma conveniente. El autómata tendrá una locación $[\Theta]$ por cada configuración Θ del patrón más una locación trampa. Estando en cualquier locación el autómata podrá saltear el siguiente evento (arista *Skip*), marcar un punto sobre el siguiente evento (arista *Mark*), marcar un punto sobre el siguiente instante (arista *Instant*) o abortar la construcción del matching (arista *Fail*). Estando en la locación trampa, sólo se podrá ciclar sobre dicha locación (aristas *Trap*). El autómata tendrá un reloj por cada punto del patrón que serán reseteados cuando se marque dicho punto, más un reloj distinguido que medirá el tiempo total transcurrido desde el comienzo de la ejecución y que no será reseteado en ningún momento. Los invariantes de todas las locaciones seguirán siendo $\top$ y la locación inicial será $[\varnothing]$.

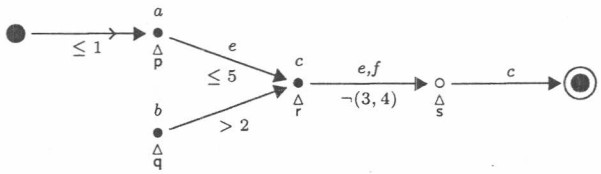
Definición 4.1.13. Autómata reconocedor para patrones de eventos.
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, $\mathcal{A}_{\mathcal{P}}$ es un autómata temporizado de la forma $\mathcal{A}_{\mathcal{P}} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$ donde:

- (*Locaciones*) $S = \{[\Theta] \mid \Theta \in \Theta_{\mathcal{P}}\} \cup \{s_{trap}\}$, donde s_{trap} es una locación distinta de todas las demás.
  Notación. Llamaremos $s_{accept} = [P]$.
- (*Relojes*) $X = \{z_{\mathsf{x}} \mid \mathsf{x} \in P\} \cup \{z_{\mathbf{0}}\}$
- (*Eventos*) $\Sigma_{\mathcal{A}_{\mathcal{P}}} = \Sigma_{\mathcal{P}}$
- (*Aristas*) $A = Mark \cup Instant \cup Skip \cup Fail \cup Trap$, donde:

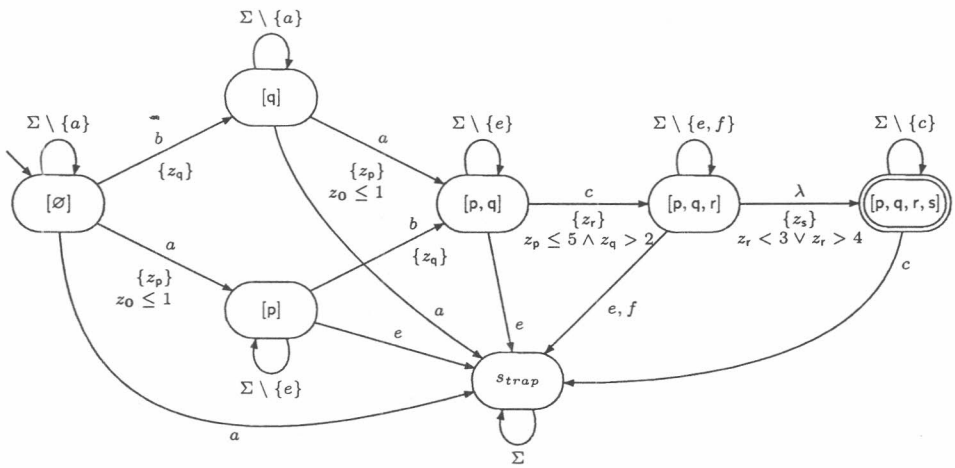
$$\begin{array}{rcll}
Mark & = & \{ \ \langle [\Theta], l, \psi_{\Theta}^{\star \mathsf{e}}, \{z_{\mathsf{e}}\}, [\Theta \uplus \{\mathsf{e}\}] \rangle & \mid \ \mathsf{e} \in E,\ l \in \ell(\mathsf{e}) \ \text{ y } \ l \notin \Gamma^{\star}_{\triangleright \mathsf{e}}(\Theta) \ \} \\
Instant & = & \{ \ \langle [\Theta], \lambda, \psi_{\Theta}^{\star \mathsf{i}}, \{z_{\mathsf{i}}\}, [\Theta \uplus \{\mathsf{i}\}] \rangle & \mid \ \mathsf{i} \in I \ \} \\
Skip & = & \{ \ \langle [\Theta], l, \top, \varnothing, [\Theta] \rangle & \mid \ l \in \Sigma \ \text{ y } \ l \notin \Gamma^{\star}(\Theta) \ \} \\
Fail & = & \{ \ \langle [\Theta], l, \top, \varnothing, s_{trap} \rangle & \mid \ l \in \Gamma^{\star}(\Theta) \ \} \\
Trap & = & \{ \ \langle s_{trap}, l, \top, \varnothing, s_{trap} \rangle & \mid \ l \in \Sigma \ \}
\end{array}$$

- (*Invariantes*) $\forall s \in S,\ \mathcal{I}(s) = \top$
- (*Locación inicial*) $s_0 = [\varnothing]$

Si bien esta definición parece exactamente igual a la Definición 4.1.8 excepto por el agregado del reloj $z_{\mathbf{0}}$ y que se reemplazó Γ, $\Gamma_{\triangleright \mathsf{p}}$ y $\psi_{\Theta}^{\mathsf{p}}$ por $\Gamma^{\star}$, $\Gamma^{\star}_{\triangleright \mathsf{p}}$ y $\psi_{\Theta}^{\star \mathsf{p}}$, respectivamente, existen más diferencias que estas entre ambos autómatas reconocedores. En particular, el cambio de Γ por $\Gamma^{\star}$ tiene un impacto bastante grande en la estructura del autómata. Sabemos que $\Gamma(P)$ es siempre vacío y esto implicaba en los casos anteriores que la locación s_{accept} no tenía aristas salientes. En este caso, $\Gamma^{\star}(P)$ no será vacío siempre que exista alguna restricción de eventos entre algún punto del patrón y el punto final. Esto resulta bastante lógico porque sabemos que esos eventos estarán prohibidos "para siempre", aún después de haber marcado el último punto.



(a)



(b)

Figura 4.3: Patrón de eventos (a) y su autómata reconocedor (b)

Conceptualmente, que $\Gamma^*(P)$ no sea vacío significa que ya no es suficiente con marcar todos los puntos del patrón para garantizar que se ha construido un matching (recordemos que la satisfacción de patrones de eventos no es cerrada por extensiones) y no alcanza con mirar ningún prefijo finito de la ejecución para hacerlo.

Ejemplo. *Autómata reconocedor para un patrón de eventos.*
La Figura 4.3 muestra un patrón de eventos y su autómata reconocedor. Se puede ver que en este caso la locación s_{accept} no es una trampa. La arista que va de $[\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}]$ a s_{trap} es un ejemplo de arista *Fail* saliente de s_{accept}. La arista que va de $[\varnothing]$ a s_{trap} por a es un ejemplo de arista *Fail* originada por una restricción de eventos entre el punto inicial y p. La guarda de la arista que va de $[\mathsf{q}]$ a $[\mathsf{p}, \mathsf{q}]$ es un ejemplo de guarda originada por una restricción temporal entre el punto inicial y el punto p. Nuevamente, el reloj z_s no se utiliza en ninguna restricción y por lo tanto podría ser eliminado.

Propiedades de los autómatas reconocedores para patrones de eventos

En el caso de los autómatas reconocedores para patrones de eventos, en general la locación s_{accept} no será una trampa. Sin embargo esto no siempre es así: basta con recordar que los patrones básicos y los patrones temporizados son casos particulares de patrones de eventos. Dado un patrón de eventos $\mathcal{P}$ y su autómata reconocedor $\mathcal{A}_\mathcal{P}$, la locación s_{accept} no será una trampa precisamente cuando el conjunto $\Gamma^*(P)$ sea no vacío. Ahora bien, por definición,

$$\Gamma^*(P) = \Gamma(P) \cup \bigcup_{x \notin P} \gamma(\mathbf{0}, x) \cup \bigcup_{x \in P} \gamma(x, \infty)$$

Sabemos que $\Gamma(P)$ es vacío y necesariamente $\bigcup_{x \notin P} \gamma(\mathbf{0}, x)$ debe ser vacío porque no quedan puntos por marcar. Quiere decir que:

$$\Gamma^*(P) = \bigcup_{x \in P} \gamma(x, \infty) = \mathbb{L}_{\mathcal{P}} \tag{4.1}$$

Es decir, los eventos que pueden provocar que el autómata abandone la locación de aceptación son exactamente los que están prohibidos hasta el final de la ejecución.

En el caso de los autómatas reconocedores que acabamos de presentar, también vale que los cinco conjuntos en los que están divididas las aristas son disjuntos dos a dos dado que la estructura de las mismas es similar a la del caso temporizado.

Observación. En todo autómata reconocedor para un patrón de eventos los conjuntos de aristas *Mark*, *Instant*, *Skip*, *Fail* y *Trap* son disjuntos dos a dos.

Y también se mantiene una de las propiedades más importantes de los autómatas reconocedores: que las evoluciones del autómata siempre van extendiendo la configuración actual o a lo sumo la mantienen igual, pero nunca la reducen.

Propiedad 4.11. *Dado un patrón de eventos* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ *y su autómata reconocedor* $\mathcal{A}_{\mathcal{P}} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, *sean* Θ *y* Θ' *configuraciones de* $\mathcal{P}$. *Si* $[\Theta] \Rightarrow [\Theta']$, *entonces toda evolución entre* $[\Theta]$ *y* $[\Theta']$ *tiene la forma:*

$$([\Theta_0], v_0) \Rightarrow_{t_0}^{a_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{a_1} \ldots ([\Theta_i], v_i) \Rightarrow_{t_i}^{a_i} \ldots \Rightarrow_{t_{n-1}}^{a_{n-1}} ([\Theta_n], v_n)$$

donde $\Theta_0 = \Theta$, $\Theta_n = \Theta'$ *y para todo* i, $0 \leq i < n$, $v_i \in \mathcal{V}_X$, $a_i \in \Sigma \cup \{\lambda\}$, $t_i \in \Re^+$ *y* Θ_{i+1} ***es igual a*** Θ_i ***o es una extensión de*** Θ_i.

Extendemos los conceptos de *matching básico parcial* y *matching temporizado parcial* al caso de los patrones de eventos.

Definición 4.1.14. Matching parcial.
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, un conjunto de puntos $C \subseteq P$, una ejecución temporizada $\sigma = \langle \varsigma, \tau \rangle$ sobre Σ y un mapeo $\hat{\cdot} : C \mapsto \Pi_\sigma$, decimos que $\hat{\cdot}$ es un *matching parcial* entre σ y $\mathcal{P}$ restringido a C sii verifica las condiciones **MP1-MP4 + MPT1-MPT2** y además se verifica que:

MPI	$\forall x \in C,$	$\sigma_{\hat{x}}) \cap \gamma(\mathbf{0}, x) = \varnothing$
MPS	$\forall x \in C,$	$\sigma_{(\hat{x}} \cap \gamma(x, \infty) = \varnothing$
MPTI	$\forall x \in C,$	$\Delta(\tau_{\hat{x}]}) \vDash \delta(\mathbf{0}, x)$

Un *matching parcial* caracteriza la forma en que queremos que funcione nuestro autómata: a medida que va marcando puntos, debe cumplir con todas las condiciones que sean necesarias para que el mapeo sea finalmente un matching. **M3 + MPI** garantizan que los puntos marcados no violen ninguna restricción de eventos que existiera entre ellos (incluyendo al punto inicial que siempre está "marcado"). **MPT + MPTI** garantizan lo mismo pero para las restricciones temporales y, finalmente, **MP4 + MPS** garantizan que ninguna posición consumida de la ejecución violara una restricción de eventos activa. La diferencia entre **MP4** y **MPS** es que la primera se refiere a restricciones de eventos que eventualmente serán desactivadas mientras que la segunda hace referencia a restricciones que siempre permanecerán activas, es decir, a eventos en el conjunto $\mathbb{L}_{\mathcal{P}}$.

Una vez más, a partir de matching parcial podemos definir una nueva noción de satisfacción parcial:

Definición 4.1.15. Satisfacción parcial de patrones de eventos.
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ y un conjunto de puntos $C \subseteq P$, una ejecución temporizada σ sobre Σ *verifica* o *satisface parcialmente* $\mathcal{P}$ *restringido a* C (notado $\sigma \models_C \mathcal{P}$) sii existe un matching parcial entre σ y $\mathcal{P}$ restringido a C.

En este caso, como en los anteriores, para cualquier patrón de eventos $\mathcal{P}$ y cualquier ejecución temporizada σ, $\sigma \models \mathcal{P}$ si y sólo si $\sigma \models_P \mathcal{P}$.

Planteamos y demostramos una vez más las propiedades de Completitud y Corrección del autómata reconocedor, en este caso, para patrones de eventos.

Propiedad 4.12. *Corrección del autómata reconocedor para patrones de eventos.*
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ *y su autómata reconocedor* $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, *para toda configuración* $\Theta \in \Theta_\mathcal{P}$ *y toda ejecución finita* $\sigma = \langle \varsigma, \tau \rangle$ *sobre* Σ,

$$si\ q_{init} \Rightarrow_\tau^\varsigma [\Theta]\ entonces\ \sigma \models_\Theta \mathcal{P}$$

Intuitivamente, se puede ver que esta propiedad debe valer en el caso de los patrones de eventos tanto como valía para el caso temporizado. Dado que podemos asimilar las restricciones referentes al punto inicial con el resto de las restricciones entre puntos comunes asumiendo que el punto inicial esta siempre "marcado" y además podemos asimilar las restricciones referentes al punto final con el resto de las restricciones asumiendo que el punto final nunca será "marcado", el funcionamiento de este autómata es muy similar al del caso temporizado, al menos en lo que se refiere a consumir prefijos finitos de una ejecución. La demostración completa se encuentra en el Apéndice A.

Propiedad 4.13. *Completitud del autómata reconocedor para patrones de eventos.*
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ *y su autómata reconocedor* $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, *para toda configuración* $\Theta \in \Theta_\mathcal{P}$ *y toda ejecución finita* σ *sobre* Σ,

$$si\ \sigma \models_\Theta \mathcal{P}\ entonces\ q_{init} \Rightarrow^\sigma [\Theta]$$

Nuevamente, si tratamos a las restricciones con respecto al punto inicial simplemente como restricciones que empiezan estando activas y eventualmente serán desactivadas y tratamos a las restricciones con respecto al punto final como restricciones que empiezan estando desactivadas y eventualmente serán activadas, es fácil ver que el funcionamiento de estos autómatas reconocedores es muy similar al de los autómatas del caso temporizado. La demostración sigue la misma línea que en los dos casos anteriores: muestra como construir una evolución sobre $\mathcal{A}_\mathcal{P}$ que consuma la ejecución y que llegue hasta $[\Theta]$ a partir de un matching parcial. La demostración completa se encuentra en el Apéndice A.

Dado que sabemos que el autómata reconocedor verifica las propiedades de Completitud y Corrección, podemos garantizar que el autómata captura todas las formas posibles de marcar los puntos del patrón de eventos asociado sobre un prefijo finito de una ejecución. En el caso de los patrones básicos o de los patrones temporizados, esto equivalía a garantizar que capturaba todos los matchings. En el caso de los patrones de eventos, como no son cerrados por extensiones, esta conclusión no es tan directa. Sin embargo, veremos a continuación que agregando la condición de aceptación de Büchi $\{s_{accept}\}$ al autómata reconocedor, el ω-autómata resultante captura exactamente el lenguaje del patrón.

Tableau para patrones de eventos

Teorema 4.14. *Tableau para patrones de eventos. Dado un patrón de eventos* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, *el autómata de Büchi temporizado* $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$ *reconoce el lenguaje* $\mathcal{L}(\mathcal{P})$.

La condición de aceptación de Büchi $\{s_{accept}\}$ exige que se visite un número infinito de veces s_{accept}. Sabemos que los autómatas reconocedores para patrones de eventos no tienen ciclos más allá de los loops sobre una misma locación. Esto quiere decir que el autómata estará obligado a permanecer para siempre en s_{accept} para cumplir con la condición de aceptación. En particular, no podrá fallar por ningún evento (es decir, consumir un evento prohibido) porque eso le impediría volver a s_{accept}.

En el Apéndice A se demuestra la doble inclusión entre $\mathcal{L}(\mathcal{T}_\mathcal{P})$ y $\mathcal{L}(\mathcal{P})$.

4.2 Model-checking de patrones de mal comportamiento

Ya hemos mostrado como construir un autómata reconocedor y a partir de éste un tableau para cada uno de los tres tipos de patrones de eventos. En particular, la construcción hecha para los patrones básicos y los temporizados son casos particulares de la construcción para patrones de eventos.

Dado un sistema S modelado con un autómata temporizado $\mathcal{A}_S$ y dado un requerimiento R del sistema, donde el patrón de mal comportamiento $\mathcal{P}_{\neg R}$ captura los comportamientos que no cumplen R, el problema de *model-checking de patrones de mal comportamiento* consiste en decidir si $\mathcal{A}_S \models \mathcal{P}_{\neg R}$. Dado que el patrón captura comportamientos *no deseados* en el sistema, alcanza con que alguna de las ejecuciones del lenguaje $\mathcal{L}(\mathcal{A}_S)$ satisfaga el patrón para concluir que el sistema viola el requerimiento R. Formalmente, definimos la relación $\models$ entre un autómata temporizado y un patrón de mal comportamiento cómo:

Definición 4.2.1. Satisfacción de patrones de mal comportamiento.
Dado un autómata temporizado $\mathcal{A} = \langle S, X, \Sigma', A, \mathcal{I}, s_0 \rangle$ y el patrón de mal comportamiento $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, con $\Sigma \subseteq \Sigma'$,

$$\mathcal{A} \models \mathcal{P} \quad \text{sii} \quad \mathcal{L}(\mathcal{A}) \cap \mathcal{L}_{\Sigma'}(\mathcal{P}) \neq \varnothing$$

El alfabeto de un patrón de mal comportamiento (en este caso Σ) podría ser mas reducido que el alfabeto del sistema (en este caso Σ'). Esto podría darse porque el patrón podría hacer referencia a un numero relativamente pequeño de eventos en el sistema. Si éste es el caso, alcanza con considerar el lenguaje *ampliado* de $\mathcal{P}$ según el alfabeto Σ' (Definición 3.3.5). Recordemos que los eventos que no son explícitamente mencionados por un patrón pueden ser ignorados a la hora de decidir satisfacción.

Nuestro algoritmo de model-checking no está basado en la definición anterior. Reescribiremos el problema de la intersección de lenguajes de alguna forma que nos resulte más conveniente para realizar el chequeo. En la sección anterior mostramos que dado un patrón de mal comportamiento el tableau $\mathcal{T}_\mathcal{P}$ captura exactamente el lenguaje del patrón. Veremos ahora que, además, al componer el autómata $\mathcal{A}$ con el tableau del patrón $\mathcal{P}$, el lenguaje del autómata (de Büchi) obtenido es distinto de vacío sólo cuando $\mathcal{A} \models \mathcal{P}$. De alguna manera, el tableau del patrón puede pensarse como un *Büchi Observer* al estilo de los presentados en [Bra00]. Dado que $\mathcal{T}_\mathcal{P}$ tiene aristas por todos los eventos y todos los invariantes de sus locaciones son $\top$, podemos decir que $\mathcal{T}_\mathcal{P}$ "acompaña" a $\mathcal{A}$. Es decir, $\mathcal{T}_\mathcal{P}$ no restringirá ninguna de las evoluciones posibles en $\mathcal{A}$. Simplemente, en paralelo con cada una de esas evoluciones el observador intentará construir un matching para ejecución expuesta por la evolución. Si el observador logra alcanzar s_{accept} y permanecer indefinidamente en dicha locación significará que ha logrado construir un matching para alguna ejecución aceptada por $\mathcal{A}$ y, por lo tanto, $\mathcal{A}$ verificaba el patrón $\mathcal{P}$.

Teorema 4.15. *Dado un autómata temporizado* $\mathcal{A} = \langle S, X, \Sigma', A, \mathcal{I}, s_0 \rangle$ *y un patrón de mal comportamiento* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, *con* $\Sigma \subseteq \Sigma'$,

$$\mathcal{A} \vDash \mathcal{P} \qquad sii \qquad \mathcal{L}(\mathcal{A} \otimes \mathcal{T}_{\mathcal{P}}) \neq \varnothing$$

Demostración. Por la definición de satisfacción de patrones de eventos, sabemos que $\mathcal{A} \vDash \mathcal{P}$ si y sólo si $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}_{\Sigma'}(\mathcal{P}) \neq \varnothing$, es decir, si y sólo si existe una ejecución σ sobre Σ' tal que $\sigma \in \mathcal{L}(\mathcal{A})$ y $\sigma\lceil_{\Sigma} \in \mathcal{L}(\mathcal{P})$. Por el Teorema Tableau 4.14, sabemos que $\sigma\lceil_{\Sigma} \in \mathcal{L}(\mathcal{P})$ si y sólo si $\sigma\lceil_{\Sigma} \in \mathcal{L}(\mathcal{T}_{\mathcal{P}})$.
Quiere decir que $\mathcal{A} \vDash \mathcal{P}$ si y sólo si existe alguna ejecución σ sobre Σ' en $\mathcal{L}(\mathcal{A} \otimes \mathcal{T}_{\mathcal{P}})$ (Observación 2.2), es decir, si y sólo si $\mathcal{L}(\mathcal{A} \otimes \mathcal{T}_{\mathcal{P}}) \neq \varnothing$.

□

Como dijimos antes, el observador acompaña las evoluciones del autómata $\mathcal{A}$ y sabremos que ha encontrado una ejecución de $\mathcal{A}$ que satisface el patrón si logra alcanzar locaciones compuestas con s_{accept} y, además, logra permanecer indefinidamente en dichas locaciones (recordemos que en $\mathcal{A}_{\mathcal{P}}$ no se puede volver a s_{accept} una vez dejada esa locación). Supongamos que asociamos a todas las locaciones compuestas con s_{accept} la proposición *accept*. La fórmula TCTL $\exists\Diamond accept$ dice que *accept* es alcanzable por al menos una evolución. Si evaluamos esa fórmula en el estado inicial del autómata compuesto, la fórmula será verdadera si y sólo si el observador logra marcar todos los puntos del patrón sobre alguna ejecución del autómata. Sabemos que esto es una condición necesaria para que la ejecución satisfaga el patrón, pero no suficiente (propiedad de Clausura débil por extensiones). Tenemos que garantizar, además, que el observador pueda permanecer indefinidamente en locaciones donde valga *accept*. Esto también puede expresarse en TCTL como $\exists\Box accept$, es decir, existe al menos una evolución en la cual siempre vale *accept*. Quiere decir que si la fórmula $\exists\Diamond(\exists\Box accept)$ valiera evaluada en el estado inicial del autómata compuesto, indicaría que necesariamente alguna de las ejecuciones de $\mathcal{A}$ satisface el patrón de mal comportamiento. Por lo tanto, obtenemos el siguiente corolario a partir del teorema anterior:

Corolario 4.16. *(Model-checking de patrones de mal comportamiento)*
Dado un autómata temporizado $\mathcal{A} = \langle S, X, \Sigma', A, \mathcal{I}, s_0 \rangle$ *y un patrón de mal comportamiento* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, *con* $\Sigma \subseteq \Sigma'$,

$$\mathcal{A} \vDash \mathcal{P} \qquad sii \qquad \mathcal{A} || \mathcal{A}_{\mathcal{P}} \vDash init \Rightarrow \exists\Diamond(\exists\Box accept)$$

donde $\mathbf{P}(accept) = S \times \{s_{accept}\}$ *y* $\mathbf{P}(init) = \{(s_{0_{\mathcal{A}}}, s_{0_{\mathcal{A}_{\mathcal{P}}}})\}$.

Quiere decir que hemos reducido nuestro problema de model-checking de patrones de mal comportamiento a un problema de model-checking de autómatas temporizados, que se sabe decidible (ver por ejemplo [HNSY92]). Con lo cual como segundo corolario tenemos que:

Corolario 4.17. *(Decidibilidad de model-checking de patrones de mal comportamiento)*
Dado un autómata temporizado $\mathcal{A} = \langle S, X, \Sigma', A, \mathcal{I}, s_0 \rangle$ *y un patrón de mal comportamiento* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, *con* $\Sigma \subseteq \Sigma'$, *el problema de determinar si* $\mathcal{A} \vDash \mathcal{P}$ *es* decidible.

Más adelante veremos que la reducción del chequeo de patrones a model-checking de autómatas temporizados sirve además como un método práctico para construir un algoritmo de verificación de patrones. El esquema general de ese algoritmo de verificación fue esbozado en la Figura 1.5.

Capítulo 5

Casos de estudio

5.1 Sistema *Mine Drainage Controller*

El primer caso de estudio que analizaremos corresponde a un ejemplo bien conocido basado en un caso real: el diseño de un controlador para un sistema de extracción de agua de una mina. Este ejemplo es comúnmente referenciado en la literatura (por ejemplo en [M.J96, BW96]) e ilustra muchas de las características que poseen los sistemas de tiempo real embebidos. La versión sobre la cual trabajamos es una extensión presentada en [Bra00].

5.1.1 Descripción del Sistema

El sistema es utilizado para bombear el agua que se colecta en la base de una mina y llevarla a la superficie. Para evitar el peligro de explosión, la bomba no debe ser operada cuando el nivel de gas metano en la mina alcanza cierto valor crítico.
Los valores del ambiente (flujo de aire, monóxido de carbono, flujo de agua) son leídos periódicamente. El nivel del agua (alto / bajo) es comunicado a través de interrupciones.
El objeto protegido *Motor* provee los servicios para operar la bomba y observar el status del motor. El objeto protegido *CH4Status* conserva el valor de la última lectura del gas metano.
Cuando hay una situación de riesgo (el nivel de gas o el flujo de aire se vuelven críticos, el nivel del flujo de agua leído no coincide con el status del motor, etc) se informa una alarma al objeto protegido *Operator Console*, para ser eventualmente señalizada a una consola remota en donde está el operador. Las operaciones y lecturas son registradas en el objeto *Log*. Existe una tarea esporádica *Command* utilizada para atender las solicitudes del operador remoto. Estas solicitudes son: inspeccionar el status del motor, prender o apagar la bomba.
Los sensores de CO y CH4 usan la técnica "desplazamiento periódico" para realizar las lecturas: solicitan una lectura que debería estar disponible en el siguiente período (si no se produciría una alarma).
Se adiciona al sistema otro mecanismo de detección de fallas que consiste en una tarea *watchdog* que chequea periódicamente la disponibilidad del sensor de nivel de agua. Primero envía una solicitud y después extrae los *acknowledges* recibidos y encolados por una tarea esporádica en los ciclos anteriores. Si el watchdog encuentra la cola de ACKs vacía, lo informa como una situación errónea.
La Figura 5.1 muestra el diseño del sistema usando una notación similar a la utilizada en [BW95].

5.1.2 Requerimientos

Nos interesa verificar que el diseño anteriormente descripto cumple con una serie de requerimientos que detallamos a continuación. Estos requerimientos están extraídos de [Bra00]. Para cada uno de ellos, mostramos un patrón de mal comportamiento que captura los escenarios en los que no se cumple el requerimiento.

Requerimiento 5.1. *(Separación) Dos lecturas consecutivas del Water-flow Sensor deben estar separadas entre 960 u.t. y 1040 u.t.*

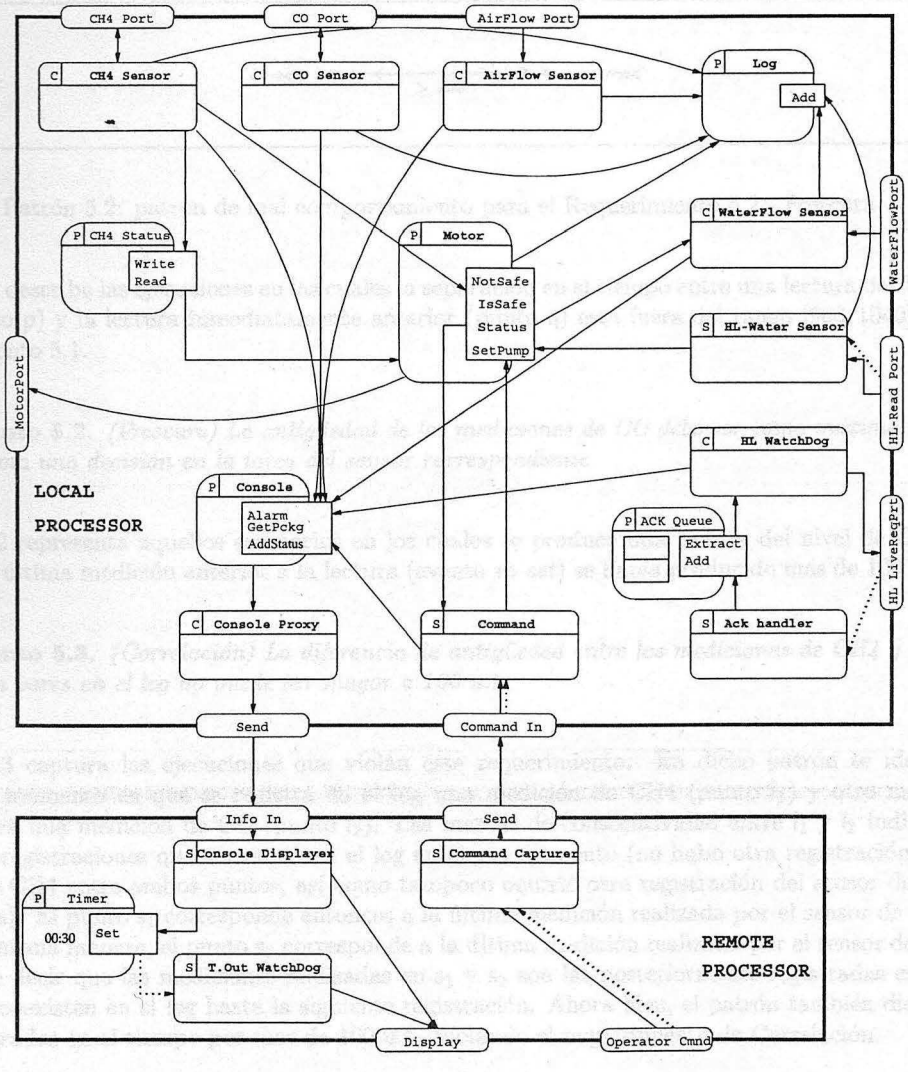
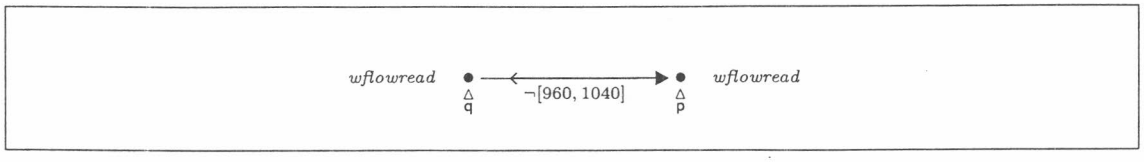
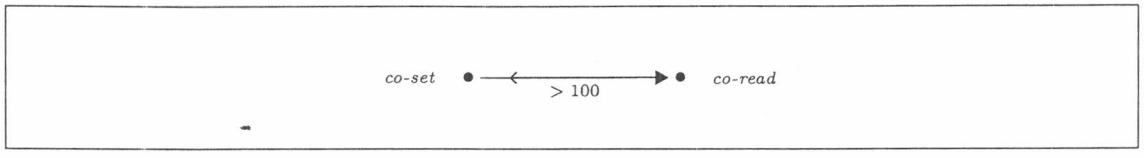


Figura 5.1: Diseño del sistema de *Mine Drainage*



Patrón 5.1: patrón de mal comportamiento para el Requerimiento 5.1 - Separación



Patrón 5.2: patrón de mal comportamiento para el Requerimiento 5.2 - Frescura

El Patrón 5.1 describe las ejecuciones en las cuales la separación en el tiempo entre una lectura del *Water-flow Sensor* (punto p) y la lectura inmediatamente anterior (punto q) está fuera del rango [960, 1040], violando el Requerimiento 5.1.

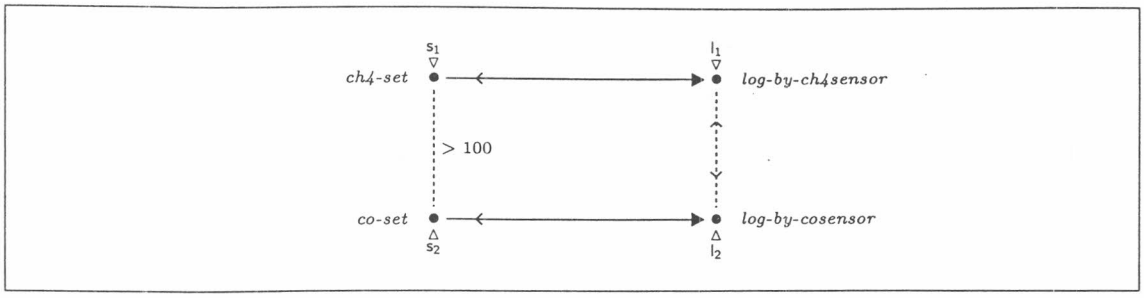
Requerimiento 5.2. *(Frescura) La antigüedad de las mediciones de CO debe ser como máximo de 100 u.t. cuando se toma una decisión en la tarea del sensor correspondiente.*

El Patrón 5.2 representa aquellos escenarios en los cuales se produce una lectura del nivel de CO (evento *co-read*) y la última medición anterior a la lectura (evento *co-set*) se había producido más de 100 u.t. antes.

Requerimiento 5.3. *(Correlación) La diferencia de antigüedad entre las mediciones de CH4 y CO que se registran de a pares en el log no puede ser mayor a 100 u.t..*

El Patrón 5.3 captura las ejecuciones que violan este requerimiento. En dicho patrón se identifica un determinado momento en que se registra en el log una medición de CH4 (punto l_1) y otro momento en que se registra una medición de CO (punto l_2). Las marcas de consecutividad entre l_1 y l_2 indican que se trata de dos registraciones que coexisten en el log en cierto momento (no hubo otra registración por parte del sensor de CH4 entre ambos puntos, así como tampoco ocurrió otra registración del sensor de CO entre dichos puntos). El punto s_1 corresponde entonces a la última medición realizada por el sensor de CH4 antes de l_1. De la misma manera, el punto s_2 corresponde a la última medición realizada por el sensor de CO antes de l_2. Quiere decir que las mediciones realizadas en s_1 y s_2 son las posteriormente registradas en l_1 y l_2, y por lo tanto coexisten en el log hasta la siguiente registración. Ahora bien, el patrón también dice que s_1 y s_2 están separados en el tiempo por mas de 100 u.t., violando el requerimiento de Correlación.

El ejemplo anterior evidencia la capacidad de abstracción que brindan los patrones de eventos para expresar relaciones de causalidad complejas entre eventos. El hecho de no haber tenido que precisar el orden de ocurrencia entre s_1 y s_2 o entre l_1 y l_2 permite una representación compacta y elegante de la propiedad de correlación. A modo de comparación, el autómata observador para el Requerimiento 5.3 presentado en [Bra00] contaba con 11 locaciones y 24 aristas.



Patrón 5.3: patrón de mal comportamiento para el Requerimiento 5.3 - Correlación

5.2 Protocolo CSMA/CD

El segundo caso de estudio que analizaremos está basado en el protocolo de comunicación CSMA/CD [Tan96, IEE85], usado para controlar el uso de un bus compartido en una red *broadcast*. La descripción del sistema que utilizamos sigue la línea de [XJS92].

5.2.1 Descripción del sistema

El protocolo CSMA/CD (*Carrier Sense, Multiple Access with Collision Detection*) es muy utilizado en LANs en la capa MAC (control de acceso al medio físico). Resuelve el problema de compartir un único canal en una red broadcast (canal *multi-access*). Cuando una estación tiene datos para enviar, primero escucha el canal para determinar si está siendo usado por otra estación. Si el bus parece no estar siendo utilizado, la estación comienza a enviar el mensaje. Si el bus está ocupado, la estación espera una cantidad de tiempo aleatoria y repite el proceso. Cuando ocurre una colisión, la transmisión se aborta en forma simultánea por todas las estaciones que estuvieran transmitiendo en ese momento y todas esperan una cantidad aleatoria de tiempo antes de reintentar el envío.

Las estaciones comparten un único canal. Asumimos que el canal es un bus Ethernet de 10Mbps, con tiempo de propagación de $\sigma = 26\,\mu$s en el peor caso. Los mensajes tienen un tamaño fijo de 1024 bytes y el tiempo para enviar un mensaje completo, incluyendo el tiempo de propagación es, entonces, de $808\,\mu$s. El bus es libre de errores, no se realiza buffering de mensajes entrantes. La señal de colisión tarda a lo sumo σ para llegar a todos los emisores.

La Figura 5.2 muestra la estructura del sistema para el protocolo CSMA/CD. Cada caja representa un componente del sistema. Las líneas representan la sincronización entre los componentes. Cada uno de los *senders* y el *bus* se sincronizan mediante los siguientes eventos:

- $begin_i$ — El $sender_i$ comienza a enviar un mensaje (el bus no estaba ocupado)
- $busy_i$ — El $sender_i$ encuentra el bus ocupado
- end_i — El $sender_i$ completa la transmisión del mensaje
- cd_i — El $sender_i$ detecta una colisión

5.2.2 Requerimientos

Requerimiento 5.4. *Si uno de los senders comienza a transmitir mientras el otro está transmitiendo, ambos deben recibir la notificación de colisión antes de que alguno de los dos finalice con éxito la transmisión.*

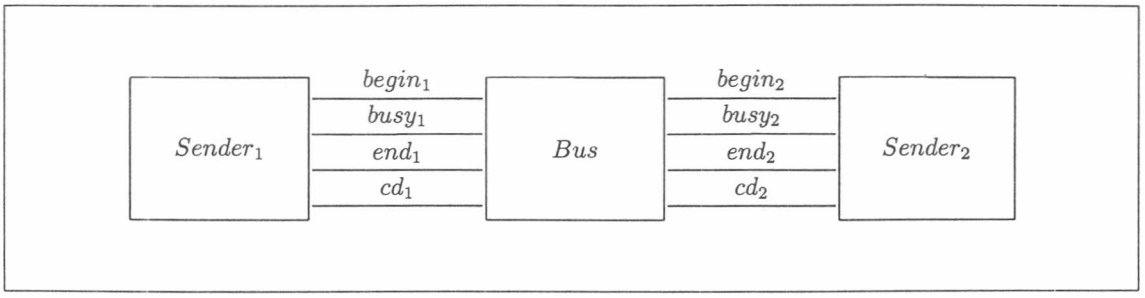
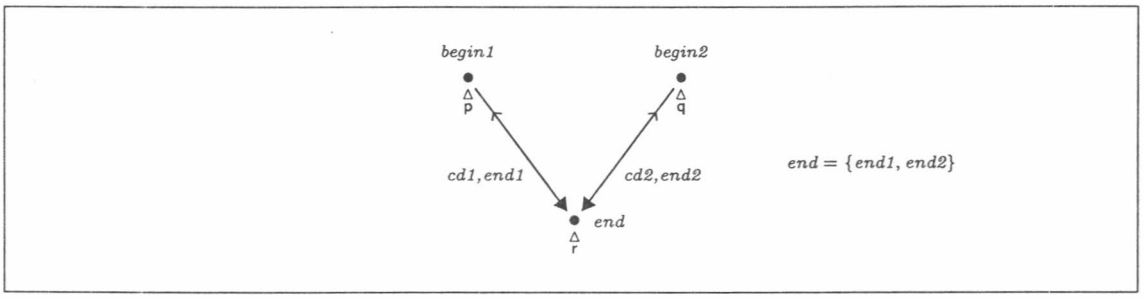


Figura 5.2: Estructura del sistema para el protocolo CSMA/CD



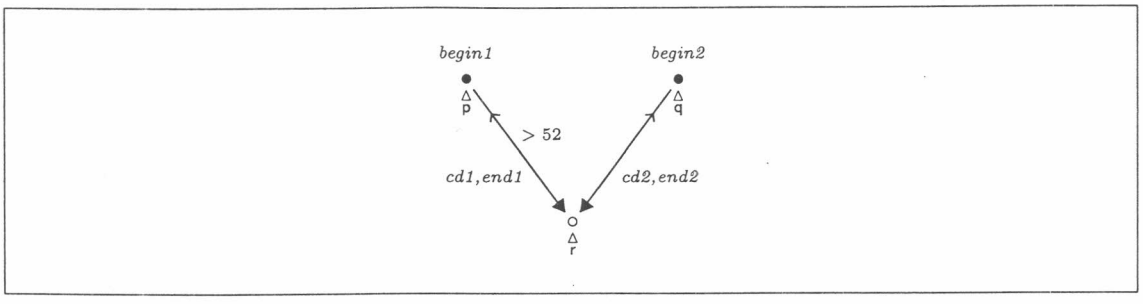
Patrón 5.4: patrón de mal comportamiento para el Requerimiento 5.4

El Requerimiento 5.4 resume una de las propiedades más importantes del protocolo CSMA/CD: en ningún momento una colisión puede pasar inadvertida para alguno de los senders de forma tal de que crea que ha finalizado con éxito la transmisión de su mensaje. El Patrón 5.4 representa los escenarios en los que se viola esta propiedad. El punto r corresponde a alguno de los dos senders finalizando (aparentemente con éxito) la transmisión de su mensaje. "Retrocediendo" en la ejecución encontramos que en el punto p el $sender_1$ había comenzado a transmitir un mensaje. Dado que entre p y r que dicho sender no detectó ninguna colisión ni terminó de enviar su mensaje (no hubo ningún evento cd_1 ni end_1), quiere decir que el $sender_1$ continuó enviando su mensaje al menos hasta r. Por el otro lado, en el punto q, el $sender_2$ también comenzó a enviar un mensaje. Dado que entre q y r el $sender_2$ tampoco dejo de transmitir, el segundo sender también se encontraba transmitiendo al menos hasta el punto r. Quiere decir que los dos senders transmitieron, al menos por un período de tiempo, simultáneamente y alguno de los dos terminó la transmisión con la conclusión errónea de que había sido exitosa.

Una vez más, la posibilidad de no tener que fijar arbitrariamente el orden de ocurrencia de todos los eventos (es decir, la posibilidad de especificar un orden parcial entre los mismos) nos permitió escribir en forma compacta el Patrón 5.4. Este patrón captura todas las combinaciones posibles sobre cual de los senders comienza a transmitir primero y, además, cual de los senders termina de transmitir primero.

Requerimiento 5.5. *Si uno de los senders comienza a transmitir mientras el otro estaba transmitiendo, ambos deben recibir la notificación de colisión dentro de los primeros* $52\,\mu s$ *de transmisión.*

Este requerimiento es otra de las propiedades fuertes del protocolo CSMA/CD. Dado que estamos asumiendo que el tiempo de propagación es en el peor caso de $\sigma = 26\,\mu s$, desde que un sender comienza a transmitir hasta que el otro sender percibe que el bus está ocupado podrían pasar como máximo $26\,\mu s$. Si este fuera el caso, querría decir que el primer sender notaría la colisión aproximadamente $26\,\mu s$ más tarde, es decir, a lo sumo $52\,\mu s$ después de que comenzara a transmitir. Pasados los $26\,\mu s$ iniciales el otro sender directamente



Patrón 5.5: patrón de mal comportamiento para el Requerimiento 5.5

no podría comenzar a transmitir porque notaría que el bus estaba siendo utilizado. Quiere decir que cuando un sender comienza a transmitir, o bien recibe una señal de colisión dentro de los primero $52\,\mu s$ o puede estar seguro de que terminará de transmitir exitosamente (recordemos que estamos asumiendo que el canal está libre de errores).

El Patrón 5.5 muestra los casos en los cuales el Requerimiento 5.5 sería violado. El punto r representa un instante dentro de la ejecución con la siguiente propiedad: tanto el $sender_1$ como el $sender_2$ se encuentran transmitiendo y además el $sender_1$ estuvo transmitiendo sin percibir la colisión por más de $52\,\mu s$. Implícitamente, el patrón asume que el $sender_1$ fue el que comenzó primero la transmisión. Para cubrir todos los casos tendríamos que verificar también otro patrón simétrico a éste en el cual $sender_2$ fuera el que comenzó primero. Sin embargo, dado que el sistema es simétrico con respecto a los senders (ninguno tiene propiedades distintas del otro), si se da el comportamiento no deseado asumiendo que comienza siempre el $sender_1$, necesariamente debe también se violará el requerimiento asumiendo que comienza el $sender_2$. Y también es cierta la propiedad recíproca: si se demuestra que que el sistema es correcto con respecto al Requerimiento 5.5 asumiendo que comienza el $sender_1$, también será correcto si asumiéramos que el $sender_2$ es el que comienza.

Más allá de esta simetría entre los senders en el modelo del protocolo CSMA/CD, acabamos de identificar una limitación del lenguaje basado en patrones de eventos. Para solucionar este problema, sería interesante que el lenguaje permitiera hablar de "el último de los eventos ocurridos" o el "primero de los eventos ocurridos" en un cierto conjunto. Por ejemplo en este caso quisiéramos decir que sin importar si $begin_1$ ocurre antes que $begin_2$ o viceversa, el tiempo transcurrido entre el que haya ocurrido primero y el punto r sea mayor a $52\,\mu s$. Entendemos que la incorporación de estos elementos al lenguaje aumentará el poder expresivo del mismo y por eso mismo está incluida como una de las extensiones que forman parte de nuestra propuesta de trabajo futuro.

5.3 Analizador de variables ambientales

Este último caso de estudio que analizaremos está basado en un hipotético sistema de tiempo real distribuido. Si bien no representa un caso real, comparte las características de muchos sistemas de este tipo y nos permitirá mostrar el poder expresivo de los patrones de mal comportamiento para expresar propiedades de concurrencia y causalidad.

5.3.1 Descripción del sistema

El sistema que analizaremos constituye un sistema de tiempo real distribuido, compuesto por un nodo central de monitoreo y dos nodos dedicados al muestreo y preprocesamiento de variables ambientales. Los nodos de muestreo tienen una tarea periódica que lee los datos del sensor, los procesa y escribe el resultado en una variable protegida. Otra tarea esporádica se dedica a recibir pedidos del exterior, empaquetar y enviar los últimos datos procesados y guardarlos en la variable compartida.

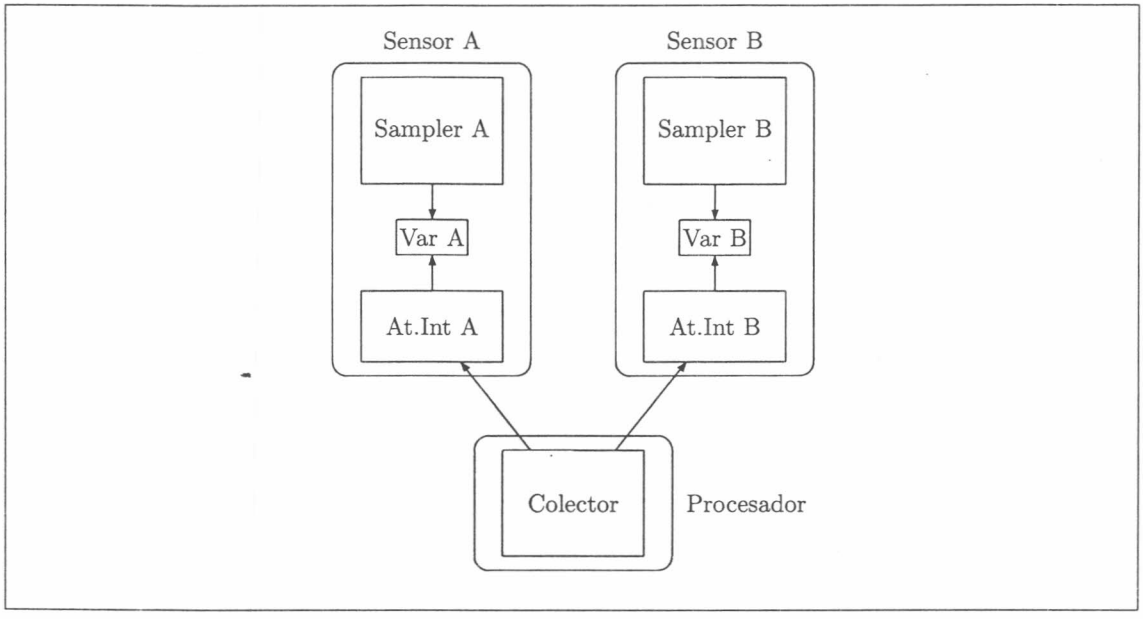


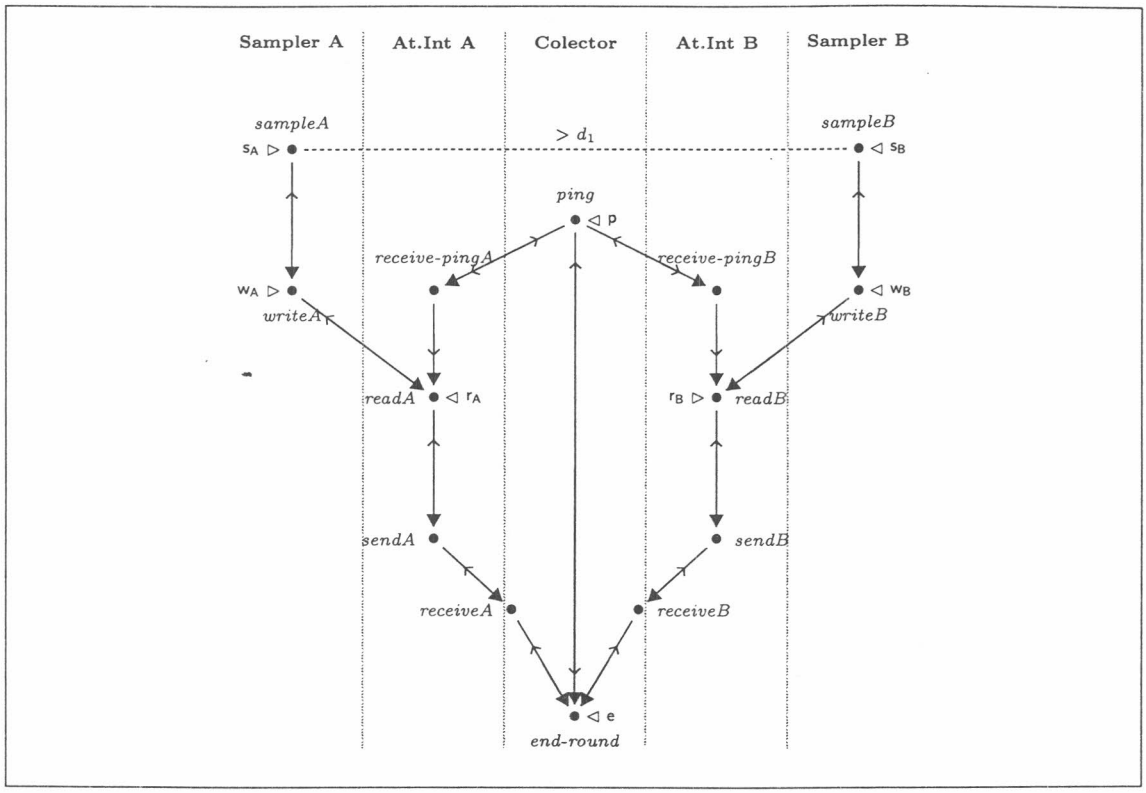
Figura 5.3: Estructura del Analizador de variables ambientales

La Figura 5.3 muestra la estructura del sistema. El nodo central, en caso de que se le haga un requerimiento comienza su tarea de recolección de datos muestreados enviando el pedido a ambas estaciones. Luego, espera ambas respuestas y aparea los datos para finalizar su tarea. Todas estas acciones forman parte de una *ronda de muestreo*. Se sabe que los pedidos y sus respuestas no son almacenados en un buffer.

5.3.2 Requerimientos

Requerimiento 5.6. *(Correlación) La antigüedad de los datos apareados no difiere en más de* d_1 ms.

El sistema toma decisiones en base a los valores medidos por el sensor A y el sensor B. Es importante por lo tanto que las mediciones no difieran mucho en antigüedad para que la decisión tomada por el sistema pueda ser considerada válida. El Patrón 5.6 describe las ejecuciones que violarían este requerimiento. El patrón identifica los puntos p y e que corresponden al inicio y al final de una misma ronda de muestreo. A partir del momento en que termina dicha ronda (punto e), podemos "retroceder" en la ejecución identificando los momentos en que se recibió el valor sampleado por A (evento *receiveA*), dicho valor fue enviado por el sensor A (evento *sendA*), el valor fue leído desde la variable compartida (evento *readA*) y finalmente el momento en que se recibió el request desde el colector (evento $receive - pingA$). De la misma manera, se pueden identificar los eventos correspondientes al sensor B, desde que se recibe el ping del colector hasta que este último recibe el dato. Se identifican todos estos puntos empezando desde la finalización de la ronda (punto e) y contextualizándolos con el inicio de la misma (punto p) para garantizar que corresponden todos a la misma ronda y que no hay posibilidad de tomar eventos de rondas anteriores. De esta forma sabemos que los puntos r_A y r_B corresponden a las lecturas de las mediciones de los sensores A y B, respectivamente, usados por el procesador al final de la ronda. A partir de r_A y r_B podemos identificar la última escritura anterior a sendas lecturas (puntos w_A y w_B, respectivamente). A su vez, a partir de w_A y w_B se puede identificar el momento en que fueron sampleados los valores que luego fueron escritos (últimas ocurrencias de *sampleA* y *sampleB* antes de las escrituras). Quiere decir que los valores sampleados en s_A y s_B son los que finalmente fueron apareados al finalizar la ronda. El patrón describe, por lo tanto, todos aquellos casos donde s_A y s_B ocurrieron con más de d_1 ms de separación y por lo tanto no se cumple el requerimiento de correlación.



Patrón 5.6: patrón de mal comportamiento para el Requerimiento 5.6 - Correlación

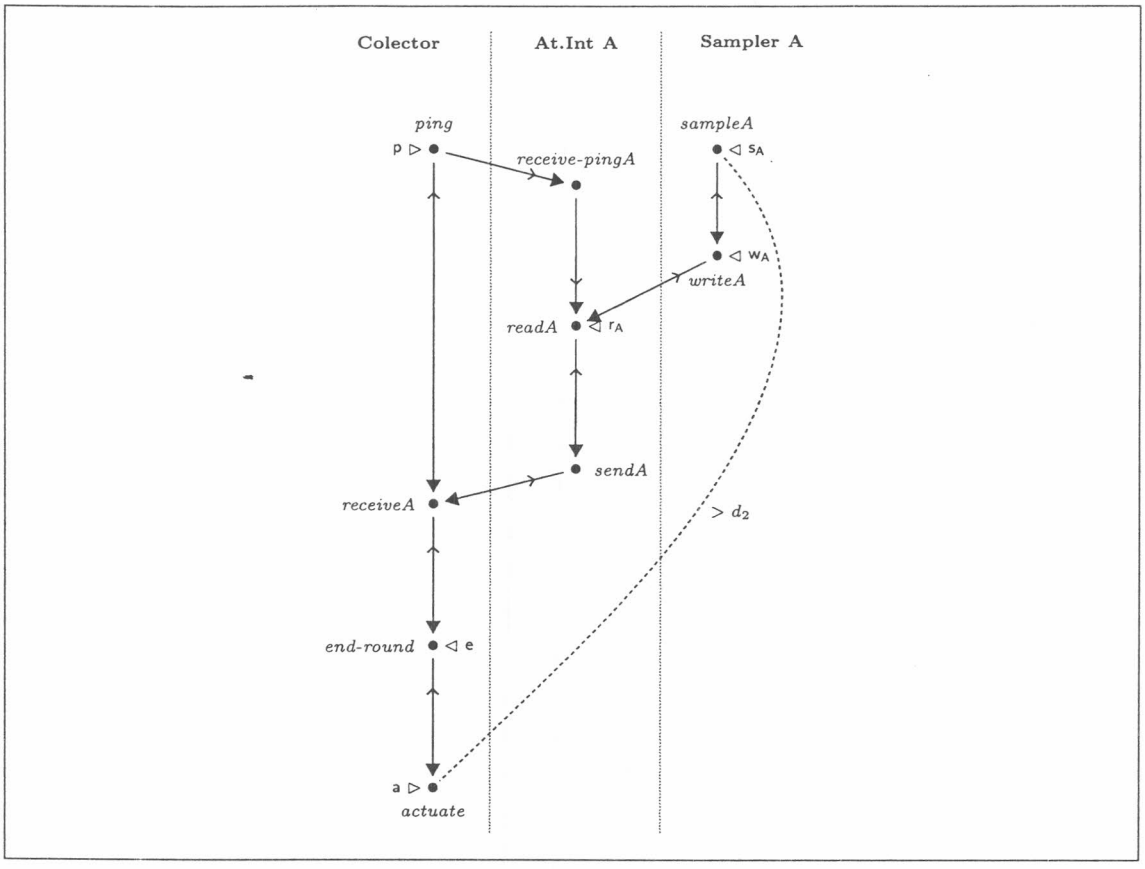
Para facilitar la comprensión del Patrón 5.6, se organizaron los eventos en *andariveles*[1] según la tarea a la cual pertenecían. Las tareas *Sampler A* y *Sampler B* corresponden a las tareas periódicas de recolección de los datos medidos por sendos sensores. Las tareas *Atención Interrupciones A* y *Atención Interrupciones B* corresponden a las tareas esporádicas que atienden los pedidos generados por el *Colector*.

En este ejemplo se ve claramente el uso del orden parcial entre eventos para indicar paralelismo y sincronización entre los eventos de procesos distintos. Si bien el Patrón 5.6 es sin lugar a dudas el patrón mas complejo que analizamos en este trabajo, su complejidad está intrínsecamente relacionada con la complejidad de la propiedad que queríamos expresar. Aunque hayamos enunciado la propiedad en castellano en una sóla línea, para expresarla de forma tal de que pueda ser verificada en forma automática es necesario ser precisos con respecto a la correspondencia entre eventos (por ejemplo para no permitir que se compare la lectura de un dato ocurrida en una ronda con otra lectura ocurrida en una ronda anterior). Lo que termina siendo un patrón de mal comportamiento mediano hubiera sido prácticamente imposible de escribir (desde el punto de vista de un ingeniero) usando un lenguaje basado en *interleaving* de eventos como los autómatas temporizados.

Requerimiento 5.7. *(Frescura) En el momento de ser recolectados y apareados, los datos no deben tener más de* d_2 ms *de antigüedad.*

El Patrón 5.7 captura los comportamientos que violan este requerimiento con respecto al sensor A. Dado que los sensores A y B son indistinguibles para nosotros no es necesario verificar aparte el caso para el sensor B. Como en el caso anterior, se identifican los puntos de comienzo (punto p) y finalización (punto e) de una ronda correspondiente a una cierta actuación del procesador (punto a). Es decir, los datos usados en a

[1] El uso de *andariveles* es un recurso puramente gráfico y no forma parte del lenguaje de patrones de eventos.



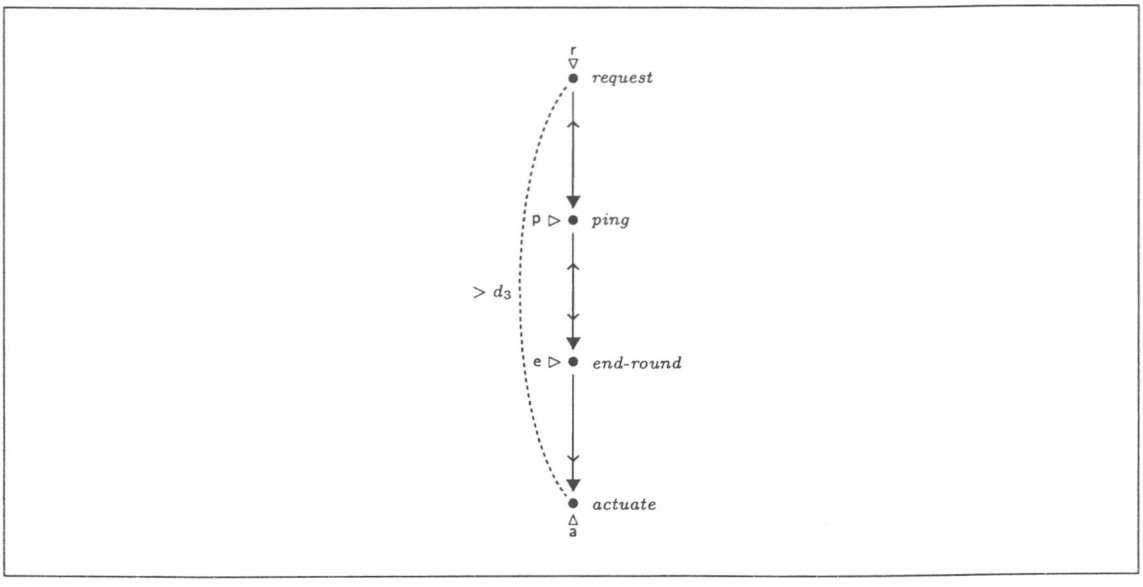
Patrón 5.7: patrón de mal comportamiento para el Requerimiento 5.7 - Frescura

fueron recolectados en la ronda demarcada por p y e. Una vez más, usamos la estrategia de "retroceder" en la ejecución buscando los momentos en que se realiza la recepción de datos del sensor (evento *receiveA*), el sensor envían esos datos (evento *sendA*)[2]y finalmente el momento en que se lee la variable compartida escrita por el sampler (evento *readA*). Sabemos que este *readA* fue además el primero en ser realizado desde que se recibió el ping del colector (evento *receive* − *pingA*) y por lo tanto podemos estar seguros que r_A corresponde a la lectura de los valores que fueron posteriormente utilizados en a. Desde r_A podemos localizar la última escritura realizada sobre la variable compartida (punto w_A) y a su vez el último muestreo antes de esa escritura (punto s_a). Quiere decir que en s_A se realiza la medición que posteriormente fuera utilizada en a. El patrón describe aquellos escenarios en los cuales la medición ocurrió más de d_2 ms antes de su utilización y por lo tanto los datos utilizados no eran suficientemente frescos.

Requerimiento 5.8. *(Respuesta acotada) El tiempo de respuesta entre el pedido de apareo y el apareo en sí mismo no supera los* d_3 ms.

El Patrón 5.8 identifica el comienzo (punto p) y la finalización (punto e) de una ronda de muestreo. El pedido de apareo (evento *request*) correspondiente a dicha ronda es el último antes de p y está representado por el punto r. La actuación correspondiente a la ronda (en donde se aparean de los datos) es la primera después de e y está representada por el punto a. El patrón describe aquellas ejecuciones en las cuales entre r y a transcurren más de d_3 ms y por lo tanto se viola el requerimiento de respuesta acotada.

[2]Como sabemos que no se utilizan buffers en el sistema, aparear la recepción de un mensaje con su envío es muy simple y consiste en encontrar el último envío anterior a la recepción.



Patrón 5.8: patrón de mal comportamiento para el Requerimiento 5.8 - Respuesta acotada

Capítulo 6

Implementación

En los capítulos anteriores vimos que el problema de model-checking de patrones de mal comportamiento es decidible y lo demostramos reduciéndolo a un problema de verificación de autómatas temporizados. El enfoque basado en *autómatas reconocedores* que utilizamos en el capítulo 4 servirá también como base para la implementación de un *verificador de patrones de mal comportamiento.*

Dado un sistema S modelado con un autómata temporizado $\mathcal{A}_S$ y dado un requerimiento R sobre S, el diseñador deberá construir un patrón de eventos $\mathcal{P}_{\neg R}$ que represente los comportamientos que violan R. Este patrón y el modelo del sistema serán el input para el *Verificador de patrones.* La Figura 1.5 muestra esquemáticamente la estructura de nuestro verificador.

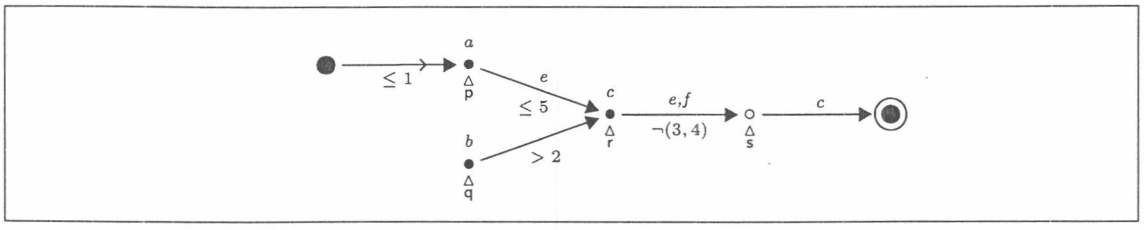
6.1 Verificación de patrones de mal comportamiento

En esta sección presentamos el algoritmo básico implementado por nuestra herramienta de verificación de patrones de mal comportamiento. Dado un autómata temporizado $\mathcal{A}_S$ que modela el comportamiento de un sistema S y un patrón de mal comportamiento $\mathcal{P}$, nuestra herramienta decide si $\mathcal{A}_S \vDash \mathcal{P}$, es decir, si el sistema S admite algún comportamiento no deseado descrito por $\mathcal{P}$.
El algoritmo de verificación de patrones utiliza los conceptos presentados en el capitulo 4 en cuanto a reducción a un problema de verificación de autómatas temporizados y se basa en la herramienta Kronos como motor de decisión.

Dado un autómata temporizado $\mathcal{A}_S$ y un patrón de mal comportamiento $\mathcal{P}$, nuestro algoritmo de verificación realiza los siguientes pasos:

1. Construye un autómata reconocedor $\mathcal{A}_\mathcal{P}$ a partir del patrón de mal comportamiento utilizando un algoritmo basado en la definición 4.1.13 que describiremos más adelante
2. Utiliza la herramienta Kronos para decidir si $\mathcal{A}_S || \mathcal{A}_\mathcal{P} \vDash \varphi_{accept}$, donde φ_{accept} es la fórmula TCTL presentada en el Corolario 4.16
3. El algoritmo contesta OK si y sólo si Kronos contestó OK

La próxima sección describe el algoritmo de construcción de autómatas reconocedores. En el Apéndice B se mencionan las principales características de la implementación de la herramienta de verificación en el lenguaje Java y se muestra la aplicación del Verificador de patrones a los requerimientos para el protocolo CSMA/CD presentados en el capítulo 5.



Patrón 6.1: Patrón de mal comportamiento

6.1.1 Algoritmo para construcción de autómatas reconocedores

El algoritmo que utilizaremos para la construcción de autómatas reconocedores para patrones de mal comportamiento está basado en la definición 4.1.13, aunque no es una implementación directa del mismo. Para construir el autómata, comenzaremos analizando la locación inicial del mismo (correspondiente a la configuración vacía) y evaluaremos todas las posibles formas de extender dicha configuración. Para cada una de las extensiones generadas, repetiremos este procedimiento en forma recursiva hasta saturar el conjunto de locaciones del autómata, es decir, hasta que no haya forma de agregar más locaciones. Informalmente, a partir de un patrón $\mathcal{P}$ el algoritmo realizará los siguientes pasos:

1. Crear el conjunto de locaciones con sólo las locaciones $\{[\varnothing], s_{trap}\}$
2. Mientras se puedan seguir agregando locaciones:
   (a) Para cada locación del conjunto, generar todas las aristas salientes de dicha locación. Si el destino de alguna de estas aristas no pertenece al conjunto de locaciones, agregarlo.
3. Generar todas las aristas *Trap*
4. Poner $\top$ a todos los invariantes de todas las locaciones
5. Construir X, el conjunto de relojes del autómata
6. Poner $[\varnothing]$ como locación inicial

Claramente, nuestro algoritmo no generará configuraciones que no sean alcanzables desde la configuración inicial. La forma *naive* de generar todas las posibles aristas salientes desde una locación consiste en probar uno a uno con todos los puntos y todos los eventos y verificar si se cumple con las condiciones que establece la definición 4.1.13. Sin embargo, esta estrategia puede ser mejorada fácilmente. Veamos, en primera instancia, qué estrategia se puede utilizar para generar todas las aristas *Mark* e *Instant*.

La primera condición para que exista una arista saliente *Mark* o *Instant* desde una locación es que exista alguna forma de *extender* la configuración asociada (definición 4.1.2). Nuevamente, existe una forma muy ingenua de generar todas las posibles extensiones de una configuración que consiste en tomar uno a uno todos los puntos que no pertenezcan a la configuración y verificar si todos los predecesores de dicho punto pertenecen a la configuración. Sin embargo, se puede restringir la búsqueda de extensiones a un conjunto más acotado de *candidatos*.
Por ejemplo, ningún punto que tenga al menos un predecesor podrá servir para extender la configuración vacía. Quiere decir que los únicos *candidatos a extensión* de la configuración vacía son los elementos *ínfimos* del orden parcial $\prec_{\mathcal{P}}$. Por otro lado, un candidato para extender una configuración será también candidato de todas las extensiones de dicha configuración que no lo contengan. Conociendo el conjunto de candidatos para una configuración se puede calcular el conjunto de candidatos de cualquiera de sus extensiones simplemente eliminando el punto que ha sido agregado y agregando los sucesores directos del punto en cuestión. Tomemos, por ejemplo el Patrón 6.1. Estando en la configuración $\varnothing$, los candidatos para extensiones son $\{\mathsf{p}, \mathsf{q}\}$. Supongamos que decidimos extender a $\varnothing$ por p, el conjunto de candidatos para la nueva configuración será $\{\mathsf{q}, \mathsf{r}\}$. Claramente, s no puede ser de ninguna manera una extensión posible para esa configuración. El hecho

de que un punto sea candidato para extensión con respecto a una determinada configuración no significa que en efecto sirva como extensión. Por ejemplo, en el caso anterior, r es un candidato a extensión con respecto a la configuración {p} pero no sirve como extensión porque q, el otro predecesor de r, no ha sido incorporado todavía a la configuración.

Formalmente, definimos inductivamente la función $Cand : \Theta_{\mathcal{P}} \longrightarrow 2^P$ que asocia a cada configuración un conjunto de *candidatos para extensión* de la siguiente manera:
Para todo $\Theta \in \Theta_{\mathcal{P}}$, $x \in P$:

$$Cand(\varnothing) = infimos(\prec_{\mathcal{P}})$$
$$Cand(\Theta \uplus \{x\}) = Cand(\Theta) \setminus \{x\} \cup suc(x)$$

$infimos_{\prec_{\mathcal{P}}}$ es el conjunto de puntos del patrón que no tienen predecesores y $suc(x)$ es el conjunto de sucesores directos de x. Para calcular el conjunto de ínfimos no tenemos en cuenta al elemento $\mathbf{0}$.

Dada una configuración Θ y un punto x en $Cand(\Theta)$, debemos ver si todos los predecesores (directos) de x están incluidos en la configuración. Para poder chequear esto en forma eficiente utilizaremos un *contador de predecesores* $\#pred$ asociado a cada configuración y cada punto. Dada una configuración Θ y un punto x, el contador determinará cuantos predecesores directos de x no pertenecen a Θ. Claramente, x podrá ser una extensión de Θ sólo si $\#pred(\Theta)(x) = 0$. Formalmente, decimos que $\#pred : \Theta_{\mathcal{P}} \longrightarrow (P \longrightarrow \mathbf{N})$ es una función definida como:
Para todo $\Theta \in \Theta_{\mathcal{P}}$, $x, y \in P$:

$$\#pred(\varnothing)(x) = \#(pred(x))$$
$$\#pred(\Theta \uplus \{y\})(x) = \begin{cases} \#pred(\Theta)(y) - 1 & \text{si } y \rightarrow x \\ \#pred(\Theta)(y) & \text{en otro caso} \end{cases}$$

donde $\#(pred(x))$ es el cardinal del conjunto de predecesores directos de x. Contando con C y con $\#pred$ podemos decidir en forma eficiente cuales son todas las formas posible de extender una configuración.

Proposición 6.1. *Dado un patrón de eventos* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, *una configuración* $\Theta \in \Theta_{\mathcal{P}}$ *y un punto* $x \in P$,

$$\Theta \uplus \{x\} \in \Theta_{\mathcal{P}} \qquad sii \qquad x \in Cand(\Theta) \;\; y \;\; \#pred(\Theta)(x) = 0$$

Una vez que encontramos alguna forma de extender la configuración asociada a una locación, si la extensión corresponde a un punto de la clase I, entonces siempre se podrá generar una arista *Instant* que marque ese punto. Sin embargo, si la extensión corresponde a un punto de la clase E, hay que determinar para cuales de los eventos que tiene asociados se puede construir una arista *Mark*. Nuevamente, esto se puede realizar en forma *naive* calculando en forma explícita el valor de $\Gamma^*_{\triangleright x}(\Theta)$ y calculando cuales de los eventos asociados a x no pertenecen a dicho conjunto. En lugar de hacer esto, utilizaremos el concepto de *restricciones activas* que introdujimos en el capítulo 4. Diremos que una restricción de eventos se *activa* cuando se marca el primero de los puntos involucrados y se *desactiva* cuando se marca el otro. En los ejemplos presentados en la introducción del capítulo 4 vimos que contar con el conjunto de eventos "prohibidos" en cada configuración alcanzaba para decidir si era seguro saltear un determinado evento pero que no era suficiente para decidir si era seguro marcar un determinado punto sobre el siguiente evento. Mostramos a través de ejemplos que necesitábamos saber qué puntos determinan que un evento esté prohibido para poder decidir si era seguro marcar un punto sobre ese evento. Dado un punto x y un evento e, lo primero que nos interesará saber es si e figura como restricción entre algún punto y x y si es así, cuáles son esos puntos. Definimos la función $RestActivadasPor : P \cup \{\mathbf{0}\} \longrightarrow (\Sigma \longrightarrow 2^{P \cup \{\infty\}})$ de la siguiente manera:
Para todo $x \in P$, $e \in \Sigma$:

	a	b	c	e	f
$\mathbf{0}$	$\{\mathsf{p}\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
p	$\varnothing$	$\varnothing$	$\varnothing$	$\{\mathsf{r}\}$	$\varnothing$
q	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
r	$\varnothing$	$\varnothing$	$\varnothing$	$\{\mathsf{p},\mathsf{s}\}$	$\{\mathsf{s}\}$
s	$\varnothing$	$\varnothing$	$\{\infty\}$	$\{\mathsf{r}\}$	$\{\mathsf{r}\}$

Tabla 6.1: Valores de *RestActivadasPor* para el Patrón 6.1

	a	b	c	e	f
$\varnothing$	$\{\mathsf{p}\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$\{\mathsf{p}\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\{\mathsf{r}\}$	$\varnothing$
$\{\mathsf{q}\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$\{\mathsf{p},\mathsf{q}\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\{\mathsf{r}\}$	$\varnothing$
$\{\mathsf{p},\mathsf{q},\mathsf{r}\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\{\mathsf{s}\}$	$\{\mathsf{s}\}$
$\{\mathsf{p},\mathsf{q},\mathsf{r},\mathsf{s}\}$	$\varnothing$	$\varnothing$	$\{\infty\}$	$\varnothing$	$\varnothing$

Tabla 6.2: Valores de *RestActivasPorEvento* para el Patrón 6.1

$$\mathit{RestActivadasPor}(\mathsf{x})(e) = \{\mathsf{y} \in P \cup \{\infty\} \mid e \in \gamma(\mathsf{x},\mathsf{y})\}$$

Para el Patrón 6.1, la Tabla 6.1 muestra cual sería el valor de *RestActivadasPor* para cada punto y cada evento.

Definimos también la función *RestActivasPorEvento* que a cada evento le asocia los puntos que determinan que dicho evento esté prohibido en la configuración actual:
$\mathit{RestActivasPorEvento} : \Theta_{\mathcal{P}} \longrightarrow (\Sigma \longrightarrow 2^{P \cup \{\infty\}})$ donde para todo $\Theta \in \Theta_{\mathcal{P}}$ y para todo evento $e \in \Sigma$:

$$\mathit{RestActivasPorEvento}(\varnothing)(e) = \mathit{RestActivadasPor}(\mathbf{0})(e)$$
$$\mathit{RestActivasPorEvento}(\Theta \uplus \{\mathsf{x}\})(e) = (\mathit{RestActivasPorEvento}(\Theta)(e) \cup \mathit{RestActivadasPor}(\mathsf{x})(e)) \setminus (\Theta \cup \{\mathsf{x}\})$$

Claramente, los puntos que determinan que un cierto evento e esté prohibido en la configuración inicial son aquellos que tienen a e como restricción de eventos con respecto al comienzo de la ejecución. Teniendo las restricciones activas en una configuración Θ, las restricciones activas de una extensión de Θ serán las que estaban activas en Θ más las que activa x, sin contar todas las referencias a puntos ya marcados (es decir, pertenecientes a la nueva configuración). Dado que γ es una función simétrica, al marcar x, $\mathit{RestActivadasPor}(\mathsf{x}, e)$ puede hacer referencia tanto a puntos ya marcados como a puntos todavía sin marcar. Si se da el primer caso, los puntos ya marcados no deben figurar como restricciones activas de la configuración porque justamente han sido desactivadas al marcar x. Es por eso que a la unión entre las restricciones activas hasta la configuración anterior y las restricciones activadas por x es necesario sacarle todos los puntos ya marcados. La Tabla 6.2 muestra el valor de *RestActivasPorEvento* para cada configuración y cada evento del Patrón 6.1.

Proposición 6.2. *Dado un patrón de eventos* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, *una configuración* $\Theta \in \Theta_{\mathcal{P}}$ *y un evento* $e \in \Sigma$,

$$e \in \Gamma(\Theta) \quad \textit{sii} \quad \mathit{RestActivasPorEvento}(\Theta)(e) \neq \varnothing$$

	p	q	r	s
0	$z_0 \leq 1$	$\top$	$\top$	$\top$
p	$\top$	$\top$	$z_p \leq 5$	$\top$
q	$\top$	$\top$	$z_q > 2$	$\top$
r	$z_r \leq 5$	$z_r > 2$	$\top$	$z_r \leq 3 \wedge z_r \geq 4$
s	$\top$	$\top$	$z_s \leq 3 \wedge z_s \geq 4$	$\top$

Tabla 6.3: Valores de *RestTemp* para el Patrón 6.1

Proposición 6.3. *Dado un patrón de eventos* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$*, una configuración* $\Theta \in \Theta_{\mathcal{P}}$*, un punto* $x \in P$ *y un evento* $e \in \Sigma$*,*

$$e \in \Gamma_{\triangleright x}(\Theta) \qquad sii \qquad RestActivasPorEvento(\Theta)(e) \setminus \{x\} \neq \varnothing$$

Contando con la función *RestActivasPorEvento*, para decidir si se puede agregar una arista $Mark$ saliente desde una configuración Θ por un evento e hacia una extensión $\Theta \uplus \{x\}$ alcanza con verificar que $RestActivasPorEvento(\Theta)(e) \setminus \{x\} = \varnothing$. Es decir, que no haya ningún punto no marcado distinto de x que prohíba la ocurrencia del evento e. La misma función *RestActivasPorEvento* nos permite generar las aristas $Skip$ y $Fail$ simplemente verificando si $RestActivasPorEvento(\Theta)(e)$ es vacío o no lo es, respectivamente.

Para generar las guardas de las aristas $Mark$ e $Instant$ usaremos una función que dado un par de puntos devuelve la restricción sobre los relojes que representa a la restricción temporal correspondiente. Es decir, $RestTemp : P \cup \{\mathbf{0}\} \times P \longrightarrow \Psi_X$ es una función tal que para todo par de puntos $x \in P \cup \{\mathbf{0}\}$, $y \in P$:

$$RestTemp(x, y) = \psi_{z_x}(\delta(x, y))$$

La Tabla 6.3 muestra el valor de *RestTemp* para todo par de puntos del Patrón 6.1.

Se puede ver que si bien δ es una función simétrica, *RestTemp* no lo es. Esto es así porque $RestTemp(x, y)$ da la restricción sobre los relojes que representa a $\delta(x, y)$ pero asumiendo que x fue marcado antes que y. Estando en una configuración Θ, la guarda de una arista $Mark$ o $Instant$ que marque al punto x estará dada por la función $Guarda : \Theta_{\mathcal{P}} \times P \longrightarrow \Psi_X$ donde para todo $\Theta \in \Theta_{\mathcal{P}}$ y $x \in P$:

$$Guarda(\Theta, x) = \bigwedge_{y \in \Theta \cup \{\mathbf{0}\}} RestTemp(y, x)$$

La Tabla 6.4 muestra el valor de *Guarda* para cada configuración y cada punto, aplicada al Patrón 6.1.

Proposición 6.4. *Dado un patrón de eventos* $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$*, una configuración* $\Theta \in \Theta_{\mathcal{P}}$ *y un punto* $x \in P$*,*

$$\psi^x_\Theta = Guarda(\Theta, x)$$

donde la igualdad debe ser interpretada como equivalencia lógica.

A la hora de generar las guardas y de generar el conjunto de relojes del autómata reconocedor, diferenciaremos las restricciones temporales *triviales* de las *no triviales*. Diremos que una restricción temporal es trivial cuando se refiere al intervalo $[0, \infty)$. Análogamente, diremos que una restricción sobre relojes es trivial cuando es lógicamente equivalente a $\top$. Cuando las únicas restricciones sobre relojes que se apliquen sobre un determinado reloj sean triviales, ese reloj no será agregado al conjunto de relojes. De la misma manera, eliminaremos los términos triviales de la conjunción definida por *Guarda*.

	p	q	r	s
$\varnothing$	$z_0 \leq 1$	$\top$	$\top$	$\top$
{p}	$\top$	$\top$	$z_p \leq 5$	$\top$
{q}	$\top$	$\top$	$z_q > 2$	$\top$
{p, q}	$\top$	$\top$	$z_p \leq 5 \wedge z_q > 2$	$\top$
{p, q, r}	$\top$	$\top$	$\top$	$z_r \leq 3 \wedge z_r \geq 4$
{p, q, r, s}	$\top$	$\top$	$\top$	$\top$

Tabla 6.4: Valores de *Guarda* para el Patrón 6.1

Estructuras de datos

Aprovechando el hecho de que la cantidad de puntos y de eventos es finita y que por el uso que se les dará a los patrones es de esperarse que además no sea demasiado grande[1] usaremos *bitsets* para representar conjuntos de puntos y de eventos. Para esto asumiremos que los puntos y los eventos pueden ser mapeados en forma unívoca en el rango $[0, n_P)$ y $[0, n_\Sigma)$, respectivamente, donde n_P es el cardinal del conjunto de puntos y n_Σ es el cardinal del alfabeto de eventos.

La función *Cand* no será materializada en ningún momento. El conjunto de candidatos para una configuración será calculado en función de los candidatos de la configuración anterior, aprovechando la definición recursiva de *Cand*.

De la misma manera, *#pred* no será calculada explícitamente para todas las configuraciones sino que será calculada incrementalmente para cada configuración. Para cada configuración Θ, $\#pred(\Theta)$ estará representado por un array de enteros de tamaño n_P, donde la iésima celda determina el valor de $\#pred(\Theta)$ para el iésimo punto.

La función *RestActivadasPor* será materializada en una tabla de n_P por n_Σ, donde cada celda contiene un conjunto de puntos. En cambio, *RestActivasPorEvento* no será calculada explícitamente sino que será calculada usando su definición recursiva. Para cada configuración Θ, $RestActivasPorEvento(\Theta)$ estará representada por un array de n_Σ, donde cada celda contiene un conjunto de puntos.

Finalmente, *RestTemp* será representada por una tabla de n_P por n_P, donde cada celda contendrá una restricción sobre relojes. Por otro lado, *Guarda* será calculada *on-the-fly* por una función auxiliar.

Llamaremos *contexto de la configuración* (o simplemente *contexto*) a la tupla tupla $\langle \Theta, C, R, \#P \rangle$ donde para cada $\Theta \in \Theta_\mathcal{P}$:

- $C = Cand(\Theta)$
- $R = RestActivasPorEvento(\Theta)$
- $\#P = \#pred(\Theta)$

A continuación, mostramos el pseudocódigo de la función de construcción del autómata reconocedor para un patrón de eventos.

Pseudocódigo

La función principal *GenerarAutomataReconocedor*, mostrada en el Cuadro 1, toma como entrada un patrón de eventos y devuelve un autómata temporizado según la definición 4.1.13.

[1] Es de esperarse que tengan no más de 10 puntos y 20 eventos dado que pasado ese punto la complejidad del patrón sería demasiado grande como para ser generado gráficamente por un diseñador. De todas formas, el análisis que realizamos es válido aún suponiendo que hubiera 1000 puntos y 1000 eventos

```
function GenerarAutomataReconocedor(P = ⟨Σ, P, ℓ, →, γ, δ, 0, ∞⟩) returns A = ⟨S, X, Σ, A, I, s0⟩
    X ⟵ ∅
    A ⟵ ∅

    s0 ⟵ [∅]
    S ⟵ {s_trap} ∪ {s0}

    Explorar(P, PrimerContextoDeConfiguracion(P), S, A)

    for all s ∈ S do
      I(s) ⟵ ⊤
    end for

    for all l ∈ Σ do
      AgregarAristaTrap(A, l)
    end for
end
```

Cuadro 1: Función principal para la generación de autómatas reconocedores.

Explorar, mostrada en el Cuadro 2, es una función recursiva que para cada configuración genera todas las aristas salientes desde la locación correspondiente. Si el destino de alguna de las aristas salientes es una locación todavía no "explorada" (es decir, que todavía no fue agregada al conjunto S), entonces repite el proceso en forma recursiva para esa locación.

Lo primero que hace *Explorar* es analizar el conjunto C de candidatos a extensión para la configuración actual. Para cada uno de los puntos de ese conjunto, verifica si es una extensión válida (en el sentido de que todos los predecesores del punto estén incluidos en la configuración actual y además exista al menos una forma de marcar en forma segura alguno de los eventos asociados al punto). Si ese es el caso, verifica si la locación correspondiente a la configuración extendida ya fue generada. Si se trata de una locación no vista hasta el momento, repite recursivamente el proceso para esa locación, calculando antes el contexto correspondiente a la nueva configuración (función *Extender*). Cuando termina de generar todas las locaciones alcanzables desde la configuración actual, genera las aristas *Mark* o *Instant* salientes que correspondan, según x pertenezca a I o a E. Sólo se generarán aristas *Mark* para eventos que no pertenezcan al conjunto $\Gamma_{\triangleright \mathsf{x}}(\Theta)$. La función *EventosHabilitados* calcula el conjunto de eventos para el cual hay que generar una arista *Mark* o *Instant*. Finalmente, *Explorar* genera las aristas *Skip* y *Fail* salientes desde la configuración actual. Para cada evento e, o bien está prohibido por algún punto pendiente de ser marcado ($R(e) \neq \varnothing$) y por lo tanto $e \in \Gamma(\Theta)$ o no está prohibido y $e \notin \Gamma(\Theta)$.

La función auxiliar *PrimerContextoDeConfiguracion* (Cuadro 3) construye el contexto para la configuración inicial, es decir $\varnothing$.

El Cuadro 4 muestra el pseudocódigo de la función *ExtensionValida* y todas las funciones auxiliares que ésta utiliza. La función *ExtensionValida* determina si todos los predecesores de un cierto punto x pertenecen a una configuración Θ y, si es así, si existe alguna posibilidad de generar al menos una arista entre $[\Theta]$ y $[\Theta \uplus \{\mathsf{x}\}]$. Para que esta última condición valga, x debe pertenecer a I o alguno de los eventos asociados a dicho punto debe estar *habilitado*. Diremos que un evento e asociado a un punto $\mathsf{x} \in E$ en una configuración Θ está *habilitado* si $e \notin \Gamma^*_{\triangleright \mathsf{x}}(\Theta)$. La función *EventoNoRestringido* es la responsable de determinar si un evento está habilitado. Si un evento e no está restringido por ningún punto (caso $R(e) = \varnothing$), entonces es claro que el evento está habilitado. Si el único punto que prohíbe a e es x (caso $\mathsf{x} \in R(e)$ y $\#(R(e)) = 1$), entonces el evento también está habilitado. En todos los demás casos, el evento está prohibido por otro punto aparte de x y por lo tanto no está habilitado.
Notemos que si $\mathsf{x} \in I$, *EventosHabilitados* devuelve un conjunto cuyo único elemento es λ. Esto permite tratar uniformemente a los puntos de ambas clases. Si el punto pertenece a E, entonces hay que analizar

```
function Explorar(P, ⟨Θ, C, R, #P⟩, S, A, X) returns
    for all x ∈ C do
      if ExtensionValida(P,⟨Θ, C, R, #P⟩,x) then
        if [Θ ∪ {x}] ∉ S then
          S ⟵ S ∪ {[Θ ∪ {x}]}
          Explorar(P, Extender(⟨Θ, C, R, #P⟩, x), S, A, X)
        end if
        guarda ⟵ Guarda(Θ, x)
        reset ⟵ Reset(P, Θ, x, X)
        for all e ∈ EventosHabilitados(P,⟨Θ, C, R, #P⟩,x) do
          AgregarAristaMarkoInstant(A,[Θ], e, guarda, reset,[Θ ∪ {x}])
        end for
      end if
    end for

    for all e ∈ Σ do
      if R(e) ≠ ∅ then
        AgregarAristaFail(A, [Θ], e)
      else
        AgregarAristaSkip(A, [Θ], e)
      end if
    end for
    AgregarAristaFail(A, [Θ], λ)
    AgregarAristaSkip(A, [Θ], λ)
end
```

Cuadro 2: Función recursiva para la generación de todas las locaciones alcanzables desde la inicial.

```
function PrimerContextoDeConfiguracion(P = ⟨Σ, P, ℓ, →, γ, δ, 0, ∞⟩) returns ⟨Θ, C, R, #P⟩
    Θ ⟵ ∅
    C ⟵ ínfimos(→)
    R ⟵ RestActivadasPor(0)
    for all x ∈ P do
      #P(x) ⟵ #(pred(x))
    end for
end
```

Cuadro 3: Contexto para la configuración inicial

cuales de los eventos asociados al punto están habilitados para decidir qué aristas *Mark* hay que generar. Si, en cambio, el punto pertenece a I siempre hay que generar una arista *Instant* por λ.

```
function ExtensionValida(P,⟨Θ, C, R, #P⟩, x) returns res : true|false
    res ⟵ PredecesoresIncluidos(⟨Θ, C, R, #P⟩, x) y EventosHabilitados(P,⟨Θ, C, R, #P⟩,x) ≠ ∅
end

function PredecesoresIncluidos(⟨Θ, C, R, #P⟩, x) returns res: true|false
    res ⟵ #P(x) = 0
end

function EventosHabilitados(P,⟨Θ, C, R, #P⟩, x) returns L ⊆ 2^(Σ∪{λ})
    if x ∈ E then
      L ⟵ ∅
      for all e ∈ ℓ(x) do
        if EventoNoRestringido(⟨Θ, C, R, #P⟩, x, e) then
          L ⟵ L ∪ {e}
        end if
      end for
    else
      L ⟵ {λ}
    end if
end

function EventoNoRestringido(⟨Θ, C, R, #P⟩, x, e) returns res: true|false
    r ⟵ R(e)
    res ⟵ (r = ∅ o (x ∈ r y #(r) = 1))
end
```

Cuadro 4: Extensiones válidas de una configuración

La función *Extender* (Cuadro 5) genera un nuevo contexto para la extensión de Θ por x. Como dijimos antes, el conjunto C de candidatos, la función R de restricciones y la función $\#P$ que cuenta predecesores todavía no marcados se calculan para la nueva configuración a partir de las correspondientes a Θ, utilizando la definición recursiva de cada una de ellas.

La función *Guarda* (Cuadro 6) calcula *on-the-fly* el valor de ψ^{x}_{Θ}. La función *Reset* verifica si tiene sentido agregar un reloj para medir el tiempo transcurrido desde que se marcó al punto x. Tendrá sentido medir ese tiempo si x tiene alguna restricción temporal no trivial con algún punto no marcado todavía. Si este último fuera el caso, se agregará el reloj z_x al conjunto X de relojes y se lo reseteará en la arista que marca x. Como mencionamos antes, una restricción temporal será considerada *trivial* si siempre es verdadera.

Finalmente, el Cuadro 7 muestra las funciones auxiliares que agregan aristas de los distintos tipos al conjunto A.

```
function Extender(⟨Θ, C, R, #P⟩, x) returns ⟨Θ', C', R', #P'⟩
    Θ' ⟵ Θ ∪ {x}
    C' ⟵ C \ {x} ∪ suc(x)
    for all e ∈ Σ do
      R'(e) ⟵ (R(e) ∪ RestActivadasPor(x)(e)) \ (Θ ∪ {x})
    end for
    for all y ∈ P do
      if x → y then
        #P'(y) ⟵ #P(y) − 1
      else
        #P'(y) ⟵ #P(y)
      end if
    end for
end
```

Cuadro 5: Extensión de una configuración

```
function Guarda(Θ, x) returns ψ
    ψ ⟵ ⋀_{y∈Θ∪{0}} Δ(y, x)
end

function Reset(P, Θ, x, X) returns reset ⊆ 2^X
    if TieneRestriccionesNoTriviales(E, Θ, x) then
      X ⟵ X ∪ {z_x}
      reset ⟵ {z_x}
    else
      reset ⟵ ∅
    end if
end

function TieneRestriccionesNoTriviales(E, Θ, x) returns res: true|false
    res ⟵ false
    for all y ∈ P \ Θ do
      res ⟵ res o no Trivial(Δ(x, y))
    end for
end
```

Cuadro 6: Generación de guardas y relojes.

function AgregarAristaMarkoInstant(A, s, l, g, r, s')
    $A \longleftarrow A \cup \{\langle s, l, g, r, s' \rangle\}$
end

function AgregarAristaSkip(A, s, l)
    $A \longleftarrow A \cup \{\langle s, l, \top, \varnothing, s \rangle\}$
end

function AgregarAristaFail(A, s, l)
    $A \longleftarrow A \cup \{\langle s, l, \top, \varnothing, s_{trap} \rangle\}$
end

function AgregarAristaTrap(A, l)
    $A \longleftarrow A \cup \{\langle s_{trap}, l, \top, \varnothing, s_{trap} \rangle\}$
end

Cuadro 7: Generación de aristas

Capítulo 7

Conclusiones, trabajo relacionado y trabajo futuro

7.1 Conclusiones

Las técnicas de verificación formales han despertado el interés de muchos grupos de investigación durante las últimas dos décadas. Habiéndose superado los obstáculos relacionados con la construcción de algoritmos de verificación automática que puedan ser aplicados en forma efectiva a casos reales, la atención comienza a orientarse hacia la transferencia tecnológica de estas herramientas. Vimos que el principal punto pendiente es brindar soporte a los diseñadores y desarrolladores en las tareas de especificación formal de sistemas. La mayoría de los formalismos de especificación existentes están basados en formalismos matemáticos con los que resulta difícil trabajar, especialmente en sistemas de complejidad media o grande como los que se encuentran habitualmente en la industria.

Nuestro trabajo tenía como objetivo definir las bases para un lenguaje de alto nivel para expresar requerimientos de sistemas concurrentes y de tiempo real. A lo largo de este trabajo presentamos un formalismo para expresar *comportamientos no deseados* de un sistema basado en órdenes parciales de eventos y mostramos como puede ser aplicado a la verificación formal de sistemas. Consideramos que este formalismo, al cual llamamos *patrones de eventos*, tiene las siguientes ventajas:

- Admite una notación gráfica.
- Permite expresar en forma compacta propiedades de concurrencia o causalidad entre eventos, lo cual lo hace especialmente adecuado para ser aplicado en la verificación de sistemas concurrentes.
- Permite expresar restricciones temporales cuantitativas, lo cual lo hace especialmente adecuado para verificación de sistemas de tiempo real.
- Posee una semántica simple, basada en el concepto de *pattern matching*.

Mostramos cómo pueden ser utilizados estos patrones de eventos en la especificación de *patrones de mal comportamiento*, es decir, en la descripción de comportamientos no deseados en el sistema. Vimos que en general resulta más simple y directo definir cuales son los comportamientos "erróneos" de un sistema que describir en forma general los casos en los cuales no se produce ningún error. Comprobamos mediante ejemplos concretos que el soporte a las propiedades de concurrencia y causalidad de los patrones de eventos hace posible escribir propiedades que sería impracticable escribir en una lógica temporizada o incluso con autómatas temporizados.

En este trabajo también demostramos formalmente que la verificación de patrones de mal comportamiento es decidible y construimos además un prototipo de verificador basado en autómatas temporizados.

Entendemos el que este trabajo sienta una base sólida para la construcción de un *framework* de especificación formal que pueda ser utilizado por diseñadores y desarrolladores de sistemas fuera del ámbito académico, aunque queden todavía varios puntos sobre los cuales profundizar.

7.2 Trabajos relacionados

Como mencionamos anteriormente, existe un interés creciente en este área de investigación y se han realizado varios trabajos orientados a proveer herramientas y notaciones de especificación de propiedades mas *amigables*.

En [DAC98] se analizaron cientos de ejemplos de especificaciones formales generadas por distintos grupos de investigación y expresadas usando distintos formalismos. Una de las conclusiones más interesantes de ese trabajo es que a pesar del enorme poder expresivo que tienen la mayor parte de los formalismos de especificación existentes, el 80% de las especificaciones que analizaron corresponden a patrones simples como **response**[1], **universality**[2]y **absence**[3].
El trabajo presenta un catálogo de *patterns* de especificación siguiendo el espíritu de los *design patterns* de [GHJV95]. El catálogo indica como adaptar cada uno los *patterns* de especificación a algunos formalismos muy difundidos, incluyendo tanto formalismos basados en eventos como basados en estados. Un componente importante de estos *patterns* es el concepto de *contexto* (*scope* en inglés) que indica sobre que porción de la ejecución debe valer una propiedad. Ejemplos de contextos son: **global**, debe valer durante toda la ejecución; **before**, debe valer antes de cierto evento / estado y **between**, debe valer entre cierto estado / evento Q y cierto estado / evento R.
Aunque el uso de estos *patterns* de especificación facilita la tarea de especificación de propiedades, el catálogo se limita (intencionalmente) a un conjunto acotado y pequeño de *patterns* e incluye soporte para un número relativamente pequeño de formalismos. Por otro lado, los *patterns* soportados por este catálogo asumen un modelo no temporizado y por lo tanto no son adecuados para sistemas de tiempo real. La principal diferencia con nuestro enfoque es que el objetivo de ese trabajo no era desarrollar un nuevo lenguaje sino dar soporte al uso de los ya existentes.

En [UKM02] se introdujo un lenguaje para describir escenarios negativos basado en MSCs con el fin de capturar requerimientos. Algo similar, aunque con un propósito distinto, son los LSCs presentados en [DH99]. Aunque nuestro enfoque comparte la idea de trabajar con órdenes parciales para describir escenarios prohibidos, nuestro enfoque se diferencia de estos otros dos en varios aspectos. En primer lugar, usamos nuestro lenguaje como medio para expresar propiedades que serán verificadas contra un *modelo* o *implementación bajo análisis*. En segundo lugar, no estamos limitados a describir intercambio de mensajes ni tampoco instancias que generan eventos. Más aún, la visibilidad de los eventos es tratada en forma bastante distinta en nuestro trabajo. La consecutividad entre eventos no es una característica primitiva (es decir, dos eventos consecutivos en un patrón de eventos no necesariamente deben matchear ocurrencias consecutivas de dichos eventos en una ejecución). Por otro lado, en nuestro enfoque no es necesario recurrir a la notación *after/until* o utilizar *triggering conditions* para expresar cuando un matching es válido. En tercer lugar, nuestro lenguaje permite expresar restricciones temporales explícitas. A diferencia de LSCs, la semántica de nuestro patrones está dada en forma declarativa y el procedimiento de construcción del tableau muestra la existencia de una solución algorítmica al problema de verificación. Finalmente, nuestros patrones permiten expresar ciertos requerimientos de *liveness* como por ejemplo que un estímulo nunca sea respondido.

En [DKM+94] se propone un lenguaje gráfico llamado GIL (*Graphical Interval Logic*). El corazón del lenguaje son los *intervalos* en los que se evalúan distintas fórmulas lógicas. Más allá de este soporte incorporado para hablar de intervalos en una ejecución y de la estructuración gráfica de las fórmulas, el lenguaje hereda la mayor parte de la complejidad de las lógicas temporizadas y de intervalos. GIL utiliza un modelo no temporizado y asume un orden total entre los eventos del sistema, con lo cual no resulta adecuado para sistemas concurrentes y/o de tiempo real.

[3]Cierto estado / evento P debe estar siempre seguido del estado / evento Q
[3]Cierto estado / evento ocurre durante toda la duración de determinado contexto
[3]Cierto estado / evento no ocurre en determinado contexto

La herramienta TimeEdit[TIM01] desarrollada por Bell Labs tiene una filosofía similar a la de este trabajo desde el punto de vista de que busca simplificar la tarea de escribir escenarios que nunca deberían ocurrir, en este caso para las herramientas de verificación Spin[Hol97] y FeaVer. Sin embargo TimeEdit tampoco soporta restricciones temporales ni ordenamiento parcial de eventos.

Finalmente, en [AEY00, AY99] se usan *Message Sequence Charts* (MSCs) para especificar formalmente el comportamiento de los sistemas. Sin embargo, el enfoque de estos trabajos es distinto al nuestro: utilizan los MSCs (es decir, un formalismo gráfico de alto nivel) para modelar el sistema y utilizan formalismos existentes (en este caso autómatas) para expresar propiedades a verificar. En nuestro enfoque, el énfasis esta puesto en facilitar la tarea de especificar y verificar formalmente propiedades del sistema. Los MSCs asumen un orden parcial entre los eventos del sistema, permitiendo una representación más abstracta y compacta de la concurrencia y de las dependencias de causalidad, característica que comparten con nuestros patrones de eventos. Nuevamente, en estos trabajos se usa un modelo no temporizado con lo cual tampoco se brinda un soporte fuerte a la verificación de sistemas de tiempo real.

7.3 Trabajo futuro

Como resultado de este trabajo hemos establecido una base sólida sobre la que se abren numerosas posibilidades de investigación futura.

Uno de los puntos teóricos que quedan pendientes es la caracterización del poder expresivo de los patrones de eventos y la determinación de una cota precisa de la complejidad de la verificación de patrones de mal comportamiento.

Teniendo en cuenta el objetivo de la transferencia de tecnología hacia la industria de desarrollo de sistemas, creemos que es fundamental construir una herramienta de edición gráfica para los patrones de eventos. La notación gráfica presentada en este trabajo tiene carácter de prototipo y creemos que es necesario trabajar sobre una notación que resulte visualmente atractiva y cuya semántica resulte también intuitiva. Otro punto interesante para explorar es la utilización de asistentes (*wizards*) para orientar a los diseñadores en la construcción de patrones complejos (esta idea fue aplicada en [SACO02] a la construcción de autómatas temporizados).

Entendemos que una parte del trabajo futuro está relacionada con el análisis de extensiones para los patrones de eventos y otra parte está relacionada con la exploración de otras aplicaciones para los mismos.
Una de las posibles extensiones consiste en permitir predicar sobre el *primer* o el *último* evento ocurrido entre un conjunto de eventos no relacionados causalmente entre sí. Esta extensión agregaría poder expresivo a los patrones. Otras posibles extensiones serían la incorporación de *proposiciones* a los patrones de eventos, la modularización de patrones y la posibilidad de definir patrones paramétricos.

Este trabajo estuvo orientado a la utilización de patrones de eventos para verificar formalmente la ausencia de comportamientos no deseados en un sistema. Creemos interesante investigar la aplicación de patrones de eventos en el monitoreo de sistemas en ejecución. En ese contexto, podrían utilizarse los patrones de eventos como *oráculos*, contrastándolos con los logs que genera el sistema para ver si se alcanza alguna situación no deseada (similar al enfoque de [BOB03]).

Otra área en la que creemos que podría resultar útil la aplicación de patrones de eventos es en la restricción de modelos de un sistema. Los patrones de eventos podrían ser utilizados para restringir el conjunto de ejecuciones generadas por una modelización del sistema. En ese sentido, usaríamos patrones de eventos para expresar cosas tales como: "las ejecuciones que verifican el patrón $\mathcal{P}$ no se corresponden con la realidad".

Por otro lado, este trabajo se orienta a la verificación de sistemas con semánticas de *interleaving*, concretamente a sistemas modelados usando autómatas temporizados. Quiere decir que si bien los patrones de eventos brindan un soporte sintáctico a la especificación de propiedades de concurrencia y causalidad, este

soporte se pierde a nivel semántico y en el algoritmo de verificación. Resulta interesante explorar otras semánticas posibles para los patrones de eventos que permitan conservar la representación compacta de la concurrencia y que permitan, además, aplicar los patrones en la verificación de modelos con estructura de orden parcial (por ejemplo, sistemas modelados con MSCs Graphs).

Finalmente, en este trabajo mostramos un método de verificación de patrones de mal comportamiento basado en la construcción de un autómata reconocedor. En este punto se abren dos caminos posibles igualmente interesantes: por un lado, explorar otros métodos de verificación específicamente diseñados para los patrones de eventos que permitan utilizar eficientemente las características propias de los patrones (por ejemplo, para obtener algoritmos más eficientes en tiempo y espacio) y por otro lado, revisar la construcción de los autómatas reconocedores teniendo en cuenta las herramientas existentes para optimización de autómatas temporizados por reducción del número de relojes o del número de locaciones (por ejemplo [BGO02, DY96]).

Apéndice A

Demostraciones

A.1 Propiedades de la composición paralela de autómatas temporizados

Proposición 2.1. Dados dos autómatas temporizados $\mathcal{A}_i = \langle S_i, X_i, \Sigma_i, A_i, \mathcal{I}_i, s_{0i} \rangle$ con $i = 1, 2$ y una ejecución σ sobre $\Sigma_1 \cup \Sigma_2$,

$$\sigma \in \mathcal{L}(\mathcal{A}_1 || \mathcal{A}_2) \quad \text{sii} \quad \sigma\lceil_{\Sigma_1} \in \mathcal{L}(\mathcal{A}_1) \text{ y } \sigma\lceil_{\Sigma_2} \in \mathcal{L}(\mathcal{A}_2)$$

Demostración. Queremos ver que existe una forma de evolucionar siguiendo σ en $\mathcal{A}_1||\mathcal{A}_2$ si y sólo si existe una forma de evolucionar en $\mathcal{A}_1$ siguiendo $\sigma\lceil_{\Sigma_1}$ y en $\mathcal{A}_2$ siguiendo $\sigma\lceil_{\Sigma_2}$. Quiere decir que para una ejecución $\sigma = \langle \varsigma_0\varsigma_1 \ldots \varsigma_i \ldots, t_0 \lhd t_1 \lhd \ldots \lhd t_i \lhd \ldots \rangle$ deben existir las sucesiones infinitas de estados $\{q_n\}$ en $\mathcal{A}_1||\mathcal{A}_2$, $\{q_n^1\}$ en $\mathcal{A}_1$ y $\{q_n^2\}$ tales que:

$$q_i \Rightarrow_{t_i}^{\varsigma_i} q_{i+1} \tag{A.1}$$

sii

$$q_i^1 \Rightarrow_{t_i}^{a_i} q_{i+1}^1 \quad \text{y} \quad q_i^2 \Rightarrow_{t_i}^{b_i} q_{i+1}^2 \tag{A.2}$$

donde

$$a_i = \begin{cases} \varsigma_i & \text{si } \varsigma_i \in \Sigma_1 \\ \lambda & \text{en otro caso} \end{cases}$$

$$b_i = \begin{cases} \varsigma_i & \text{si } \varsigma_i \in \Sigma_2 \\ \lambda & \text{en otro caso} \end{cases}$$

Ahora bien, por definición de $||$, habrá una evolución en un paso $q \Rightarrow_t^a q'$ en $\mathcal{A}_1||\mathcal{A}_2$ si y sólo si existen las evoluciones en un paso:

$$\begin{cases} \pi_1(q) \Rightarrow_t^a \pi_1(q') & \text{si } a \in \Sigma_1 \\ \pi_1(q) \Rightarrow_t^\lambda \pi_1(q') & \text{en otro caso} \end{cases} \quad \text{en } \mathcal{A}_1$$

$$\begin{cases} \pi_2(q) \Rightarrow_t^a \pi_2(q') & \text{si } a \in \Sigma_2 \\ \pi_2(q) \Rightarrow_t^\lambda \pi_2(q') & \text{en otro caso} \end{cases} \quad \text{en } \mathcal{A}_2$$

Quiere decir que si existe una sucesión de estados $\{q_n\}$ en $\mathcal{A}_1||\mathcal{A}_2$ que verifique (A.1), entonces las sucesiones $\{q_n^1\}$ y $\{q_n^2\}$ tales que $q_i^1 = \pi_1(q_i)$ y $q_i^2 = \pi_2(q_i)$ para todo i, verifican (A.2). Por otro lado, si existen por separado $\{q_n^1\}$ y $\{q_n^2\}$ en $\mathcal{A}_1$ y $\mathcal{A}_2$, respectivamente, que verifique (A.2) entonces la sucesión $\{q_n\}$ tal que $q_i = (q_i^1, q_i^2)$ para todo i verifica (A.1).

□

A.2 Propiedades del producto de autómatas de Büchi temporizados

Proposición 2.2. Dados dos autómatas temporizados $\mathcal{A}_i = \langle S_i, X_i, \Sigma_i, A_i, \mathcal{I}_i, s_{0i} \rangle$, con $i = 1, 2$, un autómata de Büchi temporizado $\mathcal{B} = \langle \mathcal{A}_2, \mathcal{F} \rangle$, con $\mathcal{F} \subseteq S_2$ y una ejecución σ sobre $\Sigma_1 \cup \Sigma_2$,

$$\sigma \in \mathcal{L}(\mathcal{A}_1 \otimes \mathcal{B}) \quad \text{sii} \quad \sigma\restriction_{\Sigma_1} \in \mathcal{L}(\mathcal{A}_1) \text{ y } \sigma\restriction_{\Sigma_2} \in \mathcal{L}(\mathcal{B})$$

Demostración. $\mathcal{A}_1 \otimes \mathcal{B}$ es el autómata de Büchi $\langle \mathcal{A}_1 || \mathcal{A}_2, S_1 \times \mathcal{F} \rangle$. Quiere decir que σ será aceptada por $\mathcal{A}_1 \otimes \mathcal{B}$ si y sólo si pertenece al lenguaje $\mathcal{L}(\mathcal{A}_1 || \mathcal{A}_2)$ y además existe una forma de evolucionar siguiendo σ en $\mathcal{A}_1 || \mathcal{A}_2$ de forma tal de visitar un número infinito de veces locaciones en $S_1 \times \mathcal{F}$. Por la Observación 2.1, sabemos que σ estará en el lenguaje $\mathcal{L}(\mathcal{A}_1 || \mathcal{A}_2)$ si y sólo si $\sigma\restriction_{\Sigma_1}$ pertenece a $\mathcal{L}(\mathcal{A}_1)$ y $\sigma\restriction_{\Sigma_2}$ pertenece a $\mathcal{L}(\mathcal{A}_2)$. Por otro lado, que exista una forma de evolucionar siguiendo en σ en $\mathcal{A}_1 || \mathcal{A}_2$ que verifique la condición de aceptación de $\mathcal{A}_1 \otimes \mathcal{B}$ es equivalente a pedir que haya una forma de evolucionar en $\mathcal{A}_2$ siguiendo $\sigma\restriction_{\Sigma_2}$ de forma tal de visitar un número infinito de veces locaciones en $\mathcal{F}$ (porque implícitamente, todas las locaciones de $\mathcal{A}_1$ son *locaciones de aceptación*). Finalmente, que $\sigma\restriction_{\Sigma_2} \in \mathcal{L}(\mathcal{A}_2)$ y que sea posible evolucionar en $\mathcal{A}_2$ siguiendo $\sigma\restriction_{\Sigma_2}$ y verificando la condición de aceptación de $\mathcal{B}$ equivale a decir que $\sigma\restriction_{\Sigma_2} \in \mathcal{L}(\mathcal{B})$. □

A.3 Propiedades de la satisfacción de patrones básicos

Propiedad 3.1. *Clausura por extensiones.*
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ y una ejecución finita ς sobre Σ, si $\varsigma \vDash \mathcal{P}$ entonces para cualquier ejecución ς' sobre Σ, $\varsigma\varsigma' \vDash \mathcal{P}$.

Demostración. Supongamos que $\varsigma \vDash \mathcal{P}$. Eso quiere decir que existe al menos un matching básico $\hat{\cdot}$ entre $\mathcal{P}$ y ς, es decir, un mapeo que verifica las condiciones **M1**-**M3**. Dado que estas tres condiciones sólo predican sobre posiciones de ς, es fácil ver que $\hat{\cdot}$ con el codominio ampliado a $\Pi_{\varsigma\varsigma'}$ será también un matching básico entre $\varsigma\varsigma'$ y $\mathcal{P}$, sin importar cual sea ς'. □

Propiedad 3.2. *Satisfacción finita.*
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ y una ejecución ς sobre Σ, si $\varsigma \vDash \mathcal{P}$ entonces existe una posición $i \in \Pi_\varsigma$ tal que $\varsigma_{i]} \vDash \mathcal{P}$.

Demostración. Supongamos que $\varsigma \vDash \mathcal{P}$. Eso quiere decir que existe al menos un matching básico $\hat{\cdot}$ entre $\mathcal{P}$ y ς. Sea $n = \max_{x \in P} \hat{x}$, o sea, el máximo de la imagen de $\hat{\cdot}$.

Por la forma en que elegimos a n, todas las posiciones a las cuales $\hat{\cdot}$ mapea puntos del patrón están incluidas en el prefijo $\varsigma_{n]}$. Esto significa que $\hat{\cdot}$ verifica trivialmente las condiciones **M1** - **M3** aplicadas a $\varsigma_{n]}$ y a $\mathcal{P}$, dado que en ninguna de esas condiciones se hace referencia a posiciones de la ejecución que estén más allá de la última posición resaltada. Quiere decir que $\hat{\cdot}$ es un matching básico entre $\varsigma_{n]}$ y $\mathcal{P}$ y, por lo tanto, $\varsigma_{n]} \vDash \mathcal{P}$. □

A.4 Propiedades de la satisfacción de patrones temporizados

Propiedad 3.3. *Clausura por extensiones.*
Dado un patrón $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ y una ejecución finita σ sobre Σ, si $\sigma \vDash \mathcal{P}$ entonces para cualquier ejecución σ' sobre Σ, $\sigma\sigma' \vDash \mathcal{P}$.

Demostración. La demostración de esta propiedad es similar a la demostración de la Propiedad 3.1 para el caso no temporizado.

Como en ese caso, decir que una ejecución $\sigma \vDash \mathcal{P}$ significa que existe al menos un matching temporizado $\hat{\cdot}$ entre $\mathcal{P}$ y σ y esto, a su vez, significa que existe un mapeo $\hat{\cdot}$ que verifica las condiciones **M1-M3 + MT2**. Nuevamente, estas condiciones aplicadas a $\hat{\cdot}$ sólo predican sobre posiciones de σ, con lo cual el mismo mapeo pero con el codominio ampliado a $\Pi_{\sigma\sigma'}$ será un matching temporizado entre $\sigma\sigma'$ y $\mathcal{P}$, sin importar cual sea σ'.

□

Propiedad 3.4. *Satisfacción finita.*
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ y una ejecución temporizada σ sobre Σ, si $\sigma \vDash \mathcal{P}$ entonces existe una posición $i \in \Pi_\sigma$ tal que $\sigma_{i]} \vDash \mathcal{P}$.

Demostración. Siguiendo el razonamiento que hicimos en la página 93, en la demostración de la Propiedad 3.2 para el caso no temporizado, supongamos que $\sigma \vDash \mathcal{P}$, y $\hat{\cdot} : P \mapsto \Pi_\sigma$ es uno de los matchings temporizados entre σ y $\mathcal{P}$.

Es fácil ver que $\hat{\cdot}$ verifica trivialmente las condiciones **M1-M3 + MT2** aplicadas a $\sigma_{n]}$ y $\mathcal{P}$, donde $n = \max_{x \in P} \hat{x}$, o sea, el máximo de la imagen de $\hat{\cdot}$. Como $\sigma_{n]}$ es un prefijo finito de σ, queda demostrada la propiedad. □

A.5 Propiedades de la satisfacción de patrones de eventos

Propiedad 3.5. *Clausura débil por extensiones.*
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ y una ejecución temporizada finita σ sobre Σ, si $\sigma \vDash \mathcal{P}$ entonces para cualquier ejecución $\sigma' = \langle \varsigma, \tau \rangle$ tal que $\varsigma \cap \mathbb{L}_\mathcal{P} = \varnothing$, $\sigma\sigma' \vDash \mathcal{P}$.

Demostración. Para demostrar esta propiedad vamos a seguir la idea usada en las demostraciones de la propiedad clausura por extensiones para el caso sin temporizar (Propiedad 3.1) y el caso temporizado (Propiedad 3.3). Como hicimos en esos casos, vamos a suponer que $\sigma \vDash \mathcal{P}$ y que $\hat{\cdot}$ es un matching entre $\mathcal{P}$ y σ y vamos a mostrar la forma de construir un matching $\tilde{\cdot}$ entre $\sigma\sigma'$ y $\mathcal{P}$ (suponiendo que σ' cumple con las hipótesis de la propiedad).

Sea $\sigma' = \langle \varsigma', \tau' \rangle$ una ejecución sobre Σ tal que $\varsigma' \cap \mathbb{L}_\mathcal{P} = \varnothing$. Sea $\tilde{\cdot}$ el mapeo $\hat{\cdot}$ pero con el codominio ampliado a $\Pi_{\sigma\sigma'}$.
Es fácil ver que $\tilde{\cdot}$ verifica **M1-M3+MT2+MI+MTI** por las mismas razones expuestas en los casos anteriores. Sin embargo, no es tan directo ver que $\tilde{\cdot}$ verifica **MS**.

Sabemos que para todo punto $x \in P$, $(\varsigma\varsigma')_{(\tilde{x}} = \varsigma_{(\hat{x}}\varsigma'$. También sabemos que:

$$\forall i,\ \hat{x} < i < |\sigma|,\ \varsigma_i \notin \gamma(x, \infty) \tag{A.3}$$

porque $\hat{\cdot}$ verifica **MS** y por hipótesis:

$$\forall i,\ 0 \leq i < |\sigma'|,\ \varsigma'_i \notin \mathbb{L}_\mathcal{P} \tag{A.4}$$

y, en particular, $\varsigma'_i \notin \gamma(x, \infty)$.

Quiere decir que para todo punto $x \in P$, $\varsigma_{(\hat{x}}\varsigma' \cap \gamma(x, \infty) = \varnothing$ y por lo tanto $\tilde{\cdot}$ también verifica **MS**.

Dado que mostramos que $\tilde{\cdot}$ es un matching entre $\sigma\sigma'$ y $\mathcal{P}$, $\sigma\sigma' \vDash \mathcal{P}$.

□

Propiedad 3.6. *Satisfacción finita.*
Dado un patrón $\mathcal{P} = \langle \Sigma, E, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ y una ejecución temporizada σ sobre Σ, si $\sigma \vDash \mathcal{P}$ entonces existe una posición $i \in \Pi_\sigma$ tal que $\sigma_{i]} \vDash \mathcal{P}$ y, además, $\sigma_{(i} \cap \mathbb{L}_\mathcal{P} = \varnothing$.

Demostración. Siguiendo el razonamiento usado para demostrar la propiedad de Satisfacción finita para los casos no temporizado y temporizado, supongamos que $\sigma \vDash \mathcal{P}$, y $\hat{\cdot} : P \mapsto \Pi_\sigma$ es uno de los matchings entre σ y $\mathcal{P}$.

Es fácil ver que $\hat{\cdot}$ verifica trivialmente las condiciones **M1-M3** + **MT2** + **MI** + **MTI** aplicadas a $\sigma_{n]}$ y $\mathcal{P}$, donde $n = \max_{x \in P} \hat{x}$, o sea, el máximo de la imagen de $\hat{\cdot}$. Es también bastante directo ver que lo mismo ocurre con **MS**. Dado que sabemos que para todo punto $x \in P$, $\varsigma_{(\hat{x}} \cap \gamma(x, \infty) = \varnothing$, en particular debe valer que $\varsigma_{(\hat{x},n]} \cap \gamma(x, \infty) = \varnothing$.

Para ver que, además, $\varsigma_{(n} \cap \mathbb{L}_\mathcal{P} = \varnothing$, podemos observar que dado que $\hat{\cdot}$ verifica **MS**, para todo x, $\varsigma_{(\hat{x}} \cap \gamma(x, \infty) = \varnothing$. Pero también debe pasar que para todo x, $\varsigma_{(n} \cap \gamma(x, \infty) = \varnothing$, y por lo tanto, $\varsigma_{(n} \cap \mathbb{L}_\mathcal{P} = \varnothing$. □

A.6 Propiedades de autómatas reconocedores de patrones básicos

Propiedad 4.1. Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, sean Θ y Θ' configuraciones de $\mathcal{P}$. Si $[\Theta] \Rightarrow [\Theta']$, entonces toda evolución entre $[\Theta]$ y $[\Theta']$ tiene la forma:

$$([\Theta_0], v_0) \Rightarrow_{t_0}^{a_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{a_1} \ldots ([\Theta_i], v_i) \Rightarrow_{t_i}^{a_i} \ldots \Rightarrow_{t_{n-1}}^{a_{n-1}} ([\Theta_n], v_n)$$

donde $\Theta_0 = \Theta$, $\Theta_n = \Theta'$ y para todo i, $0 \leq i < n$, $v_i \in \mathcal{V}_X$, $a_i \in \Sigma \cup \{\lambda\}$, $t_i \in \Re^+$ y Θ_{i+1} **es igual a** Θ_i **o es una extensión de** Θ_i.

Demostración. Si existiera una evolución entre $[\Theta]$ y $[\Theta']$, tendría la forma:

$$([\Theta_0], v_0) \Rightarrow_{t_0}^{a_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{a_1} \ldots ([\Theta_i], v_i) \Rightarrow_{t_i}^{a_i} \ldots \Rightarrow_{t_{n-1}}^{a_{n-1}} ([\Theta_n], v_n)$$

donde $\Theta_0 = \Theta$, $\Theta_n = \Theta'$ y para todo i, $0 \leq i < n$, $v_i \in \mathcal{V}_X$, $a_i \in \Sigma \cup \{\lambda\}$, $t_i \in \Re^+$. Esto debe ser así porque s_{trap} no puede formar parte de ninguna evolución entre $[\Theta]$ y $[\Theta']$ por ser s_{trap} una locación trampa distinta de $[\Theta]$ y $[\Theta']$.
Queremos ver que Θ_{i+1} debe ser igual a Θ_i o debe ser una extensión de Θ_i.
Tomemos un i arbitrario, $0 \leq i < n$. Sabemos que $([\Theta_i], v_i) \Rightarrow_{t_i}^{a_i} ([\Theta_{i+1}], v_{i+1})$. Quiere decir que o bien existe una arista $\langle [\Theta_i], a_i, \psi, \rho, [\Theta_{i+1}] \rangle$ en $\mathcal{A}_\mathcal{P}$ o $a = \lambda$ y $[\Theta_i] = [\Theta_{i+1}]$. El segundo caso es trivial, veamos el primero. La arista entre $[\Theta_i]$ y $[\Theta_{i+1}]$ sólo puede ser *Mark* o *Skip*. Si pertenece a *Skip* entonces es un *loop* y por lo tanto $\Theta_i = \Theta_{i+1}$. Si, en cambio, pertenece a *Mark*, entonces necesariamente Θ_{i+1} debe ser una extensión de Θ_i. □

Propiedad 4.2. *Corrección del autómata reconocedor para patrones básicos*
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, para toda configuración $\Theta \in \Theta_\mathcal{P}$ y toda ejecución finita ς sobre Σ,

$$\text{si } q_{init} \Rightarrow^\varsigma [\Theta] \text{ entonces } \varsigma \vDash_\Theta \mathcal{P}$$

Demostración. Sea Θ una configuración de $\mathcal{P}$. Sea ς una ejecución finita. Supongamos que $q_{init} \Rightarrow^\varsigma [\Theta]$. Queremos ver que esto es una condición suficiente para que ς satisfaga parcialmente $\mathcal{P}$, restringido a Θ. Para ver que ésto es cierto, alcanza con demostrar que existe un matching básico parcial entre ς y $\mathcal{P}$ restringido a Θ. Construiremos un mapeo a partir de una evolución entre q_{init} y $[\Theta]$ y mostraremos que cumple las condiciones **MP1-MP4**.
Sea $n = |\varsigma|$. Por la Propiedad 4.1, una evolución entre q_{init} y s_{accept} etiquetada por ς debe ser de la forma:

$$q_{init} = ([\Theta_0], v_0) \Rightarrow_{t_0}^{\varsigma_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{\varsigma_1} \ldots ([\Theta_i], v_i) \Rightarrow_{t_i}^{\varsigma_i} \ldots \Rightarrow_{t_{n-1}}^{\varsigma_{n-1}} ([\Theta_n], v_n)$$

donde $[\Theta_0] = [\varnothing] = s_0$, $v_0 = \overline{0}$, $[\Theta_n] = [\Theta]$ y para todo i, $0 < i < n$, $v_i \in \mathcal{V}_X$, $t_i \in \Re^+$ y Θ_{i+1} es igual a Θ_i o es una extensión de Θ_i. En particular, sabemos que:

$$[\Theta_i] \Rightarrow^{\varsigma_i} [\Theta_{i+1}] \qquad 0 \leq i < n \tag{A.5}$$

Sea $\hat{\cdot}$ el mapeo construido según la siguiente regla:

$$\hat{x} = \max_{\substack{0 \leq i < n \\ x \notin \Theta_i}} i \qquad \text{para } x \in \Theta \tag{A.6}$$

Es decir, para un punto $x \in \Theta$, $\hat{x}$ determina la última locación de la evolución que no contiene a x y $[\Theta_{\hat{x}+1}]$ es la primera locación que si lo contiene.
Probemos ahora que $\hat{\cdot}$ cumple las condiciones de validez **MP1** - **MP4**, es decir:

MP1) Queremos ver que para todo punto $e \in \Theta \cap E$, $\varsigma_{\hat{e}} \in \ell(e)$.
Sea e un punto cualquiera en Θ. Por (A.6), $[\Theta_{\hat{e}}]$ es la última locación de la evolución que no contiene a e y $[\Theta_{\hat{e}+1}]$ es la primera locación que si lo contiene. Dado que por (A.5) sabemos que $[\Theta_{\hat{e}}] \Rightarrow^{\varsigma_{\hat{e}}} [\Theta_{\hat{e}+1}]$ y también sabemos que $[\Theta_{\hat{e}}] \neq [\Theta_{\hat{e}+1}]$ y $e \in E$, debe existir una arista *Mark* de la forma $\langle [\Theta_{\hat{e}}], \varsigma_{\hat{e}}, \psi, \rho, [\Theta_{\hat{e}+1}] \rangle$. Por la forma en que fue construido *Mark*, debe pasar que $\varsigma_{\hat{e}} \in \ell(e)$.

MP2) Queremos ver que para todo par de puntos $x, y \in \Theta$, si $x \prec y$ entonces $\hat{x} < \hat{y}$.
Sea x un punto cualquiera en Θ. Supongamos que existe un punto $y \in \Theta$ tal que $y \prec x$. Por (A.6), $[\Theta_{\hat{x}+1}]$ es la primera locación que contiene a x y $[\Theta_{\hat{y}+1}]$ es la primera locación que contiene a y. Dado que las locaciones contienen configuraciones (por la forma en que se construyó S), es necesario que $[\Theta_{\hat{y}+1}]$ aparezca en la evolución antes que $[\Theta_{\hat{x}+1}]$ (si no, $[\Theta_{\hat{x}+1}]$ contendría a x y no a y y no sería una configuración). Por lo tanto, necesariamente $\hat{y} + 1 < \hat{x} + 1$, lo que es equivalente a $\hat{y} < \hat{x}$.

MP3) Queremos ver que para todo par de puntos $x, y \in \Theta$, si $\hat{x} < \hat{y}$ entonces $\varsigma_{(\hat{x},\hat{y})} \cap \gamma(x, y) = \varnothing$.
Sean x, y dos puntos cualesquiera en Θ. Supongamos que $\hat{x} < \hat{y}$. Queremos ver que para toda posición i, $\hat{x} < i < \hat{y}$, $\varsigma_i \notin \gamma(x, y)$.
Sea i una posición cualquiera entre $\hat{x}$ y $\hat{y}$. Dado que $i > \hat{x}$, por (A.6), $x \in \Theta_i$. Análogamente, dado que $i < \hat{y}$, $y \notin \Theta_i$.
Por otro lado, por (A.5) sabemos que $[\Theta_i] \Rightarrow^{\varsigma_i} [\Theta_{i+1}]$. Esto quiere decir que o bien $\varsigma_i = \lambda$, con lo cual trivialmente no pertenece a $\gamma(x, y)$, o $\varsigma_i \in \Sigma$ y debe existir en A una arista de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$. Es fácil ver que si ese es el caso, dicha arista sólo puede ser *Skip* o *Mark*.
Supongamos que $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$ es una arista *Skip*. Esto implica, entre otras cosas, que $\varsigma_i \notin \Gamma(\Theta_i)$ y, por definición de Γ, esto quiere decir que dado que $x \in \Theta_i$ e $y \notin \Theta_i$, ς_i no puede aparecer en $\gamma(x, y)$.
Supongamos, ahora, que $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$ es una arista *Mark*. Esto implica, entre otras cosas, que $\Theta_{i+1} = \Theta_i \uplus \{e\}$, para algún punto $e \in E$ y que $\varsigma_i \notin \Gamma_{\triangleright e}(\Theta_i)$. Entonces, dado que $i + 1 \leq \hat{y}$, $y \notin \Theta_{i+1}$, con lo cual $e \neq y$. Dado que $x \in \Theta_i$, $y \notin \Theta_i$ y que $y \neq e$, necesariamente $\varsigma_i \notin \gamma(x, y)$.
En los tres casos llegamos a que $\varsigma_i \notin \gamma(x, y)$.

MP4) Por último, queremos ver que para todo par de puntos $x, y \in P$, si $x \in \Theta$, $y \in P \setminus \Theta$, entonces $\varsigma_{(\hat{x}} \cap \gamma(x, y) = \varnothing$.
Sea x un punto cualquiera en Θ. Sea y otro punto cualquiera en $P \setminus \Theta$. Queremos ver que para toda posición i, $\hat{x} < i < n$, $\varsigma_i \notin \gamma(x, y)$.
Sea i una posición cualquiera entre $\hat{x}$ y n. Dado que $i > \hat{x}$, por (A.6), $x \in \Theta_i$. Por otro lado, dado que todas las configuraciones del camino están incluidas en Θ y y está en el complemento de Θ, necesariamente $y \notin \Theta_i$. Siguiendo el mismo razonamiento que en el caso anterior, sabemos que o bien $\varsigma_i = \lambda$, con lo cual claramente no pertenece a $\gamma(x, y)$ o debe existir una arista de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$, que sólo podrá ser *Skip* o *Mark*.
Supongamos que existe una arista $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$ en *Skip*. Como dijimos antes, esto implica que $\varsigma_i \notin \Gamma(\Theta_i)$, y como vimos que $x \in \Theta_i$ y $y \notin \Theta_i$, entonces necesariamente $\varsigma_i \notin \gamma(x, y)$.
Supongamos, por otro lado, que existe una arista $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$ en *Mark*. Nuevamente, esto quiere decir que $\Theta_{i+1} = \Theta_i \uplus \{e\}$, para algún punto $e \in E$ y que $\varsigma_i \notin \Gamma_{\triangleright e}(\Theta_i)$. Dado que $e \in \Theta_{i+1} \subset \Theta$, y no puede ser igual a e. Como $x \in \Theta_i$, $y \notin \Theta_i$ e $y \neq e$, por def. de $\Gamma_{\triangleright e}$ necesariamente $\varsigma_i \notin \gamma(x, y)$.
En los tres casos, llegamos a que $\varsigma_i \notin \gamma(x, y)$.

Con esto queda demostrado que $\hat{\cdot}$ es un matching básico parcial entre ς y $\mathcal{P}$ restringido a Θ y en consecuencia $\varsigma \vDash_\Theta \mathcal{P}$.

□

Propiedad 4.3. *Completitud del autómata reconocedor para patrones básicos.*
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, para toda configuración $\Theta \in \Theta_\mathcal{P}$ y toda ejecución finita ς sobre Σ,

$$\text{si } \varsigma \vDash_\Theta \mathcal{P} \text{ entonces } q_{init} \Rightarrow^\varsigma [\Theta]$$

Demostración. Sea Θ una configuración de $\mathcal{P}$ y ς una ejecución finita sobre Σ. Supongamos que $\varsigma \vDash_\Theta \mathcal{P}$. Eso quiere decir que existe un matching básico parcial $\hat{\cdot}$ entre ς y $\mathcal{P}$ restringido a Θ. Para demostrar que se puede evolucionar desde q_{init} hacia $[\Theta]$ consumiendo ς, vamos a dar una forma de construir una evolución entre dichas locaciones a partir de $\hat{\cdot}$.
Sea $n = |\varsigma|$. Consideremos la siguiente secuencia de conjuntos de puntos:

$$\Theta_0, \Theta_1, \ldots, \Theta_n$$

donde:

$$\Theta_i = \{\mathsf{x} \in \Theta \mid \hat{\mathsf{x}} < i\}, \text{ para } 0 \leq i \leq n \tag{A.7}$$

Podemos observar que:

- $\Theta_0 = \varnothing$,

  porque ningún punto puede ser mapeado a una posición menor que 0.

- $\Theta_n = \Theta$,

  porque $\hat{\cdot}$ mapea puntos en posiciones de ς, es decir, entre 0 y $n-1$ y por lo tanto, para todo punto $\mathsf{x} \in \Theta$, $\hat{\mathsf{x}} < n$.

- Para todo i, $0 \leq i \leq n$, Θ_i es una configuración.

  Para ver que esto es cierto tenemos que ver que para cualquier punto $\mathsf{y} \in \Theta_i$, todo predecesor de y también pertenece a Θ_i. Dado $\mathsf{x} \prec \mathsf{y}$, por **MP2** sabemos que $\hat{\mathsf{x}} < \hat{\mathsf{y}}$. Quiere decir que $\hat{\mathsf{x}} < \hat{\mathsf{y}} < i$ y por lo tanto $\mathsf{x} \in \Theta_i$.

- Para todo i, $0 \leq i \leq n$, $[\Theta_i]$ es una locación de $\mathcal{A}_\mathcal{P}$.

  Porque vimos que los Θ_i son configuraciones.

- Cada Θ_{i+1}, con $0 \leq i < n$ es igual a Θ_i o es una extensión de Θ_i.

  Por la forma en que construimos los Θ_i, cada conjunto puede diferir del siguiente en a lo sumo aquellos puntos x para los cuales $\hat{\mathsf{x}} = i$. Pero como sabemos que $\hat{\cdot}$ es inyectiva eso quiere decir que o bien no existe ningún x tal que $\hat{\mathsf{x}} = i$, con lo cual $\Theta_{i+1} = \Theta_i$, o existe exactamente uno, llamémoslo p, y $\Theta_{i+1} = \Theta_i \uplus \{\mathsf{p}\}$.

- Para todo i, $0 \leq i < n$, si $\varsigma_i \neq \lambda$ entonces existe una arista de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$.

  Dado un i cualquiera, $0 \leq i < n$, sup $\varsigma_i \in \Sigma$.

  Si $\Theta_{i+1} = \Theta_i$, por **MP3** y **MP4** sabemos que $\varsigma_i \notin \Gamma(\Theta_i)$. Si este no fuera el caso, existirían $\mathsf{x} \in \Theta_i$ e $\mathsf{y} \notin \Theta_i = \Theta_{i+1}$ tales que $\varsigma_i \in \gamma(\mathsf{x}, \mathsf{y})$. Con lo cual, si $\mathsf{y} \in \Theta$ (es decir, $\hat{\mathsf{x}} < i < \hat{\mathsf{y}}$), $\hat{\cdot}$ violaría **MP3** y si $\mathsf{y} \in P \setminus \Theta$, como $\hat{\mathsf{x}} < i$, $\hat{\cdot}$ violaría **MP4**. Por lo tanto, debe existir una arista *Skip* de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_i] \rangle$

  Si $\Theta_{i+1} = \Theta_i \uplus \{\mathsf{p}\}$, por **MP3** y **MP4** sabemos que $\varsigma_i \notin \Gamma_{\triangleright \mathsf{p}}(\Theta_i)$. Si este no fuera el caso, existirían $\mathsf{x} \in \Theta_i$ e $\mathsf{y} \notin \Theta_i$ tales que $\mathsf{y} \neq \mathsf{p}$ y $\varsigma_i \in \gamma(\mathsf{x}, \mathsf{y})$. Con lo cual, si $\mathsf{y} \in \Theta$ (es decir, $\hat{\mathsf{x}} < i = \hat{\mathsf{p}} < \hat{\mathsf{y}}$), $\hat{\cdot}$ violaría **MP3** y si $\mathsf{y} \in P \setminus \Theta$, $\hat{\cdot}$ violaría **MP4**. Por otro lado, por **MP1** sabemos que $\mathsf{p} \in E$ y que $\varsigma_i = \varsigma_{\hat{\mathsf{p}}} \in \ell(\mathsf{p})$. Quiere decir debe existir una arista *Mark* de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_i \uplus \{\mathsf{p}\}] \rangle$.

- Para todo i, $0 \leq i < n$, si $\varsigma_i = \lambda$ entonces $[\Theta_i] = [\Theta_{i+1}]$.

  Dado un i cualquiera, $0 \leq i < n$, $\sup \varsigma_i = \lambda$. Si $\Theta_{i+1} = \Theta_i \uplus \{\mathsf{p}\}$, querría decir que $\hat{\mathsf{p}} = i$. Pero entonces $\hat{}$ violaría **MP1** porque $\lambda = \varsigma_i \notin \ell(\mathsf{p})$.

- Para todo i, $0 \leq i < n$, $([\Theta_i], \overline{\mathbf{0}}) \Rightarrow_0^{\varsigma_i} ([\Theta_{i+1}], \overline{\mathbf{0}})$.

  Dado un i cualquiera, $0 \leq i < n$, sabemos que existen la transición temporal $([\Theta_i], \overline{\mathbf{0}}) \rightarrow^0 ([\Theta_i], \overline{\mathbf{0}})$ y la transición discreta $([\Theta_i], \overline{\mathbf{0}}) \rightarrow^{\varsigma_i} ([\Theta_{i+1}], \overline{\mathbf{0}})$ y, por lo tanto, la evolución en un paso $([\Theta_i], \overline{\mathbf{0}}) \Rightarrow_0^{\varsigma_i} ([\Theta_{i+1}], \overline{\mathbf{0}})$.

Con estas observaciones en mente, podemos concluir que:

$$([\Theta_0], \overline{\mathbf{0}}) \Rightarrow_0^{\varsigma_0} ([\Theta_1], \overline{\mathbf{0}}) \Rightarrow_0^{\varsigma_1} \ldots \Rightarrow_0^{\varsigma_{n-1}} ([\Theta_n], \overline{\mathbf{0}})$$

lo cual implica que $q_{init} = ([\varnothing], \overline{\mathbf{0}}) = ([\Theta_0], \overline{\mathbf{0}}) \Rightarrow^{\varsigma} [\Theta_n] = [\Theta]$.

□

Teorema 4.4. *Tableau para patrones básicos.*
Dado un patrón básico $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, el autómata de Büchi temporizado $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$ reconoce el lenguaje $\mathcal{L}(\mathcal{P})$.

Demostración. La demostración del teorema está dividida en dos lemas. El Lema A.1 prueba que $\mathcal{L}^\star(\mathcal{T}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P})$ y, por el otro lado, el Lema A.2 prueba que $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}^\star(\mathcal{T}_\mathcal{P})$.

□

Lema A.1. *(Corrección del tableau)*
Dado un patrón básico $\mathcal{P}$ y su autómata reconocedor $\mathcal{A}_\mathcal{P}$ y sea el autómata de Büchi temporizado $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$,

$$\mathcal{L}^\star(\mathcal{T}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P})$$

Demostración. Queremos ver que $\mathcal{L}^\star(\mathcal{T}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P})$, es decir, que todas las ejecuciones aceptadas por $\mathcal{T}_\mathcal{P}$ corresponden sólo a ejecuciones aceptadas por $\mathcal{P}$.

Sea ς una ejecución en $\mathcal{L}^\star(\mathcal{T}_\mathcal{P})$. Por la definición de *lenguaje* de un autómata de Büchi temporizado, sabemos que existen $r \in \mathcal{R}^\infty(\mathcal{A}_\mathcal{P})$ y τ tales que $\overline{r} = \langle \varsigma, \tau \rangle$ y $s_{accept} \in inf(r)$.

Que r visite un número infinito de veces la locación s_{accept} implica, entre otras cosas, que en un número finito de pasos r alcanza s_{accept}. Quiere decir que existe un $i \in \mathbf{N}$ tal que:

$$q_{init} \Rightarrow_{\tau_{i]}}^{\varsigma_{i]}} s_{accept}$$

Por la propiedad de Corrección del autómata reconocedor para patrones básicos 4.2 sabemos que $\varsigma_{i]} \vDash_P \mathcal{P}$, y por lo tanto, $\varsigma_{i]} \vDash \mathcal{P}$. Como los patrones básicos son cerrados por extensiones (Propiedad 3.1) podemos extender este resultado a todo ς, es decir, $\varsigma \vDash \mathcal{P}$.

Finalmente, como r es divergente y por lo tanto infinita, ς también debe serlo y debe pasar que $\varsigma \in \mathcal{L}(\mathcal{P})$. □

Lema A.2. *(Completitud del tableau)*
Dado un patrón básico $\mathcal{P}$ y su autómata reconocedor $\mathcal{A}_\mathcal{P}$ y sea el autómata de Büchi temporizado $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$,

$$\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}^\star(\mathcal{T}_\mathcal{P})$$

Demostración. Queremos ver que $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}^*(\mathcal{T}_\mathcal{P})$, es decir, que todas las ejecuciones infinitas aceptadas por $\mathcal{P}$ son también aceptadas por $\mathcal{T}_\mathcal{P}$. Para ésto, tenemos que mostrar que para cualquier ejecución infinita ς aceptada por $\mathcal{P}$ existe una forma de evolucionar en $\mathcal{A}_\mathcal{P}$ consumiendo dicha ejecución de forma tal de pasar un número infinito de veces por la locación s_{accept}.

Veamos que es posible construir a partir de ς una evolución finita que permita llegar desde q_{init} hasta s_{accept} y una evolución infinita y divergente que permita quedarse para siempre en s_{accept}, verificando la condición de aceptación de $\mathcal{T}_\mathcal{P}$. Sea ς una ejecución tal que $\varsigma \in \mathcal{L}(\mathcal{P})$. Por definición, sabemos que $\varsigma \vDash \mathcal{P}$ y que $|\varsigma| = \infty$. Por la propiedad de satisfacción finita de patrones básicos (Propiedad 3.2), sabemos que existe una posición i de ς tal que $\varsigma_{i]} \vDash \mathcal{P}$. Sabemos que también vale que $\varsigma_{i]} \vDash_P \mathcal{P}$. Por la propiedad de Completitud del autómata reconocedor para patrones básicos 4.3, existe una evolución entre q_{init} y $[P]$ de la forma $q_{init} \Rightarrow^{\varsigma_{i]}} [P]$. Pero como $[P] = s_{accept}$:

$$q_{init} \Rightarrow^{\varsigma_{i]}} s_{accept}$$

Esto quiere decir que existen $v_1, \ldots, v_i \in \mathcal{V}_X$ y $t_0, \ldots, t_i \in \Re^+$ tales que:

$$r_1 = (s_0, \overline{0}) \Rightarrow_{t_0}^{\varsigma_0} (s_1, v_1) \Rightarrow_{t_1}^{\varsigma_1} \cdots \Rightarrow_{t_i}^{\varsigma_i} (s_{accept}, v_i)$$

Por otro lado, sabiendo que en s_{accept} existen transiciones *Skip* para todos los eventos y que el invariante es $\top$, existe la evolución infinita y divergente:

$$r_2 = (s_{accept}, v_i) \Rightarrow_1^{\varsigma_{i+1}} (s_{accept}, v_i + 1) \Rightarrow_1^{\varsigma_{i+2}} \cdots \Rightarrow_1^{\varsigma_{i+j}} (s_{accept}, v_i + j) \Rightarrow_1^{\varsigma_{i+j+1}} \cdots$$

Llamemos τ a la secuencia temporal finita de la forma $\tau = t_0 \triangleleft \cdots \triangleleft t_i$ y τ' a la secuencia temporal infinita de la forma $\tau' = 1 \triangleleft 1 \triangleleft \cdots$. Se puede ver que $\overline{r_1 r_2} = \langle \varsigma, \tau \triangleleft \tau' \rangle$, $r_1 r_2 \in \mathcal{R}^\infty(\mathcal{A}_\mathcal{P})$ y además $r_1 r_2$ visita s_{accept} infinitas veces. Con esto hemos demostrado que $\varsigma \in \mathcal{L}^*(\mathcal{T}_\mathcal{P})$. □

A.7 Propiedades de autómatas reconocedores de patrones temporizados

Proposición 4.5. *Preservación de verdad.* Dada una restricción temporal φ, para todo real no negativo t,

$$\psi_x(\varphi)[x|t] \text{ es verdadero sii } t \vDash \varphi$$

Demostración. (Por inducción en la complejidad de la restricción)
Si $\varphi \in \mathfrak{I}_\mathbf{N}$, entonces φ será de la forma:

$$\varphi = \{\alpha, \beta\} \tag{A.8}$$

donde $\alpha \in \mathbf{N}$ y $\beta \in \mathbf{N}$ o $\beta = \infty$ y $\{= (\ |\ [, \}\ =)\ |\ ]$. Quiere decir que $\psi_x(\varphi)$ será una restricción sobre x de la forma:

$$\alpha \sim x \wedge x \sim \beta \tag{A.9}$$

donde $\sim \in \{<, \leq\}$. Por lo tanto, $\psi_x(\varphi)[x|t]$ tendrá la forma:

$$\alpha \sim t \wedge t \sim \beta \tag{A.10}$$

con lo cual $\psi_x(\varphi)[x|t]$ será verdadero si y sólo si $t \in \{\alpha, \beta\}$, es decir, si y sólo si $t \vDash \varphi$.
Si $\varphi = \neg\theta$, con $\theta \in \mathfrak{I}_\mathbf{N}$, entonces $\psi_x(\varphi)[x|t] = \neg(\psi_x(\theta))[x|t] = \neg(\psi_x(\theta)[x|t])$. Por hipótesis inductiva, $\psi_x(\theta)[x|t]$ es verdadero si y sólo si $t \vDash \theta$, quiere decir que $\neg(\psi_x(\theta)[x|t])$ sii $t \nvDash \theta$, es decir, sii $t \vDash \varphi$.
□

Proposición 4.6. Dada una restricción temporal φ y un reloj x,

$$v \vDash \psi_x(\varphi) \text{ sii } v(x) \vDash \varphi$$

Demostración. Dado que x es el único reloj que aparece en $\psi_x(\varphi)$, sólo interesa $v(x)$ para determinar si v satisface o no $\psi_x(\varphi)$.
Es decir, $v \vDash \psi_x(\varphi)$ sii $\psi_x(\varphi)[x|v(x)]$ es verdadero, y por la Proposición 4.5, ésto último pasa sii $v(x) \vDash \varphi$. □

Propiedad 4.7. Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, sean Θ y Θ' configuraciones de $\mathcal{P}$. Si $[\Theta] \Rightarrow [\Theta']$, entonces toda evolución entre $[\Theta]$ y $[\Theta']$ tiene la forma:

$$([\Theta_0], v_0) \Rightarrow_{t_0}^{a_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{a_1} \ldots ([\Theta_i], v_i) \Rightarrow_{t_i}^{a_i} \ldots \Rightarrow_{t_{n-1}}^{a_{n-1}} ([\Theta_n], v_n)$$

donde $\Theta_0 = \Theta$, $\Theta_n = \Theta'$ y para todo i, $0 \leq i < n$, $v_i \in \mathcal{V}_X$, $a_i \in \Sigma \cup \{\lambda\}$, $t_i \in \Re^+$ y Θ_{i+1} **es igual a** Θ_i **o es una extensión de** Θ_i.

Demostración. Como vimos para el caso no temporizado, si existiera una evolución entre $[\Theta]$ y $[\Theta']$, tendría la forma:

$$([\Theta_0], v_0) \Rightarrow_{t_0}^{a_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{a_1} \ldots ([\Theta_i], v_i) \Rightarrow_{t_i}^{a_i} \ldots \Rightarrow_{t_{n-1}}^{a_{n-1}} ([\Theta_n], v_n)$$

donde $\Theta_0 = \Theta$, $\Theta_n = \Theta'$ y para todo i, $0 \leq i < n$, $v_i \in \mathcal{V}_X$, $a_i \in \Sigma \cup \{\lambda\}$, $t_i \in \Re^+$. Queremos ver que Θ_{i+1} debe ser igual a Θ_i o debe ser una extensión de Θ_i.
Dado un i arbitrario, $0 \leq i < n$, sabemos que $([\Theta_i], v_i) \Rightarrow_{t_i}^{a_i} ([\Theta_{i+1}], v_{i+1})$ y, por lo tanto, debe existir una arista de la forma $\langle [\Theta_i], a_i, \psi, \rho, [\Theta_{i+1}] \rangle$ o bien $a_i = \lambda$ y $[\Theta_i] = [\Theta_{i+1}]$. Además, en el primer caso la arista sólo puede ser *Mark*, *Instant* o *Skip*. En todos los caso, o bien $\Theta_i = \Theta_{i+1}$ o Θ_{i+1} debe ser una extensión de Θ_i. □

Propiedad 4.8. *Corrección del autómata reconocedor para patrones temporizados.*
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, para toda configuración $\Theta \in \Theta_\mathcal{P}$ y toda ejecución finita $\sigma = \langle \varsigma, \tau \rangle$ sobre Σ,

$$\text{si } q_{init} \Rightarrow_\tau^\varsigma [\Theta] \text{ entonces } \sigma \vDash_\Theta \mathcal{P}$$

Demostración. Sea Θ una configuración de $\mathcal{P}$, sea σ una ejecución finita y supongamos que $q_{init} \Rightarrow_\tau^\varsigma [\Theta]$. Siguiendo el mismo razonamiento que usamos en la demostración de la propiedad de Corrección para el caso no temporizado (página 95), mostraremos que existe un matching temporal parcial entre σ y $\mathcal{P}$ restringido a Θ. Construiremos un mapeo a partir de una evolución entre q_{init} y $[\Theta]$ y mostraremos que cumple las condiciones **MP1-MP4 + MPT1-MPT2**.
Sea $n = |\sigma|$. Sean $t_0, \ldots, t_{n-1} \in \Re^+$ tales que τ se pueda escribir como $\tau = t_0 \triangleleft t_1 \triangleleft \cdots \triangleleft t_{n-1}$. Dado que σ etiqueta alguna evolución entre q_{init} y $[\Theta]$, por la Propiedad 4.7, dicha evolución debe ser de la forma:

$$q_{init} = ([\Theta_0], v_0) \Rightarrow_{t_0}^{\varsigma_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{\varsigma_1} \ldots ([\Theta_i], v_i) \Rightarrow_{t_i}^{\varsigma_i} \ldots \Rightarrow_{t_{n-1}}^{\varsigma_{n-1}} ([\Theta_n], v_n) \tag{A.11}$$

donde $[\Theta_0] = [\varnothing] = s_0$, $v_0 = \overline{0}$, $[\Theta_n] = [\Theta]$ y para todo i, $0 < i < n$, $v_i \in \mathcal{V}_X$ y Θ_{i+1} es igual a Θ_i o es una extensión de Θ_i.
Como en el caso no temporizado, elijamos $\hat{\cdot}$ de la siguiente manera:

$$\hat{x} = \max_{\substack{0 \leq i < n \\ x \notin \Theta_i}} i \qquad \text{para } x \in \Theta \tag{A.12}$$

Es fácil ver que $\hat{\cdot}$ verifica **MP1-MP4** usando un razonamiento análogo al presentado en la página 95. Veamos que, además, $\hat{\cdot}$ verifica **MPT1** y **MPT2**.

MPT1) Queremos ver que para todo punto $x \in I$, $\hat{x} = \lambda$.
Sea i un punto cualquiera en $\Theta \cap I$. Por (A.12), $[\Theta_{\hat{i}}]$ es la última locación de la evolución que no contiene a i y $[\Theta_{\hat{i}+1}]$ es la primera locación que si lo contiene. Sabemos también que $[\Theta_{\hat{i}}] \Rightarrow^{\varsigma_{\hat{i}}} [\Theta_{\hat{i}+1}]$ y también sabemos

que $[\Theta_{\hat{i}}] \neq [\Theta_{\hat{i}+1}]$, debe existir una arista *Instant* de la forma $\langle [\Theta_{\hat{i}}], \varsigma_{\hat{i}}, \psi, \rho, [\Theta_{\hat{i}+1}] \rangle$. Por la forma en que fue definido el conjunto *Instant*, debe pasar que $\varsigma_{\hat{i}} = \lambda$.

MPT2) Queremos ver que para todo par de puntos $\mathsf{x}, \mathsf{y} \in \Theta$, $\hat{\mathsf{x}} < \hat{\mathsf{y}} \Rightarrow \Delta(\tau_{[\hat{\mathsf{x}},\hat{\mathsf{y}}]}) \vDash \delta(\mathsf{x}, \mathsf{y})$.
Sean x, y dos puntos cualesquiera en Θ. Supongamos que $\hat{\mathsf{x}} < \hat{\mathsf{y}}$. Por (A.12) sabemos que $\mathsf{y} \notin \Theta_{\hat{\mathsf{y}}}$ pero $\mathsf{y} \in \Theta_{\hat{\mathsf{y}}+1}$, con lo cual debe pasar que $\Theta_{\hat{\mathsf{y}}+1} = \Theta_{\hat{\mathsf{y}}} \uplus \{\mathsf{y}\}$. Por (A.11) sabemos que $([\Theta_{\hat{\mathsf{y}}}], v_{\hat{\mathsf{y}}}) \Rightarrow_{t_{\hat{\mathsf{y}}}}^{\varsigma_{\hat{\mathsf{y}}}} ([\Theta_{\hat{\mathsf{y}}+1}], v_{\hat{\mathsf{y}}+1}) = ([\Theta_{\hat{\mathsf{y}}} \uplus \{\mathsf{y}\}], v_{\hat{\mathsf{y}}+1})$, por lo tanto, debe existir una transición discreta entre $[\Theta_{\hat{\mathsf{y}}}]$ y $[\Theta_{\hat{\mathsf{y}}} \uplus \{\mathsf{y}\}]$ correspondiente a una arista de la forma $\langle [\Theta_{\hat{\mathsf{y}}}], \varsigma_{\hat{\mathsf{y}}}, \psi, \rho, [\Theta_{\hat{\mathsf{y}}} \uplus \{\mathsf{y}\}] \rangle$. Es fácil ver que dicha arista sólo puede pertenecer a *Mark* o a *Instant*, con lo cual $\psi = \psi^{\mathsf{y}}_{\Theta_{\hat{\mathsf{y}}}} = \bigwedge_{\mathsf{p} \in \Theta_{\hat{\mathsf{y}}}} \psi_{z_{\mathsf{p}}}(\delta(\mathsf{p}, \mathsf{y}))$ y $v_{\hat{\mathsf{y}}} + t_{\hat{\mathsf{y}}} \vDash \psi^{\mathsf{y}}_{\Theta_{\hat{\mathsf{y}}}}$.
En particular, $v_{\hat{\mathsf{y}}} + t_{\hat{\mathsf{y}}} \vDash \psi_{z_{\mathsf{x}}}(\delta(\mathsf{x}, \mathsf{y}))$. Usando la Proposición 4.6, sabemos que $v_{\hat{\mathsf{y}}} + t_{\hat{\mathsf{y}}} \vDash \psi_{z_{\mathsf{x}}}(\delta(\mathsf{x}, \mathsf{y}))$ equivale a que:

$$(v_{\hat{\mathsf{y}}} + t_{\hat{\mathsf{y}}})(z_{\mathsf{x}}) \vDash \delta(\mathsf{x}, \mathsf{y})$$

Análogamente, sabemos que $\Theta_{\hat{\mathsf{x}}+1} = \Theta_{\hat{\mathsf{x}}} \uplus \{\mathsf{x}\}$ y que existe una arista *Mark* o *Instant* de la forma $\langle [\Theta_{\hat{\mathsf{x}}}], \varsigma_{\hat{\mathsf{x}}}, \psi', \rho', [\Theta_{\hat{\mathsf{x}}} \uplus \{\mathsf{x}\}] \rangle$. Con lo cual, $\rho' = \{z_{\mathsf{x}}\}$.
Por la forma en que se construyeron las aristas de $\mathcal{A}_{\mathcal{P}}$, las únicas aristas que resetean relojes son las aristas *Mark* e *Instant*. Además, un reloj z_{x}, correspondiente al punto x, sólo se resetea cuando se *marca* x. Esto quiere decir que en cualquier evolución como la presentada en (A.11) un reloj se resetea a lo suma una vez. Quiere decir que z_{x} se resetea en la transición entre $[\Theta_{\hat{\mathsf{x}}}]$ y $[\Theta_{\hat{\mathsf{x}}+1}]$ y no se vuelve a resetear después de la posición $\hat{\mathsf{x}} + 1$. Por lo tanto:

$$\begin{array}{lcl} v_{\hat{\mathsf{x}}+1}(z_{\mathsf{x}}) & = & 0 \\ v_{\hat{\mathsf{x}}+2}(z_{\mathsf{x}}) & = & t_{\hat{\mathsf{x}}+1} \\ v_{\hat{\mathsf{x}}+3}(z_{\mathsf{x}}) & = & t_{\hat{\mathsf{x}}+1} + t_{\hat{\mathsf{x}}+2} \\ \vdots & & \vdots \\ v_{\hat{\mathsf{y}}}(z_{\mathsf{x}}) & = & t_{\hat{\mathsf{x}}+1} + \cdots + t_{\hat{\mathsf{y}}-1} \end{array}$$

Por último, observemos que $\Delta(\tau_{[\hat{\mathsf{x}},\hat{\mathsf{y}}]}) = \tau_{\hat{\mathsf{y}}} - \tau_{\hat{\mathsf{x}}} = t_{\hat{\mathsf{x}}+1} + \cdots + t_{\hat{\mathsf{y}}}$. Quiere decir que $\Delta(\tau_{[\hat{\mathsf{x}},\hat{\mathsf{y}}]}) = v_{\hat{\mathsf{y}}}(z_{\mathsf{x}}) + t_{\hat{\mathsf{y}}} = (v_{\hat{\mathsf{y}}} + t_{\hat{\mathsf{y}}})(z_{\mathsf{x}})$ y, por lo que acabamos de demostrar, $\Delta(\tau_{[\hat{\mathsf{x}},\hat{\mathsf{y}}]}) \vDash \delta(\mathsf{x}, \mathsf{y})$.
Con esto queda demostrado que $\hat{\cdot}$ es un matching temporal parcial entre σ y $\mathcal{P}$ restringido a Θ y en consecuencia $\sigma \vDash_{\Theta} \mathcal{P}$.

□

Propiedad 4.9. *Completitud del autómata reconocedor para patrones temporizados.*
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ y su autómata reconocedor $\mathcal{A}_{\mathcal{P}} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, para toda configuración $\Theta \in \Theta_{\mathcal{P}}$ y toda ejecución finita σ sobre Σ,

$$\text{si } \sigma \vDash_{\Theta} \mathcal{P} \text{ entonces } q_{init} \Rightarrow^{\sigma} [\Theta]$$

Demostración. Sea Θ una configuración de $\mathcal{P}$ y $\sigma = \langle \varsigma, \tau \rangle$ una ejecución finita sobre Σ. Supongamos, además que $\sigma \vDash_{\Theta} \mathcal{P}$. Como en el caso anterior, seguiremos el razonamiento usado en la demostración de la propiedad de Completitud para el caso no temporizado (página 97).
Dado que $\sigma \vDash_{\Theta} \mathcal{P}$, sabemos que existe un matching temporal parcial $\hat{\cdot}$ entre σ y $\mathcal{P}$ restringido a Θ. Para demostrar que se puede evolucionar desde q_{init} hacia $[\Theta]$ por σ, vamos a dar una forma de construir una evolución entre dichas locaciones a partir de $\hat{\cdot}$.
Sea $n = |\sigma|$. Consideremos la secuencia de conjuntos de eventos presentada en la página 97:

$$\Theta_0, \Theta_1, \ldots, \Theta_n$$

donde:

$$\Theta_i = \{\mathsf{x} \in \Theta \mid \hat{\mathsf{x}} < i\}, \text{ para } 0 \leq i \leq n \tag{A.13}$$

Podemos observar que en este caso también vale que:

- $\Theta_0 = \varnothing$
- $\Theta_n = \Theta$
- Para todo i, $0 \leq i \leq n$, Θ_i es una configuración
- Para todo i, $0 \leq i \leq n$, $[\Theta_i]$ es una locación de $\mathcal{A}_\mathcal{P}$
- Cada Θ_{i+1}, con $0 \leq i < n$ es igual a Θ_i o es una extensión de Θ_i
- Para todo i, $0 \leq i < n$, si $\varsigma_i \neq \lambda$ entonces existe una arista de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$.
- Para todo i, $0 \leq i < n$, si $\varsigma_i = \lambda$ entonces $[\Theta_i] = [\Theta_{i+1}]$ o existe una arista de la forma $\langle [\Theta_i], \lambda, \psi, \rho, [\Theta_{i+1}] \rangle$.

  Si $\varsigma_i = \lambda$ y $\Theta_{i+1} = \Theta_i$ vale trivialmente esta propiedad. Veamos ahora que pasa si $\varsigma_i = \lambda$ y $\Theta_{i+1} = \Theta_i \uplus \{\mathsf{p}\}$.

  Si se diera este caso, por **MPT1** sabemos que $\mathsf{p} \notin E$, porque los eventos no pueden ser mapeados en posiciones λ. Por lo tanto, $\mathsf{p} \in I$ y por definición de *Instant*, debe existir una arista de la forma $\langle [\Theta_i], \lambda, \psi, \rho, [\Theta_i \uplus \{\mathsf{p}\}] \rangle$.

Sea $t_0, \ldots, t_{n-1}$ la sucesión de números reales no negativos tal que τ se pueda escribir como $\tau = t_0 \lhd t_1 \lhd \cdots \lhd t_{n-1}$ y sea $v_0, \ldots, v_n$ la secuencia de valuaciones dada por:

$$\begin{aligned} v_0 &= \overline{0} \\ v_{i+1} &= \begin{cases} v_i + t_i & \text{si } \Theta_{i+1} = \Theta_i \\ Reset_{z_\mathsf{x}}(v_i + t_i) & \text{si } \Theta_{i+1} = \Theta_i \uplus \{\mathsf{x}\} \end{cases} \quad 0 \leq i < n \end{aligned} \tag{A.14}$$

Veamos que también vale que para todo i, $0 \leq i < n$, $([\Theta_i], v_i) \Rightarrow_{t_i}^{\varsigma_i} ([\Theta_{i+1}], v_{i+1})$.

Dado un i cualquiera, $0 \leq i < n$, sabemos que existe la transición temporal $([\Theta_i], v_i) \rightarrow^{t_i} ([\Theta_i], v_i + t_i)$ dado que el invariante de todas las locaciones del autómata es $\top$. Por otro lado, sabemos que o bien existe una arista de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$ o $\Theta_i = \Theta_{i+1}$ y $\varsigma_i = \lambda$. Si este último fuera el caso, trivialmente existiría la transición discreta $([\Theta_i], v_i + t_i) \rightarrow^{\lambda} ([\Theta_i], v_i + t_i)$, con lo cual existiría la evolución en un paso $([\Theta_i], v_i) \Rightarrow_{t_i}^{\varsigma_i} ([\Theta_{i+1}], v_{i+1})$. Supongamos, entonces, que se da el primer caso.
Si $\Theta_{i+1} = \Theta_i$, la arista sólo puede ser *Skip* y por lo tanto la guarda debe ser $\top$ y ρ debe ser vacío, con lo cual existe la transición discreta $([\Theta_i], v_i + t_i) \rightarrow^{\varsigma_i} ([\Theta_i], v_i + t_i)$ y también en este caso existe la evolución en un paso $([\Theta_i], v_i) \Rightarrow_{t_i}^{\varsigma_i} ([\Theta_{i+1}], v_{i+1})$.
Si, en cambio, $\Theta_{i+1} = \Theta_i \uplus \{\mathsf{p}\}$ para algún punto p, entonces debe existir una arista *Mark* e *Instant*. Por la forman en que fueron construidas esas aristas sabemos que $\psi = \psi^{\mathsf{p}}_{\Theta_i}$ y $\rho = \{z_\mathsf{p}\}$ y por (A.14) sabemos que $v_{i+1} = Reset_{z_\mathsf{p}}(v_i + t_i) = Reset_\rho(v_i + t_i)$. Quiere decir que para demostrar que existe la transición discreta $([\Theta_i], v_i + t_i) \rightarrow^{\varsigma_i} ([\Theta_i], v_{i+1})$, sólo faltaría ver que $v_i + t_i \vDash \psi^{\mathsf{p}}_{\Theta_i}$.
Supongamos que $v_i + t_i \nvDash \psi^{\mathsf{p}}_{\Theta_i}$. Dado que por definición $\psi^{\mathsf{p}}_{\Theta_i} = \bigwedge_{\mathsf{x} \in \Theta_i} \psi_{z_\mathsf{x}}(\delta(\mathsf{x}, \mathsf{p}))$, debe existir algún $\mathsf{x} \in \Theta_i$ tal que $v_i + t_i \nvDash \psi_{z_\mathsf{x}}(\delta(\mathsf{x}, \mathsf{p}))$ y, por la Propiedad 4.6, $(v_i + t_i)(z_\mathsf{x}) = (v_{\hat{\mathsf{p}}} + t_{\hat{\mathsf{p}}})(z_\mathsf{x}) \nvDash \delta(\mathsf{x}, \mathsf{p})$.
Por (A.13), $\Theta_{\hat{\mathsf{x}}+1} = \Theta_{\hat{\mathsf{x}}} \uplus \{\mathsf{x}\}$, con lo cual (A.14) nos dice que $v_{\hat{\mathsf{x}}+1} = Reset_{z_\mathsf{x}}(v_{\hat{\mathsf{x}}} + t_{\hat{\mathsf{x}}})$. Además, por la forma en que fue definida la secuencia $v_0, \ldots, v_n$, cada reloj z_x es *reseteado* una única vez en la secuencia (coincidiendo con el punto en donde se agrega x en la secuencia de configuraciones que definimos al principio):

$$\begin{aligned} v_{\hat{\mathsf{x}}+1}(z_\mathsf{x}) &= 0 \\ v_{\hat{\mathsf{x}}+2}(z_\mathsf{x}) &= t_{\hat{\mathsf{x}}+1} \\ v_{\hat{\mathsf{x}}+3}(z_\mathsf{x}) &= t_{\hat{\mathsf{x}}+1} + t_{\hat{\mathsf{x}}+2} \\ &\vdots \\ v_{\hat{\mathsf{p}}}(z_\mathsf{x}) &= t_{\hat{\mathsf{x}}+1} + \cdots + t_{\hat{\mathsf{p}}-1} \end{aligned}$$

Con lo cual, $v_{\hat{\mathsf{p}}}(z_\mathsf{x}) + t_{\hat{\mathsf{p}}} = (v_{\hat{\mathsf{p}}} + t_{\hat{\mathsf{p}}})(z_\mathsf{x}) = \tau_{\hat{\mathsf{p}}} - \tau_{\hat{\mathsf{x}}} = \Delta(\tau_{[\hat{\mathsf{x}}, \hat{\mathsf{p}}]})$. Quiere decir que $\Delta(\tau_{[\hat{\mathsf{x}}, \hat{\mathsf{p}}]}) \nvDash \delta(\mathsf{x}, \mathsf{p})$ y esto violaría la hipótesis de que $\hat{}$ era un matching temporal parcial.

Por lo tanto, debe ser el caso que $v_{\hat{p}} + t_{\hat{p}} \vDash \psi^{\mathsf{p}}_{\Theta_i}$ y debe existir la evolución en un paso $([\Theta_i], v_i) \Rightarrow^{\varsigma_i}_{t_i} ([\Theta_{i+1}], v_{i+1})$.
Con estas observaciones en mente, podemos concluir que:

$$([\Theta_0], v_0) \Rightarrow^{\varsigma_0}_{t_0} ([\Theta_1], v_1) \Rightarrow^{\varsigma_1}_{t_1} \dots \Rightarrow^{\varsigma_{n-1}}_{t_{n-1}} ([\Theta_n], v_n)$$

lo cual implica que $q_{init} = ([\Theta_0], v_0) \Rightarrow^{\varsigma}_{\tau} [\Theta_n] = [\Theta]$.

□

Teorema 4.10. *Tableau para patrones temporizados.*
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, el autómata de Büchi temporizado $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$ reconoce el lenguaje $\mathcal{L}(\mathcal{P})$.

Demostración. Como en el caso anterior, la demostración del teorema está dividida en dos lemas. El lema A.3 prueba que $\mathcal{L}(\mathcal{T}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P})$ y, por el otro lado, el lema A.4 prueba que $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{T}_\mathcal{P})$.

□

Lema A.3. *(Corrección del tableau)*
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P}$ y sea el autómata de Büchi temporizado $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$,

$$\mathcal{L}(\mathcal{T}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P})$$

Demostración. Queremos ver que $\mathcal{L}(\mathcal{T}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P})$, es decir, que todas las ejecuciones aceptadas por $\mathcal{T}_\mathcal{P}$ corresponden sólo a ejecuciones aceptadas por $\mathcal{P}$.

Sea $\sigma = \langle \varsigma, \tau \rangle$ una ejecución en $\mathcal{L}(\mathcal{T}_\mathcal{P})$. Como en el caso no temporizado, la definición de *lenguaje* de un autómata de Büchi temporizado implica que existe una forma de evolucionar desde q_{init}, consumiendo σ, de forma tal de visitar s_{accept} un número infinito de veces. Quiere decir que, en particular, debe existir una forma de evolucionar desde q_{init} hasta s_{accept} consumiendo un prefijo finito de σ:

$$q_{init} \Rightarrow^{\varsigma_{i]}}_{\tau_{i]}} s_{accept}$$

para algún $i \in \mathbf{N}$. Pero a su vez, esto implica por la propiedad de Corrección del autómata reconocedor para patrones temporizados 4.8 que $\sigma_{i]} \vDash_P \mathcal{P}$, y por lo tanto, $\sigma_{i]} \vDash \mathcal{P}$. Dado que la satisfacción de patrones temporizados es cerradas por extensiones (Propiedad 3.3) podemos concluir que $\sigma \vDash \mathcal{P}$.

Como r es divergente, σ debe serlo también, y con ésto hemos demostrado que $\sigma \in \mathcal{L}(\mathcal{P})$. □

Lema A.4. *(Completitud del tableau)*
Dado un patrón temporizado $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P}$ y sea el autómata de Büchi temporizado $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$,

$$\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{T}_\mathcal{P})$$

Demostración. Queremos ver que por cada ejecución divergente aceptada por $\mathcal{P}$, existe una forma de evolucionar en $\mathcal{A}_\mathcal{P}$ consumiendo dicha ejecución, de forma tal de pasar un número infinito de veces por s_{accept}.

Sea $\sigma = \langle \varsigma, \tau \rangle$ una ejecución tal que $\sigma \in \mathcal{L}(\mathcal{P})$. Por definición de $\mathcal{L}$, sabemos que $\sigma \vDash \mathcal{P}$ y que σ es divergente.

Por la propiedad de satisfacción finita de los patrones temporizados (Propiedad 3.4), sabemos que existe un prefijo finito de σ que satisface $\mathcal{P}$. Sea i una posición tal que $\sigma_{i]} \vDash \mathcal{P}$. Sabemos que entonces también vale que $\sigma_{i]} \vDash_P \mathcal{P}$. Por la propiedad de Completitud del autómata reconocedor para patrones temporizados 4.9 sabemos que esto último significa que:

$$q_{init} \Rightarrow^{\sigma_{i]}} s_{accept}$$

es decir, existe al menos una evolución, llamémosla r_1, que partiendo desde q_{init} alcance s_{accept}, consumiendo las primeras i posiciones de σ.

Siguiendo el mismo razonamiento que para el caso no temporizado, sabemos que existe al menos una evolución divergente r_2 que partiendo desde el último estado de r_1, permanezca para siempre en s_{accept} (cumpliendo de esa forma la condición de aceptación de $\mathcal{T}_\mathcal{P}$) y además $\overline{r_1 r_2} = \sigma$. Con esto queda demostrado que $\sigma \in \mathcal{L}(\mathcal{T}_\mathcal{P})$. □

A.8 Propiedades de autómatas reconocedores de patrones de eventos

Propiedad 4.12. *Corrección del autómata reconocedor para patrones de eventos.*
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, para toda configuración $\Theta \in \Theta_\mathcal{P}$ y toda ejecución finita $\sigma = \langle \varsigma, \tau \rangle$ sobre Σ,

$$\text{si } q_{init} \Rightarrow_\tau^\varsigma [\Theta] \text{ entonces } \sigma \vDash_\Theta \mathcal{P}$$

Demostración. Sea Θ una configuración de $\mathcal{P}$, sea σ una ejecución finita y supongamos que $q_{init} \Rightarrow_\tau^\varsigma [\Theta]$. Siguiendo el mismo razonamiento que usamos en la demostración de la propiedad de Corrección para los casos temporizado y no temporizado (páginas 59 y 95, respectivamente), mostraremos que existe un matching parcial entre σ y $\mathcal{P}$ restringido a Θ. Para ésto, construiremos un mapeo a partir de una evolución entre q_{init} y $[\Theta]$ y mostraremos que cumple las condiciones **MP1-MP4 + MPT1-MPT2 + MPI + MPS + MPTI**.

Sea $n = |\sigma|$. Sean $t_0, \ldots, t_{n-1} \in \Re^+$ tales que τ se pueda escribir como $\tau = t_0 \triangleleft t_1 \triangleleft \cdots \triangleleft t_{n-1}$. Dado que σ etiqueta alguna evolución entre q_{init} y $[\Theta]$, por la Propiedad 4.11, dicha evolución debe ser de la forma:

$$q_{init} = ([\Theta_0], v_0) \Rightarrow_{t_0}^{\varsigma_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{\varsigma_1} \ldots ([\Theta_i], v_i) \Rightarrow_{t_i}^{\varsigma_i} \ldots \Rightarrow_{t_{n-1}}^{\varsigma_{n-1}} ([\Theta_n], v_n) \tag{A.15}$$

donde $[\Theta_0] = [\varnothing] = s_0$, $v_0 = \overline{\mathbf{0}}$, $[\Theta_n] = [\Theta]$ y para todo i, $0 < i < n$, $v_i \in \mathcal{V}_X$ y Θ_{i+1} es igual a Θ_i o es una extensión de Θ_i. En particular, sabemos que:

$$[\Theta_i] \Rightarrow^{\varsigma_i} [\Theta_{i+1}] \qquad 0 \leq i < n \tag{A.16}$$

Como en los casos anteriores, elijamos $\hat{\cdot}$ de la siguiente manera:

$$\hat{\mathrm{x}} = \max_{\substack{0 \leq i < n \\ \mathrm{x} \notin \Theta_i}} i \qquad \text{para } \mathrm{x} \in \Theta \tag{A.17}$$

Es fácil ver que $\hat{\cdot}$ verifica **MP1-MP4 + MPT1-MPT2** usando un razonamiento análogo al presentado en las páginas 95 y 59. Veamos que, además, $\hat{\cdot}$ verifica **MPI + MPS + MPTI**.

MPI) Queremos ver que para todo punto $x \in \Theta$, $\varsigma_{\hat{x})} \cap \gamma(\mathbf{0}, x) = \varnothing$.

Sea $x \in \Theta$. Queremos ver que para todo i, $0 \leq i < \hat{x}$, $\varsigma_i \notin \gamma(\mathbf{0}, x)$. Sea i cualquiera en el rango mencionado. Sabemos que $x \notin \Theta_{i+1}$ porque $i + 1 \leq \hat{x}$. Además, por (A.16) sabemos que o bien existe una arista de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$ o $\Theta_i = \Theta_{i+1}$ y $\varsigma_i = \lambda$.

Si $\varsigma_i = \lambda$, listo, porque $\lambda \notin \gamma(\mathbf{0}, x) \subseteq \Sigma$. Si, en cambio, $\varsigma_i \in \Sigma$ y existe una arista $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$ en el autómata, entonces sólo puede pasar que dicha arista sea *Skip* o *Mark*.

Si la arista es *Skip* entonces necesariamente $\varsigma_i \notin \Gamma^\star(\Theta_i)$ y dado que $x \notin \Theta_i$ y $\varsigma_i \notin \Gamma^\star(\Theta_i)$, sólo puede pasar que $\varsigma_i \notin \gamma(\mathbf{0}, x)$.

Si la arista es *Mark* entonces $\Theta_{i+1} = \Theta_i \uplus \{y\}$, para algún evento y y necesariamente $\varsigma_i \notin \Gamma^\star_{\triangleright y}(\Theta_i)$. Dado que $\hat{y} = i < \hat{x}$, $y \neq x$, $x \notin \Theta_i$ y $\varsigma_i \notin \Gamma^\star_{\triangleright y}(\Theta_i)$, entonces sólo puede pasar que $\varsigma_i \notin \gamma(\mathbf{0}, x)$.

En todos los casos llegamos a que $\varsigma_i \notin \gamma(\mathbf{0}, x)$.

MPS) Queremos ver que para todo punto $x \in \Theta$, $\varsigma_{(\hat{x}} \cap \gamma(x, \infty) = \varnothing$.

Sea $x \in \Theta$. Queremos ver que para todo i, $\hat{x} < i < n$, $\varsigma_i \notin \gamma(x, \infty)$. Sea i cualquiera en ese rango. Sabemos que $x \in \Theta_i$. Por (A.16) también sabemos o bien existe una arista de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$ o $\Theta_i = \Theta_{i+1}$ y $\varsigma_i = \lambda$.

Si $\varsigma_i = \lambda$, listo, porque $\lambda \notin \gamma(x, \infty) \subseteq \Sigma$. Si $\varsigma_i \in \Sigma$, entonces sólo puede pasar que exista una arista *Skip* o *Mark* de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$.

Si la arista es *Skip*, entonces $\varsigma_i \notin \gamma(x, \infty)$ (porque $\varsigma_i \notin \Gamma^\star(\Theta_i)$ y $x \in \Theta_i$).

Si la arista es *Mark* entonces $\Theta_{i+1} = \Theta \uplus \{y\}$, para algún punto y. Con lo cual tenemos que $x \in \Theta_i$ y $\varsigma_i \notin \Gamma^\star_{\triangleright y}(\Theta_i)$, luego sólo puede pasar que $\varsigma_i \notin \gamma(x, \infty)$.

En todos los casos tenemos que $\varsigma_i \notin \gamma(x, \infty)$.

MPTI) Queremos ver que para todo punto $x \in \Theta$, $\Delta(\tau_{\hat{x}]}) \vDash \delta(\mathbf{0}, x)$.

Sea $x \in \Theta$. Sabemos que $\Theta_{\hat{x}+1} = \Theta_{\hat{x}} \uplus \{x\}$ y que $([\Theta_{\hat{x}}], v_{\hat{x}}) \Rightarrow^{\varsigma_{\hat{x}}}_{t_{\hat{x}}} ([\Theta_{\hat{x}+1}], v_{\hat{x}+1})$. Además, la transición discreta sólo puede corresponder a una arista *Mark* o *Instant*, con lo cual $v_{\hat{x}} + t_{\hat{x}} \vDash \psi^{\star x}_{\Theta_{\hat{x}}}$ y, por lo tanto, $v_{\hat{x}} + t_{\hat{x}} \vDash \psi_{z_x}(\delta(\mathbf{0}, x))$.

Observemos que los z_x sólo se resetean en las aristas *Mark* o *Instant* que *reconocen* o *marcan* el punto x, y ésto sólo pasa una vez a lo largo de toda la evolución. Además, z_0 no se resetea nunca. Quiere decir que podemos observar el valor del reloj z_0 a lo largo de toda la evolución:

$$\begin{array}{lcl} v_0(z_0) & = & \overline{\mathbf{0}}(z_0) = 0 \\ v_1(z_0) & = & t_0 \\ v_2(z_0) & = & t_0 + t_1 \\ \vdots & & \vdots \\ v_{\hat{x}}(z_0) & = & t_0 + \cdots + t_{\hat{x}-1} \end{array}$$

Sabemos que $v_{\hat{x}} + t_{\hat{x}} \vDash \psi_{z_0}(\delta(\mathbf{0}, x))$. Por la Proposición 4.6, $(v_{\hat{x}} + t_{\hat{x}})(z_0) \vDash \delta(\mathbf{0}, x)$.

Pero $(v_{\hat{x}} + t_{\hat{x}})(z_0) = v_{\hat{x}}(z_0) + t_{\hat{x}} = t_0 + \cdots + t_{\hat{x}} = \Delta(\tau_{\hat{x}]})$. Y quiere decir que $\Delta(\tau_{\hat{x}]}) \vDash \delta(\mathbf{0}, x)$.

Con esto queda demostrado que $\hat{\cdot}$ es un matching parcial entre σ y $\mathcal{P}$ restringido a Θ y, como consecuencia directa, $\sigma \vDash_\Theta \mathcal{P}$.

□

Propiedad 4.13. *Completitud del autómata reconocedor para patrones de eventos.*
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P} = \langle S, X, \Sigma, A, \mathcal{I}, s_0 \rangle$, para toda configuración $\Theta \in \Theta_\mathcal{P}$ y toda ejecución finita σ sobre Σ,

$$\text{si } \sigma \vDash_\Theta \mathcal{P} \text{ entonces } q_{init} \Rightarrow^\sigma [\Theta]$$

Demostración. Sea Θ una configuración de $\mathcal{P}$ y $\sigma = \langle \varsigma, \tau \rangle$ una ejecución finita sobre Σ. Supongamos, además que $\sigma \vDash_\Theta \mathcal{P}$ y, por lo tanto, que existe al menos un matching parcial $\hat{\cdot}$ entre σ y $\mathcal{P}$ restringido a Θ. Como en los casos anteriores, para demostrar que se puede evolucionar desde q_{init} hacia $[\Theta]$ por σ, vamos a dar una forma de construir una evolución entre dichas locaciones a partir de $\hat{\cdot}$.

Consideremos una vez más la secuencia de conjuntos de eventos presentada en la página 97:

$$\Theta_0, \Theta_1, \ldots, \Theta_n$$

donde $n = |\sigma|$ y para todo i, $0 \leq i \leq n$:

$$\Theta_i = \{\mathsf{x} \in \Theta \mid \hat{\mathsf{x}} < i\} \tag{A.18}$$

Se puede ver que también en este caso vale que:

- $\Theta_0 = \varnothing$
- $\Theta_n = \Theta$
- Para todo i, $0 \leq i \leq n$, Θ_i es una configuración
- Para todo i, $0 \leq i \leq n$, $[\Theta_i]$ es una locación de $\mathcal{A}_\mathcal{P}$
- Cada Θ_{i+1}, con $0 \leq i < n$ es igual a Θ_i o es una extensión de Θ_i
- Para todo i, $0 \leq i < n$, existe una arista de la forma $\langle [\Theta_i], \varsigma_i, \psi, \rho, [\Theta_{i+1}] \rangle$ o $\Theta_i = \Theta_{i+1}$ y $\varsigma_i = \lambda$.

  El razonamiento es el mismo que usamos en la página 97. La única diferencia es que en esa oportunidad probamos que ς_i no violaba ninguna restricción de eventos mostrando que no pertenecía a $\Gamma(\Theta_i)$ o a $\Gamma_{\triangleright \mathsf{x}}(\Theta_i)$, según el caso. En el caso de los patrones de eventos faltaría probar que esto también vale cuando usamos $\Gamma^\star$ y $\Gamma^\star_\triangleright$.

  En la página 97, vimos que si $\varsigma_i \in \Sigma$ y $\Theta_{i+1} = \Theta_i$, necesariamente $\varsigma_i \notin \Gamma(\Theta_i)$. Por **MPS** y **MPI**, sabemos que además $\varsigma_i \notin \Gamma^\star(\Theta_i)$. Si este no fuera el caso, existiría un punto $\mathsf{x} \in \Theta_i$ (y $\hat{\mathsf{x}} < i$) tal que $\varsigma_i \in \gamma(\mathsf{x}, \infty)$, con lo cual $\hat{\cdot}$ violaría **MPS**, o existiría un punto $\mathsf{y} \notin \Theta_i = \Theta_{i+1}$ (e $i < \hat{\mathsf{y}}$) tal que $\varsigma_i \in \gamma(\mathbf{0}, \mathsf{y})$, con lo cual $\hat{\cdot}$ violaría **MPI**. Por lo tanto, existe una arista *Skip* de la forma $\langle [\Theta_i], \varsigma_i, \top, \varnothing, [\Theta_i] \rangle$.

  Por otro lado, si $\varsigma_i \in \Sigma$ y $\Theta_{i+1} = \Theta_i \uplus \{\mathsf{p}\}$, para algún punto p, entonces como en el caso temporizado $\varsigma_i \notin \Gamma_{\triangleright \mathsf{p}}(\Theta_i)$. Nuevamente, **MPI** y **MPS** no aseguran que $\varsigma_i \notin \Gamma^\star_{\triangleright \mathsf{p}}(\Theta_i)$. Si este no fuera el caso, existiría un punto $\mathsf{x} \in \Theta_i$ tal que $\varsigma_i \in \gamma(\mathsf{x}, \infty)$, con lo cual $\hat{\cdot}$ violaría **MPS**, o existiría un punto $\mathsf{y} \notin \Theta_i$ tal que $\mathsf{y} \neq \mathsf{p}$ y $\varsigma_i \in \gamma(\mathbf{0}, \mathsf{y})$, con lo cual $\hat{\cdot}$ violaría **MPI**. Finalmente, por **MP1** sabemos que $\mathsf{p} \in E$ y $\varsigma_i = \varsigma_{\hat{\mathsf{p}}} \in \ell(\mathsf{p})$. Quiere decir que existe una arista en *Mark* de la forma $\langle [\Theta_i], \varsigma_i, \psi^{\star \mathsf{p}}_{\Theta_i}, \{z_\mathsf{p}\}, [\Theta_i \uplus \{\mathsf{p}\}] \rangle$.

  Si $\varsigma_i = \lambda$ el razonamiento es similar al del caso temporizado.

Como en el caso de los patrones temporizados, consideremos la sucesión de números reales no negativos $t_0, \ldots, t_{n-1}$ tal que τ se pueda escribir como $\tau = t_0 \lhd t_1 \lhd \cdots \lhd t_{n-1}$ y la secuencia de valuaciones $v_0, \ldots, v_n$ dada por:

$$\begin{array}{lcll} v_0 & = & \overline{0} & \\ v_{i+1} & = & \begin{cases} v_i + t_i & \text{si } \Theta_{i+1} = \Theta_i \\ Reset_{z_x}(v_i + t_i) & \text{si } \Theta_{i+1} = \Theta_i \uplus \{\mathsf{x}\} \end{cases} & 0 \leq i < n \end{array} \tag{A.19}$$

Veamos que también vale que para todo i, $0 \leq i < n$, $([\Theta_i], v_i) \Rightarrow_{t_i}^{\varsigma_i} ([\Theta_{i+1}], v_{i+1})$.

Dado un i cualquiera, $0 \leq i < n$, sabemos que existe la transición temporal $([\Theta_i], v_i) \rightarrow^{t_i} ([\Theta_i], v_i + t_i)$ dado que el invariante en todas las locaciones del autómata es $\top$. Vimos también en la página 102 que si $\Theta_{i+1} = \Theta_i$ entonces siempre existe la transición discreta $([\Theta_i], v_i + t_i) \rightarrow^{\varsigma_i} ([\Theta_{i+1}], v_{i+1})$ y que si $\Theta_{i+1} = \Theta_i \uplus \{\mathsf{p}\}$, para algún punto p, entonces existe una arista *Mark* o *Instant* de la forma $\langle [\Theta_i], \varsigma_i, \psi^{\star \mathsf{p}}_{\Theta_i}, \{z_\mathsf{p}\}, [\Theta_i \uplus \{\mathsf{p}\}] \rangle$ y para demostrar que existe la transición discretea $([\Theta_i], v_i + t_i) \rightarrow^{\varsigma_i} ([\Theta_{i+1}], v_{i+1})$ alcanza con probar que $v_i + t_i \vDash \psi^{\star \mathsf{p}}_{\Theta_i}$.

Supongamos que se da este último caso, es decir $\Theta_{i+1} = \Theta_i \uplus \{\mathsf{p}\}$. Vimos en el caso de los patrones temporizados que $v_i + t_i \vDash \psi^{\mathsf{p}}_{\Theta_i}$. Dado que $\psi^{\star \mathsf{p}}_{\Theta_i} = \psi^{\mathsf{p}}_{\Theta_i} \wedge \psi_{z_\mathbf{0}}(\delta(\mathbf{0}, \mathsf{p}))$, faltaría probar que $(v_i + t_i)(z_\mathbf{0}) \vDash \delta(\mathbf{0}, \mathsf{p})$.

Supongamos que $(v_i + t_i)(z_\mathbf{0}) \nvDash \delta(\mathbf{0}, \mathsf{p})$. Se puede ver en (A.19) que $z_\mathbf{0}$ no se resetea nunca (excepto al principio de la evolución). Por lo tanto:

$$\begin{array}{lcl} v_0(z_\mathbf{0}) & = & \overline{\mathbf{0}}(z_\mathbf{0}) = 0 \\ v_1(z_\mathbf{0}) & = & t_0 \\ v_2(z_\mathbf{0}) & = & t_0 + t_1 \\ \vdots & & \vdots \\ v_{\hat{\mathsf{p}}}(z_\mathbf{0}) & = & t_0 + \cdots + t_{\hat{\mathsf{p}}-1} \end{array}$$

Con lo cual, $(v_i + t_i)(z_\mathbf{0}) = v_{\hat{\mathsf{p}}}(z_\mathbf{0}) + t_{\hat{\mathsf{p}}} = t_0 + \cdots + t_{\hat{\mathsf{p}}} = \tau_{\hat{\mathsf{p}}} = \Delta(\tau_{\hat{\mathsf{p}}])}$. Quiere decir que $\Delta(\tau_{\hat{\mathsf{p}}]}) \nvDash \delta(\mathbf{0}, \mathsf{p})$ y esto violaría la hipótesis de que $\hat{}$ verificaba **MPTI**.

Quiere decir que en todos los caso existe la evolución en un paso $([\Theta_i], v_i) \Rightarrow_{t_i}^{\varsigma_i} ([\Theta_{i+1}], v_{i+1})$.

Una vez más, podemos concluir que:

$$([\Theta_0], v_0) \Rightarrow_{t_0}^{\varsigma_0} ([\Theta_1], v_1) \Rightarrow_{t_1}^{\varsigma_1} \ldots \Rightarrow_{t_{n-1}}^{\varsigma_{n-1}} ([\Theta_n], v_n)$$

lo cual implica que $q_{init} = ([\Theta_0], v_0) \Rightarrow^{\sigma} [\Theta_n] = [\Theta]$.

□

Teorema 4.14. *Tableau para patrones de eventos.*
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$, el autómata de Büchi temporizado $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$ reconoce el lenguaje $\mathcal{L}(\mathcal{P})$.

Demostración. Como en los dos casos anteriores, la demostración del teorema está dividida en dos lemas. El lema A.5 prueba que $\mathcal{L}(\mathcal{T}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P})$ y, por el otro lado, el lema A.6 prueba que $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{T}_\mathcal{P})$.

□

Lema A.5. *(Corrección del tableau)*
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P}$ y sea el autómata de Büchi temporizado $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$,

$$\mathcal{L}(\mathcal{T}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P})$$

Demostración. Queremos ver que $\mathcal{L}(\mathcal{T}_\mathcal{P}) \subseteq \mathcal{L}(\mathcal{P})$, es decir, que todas las ejecuciones aceptadas por $\mathcal{T}_\mathcal{P}$ corresponden sólo a ejecuciones aceptadas por $\mathcal{P}$.

Sea $\sigma = \langle \varsigma, \tau \rangle$ una ejecución en $\mathcal{L}(\mathcal{T}_\mathcal{P})$. La condición de aceptación de Büchi implica que existe una forma de evolucionar desde q_{init}, siguiendo σ, pasando infinitas veces por s_{accept}. Quiere decir que, en particular, debe existir una forma de evolucionar desde q_{init} hasta s_{accept} consumiendo un prefijo finito de σ:

$$q_{init} \Rightarrow^{\sigma_{i]}} s_{accept}$$

para algún $i \in \mathbf{N}$ y, además, se puede consumir el resto de la ejecución atravesando únicamente aristas *Skip* sobre s_{accept}. Por la propiedad de Corrección del autómata reconocedor para patrones de eventos 4.12, $\sigma_{i]} \vDash_P \mathcal{P}$, y por lo tanto, $\sigma_{i]} \vDash \mathcal{P}$. Sin embargo, a diferencia de los casos anteriores, saber que un prefijo finito de σ satisface el patrón $\mathcal{P}$ no nos alcanza para asegurar que la ejecución completa lo haga.
Para poder concluir que σ satisface el patrón, debemos mostrar que $\sigma_{(i}$ no contiene ningún evento "prohibidos" hasta el final de la ejecución. Estos eventos están dados por el conjunto $\mathbb{L}_\mathcal{P} = \bigcup_{x \in P} \gamma(x, \infty)$.
Por otro lado, sabemos que $\mathbb{L}_\mathcal{P} = \Gamma^\star(P)$ y que para todos los eventos más allá de la posición i es posible atravesar una arista *Skip*, con lo cual necesariamente ninguno de esos eventos pertenece a $\Gamma^\star(P)$. Quiere decir podemos afirmar que $\varsigma_{(i} \cap \mathbb{L}_\mathcal{P} = \varnothing$. Ahora si, la propiedad de Clausura débil por extensiones (Propiedad 3.5) nos permite concluir que $\sigma \vDash \mathcal{P}$.
Como r es divergente, σ debe serlo también, y con ésto hemos demostrado que $\sigma \in \mathcal{L}(\mathcal{P})$. □

Lema A.6. *(Completitud del tableau)*
Dado un patrón de eventos $\mathcal{P} = \langle \Sigma, P, \ell, \rightarrow, \gamma, \delta, \mathbf{0}, \infty \rangle$ y su autómata reconocedor $\mathcal{A}_\mathcal{P}$ y sea el autómata de Büchi temporizado $\mathcal{T}_\mathcal{P} = \langle \mathcal{A}_\mathcal{P}, \{s_{accept}\} \rangle$,

$$\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{T}_\mathcal{P})$$

Demostración. Queremos ver que por cada ejecución divergente aceptada por $\mathcal{P}$, existe una forma de evolucionar en $\mathcal{A}_\mathcal{P}$ consumiendo dicha ejecución, de forma tal de pasar un número infinito de veces por s_{accept}.
Sea $\sigma = \langle \varsigma, \tau \rangle$ una ejecución tal que $\sigma \in \mathcal{L}(\mathcal{P})$. Por definición de $\mathcal{L}$, sabemos que $\sigma \vDash \mathcal{P}$ y que σ es divergente. Por la propiedad de Satisfacción finita de los patrones de eventos (Propiedad 3.6), sabemos que existe una posición i de σ tal que:

$$\sigma_{i]} \vDash \mathcal{P} \tag{A.20}$$

$$\varsigma_{(i} \cap \mathbb{L}_\mathcal{P} = \varnothing \tag{A.21}$$

Sabiendo que vale (A.20), la propiedad de Completitud del autómata reconocedor de patrones de eventos 4.13 nos permite afirmar que $\sigma_{i]} \vDash_{[P]} \mathcal{P}$ y, por lo tanto:

$$q_{init} \Rightarrow^{\sigma_{i]}} s_{accept}$$

es decir, existe al menos una evolución que comienza en q_{init} y alcanza s_{accept}, consumiendo las primeras i posiciones de σ. Llamemos r_1 a dicha evolución. Vamos a probar que es posible consumir el resto de la ejecución atravesando únicamente aristas *Skip* sobre s_{accept}.
Por definición, existe una arista *Skip* en s_{accept} para cada evento que no pertenezca al conjunto $\Gamma^\star(P)$. Pero también sabemos por (4.1) que $\Gamma^\star(E) = \mathbb{L}_\mathcal{P}$ y por (A.21) podemos afirmar que $\varsigma_j \notin \Gamma^\star(P)$, para todo $j > i$. Quiere decir que existe al menos una evolución divergente r_2 que partiendo desde el estado final de r_1, permanezca para siempre en s_{accept} (y por lo tanto verifique la condición de aceptación de $\mathcal{T}_\mathcal{P}$), y que además $\overline{r_1 r_2} = \sigma$. Y esto alcanza para demostrar que $\sigma \in \mathcal{L}(\mathcal{T}_\mathcal{P})$.

□

Apéndice B

Implementación en Java

La implementación del verificador de patrones fue realizada en Java utilizando el JSDK versión 1.4.1. Dado que se trata de una versión prototipo, el verificador brinda únicamente una interfaz por línea de comando.

El verificador recibe dos archivos como input. Uno de los archivos corresponde al autómata temporizado que modela el sistema y el otro al patrón de mal comportamiento que se quiere verificar. El verificador contestará "Si" si y sólo si el sistema satisface el patrón y "No" en otro caso. El archivo con la definición del autómata temporizado que representa al sistema debe tener el formato aceptado por la herramienta Kronos [Yov]. La definición del patrón de mal comportamiento a verificar se debe dar a través de un documento XML que cumpla con la estructura especificada por el DTD incluido en la sección B.2.

B.1 Caso de estudio

Veremos a continuación un ejemplo de uso del componente **traductor** del Verificador de Patrones. Usaremos como caso de estudio el protocolo CSMA/CD presentado en la sección 5.2.

El primer listado muestra el documento XML con la definición del Patrón 5.4, correspondiente al Requerimiento 5.4.

```
<? xml version="1.0" ?>
<! DOCTYPE pattern SYSTEM "pattern.dtd" >

<pattern id="csma-cd-q">
    <alphabet>
        <event>BEGIN1</event>
        <event>BEGIN2</event>
        <event>END1</event>
        <event>END2</event>
        <event>CD1</event>
        <event>CD2</event>
    </alphabet>
    <eventsets>
        <eventset name="END">
            <event>END1</event>
            <event>END2</event>
        </eventset>
    </eventsets>
```

```
    <points>
        <e-point id="p">
            <event>BEGIN1</event>
        </e-point>
        <e-point id="q">
            <event>BEGIN2</event>
        </e-point>
        <e-point id="r">
            <eventset-ref name="END" />
        </e-point>
    </points>
    <precedence>
        <precedes>
            <point-ref id="p" />
            <point-ref id="r" />
            <event>BEGIN1</event>
            <event>END1</event>
            <event>CD1</event>
        </precedes>
        <precedes>
            <point-ref id="q" />
            <point-ref id="r" />
            <event>BEGIN2</event>
            <event>END2</event>
            <event>CD2</event>
        </precedes>
    </precedence>
</pattern>
```

XML para el Patrón 5.4

El comando:

```
traductor pattern1.xml pattern1.tg
```

permite obtener en el archivo `pattern1.tg` el automáta reconocedor para el Patrón 5.4. El siguiente listado muestra este autómata reconocedor en el formato definido por Kronos [Yov].

```
/* Automata Temporizado generado automaticamente (do not remove this line) */

/* Generales */
#states 6
#trans 42
#clocks 0

/* Locaciones y transiciones */

/* Location: { } [INIT] */
state: 0
```

```
prop: INIT
invar: TRUE
trans:
      TRUE => CD2 ; RESET{ } ; goto 0
      TRUE => BEGIN1 ; RESET{ } ; goto 0
      TRUE => BEGIN2 ; RESET{ } ; goto 0
      TRUE => BEGIN2 ; RESET{ } ; goto 5
      TRUE => END1 ; RESET{ } ; goto 0
      TRUE => END2 ; RESET{ } ; goto 0
      TRUE => CD1 ; RESET{ } ; goto 0
      TRUE => BEGIN1 ; RESET{ } ; goto 2

/* Location: Trap [TRAP] */
state: 1
prop: TRAP
invar: TRUE
trans:
      TRUE => END2 ; RESET{ } ; goto 1
      TRUE => CD2 ; RESET{ } ; goto 1
      TRUE => CD1 ; RESET{ } ; goto 1
      TRUE => BEGIN1 ; RESET{ } ; goto 1
      TRUE => BEGIN2 ; RESET{ } ; goto 1
      TRUE => END1 ; RESET{ } ; goto 1

/* Location: { p } */
state: 2
invar: TRUE
trans:
      TRUE => END1 ; RESET{ } ; goto 1
      TRUE => BEGIN2 ; RESET{ } ; goto 3
      TRUE => BEGIN2 ; RESET{ } ; goto 2
      TRUE => END2 ; RESET{ } ; goto 2
      TRUE => CD2 ; RESET{ } ; goto 2
      TRUE => BEGIN1 ; RESET{ } ; goto 1
      TRUE => CD1 ; RESET{ } ; goto 1

/* Location: { p, q } */
state: 3
invar: TRUE
trans:
      TRUE => BEGIN2 ; RESET{ } ; goto 1
      TRUE => END1 ; RESET{ } ; goto 4
      TRUE => BEGIN1 ; RESET{ } ; goto 1
      TRUE => END2 ; RESET{ } ; goto 4
      TRUE => END2 ; RESET{ } ; goto 1
      TRUE => END1 ; RESET{ } ; goto 1
      TRUE => CD1 ; RESET{ } ; goto 1
      TRUE => CD2 ; RESET{ } ; goto 1

/* Location: { p, q, r } [ACCEPT] */
state: 4
prop: ACCEPT
invar: TRUE
trans:
```

```
        TRUE => END2 ; RESET{ } ; goto 4
        TRUE => CD1 ; RESET{ } ; goto 4
        TRUE => BEGIN2 ; RESET{ } ; goto 4
        TRUE => END1 ; RESET{ } ; goto 4
        TRUE => BEGIN1 ; RESET{ } ; goto 4
        TRUE => CD2 ; RESET{ } ; goto 4

/* Location: { q } */
state: 5
invar: TRUE
trans:
        TRUE => END2 ; RESET{ } ; goto 1
        TRUE => CD2 ; RESET{ } ; goto 1
        TRUE => END1 ; RESET{ } ; goto 5
        TRUE => CD1 ; RESET{ } ; goto 5
        TRUE => BEGIN1 ; RESET{ } ; goto 3
        TRUE => BEGIN2 ; RESET{ } ; goto 1
        TRUE => BEGIN1 ; RESET{ } ; goto 5
```

Autómata reconocedor para el Patrón 5.4 generado automáticamente

Por otro lado, el listado muestra el documento XML con la definición del Patrón 5.5, correspondiente al Requerimiento 5.5.

```
<? xml version="1.0" ?>
<! DOCTYPE pattern SYSTEM "pattern.dtd" >

<pattern id="csma/cd-2">
        <alphabet>
                <event>BEGIN1</event>
                <event>BEGIN2</event>
                <event>END1</event>
                <event>END2</event>
                <event>CD1</event>
                <event>CD2</event>
        </alphabet>
        <points>
                <e-point id="p">
                        <event>BEGIN1</event>
                </e-point>
                <e-point id="q">
                        <event>BEGIN2</event>
                </e-point>
                <instant id="r" />
        </points>
        <precedence>
                <precedes>
                        <point-ref id="p" />
```

```
                <point-ref id="r" />
                <event>BEGIN1</event>
                <event>END1</event>
                <event>CD1</event>
                <interval>
                    <lower-bound value="52" included="false" />
                </interval>
            </precedes>
            <precedes>
                <point-ref id="q" />
                <point-ref id="r" />
                <event>BEGIN2</event>
                <event>END2</event>
                <event>CD2</event>
            </precedes>
        </precedence>
</pattern>
```

XML para el Patrón 5.5

Nuevamente, el comando:

traductor pattern2.xml pattern2.tg

Nos permite obtener el autómata reconocedor para el Patrón 5.5, mostrado en el siguiente listado:

```
/* Automata Temporizado generado automaticamente (do not remove this line) */

/* Generales */
#states 6
#trans 41
#clocks 2
X_p X_r

/* Locaciones y transiciones */

/* Location: { } [INIT] */
state: 0
prop: INIT
invar: TRUE
trans:
    TRUE => END2 ; RESET{ } ; goto 0
    TRUE => BEGIN1 ; RESETX_p ; goto 2
    TRUE => CD2 ; RESET{ } ; goto 0
    TRUE => CD1 ; RESET{ } ; goto 0
    TRUE => BEGIN2 ; RESET{ } ; goto 5
    TRUE => BEGIN1 ; RESET{ } ; goto 0
    TRUE => BEGIN2 ; RESET{ } ; goto 0
    TRUE => END1 ; RESET{ } ; goto 0
```

```
/* Location: Trap [TRAP] */
state: 1
prop: TRAP
invar: TRUE
trans:
      TRUE => BEGIN2 ; RESET{ } ; goto 1
      TRUE => END2 ; RESET{ } ; goto 1
      TRUE => CD1 ; RESET{ } ; goto 1
      TRUE => END1 ; RESET{ } ; goto 1
      TRUE => CD2 ; RESET{ } ; goto 1
      TRUE => BEGIN1 ; RESET{ } ; goto 1

/* Location: { p } */
state: 2
invar: TRUE
trans:
      TRUE => CD2 ; RESET{ } ; goto 2
      TRUE => CD1 ; RESET{ } ; goto 1
      TRUE => BEGIN2 ; RESET{ } ; goto 3
      TRUE => END1 ; RESET{ } ; goto 1
      TRUE => BEGIN2 ; RESET{ } ; goto 2
      TRUE => END2 ; RESET{ } ; goto 2
      TRUE => BEGIN1 ; RESET{ } ; goto 1

/* Location: { p, q } */
state: 3
invar: TRUE
trans:
      X_p > 52=> ; RESET{X_r} ; goto 4
      TRUE => BEGIN2 ; RESET{ } ; goto 1
      TRUE => CD2 ; RESET{ } ; goto 1
      TRUE => CD1 ; RESET{ } ; goto 1
      TRUE => BEGIN1 ; RESET{ } ; goto 1
      TRUE => END1 ; RESET{ } ; goto 1
      TRUE => END2 ; RESET{ } ; goto 1

/* Location: { p, q, r } [ACCEPT] */
state: 4
prop: ACCEPT
invar: TRUE
trans:
      TRUE => BEGIN2 ; RESET{ } ; goto 4
      TRUE => END1 ; RESET{ } ; goto 4
      TRUE => BEGIN1 ; RESET{ } ; goto 4
      TRUE => CD1 ; RESET{ } ; goto 4
      TRUE => END2 ; RESET{ } ; goto 4
      TRUE => CD2 ; RESET{ } ; goto 4

/* Location: { q } */
state: 5
invar: TRUE
trans:
      TRUE => END1 ; RESET{ } ; goto 5
```

```
TRUE => CD1 ; RESET{ } ; goto 5
TRUE => BEGIN2 ; RESET{ } ; goto 1
TRUE => BEGIN1 ; RESET{ } ; goto 5
TRUE => END2 ; RESET{ } ; goto 1
TRUE => BEGIN1 ; RESET{X_p} ; goto 3
TRUE => CD2 ; RESET{ } ; goto 1
```

Autómata reconocedor para el Patrón 5.5 generado automáticamente

B.2 DTD del documento para definición de patrones de mal comportamiento

```
<? xml encoding="US-ASCI" ?>

<!-- Patron de eventos. Un patron de eventos está definido por un alfabeto (conjunto de eventos), una serie
de conjuntos de eventos con algún nombre descriptivo (usado para agrupar eventos con propiedades similares),
un conjunto de puntos, una relación de precedencia entre los puntos y un conjunto de restricciones entre
pares de puntos. La definición de un relación de precedencia es opcional. Si no se define ninguna, se asume
que ningún punto está causalmente relacionado con ningún otro. Las restricciones entre puntos causalmente
relacionados pueden ser definidas cuando se define la precedencia entre dichos puntos o utilizando la sección
de "restrictions". Las restricciones entre puntos no relacionados sólo puede definirse en esa última sección.
Si no se define ninguna restricción, se asume que no hay eventos prohibidos entre ningún par de puntos y
que el delay permitido es [0, infinito) en todos los casos. -->
<!ELEMENT pattern (alphabet,labelsets?,points,precedence?,restrictions?) >
<!ATTLIST pattern id ID #REQUIRED>

<!-- Alfabeto de un patrón de eventos. Define el conjunto de eventos del patrón. Todos los eventos asociados
a puntos del patrón así como los eventos mencionados en restricciones o en conjuntos de eventos deben estar
declarados en esta sección. De no ser así, se lo considerará un error. El alfabeto puede ser vacío.
usado en: pattern -->
<!ELEMENT alphabet (event*) >

<!-- Define un único evento. El nombre del evento no necesita ser único, pero no es recomendable repetir
eventos en el mismo alfabeto.
usado en: alphabet, eventset, e-point, forbidden -->
<!ELEMENT event (#PCDATA) >

<!-- Agrupaciones de eventos. Definen conjuntos de eventos a los cuales asocia un nombre representativo
para ser referenciado en otras secciones del documento. Todos los eventos deben haber sido declarados en
el alfabeto del patrón. No es obligatorio declarar eventsets, pero si la sección está presente, entonces debe
incluir al menos un conjunto de eventos.
usado en: pattern -->
<!ELEMENT eventsets (eventset+) >

<!-- Conjunto de eventos. Define un conjunto de eventos al cual se le asocia un nombre único. Este conjunto
podrá ser referenciado en otras secciones del documento usando "eventset-ref". El conjunto no puede ser
vacío.
```

usado en: eventsets -->
<!ELEMENT eventset (event+) >
<!ATTLIST eventset name ID #REQUIRED>

<!-- *Referencia a un conjunto de eventos. Permite hacer referencia a un conjunto de eventos desde distintas secciones del documento. El conjunto debe haber sido definido en la sección "eventsets".*
usado en: e-point, forbidden -->
<!ELEMENT eventset-ref EMPTY>
<!ATTLIST eventset-ref name IDREF #IMPLIED>

<!-- *Conjunto de puntos del patrón. Define el conjunto de puntos que integran el patrón. Los puntos deben ser diferenciados en e-points e i-points. Los e-points deben tener 1 o más eventos asociadas, mientras que los i-points no tienen eventos asociadas. Los puntos declarados en esta sección (tanto e-points como i-points) pueden ser referenciados en otras partes del documento usando point-ref. La referencia se hace usando el id del punto.*
usado en: pattern -->
<!ELEMENT points (e-point*,i-point*) >

<!-- *Uno de los e-points del patrón. Debe tener asociado uno o más eventos y/o uno o más conjuntos de eventos. Desde el punto de vista del patrón, se considera la unión de todos los eventos mencionadas en esta sección, sea directamente o a través de un conjunto de eventos. Se puede hacer referencia a un e-point en otras partes del documento usando "point-ref".*
usado en: points -->
<!ELEMENT e-point ((event—eventset-ref)+) >
<!ATTLIST e-point id ID #REQUIRED>

<!-- *Uno de los i-points del patrón. Los i-points no tienen eventos asociadas. Se puede hacer referencia a un i-point en otras partes del documento usando "point-ref".*
usado en: points -->
<!ELEMENT i-point EMPTY>
<!ATTLIST i-point id ID #REQUIRED>

<!-- *Hace referencia a un e-point o i-point definido en "points".*
usado en: precedes, forbidden, delay -->
<!ELEMENT point-ref EMPTY>
<!ATTLIST point-ref id IDREF #IMPLIED>

<!-- *Punto especial que representa el comienzo de la ejecución*
usado en: precedes, forbidden, delay -->
<!ELEMENT before-start EMPTY>

<!-- *Punto especial que representa el final de la ejecución*
usado en: precedes, forbidden, delay -->
<!ELEMENT after-end EMPTY>

<!-- *Relación de precedencia entre los puntos del patrón. Define uno o más pares de puntos relacionados. No es obligatorio definir una relación de precedencia para el patrón, pero si se incluye esta sección, no puede estar vacía.*
usado en: pattern -->
<!ELEMENT precedence (precedes+) >

<!-- *Precedencia entre dos puntos del patrón. Compuesta por dos puntos, cero o más restricciones de eventos y cero o más restricciones temporales. El orden de los puntos está dado por el orden en que se*

escriben en el documento. Un punto puede ser definido por una referencia a un e-point o a un i-point o por los puntos especiales "before-start" y "after-end". Esos puntos corresponden al comienzo y al final de la ejecución, respectivamente. Las restricciones entre puntos relacionados pueden definirse en esta sección y/o en "restrictions". De todas formas, para el patrón se considera para cada par de puntos la unión de las restricciones definidas en esta sección y en "restrictions".

usado en: precedence -->

```
<!ELEMENT precedes ((point-ref—before-start—after-end),(point-ref—before-start—after-end),
(event—eventset-ref)*,interval?)>
```

<!-- Restricciones entre los puntos del patrón. Define las restricciones de eventos y temporales entre los pares de puntos del patrón. Si no se define ninguna restricción de eventos para un par de puntos dado, se asume que todos los eventos están permitidos. Si no se define ninguna restricción temporal para un par de puntos determinado, se asume que el delay permitido es [0, infinito). Para el patrón se considera para cada par de puntos la unión de las restricciones definidas en esta sección y en "precedes".

usado en: pattern -->

```
<!ELEMENT restrictions (forbidden*,delay*) >
```

<!-- Restricción de eventos entre un par de puntos del patrón. Los puntos se definen como en "precedes". En este caso no importa el orden en que aparezcan los puntos. Para cada par de puntos se puede definir una única restricción. Si esto no fuera así, se lo considerará un error. Los eventos se pueden mencionar en forma directa o a través de conjuntos de etiquetas definidos en "eventsets".

usado en: restrictions -->

```
<!ELEMENT forbidden ((point-ref—before-start—after-end),(point-ref—before-start—after-end),
(event—eventset-ref)+) >
```

<!-- Restricción de temporal entre un par de puntos del patrón. Los puntos se definen como en "precedes". En este caso no importa el orden en que aparezcan los puntos. Para cada par de puntos se puede definir una única restricción. Si esto no fuera así, se lo considerará un error.

usado en: restrictions -->

```
<!ELEMENT delay ((point-ref|before-start|after-end),(point-ref|before-start|after-end),interval) >
```

<!-- Intervalos de números reales no negativos. Un intervalo consiste en un límite inferior y un límite superior. Si el límite inferior no se indica, se asume [0 Si el límite superior no se indica, se asume infinito) -->

```
<!ELEMENT interval (lower-bound?, upper-bound?) >
<!ELEMENT lower-bound EMPTY>
<!ATTLIST lower-bound value CDATA #REQUIRED included (true|false) #REQUIRED>
<!ELEMENT upper-bound EMPTY>
<!ATTLIST upper-bound value CDATA #REQUIRED included (true|false) #REQUIRED>
```

Bibliografía

[ACD93] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AEY00] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *22nd International Conference on Software Engineering*, pages 304–313, 2000.

[AY99] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proc. 10th Intl. Conf. on Concurrency Theory*, pages 114–129. Springer Verlag, 1999.

[BGO02] V. Braberman, D. Garbervestky, and A. Olivero. Ignoring components during the verification of timed systems. *Lecture Notes in Computer Science*, 2280:21–36, 2002.

[BLL+96] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suited for the automatic verification of real-time systems. *Proceedings of Hybrid Systems III*, LNCS 1066:232–243, 1996.

[BOB03] S. Blaustein, F. Oliveto, and V. Braberman. Scenario-based validation and verification for real-time software: On run conformance and coverage for MSC-Graphs. In *25nd International Conference on Software Engineering*, 2003.

[Bra00] V. Braberman. *Modeling and Checking Real-Time Systems Designs.* PhD thesis, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2000.

[BW95] Burns and Wellings. HRT-HOOD: A structured design method for hard real-time ADA systems. Elsevier, 1995.

[BW96] Burns and Wellings. Real-time systems and programming languages. Adisson Wesley, 1996.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, December 1999.

[CK96] E. M. Clarke and R. P. Kurshan. Computer-aided verification. In *IEEE Spectr.*, volume 33, pages 61–67, June 1996.

[CWA+96] E. Clarke, J. Wing, R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[DAC98] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. Technical Report UM-CS-1998-035, , 1998.

[DH99] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. In *3rd IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.

[DKM+94] L. Dillon, G. Kutty, L. Moser, P. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, 1994.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, volume 1066, pages 208–219, Rutgers University, New Brunswick, NJ, USA, 22–25 October 1995. Springer.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. *Proceedings IEEE Real-Time Systems Symposium (RTSS '96)*, pages 73–81, December 1996.

[EC81] E. Emerson and E. Clarke. Design and synthesis of synchronization skeletons using branching-time temporal logics. In *Workshop of Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissedes. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[HHWT95] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. A User Guide to HyTech. *Proceedings of International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1019:41–71, 1995.

[HNSY92] T. A. Henzinger, X.Ñicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Proceedings of 7th Symposium on Logic in Computer Science*, pages 394–406, 1992. Ver también Information and Computation, 111(2):193-244, 1994.

[Hol97] G. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[IEE85] IEEE.ANSI/IEEE 802.3,ISO/DIS 8802/3. IEEE Computer Society Press, 1985.

[IT96] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC), 1996.

[M.J96] M.Joseph. Real time system: Specification, verification and analisys. In *International Series in Computer Science*. Prentice Hall, 1996.

[SACO02] R. Smith, G. Avrunin, L. Clarke, and L. Osterweil. Propel: an approach supporting property elucidation. In *Proceedings of the 24th international conference on Software engineering*, pages 11–21. ACM Press, 2002.

[Tan96] A. Tanenbaum. *Computer Networks*. Prentice-Hall, third edition, 1996.

[TC96] S. Tripakis and C. Courcoubetis. Extending promela and spin for real time. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 329–348, 1996.

[TIM01] Timeedit Tool. Release 1.2. http://www.bell-labs.com/project/timeedit, Agust 2001.

[UKM02] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 109–118. ACM Press, 2002.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[XJS92] X.Nicollin, J.Sifakis, and S.Yovine. Compiling realtime specification into extended automata. *IEEE Trans. on Soft. Eng., Special Issue on RealTime Systems*, 18(9):794–804, September 1992.

[Yov] S. Yovine. Kronos: A verification tool for real-time systems.

[Yov96] S. Yovine. Model checking timed automata. In *European Educational Forum: School on Embedded Systems*, pages 114–152, 1996.