Especificación y Análisis de Sistemas de Tiempo Real en Teoría de Tipos

Caso de Estudio: "The Railroad Crossing Example"

Tesis de Maestría en Informática

PEDECIBA Informática

Reporte Técnico 00-01

Carlos Daniel Luna[†]

Instituto de Computación (*InCo*). *Facultad de Ingeniería.*
Universidad de la República. Montevideo, Uruguay.

cluna@fing.edu.uy

Orientadora de Tesis: Dra. Cristina Cornes (*InCo*)

Supervisor de Maestría: Dr. Juan Echagüe (*InCo)*

Febrero del año 2000

[†] Los estudios de Maestría del autor fueron solventados con una beca del PEDECIBA Informática, Uruguay.

Resumen

Para el análisis de sistemas reactivos y de tiempo real dos importantes enfoques formales se destacan: la verificación de modelos, o *model checking*, y el análisis deductivo basado en asistentes de pruebas. El primero se caracteriza por la automaticidad pero presenta dificultades al tratar con sistemas que involucran un gran número de estados o donde se tienen parámetros variables, no acotados. El segundo permite tratar con sistemas arbitrarios pero requiere la interacción del usuario.

Este trabajo explora una metodología de trabajo que permita compatibilizar el uso de un verificador de modelos como *Kronos* y el asistente de pruebas *Coq* en el análisis de sistemas de tiempo real. Para ello formalizamos *grafos* (*autómatas*) *temporizados* y la lógica *TCTL* (y *CTL*) en el cálculo de construcciones inductivas y co-inductivas de *Coq*, a fin de disponer de lenguajes de especificación y análisis comunes a ambas herramientas. Los grafos permiten describir los sistemas, mientras que la lógica se usa para especificar los requerimientos temporales. Una parte importante del trabajo está dedicada a estudiar cómo razonar deductivamente en *Coq* sobre esta clase de sistemas –la utilidad de tipos inductivos y la necesidad de tipos co-inductivos– asumiendo inicialmente un modelo de tiempo discreto.

Un especial énfasis es puesto en el análisis de un caso de estudio, considerado como *benchmark* en diferentes trabajos: *el control de un paso a nivel de tren* ("*the railroad crossing example*"). Este problema es utilizado para evaluar y validar algunas de las formalizaciones propuestas.

Palabras Claves. Especificación y Análisis de Sistemas de Tiempo Real. Autómatas (Grafos) Temporizados. Lógicas TCTL y CTL. Verificación de Modelos ("Model Checking"). Teoría de Tipos y Coq. Verificación-Demostración de Corrección.

A mi familia, los amigos y Jóse, que siempre están;
Al InCo, por abrirme sus puertas y darme esta oportunidad;
Al PEDECIBA Informática, que financió mis estudios con una beca;
A Cristina Cornes, por su capacidad, paciencia, dedicación y comprensión;
A los docentes del Lab, y en particular a Luis, por su invaluable apoyo;
A quienes creen y luchan por una mejor educación para todos y
por una verdadera justicia social.

….. para todos ellos y gracias a ellos!

“ Pero, ¿Qué cosa es el tiempo? ¿Quién podría fácil y brevemente explicarlo? ¿Quién es el que puede formar ideas claras de lo que es el tiempo, de modo que se pueda explicar bien a otro? Y, por otra parte, ¿Qué cosa hay más común y más usada en nuestras conversaciones que el tiempo? ¿Entendemos bien lo que decimos cuando hablamos del tiempo, y lo entendemos también cuando otros nos hablan de él?

Pero, ¿Qué cosa es el tiempo? Si nadie me lo pregunta, yo lo sé, para entenderlo; pero si quiero explicarlo a quien me lo pregunta no lo sé para explicarlo. Pero me atrevo a decir que sé con certidumbre que sin ninguna cosa pasada no hubiera tiempo pasado, que si ninguna sobreviniera no habría tiempo futuro y si ninguna cosa existiera no habría tiempo presente. Pero aquellos dos tiempos que he nombrado, pasado y futuro, ¿De qué modo son o existen si el pasado ya no es y el futuro no existe todavía? Y en cuanto al tiempo presente, ¿Es cierto que si siempre fuera presente y no se mudara ni se fuera a ser pasado, ya no sería tiempo sino eternidad?

Luego, si el tiempo presente, para que sea tiempo es preciso que deje de ser presente y se convierta en pasado, cómo decimos que el presente existe y tiene ser, puesto que su ser estriba en que dejará de ser, pues no podremos decir con verdad que el presente es tiempo sino en cuanto camina a dejar de ser. ”

[San Agustín, Confesiones, Libro XI, Capítulo 14, año 400 D.C.]

Contenidos

Capítulo 1

Introducción

Cada vez son más frecuentes las aplicaciones donde el tiempo juega un rol importante. Por ejemplo en: protocolos de comunicación, controladores de robots, de comandos de aviones, de pasos a nivel de trenes, de dispositivos electrónicos (o electro-mecánicos) y de procesos industriales automatizados, aplicaciones multimedia y de internet, entre otras. En general éstas son aplicaciones críticas, en las cuales una falla o mal funcionamiento pueden acarrear consecuencias graves, tales como poner en juego vidas humanas y/o grandes inversiones económicas. El comportamiento de estos sistemas, llamados *sistemas de tiempo real*, no está determinado únicamente por la sucesión de acciones que se ejecutan, sino también por el momento en que las mismas ocurren y son procesadas. El tiempo de ejecución es "*el*" parámetro fundamental en el comportamiento de esta clase de sistemas y una gran parte, quizás la más importante, de los requerimientos de los mismos son temporales: "tal acción debe ejecutarse en un lapso de tiempo determinado", "el tiempo transcurrido entre dos eventos o señales debe estar acotado por un valor constante", "si un tren está próximo a cruzar un paso a nivel y han pasado más de *t* unidades de tiempo desde que se activó una señal de acercamiento, la barrera debe estar baja", etc.

Es indiscutible hoy la influencia que tiene en la industria y en casi todos los ámbitos el uso del software. La cantidad de aplicaciones reales y potenciales de la computación ha alcanzado cotas inimaginables apenas veinte años atrás. A pesar de su uso extensivo, uno de los costos más alto no se da en la producción del software, sino en la corrección de errores que son detectados posteriormente al desarrollo del sistema. En la actualidad, el método más usado para validar software es el "*testing*", que consiste en la simulación sobre casos de prueba representativos. No obstante, este método no garantiza la corrección del software analizado, por ser incompleto en la mayoría de los casos [MGJ91]. En las aplicaciones críticas, que tratan con vidas humanas y/o grandes inversiones económicas, la *certeza de corrección* es, en general, un criterio indispensable. De un software correcto se espera que resuelva un problema determinado por una *especificación* y que exista una justificación formal –matemática– de que el programa la satisface.

En los últimos años un gran esfuerzo de investigación se ha invertido en el desarrollo de métodos y herramientas para la especificación y el análisis de la corrección de

sistemas de tiempo real. Sin embargo no hay un formalismo, una metodología o una herramienta claramente preferibles a otras en todas circunstancias.

Para el análisis de la corrección de esta clase de sistemas dos importantes enfoques formales se destacan:

- ***Verificación* de corrección**. En este enfoque un sistema es considerado correcto cuando se prueba que *toda* ejecución posible satisface la especificación. Existen algunas técnicas bien conocidas que permiten recorrer de manera exhaustiva el espacio de ejecuciones posibles y herramientas que las implementan.

- ***Demostración* de corrección**. En este caso se construye o deriva una *prueba* matemática de que el sistema satisface su especificación. Aquí las herramientas asisten al programador en el proceso de construcción de la prueba.

1.1 Verificación de Modelos

El método de verificación llamado *verificación de modelos* ("*model checking*") fue desarrollado para verificar *sistemas reactivos* –aquellos que se comportan como una secuencia de estímulos-respuestas en relación al medio– y posteriormente ha sido extendido también a *sistemas de tiempo real*, en los cuales la corrección depende de las magnitudes de los retardos temporales.

Usando esta metodología los sistemas se modelan como *grafos* y la especificación se expresa, generalmente, mediante fórmulas en una *lógica temporal* (por ejemplo, [Pnu85, CES86, Eme95]). Un procedimiento eficiente se utiliza para determinar automáticamente si las especificaciones son satisfechas por los grafos. Esta técnica ha sido usada con éxito para detectar errores sutiles en distintos sistemas, en particular en protocolos de comunicaciones (por ejemplo, [WH95]).

Durante los últimos años el tamaño de los sistemas que pueden ser verificados de esta manera se ha incrementado notoriamente. Esto ha sido consecuencia del desarrollo de nuevas técnicas dentro de la misma estrategia, como el "*symbolic model checking*" y la verificación "*on the fly*". Entre las herramientas que implementan algunas de estas técnicas se destacan *Kronos* [Yov97], *HyTech* [HHW+97] y *Uppaal* [LPY97].

1.2 Demostración de Corrección

Esta aproximación permite construir una *demostración* –en el sentido matemático del término– de que un sistema satisface una especificación. En este trabajo estamos interesados en herramientas basadas en *teorías constructivas de tipos* [CH85, Coq86, CH88], las cuales han sido formuladas como fundamento de la Matemática Constructiva.

En la última década, varios equipos de investigación han dedicado un considerable esfuerzo al diseño e implementación de editores de prueba interactivos basados en teorías de tipos. Ejemplos de estos sistemas son *ALF* [Mag94], *Coq* [Bar+99] y *LEGO* [LP92].

Una de las principales características de los mismos es el carácter unificador de la teoría que implementan, en la cual pueden ser expresados programas, teoremas y pruebas de éstos. Otro punto destacable es que el usuario es guiado en forma interactiva por el sistema en el proceso de construcción de un programa o una prueba, siendo verificada inmediatamente la validez de cada paso del desarrollo. El principal objetivo de estos sistemas es convertirse en sofisticadas herramientas que asistan en la tarea del desarrollo incremental de programas correctos. Sin embargo, el marco conceptual necesario para desarrollar software verificado es de una muy alta complejidad y requiere cubrir muchos aspectos que en realidad escapan a la construcción de un asistente de pruebas.

Estos sistemas disponen de un lenguaje de especificación de orden superior, permiten hacer pruebas en lógica de alto orden y proveen definiciones de tipos inductivos y co-inductivos. La experimentación desarrollada en su uso se ha enfocado principalmente en demostrar la corrección de programas secuenciales, pero dado su poder expresivo consideramos que pueden ser también adecuados para razonar sobre sistemas reactivos y, en particular, de tiempo real. Algunas experiencias llevadas a cabo en esta dirección y particularmente en *Coq* son [Gim95, -96, -99], entre otras.

1.3 Complementariedad de los Enfoques

Los verificadores de modelos ("model-checkers") son usados actualmente en la industria con éxito para verificar sistemas reactivos, paralelos y de tiempo real, ya que son fáciles de usar y no requieren la asistencia del usuario en el proceso de prueba. Sin embargo, ellos en general presentan problemas al tratar con sistemas grandes (con muchos estados) o donde se tienen parámetros variables, no acotados. Los asistentes de prueba antes citados y otros como [ORS92, Gor93, Pau93b] proveen una solución alternativa para estos casos, aunque la complejidad del proceso de análisis se incrementa muchas veces en forma considerable. No obstante, el enfoque deductivo presenta como ventajas comparativas, además de las nombradas, entre otras, las siguientes:

- Al demostrar una propiedad de un sistema no sólo se tiene la certeza de que vale la propiedad, sino por qué esto es así. El desarrollo de una prueba induce a tener un conocimiento más profundo del sistema, muchas veces a descubrir nuevas propiedades o generalizarlas. Asimismo, en el proceso de prueba se puede llegar a descubrir las causas por las cuales una propiedad no vale e incluso, algunas veces, las modificaciones necesarias del sistema para que la cumpla.
- El proceso de construcción de las pruebas es incremental y composicional. Esto es, las propiedades demostradas pueden ser utilizadas en la prueba de otras, sin ser necesario hacer dicho proceso cada vez desde cero. Asimismo, una demostración bien pensada permite, en general, reutilizar una estrategia de prueba en la demostración de otros teoremas.

- El enfoque deductivo permite mayor rehuso. En el método de verificación de modelos cuando una modificación es introducida en el sistema, todo el proceso de verificación debería ser ejecutado nuevamente desde el comienzo. Sin embargo, muchas veces las pruebas, o algunas de ellas, o partes de las mismas, pueden mantenerse luego de un cambio, si la estrategia de demostración está bien estructurada (por ejemplo, al modificar algunas de las constantes del problema o alguna condición lógica entre las mismas).
- Las demostraciones de propiedades de un sistema pueden permitir generalizar la especificación del mismo, preservando las propiedades de interés. Esta característica será explotada en el trabajo en el análisis de un caso de estudio.

Los dos enfoques referidos se consideraban al comienzo diametralmente opuestos para el análisis de sistemas reactivos y de tiempo real. Sin embargo, en los últimos años surgió un interés creciente en combinarlos, debido, en parte, a que los mismos pueden ser cooperativos y no sólo competitivos entre sí. La idea es compensar las desventajas de un enfoque con las ventajas del otro, aunque aún no está claro cómo lograr dicho objetivo. Algunos referentes en esta dirección son [MN95, RSS95, HS96, BMS+97, SS99, KKP99], entre otros muchos.

Este trabajo está enmarcado en una propuesta de proyecto de investigación que vincula al proyecto *Coq* del INRIA –Rocquencourt (Francia)– y al grupo de Métodos Formales del Instituto de Computación de la Universidad de la República (Uruguay). El proyecto, titulado "Integración de dos enfoques para la verificación formal de sistemas reactivos: Teoría de Tipos y Verificación de Modelos", tiene por objetivo estudiar la integración de los dos enfoques previamente referidos para la verificación de sistemas reactivos y de tiempo real (Octubre de 1998).

En este marco, nuestro objetivo es dar un pequeño paso en una combinación posible entre ambos enfoques, estableciendo una metodología de trabajo que permita compatibilizar el uso de un *model checker* como *Kronos* y el asistente de pruebas *Coq*. A fin de lograr esto nos proponemos formalizar *grafos* (*autómatas*) *temporizados* [ACD90, Oli94, HNS+92] y la *lógica TCTL* (*timed computation tree logic*) [ACD90, HNS+92] en el *cálculo de construcciones inductivas* y *co-inductivas* de *Coq* [Bar+99]. Los grafos temporizados permiten describir sistemas de tiempo real, mientras que la lógica TCTL es un lenguaje adecuado y ampliamente usado para especificar propiedades (requerimientos) temporales. *Kronos* permite verificar si un grafo temporizado satisface una fórmula TCTL, siempre que los parámetros del sistema sean valores constantes. El análisis deductivo nos permite trabajar con parámetros variables y de esta manera verificar sistemas más generales. En este contexto, a las ventajas citadas del enfoque deductivo se suma una muy importante: la posibilidad de realizar *síntesis de programas* a partir de una formalización en teoría de tipos. Esta es una línea que no será explotada en el trabajo pero que justifica aún más el interés de esta experiencia.

1.4 Acerca de *Coq*

El asistente de pruebas *Coq* es una implementación del *cálculo de construcciones inductivas*, una lógica intuicionista de alto orden con tipos dependientes y tipos inductivos como objetos primitivos [CH88, P-M93]. El usuario introduce definiciones y hace demostraciones en un estilo de *deducción natural*, las cuales son chequeadas mecánicamente por el sistema.

Dicho formalismo permite especificar y probar en lógica intuicionista de alto orden. Esta lógica asocia una interpretación computacional a las pruebas, la noción de veracidad de una proposición corresponde a la existencia de una prueba. Curry y Howard demostraron que los siguientes juicios son equivalentes:

- "*t* es una demostración de la proposición *A*".
- "el término *t* tiene tipo *A*".
- "*t* es un programa de la especificación *A*".

Esto es, probar una proposición *A* es equivalente a construir un término de tipo *A*. Esta equivalencia es conocida como isomorfismo de Curry-Howard [CF58, How80] y está basada en que las pruebas en deducción natural pueden ser representadas como términos de un cálculo lambda tipado [How80], o que es lo mismo, de un lenguaje de programación funcional (por ejemplo, *Caml* [WL99]). Consecuentemente, *Coq* puede ser considerado, rigurosamente hablando, un chequeador de tipos ("type-checker").

A los efectos de diferenciar objetos computacionales de información lógica, distinguimos dos clases de tipos importantes en *Coq*: *Prop* para los tipos que contienen términos lógicos (las proposiciones) y *Set* que agrupa a los tipos que contienen información computacional (los conjuntos). Por ejemplo, los números naturales se definen en *Set* y las relaciones $\leq$ y la conjunción $\wedge$ en *Prop*. *Set* y *Prop* pertenecen a *Type*, que es en realidad una familia infinita de tipos notada de esta forma. Un procedimiento de *extracción de programas* puede ser usado para remover las partes lógicas de los términos, manteniendo sólo las partes de información computacional [P-M89a, P-M89b].

Además de los habituales tipos inductivos (o sea, conjuntos definidos inductivamente, como por ejemplo los números naturales o las listas finitas), *Coq* permite también la definición de tipos *co-inductivos*. Estos son tipos recursivos que pueden contener objetos infinitos, no bien fundados. Un ejemplo de tipos co-inductivos es el de las secuencias infinitas, usualmente llamadas *streams*, de elementos de un tipo dado.

En el proceso de prueba de un teorema, *Coq* entra en un ciclo interactivo donde el usuario completa la demostración usando *tácticas*, las cuales implementan reglas de inferencia o de tipado (esquemas de prueba). El conjunto de tácticas puede ser incrementado por el usuario, a partir de un lenguaje diseñado para tal fin.

En este trabajo no estamos interesados en dar una descripción completa del cálculo de construcciones inductivas y co-inductivas, ni del sistema de *Coq*, en general. Por aspectos teóricos el lector puede referirse a [CH85, Coq86, CH88] por el cálculo puro de

construcciones, a [P-M93] por tipos inductivos y a [Coq93, Pau93a, Gim96] por tipos co-inductivos. Acerca de *Coq*, una buena introducción son los tutoriales [HKP96, Gim98] y detalles adicionales pueden encontrarse en el manual de referencia [Bar+99].

1.5 Organización del Trabajo

La organización del trabajo es como sigue. En el capítulo 2 introducimos los grafos temporizados, usados para describir sistemas de tiempo real, y la lógica TCTL, utilizada como lenguaje de especificación de propiedades temporales. Asimismo, analizamos una discretización del dominio temporal continuo inherente a los grafos y la lógica considerados, en base a la cual obtenemos una semántica alternativa, que asumiremos en el resto de este trabajo.

En el capítulo 3 formalizamos en *Coq* grafos temporizados y la lógica TCTL. Incluimos además operadores de la lógica CTL (*computation tree logic*) [CES86] y algunas definiciones alternativas para los operadores que permiten expresar *invarianza* y *alcanzabilidad*, con tipos inductivos (bajo la noción de estados alcanzables) y co-inductivos (bajo la noción de trazas –infinitas– de ejecución). Asimismo, formulamos y demostramos algunas propiedades generales de los operadores temporales.

En el capítulo 4 consideramos un caso de estudio: *el control de un paso a nivel de tren* ("*the railroad crossing example*"). Especificamos el sistema en *Coq* en base a las formalizaciones introducidas en el capítulo 3. Analizamos luego la demostración de *invariantes*, incluyendo la propiedad de seguridad esencial del sistema, y verificamos la divergencia del tiempo en las ejecuciones. Posteriormente generalizamos la especificación: parametrizamos las constantes del problema y definimos restricciones entre estos parámetros y los relojes del sistema a fin de preservar las propiedades estudiadas. La mayor parte del trabajo está dedicada al análisis de este caso de estudio.

En el capítulo 5 definimos dos representaciones genéricas de grafos temporizados para la semántica de tiempo discreto considerada en los capítulos previos, una de las cuales es extendida a tiempo continuo. Estas representaciones permiten definir sistemas como instancias particulares y simplifican el proceso de definición de sistemas compuestos.

Las conclusiones, algunos trabajos relacionados y los trabajos futuros se incluyen en el capítulo 6. En particular, describimos una metodología de trabajo para la especificación y el análisis de grafos temporizados en función de las formalizaciones introducidas en los capítulos 3 y 5, y el caso de estudio abordado en el capítulo 4. Finalmente, las formalizaciones y las pruebas completas del trabajo se adjuntan como apéndices.

Capítulo 2

Grafos Temporizados y TCTL

En este capítulo introducimos los grafos temporizados, que permiten describir sistemas de tiempo real, y la lógica TCTL, usada como lenguaje de especificación de propiedades temporales. Asimismo, analizamos una discretización del dominio temporal continuo inherente a los grafos y la lógica considerados, en base a la cual obtenemos una semántica alternativa, que asumiremos en el resto de este trabajo.

2.1 Grafos (Autómatas) Temporizados

Los grafos temporizados constituyen un modelo matemático-computacional en el cual pueden representarse de manera formal, simple, clara y modular muchos problemas del mundo real donde hay restricciones temporales que interfieren con transiciones discretas, las cuales representan acciones o eventos. Varias definiciones similares de *grafos* (*autómatas*) *temporizados* han sido propuestas, como es el caso de [ACD90, HNS+92, Yov93, Oli94, AD94].

Un grafo temporizado es un autómata extendido con un conjunto *finito* de variables reales, llamadas *relojes*, cuyos valores se incrementan uniformemente con el paso del tiempo. Dichos relojes son utilizados para medir, por ejemplo, el tiempo transcurrido entre dos eventos, el tiempo de espera o la demora de una comunicación. Las restricciones temporales inherentes a las acciones del sistema son expresadas ligando condiciones de *activación* a cada una de las transiciones del autómata. Una transición está *habilitada*, es decir puede ser atravesada, cuando su condición asociada es satisfecha por los valores de los relojes. Un reloj puede ser puesto a cero en cualquier transición. A todo instante, el valor de un reloj es igual al tiempo transcurrido desde la última vez en que fue puesto a cero [ACD90, DOY96].

Definición 2.1. Sea A un conjunto (global) de *etiquetas* que asumiremos fijo en este capítulo. Un *grafo* (*autómata*) *temporizado* es una quíntupla $G = \langle L, X, E, l^0, I \rangle$, donde:

- L es un conjunto finito de nodos llamados *locaciones*.
- X es un conjunto finito de *relojes*. Una *valuación* v de los relojes es una función que asigna un valor $v(x) \in \mathbb{R}^+$ a cada reloj $x \in X$, donde $\mathbb{R}^+$ son los números reales no

negativos. V denota el conjunto de las valuaciones. Escribiremos $v+t$ como la valuación v' tal que $v'(x)=v(x)+t$ para todos los relojes $x \in X$.

- E es un conjunto finito de aristas llamadas *transiciones*. Cada transición es una quíntupla $e = \langle l, \alpha, \psi_X, \rho_X, l' \rangle$ que consiste en una locación origen $l \in L$, una locación destino $l' \in L$, una etiqueta $\alpha \in A$, una condición ψ_X y un conjunto $\rho_X \subseteq X$ de relojes que son puestos a cero (reseteados) simultáneamente con la transición.
  Una condición es una combinación booleana de átomos de la forma $x \prec c$, donde $x \in X$, $\prec$ es una relación binaria en el conjunto $\{<, \leq, =, \geq, >\}$ y c es una constante entera positiva. Denotamos Ψ_X al conjunto de los predicados ψ_X definibles sobre X. La transición $e = \langle l, \alpha, \psi_X, \rho_X, l' \rangle$ está habilitada en un estado $\langle l, v \rangle$ si v satisface la condición ψ_X. Escribiremos $v[\rho_X := 0]$ como la valuación v' tal que $v'(x)=0$ si $x \in \rho_X$ y $v'(x)=v(x)$ en caso contrario.
- l^0 es la *locación inicial*. El estado inicial del sistema es $\langle l^0, v^0 \rangle$, con $v^0(x) = 0$ para todo $x \in X$, es decir, la locación inicial con todos los relojes puestos en cero.
- I es el conjunto de *invariantes de las locaciones* del grafo. Para cada $l \in L$, $I_l \in \Psi_X$ es el *invariante* de la locación. Cuando el sistema se encuentra en un estado $\langle l, v \rangle$, éste puede permanecer en la locación l dejando pasar el tiempo, mientras la valuación corriente de los relojes satisfaga el invariante I_l.

Observaciones

- Los grafos temporizados pueden ser alternativamente definidos como una séxtupla, agregando un conjunto de etiquetas en vez de considerar un conjunto global A [AD94].
- Cada transición está rotulada con una sola etiqueta. Sin embargo, algunos trabajos consideran transiciones rotuladas con conjuntos de etiquetas (por ejemplo, [AD94]).
- En algunas definiciones de grafos temporizados no se distingue un estado inicial.

En lo que sigue del trabajo usaremos indistintamente los términos grafos temporizados y autómatas temporizados.

2.1.1 Semántica Formal

La semántica de un grafo temporizado puede definirse en función de *un sistema de transiciones etiquetado* [Oli94], donde los estados son pares $\langle l, v \rangle \in L \times V$ y las transiciones son de dos tipos:

- *Temporales*. Los nodos del grafo representan una actividad continua, dada por el paso del tiempo. El paso de un tiempo t es representado por una transición etiquetada con t.

- *Discretas* (*instantáneas*). Las aristas del grafo representan acciones discretas. La ejecución de una acción α es una transición que lleva la etiqueta α.

Definición 2.2. El modelo de un grafo temporizado $G = \langle L, X, E, l^0, I\rangle$ es el sistema de transiciones etiquetado $T = \langle Q, \rightarrow\rangle$, donde:

- $Q = \{\langle l, v\rangle \mid l \in L,\ v \in V,\ I_l[v]\}$ es el conjunto de estados. $I_l[v]$ significa que I_l es verdadero para la valuación v de los relojes. Los estados de Q serán llamados *estados válidos* del sistema (safisfacen los invariantes de las locaciones). Para $q = \langle l, v\rangle \in Q$, $Loc(q)$ denota la locación l.
- $\rightarrow \subseteq Q \times (A \cup \mathbb{R}^+) \times Q$ es la relación de transición, definida por las siguientes reglas:

$$\frac{I_l[v+t'] \qquad 0 \le t' \le t}{\langle l, v\rangle \xrightarrow{\ t\ } \langle l, v+t\rangle}$$

Es una *transición temporal* sobre la locación l. El sistema puede permanecer en dicha locación mientras los valores de los relojes satisfagan I_l. El estado $\langle l, v+t\rangle$ es un *sucesor temporal* del estado $\langle l, v\rangle$.

$$\frac{\langle l, \alpha, \psi_X, \rho_X, l'\rangle \in E \qquad \psi_X[v]}{\langle l, v\rangle \xrightarrow{\ \alpha\ } \langle l', v[\rho_X := 0]\rangle}$$

Es una *transición discreta* de l a l' por la acción α, cuando v satisface la restricción ψ_X. El estado $\langle l', v[\rho_X := 0]\rangle$ es el *sucesor discreto* del estado $\langle l, v\rangle$ por α. La condición $I_l(v) \wedge I_{l'}(v[\rho_X := 0])$ está implícita en la regla anterior, ya que la relación de transición se define entre estados *válidos*, es decir $\langle l, v\rangle \in Q$ y $\langle l', v[\rho_X := 0]\rangle \in Q$.

Notación. Escribimos $q \xrightarrow{\ lab\ } q'$ en lugar de $\langle q, lab, q'\rangle \in \rightarrow$, para $lab \in A \cup \mathbb{R}^+$.

Observaciones

- El tiempo pasa sólo en las locaciones (*transiciones temporales*).
- Para todo estado q hay una transición al mismo estado que toma tiempo cero.
- Atravesar una arista no insume tiempo (*transiciones instantáneas*).
- Los valores de los relojes crecen *uniformemente* con el tiempo.
- Respecto a las *transiciones temporales*, el sistema es determinístico, pero respecto a las transiciones instantáneas, el sistema no lo es en el caso general [Oli94].

2.1.2 Trazas de Ejecución

Una *ejecución* (*traza de ejecución*) de un grafo temporizado G es una secuencia de estados $q_0 \xrightarrow{t_0} q_0' \xrightarrow{\alpha_0} q_1 \xrightarrow{t_1} q_1' \xrightarrow{\alpha_1} q_2 \ldots\ldots$, donde $q_i \xrightarrow{t_i} q_i'$ representa una transición temporal, $q_i' \xrightarrow{\alpha_i} q_{i+1}$ es una transición discreta o una transición temporal con tiempo cero, y q_0 es el estado inicial de la ejecución. Denotamos R_G al conjunto de todas las ejecuciones de G y $R_G(q_0)$ a las que comienzan en el estado q_0.

Observemos que una transición $q_i \xrightarrow{t_i} q_i'$ representa conceptualmente a los *infinitos* estados de la forma q_i+t', con $t' \leq t_i$, y que la definición de ejecución dada permite que dos o más eventos discretos sucedan consecutivamente en el tiempo, asumiendo transiciones temporales intermedias con tiempo nulo. Notemos además que dos o más transiciones temporales consecutivas pueden ser representadas por una única cuyo valor temporal está dado por la suma de los valores correspondientes a los pasos individuales o intercalando una transición temporal con tiempo cero.

Sea r una ejecución $q_0 \xrightarrow{t_0} q_0' \xrightarrow{\alpha_0} q_1 \xrightarrow{t_1} q_1' \xrightarrow{\alpha_1} q_2 \ldots\ldots$ de $R_G(q_0)$, una *posición* π de r es un par $\langle i, t \rangle \in \mathbb{N} \times \mathbb{R}^+$, con $0 \leq t \leq t_i$. Escribimos $\sigma_r(\pi)$ para denotar al estado $\langle l, v+t \rangle$ con $q_i = \langle l, v \rangle$, y $\delta_r(\pi)$ para el tiempo $\Sigma_{j<i}\, t_j + t$ transcurrido desde el comienzo de la ejecución r. Llamaremos $\Pi(r)$ al conjunto de todas las posiciones de r.

Una ejecución $r \in R_G$ es *divergente* si para cada $t \in \mathbb{R}^+$ existe una posición π de r tal que $\delta_r(\pi) > t$. Las ejecuciones divergentes son aquellas en las cuales el tiempo avanza más allá de cualquier límite (diverge); en la literatura son denominadas "*non-Zeno*" [Alu91]. Δ_G denota al conjunto de las secuencias de ejecución *divergentes* de G y $\Delta_G(q_0)$ a las que poseen comienzo q_0. Nosotros estamos interesados, al igual que los trabajos previos, en sistemas de tiempo real –grafos temporizados– en los cuales todo prefijo finito de una ejecución en R_G es prefijo de una secuencia en Δ_G. Llamaremos a los mismos *sistemas bien temporizados*. El comportamiento de los sistemas bien temporizados –cada traza de ejecución– puede ser generado, tal como se describió al comienzo, transición a transición, comenzando en un estado inicial y repetidamente eligiendo entre incrementar el tiempo (mientras no se violen las condiciones invariantes) y hacer una transición discreta (si ésta está habilitada). En estos sistemas todas las trazas de ejecución deben ser necesariamente secuencias *infinitas*. Sin embargo, esta no es una condición suficiente para asegurar *non-Zeno* (por ejemplo, un sistema cuyas trazas de ejecución son todas secuencias infinitas que repiten siempre el mismo estado, con transiciones temporales nulas, o que alternan consecutivamente en forma cíclica uno o más estados a través de transiciones discretas, no cumple *non-Zeno*).

2.2 Composición Paralela de Grafos Temporizados

Para facilitar la descripción modular de los sistemas se utiliza la *composición paralela* de grafos temporizados. La composición paralela de dos grafos temporizados es el producto

de ambos donde las transiciones que comparten etiquetas deben *sincronizar*, es decir, las acciones correspondientes tienen que ejecutarse simultáneamente. Por cada par de dichas transiciones el sistema global tendrá una única transición tal que la etiqueta es la misma, la condición es la conjunción de las condiciones y el conjunto de relojes a resetear es la unión de los conjuntos correspondientes.

2.2.1 Definición de Composición

La composición paralela de dos grafos temporizados G_1 y G_2, sincronizando las etiquetas de $\Lambda \subseteq A$ –comunes a G_1 y G_2–, es notada $G_1|[\Lambda]|G_2$ y se define como sigue. Sean $G_1 = \langle L_1, X_1, E_1, l_1^0, I_1\rangle$ y $G_2 = \langle L_2, X_2, E_2, l_2^0, I_2\rangle$ tales que $X_1 \cap X_2 = \varnothing$, $G_1|[\Lambda]|G_2$ es el grafo $G = \langle L, X, E, l^0, I\rangle$, donde:

- $X = X_1 \cup X_2$
- $L \subseteq L_1 \times L_2$
- $\langle\langle l_1, l_2\rangle, \alpha, \psi_X, \rho_X, \langle l_1', l_2'\rangle\rangle \in E$ si y sólo si:
  1. $\langle l_1, \alpha, \psi_X, \rho_X, l_1'\rangle \in E_1 \ \wedge\ \alpha \notin \Lambda \ \wedge\ l_2 = l_2'$
  2. $\langle l_2, \alpha, \psi_X, \rho_X, l_2'\rangle \in E_2 \ \wedge\ \alpha \notin \Lambda \ \wedge\ l_1 = l_1'$
  3. $\langle l_i, \alpha, \psi_{iX}, \rho_{iX}, l_i'\rangle \in E_i \ \wedge\ \alpha \in \Lambda \ \wedge\ \psi_X = (\psi_{1X} \wedge \psi_{2X}) \ \wedge\ \rho_X = (\rho_{1X} \cup \rho_{2X})$
- $I(\langle l_1, l_2\rangle) = I_1(l_1) \wedge I_2(l_2)$
- $l^0 = \langle l_1^0, l_2^0\rangle$

2.2.2 Semántica

El comportamiento del grafo $G = G_1|[\Lambda]|G_2$ está ligado a los comportamientos de G_1 y G_2. Sean $T_1 = \langle Q_1, \rightarrow_1\rangle$ y $T_2 = \langle Q_2, \rightarrow_2\rangle$ modelos de G_1 y G_2, respectivamente. El modelo de G es $\langle Q_1 \times Q_2, \rightarrow\rangle$, donde $\rightarrow$ es la relación más pequeña que satisface las reglas que se enuncian a continuación.

- *Transiciones temporales.* La evolución temporal del sistema compuesto en un estado está condicionada a la de cada componente. Si cada componente permite el paso de un tiempo *t*, el sistema compuesto puede entonces avanzar *t* unidades. Esto determina que los relojes del sistema compuesto crecen uniformemente con el tiempo. Formalmente,

$$\frac{q_1 \xrightarrow{t}_1 q_1' \qquad q_2 \xrightarrow{t}_2 q_2'}{\langle q_1, q_2\rangle \xrightarrow{t} \langle q_1', q_2'\rangle}$$

- *Transiciones discretas (instantáneas*). En este caso existen dos posibilidades:

  1. Si desde cada componente de un estado compuesto q hay una transición etiquetada con α entonces hay una transición con α desde q al estado formado por los estados destinos de cada componente. Formalmente,

$$\frac{q_1 \xrightarrow{\alpha}_1 q_1' \qquad q_2 \xrightarrow{\alpha}_2 q_2'}{\langle q_1, q_2 \rangle \xrightarrow{\alpha} \langle q_1', q_2' \rangle}$$

  2. Si desde uno de los estados componentes hay una transición etiquetada con α y α no es de sincronización ($\alpha \notin \Lambda$), entonces hay una transición discreta con α en el sistema compuesto que permite avanzar sólo este componente. Formalmente,

$$\frac{q_1 \xrightarrow{\alpha}_1 q_1' \qquad \alpha \notin \Lambda}{\langle q_1, q_2 \rangle \xrightarrow{\alpha} \langle q_1', q_2' \rangle} \quad \text{y} \quad \frac{q_2 \xrightarrow{\alpha}_2 q_2' \qquad \alpha \notin \Lambda}{\langle q_1, q_2 \rangle \xrightarrow{\alpha} \langle q_1', q_2' \rangle}$$

2.3 Lógica TCTL: "Timed Computation Tree Logic"

Muchas propiedades importantes de los sistemas encuentran una expresión natural en lógica temporal [Pnu81]. La lógica temporal de tiempo real TCTL ("timed computation tree logic") [ACD90, Alu91, HNS+92] extiende los operadores temporales $\exists\mu$ (*existe una ejecución*) y $\forall\mu$ (*para todas las ejecuciones*) de CTL ("computation tree logic") [EC82, CES86] con restricciones temporales que permiten un razonamiento "cuantitativo" del tiempo. Si bien CTL es una lógica temporal adecuada para sistemas reactivos, ésta permite razonamiento "cualitativo" del tiempo basado en la noción de *secuencialidad* en las ejecuciones, pero no es posible expresar en ella restricciones temporales cuantitativas. En CTL pueden expresarse propiedades tales como "inevitablemente sucederá el evento *e*" ó "la propiedad *p* se satisface continuamente en todas las ejecuciones del sistema". Las fórmulas de TCTL permiten expresar, además, propiedades tales como "inevitablemente antes de un tiempo *t* sucederá el evento *e*" ó "la propiedad *p* se satisface continuamente entre los tiempos t_i y t_f para toda ejecución del sistema".

Las fórmulas de TCTL son interpretadas sobre los estados de un sistema de tiempo real y se definen a partir de un conjunto de predicados básicos (proposiciones atómicas) sobre los estados. El conjunto *P* de predicados sobre los estados de un grafo temporizado se define en [Oli94] como sigue,

$$p ::= @ = l \mid x_i \prec c \mid x_i - x_j \prec d$$

donde $l \in L$, x_i, $x_j \in X$, $c \in \mathbb{N}$, $d \in \mathbb{Z}$ y $\prec \in \{<, \leq, =, \geq, >\}$. Adicionalmente suelen considerarse los siguientes predicados básicos: *init*, *enable*(*e*) y *after*(*e*) [Oli94]. Informalmente, *init* define el estado inicial, *enable*(*e*) el conjunto de los estados que

tienen una transición habilitada etiquetada con *e* y, *after*(*e*) el conjunto de los estados que son alcanzados por una transición etiquetada con *e*. Asimismo, @ = *l* define al conjunto de los estados cuya locación es *l*. La semántica formal será descripta en la próxima sección.

Las fórmulas de TCTL se definen por la siguiente gramática:

$$\varphi := p \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \exists\mu_I \varphi_2 \mid \varphi_1 \forall\mu_I \varphi_2$$

donde $p \in P$ e I es un intervalo con extremos enteros positivos (I puede ser abierto o cerrado, acotado o no acotado).

Intuitivamente, $\varphi_1 \exists\mu_I \varphi_2$ significa que existe una ejecución (divergente) del sistema tal que φ_2 se cumple en un estado de la misma, en un tiempo t dentro del intervalo I y φ_1 se satisface continuamente en los estados previos. $\varphi_1 \forall\mu_I \varphi_2$ expresa que para todas las ejecuciones la propiedad anterior se cumple.

2.3.1 Semántica de TCTL

TCTL es una lógica que considera tiempo *ramificado* y es interpretada sobre una estructura arbórea de estados. Cada árbol representa un sistema reactivo, cuyas posibles secuencias de ejecución corresponden a caminos en el árbol, generados desde la raíz.

Definición 2.3. Para un sistema de transiciones etiquetado $T = \langle Q, \rightarrow \rangle$ correspondiente a un grafo temporizado G, un estado $q = \langle l, v \rangle \in Q$ y una fórmula φ de TCTL, la relación de satisfactibilidad $q \models \varphi$ es definida por inducción en φ, de la siguiente manera [ACD90, Oli94]:

$q \models @ = l$ si y solamente si, $Loc(q) = l$

$q \models x_i \prec c$ si y solamente si, $v(x_i) \prec c$

$q \models x_i - x_j \prec d$ si y solamente si, $v(x_i) - v(x_j) \prec d$

$q \models \neg\varphi$ si y solamente si, $q \not\models \varphi$

$q \models \varphi_1 \wedge \varphi_2$ si y solamente si, $q \models \varphi_1$ y $q \models \varphi_2$

$q \models \varphi_1 \exists\mu_I \varphi_2$ si y solamente si, $\exists r \in \Delta_G(q).\ \exists \pi \in \Pi(r).\ \sigma_r(\pi) \models \varphi_2 \wedge \delta_r(\pi) \in I \wedge \forall \pi' \prec \pi.\ \sigma_r(\pi') \models \varphi_1$

$q \models \varphi_1 \forall\mu_I \varphi_2$ si y solamente si, $\forall r \in \Delta_G(q).\ \exists \pi \in \Pi(r).\ \sigma_r(\pi) \models \varphi_2 \wedge \delta_r(\pi) \in I \wedge \forall \pi' \prec \pi.\ \sigma_r(\pi') \models \varphi_1$

donde, la relación $\prec$ entre posiciones de una ejecución se define como el orden lexicográfico de los pares ($\langle i,t \rangle \prec \langle i',t' \rangle$ si y solamente si: $i < i'$ ó, $i = i'$ y $t < t'$).

El conjunto de estados que satisface una fórmula φ es $|[\varphi]|$, el *conjunto característico* de φ.

2.3.2 Algunas Abreviaturas

- Escribiremos $\exists\mu$ y $\forall\mu$ para $I=[0, \infty)$; $\exists\mu_{\leq c}$ por $\exists\mu_{[0,c]}$ y $\forall\mu_{\leq c}$ por $\forall\mu_{[0,c]}$.
- $\exists\Diamond_I\varphi$ (*posible* φ) en lugar de $true\exists\mu_I\varphi_2$.
- $\forall\Diamond_I\varphi$ (*inevitable* φ) en lugar de $true\forall\mu_I\varphi_2$.
- $\exists\Box_I\varphi$ en lugar de $\neg\forall\Diamond_I\neg\varphi$.
- $\forall\Box_I\varphi$ (*siempre*φ) en lugar de $\neg\exists\Diamond_I\neg\varphi$.

De acuerdo a las abreviaturas dadas, $\exists\Diamond_I\varphi$ es satisfecha por el conjunto de estados desde los cuales es posible alcanzar un estado que verifica la fórmula φ, dentro del intervalo I. De esta manera se especifican los problemas de *alcanzabilidad acotada*. $\forall\Diamond_I\varphi$ establece que φ es *inevitable*. Un estado q satisface esta fórmula si, y solamente si, a partir de toda ejecución con estado inicial q existe un estado, que dentro del intervalo I, satisface φ. La fórmula $\forall\Diamond_{\leq c}\varphi$ expresa la propiedad de tiempo real de respuesta en tiempo acotado, en la cual un evento debe ocurrir antes de un cierto tiempo acotado c. $\forall\Box_I\varphi$ especifica que φ es un *invariante*. Esto es, un estado q satisface $\forall\Box_I\varphi$ si, y solamente si, toda ejecución con comienzo en q satisface continuamente φ, dentro del intervalo I. Finalmente, $\exists\Box_I\varphi$ caracteriza al conjunto de estados a partir de los cuales existe una ejecución que satisface continuamente φ, dentro del intervalo I.

2.4 Verificación de Modelos: Grafos Temporizados y TCTL

Dado un grafo temporizado G, que describe un sistema de tiempo real T (un sistema de transiciones etiquetado), y una fórmula φ de TCTL, que especifica un requerimiento, el problema de decidir, algorítmicamente, si T *satisface* φ es una instancia del problema de *verificación de modelos* ("*model-checking problem*"). Para esta instancia el problema fue solucionado por Alur, Courcoubetis y Dill [ACD90]. Su solución está basada en la construcción explícita, a partir del grafo de transiciones de estados T *infinito*, de un grafo cociente finito, llamado *grafo de regiones* ("*region graph*"). Las regiones están determinadas por una relación de equivalencia sobre el conjunto X –de los relojes de un grafo– que unifica configuraciones de los relojes desde las cuales los comportamientos futuros son esencialmente idénticos. Es decir, dos relojes x_1 y x_2 son equivalentes si las mismas secuencias de transiciones discretas son posibles desde x_1 y x_2.

La construcción del grafo de regiones conduce a un algoritmo de verificación de modelos que es exponencial en el número de relojes de entrada y en el tamaño de la más grande constante de G. Sin embargo, en la práctica es a menudo innecesario construir el

grafo de regiones entero. Puede representarse únicamente el conjunto de estados requerido para solucionar un problema específico (para una sentencia φ dada) y pueden representarse simbólicamente conjuntos de estados como predicados de estados. En otras palabras, en lugar de enumerar todas las regiones, un algoritmo de verificación de modelos *simbólico* ("*symbolic model-checking algorithm*") computa las regiones selectivamente y simbólicamente. El algoritmo propuesto en [HNS+92] usa transformadores de predicados para computar el conjunto de *sucesores* –análisis hacia adelante ("*forward analysis*")– o *predecesores* –análisis hacia atrás ("*backward analysis*")– de conjuntos de estados. El conjunto característico $|[\varphi]|$ de una fórmula φ es computado como un punto fijo definido en términos de los transformadores de predicados. Una explicación detallada del método de verificación de modelos simbólico puede encontrarse en [HNS+92, ACH+95]. Otros algoritmos son descriptos, por ejemplo, en [LL95, SS95, HKV96].

2.5 Discretización del Tiempo

2.5.1 Tiempo Continuo versus Tiempo Discreto

El *tiempo continuo* es generalmente representado usando números reales o racionales. Estos proveen todas las posibilidades de una matemática rica que generalmente es necesaria para tratar muchas clases de problemas. Los reales tienen dos propiedades principales que lo diferencian de los enteros y los racionales: la *densidad* y la *completitud*. La propiedad de densidad establece que entre dos puntos de tiempo cualesquiera existe siempre otro; la propiedad de completitud, que si una serie de puntos está acotada superiormente entonces existe un punto que es la mínima cota superior de la serie. La densidad distingue a los reales y los racionales de los enteros, la completitud distingue a los reales de los otros.

La principal ventaja de esta representación es que los reales (y los racionales) son los números generalmente usados en física para representar el tiempo. De esta manera es más fácil basar teorías sobre leyes físicas y representar cambio continuo. Asimismo, los reales (y los racionales) son un superconjunto de los enteros, por lo que son más generales y apropiados que estos últimos para representar tiempo, aunque poseen también algunas desventajas. Es difícil hacer teorías basados en los reales, debido en parte a la propiedad de densidad, la cual introduce el concepto de "infinito" aún en un intervalo acotado. Otra desventaja es en cuanto a su representación, especialmente en computadoras; la manera usual de representarlos es por medio de números de punto flotante, que son sólo una aproximación. Luego, surgen problemas, por ejemplo, para determinar si dos números son iguales o no (ellos pueden ser diferentes aproximaciones del mismo número).

El *tiempo discreto* es generalmente representado usando números enteros y más precisamente, naturales. La principal ventaja de esta representación es que una teoría

basada en enteros es más fácil de formular y de usar. Asimismo, los enteros tienen una representación directa y exacta en las computadoras, no es necesario trabajar con aproximaciones. La principal desventaja de los enteros es debida a su falta de expresividad, especialmente cuando deseamos representar cambio continuo. Por ejemplo, con enteros no es posible representar la solución de un sistema de ecuaciones si la misma es fraccional. Sin embargo, existen varios enfoques que permiten representar cambio continuo (es decir, funciones continuas) usando tiempo discreto. Por ejemplo, enfoques basados en los conceptos de "errores" y "granularidad" [Hob85, BB94, Euz95, BM97].

2.5.2 Un Modelo de Tiempo Discreto

En este trabajo estamos interesados en el análisis de sistemas de tiempo real para un dominio temporal discreto y particularmente en el marco de *grafos temporizados*. Numerosos trabajos previos consideran modelos de tiempo discreto que usan a los números naturales para modelar el tiempo [EMS+89, Ost87, Bur89, AH92, PL92, HMP92, AH93, AH94]. Este modelo es apropiado para, por ejemplo, ciertas clases de circuitos digitales sincrónicos, donde los eventos ocurren a valores exactos de incremento del tiempo de un reloj global.

En los casos en que ello no ocurre, el enfoque puede resultar igualmente válido y requiere que el tiempo continuo sea aproximado por la elección de una determinada *granularidad*: la unidad temporal más pequeña referenciable en el sistema. La idea es que la elección de una granularidad pequeña resulta suficiente para verificar ciertas propiedades donde valores temporales menores son indistinguibles para el proceso de verificación [EMS+89, ACD90].

En el enfoque de *relojes ficticios* las transiciones (los eventos) pueden ocurrir en tiempo continuo pero son registradas por un reloj global, en tiempo discreto. En este modelo se introduce generalmente una transición especial *tick* y el tiempo incrementa una unidad con cada *tick* (ver, por ejemplo, [Ost87, Bur89, HMP92, AH93, AH94]). Con la semántica de relojes ficticios, si varios eventos ocurren entre dos instantes consecutivos de tiempo (entre dos *ticks*) ellos pueden ser distinguidos solamente por el ordenamiento temporal, no por el valor real del tiempo.

Para los grafos temporizados hay una semántica alternativa basada en tiempo discreto (tomando valores en $\mathbb{N}$), la cual ha sido discutida en trabajos previos (ver, por ejemplo, [AH92]). De acuerdo a esta visión, los pasos temporales son múltiplos de una constante (*ticks* de un reloj global), y a cada instante el autómata puede elegir entre incrementar el tiempo o hacer alguna transición discreta. Consideremos el fragmento de un autómata temporizado con 2 relojes, de la figura 2.1(a). El autómata puede permanecer en el estado dejando pasar el tiempo (es decir, los valores de x_1 y x_2 aumentan simultáneamente) mientras $x_1 \leq u$. Asimismo, cuando x_1 alcanza un valor k (asumiendo que $k \leq u$) el autómata puede tomar una transición a otro estado y resetear x_1 a cero. Restringiendo el dominio del tiempo a $\mathbb{N}$, las condiciones de permanencia en cada estado (los *invariantes* de las

locaciones) pueden ser reemplazadas –interpretadas– por transiciones *tick* como se muestra en la figura 2.1(b).

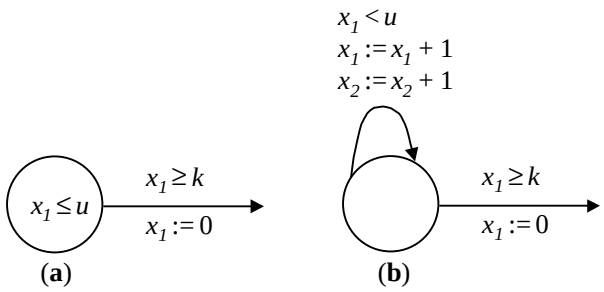

Fig. 2.1: Un autómata temporizado y su interpretación en tiempo discreto

Bajo esta interpretación los relojes son simplemente variables enteras no negativas acotadas, cuyos valores son incrementados simultáneamente por transiciones temporales y algunas de ellas reseteadas a cero por transiciones discretas. En particular, cualquier esquema de representación para una semántica densa, basada en desigualdades de relojes, puede ser especializado para una semántica discreta [BMT99]. Dado que, sobre un orden discreto, cualquier desigualdad de la forma $x_i < c$ puede ser escrita como la desigualdad no estricta $x_i \le c$-1, las *regiones* discretas pueden ser expresadas usando exclusivamente desigualdades no estrictas. Esta observación que ha sido aprovechada para mejorar la eficiencia de algoritmos de verificación modelos (ver, por ejemplo, [BMT99]), es considerada con particular interés en el presente trabajo ya que permite resaltar que con una discretización del tiempo "no mucho se pierde" [HMP92, GPV94, HK97, AMP98, BMT99]. En particular, [GPV94] construye dos discretizaciones de autómatas temporizados las cuales generan los mismos lenguajes "no temporizados" que sus versiones densas y satisfacen el mismo conjunto de propiedades TCTL.

Con una discretización, una *ejecución discreta* puede ser una ligera variación de algunas de las *ejecuciones densas* que ella representa, donde algunas transiciones tienen que tomarse simultáneamente mientras que en una ejecución densa las transiciones son separadas por una pequeña cantidad de tiempo. En el caso de estudio que abordaremos en el capítulo 4, las propiedades que son verificadas sobre un dominio discreto valen para el dominio denso de los reales o los racionales, según los resultados establecidos en [HMP92, AMP98, BMT99].

2.5.3 Interpretación Algorítmica de Grafos Temporizados

Dado un grafo temporizado $G = \langle L, X = \{x_1, \ldots, x_n\}, E, l^0, I \rangle$ y su sistema de transiciones etiquetado $T = \langle Q, \rightarrow \rangle$, una interpretación algorítmica del sistema en un dominio temporal discreto puede concebirse como un "loop" –sin cota de terminación– con inicialización,

en un lenguaje imperativo tipo *Small* [Gri81]. Las transiciones temporales y discretas son interpretadas por comandos guardados de un ciclo *Do*. Las guardas no son necesariamente disjuntas y se asume una elección *no determinista* de las mismas en la ejecución –el *Do* es no *determinístico*. El algoritmo tiene como precondición: el estado inicial del sistema es *válido* (satisface el invariante de la locación inicial). La figura 2.2 describe la interpretación algorítmica del grafo *G*.

$state_G$: L; // tipo de la variable estado del programa

$state_G := l^0$; // inicialización del estado

$x_1 := 0; \ldots; x_n := 0$ // inicialización de los relojes

{ precondition: $I_{l^0}(x_1, \ldots, x_n) = true$ } // precondición: el estado inicial es *válido*

Do // ciclo *Do*

$\square\ \psi_1(x_1, \ldots, x_n) \wedge state_G = l_1 \wedge I_{l1'}((\text{update}_X(x_1, \rho_1), \ldots, \text{update}_X(x_n, \rho_1))$

$\mapsto_{\alpha 1}$ $state_G := l_1'$;

$x_1 := \text{update}_X(x_1, \rho_1); \ldots; x_n := \text{update}_X(x_n, \rho_1)$

…….

$\square\ \psi_m(x_1, \ldots, x_n) \wedge state_G = l_m \wedge I_{lm'}((\text{update}_X(x_1, \rho_m), \ldots, \text{update}_X(x_n, \rho_m))$

$\mapsto_{\alpha m}$ $state_G := l_m'$;

$x_1 := \text{update}_X(x_1, \rho_m); \ldots; x_n := \text{update}_X(x_n, \rho_m)$

$\square\ I_{state_G}(x_1 + \delta, \ldots, x_n + \delta) \mapsto_{tick} x_1 := x_1 + \delta; \ldots; x_n := x_n + \delta$

EndDo

Fig. 2.2: Interpretación algorítmica de un grafo temporizado

Cada uno de los *m* primeros comandos guardados rotulados con etiquetas $\alpha_i \in \Lambda$ (con Λ el conjunto de etiquetas del sistema) corresponde a una transición $\langle l_i, \alpha_i, \psi_i, \rho_i, l_i' \rangle \in E$; donde, $\text{update}_X(x_j, \rho_i) = 0$, si $x_j \in \rho_i$ y $\text{update}_X(x_j, \rho_i) = x_j$, en caso contrario (cada rótulo α_i sería interpretado como una sentencia que inspecciona la ocurrencia del evento α_i). El último comando guardado representa un *tick* del *reloj global* que se refleja en el incremento de cada reloj de X con δ, el valor de granularidad elegido para el sistema. Un valor de $\delta = 1/M$, con $M > |X|$ resulta suficiente para asegurar que cada desigualdad de la forma $0 < x_1 < \ldots < x_n < 1$ tiene al menos una asignación posible (una solución, $x_j = j * \delta$). De esta manera aseguramos que todo estado posible del sistema tiene al menos un representante, un elemento en cada clase de equivalencia del grafo de regiones que caracteriza al sistema. Elegir a $\delta = 1/M$ es equivalente a asumir a $\mathbb{N}$ como dominio

temporal y multiplicar la escala del problema, es decir las constantes, por M –tanto en el grafo como en las propiedades a verificar. Esto permite trabajar con la aritmética de $\mathbb{N}$.

Las primitivas básicas consideradas de los relojes son 4: de *inicialización*, *reseteo*, *incremento* e *inspección* (*consulta*). Las primeras dos corresponden a una asignación con cero, el incremento es con δ y la inspección de un reloj es su valor.

Una versión equivalente a la anterior que hace explícita la utilización de un único *reloj global* se obtiene con ligeras variaciones. Se incorpora un reloj global *clock* al ambiente e inicializa cada reloj x_j del sistema representado por el grafo G con el valor de *clock*. Las condiciones $\psi_i(x_1, \ldots, x_n)$ e $I_{loc}(vx_1, \ldots, vx_n)$ del cuerpo del *Do* son reemplazadas por $\psi_i(clock - x_1, \ldots, clock - x_n)$ e $I_{loc}(clock - vx_1, \ldots, clock - vx_n)$. $\text{update}_X(x_j, \rho_i)$ se redefine como $\text{update}_X(x_j, \rho_i) = clock$, si $x_j \in \rho_i$ y $\text{update}_X(x_j, \rho_i) = x_j$, en caso contrario. El último comando guardado (*tick*) sólo incrementa el reloj global *clock* con δ (los x_j no son modificados). Esta versión resalta la existencia de un único reloj –el reloj del sistema– que regula el paso del tiempo y que no es reseteado nunca. Cada x_j es en realidad un cronómetro que mide el tiempo transcurrido desde la última vez que fue reseteado. Consecuentemente, las cuatro primitivas básicas de los relojes x_j del sistema se definen para esta versión como sigue. Las primitivas de inicialización y reseteo corresponden a la asignación $x_j := clock$; el incremento es implícito para cada x_j (sólo se incrementa explícitamente el reloj global *clock*); y, la *inspección* del reloj x_j es el valor $clock - x_j$.

Cada ejecución de un programa *Do* es una traza de ejecución válida del grafo G, a partir de T, y el conjunto de todas las ejecuciones es capturado por el *no-determinismo* del programa.

La *composición paralela* de dos programas *Do* –como el de la figura 2.2– es un programa *Do* que compone las acciones de inicialización de cada componente y donde cada comando guardado resulta de la composición de las transiciones correspondientes. Esto es, si α_i es de sincronización entonces se genera un comando guardado que compone con conjunción las guardas y con ";" los comandos. Para cada α_i que no es de sincronización, el comando guardado del *Do* componente se incluye en el *Do* de la composición. Los comandos guardados de incremento temporal (*tick*) reciben el mismo tratamiento que las transiciones con etiquetas de sincronización, ya que un paso temporal es el mismo en cada componente.

Verificar que un predicado P es un invariante de un programa *Do* corresponde a probar las siguientes dos condiciones:

- $Init \Rightarrow P$
- $P \wedge B_k \Rightarrow wp(S_k, P)$, para todo comando $B_k \mapsto_{lab} S_k$, con $lab \in \Lambda \cup \{tick\}$

donde *Init* es el predicado que caracteriza al estado inicial y *wp* (weakest precondition) es el tranformador de predicados propuesto por Dijkstra en [Dij79]. En el cálculo *wp*, un programa S termina en un estado verificando un predicado q, cuando la ejecución es comenzada en un estado que satisface el predicado $wp(S,q)$. Cabe notar que si se

demuestra que *P* es un invariante, para cualquier elección del no-determinismo, *P* vale continuamente.

Existen extensiones recientes del cálculo *wp* que definen nuevos transformadores de predicados para trabajar, en el contexto de programas concurrentes, con propiedades de seguridad y estabilidad de propiedades en un proceso –*safety*–, la invarianza de predicados en las computaciones –*to-always*– y propiedades de progreso –*progress*–, tales como la *alcanzabilidad* de un estado (predicado) desde otro –*leads-to*. Algunos de estos trabajos son: [JKR89, Kna90, Lam90, ChS95, DS97]. Las áreas de aplicación consideradas en la práctica hasta el momento se restringen al análisis de flujo de datos no determinístico en redes, sistemas tolerantes a fallas y protocolos simples de comunicación [DS97]. Sin embargo, la considerable complejidad del enfoque en las extensiones, el escaso estado de desarrollo, la indisponibilidad de herramientas computacionales de asistencia y la falta de consideraciones de nociones temporales, hacen que este enfoque sea, por un lado, insatisfactorio para la especificación de sistemas con requerimientos temporales y por el otro, una promisoria dirección para la realización de nuevos trabajos de investigación.

Capítulo 3

Formalización de Grafos Temporizados y TCTL en Coq

En este capítulo formalizamos en *Coq grafos temporizados* y la lógica *TCTL*. Incluimos además operadores de la lógica *CTL* ("computation tree logic") [CES86] y algunas definiciones alternativas para los operadores que permiten expresar *invarianza* y *alcanzabilidad*, con tipos *inductivos* (bajo la noción de estados alcanzables) y *co-inductivos* (bajo la noción de trazas –infinitas– de ejecución). Las formalizaciones introducidas serán utilizadas en el próximo capítulo en el análisis de un caso de estudio y en el capítulo 5, en descripciones *genéricas* de grafos temporizados.

3.1 Nociones Temporales

En nuestra formalización asumiremos a ℕ como dominio temporal discreto. Este corresponde al tipo inductivo `nat` de *Coq*. A continuación introducimos algunas definiciones básicas en una sección *Coq*.

```
Section Time_Clocks.

   Definition Instant  := nat.
   Definition Clock    := nat.

   Definition lt_Ck    := lt.
   Definition le_Ck    := le.
   Definition gt_Ck    := gt.
   Definition ge_Ck    := ge.
   Definition eq_Ck    := [x,y:Clock] x=y.

   Definition Ini_Ck   := O.
   Definition tick     :  Instant := (1).
   Definition plus_Ck  := plus.
   Definition Inc      := [x:Clock] (plus_Ck x tick).
```

```
  Definition Reset    := 0.
  Definition time0    := 0.

End Time_Clocks.
```

`Instant` y `Clock` corresponden al tipo de los valores temporales y las valuaciones de los relojes; `lt_Ck` (<), `le_Ck` (≤), `gt_Ck` (>), `ge_Ck` (≥) y `eq_Ck` (=) son las relaciones binarias habituales sobre ℕ; `Ini_Ck` es el valor inicial de los relojes; `tick` es la *granularidad* del sistema (la más pequeña unidad temporal referenciable); `plus_Ck` es la suma de `nat`; `Inc` es la operación que incrementa un `tick` (es una función que recibe un objeto `x` de tipo `Clock` y retorna su suma con `tick`) y, `Reset` y `time0` corresponden al valor de reseteo de los relojes y el tiempo inicial del sistema, respectivamente.

Nota 3.1. En las formalizaciones en *Coq* de este capítulo y del resto del trabajo obviaremos los argumentos en las aplicaciones que pueden ser automáticamente inferidos por el sistema. Esto se consigue en *Coq* inicializando el desarrollo con el comando `Implicit Arguments On` [Bar+99].

3.2 Definiciones Elementales de Grafos Temporizados

Sea $G = \langle Loc = \{loc_0, \ldots, loc_n\}, X, Trans, loc_0, Inv \rangle$ un grafo temporizado. En la formalización de los sistemas asumiremos un conjunto global `Label` de etiquetas en vez de conjuntos particulares para cada grafo *G*, de acuerdo a lo expresado en el capítulo 2 (definición 2.1). El conjunto `Label` puede definirse por un tipo inductivo de la siguiente manera:

```
Inductive Label : Set := Lab₁ : Label | ... | Labₘ : Label
  | Tick : Label.
```

$Lab_1, \ldots, Lab_m$ y *Tick* son los constructores del tipo Label. $Lab_1, \ldots, Lab_m$ representan las etiquetas que corresponden a las transiciones discretas; *Tick* es una etiqueta distinguida que caracteriza a las transiciones temporales.

Las *locaciones* del grafo temporizado *G* pueden especificarse también como un tipo enumerado,

```
Inductive Loc : Set := Loc₀ : Loc | ...| Locₙ : Loc.
```

El estado del sistema, en un instante dado, está determinado por una locación de `Loc` y una tupla de valores de tipo `Clock` (la cantidad de componentes de esta tupla dependerá en cada caso del sistema concreto que se esté analizando). Asumiremos que `Clocks` representa el tipo de dicha tupla (correspondiente al conjunto *X*); `InicAll` es la tupla de

relojes inicializados (todos puestos en cero); e, `IncAll` es la función que incrementa todos los relojes de *X* con la función `Inc` (formalizada en la sección 3.1).

```
Definition State := Loc * Clocks.
Definition St_ini : State := (Loc₀,InicAll).
```

`St_ini` es el estado inicial del sistema, compuesto por la locación inicial y la tupla de relojes inicializados.

Notación. La cuantificación universal sobre un tipo `S` se escribe en *Coq* "`(x:S) P`". Sin embargo, usaremos la notación "`∀x∈S P`" en este trabajo para dar más claridad a las especificaciones.

Los *invariantes* de las locaciones del grafo pueden ser definidos por predicados sobre los estados, bajo el siguiente patrón

```
Inductive Inv : State -> Prop :=
  ILoc₀  : ∀x∈Clocks (Icond₁ x) -> (Inv (Loc₀,x))
  ...
  |ILocₙ : ∀x∈Clocks (Icondₙ x) -> (Inv (Locₙ,x)).
```

$\texttt{Icond}_i$ es el predicado de permanencia en la locación $\texttt{Loc}_i$, para $0 \leq i \leq n$. El constructor $\texttt{ILoc}_i$ construye pruebas del invariante en la locación $\texttt{Loc}_i$, para los valores de los relojes que cumplen $\texttt{Icond}_i$. Esto es, dado un objeto `x` de tipo `Clocks` y una prueba `H` de $(\texttt{Icond}_i\ x)$, $(\texttt{ILoc}_i\ x\ H)$ es una prueba de $(\texttt{Inv}\ (\texttt{Loc}_i,x))$.

Nota 3.2. Asumiremos de aquí en adelante en este trabajo que el predicado que describe a los invariantes de las locaciones `Inv` de un sistema *G* se satisface para el estado inicial `St_ini` de *G* (esto es, `St_ini` es un estado *válido* de *G*).

Las transiciones discretas y temporales pueden ser definidas como relaciones entre estados y etiquetas. En *Coq*, usando `Inductive`, de la siguiente manera:

```
Inductive Trans : State -> Label -> State -> Prop :=
  trans₁  :∀x∈Clocks (Tcond₁ x) -> (Inv (Locⱼ,new_x₁)) ->
           (Trans (Locᵢ,x) Labₖ (Locⱼ,new_x₁))
  ...
  |transₚ :∀x∈Clocks (Tcondₚ x) -> (Inv (Locⱼ,new_xₚ)) ->
           (Trans (Locᵢ,x) Labₖ (Locⱼ,new_xₚ))
  |tTick  :∀x∈Clocks ∀l∈Loc (Inv (l,(IncAll x))) ->
           (Trans (l,x) Tick (l,(IncAll x)))
```

$\texttt{Tcond}_1$, ..., $\texttt{Tcond}_p$ son los predicados asociados a cada una de las *p* transiciones del grafo; $\texttt{new\_x}_i$ es la tupla de `Clocks` donde cada reloj conserva su valor de *x* o es reseteado a cero, con la constante `Reset`; trans_1, ..., trans_p corresponden a transiciones discretas (del conjunto *Trans* de *G*) y *tTick* al conjunto de transiciones temporales –una por locación.

trans_i (para $1 \leq i \leq p$) construye pruebas de transiciones de una locación $\texttt{Loc}_i$ a otra $\texttt{Loc}_j$ por una etiqueta $\texttt{Lab}_k$, para los valores de los relojes que satisfacen $\texttt{Tcond}_i$ e $\texttt{(Inv (Loc}_j\texttt{,new\_x}_i\texttt{))}$; *tTick* construye pruebas de transiciones de una locación a ella misma, incrementando el valor de los relojes en un *tick* con `IncAll`, siempre que se cumple el predicado `Inv` para los nuevos valores de los relojes en la misma locación.

Observaciones

- Al asumir un conjunto global Label de etiquetas simplificamos las formalizaciones, reduciendo también el tamaño de las pruebas sobre los sistemas, ya que evitamos trabajar con conjuntos y con operaciones de conjuntos (como la pertenencia) en el análisis de las transiciones de un sistema compuesto. Es importante aclarar que esta hipótesis no condiciona el incremento de un sistema al agregar nuevos componentes.
- El estado inicial del sistema asume valores nulos para los relojes, en relación a la definición de grafos temporizados dada en la sección 2.1. Sin embargo, esta es una convención y podríamos entonces permitir valores iniciales distintos de cero e incluso diferentes para cada reloj.
- Las transiciones, discretas y temporales, formalizadas preservan el cumplimiento del predicado `Inv` y asumen que éste vale en el estado de origen de la transición. Esta suposición se justifica porque consideraremos conjuntos de estados generados por transiciones –trazas de ejecución– a partir de un estado inicial *válido*. Luego, todos los estados de una ejecución serán estados válidos del sistema, tal cual lo establece la definición 2.2 de la sección 2.1.1.

3.3 Operadores Temporales con Tipos Inductivos

En esta sección formalizamos dos operadores temporales con tipos inductivos que permiten especificar *invarianza* y *alcanzabilidad*. Incluimos la versión de CTL y la cuantitativa temporal de TCTL.

3.3.1 Estados Alcanzables

Los estados *alcanzables* desde uno inicial dado representan estados pertenecientes a trazas de ejecución de un grafo temporizado que comienzan en dicho estado. Sea `S` el tipo de los estados de un sistema de tiempo real y `tr` una relación de transición entre estados

de `S` que respeta el formato dado en la sección previa. El conjunto de estados alcanzables desde uno inicial `Sini` a través de `tr` puede definirse inductivamente como sigue,

```
Variable S  : Set.
Variable tr : S -> Label -> S -> Prop.

Inductive RState [Sini:S] : S -> Prop :=
  rsIni : (RState Sini Sini)

  |rsNext : ∀s1,s2∈S ∀l∈Label
    (RState Sini s1) ->
    (tr s1 l s2) ->
    (RState Sini s2).
```

Esto es, el estado inicial es alcanzable (`rsIni`) y los estados a los que se llega desde un estado alcanzable por una transición también son alcanzables (`rsNext`). Asimismo, los estados alcanzables en tiempo *t* desde un estado inicial `Sini` pueden definirse como una generalización de la definición previa.

```
Inductive RState_T [Sini:S] : S -> Instant  -> Prop :=
  rsIni_T : (RState_T Sini Sini time0)

  |rsNoTime_T : ∀s1,s2∈S ∀l∈Label ∀t∈Instant
    (RState_T Sini s1 t) ->
    ~l=Tick ->
    (tr s1 l s2) ->
    (RState_T Sini s2 t)

  |rsTime_T : ∀s1,s2∈S ∀t∈Instant
    (RState_T Sini s1 t) ->
    (tr s1 Tick s2) ->
    (RState_T Sini s2 (Inc t)).
```

El estado inicial es alcanzable en tiempo `time0` (`rsIni_T`); las transiciones discretas no insumen tiempo (`rsNoTime_T`); y, las transiciones temporales representan el paso de un *tick* (`rsTime_T`).

3.3.2 Invarianza y Alcanzabilidad

En esta sección formalizamos los operadores temporales cualitativos $\forall\Box$ y $\exists\Diamond$, y sus versiones temporales cuantitativas, con tipos inductivos.

- $\forall\Box P$ y $\forall\Box_I P$ permiten especificar propiedades invariantes e invarianza acotada (ver sección 2.3.2).

```
Definition ForAll := [Sini:S; P:S->Prop]
  ∀s∈S (RState Sini s) -> (P s).

Definition ForAll_T := [Sini:S; P:S->Prop; bound:Instant->Prop]
  ∀s∈S ∀t∈Instant (bound t) -> (RState_T Sini s t) -> (P s).
```

`ForAll` especifica que todos los estados alcanzables desde un estado inicial `Sini` cumplen la propiedad `P`, mientras que `ForAll_T` especifica la propiedad anterior para todos los valores temporales `t` que satisfacen `(bound t)`, es decir $t \in I$. La primera definición se obtiene de la segunda instanciando a `bound` con el predicado constante `True`, para todo `t∈Instant`.

- $\exists\Diamond P$ y $\exists\Diamond_I P$ permiten especificar problemas de alcanzabilidad no acotada y acotada, respectivamente (ver sección 2.3.2).

```
Inductive Exists [Sini:S; P:S->Prop] : Prop :=
  exists : ∀s∈S (RState Sini s) -> (P s) ->
    (Exists Sini P).

Inductive Exists_T [Sini:S; P:S->Prop; bound:Instant->Prop] : Prop :=
  exists_T : ∀s∈S ∀t∈Instant (bound t) ->
    (RState_T Sini s t) -> (P s) ->
    (Exists_T Sini P bound).
```

La relación inductiva `Exists` especifica que existe un estado alcanzable desde un estado inicial `Sini` que cumple la propiedad `P`, mientras que `Exists_T` define la propiedad anterior para todos los valores temporales `t` que satisfacen `(bound t)`. Luego, una prueba de $\exists\Diamond P$ ($\exists\Diamond_I P$) se obtiene a partir de un estado alcanzable que verifica P (y `(bound t)` en la versión temporal acotada sobre un intervalo I) y consiste en la construcción de un camino desde el estado inicial.

Nota 3.3. En la demostración de propiedades de un sistema G que involucran los operadores $\forall\Box$ ($\forall\Box_I$) y $\exists\Diamond$ ($\exists\Diamond_I$) requerimos que el estado inicial considerado satisfaga el invariante de las locaciones *inv* de G. Es decir, si este estado es el inicial de G ($Sini_G$) el requerimiento se satisface pues previamente exigimos que $Sini_G$ debe ser un estado *válido* de G (ver nota 3.2). En otro caso, las propiedades demostradas tienen sentido si el estado inicial considerado es *válido*. Este requerimiento podría ser explicitado en la definición de estados alcanzables (el estado inicial es alcanzable a partir de él mismo si cumple el invariante *inv*), sin embargo de esta manera complicaríamos la estructura de las pruebas (particularmente las de alcanzabilidad) y aumentaríamos el tamaño de los términos de prueba. Asimismo, puesto que las transiciones preservan, por definición, el cumplimiento de *inv*, en propiedades compuestas del tipo $\forall\Box\exists\Diamond P$, los estados iniciales considerados para $\exists\Diamond P$ trivialmente satisfacen el requerimiento si el estado inicial para $\forall\Box\exists\Diamond P$ lo

cumple. Luego, en la demostración de toda propiedad requerimos la prueba adicional de que el estado inicial es *válido*.

Definiciones más generales de los operadores $\forall\Box$ ($\forall\Box_I$) y $\exists\Diamond$ ($\exists\Diamond_I$) pueden ser útiles para especificar propiedades donde los estados iniciales están caracterizados por un predicado *Q*. Estas versiones pueden definirse a partir de las previas, como sigue.

- $\forall\Box(Q \Rightarrow \forall\Box P)$ y $\forall\Box(Q \Rightarrow \forall\Box_I P)$

```
Definition ForAll_from := [Sini:S; Q,P:S->Prop]
  (ForAll Sini ( [s:S] (Q s)->(ForAll s P) )).

Definition ForAll_from_T :=
  [Sini:S; Q,P:S->Prop, bound:Instant->Prop]
    (ForAll Sini ( [s:S] (Q s)->(ForAll_T s P bound) )).
```

- $\forall\Box(Q \Rightarrow \exists\Diamond P)$ y $\forall\Box(Q \Rightarrow \exists\Diamond_I P)$.

```
Definition Exists_from := [Sini:S; Q,P:S->Prop]
  (ForAll Sini ( [s:S] (Q s)->(Exists s P) )).

Definition Exists_from_T :=
  [Sini:S; Q,P:S->Prop; bound:Instant->Prop]
    (ForAll Sini ( [s:S] (Q s)->(Exists_T s P bound)) ).
```

3.3.3 Algunas Propiedades

Propiedades conocidas de los operadores temporales formalizados en la sección previa pueden demostrarse utilizando el asistente de pruebas *Coq*. A continuación destacamos, a modo de ejemplo, algunas de ellas. Las demostraciones pueden consultarse en el apéndice B.1.2.

```
Theorem Mon_I : ∀Sini∈S ∀Pg,Pp∈(S->Prop)
  (ForAll Sini Pg) -> (∀s∈S (Pg s)->(Pp s)) -> (ForAll Sini Pp).

Theorem Mon_I_T : ∀Sini∈S ∀Pg,Pp∈(S->Prop) ∀bound∈(Instant->Prop)
  (ForAll_T Sini Pg bound) -> (∀s∈S (Pg s)->(Pp s)) ->
  (ForAll_T Sini Pp bound).

Theorem Mon_I_EX : ∀Sini∈S ∀∈Pg,Pp∈(S->Prop)
  (Exists Sini Pg) -> (∀s∈S (Pg s)->(Pp s)) -> (Exists Sini Pp).
```

```
Theorem Mon_I_EX_T : ∀Sini∈S ∀∈Pg,Pp∈(S->Prop)
  ∀bound∈(Instant->Prop) (Exists_T Sini Pg bound) ->
  (∀s∈S (Pg s)->(Pp s)) -> (Exists_T Sini Pp bound).

Theorem Conj : ∀Sini∈S ∀P1,P2∈(S->Prop) (ForAll Sini P1) ->
  (ForAll Sini P2) -> (ForAll Sini ( [s:S](P1 s)/\(P2 s) )).

Theorem Conj_T : ∀Sini∈S ∀P1,P2∈(S->Prop) ∀bound∈(Instant->Prop)
  (ForAll_T Sini P1 bound) -> (ForAll_T Sini P2 bound) ->
  (ForAll_T Sini ( [s:S](P1 s)/\(P2 s) ) bound).
```

El teorema `Mon_I` permite probar un invariante `Pp` a partir del invariante `Pg` si `Pp` es consecuencia lógica de `Pg`. `Mon_I_EX` es la propiedad previa para el problema de alcanzabilidad ($\exists\Diamond$). `Conj` establece que la conjunción de invariantes es un invariante. `Mon_I_T`, `Mon_I_EX_T` y `Conj_T` son las versiones para los operadores temporales acotados de `Mon_I`, `Mon_I_EX` y `Conj`, respectivamente.

```
Lemma RState_Trans : ∀s1,s2,s3∈S
  (RState s1 s2) -> (RState s2 s3) -> (RState s1 s3).

Lemma RState_Trans_T : ∀s1,s2,s3∈S ∀t1,t2∈Instant
  (RState_T s1 s2 t1) -> (RState_T s2 s3 t2) ->
  (RState_T s1 s3 (plus_Ck t1 t2)).

Theorem StepsEX : ∀s1,s2∈S ∀P∈(S->Prop)
  (RState s1 s2) -> (Exists s2 P) -> (Exists s1 P).
```

`RState_Trans` es la transitividad de la relación de alcanzabilidad `RState` y, `RState_Trans_T` la correspondiente a la relación `RState_T`. `StepsEX` establece que si $\exists\Diamond P$ vale a partir de una estado inicial `s2` y `s2` es alcanzable a partir del estado `s1` entonces $\exists\Diamond P$ vale a partir del estado inicial `s1`. `StepsEX` usa en su prueba al lema `RState_Trans`.

Los siguientes dos teoremas corresponden a las equivalencias $\forall\Box P \Leftrightarrow \neg\exists\Diamond\neg P$ y $\forall\Box_I P \Leftrightarrow \neg\exists\Diamond_I\neg P$. Los mismos se prueban en lógica clásica, usando el principio del tercero excluido; en *Coq*, el axioma `classic: ∀P∈Prop P\/~P`.

```
Theorem ForAll_EX : ∀Sini∈S ∀P∈(S->Prop)
  (ForAll Sini P) <-> ~(Exists Sini ( [s:S]~(P s) )).

Theorem ForAll_EX_T : ∀Sini∈S ∀P∈(S->Prop) ∀bound∈(Instant->Prop)
  (ForAll_T Sini P bound) <-> ~(Exists_T Sini ( [s:S]~(P s) ) bound).
```

3.4 Necesidad de Tipos Co-Inductivos

Si bien muchas propiedades pueden especificarse usando los operadores $\exists\Diamond$ y $\forall\Box$ ($\exists\Diamond_I$ y $\forall\Box_I$), otras requieren el uso de las versiones más generales de $\exists\mu$ y $\forall\mu$ ($\exists\mu_I$ y $\forall\mu_I$). Por ejemplo, los problemas de respuesta en tiempo acotado ("siempre antes de un tiempo *t* se cumple una propiedad *P*, para toda ejecución del sistema"). Los operadores $\forall\Diamond$ y $\exists\Box$ ($\forall\Diamond_I$ y $\exists\Box_I$) no pueden ser formalizados en base a solamente tipos inductivos y la noción de *estados alcanzables*, sino que resulta necesario formalizar el concepto de *traza de ejecución* y consecuentemente, la definición de tipos *co-inductivos* para una descripción completa de todas las fórmulas de TCTL.

3.4.1 Nociones Preliminares

Como expresamos en el capítulo 1, los tipos *co-inductivos* son tipos recursivos que pueden contener objetos infinitos (no bien fundados). Un ejemplo de éstos es el tipo de las secuencias infinitas formadas con elementos de un tipo `T`, también llamadas "*streams*". Este tipo se formaliza en *Coq* de la siguiente manera (y se carga en el ambiente *Coq* con la declaración `Require Streams`):

```
CoInductive Stream [T:Set]: Set := Cons: T -> (Stream T) -> (Stream T).
```

`Cons` es el constructor del tipo `Stream`. Asumiremos alternativamente una notación infija para `Cons` dada por operador `^`, que definimos asociativo a derecha con el uso del comando `Infix RIGHTA` de *Coq*.

```
Infix RIGHTA 9 "^" Cons.
```

La *inducción estructural* es la manera de expresar que los tipos *inductivos* pueden contener sólo objetos *bien fundados*. Para tipos co-inductivos este principio de eliminación no es válido, aunque sí el principio de *análisis de casos* (válido también para tipos inductivos). Este principio puede ser usado, por ejemplo, para definir las funciones `hd` y `tl` que retornan la cabeza y la cola, respectivamente, de una secuencia infinita de tipo `Stream` [Gim98].

```
Definition hd := [T:Set; x:(Stream T)] Cases x of (Cons a y) => a end.

Definition tl := [T:Set; x:(Stream T)] Cases x of (Cons a y) => y end.
```

Por más información acerca de *streams* y tipos co-inductivos en general, ver [Coq93, Pau93a, Gim95, Gim96, Gim98, Bar+99].

3.4.2 Operadores Temporales Cualitativos

En esta sección presentamos una formalización de los operadores temporales cualitativos de CTL. Es decir, del fragmento de TCTL que no permite razonamiento cuantitativo del tiempo.

Sea `S` el tipo de los estados de un sistema de tiempo real y `tr` una relación de transición entre estados de `S` que respeta el formato dado en la sección 3.3.1. Definimos las trazas de estados generadas por `tr` como un predicado co-inductivo `isTrace` sobre secuencias infinitas (`Stream`) de estados de `S`, que llamaremos `SPath`. Más precisamente, como el más grande predicado cerrado bajo la regla de introducción *`is_trace`* que se enuncia abajo. `isTraceFrom` define a las trazas que poseen comienzo en un estado inicial dado.

```
Variable S  : Set.
Variable tr : S -> Label -> S -> Prop.

Definition SPath := (Stream S).

CoInductive isTrace : SPath -> Prop :=
  is_trace : ∀x∈SPath ∀s1,s2∈S ∀l∈Label
    (tr s1 l s2) -> (isTrace s2^x) ->
    (isTrace s1^s2^x).

Definition isTraceFrom := [Sini:S; x:SPath]
  Sini=(hd x) /\ (isTrace x).
```

Una prueba de `(isTrace x)` es semejante a una secuencia infinita decorada en cada posición con un subsecuencia `z` de `x` y una prueba de que hay una transición al comienzo de `z`.

Observemos que, al igual que en la sección 3.3, no explicitamos en la formalización de `isTraceFrom` que el estado inicial de la traza debe ser un estado *válido*. Las razones son las mismas que las expuestas en dicha sección.

Usualmente las propiedades se definen sobre estados, sin embargo en esta sección consideramos que las mismas tienen como dominio a *streams* de estados, a fin de permitir definir a los operadores temporales composicionales (es decir, poder expresar fórmulas compuestas tales como: $\forall\Box\exists\Diamond P$, $\forall\Diamond\exists\Box P$, etc). Esta concepción generaliza y no restringe las propiedades definibles –las propiedades sobre los estados refieren al primer elemento del *stream*.

El operador *until* (μ) –no cuantificado– puede ser definido por un predicado inductivo sobre las secuencias infinitas de estados. $P\mu Q$ expresa que Q vale en un estado y P continuamente en los estados previos.

```
Inductive Until [P,Q:SPath->Prop] : SPath -> Prop :=
  UntilFurther : ∀s∈S ∀x∈SPath (P s^x) -> (Until P Q x) ->
    (Until P Q s^x)
  |UntilHere : ∀x∈SPath (Q x) ->
    (Until P Q x).
```

Una prueba de $P\mu Q$ es un objeto finito. Esto significa que la búsqueda de un testigo de la verdad de Q no puede postergarse indefinidamente. Es decir, la prueba sólo aplica `UntilFurther` un número finito de veces (posiblemente nulo) antes de aplicar `UntilHere`.

Podemos formalizar ahora los operadores $\exists\mu$ ($P\exists\mu Q$) y $\forall\mu$ ($P\forall\mu Q$) sobre las trazas de un sistema con estado de origen dado, como sigue:

```
Inductive EX_Until [Sini:S; P,Q:SPath->Prop] : Prop :=
  ExUntil : ∀x∈SPath (isTraceFrom Sini x) -> (Until P Q x) ->
    (EX_Until Sini P Q).

 Definition FA_Until := [Sini:S; P,Q:SPath->Prop] ∀x∈SPath
  (isTraceFrom Sini x) -> (Until P Q x).
```

`EX_Until` es un predicado inductivo que permite expresar una propiedad $P\exists\mu Q$ para una traza con origen en un estado inicial `Sini`; `FA_Until` es una definición para $P\forall\mu Q$ sobre las trazas con estado inicial `Sini`.

Luego, podríamos definir las abreviaturas dadas en 2.3.2 para los operadores $\exists\Diamond$, $\forall\Diamond$, $\exists\Box$ y $\forall\Box$ a partir de las formalizaciones para $\exists\mu$ y $\forall\mu$. Sin embargo optamos por dar una definición explícita de éstos para simplificar las pruebas, ya que sino las mismas se tornan más complejas, porque es necesario trabajar con negaciones. Por ejemplo, según la sección 2.3.2, $\forall\Box P = \neg(true\exists\mu\neg P)$ y esto equivale a probar que no existe un estado alcanzable –subtraza– que cumpla $\neg P$ (es decir, que no existe un prefijo finito de una traza infinita que cumpla $\neg P$). Sin embargo, resulta más fácil verificar que todos los estados alcanzables –todas las subtrazas– cumplen P. Nuevamente, estamos razonando bajo la hipótesis del tercero excluido, en lógica clásica.

A continuación formalizamos $\Box P$ como un predicado co-inductivo `ForAllS` y $\Diamond P$ como un predicado inductivo `ExistsS`, ambos sobre secuencias infinitas de estados (`SPath`).

```
CoInductive ForAllS [P:SPath->Prop]: Spath -> Prop :=
  ForallS : ∀x∈SPath ∀s∈S (P s^x) -> (ForAllS P x) ->
    (ForAllS P s^x).
```

```
Inductive ExistsS [P:SPath->Prop] : SPath -> Prop :=
  Here : ∀x∈SPath (P x) -> (ExistsS P x)
  |Further : ∀x∈SPath ∀s∈S (ExistsS P x) -> (ExistsS P s^x).
```

`ForAllS` caracteriza a las secuencias infinitas que satisfacen continuamente un predicado `P`. `ExistsS` a las que poseen al menos una subsecuencia que cumple `P`. Una prueba de la proposición $\Box P(x)$ es semejante a una secuencia infinita decorada en cada posición con una subsecuencia z de x y una prueba de que z satisface P. En tanto que una prueba de $\Diamond P(x)$ es un camino desde x a alguna subsecuencia z de x que verifica P. En este último caso, la exhibición de z no puede ser postergada indefinidamente y por ello $\Diamond$ es definido inductivamente.

En función de `ForAllS` y `ExistsS` especificamos $\exists\Diamond$ (`Posible`), $\forall\Diamond$ (`Inevitable`), $\exists\Box$ (`SafePath`) y $\forall\Box$ (`Always`). Las propiedades existenciales las formalizamos como predicados inductivos y las universales como definiciones *Coq*. Todas ellas usan `isTraceFrom` para permitir cuantificar sobre las trazas de ejecución generadas a partir de un estado inicial `Sini`.

```
Inductive Posible [Sini:S; P:SPath->Prop] : Prop :=
  posible : ∀x∈SPath (isTraceFrom Sini x) -> (ExistsS P x) ->
    (PosibleS Sini P).

Definition Inevitable := [Sini:S; P:SPath->Prop]
  ∀x∈SPath (isTraceFrom Sini x) -> (ExistsS P x).

Inductive SafePath [Sini:S; P:SPath->Prop] : Prop :=
  safePath : ∀x∈SPath (isTraceFrom Sini x) -> (ForAllS P x) ->
    (SafePath Sini P).

Definition Always := [Sini:S; P:SPath->Prop]
  ∀x∈SPath (isTraceFrom Sini x) -> (ForAllS P x).
```

Nota 3.4. Es importante remarcar que es imposible representar los operadores $\forall\Diamond$ y $\exists\Box$ exclusivamente con tipos inductivos, tal como lo hicimos en la sección 3.3 para $\exists\Diamond$ y $\forall\Box$, debido a que no se puede expresar inductivamente la noción de secuencialidad entre los estados por ser ésta de naturaleza *infinita* para la clase de sistemas en estudio. Es por ello que los tipos co-inductivos resultan necesarios y en particular la formalización de *traza de ejecución* –como una secuencia infinita (*stream*) de estados– en lugar de *conjunto de estados alcanzables* –suficiente para verificar invarianza y alcanzabilidad (acotadas o no).

Ejemplo 3.1. Sea P una propiedad sobre los estados del sistema e *IsSini* el predicado característico de un estado *Sini*. La fórmula $IsSini \Rightarrow \forall\Box P$ se especifica como sigue: `(Always tr Sini P')`, donde `tr` es la relación de transición del sistema y `P'`

corresponde al predicado "`[x:SPath] (P (hd x))`" –con `SPath` el tipo `(Stream S)` y `S` es el tipo de los estados (`Sini∈S`). Asimismo, una fórmula temporal compuesta *IsSini* $\Rightarrow \forall\Diamond\exists\Diamond P$ puede especificarse como: `(Inevitable tr Sini Q)`, donde `Q` es el predicado "`[x:SPath] (Posible tr (hd x) P')`" y `P'` el predicado anterior. En las fórmulas anteriores, si *P* es una propiedad sobre los *streams*, entonces `P'` es `P` en las especificaciones dadas. Si en lugar de tener el predicado *IsSini*, que caracteriza a un único estado, disponemos de un predicado *Q* que caracteriza a múltiples estados, las especificaciones se expresan en forma análoga a como señalamos al final de la sección 3.3.2 (con `Always` al comienzo).

Las abreviaturas dadas en 2.3.2 para $\Diamond$ y $\Box$ pueden ahora ser demostradas como teoremas. A continuación formulamos los teoremas, cuyas demostraciones se incluyen en el apéndice B.2.2.

```
Theorem Equiv1 : ∀Sini∈S ∀P∈(SPath->Prop)
  (Posible Sini P) <-> (EX_Until Sini ( [_:SPath] True ) P).

Theorem Equiv2 : ∀Sini∈S ∀P∈(SPath->Prop)
  (Inevitable Sini P) <-> (FA_Until Sini ( [_:SPath] True ) P).

Theorem Equiv3 : ∀Sini∈S ∀P∈(SPath->Prop)
  (Always Sini P) <-> ~(Posible Sini ( [s:SPath] ~(P s) )).

Theorem Equiv4 : ∀Sini∈S ∀P∈(SPath->Prop)
  (SafePath Sini P) <-> ~(Inevitable Sini ( [s:SPath] ~(P s) )).
```

Los teoremas `Equiv1` y `Equiv2` se prueban por inducción, mientras que `Equiv3` y `Equiv4` requieren co-inducción y se demuestran en lógica clásica con el uso del principio del tercero excluido.

Asimismo, algunas de las propiedades previamente formuladas en 3.3.3 pueden ser probadas para la definición de los operadores temporales sobre *streams* de estados.

```
Theorem Mon_I : ∀x∈SPath ∀Pg,Pp∈(SPath->Prop)
  (ForAllS Pg x) -> (∀s∈SPath (Pg s)->(Pp s)) -> (ForAllS Pp x).

Theorem Conj : ∀x∈SPath ∀P1,P2∈(SPath->Prop)
  (ForAllS P1 x) -> (ForAllS P2 x) ->
    (ForAllS ( [s:SPath] (P1 s)/\(P2 s) ) x).

Theorem Mon_I_EX : ∀x∈SPath ∀Pg,Pp∈(SPath->Prop)
  (ExistsS Pg x) -> (∀s∈SPath (Pg s)->(Pp s)) -> (ExistsS Pp x).

Theorem OneStep_EX : ∀x∈SPath ∀P∈(SPath->Prop) (ExistsS P x) ->
  ∀s∈S (ExistsS P s^x).
```

Los cuatro teoremas se demuestran fácilmente. En los dos primeros la prueba es por co-inducción. Las demostraciones se incluyen en el apéndice B.2.2.

3.4.3 Operadores Temporales Cuantitativos

Las formalizaciones dadas en la sección previa de los operadores temporales pueden extenderse para caracterizar la versión general de la lógica TCTL, que permite razonamiento cuantitativo temporal, con su semántica de tiempo discreto.

Nuevamente, sea `S` el tipo de los estados de un sistema de tiempo real y `tr` una relación de transición entre estados de `S`. Asumimos además que la pertenencia de un valor temporal a un intervalo es representada por un predicado `bound:Instant->Prop`.

Definimos las trazas de estados generadas por `tr` como un predicado co-inductivo `isTrace_T` sobre secuencias infinitas (`SPath_T`) de pares (`State_T`) de estados de `S` y valores temporales. Cada elemento de estas trazas es un par que representa al estado del sistema en un punto dado y al valor del *reloj global* en dicho punto. Este reloj es transparente para los sistemas, no es reseteado nunca, y se utiliza para demarcar el tiempo de las ejecuciones. `IsTraceFrom_T` define a las trazas temporizadas que poseen comienzo en un estado inicial dado (para un determinado valor del reloj global).

```
Variable S     : Set.
Variable tr    : S -> Label -> S -> Set.
Variable bound : Instant -> Prop.

Definition State_T  := S * Instant.
Definition SPath_T  := (Stream State_T).

CoInductive isTrace_T : SPath_T -> Prop :=
  isTraceDisc : ∀x∈SPath_T ∀s1,s2∈S ∀l∈Label
    (tr s1 l s2) ->
    ~l=Tick ->
    (isTrace_T (s2,t)^x) ->
    (isTrace_T ( (s1,t)^(s2,t)^x ))
  |isTraceTick : ∀x∈SPath_T ∀s1,s2∈S ∀t∈Instant
    (tr s1 Tick s2) ->
    (isTrace_T (s2,(Inc t))^x) ->
    (isTrace_T ( (s1,t)^(s2,(Inc t))^x )).

Definition isTraceFrom_T := [Sini:State_T; x:SPath_T]
  Sini=(hd x) /\ (isTrace_T x).
```

isTraceDisc corresponde a una transición discreta, la cual no insume tiempo. *isTraceTick* a una transición temporal etiquetada con Tick. El paso del tiempo está representado por Inc, que incrementa el *reloj global* del sistema en un *tick*.

Al igual que en la sección previa, consideramos que las propiedades tienen como dominio a *streams* de estados, a fin de permitir caracterizar operadores temporales composicionales; pero donde estos estados son pares compuestos por un estado del sistema y el valor del reloj global (elementos de tipo State_T). Dado x∈SPath_T, las propiedades que sólo refieren al estado del sistema se expresan sobre el elemento Fst(hd x). Donde, la función Fst retorna el primer componente de un par (Snd retorna el segundo).

La versión de tiempo acotado del operador *until* (μ_I) se define como una extensión natural de la planteada en 3.4.2. $P\mu_I Q$ se formaliza como un predicado inductivo de la siguiente manera:

```
Inductive Until_bound [P,Q:SPath_T->Prop]: SPath_T -> Prop :=
  UntilFurther_bound : ∀s∈State_T ∀x∈SPath_T
    (P s^x) -> (Until_bound P Q x) ->
    (Until_bound P Q s^x)
  |UntilHere_bound : ∀s∈State_T ∀t∈Instant ∀x∈SPath_T
    (Q (s,t)^x) -> (bound t) ->
    (Until_bound P Q (s,t)^x).
```

Respecto a la definición no acotada de *until*, la prueba de satisfactibilidad de un predicado *Q* sobre un *stream* x está restringida al cumplimiento de bound sobre la componente temporal del primer estado del *stream*.

A partir de la definición anterior, las formalizaciones de los operadores $\exists\mu_I$ ($P\exists\mu_I Q$) y $\forall\mu_I$ ($P\forall\mu_I Q$), sobre trazas con estado de origen dado, se obtienen generalizando EX_Until y FA_Until sobre intervalos de tiempo especificados con bound.

```
Inductive EX_Until_bound [Sini:State_T; P,Q:SPath_T->Prop] : Prop :=
  ExUntil_bound : ∀x∈SPath_T (isTraceFrom_T Sini x) ->
    (Until_bound P Q x) ->
    (EX_Until_bound Sini P Q).

Definition FA_Until_bound := [Sini:State_T; P,Q:SPath_T->Prop]
  ∀x∈SPath_T (isTraceFrom_T Sini x) -> (Until_bound P Q x).
```

Al igual que en la sección previa definimos algunas de las abreviaturas más comúnmente usadas –introducidas en la sección 2.3.2– a través de ExistsS y ForAllS. $\exists\Diamond_I P$, $\forall\Diamond_I P$, $\exists\Box_I P$ y $\forall\Box_I P$ se obtienen a partir de Posible_T, Inevitable_T, SafePath_T y Always_T, respectivamente.

```
Inductive Posible_T [Sini:State_T; P:SPath_T->Prop] : Prop :=
  posible_T : ∀x∈SPath_T (isTraceFrom_T Sini x) ->
    (ExistsS ( [s:SPath_T] (bound (Snd (hd s))) /\ (P s) ) x) ->
    (PosibleS_T Sini P).

Definition Inevitable_T := [Sini:State_T; P:SPath_T->Prop]
  ∀x∈SPath_T (isTraceFrom_T Sini x) ->
    (ExistsS ( [s:SPath_T] (bound (Snd (hd s))) /\ (P s) ) x).

Inductive SafePath_T [Sini:State_T; P:SPath_T->Prop]: Prop :=
  safePath_T : ∀x∈SPath_T (isTraceFrom_T Sini x) ->
    (ForAllS ( [s:Path_T] (bound (Snd (hd s))) -> (P s) ) x) ->
    (SafePath_T Sini P).

Definition Always_T := [Sini:State_T; P:SPath_T->Prop]
  ∀x∈SPath_T (isTraceFrom_T Sini x) ->
    (ForAllS ( [s:SPath_T] (bound (Snd (hd s))) -> (P s) ) x).
```

Ejemplo 3.2. Sea *P* una propiedad básica sobre los estados del sistema e *IsSini* el predicado característico de un estado *Sini*. La fórmula $IsSini \Rightarrow \forall\Box_I P$ se especifica `(Always_T tr boundI (Sini,time0) P')`; donde, `tr` es la relación de transición del sistema, `boundI` es el predicado que corresponde a la pertenencia del valor del reloj global al intervalo *I* y `P'` es el predicado "`[x:SPath_T] (P (Fst (hd x)))`" –con `SPath_T` el tipo `(Stream S*Instant)`, `S` el tipo de los estados (`Sini∈S`) y `time0` el instante de tiempo inicial. Asimismo, una fórmula compuesta $IsSini \Rightarrow \forall\Diamond_I\exists\Diamond_J P$ puede especificarse como sigue: `(Inevitable_T tr boundI (Sini,time0) Q)`; donde, `Q` es el predicado "`[x:SPath_T] (Posible_T tr boundJ' (hd x) P')`", con `P'` el predicado anterior y `boundJ'` la actualización del predicado `boundJ` "`[t:Instant] (boundJ (plus_Ck t (Snd (hd x))))`" (`boundJ` es el predicado que corresponde a la pertenencia de un valor temporal al intervalo *J*). En las fórmulas anteriores el valor `time0` puede generalizarse a cualquier instante `t`, actualizando el predicado `boundI` de la misma manera que describimos para `boundJ` ("`[t':Instant] (boundI (plus_Ck t t'))`"). Si en lugar de tener el predicado *IsSini*, que caracteriza a un único estado, disponemos de un predicado *Q* que caracteriza a múltiples estados, las especificaciones se expresan en forma análoga a como señalamos al final de la sección 3.3.2 (con `Always_T` al comienzo y "`[t:Instant]True`" el predicado correspondiente a su intervalo acotado).

Nuevamente, las abreviaturas dadas en 2.3.2 pueden ser demostradas como teoremas, ahora para las versiones más generales de los operadores $\exists\mu_I$ y $\forall\mu_I$.

```
Theorem Equiv1_T : ∀Sini∈State_T ∀P∈(SPath_T->Prop)
  (Posible_T Sini P) <->
  (EX_Until_bound Sini ( [_:SPath_T] True ) P).

Theorem Equiv2_T : ∀Sini∈State_T ∀P∈(SPath_T->Prop)
  (Inevitable_T Sini P) <->
  (FA_Until_bound Sini ( [_:SPath_T] True ) P).

Theorem Equiv3_T : ∀Sini∈State_T ∀P∈(SPath_T->Prop)
  (Always_T Sini P) <->
  ~(Posible_T Sini ( [s:SPath_T] ~(P s) )).

Theorem Equiv4_T : ∀Sini∈State_T ∀P∈(SPath_T->Prop)
  (SafePath_T Sini P) <->
  ~(Inevitable_T Sini ( [s:SPath_T] ~(P s) )).
```

`Equiv3_T` y `Equiv4_T` se demuestran en lógica clásica, con el uso del principio del tercero excluido. Las pruebas para estos dos teoremas son por co-inducción, mientras que `Equiv1_T` y `Equiv2_T` se demuestran por inducción. Las demostraciones se incluyen en el apéndice B.3.2.

Nota 3.5. Las formalizaciones dadas en la sección 3.4.2 resultan de las presentadas en esta sección para trazas temporizadas, asumiendo que los intervalos son no acotados; es decir, cada predicado `bound:Instant->Prop` es `True` para todo `t∈Instant`.

Capítulo 4

Caso de Estudio: Control de un Paso a Nivel de Tren

En este capítulo consideramos un caso de estudio: *el control de un paso a nivel de tren* ("*the railroad crossing example*"). Especificamos el sistema en *Coq* en base a las formalizaciones introducidas en el capítulo 3. Analizamos luego la demostración de *invariantes*, incluyendo la propiedad de seguridad esencial del sistema, y verificamos la divergencia del tiempo en las ejecuciones. Posteriormente generalizamos la especificación, parametrizando las constantes del problema y definimos restricciones entre estos parámetros y los relojes del sistema a fin de preservar las propiedades analizadas.

4.1 Descripción del Problema

Numerosos trabajos consideran a este problema como "*benchmark*" para analizar diferentes técnicas de especificación y herramientas de análisis. Entre otros, [HNS+92, CHR92, Sha93, HJL93, HK94, DY95, AB96, BMS+97]. Nosotros tomamos la especificación del problema de [AD94] (basada en [LS85]).

El sistema consiste de tres procesos paralelos: un tren (*train*), un controlador (*controller*) y una barrera (*gate*). Cada proceso puede ser modelado por un grafo temporizado tal como lo ilustra la figura 4.1.

La variable de control *st* del tren varía sobre tres locaciones: *st = Far* si el tren está lejos de cruzar el paso a nivel (no hay trenes cerca del cruce); *st = Near* si el tren está próximo a cruzar; y, *st = Inside* si está cruzando. *Far* es la locación del estado inicial. Cuando el tren está próximo a cruzar (es decir, se mueve de la locación *Far* a *Near*), él envía una señal *Approach* al controlador, avisando sobre la proximidad de un tren. Esto ocurre *n* unidades de tiempo antes, con *n > kt1*, que el tren este cruzando el paso a nivel. Cuando el tren termina de cruzar (es decir, se mueve de la locación *Inside* a *Far*) envía una señal *Exit* al controlador, indicando el alejamiento del tren del paso a nivel. Esto ocurre antes que *kt2* unidades de tiempo han pasado desde la señal *Approach*.

La variable de control *sc* del controlador varía sobre cuatro locaciones: *sc = Sc1* si el controlador está esperando que el tren arribe; *sc = Sc2* si la señal *Approach* ha sido

recibida; *sc* = *Sc3* si el controlador está esperando la señal *Exit*; y, *sc* = *Sc4* si la señal *Exit* ha sido recibida. *Sc1* es la locación del estado inicial. Cuando la señal *Approach* es recibida (es decir, el controlador se mueve de *Sc1* a *Sc2*), el controlador envía a la barrera la señal *Lower*, exactamente *kc1* unidades de tiempo después, indicando que ésta debe bajar. Cuando *Exit* es recibida (es decir, el controlador se mueve de *Sc3* a *Sc4*), antes de *kc2* unidades de tiempo el controlador envía a la barrera la señal *Raise*, para que la barrera comience a subir.

La variable de control *sg* de la barrera varía sobre cuatro locaciones: *sg* = *Open* si la barrera está levantada y esperando la señal *Lower*; *sg* = *Lowering* si la señal *Lower* ha sido recibida; *sg* = *Closed* si la barrera está baja y esperando la señal *Raise*; y, *sg* = *Raising* si la señal *Raise* ha sido recibida. *Open* es la locación del estado inicial. Cuando la señal *Lower* es recibida (es decir, la barrera se mueve de *Open* a *Lowering*), la barrera está baja antes de *kg1* unidades de tiempo y cuando *Raise* llega, antes de *kg3* y por lo menos *kg2* unidades de tiempo después la barrera está levantada nuevamente.

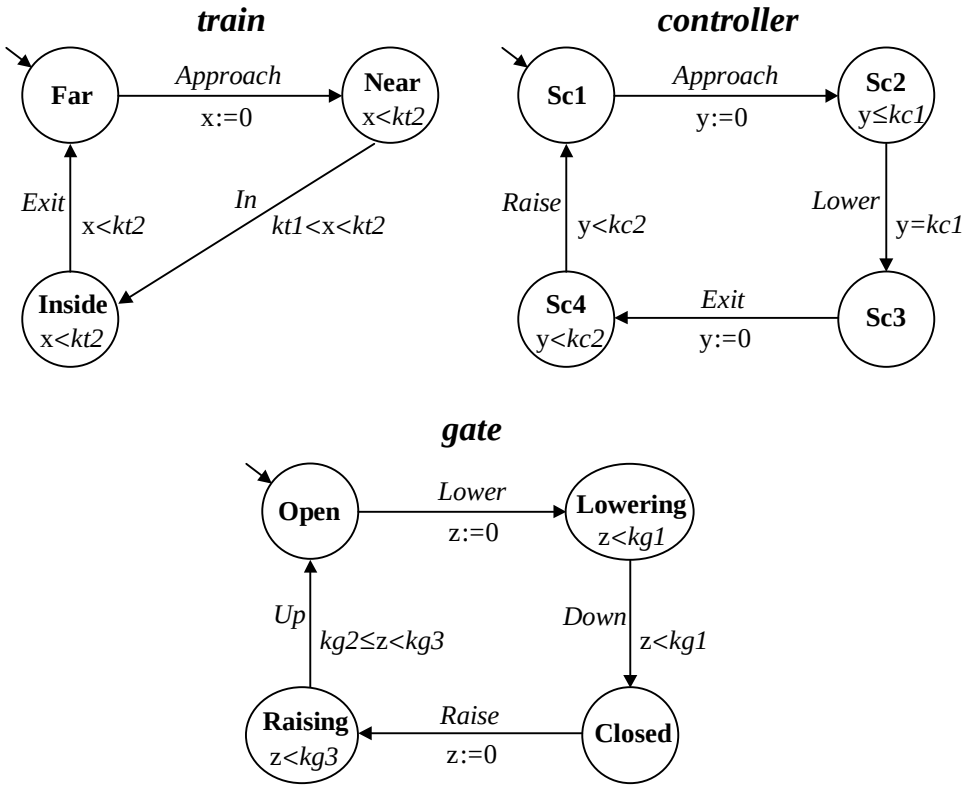

Fig. 4.1: Grafos temporizados del sistema "control de un paso a nivel de tren"

En [AD94] los valores de las constantes son los siguientes: *kt1* = 2, *kt2* = 5, *kc1* = 1, *kc2* = 1, *kg1* = 1, *kg2* = 1 y *kg3* = 2. En este trabajo asumimos dichos valores de base

multiplicados por 4. Esto es $M = 4 > |X|$, en virtud de las consideraciones de la sección 2.5.3, con X el conjunto de los relojes del sistema compuesto.

4.2 Especificación del Sistema en Coq

En esta sección especificamos en *Coq* el sistema descripto en la sección previa, siguiendo los lineamientos generales dados en la sección 3.2. Llamaremos TCG al sistema compuesto (*train* || *controller* || *gate*).

El conjunto de las etiquetas del sistema está integrado por siete componentes correspondientes a transiciones discretas y una que rotula las transiciones temporales. En el primer grupo están `In`, `Down`, `Up`, `Approach`, `Exit`, `Lower` y `Raise`. Las tres primeras no son de sincronización, mientras que las cuatro últimas si lo son (`Approach` y `Exit` entre el tren y el controlador; `Lower` y `Raise` entre el controlador y la barrera). La etiqueta que representa las transiciones temporales es `Tick`, que es también de sincronización.

```
Inductive Label : Set := Approach : Label | In : Label | Exit: Label |
  Lower : Label | Raise : Label | Up : Label | Down : Label |
  Tick : Label.
```

Las locaciones del tren (`ST`), del controlador (`SC`) y la barrera (`SG`) se definen por los siguientes tipos enumerados,

```
Inductive ST : Set := Far : ST | Near : ST | Inside : ST.
Inductive SC : Set := Sc1 : SC | Sc2 : SC | Sc3 : SC | Sc4 : SC.
Inductive SG : Set := Open : SG | Lowering : SG | Closed : SG |
  Raising: SG.
```

Cada grafo temporizado del sistema TCG posee un reloj. Definimos el tipo de los estados compuestos por un único reloj como el siguiente tipo paramétrico,

```
Definition S_Ck := [State:Set] State * Clock.
```

Los estados del tren tienen tipo `(S_Ck ST)`, los del controlador `(S_Ck SC)` y los de la barrera `(S_Ck SG)`.

Las constantes del sistema TCG se especifican con las siguientes declaraciones:

```
Definition kt1 : Instant := (8).
Definition kt2 : Instant := (20).
Definition kc1 : Instant := (4).
Definition kc2 : Instant := (4).
```

```
Definition kg1 : Instant := (4).
Definition kg2 : Instant := (4).
Definition kg3 : Instant := (8).
```

Los invariantes de las locaciones son predicados sobre los estados que limitan la permanencia del sistema en las locaciones. Para el tren, el predicado es el siguiente,

```
Inductive InvT: (S_Ck ST) -> Prop :=
  Ifar     : ∀x∈Clock (InvT (Far,x))
  |Inear   : ∀x∈Clock (lt_Ck x kt2) -> (InvT (Near,x))
  |Iinside : ∀x∈Clock (lt_Ck x kt2) -> (InvT (Inside,x)).
```

El invariante de la locación `Far` es `True` (el sistema puede permanecer en dicha locación sin restricciones) y el de las locaciones `Near` e `Inside`, `(lt_Ck x kt2)`. Para el controlador y la barrera, los invariantes se definen como sigue

```
Inductive InvC: (S_Ck SC) -> Prop :=
  Isc1  : ∀y∈Clock (InvC (Sc1,y))
  |Isc2 : ∀y∈Clock (le_Ck y kc1) -> (InvC (Sc2,y))
  |Isc3 : ∀y∈Clock (InvC (Sc3,y))
  |Isc4 : ∀y∈Clock (lt_Ck y kc2) -> (InvC (Sc4,y)).

Inductive InvG: (S_Ck SG) -> Prop :=
  Iopen      : ∀z∈Clock (InvG (Open,z))
  |Ilowering : ∀z∈Clock (lt_Ck z kg1) -> (InvG (Lowering,z))
  |Iclosed   : ∀z∈Clock (InvG (Closed,z))
  |Iraising  : ∀z∈Clock (lt_Ck z kg3) -> (InvG (Raising,z)).
```

Las transiciones son relaciones entre estados e involucran etiquetas. La especificación de éstas para cada uno de los tres sistemas componentes sigue la estructura descripta en la sección 3.2, con una pequeña modificación: no se exige el cumplimiento del invariante de las locaciones para el estado destino de una transición discreta. Esta simplificación es válida debido a que las transiciones discretas para cada uno de estos tres sistemas preservan el cumplimiento del invariante de las locaciones correspondiente. Formalmente esta simplificación se justifica probando que: $SiniT \Rightarrow \forall\Box InvT$, $SiniC \Rightarrow \forall\Box InvC$ y $SiniG \Rightarrow \forall\Box InvG$; donde, *SiniT*, *SiniC* y *SiniG* son los estados iniciales del tren, el controlador y la barrea, respectivamente. Las demostraciones pueden consultarse en el apéndice C.2.4. Observemos que tanto las transiciones discretas como las temporales asumen que el estado de origen es *válido*, de acuerdo a lo establecido en la sección 3.2.

La figura 4.2 presenta los sistemas de transiciones para los tres componentes, según la notación introducida en el capítulo 2.

$$\frac{}{\langle Far, x\rangle \xrightarrow{Approach}_T \langle Near, 0\rangle} \qquad \frac{kt1 < x < kt2}{\langle Near, x\rangle \xrightarrow{In}_T \langle Inside, x\rangle}$$

$$\frac{x < kt2}{\langle Inside, x\rangle \xrightarrow{Exit}_T \langle Far, x\rangle} \qquad \frac{InvT(st, x + tick)}{\langle st, x\rangle \xrightarrow{Tick}_T \langle st, x + tick\rangle}$$

$$\frac{}{\langle Sc1, y\rangle \xrightarrow{Approaxh}_C \langle Sc2, 0\rangle} \qquad \frac{y = kc1}{\langle Sc2, y\rangle \xrightarrow{Lower}_C \langle Sc3, y\rangle}$$

$$\frac{}{\langle Sc3, y\rangle \xrightarrow{Exit}_C \langle Sc4, 0\rangle} \qquad \frac{y < kc2}{\langle Sc4, y\rangle \xrightarrow{Raise}_C \langle Sc1, y\rangle}$$

$$\frac{InvC(sc, y + tick)}{\langle sc, y\rangle \xrightarrow{Tick}_C \langle sc, y + tick\rangle}$$

$$\frac{}{\langle Open, z\rangle \xrightarrow{Lower}_G \langle Lowering, 0\rangle} \qquad \frac{z < kg1}{\langle Lowering, z\rangle \xrightarrow{Down}_G \langle Closed, z\rangle}$$

$$\frac{}{\langle Closed, z\rangle \xrightarrow{Raise}_G \langle Raising, 0\rangle} \qquad \frac{kg2 \le z < kg3}{\langle Raising, z\rangle \xrightarrow{Up}_G \langle Open, z\rangle}$$

$$\frac{InvG(sg, z + tick)}{\langle sg, z\rangle \xrightarrow{Tick}_G \langle sg, z + tick\rangle}$$

Fig. 4.2: Transiciones del tren, el controlador y la barrera

En *Coq*, las transiciones discretas y temporales del tren se definen por la siguiente relación inductiva (`TrT` representa la relación $\rightarrow_T$),

```
Inductive TrT : (S_Ck ST) -> Label -> (S_Ck ST) -> Prop :=
  ttApproach : ∀x∈Clock (TrT (Far,x) Approach (Near,Reset))
  |ttIn      : ∀x∈Clock (gt_Ck x kt1) -> (lt_Ck x kt2) ->
               (TrT (Near,x) In (Inside,x))
  |ttExit    : ∀x∈Clock (lt_Ck x kt2) -> (TrT (Inside,x) Exit (Far,x))
  |ttTick    : ∀x∈Clock ∀s∈ST (InvT (s,(Inc x))) ->
               (TrT (s,x) Tick (s,(Inc x))).
```

ttApproach permite construir transiciones de la locación `Far` a la locación `Near` para cualquier valor del reloj (la condición de activación es `True`); *ttIn* de `Near` a `Inside` para los valores del reloj `x` que cumplen la conjunción de las condiciones `(gt_Ck x kt1)` y `(lt_Ck x kt2)`; *ttExit* de `Inside` a `Far` para los valores del reloj `x`

que verifican la restricción (`lt_Ck x kt2`); finalmente, *`ttTick`* de una locación a ella misma incrementando el reloj en un *tick*, si el invariante se satisface para el nuevo valor.

Las transiciones para el controlador y la barrera se definen de manera similar (`TrC` representa a $\rightarrow_C$ y `TrG` a $\rightarrow_G$).

```
Inductive TrC : (S_Ck SC) -> Label -> (S_Ck SC) -> Prop :=
  tcApproach : ∀y∈Clock (TrC (Sc1,y) Approach (Sc2,Reset))
  |tcLower   : ∀y∈Clock (eq_Ck y kc1) -> (TrC (Sc2,y) Lower (Sc3,y))
  |tcExit    : ∀y∈Clock (TrC (Sc3,y) Exit (Sc4,Reset))
  |tcRaise   : ∀y∈Clock (lt_Ck y kc2) -> (TrC (Sc4,y) Raise (Sc1,y))
  |tcTick    : ∀y∈Clock ∀s∈SC (InvC (s,(Inc y))) ->
               (TrC (s,y) Tick (s,(Inc y))).

Inductive TrG : (S_Ck SG) -> Label -> (S_Ck SG) -> Set :=
  tgLower  : ∀z∈Clock (TrG (Open,z) Lower (Lowering,Reset))
  |tgDown  : ∀z∈Clock (lt_Ck z kg1) ->
             (TrG (Lowering,z) Down (Closed,z))
  |tgRaise : ∀z∈Clock (TrG (Closed,z) Raise (Raising,Reset))
  |tgUp    : ∀z∈Clock (ge_Ck z kg2) -> (lt_Ck z kg3) ->
             (TrG (Raising,z) Up (Open,z))
  | tgTick : ∀z∈Clock ∀s∈SG (InvG (s,(Inc z))) ->
             (TrG (s,z) Tick (s,(Inc z))).
```

Los estados del sistema compuesto TCG son una séxtupla, que definimos por el constructor de pares de tipos `*`.

```
Definition St_Global := (S_Ck ST) * (S_Ck SC) * (S_Ck SG).
```

Un estado de `St_Global` consta de las locaciones y los valores de los relojes para cada uno de los tres sistemas componentes (esto equivale al producto de las locaciones y la unión de los relojes en el modelo de base).

Notación. Un elemento de tipo `St_Global` es un par $(st,(sc,sg))$ ($*$ asocia a derecha). Sin embargo, para simplificar la notación, escribiremos la terna (st,sc,sg) por $(st,(sc,sg))$.

Las transiciones para grafos compuestos las especificamos teniendo en cuenta la definición de la sección 2.2. La figura 4.3 exhibe las transiciones del sistema TCG, según la notación del capítulo 2.

$$\frac{\langle st, x\rangle \xrightarrow{In}_T \langle st', x'\rangle}{\langle\langle st, sc, sg\rangle, \langle x, y, z\rangle\rangle \xrightarrow{In}_{TCG} \langle\langle st', sc, sg\rangle, \langle x', y, z\rangle\rangle}$$

$$\frac{\langle sg, z\rangle \xrightarrow{L}_G \langle sg', z'\rangle}{\langle\langle st, sc, sg\rangle, \langle x, y, z\rangle\rangle \xrightarrow{L}_{TCG} \langle\langle st, sc, sg'\rangle, \langle x, y, z'\rangle\rangle} \quad L \in \{Down, Up\}$$

$$\frac{\langle st, x\rangle \xrightarrow{L}_T \langle st', x'\rangle \qquad \langle sc, y\rangle \xrightarrow{L}_C \langle sc', y'\rangle}{\langle\langle st, sc, sg\rangle, \langle x, y, z\rangle\rangle \xrightarrow{L}_{TCG} \langle\langle st', sc', sg\rangle, \langle x', y', z\rangle\rangle} \quad L \in \{Approach, Exit\}$$

$$\frac{\langle sc, y\rangle \xrightarrow{L}_C \langle sc', y'\rangle \qquad \langle sg, z\rangle \xrightarrow{L}_G \langle sg', z'\rangle}{\langle\langle st, sc, sg\rangle, \langle x, y, z\rangle\rangle \xrightarrow{L}_{TCG} \langle\langle st, sc', sg'\rangle, \langle x, y', z'\rangle\rangle} \quad L \in \{Lower, Raise\}$$

$$\frac{\langle st, x\rangle \xrightarrow{Tick}_T \langle st', x'\rangle \qquad \langle sc, y\rangle \xrightarrow{Tick}_C \langle sc', y'\rangle \qquad \langle sg, z\rangle \xrightarrow{Tick}_G \langle sg', z'\rangle}{\langle\langle st, sc, sg\rangle, \langle x, y, z\rangle\rangle \xrightarrow{Tick}_{TCG} \langle\langle st', sc', sg'\rangle, \langle x', y', z'\rangle\rangle}$$

Fig. 4.3: Transiciones del sistema compuesto TCG.

En *Coq*, la relación de transición entre estados de `St_Global` puede especificarse como sigue (`TrGlobal` representa la relación $\rightarrow_{TCG}$):

```
Inductive TrGlobal : StGlobal -> Label -> StGlobal -> Prop :=
  |tGl_In : ∀st1,st2∈(S_Ck ST) (TrT st1 In st2) ->
    ∀sc∈(S_Ck SC) ∀sg∈(S_Ck SG)
    (TrGlobal (st1,sc,sg) In (st2,sc,sg))
  |tGl_Down : ∀sg1,sg2∈(S_Ck SG) (TrG sg1 Down sg2) ->
    ∀st∈(S_Ck ST) ∀sc∈(S_Ck SC)
    (TrGlobal (st,sc,sg1) Down (st,sc,sg2))
  |tGl_Up : ∀sg1,sg2∀∈(S_Ck SG) (TrG sg1 Up sg2) ->
    ∀st∈(S_Ck ST) ∀sc∈(S_Ck SC)
    (TrGlobal (st,sc,sg1) Up (st,sc,sg2))
  |tGl_Approach : ∀st1,st2∈(S_Ck ST) ∀sc1,sc2∈(S_Ck SC)
    (TrT st1 Approach st2) -> (TrC sc1 Approach sc2) -> ∀sg∈(S_Ck SG)
    (TrGlobal (st1,sc1,sg) Approach (st2,sc2,sg))
  |tGl_Exit : ∀st1,st2∈(S_Ck ST) ∀sc1,sc2∈(S_Ck SC)
    (TrT st1 Exit st2) -> (TrC sc1 Exit sc2) -> ∀sg∈(S_Ck SG)
    (TrGlobal (st1,sc1,sg) Exit (st2,sc2,sg))
```

```
|tGl_Lower : ∀sc1,sc2∈(S_Ck SC) (sg1,sg2:(S_Ck SG))
   (TrC sc1 Lower sc2) -> (TrG sg1 Lower sg2) -> ∀st∈(S_Ck ST)
   (TrGlobal (st,sc1,sg1) Lower (st,sc2,sg2))
|tGl_Raise : ∀sc1,sc2∈(S_Ck SC) ∀sg1,sg2∈(S_Ck SG)
   (TrC sc1 Raise sc2) -> (TrG sg1 Raise sg2) -> ∀st∈(S_Ck ST)
   (TrGlobal (st,sc1,sg1) Raise (st,sc2,sg2))
|tGl_Tick : ∀st1,st2∈(S_Ck ST) ∀sc1,sc2∈(S_Ck SC)
   ∀sg1,sg2∈(S_Ck SG)
   (TrT st1 Tick st2) -> (TrC sc1 Tick sc2) -> (TrG sg1 Tick sg2) ->
   (TrGlobal (st1,sc1,sg1) Tick (st2,sc2,sg2)).
```

tGl_In corresponde a la transición discreta del tren rotulada con In, en el sistema compuesto TCG; *tGl_Down* y *tGl_Up* a las de la barrera, rotuladas con Down y Up. Estas tres no son de sincronización, permiten modificar sólo la componente del estado que corresponde al grafo al cual pertenece la transición básica. *tGl_Approach* y *tGl_Exit* corresponden a las transiciones discretas que sincronizan el tren y el controlador con Approach y Exit. *tGl_Lower* y *tGl_Raise* a las que sincronizan el controlador y la barrera con Lower y Raise. *tGl_Tick* representa a las transiciones temporales para el sistema TCG. La misma se especifica como de sincronización de las correspondientes a cada uno de los tres sistemas componentes. Esto es posible debido a que todos los incrementos temporales suman una misma cantidad (un *tick*). La condición asociada es la conjunción de los invariantes correspondientes a cada uno de los sistemas básicos. Luego, bajo la semántica de tiempo discreto asumida para la representación y el análisis de grafos temporizados, las transiciones temporales pueden ser tratadas igual que las discretas de sincronización.

El estado inicial para el sistema TCG se especifica a partir de los estados iniciales de cada uno de los grafos componentes. Esto es,

```
Definition SiniT   :(S_Ck ST)  := (Far,Ini_Ck).
Definition SiniC   :(S_Ck SC)  := (Sc1,Ini_Ck).
Definition SiniG   :(S_Ck SG)  := (Open,Ini_Ck).

Definition SiniTCG :StGlobal   := (SiniT,SiniC,SiniG).
```

Observemos que el estado inicial de cada grafo temporizado satisface el *invariante* de las locaciones correspondiente (se cumple el requisito formulado en la sección 3.2). Esto es, valen InvT(SiniT), InvC(SiniC), InvG(SiniG) y obviamente el invariante InvTCG del sistema TCG para el estado compuesto SiniTCG. Donde,

```
Definition InvTCG := [s:StGlobal] Cases s of (st_x,sc_y,sg_z) =>
   ((InvT st_x) /\ (InvC sc_y) /\ (InvG sg_z))  end.
```

4.3 Análisis del Sistema TCG

En esta parte del trabajo estamos interesados en analizar la demostración de dos clases típicas de propiedades: *safety* y *liveness*. Las primeras se usan para especificar que "nada malo ocurrirá durante cierto tiempo". Las mismas se formulan, generalmente, mediante el operador $\Box_I$. Las propiedades de *liveness* permiten especificar que "algo bueno ocurrirá dentro de una intervalo de tiempo especificado". Estas se formalizan, generalmente, mediante el operador $\Diamond_I$. Una definición formal de propiedades de *safety* y *liveness* fue dada por Alpern y Shneider en [AS85], donde ellos muestran que cualquier propiedad de una traza (ejecución) puede ser expresada como una conjunción de una propiedad de *safety* y una de *liveness*.

Un requerimiento de seguridad asociado al sistema TCG es la propiedad de *safety* "siempre que el tren está cruzando el paso a nivel, la barrera se encuentra baja". Observemos que aunque esta propiedad es puramente *cualitativa*, la misma no vale si eliminamos las restricciones temporales o cambiamos ciertos valores de las mismas. Una propiedad esencial de todo sistema es la propiedad de *liveness* "*sistema bien temporizado*", la cual asegura que el tiempo diverge en todas las ejecuciones del sistema (ver sección 2.1.2). En las dos próximas subsecciones analizaremos la demostración de propiedades *invariantes*, incluyendo la propiedad de *safety* previamente descripta. La propiedad "*sistema bien temporizado*", para el sistema TCG, será tratada en la sección 4.3.3.

4.3.1 Demostración de Invariantes

Para verificar una propiedad del tipo $Q \Rightarrow \forall\Box_I P$ sobre un grafo temporizado G podemos proceder según analizamos en la sección 2.5.3. Esto equivale, según la formalización de G y la definición con tipos inductivos del operador $\forall\Box$, a hacer inducción sobre el conjunto de estados alcanzables y la definición de las transiciones. En esta sección analizaremos la prueba de invariantes para el sistema TCG, los cuales nos permitirán adquirir un conocimiento mayor del sistema, a fin de analizar otras propiedades más complejas, por ejemplo la de "*sistema bien temporizado*". Para nuestro caso de estudio el intervalo I no será significativo, esto es $I=[0, \infty)$ –usaremos la versión no acotada de $\forall\Box$. Adoptaremos la formalización con tipos inductivos del operador $\forall\Box$ dada en 3.3.2, en vez de la correspondiente con tipos co-inductivos presentada en 3.4.2 (aunque los mismos resultados se siguen de ambas representaciones).

Invariantes

Seguidamente describimos algunas de las propiedades invariantes analizadas del sistema TCG. Las mismas fueron demostradas con el asistente de pruebas *Coq* y la utilización de algunas tácticas especialmente definidas para este caso de estudio.

En las demostraciones de las propiedades hacemos uso de una de las principales ventajas de los métodos deductivos, la *composicionalidad* de las pruebas. Esto es, algunos invariantes (y teoremas en general) serán utilizados en las demostraciones de otros invariantes u otra clase de propiedades.

La especificación de una propiedad invariante Inv_i del sistema TCG corresponde a la fórmula $Init \Rightarrow \forall\Box Inv_i$. Donde *Init* es el predicado que caracteriza al estado inicial del sistema: `SiniTCG`.

Adoptaremos la siguiente abreviatura para el operador `ForAll` sobre propiedades del sistema TCG:

```
Definition ForAll_TCG: StGlobal -> (StGlobal->Prop) -> Prop :=
  (ForAll TrGlobal).
```

Dado un estado *Sini* y una propiedad Inv_i, `ForAll_TCG` especifica que todos los estados (de tipo `StGlobal`) alcanzables desde el estado *Sini* (con transiciones de tipo `TrGlobal`) satisfacen Inv_i. Luego, la demostración de una propiedad

$$Init \Rightarrow \forall\Box Inv_i$$

corresponderá a la prueba de un lema en *Coq*, que puede enunciarse como sigue:

```
Lemma lema_Invi : (ForAll_TCG SiniTCG Invi).
```

A continuación enunciamos los invariantes analizados del sistema TCG, las demostraciones son omitidas y pueden consultarse en el apéndice C.2.4.

- *Invariante 1*.

```
Definition Inv1 := [s:StGlobal] Cases s of ((st,_),(sc,_),(sg,_)) =>
  st=Near \/ st=Inside -> sc=Sc3 -> sg=Closed \/ sg=Lowering end.
```

Esto es, siempre se cumple que si el tren está próximo a cruzar (`st=Near`) o está cruzando (`st=Inside`) el paso a nivel, y el controlador está esperando la señal *exit* (`sc=Sc3`), la barrera se encuentra baja (`sg=Closed`) o está descendiendo (`sg=Lowering`).

- *Invariante 2*.

```
Definition Inv2 := [s:StGlobal] Cases s of ((st,_),(sc,_),(sg,_)) =>
  st=Far -> sg=Open \/ sg=Raising -> sc=Sc1 end.
```

Siempre se cumple que: si el tren está lejos –no hay un tren próximo ni cruzando– (`st=Far`) y la barrera está arriba (`sg=Open`) o levantándose (`sg=Raising`), entonces el controlador permanece a la espera de la señal *Approach* de un nuevo tren (`sc=Sc1`).

- *Invariante 3*.

```
Definition Inv₃ := [s:StGlobal] Cases s of ((st,x),(_,y),_) =>
   st=Near \/ st=Inside  -> (eq_Ck x y) end.
```

Siempre se cumple que: si el tren está próximo a cruzar (`st=Near`) o está cruzando (`st=Inside`) el paso a nivel, los valores de los relojes de ambos sistemas coinciden (ambos relojes son inicializados al mismo tiempo cuando el tren y el controlador se comunican a través de la señal *Approach*). Esta propiedad se relaciona más con la formalización del problema que con el problema en sí. La misma es utilizada para demostrar otras propiedades más relevantes del sistema.

- *Invariante 4*.

```
Definition Inv₄ := [s:StGlobal] Cases s of ((st,_),(sc,_),_) =>
   st=Near  -> sc=Sc2 \/ sc=Sc3  end.
```

Siempre se cumple que: si el tren está próximo a cruzar el paso a nivel (`st=Near`) entonces el controlador ha recibido la señal *Approach*, aunque no ha enviado aún la señal *lower* (`sc=Sc2`), o bien si la envió y está esperando la señal *exit* del tren (`sc=Sc3`).

- *Invariante 5*.

```
Definition Inv₅ := [s:StGlobal] Cases s of ((st,_),(sc,y),(_,z)) =>
   st=Near  -> sc=Sc3  -> (eq_Ck y (plus_Ck z kc1)) end.
```

Siempre se cumple que: si el tren está próximo a cruzar el paso a nivel (`st=Near`) y el controlador está esperando la señal *exit* del tren (`sc=Sc3`), el tiempo transcurrido desde que la señal *Approach* fue procesada por el controlador excede en `kc1` unidades al tiempo que pasó desde que la barrera comenzó a bajar. Esto se observa intuitivamente a partir de la transición que sincroniza el controlador y la barrera con *lower*.

- *Invariante 6*.

```
Definition Inv₆ := [s:StGlobal] Cases s of ((st,x),(sc,_),_) =>
   st=Near/\(gt_Ck x kc1) \/ st=Inside  -> sc=Sc3 end.
```

Siempre se cumple que: si el tren está próximo a cruzar el paso a nivel (`st=Near`) y el tiempo transcurrido en ese estado (desde el origen de la señal *Approach*) superó ya `kc1`

unidades, o bien el tren está cruzando el paso a nivel (`st=Inside`), entonces el controlador se encuentra esperando la señal *exit* y ha enviado la señal *lower* a la barrera (`sc=Sc3`).

- *Invariante 7*.

```
Definition Inv7 := [s:StGlobal] Cases s of ((st,_),(sc,_),(sg,_)) =>
   st=Near \/ st=Inside -> sc=Sc3 -> sg=Lowering \/ sg=Closed end.
```

Siempre se cumple que: si el tren está próximo a cruzar (`st=Near`) o está cruzando el paso a nivel (`st=Inside`), y el controlador está esperando la señal *exit* y ha enviado ya la señal *lower* a la barrera, entonces ésta se encuentra descendiendo (`sg=Lowering`) o está baja (`sg=Closed`).

- *Invariante 8*.

```
Definition Inv8 := [s:StGlobal] Cases s of ((st,x),(sc,_),(sg,_)) =>
   st=Near/\(gt_Ck x kt1) \/ st=Inside -> sg=Closed end.
```

Siempre se cumple que: si el tren está próximo a cruzar el paso a nivel (`st=Near`) y el tiempo transcurrido en ese estado (desde el origen de la señal *Approach*) superó ya `kt1` unidades, o bien el tren está cruzando el paso a nivel (`st=Inside`), entonces la barrera se encuentra baja (`sg=Closed`). Este invariante asegura la satisfacción de la propiedad de seguridad más importante del sistema.

- *Invariante 9*.

```
Definition Inv9 := [s:StGlobal] Cases s of ((st,x),(_,y),(_,z)) =>
   st=Near/\(gt_Ck x kc1) \/ st=Inside -> (eq_Ck y (plus_Ck z kc1)) end.
```

Siempre se cumple que: si el tren está próximo a cruzar el paso a nivel (`st=Near`) y el tiempo transcurrido en ese estado (desde el origen de la señal *Approach*) superó ya `kc1` unidades, o bien el tren está cruzando el paso a nivel (`st=Inside`), entonces el tiempo transcurrido desde que la señal *Approach* fue procesada por el controlador excede en `kc1` unidades al tiempo que pasó desde que la barrera comenzó a bajar.

- *Invariante 10*.

```
Definition Inv10 := [s:StGlobal] Cases s of (_,(sc,_),(sg,_)) =>
   sc=Sc1 \/ sc=Sc2 -> sg=Open \/ sg=Raising end.
```

Siempre se cumple que: si el controlador está esperando la señal *Approach* del tren (`sc=Sc1`) o ya la recibió pero aún no envió la señal *lower* a la barrea (`sc=Sc2`), entonces ésta se encuentra alta (`sg=Open`) o está levantándose (`sg=Raising`).

- *Invariante 11*.

```
Definition Inv11 := [s:StGlobal] Cases s of (_,(sc,_),(sg,_)) =>
   sg=Lowering \/ sg=Closed  ->  sc=Sc3 \/ sc=Sc4 end.
```

Esta propiedad expresa básicamente lo mismo que el invariante Inv_{10} (es en cierto sentido su contra-recíproco). La misma se demuestra a través del teorema `Mon_I` (ver sección 3.3.3), que permite probar invariantes a partir de otros invariantes cuando existe una relación de consecuencia entre dichas fórmulas.

- *Invariante 12*.

```
Definition Inv12 := [s:StGlobal] Cases s of (_,(sc,y),(_,z)) =>
   sc=Sc2  ->  (ge_Ck z y) end.
```

Siempre se cumple que: si el controlador recibió la señal *Approach* pero aún no envió la señal *lower* a la barrea (`sc=Sc2`), el tiempo transcurrido desde que la barrera comenzó a subir la última vez es por lo menos el tiempo que pasó desde que el controlador proceso la señal *Approach*.

- *Invariante 13*.

```
Definition Inv13 := [s:StGlobal] Cases s of ((st,_),(sc,_),_) =>
   sc=Sc2  ->  st=Near end.
```

Siempre se cumple que: si el controlador recibió la señal *Approach* pero aún no envió la señal *lower* a la barrea (`sc=Sc2`), entonces el tren se encuentra en el estado próximo a cruzar el paso a nivel (`sc=Near`).

- *Invariante 14*.

```
Definition Inv14 := [s:StGlobal] Cases s of ((st,_),(sc,_),(sg,_)) =>
   sc=Sc4  ->  st=Far /\ sg=Closed end.
```

Siempre se cumple que: si el controlador recibió la señal *exit* del tren pero aún no envió la señal *raise* a la barrea (`sc=Sc4`), entonces el tren se encuentra lejos –no hay un tren próximo ni cruzando– y la barrera está baja.

- *Invariante 15*.

```
Definition InvSc3 := [s:StGlobal] Cases s of (_,(sc,y),_) =>
   sc=Sc3 -> (ge_Ck y kc1) end.
```

Siempre se cumple que: si el controlador está esperando la señal *exit*, el tiempo transcurrido desde el comienzo de la aproximación del tren al paso a nivel (origen de la señal *Approach*) es por lo menos `kc1` unidades.

- *Invariante 16*.

```
Definition InvInside := [s:StGlobal] Cases s of ((st,x),_) =>
   st=Inside -> (gt_Ck x kt1) end.
```

Siempre se cumple que: si el tren está cruzando el paso a nivel, el tiempo transcurrido desde que comenzó su aproximación al cruce (origen de la señal *Approach*) excede `kt1` unidades.

Comentarios

Aunque no usamos la versión acotada del operador $\forall\square$, muchas de las propiedades invariantes previas expresan restricciones cuantitativas de tiempo. Por ejemplo, para indicar el tiempo transcurrido desde un evento (una señal) –invariantes 15 y 16– o el tiempo de separación entre dos eventos –invariantes 5, 9 y 12. Otros invariantes reflejan propiedades básicamente cualitativas que relacionan las locaciones de los estados posibles –invariantes 1, 2, 4, 7, 10, 11, 13 y 14. Finalmente, existen propiedades que son al mismo tiempo cualitativas y cuantitativas, según la clasificación previa. Por ejemplo, los invariantes 6 y 8.

Algunas Tácticas (Simples) Útiles

Una táctica de prueba en *Coq* permite abreviar un esquema de demostración. Las tácticas implementan razonamiento "hacia atrás", es decir, la clase de razonamiento "para probar una meta *M* es necesario probar esto y esto…" [Bar+99]. Si bien existen varios niveles en los que se clasifican las tácticas, nosotros consideraremos tácticas básicas (simples) que incluyen (posiblemente) a otras tácticas básicas.

A continuación describimos algunas de ellas a modo de ejemplo. El conjunto completo de tácticas utilizadas en el análisis del sistema TCG puede ser consultado en el apéndice C.2.3.

La táctica `BeginForAll` permite inicializar una prueba para el operador $\forall\square$ formalizado con tipos inductivos (`ForAll`). La misma aplica inducción sobre el conjunto de estados alcanzables (de `RState`), y luego simplifica y trata de probar el caso base –la propiedad para el estado inicial– con la táctica `Simpl` y la táctica `Easy`. Esta última

táctica intenta probar un objetivo usando la táctica `Discriminate` y posteriormente, si esta no tiene éxito, la táctica automática `Auto`.

```
Tactic Definition BeginForAll :=
   [<:tactic:<
      Unfold ForAll; Induction 1; [Simpl; Intros; Easy | Idtac]
   >>].
```

Las tácticas `SplitTrans` y `SplitTrans_Simpl` simplifican la prueba por inducción de una propiedad *P* sobre los estados alcanzables (de `RState`) para los pasos inductivos, con `TrGlobal` el tipo de las transiciones. Estas tácticas se usan en las pruebas luego de haber aplicado la táctica `BeginForAll` y probado el caso base. Esto es, simplifican las pruebas del tipo $P \wedge B_i \Rightarrow wp(S_i, P)$ para toda transición cuya condición de activación es B_i y su efecto en el estado del sistema está determinado por S_i. Las dos tácticas intentan probar automáticamente las implicaciones para algunos casos triviales de antecedente falso, consecuente verdadero o donde la prueba del consecuente se sigue trivialmente del antecedente. Los casos a considerar son ocho, uno para cada transición de `TrGlobal` (siete transiciones discretas y una que representa a las transiciones temporales).

```
Tactic Definition SplitTrans :=
   [<:tactic:<
      Intros s1 s2 l rs_s1 p_s1 tr_gl;
      (Generalize rs_s1; Clear rs_s1);
      (Generalize p_s1; Clear p_s1);
      Elim tr_gl
   >>].

Tactic Definition SplitTrans_Simpl :=
   [<:tactic:<
      SplitTrans; [Simpl_In | Simpl_Down | Simpl_Up | Simpl_Approach |
      Simpl_Exit | Simpl_Lower | Simpl_Raise | Simpl_Tick]
   >>].
```

Las tácticas `Simpl_In`, `Simpl_Down`, `Simpl_Up`, `Simpl_Approach`, `Simpl_Exit`, `Simpl_Lower`, `Simpl_Raise` y `Simpl_Tick` permiten simplificar las instancias correspondientes de $P \wedge B_i \Rightarrow wp(S_i, P)$. Estas tácticas podrían ser reducidas a tres si agrupamos las transiciones que no sincronizan, las que sincronizan dos componentes y las que sincronizan tres (esto último para el caso de la transiciones temporales). Sin embargo, definimos una para cada transición a fin de poder usar lemas de inversión particulares. Estos lemas definen principios de análisis de casos para subfamilias [Bar+99]. El uso de lemas de inversión nos permite evitar repetir una misma prueba de inversión en la

demostración de distintas propiedades (o de una misma), logrando de esta manera reducir el tamaño de los términos de prueba y obtener mayor claridad sobre los mismos.

Tres de las ocho tácticas se enuncian a continuación: `Simpl_In` simplifica el caso correspondiente a una transición etiquetada con `In` –que no es de sincronismo–; `Simpl_Approach` simplifica el caso correspondiente a una transición de sincronismo etiquetada con `Approach`; y, `Simpl_Tick` simplifica el caso correspondiente a una transición de sincronismo entre los tres sistemas etiquetada con `Tick`.

```
Tactic Definition Simpl_In :=
   [<:tactic:<
      Intros st1 st2 trt_In;
      Inversion trt_In using cl_TRT_IN;
      Intros x gt_x_kt1 lt_x_kt2 sc sg;
      Elim sc; Elim sg;
      Simpl; Intros; Easy
   >>].

Tactic Definition Simpl_Approach :=
   [<:tactic:<
      Intros st1 st2 sc1 sc2 trt_Approach trc_Approach;
      Inversion trt_Approach using cl_TRT_APPROACH;
      Inversion trc_Approach using cl_TRC_APPROACH;
      Intros y x sg;
      Elim sg;
      Simpl; Intros; Easy
   >>].

Tactic Definition Simpl_Tick :=
   [<:tactic:<
      Intros st1 st2 sc1 sc2 sg1 sg2 trt_Tick trc_Tick trg_Tick;
      Inversion trt_Tick using cl_TRT_INC_TIME;
      Inversion trc_Tick using cl_TRC_INC_TIME;
      Inversion trg_Tick using cl_TRG_INC_TIME;
      Simpl; Intros; Easy
   >>].
```

Los lemas de inversión: `cl_TRT_IN`, `cl_TRT_APPROACH`, `cl_TRC_APPROACH`, `cl_TRT_INC_TIME`, `cl_TRC_INC_TIME` y `cl_TRG_INC_TIME` son generados por las siguientes declaraciones *Coq*

```
Derive Inversion_clear cl_TRT_IN with (st1,st2:(S_Ck ST))
  (TrT st1 In st2) Sort Prop.

Derive Inversion_clear cl_TRT_APPROACH with (st1,st2:(S_Ck ST))
  (TrT st1 Approach st2) Sort Prop.

Derive Inversion_clear cl_TRC_APPROACH with (sc1,sc2:(S_Ck SC))
  (TrC sc1 Approach sc2) Sort Prop.

Derive Inversion_clear cl_TRT_INC_TIME with (st1,st2:(S_Ck ST))
  (TrT st1 Tick st2) Sort Prop.

Derive Inversion_clear cl_TRC_INC_TIME with (sc1,sc2:(S_Ck SC))
  (TrC sc1 Tick sc2) Sort Prop.

Derive Inversion_clear cl_TRG_INC_TIME with (sg1,sg2:(S_Ck SG))
  (TrG sg1 Tick sg2) Sort Prop.
```

El conjunto completo de lemas de inversión utilizados puede consultarse en el apéndice C.2.2.

Ejemplo 4.1. La demostración del invariante $\texttt{Inv}_1$ corresponde a probar el siguiente lema:

```
Lemma lema_Inv1 : (ForAll_TCG Inv1 SiniTCG).
```

Este lema se demuestra fácilmente por inducción en los estados alcanzables utilizando algunas de las tácticas desarrolladas.

```
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Simpl_or (p_s1 H0 H1).
Qed.
```

Observemos que luego de aplicar las tácticas `BeginForAll` y `SplitTrans_Simpl`, la propiedad para el caso base fue probada (con `BeginForAll`) y de las ocho transiciones del sistema TCG sólo una necesitó la aplicación de tácticas adicionales (la transición etiquetada con *Up*). `Simpl_or` aplica el principio de eliminación a su argumento (en general, una disyunción) y luego intenta probar la meta con la táctica `Easy`.

4.3.2 Propiedad Fundamental de *Safety*

El requerimiento de seguridad más importante del sistema es: "*siempre que el tren este cruzando el paso a nivel, la barrera debe estar baja*". Este puede ser formalizado como una propiedad de *safety* a través del siguiente invariante:

```
Definition safeTCG := [s:StGlobal] Cases s of ((st,_),_,(sg,_)) =>
   st=Inside -> sg=Closed end.
```

La demostración de $Init \Rightarrow \forall\Box safeTCG$ corresponde a probar el siguiente lema:

```
Lemma lema_safeTCG : (ForAll_TCG SiniTCG safeTCG).
```

La demostración se sigue del invariante Inv_8, que es más general que la propiedad *safeTCG*, ya que no sólo asegura que cuando el tren está cruzando el paso a nivel la barrera está baja, sino que también luego de que han pasado más de `kt1` unidades de tiempo desde que el tren comenzó a aproximarse al cruce (origen de la señal *Approach*). La prueba usa el teorema `Mon_I` (ver sección 3.3.3), ya que vale: $Inv_8 \Rightarrow safeTCG$. La misma puede consultarse en el apéndice C.2.4.

Algunos de los invariantes formulados en la sección previa especifican propiedades relevantes para el análisis de *safeTCG*, los mismos fueron concebidos a partir de las condiciones de prueba necesarias para la demostración de esta propiedad de *safety*. Sin embargo, no se usaron técnicas de deducción automática de invariantes para este caso de estudio.

4.3.3 Propiedad de *Liveness Non-Zeno*

El interés fundamental en el proceso de verificación del sistema TCG residió en la demostración de invariantes. Estos permiten describir el comportamiento del sistema en cualquier estado en el cual éste se encuentre. Los invariantes, como vimos en la sección previa, permiten establecer propiedades de *safety* y como veremos en esta sección, ayudan a demostrar también otro tipo de propiedades, como las de *liveness*. En esta sección estamos particularmente interesados en el análisis de la propiedad de *liveness Non-Zeno* ("*sistema bien temporizado*"). Esta propiedad asegura que el tiempo no se bloquea, diverge en todas las ejecuciones del sistema. La importancia de *Non-Zeno* para un sistema de tiempo real fue destacada en la sección 2.1.2. *Non-Zeno* puede ser descripta por la siguiente fórmula TCTL:

$$Init \Rightarrow \forall\Box\exists\Diamond_{=1} True$$

Es decir, desde cualquier estado alcanzable del sistema es posible alcanzar un nuevo estado, habiendo transcurrido un tiempo estrictamente positivo. Esta propiedad puede especificarse en *Coq* para el sistema TCG a través de las formalizaciones introducidas en

el capítulo 3 (en particular, las que corresponden a las definiciones de los operadores con tipos inductivos), como sigue:

```
Lemma NonZeno : (ForAll_TCG SiniTCG
   ( [s:StGlobal] (Exists_T_TCG s ([_:StGlobal] True) (eq_Ck tick)) )).
```

La especificación de "puede avanzar" es expresada a través del incremento de un *tick* (condición que es equivalente a `(gt_Ck O)`). `Exists_T_TCG` es la abreviatura

```
Definition Exists_T_TCG : StGlobal -> (StGlobal->Prop) -> Prop :=
   (Exists_T TrGlobal).
```

Una formalización equivalente a la anterior que hace uso de la versión no acotada del operador ∃◊ puede expresarse como sigue,

```
Lemma NonZeno : (ForAll_TCG SiniTCG
   ( [s:StGlobal] (Exists_TCG s InvTick) )).
```

donde, `InvTick` es la condición de habilitación para la transición temporal del sistema TCG ("el tiempo puede avanzar") y `Exists_TCG` es una abreviatura. Formalmente,

```
Definition InvTick := [s:StGlobal] Cases s of ((st,x),(sc,y),(sg,z)) =>
   ( InvTCG ((st,(Inc x)),(sc,(Inc y)),(sg,(Inc z))) ) end.

Definition Exists_TCG: StGlobal -> (StGlobal->Prop) -> Prop :=
   (Exists TrGlobal).
```

Para la demostración de *Non-Zeno* adoptamos su segunda formalización, que corresponde a la fórmula TCTL *Init* ⇒ ∀□∃◊ *InvTick*. La estructura general de la prueba corresponde a una inducción en los estados alcanzables (desde el estado inicial `SiniTCG`) y las transiciones del sistema TCG, en función del operador ∀□. La propiedad a verificar en cada instancia corresponde a la construcción de un camino hacia un estado a partir del cual es posible incrementar el tiempo (∃◊ *InvTick*). En este proceso se utilizaron los invariantes y las tácticas especificadas en la sección previa. Asimismo, se formularon abreviaciones y tácticas específicas, y se utilizó el teorema `StepsEX` introducido en la sección 3.3.3.

En la demostración, el análisis para la transición temporal resultó más complejo que en los otros casos. La prueba en este punto se divide en dos partes según el principio del tercero excluido: vale *InvTick* en el estado alcanzable por la transición temporal –la prueba se sigue trivialmente en este caso– o no vale *InvTick* y entonces la prueba se sigue por análisis de casos de las situaciones posibles: las que pueden violar los invariantes de las locaciones. Por ejemplo, el tren podría estar en la locación *Near* y el valor del reloj

podría superar, luego del incremento, la constante *kt2*, pero entonces sabríamos que el valor sin incrementar del reloj sería mayor que la constante *kt1*. Notemos sin embargo, que si *InvTick* valiera para el controlador y la barrera, el tren no podría estar en la locación *Far*, pues su invariante es *True*. Los casos considerados son seis y relacionan las locaciones cuyos invariantes no son *True* (ver el lema `noInvTick` en el apéndice C.2.5 –sección: lemas, definiciones y tácticas auxiliares–). La demostración completa de *Non-Zeno* puede consultarse en el apéndice C.2.5 –sección: demostración de *Non-Zeno*.

Un Sistema Mal Temporizado

El sistema de "control de un paso a nivel de tren" presentado en [AD94] –tomado de base en este trabajo– difiere del presentado en la sección 4.1 únicamente en la condición de activación para la transición de la barrera, rotulada con *Up*, que vincula las locaciones *Raising* y *Open*. En [AD94] esta condición es $kg2 < z < kg3$, mientras que en la formulación del problema dada en la sección 4.1 la misma es debilitada a $kg2 \leq z < kg3$.

En este trabajo:

$$\frac{kg2 \leq z < kg3}{\langle Raising, z\rangle \xrightarrow{Up}_G \langle Open, z\rangle}$$

En [AD94]:

$$\frac{kg2 < z < kg3}{\langle Raising, z\rangle \xrightarrow{Up}_G \langle Open, z\rangle}$$

Fig. 4.4: Diferencia con la formalización dada en [AD94]

La modificación previa fue introducida en este trabajo debido a que la propiedad *Non-Zeno* para el sistema presentado en [AD94] no se cumple. La especificación del sistema original no garantiza que el tiempo diverge en todas las ejecuciones. A partir del análisis de *Non-Zeno* para dicho sistema dedujimos una condición suficiente que nos permitió transformar el sistema mal temporizado en el sistema bien temporizado analizado en este capítulo. La condición suficiente es precisamente la que distingue a ambas versiones y fue deducida en la etapa de prueba de *Non-Zeno* que corresponde al análisis de la transición temporal.

Un contra-ejemplo para el sistema original está dado por la siguiente traza de ejecución, generada a partir del estado inicial *SiniTCG* = ((*Far*,0),(*Sc1*,0),(*Open*,0)):

$$\underline{((Far,0),(Sc1,0),(Open,0))} \xrightarrow{App} ((Near,0),(Sc2,0),(Open,0)) \xrightarrow{t=1}$$
$$((Near,1),(Sc2,1),(Open,1)) \xrightarrow{Lower} ((Near,1),(Sc3,1),(Lowering,0)) \xrightarrow{Down}$$
$$((Near,1),(Sc3,1),(Closed,0)) \xrightarrow{t=2} ((Near,3),(Sc3,3),(Closed,2)) \xrightarrow{In}$$
$$((Inside,3),(Sc3,3),(Closed,2)) \xrightarrow{Exit} ((Far,3),(Sc4,0),(Closed,2)) \xrightarrow{Raise}$$
$$((Far,3),(Sc1,0),(Raising,0)) \xrightarrow{Approach} ((Near,0),(Sc2,0),(Raising,0)) \xrightarrow{t=1}$$
$$\underline{((Near,1),(Sc2,1),(Raising,1))}$$

El estado ((*Near*,1),(*Sc2*,1),(*Raising*,1)) es alcanzable a partir de *SiniTCG*, sin embargo ninguna transición está habilitada para este estado en el sistema presentado en [AD94]. La modificación al sistema introducida en este trabajo garantiza, en particular, que la transición *Up* está habilitada en este punto y, en general, que *Non-Zeno* vale.

4.3.4 Acerca de los Términos de Prueba

El tamaño de los términos de prueba de los invariantes analizados es proporcional a la suma de los tamaños de las demostraciones para las transiciones del sistema compuesto (hacemos inducción en el conjunto de estados alcanzables). Para la propiedad de *safety*, se sigue del invariante Inv_8. El tamaño del término de prueba para *NonZeno* es proporcional a la suma de los tamaños de los caminos construidos a partir de cada transición del sistema compuesto. La longitud de estos caminos no supera la construcción de tres pasos o transiciones –salvo para el análisis de la transición temporal– y si bien fue necesario hacer análisis de casos sobre las locaciones de los estados, nunca se analizaron simultáneamente todos los casos para los tres sistemas componentes; se usaron los invariantes demostrados para deducir los valores de algunas locaciones y ciertas condiciones de los relojes. El camino de prueba de mayor costo corresponde al de la transición temporal, ya que este se dividió en seis a partir del caso en que no vale *InvTick* (ver sección 4.3.3).

Para simplificar la construcción de las pruebas se desarrollaron tácticas (ver sección 4.3.1) que automatizan ciertas partes de las mismas. Estas si bien permiten reducir el *script* de prueba –lista de tácticas usadas en la demostración–, no reducen el tamaño de los términos de prueba. Para el caso de estudio de este capítulo algunas decisiones tomadas condujeron a disminuir el tamaño de los términos de prueba para las propiedades analizadas:

- Uso de lemas de inversión para las transiciones y los invariantes de las locaciones. El desarrollo de estos lemas nos permite evitar repetir más de una vez en las pruebas el principio de análisis de casos para algunas subfamilas (ver sección 4.3.1).
- Abstracción en las transiciones discretas de la condición que establece que el invariante de la locación destino de las transiciones se satisface (ver sección 4.2). Esta es una simplificación sobre el formato establecido en la sección 3.2.
- Definición particular de la relación de transición para el sistema compuesto. La definición de composición de transiciones puede ser expresada con un operador genérico, como veremos en el capítulo 5, pero de esta manera se incrementan los tamaños y oscurece la estructura de las pruebas, si no se implementan tácticas adicionales.

- Utilización de la formalización de estados alcanzables que no explicita que los estados iniciales deben satisfacer los invariantes de las locaciones (ver discusión en la sección 3.3.2). `InvTCG` vale para el estado inicial considerado en cada propiedad analizada: `SiniTCG`.
- Utilización de un conjunto global de etiquetas en vez de conjuntos particulares para cada subsistema. Esta discusión fue planteada a nivel general en la sección 3.2, de acuerdo a la hipótesis establecida en la sección 2.1.

4.4 Parametrización del Sistema TCG

En esta sección mostramos una posible generalización de la especificación del sistema presentado en la sección 4.1 que preserva las propiedades analizadas del mismo. El objetivo es poder definir una familia de sistemas que cumplan las propiedades de interés; donde el sistema especificado al comienzo de este capítulo sea un miembro particular de la familia.

Para la formulación de las propiedades analizadas a lo largo de la sección 4.3 consideramos los valores de las constantes del sistema TCG especificados en [AD94] multiplicados por cuatro, tal como lo señalamos en la sección 4.1. En la demostración de las propiedades consideramos lemas especiales para aquellas demostraciones que involucraban relaciones elementales (triviales) entre los valores de las constantes y los relojes del sistema. El conjunto completo de lemas, con sus demostraciones, se incluye en el apéndice C.2.1. A continuación exhibimos algunos ejemplos.

```
Lemma Trivial6  : ∀x,y∈Clock (ge_Ck x y) -> (ge_Ck (Inc x) (Inc y)).

Lemma Trivial10 : ∀x∈Clock (gt_Ck x kt1) -> (gt_Ck x kc1).

Lemma Trivial12 : ∀x,y∈Clock (gt_Ck (Inc x) y) -> (eq_Ck x y) \/ (gt_Ck x y).

Lemma Trivial18 : (lt_Ck (Inc Reset) kc2).

Lemma Trivial30 : ∀x∈Clock (lt_Ck x kg2) -> (lt_Ck (Inc x) kg3).
```

`Trivial6` es una de las propiedades elementales acerca de las relaciones binarias de comparación y la operación de incremento temporal. `Trivial10` establece una relación entre valores constantes del sistema: `kt1`≥`kc1`. `Trivial12` es una propiedad fundamental de la discretización que expresa que los incrementos temporales son los mínimos para el dominio elegido (`N` en este caso). Esta propiedad podría ser considerada un axioma de la discretización si deseáramos axiomatizar la estructura temporal. `Trivial18` expresa que la constante `tick` es menor que `kc2`; esta propiedad vale por el factor de discretización elegido $\delta = 1/M$, donde M es la constante en base a la cual aumentamos la escala del problema. `Trivial30` es otra relación entre constantes del problema que expresa la condición: `kg3>kg2`.

Muchas de estas propiedades se demuestran fácilmente a partir de la estructura temporal elegida (`N` y las relaciones binarias de comparación habituales), teniendo en cuenta el factor de multiplicación de escala *M*. Por ejemplo `Trivial6`, `Trivial12` y `Trivial18`. Las propiedades que establecen relaciones entre las constantes del problema se demuestran también muy fácilmente a partir de sus valores. Sin embargo, si consideramos a estas últimas como axiomas podemos entonces extender la especificación del problema, parametrizando las constantes. De esta manera obtenemos una formulación más general del sistema que preserva las propiedades analizadas.

Las propiedades que vinculan a las constantes del problema en la demostración de *safeTCG* resultan válidas si se consideramos la siguiente restricción:

$$kt1 - kc1 + 1 \geq kg1$$

La especificación del sistema TCG que preserva *safeTCG* puede entonces darse para valores cualesquiera de las constantes que cumplan con la desigualdad anterior, que es una condición suficiente.

Todas las propiedades elementales utilizadas –que involucran a las constantes del problema– en la demostración de las propiedades analizadas en este trabajo (incluyendo *Non-Zeno*) se satisfacen para el siguiente conjunto de restricciones entre los parámetros del sistema:

- $kt1 \geq kg1 + kc1$
- $kg3 > kg2$
- $kc1 \geq kg2$
- $kt2 - 1 > kt1$
- $kc1 \geq kc2$
- $ki > 1$, para todo parámetro $ki \in \{kt1, kt2, kc1, kc2, kg1, kg2, kg3\}$.

Esta última condición resulta válida para el factor de discretización en función de constantes no nulas.

Las restricciones presentadas son condiciones suficientes, deducidas a partir de las propiedades elementales utilizadas en las pruebas de las propiedades destacadas del sistema TCG. Las mismas generalizan "parametrizan" la especificación en función de las propiedades de interés. Luego, podemos incluir como variables las constantes del sistema y especificar como axiomas las condiciones establecidas:

```
Parameter kt1,kt2,kc1,kc2,kg1,kg2,kg3 : Instant.
Axiom AxTCG_1 : kt1 – kc1 + 1 ≥ kg1.
.....
```

4.5 Trabajos Relacionados

Como señalamos al comienzo de este capítulo, la especificación del sistema abordado fue tomada de [AD94]. Numerosos trabajos previos consideran a este problema como caso de estudio ("benchmark") para analizar diferentes técnicas de especificación y herramientas de análisis. Entre otros, [HNS+92, CHR92, Sha93, HJL93, HK94, DY95, AB96, BMS+97]. En particular, en [BMS+97] se hace un análisis comparativo del caso de estudio en diversos formalismos. Algunos de los trabajos citados consideran una versión generalizada del problema que permite modelar un número arbitrario de trenes en el sistema. En nuestro caso de estudio el sistema fuerza a secuencializar la subida y bajada de la barrera (y por lo tanto impone una distancia entre trenes que difiere al menos en el tiempo que lleva subir y bajar la barrera).

Respecto a los trabajos previos, nuestro enfoque consiste en el análisis del sistema a partir de su formulación como un conjunto de grafos temporizados (sistemas de transiciones de estados que permiten la utilización de relojes explícitos), que interactúan según la composición paralela de los mismos. Consecuentemente, no son necesarios operadores tales como *since* para registrar el tiempo transcurrido desde la última vez que valía una propiedad ni otros similares [Sha93]. En la formalización de sistemas de tiempo real con el operador *since* no se requieren contadores o relojes para medir el tiempo transcurrido en torno a una propiedad, la especificación de *since* está dada por un conjunto de axiomas y el comportamiento de un sistema se expresa a través de condiciones en una lógica de orden superior que permite el uso del operador *since* sobre los predicados.

La utilización de múltiples relojes, que pueden ser reseteados en cualquier transición, y la disponibilidad de un mecanismo de definición composicional facilitan el proceso definición del sistema, su comprensión (aún por personas no expertas) y también su análisis, ya que resulta bastante fácil y natural la formulación de propiedades. Las demostraciones de invariantes y de la propiedad de *safety* considerada son bien tratables con el enfoque seguido, aunque en general las propiedades de *liveness* como *Non-Zeno* son complejas (al igual que en los demás enfoques deductivos). Luego, estas últimas podrían ser probadas con un *model checker* automático sobre el grafo y consideradas axiomas en el ambiente de teoría de tipos.

En la especificación de un grafo temporizado necesitamos, en general, la definición de valores constantes para los parámetros del sistema, a fin de poder verificar propiedades con un *model checker*. Sin embargo, como vimos en la sección 4.4, es posible generalizar la especificación del sistema en estudio preservando ciertas propiedades de interés, asumiendo parámetros variables sujetos a determinadas restricciones (deducidas a partir de condiciones de prueba de las propiedades que se intentan preservar).

Otra ventaja de la formalización del sistema en torno a grafos temporizados, en contraste con métodos axiomáticos para especificar restricciones temporales, reside en el contenido algorítmico de la especificación, tal como analizamos en la sección 2.5.3, que

permite simular el sistema con el control de los múltiples relojes. Creemos que esta formalización, además de permitirnos usar un *model checker* automático como asistente para la demostración de ciertas propiedades, es adecuada para la síntesis de programas, en el marco de teoría de tipos y en particular del cálculo de construcciones inductivas y co-inductivas de *Coq*.

Capítulo 5

Representaciones Genéricas de Grafos Temporizados

En el capítulo 3 incluimos algunas definiciones básicas para describir grafos temporizados simples. Las mismas fueron utilizadas para la especificación y el análisis del caso de estudio abordado en el capítulo 4. En este capítulo definimos dos representaciones genéricas de grafos temporizados para la semántica de tiempo discreto considerada en los capítulos previos, una de las cuales es extendida a tiempo continuo. Estas representaciones permiten definir sistemas como instancias particulares y simplifican el proceso de definición de sistemas compuestos.

5.1 Grafos Temporizados Genéricos en *Coq*

En el capítulo 4 especificamos la composición paralela de los tres sistemas componentes del sistema TCG de manera particular, sin el uso de un operador genérico de composición. Sin embargo, de la sección 2.2 se desprende que la definición de tal operador es posible. En esta sección presentamos dos formalizaciones genéricas de grafos temporizados que permiten simplificar el proceso de definición de sistemas compuestos.

Asumiremos, al igual que en los capítulos previos, un tipo `Label` que representa el conjunto global de etiquetas. Sin embargo, no exigimos la existencia de una etiqueta distinguida `Tick` para caracterizar a las transiciones temporales.

5.1.1 Representación

Un grafo temporizado puede ser visto como una 4-upla paramétrica en el tipo de los estados, compuesta por una relación que define a las transiciones discretas, un estado inicial, un predicado que caracteriza a los invariantes de las locaciones (utilizado para caracterizar a las transiciones temporales) y una función que permite incrementar los relojes de los estados para un valor temporal dado. En *Coq* este tipo puede formalizarse a través de *registros paramétricos* como sigue:

```
Record Tgraph [S:Set] : Type :=
   mkTG  {  trans: S->Label->S->Prop;
            sini  : S;
            inv   : S->Prop;
            inc   : S->Instant->S
         }.
```

`mkTG` construye objetos de tipo `(Tgraph S)` y `trans`, `sini`, `inv` e `inc` son las proyecciones de esta clase de objetos. Esto es, dados `tr` de tipo `S->Label->S->Prop`, `si` de tipo `S`, `iv` de tipo `S->Prop` e `ic` de tipo `S->Instant->S`, `(mkTG tr si iv ic)` es un objeto de tipo `(Tgraph S)`. Asimismo, dado `TG` de tipo `(Tgraph S)`, `(trans TG)` es un objeto de tipo `S->Label->S->Prop`, `(sini TG)` tiene tipo `S`, `(inv TG)` tiene tipo `S->Prop` e `(inc TG)`, tipo `S->Instant->S`.

El estado inicial de un grafo `G` debe ser un estado válido, esto es la proyección `sini` de `G` tiene que satisfacer el predicado `(inv G)`. Esta restricción puede plasmarse en el siguiente tipo de grafos válidos:

```
Record Tgraph_V [S:Set] : Type :=
     mkTG_V { TG : (Tgraph S); siniV : ((inv TG) (sini TG)) }.
```

Nota 5.1. En lo que sigue nos referiremos a grafos de tipo `Tgraph` pero asumiremos que estos corresponden a grafos válidos, es decir son la proyección de un objeto de tipo `Tgraph_V`.

Ejemplo 5.1. Para el caso de estudio del capítulo 4, los grafos componentes del sistema TCG pueden instanciarse de la siguiente manera:

```
Definition Inc_simple  := [S:Set; st:(S_Ck S); t:Instant]
   Cases st of (s,x) => (s,(plus_Ck x t)) end.

Definition Train      := (mkTG TrT' SiniT InvT (Inc_simple 1!ST)).

Definition Controller := (mkTG TrC' SiniC InvC (Inc_simple 1!SC)).

Definition Gate       := (mkTG TrG' SiniG InvG (Inc_simple 1!SG)).
```

`TrT'`, `TrC'` y `TrG'` se definen como las relaciones `TrT`, `TrC` y `TrG` de la sección 4.2 para las transiciones discretas. Esto es, sin considerar las reglas para los constructores *`ttTick`*, *`tcTick`* y *`tgTick`*. `Inc_simple` es la operación de incremento temporal sobre estados con un único reloj. Con `1!` explicitamos el primer argumento de `Inc_simple`, que es asumido implícito en función del segundo argumento de la función.

5.1.2 Composición Paralela

La composición paralela de dos grafos temporizados puede especificarse de manera genérica para la representación introducida en la sección previa. El tipo de los estados del sistema compuesto es definido como el producto de los correspondientes a cada subsistema.

Las transiciones del sistema compuesto se definen de manera similar que en la sección 2.2, para las transiciones discretas. Distinguimos transiciones de sincronización, que comparten una misma etiqueta, y asíncronas. Estas últimas contemplan dos casos, uno para cada componente. La caracterización de "no es una transición de sincronización" está dada por un predicado `no_label` que especifica que una etiqueta no pertenece a las transiciones de un grafo. Formalmente,

```
Definition no_label := [S:Set; tr:S->Label->S->Prop; l:Label]
  ∀s,s'∈S ~(tr s l s').

Section Composition.

  Variable S1,S2:Set.
  Variable tr1:S1->Label->S1->Prop.
  Variable tr2:S2->Label->S2->Prop.

  Inductive tr_c : (S1*S2)->Label->(S1*S2)->Prop :=

    tr_c_syn : ∀s1,s1'∈S1 ∀s2,s2'∈S2 ∀l∈Label
      (tr1 s1 l s1') ->
      (tr2 s2 l s2') ->
      (tr_c (s1,s2) l (s1',s2'))

    |tr_c_nosyn1 : ∀s1,s1'∈S1 ∀s2,s2'∈S2 ∀l∈Label
      (tr1 s1 l s1') ->
      (no_label tr2 l) ->
      (tr_c (s1,s2) l (s1',s2))

    |tr_c_nosyn2 : ∀s1,s1'∈S1 ∀s2,s2'∈S2 ∀l∈Label
      (tr2 s2 l s2') ->
      (no_label tr1 l) ->
      (tr_c (s1,s2) l (s1,s2')).

End Composition.
```

El predicado que define a los invariantes de las locaciones de la composición está dado por la conjunción de los invariantes correspondientes y la operación de incremento temporal se define a partir de la de cada componente. Esto es,

```
Definition inv_c := [S1,S2:Set; inv1:S1->Prop; inv2:S2->Prop;
   s:(S1*S2)]
      Cases s of (s1,s2) => (inv1 s1) /\ (inv2 s2) end.

Definition inc_c := [S1,S2:Set; inc1:S1->Instant->S1;
   inc2:S2->Instant->S2; s:(S1*S2); t:Instant]
      Cases s of (s1,s2) => ((inc1 s1 t),(inc2 s2 t)) end.
```

Finalmente, la composición paralela de dos grafos temporizados G_1 y G_2, $G_1 \| G_2$, se define de la siguiente manera:

```
Definition Parallel_composition :=
   [S1,S2:Set; G1:(Tgraph S1); G2:(Tgraph S2)]
      (mkTG
         (tr_c (trans G1) (trans G2))
         ((sini G1),(sini G2))
         (inv_c (inv G1) (inv G2))
         (inc_c (inc G1) (inc G2))).
```

Las transiciones temporales de la composición, al igual que las de cada componente, están implícitamente representadas por los invariantes de las locaciones y serán explicitadas en la definición de trazas de ejecución.

Ejemplo 5.2. Para el caso de estudio del capítulo 4, el grafo correspondiente a la composición paralela (`Train` || `Controller` || `Gate`) puede especificarse como sigue

```
Definition system_TCG :=
   (Parallel_composition Train (Parallel_composition Controller Gate)).
```

El tipo de los estados de la composición de los tres sistemas coincide con el asumido en el capítulo 4: `(S_Ck ST)*(S_Ck SC)*(S_Ck SG)`. El estado inicial `SiniTCG` es `(sini system_TCG)` y las transiciones discretas del tipo `TrGlobal` (todas menos la correspondiente al constructor *`tGl_Tick`*) corresponden conceptualmente a las transiciones del tipo `(tr system_TCG)`.

5.1.3 Trazas de Ejecución

Dado un grafo temporizado *G*, las trazas de ejecución válidas de *G*, que no involucran referencias temporales (correspondiente a `isTrace` de la sección 3.4.2), pueden ser caracterizadas por el siguiente predicado co-inductivo:

```
CoInductive isTraceTG [S:Set; G:(Tgraph S)] : (Stream S)->Prop :=
  isTraceDisc : ∀x∈(Stream S) ∀s1,s2∈S ∀l∈Label
    (trans G s1 l s2) ->
    (inv G s2) ->
    (isTraceTG S G s2^x) ->
    (isTraceTG S G s1^s2^x)
  |isTraceTick : ∀x∈(Stream S) ∀s∈S
    (inv G (inc G s tick)) ->
    (isTraceTG S G (inc G s tick)^x) ->
    (isTraceTG S G s^(inc G s tick)^x).
```

isTraceDisc corresponde a pasos representados por transiciones discretas; *isTraceTick* construye pruebas de transiciones temporales de un estado s_1 a otro s_2 –que posee la misma locación que s_1 pero con los relojes incrementados en un `tick`– si se satisface el invariante de la locación en s_2. En la formalización de las transiciones, tanto discretas como temporales, explicitamos que el estado destino de la transición debe cumplir el invariante de la locación (en el capítulo 3, sección 3.2, esta hipótesis se refleja en el formato exigido a las transiciones).

Notemos que mientras los pasos discretos en una traza se obtienen por transiciones rotuladas del grafo, los pasos temporales están representados por transiciones implícitamente rotuladas y definidos en función de los invariantes de las locaciones.

Las trazas generadas a partir de un estado inicial dado pueden ser definidas como sigue,

```
Definition isTraceFromTG :=
  [S:Set; G:(Tgraph S); Sini:S; x:(Stream S)]
    Sini=(hd x) /\ (inv G Sini) /\ (isTraceTG G x).
```

De acuerdo a la discusión planteada en el capítulo 3 (secciones 3.3.2 y 3.4.2), podemos obviar la restricción (`inv G Sini`), considerándola una hipótesis a corroborar.

Asimismo, las trazas que explicitan el valor temporal del *reloj global* y sobre las cuales definimos los operadores de la lógica TCTL se definen, al igual que en la sección 3.4.3, como una generalización de las definiciones anteriores.

```
CoInductive isTraceTG_T [S:Set; G:(Tgraph S)] :
  (Stream S*Instant)->Prop :=
  isTraceDisc_T : ∀x∈(Stream S*Instant) ∀s1,s2∈S ∀l∈Label ∀t∈Instant
    (trans G s1 l s2) ->
    (inv G s2) ->
    (isTraceTG_T S G (s2,t)^x) ->
    (isTraceTG_T S G ( (s1,t)^(s2,t)^x ))
```

```
|isTraceTick_T : ∀x∈(Stream S*Instant) ∀s∈S ∀t∈Instant
   (inv G (inc G s tick)) ->
   (isTraceTG_T S G ((inc G s tick),(Inc t))^x) ->
   (isTraceTG_T S G ( (s,t)^((inc G s tick),(Inc t))^x )).
```

Cada elemento de una traza de ejecución de un grafo *G* es un par compuesto por un estado de *G* (la locación más las valuaciones de los relojes) y el valor temporal del *reloj global*. Este último se incrementa con cada paso temporal (`isTraceTick_T`), manteniendo su valor en los pasos correspondientes a las transiciones discretas (`isTraceDisc_T`).

Asimismo, las trazas que poseen comienzo en un estado, para un valor temporal dado del reloj global se definen por la siguiente abreviatura:

```
Definition isTraceFromTG_T :=
   [S:Set; G:(Tgraph S); Sini:S*Instant; x:(Stream S*Instant)]
      Sini=(hd x) /\ (inv G Sini) /\ (isTraceTG_T G x).
```

Nuevamente, la restricción `(inv G Sini)` puede ser obviada, considerándola una hipótesis a corroborar en el análisis de propiedades sobre las trazas.

Los operadores temporales se definen como en la sección 3.4, tanto los correspondientes al fragmento CTL (sobre trazas caracterizadas con `isTraceTG`) como los de TCTL (sobre trazas caracterizadas con `isTraceTG_T`).

5.1.4 Una Representación Alternativa

Una representación genérica más simple de grafos temporizados se obtiene si consideramos, como en la sección 3.2, que las transiciones temporales y las discretas están representadas por una relación de tipo `S->Label->S->Prop`, con `S` el tipo de los estados del grafo y `Tick` de tipo `Label` una etiqueta distinguida que denota las transiciones temporales. De esta manera un grafo es simplemente un par (los invariantes de las locaciones y la función que permite incrementar los relojes son innecesarios).

```
Record Tgraph [S:Set] : Type :=
    mkTG { trans : S->Label->S->Prop; sini: S }.
```

Ejemplo 5.3. Para el caso de estudio del capítulo 4, los grafos componentes del sistema TCG se instancian de la siguiente manera:

```
Definition Train      := (mkTG TrT SiniT).
Definition Controller := (mkTG TrC SiniC).
Definition Gate       := (mkTG TrG SiniG).
```

La composición paralela de dos grafos temporizados es el grafo que compone las transiciones según la relación inductiva `tr_c` definida previamente y cuyo estado inicial se define como antes. A diferencia que en la versión previa, el constructor `tr_c_syn` de `tr_c` permite sincronizar, además de transiciones discretas, transiciones temporales con `Tick`.

```
Definition Parallel_composition :=
   [S1,S2:Set; G1:(Tgraph S1); G2:(Tgraph S2)]
      (mkTG  (tr_c (trans G1) (trans G2))
             ((sini G1),(sini G2))).
```

Para el caso de estudio del capítulo 4, el sistema TCG (`Train` || `Controller` || `Gate`) se define igual que en el ejemplo 5.2.

Las trazas temporizadas y las no temporizadas se definen igual que en la sección 3.4 (para el formato de transiciones establecido en la sección 3.2), como así también los operadores temporales de CTL y TCTL.

5.1.5 Discusión

La primera generalización permite instanciar grafos temporizados sin necesidad de incluir explícitamente la definición de las transiciones temporales. La segunda supone que éstas son definidas junto con las transiciones discretas. Mientras la primera facilita más el proceso de definición de los grafos básicos, la segunda es más compacta y permite simplificar la estructura y el tamaño de las pruebas.

Ambas generalizaciones definen a las transiciones de una composición de manera uniforme, a diferencia que en el capítulo 4 donde especificamos la relación de composición para los tres grafos básicos con una definición particular. Con el enfoque genérico simplificamos el proceso de definición; sin embargo, las demostraciones se vuelven más complejas y los términos de prueba crecen en tamaño. Con el fin de experimentar, demostramos algunas propiedades para una especificación genérica del sistema TCG analizado en el capítulo 4. Las pruebas resultaron más extensas que para la versión original, tanto en relación al número de tácticas del *script* de prueba, como al tamaño de los términos de dichas pruebas. La razón se debe a la utilización del operador genérico de composición de transiciones que aumenta el número de subpruebas estrictamente necesarias y obliga a eliminar casos inválidos (transiciones no definidas), tanto para las transiciones de sincronización como las que no sincronizan, sobre el conjunto de las etiquetas del sistema. Los casos inválidos para determinadas etiquetas pueden ser eliminados realizando *inversiones* sobre las transiciones involucradas, aunque de esta manera aumentan los tamaños de los términos de prueba.

Al trabajar con una definición general de composición de transiciones deberían desarrollarse tácticas que permitan simplificar el desarrollo de las pruebas sobre estas transiciones, reduciendo al mismo tiempo los tamaños de los términos de prueba.

5.2 Grafos con Tiempo Continuo

5.2.1 Adaptación de las Formalizaciones Previas

Coq posee librerías que permiten trabajar con números racionales (`Q`) y reales (`R`) [Bar+99]. Luego, la formalización genérica de grafos temporizados dada en la sección 5.1 puede ser adaptada a un modelo temporal continuo, salvo la propuesta alternativa presentada en la subsección 5.1.4. Las modificaciones necesarias son las siguientes:

- Los tipos `Instant` y `Clock` son abreviaciones del tipo `T`. Donde `T` representa el dominio temporal continuo elegido (`Q` o `R`). Los operadores considerados en la sección 3.1 se redefinen en función de `T` (salvo `tick` e `Inc` que no son considerados en el dominio continuo).
- Las definiciones de trazas de ejecución dadas en 5.1.3 deberían ser modificadas como sigue:

```
CoInductive isTraceTG [S:Set; G:(Tgraph S)] : (Stream S)->Prop :=
 IsTraceDisc : ∀x∈(Stream S) ∀s1,s2∈S ∀l∈Label
    .....
 |IsTraceTick : ∀x∈(Stream S) ∀s∈S ∀t∈Instant
    (∀t'∈Instant (le_Ck 0 t') -> (le_Ck t' t) ->
      (inv G (inc G s t')) ) ->
    (isTraceTG S G (inc G s t)^x) ->
    (isTraceTG S G s^(inc G s t)^x).


CoInductive isTraceTG_T [S:Set; G:(Tgraph S)] :
 (Stream S*Instant)->Prop :=
 IsTraceDisc_T : ∀x∈(Stream S*Instant) ∀s1,s2∈S
    ∀l∈Label ∀t∈Instant
    .....
 |IsTraceTick_T : ∀x∈(Stream S*Instant) ∀s∈S ∀t,tg∈Instant
    (∀t'∈Instant (le_Ck 0 t') -> (le_Ck t' t) ->
      (inv G (inc G s t')) ) ->
    (isTraceTG_T S G ((inc G s t),(plus_Ck tg t))^x) ->
    (isTraceTG_T S G ( (s,tg)^((inc G s t),(plus_Ck tg t))^x )).
```

En las transiciones temporales no existe un valor mínimo de tiempo que sea el "siguiente" del instante previo, por la propiedad de densidad, propia de un modelo continuo. Es por esto que la formalización dada en la sección 5.1.4 (basada en las formalizaciones de la sección 3.2) no es válida en un modelo de tiempo continuo, debido a que si bien las transiciones temporales de los grafos particulares de un sistema deben sincronizar con una etiqueta distinguida `Tick`, el tiempo podría evolucionar diferente en cada subsistema (los valores de incremento podrían diferir).

En las definiciones de `isTraceTG` e `isTraceTG_T` los pasos temporales son definidos como en la sección 2.1.1. Es decir, hay una transición temporal del sistema de `t` unidades de tiempo, si el invariante de las locaciones vale continuamente para todo incremento menor o igual que `t`.

Los operadores temporales se definen como en la sección 3.4, tanto los correspondientes al fragmento CTL (sobre trazas caracterizadas con `isTraceTG`) como los de TCTL (sobre trazas caracterizadas con `isTraceTG_T`).

5.2.2 Limitaciones de los Reales y los Racionales en *Coq*

Los números reales en *Coq* están formalizados a partir de una axiomatización (18 axiomas) más algunos lemas de base sobre las partes fraccionarias y enteras [Bar+99]. Consecuentemente, la manipulación de estos números es compleja, aunque todas las propiedades válidas para los mismos podrían ser probadas. Al presente, el autor no conoce trabajos que usen la formalización de los reales en aplicaciones concretas, salvo un trabajo de matemática pura relacionado con el teorema de los tres intervalos ("the three gap theorem") [May97]. Asimismo, tampoco hay tácticas que permitan automatizar o simplificar ciertas pruebas, o partes de las mismas, como en el caso de los enteros y los naturales (por ejemplo, con la táctica *Omega*), excepto con la táctica *Ring* que está definida tanto para números naturales, enteros, racionales, como reales. Esta táctica permite hacer reescritura en anillos; normaliza términos de una meta con respecto a la conmutatividad y la asociatividad y los reemplaza por su forma normal; cuando la meta es una igualdad, intenta solucionarla o simplificarla [Bar+99]. Otra limitación de los reales en *Coq* es que no es posible para los reales usar el procedimiento de *extracción de programas* de *Coq*, ya que éstos están definidos en el tipo *Type* de *Coq*. Esta última restricción puede ser significativa en una etapa de síntesis de programas.

Los números racionales en *Coq* están definidos como el producto de enteros y enteros estrictamente positivos cocientados por la relación usual [Bar+99]. En la formalización se asumen dos conjuntos de axiomas, uno correspondiente a los cocientes y el otro a subconjuntos. Las observaciones relativas a la complejidad de la utilización de los racionales son análogas que las argumentadas para los reales. La igualdad para los racionales se define en función de clases de equivalencia, ya que, por ejemplo, 1 = 2/2 pero ambos términos son constructivamente diferentes. Al igual que para los reales, el autor desconoce, hasta el momento, aplicaciones que usen a los racionales en otros

trabajos. Tampoco existen automatizaciones (tácticas especializadas) que simplifiquen pruebas en este dominio, exceptuando aquellas tratables con la táctica *Ring*.

Todas las limitaciones analizadas que complican el trabajo con los números reales y los racionales en *Coq* son líneas actuales de investigación.

Capítulo 6

Conclusiones, Trabajos Relacionados y Trabajos Futuros

6.1 Conclusiones

En este trabajo formalizamos en teoría de tipos –y en particular en *Coq*– sistemas de tiempo real representados como grafos temporizados, asumiendo una semántica de tiempo discreto. A fin de especificar y demostrar requerimientos temporales sobre los sistemas, formalizamos la lógica TCTL y su restricción CTL para operadores temporales no cuantitativos. Dimos algunas definiciones alternativas para los operadores que permiten expresar invarianza y alcanzabilidad, con tipos inductivos (bajo la noción de estados alcanzables) y co-inductivos (bajo la noción de trazas –infinitas– de ejecución). Estos últimos necesarios para una descripción completa de todas las fórmulas de la lógica TCTL (y CTL). Asimismo, formulamos y probamos en *Coq* propiedades generales de los operadores temporales que permiten simplificar ciertas pruebas.

En el trabajo pusimos un especial énfasis en la especificación y el análisis de un caso de estudio, considerado como "*benchmark*" en diferentes trabajos: *el control de un paso a nivel de tren* ("*the railroad crossing example*"). Establecimos un conjunto de invariantes y en función de ellos probamos la propiedad de seguridad esencial del sistema y la propiedad de *liveness Non-Zeno* que justifica la corrección temporal del sistema. Las demostraciones fueron parametrizadas en un conjunto de restricciones entre los parámetros del sistema, inicialmente considerados constantes y posteriormente generalizados a variables sujetas a las restricciones establecidas –deducidas a partir de las condiciones de prueba de las propiedades que se decidieron preservar. De esta manera generalizamos la especificación del sistema, que es tratable con un *model checker* sólo para valores constantes de los parámetros.

Finalmente, definimos dos representaciones genéricas de grafos temporizados para la semántica de tiempo discreto considerada, una de las cuales fue extendida a tiempo continuo. Estas representaciones permiten obtener sistemas como instancias particulares y simplifican el proceso de definición de sistemas compuestos con el uso de un operador genérico de composición (evitando definir la composición de dos o más grafos temporizados de forma particular cada vez).

Las formalizaciones de grafos temporizados y las lógicas consideradas, traducidas en bibliotecas *Coq*, y la validación de éstas a través de un *benchmark*, constituyen los aportes de este trabajo y un punto de partida para la realización de nuevos trabajos de investigación en torno al análisis de sistemas de tiempo real en teoría de tipos.

A continuación proponemos una metodología de trabajo que relaciona el contenido de los capítulos previos y busca compatibilizar el uso de un *model checker* (MC) con el desarrollo de sistemas en un ambiente de pruebas. Los objetivos de la misma son:

- Definir un esquema de representación de los sistemas común a los dos enfoques de análisis en cuestión.
- Establecer un proceso de análisis de los sistemas que vincule los resultados de la verificación de propiedades para los enfoques considerados.
- Permitir trabajar con sistemas de tiempo real con *parámetros variables*.
- Abarcar una etapa de *síntesis* de sistemas de tiempo real.

La metodología generaliza el análisis realizado para el caso de estudio abordado en el capítulo 4 y es esquematizada en la figura 6.1.

Una Metodología de Trabajo

El análisis de sistemas de tiempo real, representados por grafos temporizados, podría dividirse en dos etapas, no necesariamente independientes entre sí ni secuenciales. La primera –optativa y no profundizada en este trabajo– consiste en la especificación y verificación de ciertas propiedades de un sistema en estudio usando un MC automático, por ejemplo Kronos, HyTech o Uppaal ("delegación de pruebas" en la figura 6.1). Entre estas propiedades consideramos relevantes las de *liveness*, como *Non-Zeno*, y las de alcanzabilidad que son, en general, más difíciles de demostrar deductivamente, aunque algunas de éstas pueden transformarse en otras equivalentes más simples.

La segunda etapa consiste en la especificación y el análisis del sistema en el cálculo de construcciones inductivas y co-inductivas de *Coq* (otros sistemas de pruebas similares a *Coq* son, por ejemplo, ALF [Mag94] y LEGO [LP92]). Para esta etapa podemos asumir una semántica de tiempo discreto, como en la mayor parte de este trabajo, o trabajar en un modelo de tiempo continuo, tal como analizamos en la sección 5.2.

Una manera de incorporar las propiedades demostradas con un MC en la metodología de trabajo en teoría de tipos consiste en asumir a las mismas como axiomas o hipótesis. Como el lenguaje de especificación de los requerimientos temporales –la lógica TCTL y/o CTL– y el formalismo de descripción de los sistemas –los grafos temporizados– son los mismos en ambas etapas, no son necesarias traducciones significativas para compatibilizar el uso de un MC como Kronos con un ambiente de teoría de tipos como *Coq*.

La especificación de un sistema puede obtenerse de una representación genérica de los grafos y la composición paralela de los mismos, como describimos en las secciones previas de este capítulo, o a través de definiciones particulares para la composición, tal como procedimos en el caso de estudio del capítulo 4. Las ventajas de ambos enfoques fueron analizadas en la sección 5.1.5.

Los operadores lógicos de TCTL y CTL, formalizados en el capítulo 3, permiten formular requerimientos temporales, es decir pueden ser utilizados para la verificación de propiedades sobre los sistemas especificados. Para propiedades que sólo involucran $\exists\Diamond_I$ y $\forall\Box_I$ ($\exists\Diamond$ y $\forall\Box$) podemos optar entre una formalización con tipos inductivos y una con tipos co-inductivos. Para propiedades arbitrarias, la formalización requiere el uso de tipos co-inductivos y por lo tanto, el uso de co-inducción como mecanismo de prueba de determinadas propiedades.

Las demostraciones de las propiedades de *safety* y de *liveness* del sistema TCG se basaron fuertemente en el conocimiento de un conjunto de propiedades invariantes del sistema. La deducción y prueba de invariantes permite incrementar el conocimiento de un sistema y consecuentemente probar otras propiedades con su ayuda [MP91]. Luego, en el análisis de los sistemas sugerimos trabajar de esta manera. Para la deducción de invariantes existen algunos trabajos que podrían ser considerados [MP95, BBM97, BMS+97, BL99].

A partir de las propiedades analizadas de un sistema y consideradas relevantes puede intentarse generalizar la especificación del sistema asumiendo a las constantes del mismo como variables que deben satisfacer un conjunto de restricciones (deducidas en función de condiciones de prueba elementales entre los relojes y las constantes), tal como fue hecho en la sección 4.4 para el sistema TCG. Esta es una clara ventaja del enfoque deductivo utilizado que está condicionada respecto a la utilización de un MC. Para el sistema TCG todas las pruebas se hicieron exclusivamente en el ambiente de pruebas *Coq* y fue posible una generalización de la especificación del problema que preserva todas las propiedades analizadas. Si ciertas pruebas se hicieran con el auxilio de un MC –para determinados valores constantes de los parámetros–, la generalización completa no sería posible. Sin embargo, en función de las propiedades demostradas en el ambiente de pruebas podría ensayarse una generalización que especifique un superconjunto de valores posibles para los parámetros, que hagan verdaderas a estas propiedades. Siendo un subconjunto de éstos los valores que hacen verdaderas a todas las propiedades del sistema, incluyendo a las demostradas con el MC. Luego, al elegir valores constantes para los parámetros del sistema distintos a los usados para la verificación de las propiedades con el MC, deberían demostrarse todas estas propiedades nuevamente con el MC, pero no las demostradas con el asistente de pruebas –los nuevos valores constantes serían seleccionados a partir de las posibles instancias de los parámetros que satisfacen las propiedades demostradas deductivamente.

Finalmente, una etapa no abordada en este trabajo corresponde a la síntesis de programas, que creemos es una línea promisoria a seguir y en la cual un ambiente de

teoría de tipos como *Coq* es particularmente adecuado, ya que permite expresar pruebas y programas en una teoría unificada. El sistema *Coq* guía a un usuario en forma interactiva en el proceso de construcción de las pruebas y los programas, siendo verificada inmediatamente la validez de cada paso del desarrollo.

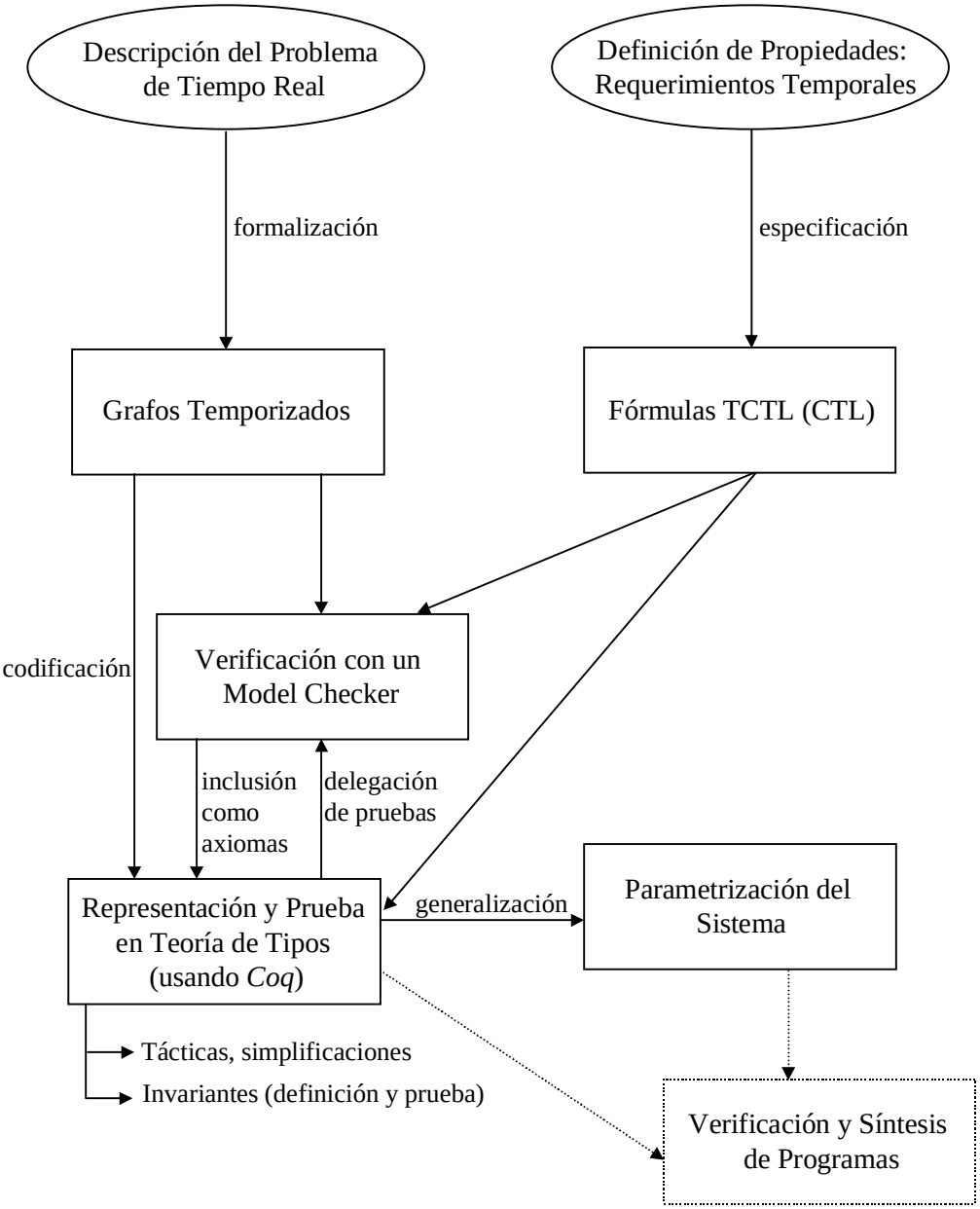

Fig. 6.1: Una metodología de trabajo

6.2 Trabajos Relacionados

En [Lun98] el autor compara algunas de las distintas opciones para la especificación, el análisis y la automatización de sistemas que involucran nociones temporales, tales como los autómatas (grafos) temporizados, las redes de Petri con tiempo y algunas lógicas temporales. Se describe una discretización de un dominio temporal continuo y un particular interés es puesto en la concepción de versiones algorítmicas de modelos temporizados de autómatas y redes de Petri, a partir de la utilización de transformadores de predicados. Se analizan variaciones y extensiones temporales –para sistemas restringidos– del *wp* (weakest precondition) propuesto por Dijkstra.

En la sección 4.5 analizamos algunos trabajos relacionados al marco de especificación y análisis de sistemas de tiempo real en torno al caso de estudio abordado en dicho capítulo. Respecto a la utilización –exclusiva– de un *model checker* en la verificación, nuestro enfoque permite trabajar con parámetros variables y suma las ventajas de los métodos deductivos de análisis, descriptas sucintamente en la sección 1.3. En la sección 4.5 analizamos también las diferencias con otros otros modelos de especificación y análisis deductivos (por ejemplo, PVS [Sha93]). Nosotros pensamos que la utilización de relojes en el modelo de especificación y la disponibilidad de un mecanismo de definición composicional facilitan el proceso de definición de los sistemas, su comprensión (aún por personas no expertas) y también su análisis, ya que resulta bastante fácil y natural la formulación de propiedades. Asimismo, los grafos temporizados tienen una interpretación algorítmica que permite simular el sistema con el control de los múltiples relojes. Creemos que esta formalización, además de permitirnos usar un *model checker* automático como asistente para la demostración de ciertas propiedades, es adecuada para la síntesis de programas, en el marco de teoría de tipos y en particular del cálculo de construcciones inductivas y co-inductivas de *Coq*.

Algunos otros trabajos que buscan relacionar métodos deductivos de prueba y *model checking* son: [MN95, RSS95, HS96, BMS+97, SS99, KKP99, Gim99]. En particular, [Gim99] analiza posibles combinaciones, en *Coq*, de métodos de demostración asistida de programas y *model checking* para la verificación de programas concurrentes sobre memorias compartidas. Sin embargo, el *tiempo* no es considerado un parámetro relevante en la clase de problemas abordados.

6.3 Trabajos Futuros

Como señalamos al comienzo de este trabajo, el objetivo del mismo era dar un pequeño paso en una combinación posible entre dos metodologías de análisis de sistemas: el enfoque deductivo, basado en ambientes de prueba, y la verificación de modelos. A partir de esta experiencia –en torno al ambiente de pruebas *Coq*– surgen algunas líneas promisorias a seguir. Algunos trabajos futuros son:

- Extender las bibliotecas de operadores temporales de CTL y TCTL con más propiedades acerca de los mismos, con el objetivo de facilitar la demostración de propiedades sobre los sistemas.
- Desarrollar tácticas generales que permitan simplificar la prueba de propiedades sobre grafos temporizados obtenidos como instancias de las representaciones genéricas.
- Analizar la construcción y corrección, en teoría de tipos, de *abstracciones* de sistemas a fin de demostrar ciertas propiedades sobre sistemas reducidos –más abstractos–, que puedan ser posteriormente generalizadas a los sistemas originales. Un trabajo relacionado en esta dirección es [MN95]. Una abstracción permite particionar el espacio de estados para obtener un sistema –autómata– más pequeño, el cuál pueda ser más tratable (en particular, por un *model checker*). La construcción formal de abstracciones constituiría una etapa en la metodología de trabajo propuesta en la sección previa para el análisis de *ciertas* propiedades. Estas propiedades se probarían sobre una versión simplificada de un sistema, pero serían válidas para el sistema original, de acuerdo a la corrección de la abstracción respecto de las propiedades analizadas, que sería demostrada a través del asistente de pruebas.
- Extender el lenguaje de descripción de sistemas de tiempo real con estructuras de datos y en particular aquellas con dominios infinitos. Asimismo, permitir trabajar con grafos temporizados más generales que consideren asignaciones a los relojes de valores no necesariamente constantes. En ambas extensiones los *model checkers* no pueden aplicarse directamente y consecuentemente una metodología de análisis que incorpore un asistente de pruebas, como la expuesta en la sección 6.1, resulta adecuada.
- Estudiar la posibilidad de realizar síntesis de programas de tiempo real en teoría de tipos.

Respecto a la última propuesta, algunas ideas surgen a partir del trabajo [Gim99]. Una noción –primitiva– de programa de tiempo real podría concebirse, inicialmente, como una función que dada una serie (un *stream*) de *eventos* construye una traza de ejecución del sistema. Sin embargo, a diferencia de [Gim99], no toda posible serie de eventos produce una traza válida de ejecución del sistema, ya que el tiempo aquí es un parámetro a considerar. Es decir, la función que dado un estado del sistema y un evento devuelve el nuevo estado del sistema, no es una función total. Podríamos entonces considerar que el tipo de los estados (`State`) incluye un estado distinguido de error (`ErrState`), a partir del cual el sistema es incorrecto. La función que construye las trazas de ejecución –en realidad, *streams* de tipo `State`, donde se incluyen posibes trazas erroneas– sería entonces una función co-recursiva (`mkTrace`) que recibe un estado y una secuencia de eventos. `mkTrace` es en realidad un simulador del sistema. Esto es,

```
Definition Trace  := (Stream State).

Definition Events := (Stream Event).

Parameter NexState : State -> Event -> State

CoFixpoint mkTrace [s:State; evs:Events]: Trace :=
  (Cons s (mkTrace (NextState s (hd evs))) (tl evs)).
```

Donde, `Event` es el tipo de los eventos (para sistemas determinísticos éste podría ser el tipo `Label`) y `NextState` es una función que implementa la relación de transición. Dado un estado y un evento, `NextState` devuelve el nuevo estado del sistema, siendo éste `ErrState` cuando el estado del sistema no es válido para el evento dado o cuando es precisamente el estado de entrada (propagación del estado de error).

Las propiedades del sistema se demuestran para las trazas *válidas*, esto es, aquellas generadas a partir de secuencias de eventos que no generan en ningún punto el estado `ErrState`.

Referencias Bibliográficas

[AB96] J. Armstrong and L. Barroca. "Specification and verification of reactive systems behaviour: The railroad crossing example". *Real-Time Systems*, 10:143-178, 1996.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. "Model-checking for real-time systems". In *Proc. 5th Symp on Logics in Computer Science*, pages 414-425. IEEE Computer Society Press, 1990.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Oliver, J. Sifakis, and S. Yovine. "The algorithmic analysis of hybrid systems". *Theorical Computer Science*, 138:3-34, 1995.

[AD94] R. Alur and D. Dill. "A theory of timed automata". *Theorical Computer Science*, 126:183-235, 1994.

[AH91] R. Alur and T. Henzinger. "Time for logic". *SIGACT News*, 22(3), 1991.

[AH92] R. Alur and T. Henzinger. "Logics and models of real time: A survey". In J. W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time Theory in Practice*, *LNCS* 600, pages 74-106. Springer-Verlag, 1992.

[AH93] R. Alur and T. Henzinger. "Real-time logics: Complexity and expressiveness". *Information and Computation*, 104(1):35-77, 1993.

[AH94] R. Alur and T. Henzinger. "A Really Temporal Logic". *Journal of the ACM*, 41(1):181-204, 1994.

[AH97] R. Alur and T. Henzinger. "Real-time system = discrete system + clock variables". *Software Tools for Technology Transfer*, 1:86-109, 1997.

[Alu91] R. Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Departament of Computer Science, Stanford University, 1991.

[AMP98] A. Asarin, O Maler, and A. Pnueli. "On the discretization of delays in timed automata and digital circuits". In R. de Simone and D. Sangiorgi (Eds.), *Proc. Concur'98, LNCS* 1466, pages 470-484, Springer-Verlag, 1998.

[AS85] B. Alpern and F. Schneider. "Defining liveness". *Information Processing Letters*, 21(4):181-185, 1985.

[Bar+99] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, Ch. Murthy, C. Parent-Vigouroux, P. Loiseleur, Ch. Paulin-Mohring, A. Saïbi, and B. Werner. "The Coq Proof Assistant. Reference Manual, Versión 6.2.4". *INRIA*, 1999.

[BB94] S. Badaloni and M. Berati. "Dealing with time granularity". Tecnical Report, Departamento de Informática – Universidad de Jaume – Castellon – Technical Report DI 06-11/95, 1995.

[BBM97] N. S. Bjørner, A. Browne, and Z. Manna. "Automatic generation of invariants and intermediate assertions". *Theorical Computer Science*, 173(1):49-87, 1997.

[Ben91] J. F. A. K. Van Benthem. *The Logic of Time (second edition)*. Reidel, 1991.

[BL99] S. Bensalem and Y. Lakhench. “Automatic generation of invariants”. Accepted in *Formal Methods in System Design*. To appear.

[BM97] F. Barber and S. Moreno. “Representation of continuos change with discrete time”. IEEE pages 175-179, 1997.

[BMS+97] N. Bjørner, Z. Manna, H. Spima, and T. Uribe. “Deductive Verification of Real-time Systems Using SteP”. *ARTS-97*, vol. 1231 of LNCS, pp. 22-43, Springer-Verlag, 1997.

[BMT99] M. Bozga, O. Maler, and S. Tripakis. “Efficient verification of timed automata using dense and discrete time semantics”. In L. Pierre and T. Kropf (Eds.), *Proc CHARME’99*, Springer-Verlag, 1999.

[Bur89] J. Burch. “Combining CTL, trace theory and timing models”, *Automatic Verification Methods for Finite State Systems, LNCS* 407, 1989.

[CES86] E. Clarke, E. Emerson, and A. Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications”. *ACM Transactions on Programming Languages and Systems*, 8(2):244-263, 1986.

[CF58] H. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[CH85] T. Coquand and G. Huet. “Constructions: A higher order proof system for mechanizing mathematics”. In *EUROCALL´85, LNCS* 203, Linz, 1985, Springer-Verlag.

[CH88] T. Coquand and G. Huet. “The calculus of constructions”. *Information and Computation*, 76(2/3), 1988.

[CHR92] Z. Chaochen, C. Hoare, and A. Ravn. “A calculus of durations”. *Information Processing Letters*, 40(5):269-276,1992.

[ChS95] K. Chandy and B. Sanders. “Predicate transformers for reasoning about concurrent computation”. *Science of Computer programming*, 24:129-148, 1995.

[Coq86] T. Coquand. “Metamathematical investigations of a calculus of constructions”. INRIA and Cambridge, University, 1986.

[Coq93] T. Coquand. “Infinite objects in type theory”. In H. Barendregt and T. Nipkow, editors, *Workshop on Types for Proofs and Programs*, number 806 in LNCS, pages 62-78. Springer-Verlag, 1993.

[Dij79] E. Dijkstra, “A Discipline of Programming”. *Prentice-Hall, Englewood Cliffs*, NJ, 1976.

[DOT+96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. “The tool Kronos”. In *Hybrid Systems III, Verification and Control, LNCS* 1066, 1996.

[DOY96] C. Draws, A. Olivero, and S. Yovine. “Verificación automática de sistemas temporizados utilizando KRONOS”. Jornadas de Informática y telecomunicaciones, IEEE Uruguay, 1996.

[DS97] R. Dijkstra and B. Sanders. “A predicate transformer for the progress property to-always”. *Formal Aspects of Computing*, 9:270-282, 1997.

[DY95] C. Daws and S. Yovine. "Verification of multirate timed automata with KRONOS: two exemples". Technical Report Spectre-95-06, VERIMAG, 1995.

[EC82] E. Amerson and E. Clarke. "Using branching-time temporal logic to synthesize synchronization skeletons". *Science of Computer Programming*, 2:241-266,1982.

[Eme95] E. Emerson. "Automated temporal reasoning about rective systems". In *Logics for Concurrency*. Springer-Verlag, 1995.

[EMS+89] E. Emerson, A. Mok, A. Sistla, and J. Srinivasan. "Quantitative temporal reasoning". *Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.

[Euz95] J. Euzenat. "An Algebraic approach to the granularity in time representation". In *Proceedings of Time 95*, 1995.

[Gim95] E. Giménez. "An application of Co-inductive Types in Coq: Verification of the Alternating Bit Protocol". In *BRA Workshop on Types for Proofs and Programs* (*TYPES'95*), LNCS 1158, pages 135-152, Springer-Verlag, 1995.

[Gim96] E. Giménez. *A Calculus of Infinite Constructions and its application to the verification of communicating systems.* PhD thesis, Ecole Normale Supérieure de Lyon, 1996, Unité de Recherche Associée au CNRS No. 1398, 1996.

[Gim98] E. Giménez. *A tutorial on recursive types in Coq*, Technical Report 0221, INRIA, 1998.

[Gim99] E. Giménez. "Two Approaches to the Verification of Concurrent Programs in Coq". To appear, 1999.

[Gor93] M. Gordon. *Introduction to HOL: a theorem proving environment based for higher order logic*. Cambridge University, Press, 1993.

[GPV94] A. Göllü, A. Puri, and P. Varaiya. "Discretization of timed automata". *Proc. 33rd CDC*, Orlando, Florida, 1994.

[Gri81] D. Gries. *The science of programming*, Springer-Verlag New York Inc., 1981.

[HHW+97] T. Henzinger, P.-H. Ho, and H. Wong-Toi. "Hytech: a model checker for hybrid systems". *Software Tools for Technology Transfer*, 1997.

[HJL93] C. Heitmeyer, R. Jeffords, and B. Labaw. "A benchmark for comparing different approaches for specifying real-time systems". *Real Time: Theory and Practice*, *LNCS* 600, REX Workshop, Mook, The Netherlands, 1991. Springer-Verlag.

[HK94] T. Henzinger and O. Kopke. "Verification methods for the divergent runs of clock systems". In *FTRTFT'94: Formal Techniques in Real-time and Fault-tolerant Systems*, volume 863 of *LNCS*, pages 351-372. Springer-Verlag, 1994.

[HK97] T. Henzinger and O. Kupferman. "From qunatity to quality". In *Proc. of the First International Workshop on Hybrid and Real-Time Systems, HART'97, LNCS* 1201, pages 48-62, Springer-Verlag, 1997.

[HKP96] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq proof assistant version 6.1, A tutorial*, 1996.

[HKV96] T. Henzinger, O Kupferman, and M. Vardi. "A space-efficient on-the-fly algorithm for real-time model checking". In *Proc 7th Conference on Concurrency Theory, LNCS* 1119, pages 514-529, Springer-Verlag, 1996.

[HLP90] E. Harel, O. Lichtentstein, and A. Pnuelli. "Explicit clock temporal logic". In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 402-413, 1990.

[HMP92] T. Henzinger, Z. Manna, and A. Pnueli. "What good are digital clocks?". In *W. Kuich, editor, ICALP 92: Automata, Languages and Programming, LNCS* 623, pages 545-558. Springer-Verlag, 1992.

[HMP94] T. Henzinger, Z. Manna, and A. Pnuelli. "Temporal Proof Methodologies for Timed Transition Systems". *Information and Computation* 112, pp. 273-337, 1994.

[HNS+92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. "Symbolic model-checking for real-time systems". In *Proc. 7th Symp on Logics in Computer Science*. IEEE Computer Society Press, 1992.

[Hob85] J. Hobbs. "Granularity". In *Proceedings of the IJCAI 85*, pages 432-435. IJCAI, 1985.

[How80] W. Howard. "The formulae-as-types notion of construction". In J. Seldin and J. Hindley, editors, *To H. Curry: Essays on combinatory logic, lambda calculus and formalism.* Academic Press, 1980. A reprint of an unpublished manuscript from 1969.

[HS96] K. Havelund and N. Shankar. "Experiments in Theorem Proving Model Checking for Protocol Verification". In *proceedings of FME'96*, Oxford. *LNCS* 1051, pages 662-681, 1996.

[JKR89] C. Julta, E. Knapp, and J. Rao. "A Predicate transformer approach to semantics of parallel programs". In *Proceeding of the 8th ACM Symposium on Principles of Distributed Computing*, 1989.

[KKP99] Y. Kesten, A. Klein, A. Pnueli, and G. Raanan. "A Perfecto Verification: combining model checking with deductive analysis to verify real-life software". In *FM'99*, Toulouse, France. *LNCS* 1709, pages 173-194, 1999.

[Kna90] E. Knapp. "A predicate transformer for progress". *Information Processing Letters*, 33, 1990.

[Lam90] L. Lamport. "*win* and *sin*: predicate transformers for concurrency". *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[LL95] F. Laroussinie and K. Larsen. "Compositional model checking of real time systems". In *Proc. 6th Conference on Concurrency Theory*, pages 27-41, Philadelphia, Springer-Verlag, 1995.

[LP92] Z. Luo and R. Pollack. "Lego proof development system: User's manual". *Technical Report* ECS-LFCS-92-211, LFCS, 1992.

[LPY97] K. Larsen, P. Pettersson, and W. Yi. "Uppaal in a nutshell". *Software Tools for Technology Transfer*, 1997.

[LS85] N. Leveson and J. Stolzy. “Analyzing safety and fault tolerance using timed Petri nets”. In *Proc. Internat. Joint Conf. On Theory and Practice of Software Development, LNCS* 186, pages 339-355, Springer-Verlag, 1985.

[Lun98] C. Luna. “Sistemas Temporizados: Especificación, análisis y verificación. Estudio comparativo”. Reporte técnico 9806, InCo, Fac. de Ingeniería, U. de la República, Montevideo, Uruguay. 1998.

[Mag94] L. Magnusson. *The implementation of ALF – a proof editor based on Matin Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Göteborg, 1994.

[May97] M. Mayero, “Le théorème des trois intervalles spécification et preuves en Coq”. *Rapport du DEA sémantique preuves et programmation.* Université Paris 6, 1997

[MGJ91] D. Mandrioli, Carlo Ghezzi, and Mehdi Jazayeri. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[MN95] Olaf Müller and T. Nipkow. “Combining Model Checking and Deduction for I/O-Automata”. In *Tools and Algorithms for the Construction and Analysis of Systems, LNCS* 1019, pages 1-16, 1995.

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

[MP91] Z. Manna and A. Pnueli. “Completing the temporal picture”. In *Theoretical Computer Science*, 83(1):97-130, 1991.

[MP92] Z. Manna and A. Pnueli. “The temporal logic of reactive and concurrent systems: specification”. *Springer-Verlag*, Berlin, 1992.

[MP95] Z. Manna and A. Pnueli. “Temporal verification of reactive systems: Safety”, *Springer-Verlag*, 1995.

[Oli94] A. Olivero. *Modélisation et Analyse de Systèmes Temporisés et Hybrides*. PhD thesis, Institut National Polytéchnique de Grenoble. France, 1994.

[ORS92] S. Owre, J. Rushby, and N. Shankar. “PVS: A prototype verification system”. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE). LNIA* 607, Saratoga, NY, 1992. Springer Verlag.

[Ost87] J. Ostroff. *Temporal logic of real-time systems*, Ph.D. thesis, Univ. of Toronto, 1987.

[Pau93a] L. Paulson. “Co-induction and Co-recursion in Higher-order Logic”. *Technical Report* 304, Computer Laboratory, University of Cambridge, 1993.

[Pau93b] L. Paulson. “The Isabelle reference manual”. *Technical Report* 283, Computer Laboratory, University, 1993.

[PL92] A. Pnueli and L. Lamport. “An old-fashioned recipe for real-time”. In J. W. De Baker, K. Huizing, W. P. De Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, *LNCS* 600, Springer-Verlag, 1992.

[P-M89a] C. Paulin-Mohring. "Extracting F_ω's programs from proofs in the calculus of constructions". In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, 1989. ACM.

[P-M89b] C. Paulin-Mohring. *Extraction de programmes dans le calcul des constructions*. Thèse de doctorat, Université de Paris VII, 1989.

[P-M93] C. Paulin-Mohring. "Inductive definitions in the system Coq – rules and properties". In M. Bezem and J. Groote, editors, *Proceeedings of the conference Typed Lambda Calculi and Aplications*, *LNCS* 664, 1993.

[Pnu81] A. Pnueli. "The temporal logic of programs". *Teorical Computer Science*, 1981.

[Pnu85] A. Pnueli. "Linear and branching structures in the semantics and logics of reactive systems". In *Proc. 12th ICALP, Nafplion, LNCS 194*, 1985.

[RSS95] S. Rajan, N. Shankar, and M. Srivas. "An integration of model checking with automated proof checking". In *Computer-Aided Verification, CAV'95*. *LNCS* 939, Belgium, 1995.

[Sha93] N. Shankar. "Verification of real-time systems using PVS". *CAV'93*, Costas Courcoubetis, editor, Elounda, Greece, *LNCS* 697, pages 280-291, 1993.

[SS95] O. Sokolsky and S. Smolka. "Local model checking for real-time systems. In *Computer Aided Verification, Proc. 7th Int. Workshop*, Liege, *LNCS* 939, pages 211-224, 1995.

[SS99] H. Saïdi and N. Shankar. "Abstract and model Check while you prove". In *CAV'99*, Trento, Italy, 1999.

[WH95] H. Wong-Toi and P. Ho. "Automated analysis of an audio control protocol". In *Proc. in Computer Aided Verification*, 1995.

[WL99] P. Weis et X. Leroy. "Le Langage CAML". Second edition, Dunod, Paris, 1999. First edition, InterEditions, Paris, 1993.

[Yov93] S. Yovine, *Méthodes et outils pour la vérification symbolique de systèmes temporisés*. PhD thesis, Institut National Polytechnique de Grenoble, France, 1993.

[Yov97] S. Yovine. "Kronos: A verification tool for real-time systems". *Software Tools for Technology Transfer*, 1997.

Apéndice A

Nociones Temporales

time_clocks.v (disponible en **http://www.fing.edu.uy/~cluna**)

Versión de *Coq*: 6.2.2. Archivo generado con el utilitario *coq2html*.

Require Export Arith.

Section Time_Clocks.

```
(* Nociones temporales para tiempo discreto *)
```

Definition Instant := nat.
Definition Clock := nat.

Definition lt_Ck := lt. (* < *)
Definition le_Ck := le. (* ≤ *)
Definition gt_Ck := gt. (* > *)
Definition ge_Ck := ge. (* ≥ *)
Definition eq_Ck := [x,y:Clock] x=y. (* = *)

Definition Ini_Ck : Instant := O.
Definition tick : Instant := (1).
Definition plus_Ck := plus. (* + *)
Definition Inc := [x:Clock] (plus_Ck x tick).
Definition Reset : Instant := O.
Definition time0 : Instant := O.

End Time_Clocks.

Apéndice B

Operadores Temporales: Definiciones y Propiedades

TemporalOperators.v (disponible en **http://www.fing.edu.uy/~cluna**)

Versión de *Coq*: 6.2.2. Archivo generado con el utilitario *coq2html*.

B.1 Invarianza y Alcanzabilidad (con Tipos Inductivos)

```
Implicit Arguments On.

Require time_clocks.  (* nociones temporales para tiempo discreto *)

(* Las formalizaciones asumen declarado un tipo Label con una etiqueta distinguida Tick  *)
(* Inductive Label : Set := L1 : Label | ... | Ln : Label | Tick : Label.                 *)
```

B.1.1 Definiciones

```
Section TemporalOperators_Ind.


 Variable S  : Set.                    (* tipo de los estados         *)
 Variable tr : S->Label->S->Prop.      (* transiciones del sistema    *)

 (* Estados alcanzables desde el estado "Sini" con transiciones "tr" *)

 Inductive RState [Sini:S]: S -> Prop :=
    rsIni    : (RState Sini Sini)
  | rsNext   : (s1,s2:S) (l:Label) (RState Sini s1) -> (tr s1 l s2) ->
               (RState Sini s2).


 (* Estados alcanzables en unidades de tiempo desde el estado "Sini" con transiciones "tr" *)

 Inductive RState_T [Sini:S]: S -> Instant -> Prop :=
    rsIni_T       : (RState_T Sini Sini time0)
  | rsTime_T      : (s1,s2:S) (t:Instant) (RState_T Sini s1 t) -> (tr s1 Tick s2) ->
                    (RState_T Sini s2 (Inc t))
  | rsNoTime_T    : (s1,s2:S) (l:Label) (t:Instant) (RState_T Sini s1 t) -> ~(l=Tick) -> (tr s1 l s2) ->
                    (RState_T Sini s2 t).
```

```
(* Init => ∀□ P *)

Definition ForAll := [Sini:S] [P:S->Prop]
  (s:S) (RState Sini s) -> (P s).


(* Init => ∀□(bound t) P *)

Definition ForAll_T := [Sini:S] [P:S->Prop] [bound:Instant->Prop]
  (s:S) (t:Instant) (bound t) -> (RState_T Sini s t) -> (P s).


(* Init => ∃◊ P *)

Inductive Exists [Sini:S; P:S->Prop] : Prop :=
  exists : (s:S) (RState Sini s) -> (P s) -> (Exists Sini P).


(* Init => ∃◊ (bound t) P *)

Inductive Exists_T [Sini:S; P:S->Prop; bound:Instant->Prop] : Prop :=
  exists_T : (s:S) (t:Instant) (bound t) -> (RState_T Sini s t) -> (P s) -> (Exists_T Sini P bound).
```

B.1.2 Propiedades

```
Theorem Mon_I : (Sini:S) (Pg,Pp:(S->Prop))
          (ForAll Sini Pg) -> ((s:S) (Pg s) -> (Pp s)) -> (ForAll Sini Pp).
Proof.
  Unfold ForAll; Intros.
  Apply H0.
  Apply H; Assumption.
Qed.


Theorem Mon_I_T : (Sini:S) (Pg,Pp:(S->Prop)) (bound:Instant->Prop)
          (ForAll_T Sini Pg bound) -> ((s:S) (Pg s) -> (Pp s)) -> (ForAll_T Sini Pp bound).
Proof.
  Unfold ForAll_T; Intros.
  Apply H0.
  Apply (H s t); Assumption.
Qed.


Theorem Conj : (Sini:S) (P1,P2:(S->Prop))
          (ForAll Sini P1) -> (ForAll Sini P2) -> (ForAll Sini ([s:S] (P1 s)/\(P2 s))).
Proof.
  Unfold ForAll; Intros.
  Split; [ Apply H | Apply H0 ]; Assumption.
Qed.
```

```
Theorem Conj_T : (Sini:S) (P1,P2:(S->Prop)) (bound:Instant->Prop)
          (ForAll_T Sini P1 bound) -> (ForAll_T Sini P2 bound) ->
          (ForAll_T Sini ([s:S] (P1 s)/\(P2 s)) bound).
Proof.
  Unfold ForAll_T; Intros.
  Split; [ Apply (H s t) | Apply (H0 s t) ]; Assumption.
Qed.

Theorem Mon_I_EX : (Sini:S) (Pg,Pp:(S->Prop))
          (Exists Sini Pg) -> ((s:S) (Pg s) -> (Pp s)) -> (Exists Sini Pp).
Proof.
  Intros.
  Inversion_clear H.
  Apply (exists H1 (H0 s H2)).
Qed.

Theorem Mon_I_EX_T : (Sini:S) (Pg,Pp:(S->Prop)) (bound:Instant->Prop)
          (Exists_T Sini Pg bound) -> ((s:S) (Pg s) -> (Pp s)) -> (Exists_T Sini Pp bound).
Proof.
  Intros.
  Inversion_clear H.
  Apply (exists_T H1 H2 (H0 s H3)).
Qed.

Lemma RState_Trans : (s1,s2,s3:S) (RState s1 s2) -> (RState s2 s3) -> (RState s1 s3).
Proof.
  Intros.
  Elim H0; [ Assumption | Intros ].
  Apply (rsNext H2 H3).
Qed.

Lemma RState_Trans_T : (s1,s2,s3:S) (t1,t2:Instant)
          (RState_T s1 s2 t1) -> (RState_T s2 s3 t2) -> (RState_T s1 s3 (plus_Ck t1 t2)).
Proof.
  Induction 2; Unfold plus_Ck; Intros.
  Rewrite (plus_sym t1 time0); Unfold time0; Simpl; Assumption.
  Unfold Inc; Unfold plus_Ck; Rewrite (plus_assoc_l t1 t tick).
  Apply (rsTime_T H2 H3).
  Apply (rsNoTime_T H2 H3 H4).
Qed.

Theorem StepsEX : (s1,s2:S) (P:S->Prop)
          (RState s1 s2) -> (Exists s2 P) -> (Exists s1 P).
Proof.
  Intros.
  Inversion H0.
  Apply (exists (RState_Trans H H1) H2).
Qed.

Require Classical.
```

```
Theorem ForAll_EX : (Sini:S) (P:S->Prop)
          (ForAll Sini P) <-> ~(Exists Sini ([s:S]~(P s))).
Proof.
  Unfold not; Unfold ForAll; Red; Intros; Split; Intros.
  Inversion H0.
  Apply (H2 (H s H1)).
  Elim (classic (P s)); [ Trivial | Intro; Absurd (Exists Sini [s:S]~(P s)) ].
  Assumption.
  Apply exists with 1:=H0; Assumption.
Qed.


Theorem ForAll_EX_T: (Sini:S) (P:S->Prop) (bound:Instant->Prop)
          (ForAll_T Sini P bound) <-> ~(Exists_T Sini ([s:S]~(P s)) bound).
Proof.
  Unfold not; Unfold ForAll_T; Red; Intros; Split; Intros.
  Inversion H0.
  Apply (H3 (H s t H1 H2)).
  Elim (classic (P s)); [ Trivial | (Intro; Absurd (Exists_T Sini [s:S]~(P s) bound)) ].
  Assumption.
  Apply exists_T with 2:= H1; Assumption.
Qed.
```

B.1.3 Definiciones Complementarias

```
(* Q => ∀□ P *)

Definition ForAll_from := [Sini:S] [Q,P:S->Prop]
  (ForAll Sini ([s:S](Q s)->(ForAll s P))).
  (* (s:S) (RState Sini s) -> (Q s) -> (ForAll s P). *)


(* Q => ∀□(bound t) P *)

Definition ForAll_from_T := [Sini:S] [Q,P:S->Prop] [bound:Instant->Prop]
  (ForAll Sini ([s:S](Q s)->(ForAll_T s P bound))).
  (* (s:S) (RState Sini s) -> (Q s) -> (ForAll_T s P bound). *)


(* Q => ∃◊ P *)

Definition Exists_from := [Sini:S] [Q,P:S->Prop]
  (ForAll Sini ([s:S](Q s)->(Exists s P))).
  (* (s:S) (RState Sini s) -> (Q s) -> (Exists s P). *)


(* Q => ∃◊(bound t) P *)

Definition Exists_from_T := [Sini:S] [Q,P:S->Prop] [bound:Instant->Prop]
  (ForAll Sini ([s:S](Q s)->(Exists_T s P bound))).
  (* (s:S) (RState Sini s) -> (Q s) -> (Exists_T s P bound). *)


End TemporalOperators_Ind.
```

B.2 Operadores CTL (con Tipos Co-Inductivos)

B.2.1 Definiciones

```
Infix RIGHTA 9 "^" Cons.

Section TemporalOperators_CTL.

 Variable S : Set.                    (* tipo de los estados        *)
 Variable tr : S->Label->S->Prop.     (* transiciones del sistema   *)


 Syntactic Definition SPath := (Stream S).


 CoInductive ForAllS [P:SPath->Prop] : SPath->Prop :=
  forallS : (x:SPath) (s:S) (P s^x) -> (ForAllS P x) -> (ForAllS P s^x).

 Inductive ExistsS [P:SPath->Prop] : SPath->Prop:=
    Here    :  (x:SPath) (P x) -> (ExistsS P x)
  | Further :  (x:SPath) (s:S) (ExistsS P x) -> (ExistsS P s^x).


(* Trazas de ejecución *)

 CoInductive isTrace : SPath->Prop :=
  is_trace : (x:SPath) (s1,s2:S) (l:Label) (tr s1 l s2) -> (isTrace s2^x) -> (isTrace s1^s2^x).

 Definition isTraceFrom := [Sini:S] [x:SPath]
  Sini=(hd x) /\ (isTrace x).


(* Definición del operador Until *)

 Inductive Until [P,Q:SPath->Prop] : SPath -> Prop :=
    UntilFurther  :  (s:S) (x:SPath) (P s^x) -> (Until P Q x) -> (Until P Q s^x)
  | UntilHere     :  (x:SPath) (Q x) -> (Until P Q x).

 Inductive EX_Until [Sini:S; P,Q:SPath->Prop] : Prop :=
  ExUntil : (x:SPath) (isTraceFrom Sini x) -> (Until P Q x) -> (EX_Until Sini P Q).

 Definition FA_Until := [Sini:S] [P,Q:SPath->Prop]
  (x:SPath) (isTraceFrom Sini x) -> (Until P Q x).


(* Init => ∀□ P *)

 Definition Always := [Sini:S] [P:SPath->Prop]
  (x:SPath) (isTraceFrom Sini x) -> (ForAllS P x).


(* Init => ∀◊ P *)

 Definition Inevitable := [Sini:S] [P:SPath->Prop]
  (x:SPath) (isTraceFrom Sini x) -> (ExistsS P x).
```

```
(* Init => ∃◊ P *)

Inductive Posible [Sini:S; P:SPath->Prop]: Prop :=
  posible : (x:SPath) (isTraceFrom Sini x) -> (ExistsS P x) -> (Posible Sini P).


(* Init => ∀∃□ P *)

Inductive SafePath [Sini:S; P:SPath->Prop] : Prop :=
  safePath : (x:SPath) (isTraceFrom Sini x) -> (ForAllS P x) -> (SafePath Sini P).
```

B.2.2 Propiedades

```
Theorem Equiv1 : (Sini:S) (P:SPath->Prop)
          (Posible Sini P) <-> (EX_Until Sini ([_:SPath] True) P).
Proof.
  Unfold iff; Intros; Split; Intro.
  Inversion_clear H.
  Apply ExUntil with P:=[_:SPath]True 1:=H0.
  Elim H1; Intros.
  Constructor 2; Assumption.
  Constructor 1; Trivial.
  Inversion_clear H.
  Apply posible with 1:=H0.
  Elim H1; Intros.
  Constructor 2; Assumption.
  Constructor 1; Assumption.
Qed.


Theorem Equiv2 : (Sini:S) (P:SPath->Prop)
          (Inevitable Sini P) <-> (FA_Until Sini ([_:SPath] True) P).
Proof.
  Unfold iff Inevitable FA_Until; Intros; Split; Intros.
  Elim (H x H0); Intros.
  Constructor 2; Assumption.
  Constructor 1; Trivial.
  Elim (H x H0); Intros.
  Constructor 2; Assumption.
  Constructor 1; Assumption.
Qed.


Lemma ConsTrace : (s1,s2:S) (x,z:SPath)
          (isTraceFrom s2 z) -> (isTraceFrom s1 s1^s2^x) -> (isTraceFrom s1 s1^z).
Proof.
  Unfold isTraceFrom; Simpl.
  Destruct z; Destruct 1; Destruct 3; Simpl; Intros.
  Compute in H0; Rewrite H0 in H4.
  Inversion_clear H4.
  Split; [ Trivial | Apply (is_trace H5 H1) ].
Qed.
```

```
Lemma notPosible : (P:SPath->Prop) (s1:S) ~(Posible s1 P) ->
          (z:SPath) (s2:S) (isTraceFrom s1 s1^s2^z) -> ~(Posible s2 P).
Proof.
  Unfold 2 not; Intros.
  Elim H1; Intros.
  Apply H; Cut (isTraceFrom s1 (Cons s1 x)).
  Intro H4; Apply (posible 2!P H4).
  Apply Further; Assumption.
  Apply ConsTrace with 1:=H2 2:=H0; Assumption.
Qed.


Theorem Equiv3 : (Sini:S) (P:SPath->Prop)
          (Always Sini P) <-> ~(Posible Sini ([s:SPath]~(P s))).
Proof.
  Unfold iff Always not; Intros; Split.
  Intros H H0; Inversion_clear H0.
  Generalize (H x H1); Elim H2; Intros.
  Inversion_clear H3 in H0.
  Apply (H0 H4).
  Inversion_clear H4.
  Apply (H3 H6).
  Generalize Sini; Cofix u.
  Destruct x; Intros; Constructor.
  Elim (classic (P s^s0)); [ Trivial | Intros ].
  Absurd (Posible Sini0 [s:SPath]~(P s)).
  Assumption.
  Apply posible with 1:=H0.
  Constructor 1; Assumption.
  Elim H0; Simpl; Intros.
  Apply (u (hd s0)); Intros.
  Generalize H0; Clear H0; Generalize H3; Clear H3.
  Rewrite Case s0; Simpl; Intros.
  Apply (notPosible 1!([s:SPath]~(P s)) H H0); Assumption.
  Inversion_clear H2; Simpl.
  Unfold isTraceFrom; Split; Trivial.
Qed.


Lemma not_EX : (P:(Stream S)->Prop) (x:(Stream S)) (s:S)
          ~(ExistsS P (Cons s x)) -> ~(ExistsS P x).
Proof.
   Unfold not; Intros.
   Apply (H (Further s H0)).
Qed.


Theorem Equiv4 : (Sini:S) (P:SPath->Prop)
          (SafePath Sini P) <-> ~(Inevitable Sini ([s:SPath]~(P s))).
Proof.
  Unfold iff Inevitable not; Intros; Split.
  Intro sp; Inversion sp; Intros.
  Generalize H0; Elim (H1 x H); Intros.
  Inversion_clear H3 in H2.
  Apply (H2 H4).
  Apply H3; Inversion_clear H4; Assumption.
  Intro H.
  Elim (not_all_ex_not (Stream S) [x:(Stream S)] (isTraceFrom Sini x)->(ExistsS [s:SPath]~(P s) x) H).
  Intros.
```

```
  Generalize (not_imply_elim2 (isTraceFrom Sini x) (ExistsS [s:SPath]~(P s) x) H0).
  Generalize (not_imply_elim (isTraceFrom Sini x) (ExistsS [s:SPath]~(P s) x) H0); Intros.
  Apply safePath with 1:=H1.
  Generalize H1; Clear H1; Generalize H2; Clear H2.
  Generalize x; Generalize Sini; Cofix u.
  Destruct x; Intros; Constructor.
  Elim (classic (P s^s0)); [ Trivial | Intro ].
  Elim (H2 (Here 1!([s:SPath]~(P s)) H3)).
  Apply u with Sini:=(hd s0).
  Generalize H2; Clear H2; Case s0; Unfold not; Intros.
  Apply (not_EX H2 H3).
  Elim H1; Intros ig trace; Inversion_clear trace.
  Unfold isTraceFrom; Split; Trivial.
Qed.


Theorem Mon_I_S : (x:SPath) (Pg,Pp:(SPath->Prop))
          (ForAllS Pg x) -> ((s:SPath) (Pg s) -> (Pp s)) -> (ForAllS Pp x).
Proof.
  Cofix u; Intro x; Case x; Intros.
  Case H; Constructor.
  Apply (H0 s1^x0 H1).
  Apply (u x0 Pg Pp H2 H0).
Qed.


Theorem Conj_S : (x:SPath) (P1,P2:(SPath->Prop))
          (ForAllS P1 x) -> (ForAllS P2 x) -> (ForAllS ([s:SPath] (P1 s)/\ (P2 s)) x).
Proof.
  Cofix u; Intro x; Case x; Intros.
  Inversion_clear H; Inversion_clear H0.
  Constructor; [ Split | Apply (u s0) ]; Assumption.
Qed.


Theorem Mon_I_EX_S: (x:SPath) (Pg,Pp:(SPath->Prop))
          (ExistsS Pg x) -> ((s:SPath) (Pg s) -> (Pp s)) -> (ExistsS Pp x).
Proof.
  Induction 1; Intros.
  Constructor 1; Apply (H1 x0 H0).
  Constructor 2; Apply (H1 H2).
Qed.


Theorem OneStep_EX: (x:SPath) (P:SPath->Prop)
          (ExistsS P x) -> (s:S) (ExistsS P s^x).
Proof.
  Intros; Constructor 2; Assumption.
Qed.

End TemporalOperators_CTL.
```

B.3 Operadores TCTL (con Tipos Co-Inductivos)

B.3.1 Definiciones

```
Section TemporalOperators_TCTL.

 Variable S     : Set.                    (* tipo de los estados            *)
 Variable tr    : S->Label->S->Prop.      (* transiciones del sistema       *)
 Variable bound: Instant->Prop.           (* invariante de las locaciones   *)

 Syntactic Definition State_T   := S * Instant.
 Syntactic Definition SPath_T   := (Stream State_T).

 (* Trazas de ejecución *)

 CoInductive isTrace_T : SPath_T->Prop :=
    isTraceTick:  (x:SPath_T) (s1,s2:S) (t:Instant) (tr s1 Tick s2) -> (isTrace_T (s2,(Inc t))^x) ->
                  (isTrace_T ( (s1,t)^(s2,(Inc t))^x ))
  | isTraceDisc:  (x:SPath_T) (s1,s2:S) (t:Instant) (l:Label) (tr s1 l s2) -> ~l=Tick -> (isTrace_T (s2,t)^x) ->
                  (isTrace_T ( (s1,t)^(s2,t)^x )).

 Definition isTraceFrom_T := [Sini:State_T] [x:SPath_T]
  Sini=(hd x) /\ (isTrace_T x).


 (* Definición del operador Until *)

 Inductive Until_bound [P,Q:SPath_T->Prop] : SPath_T -> Prop :=
    UntilFurther_bound: (s:State_T) (x:SPath_T) (P s^x) -> (Until_bound P Q x) ->
                        (Until_bound P Q s^x)
  | UntilHere_bound   :  (s:S) (t:Instant) (x:SPath_T) (bound t) -> (Q (s,t)^x) ->
                        (Until_bound P Q (s,t)^x).

 Inductive EX_Until_bound [Sini:State_T; P,Q:SPath_T->Prop] : Prop :=
  ExUntil_bound :(x:SPath_T) (isTraceFrom_T Sini x) -> (Until_bound P Q x) ->
                 (EX_Until_bound Sini P Q).

 Definition FA_Until_bound := [Sini:State_T] [P,Q:SPath_T->Prop]
  (x:SPath_T) (isTraceFrom_T Sini x) -> (Until_bound P Q x).


 (* Init => ∀□(bound t) P *)

 Definition Always_T := [Sini:State_T] [P:SPath_T->Prop]
  (x:SPath_T) (isTraceFrom_T Sini x) -> (ForAllS ([s:SPath_T](bound (Snd (hd s)))->(P s)) x).


 (* Init => ∀◊(bound t) P *)

 Definition Inevitable_T := [Sini:State_T] [P:SPath_T->Prop]
  (x:SPath_T) (isTraceFrom_T Sini x) -> (ExistsS ([s:SPath_T](bound (Snd (hd s)))/\(P s)) x).
```

```
(* Init => ∃◊(bound t) P *)

Inductive Posible_T [Sini:State_T; P:SPath_T->Prop] : Prop :=
  posible_T : (x:SPath_T) (isTraceFrom_T Sini x) -> (ExistsS ([s:SPath_T](bound (Snd (hd s)))/\(P s)) x) ->
              (Posible_T Sini P).


(* Init => ∃□(bound t) P *)

Inductive SafePath_T [Sini:State_T; P:SPath_T->Prop] : Prop :=
  safePath_T : (x:SPath_T) (isTraceFrom_T Sini x) ->
               (ForAllS ([s:SPath_T](bound (Snd (hd s)))->(P s)) x) ->
               (SafePath_T Sini P).
```

B.3.2 Propiedades

```
Theorem Equiv1_T : (Sini:State_T) (P:SPath_T->Prop)
          (Posible_T Sini P) <-> (EX_Until_bound Sini ([_:SPath_T] True) P).
Proof.
  Unfold iff; Intros; Split; Intro.
  Inversion_clear H.
  Apply ExUntil_bound with P:=[_:SPath_T]True 1:=H0.
  Elim H1.
  Destruct x; Simpl.
  Destruct p; Destruct 1; Simpl; Intros.
  Constructor 2; Assumption.
  Destruct s; Intros.
  Constructor 1; Trivial.
  Inversion_clear H.
  Apply posible_T with 1:=H0.
  Elim H1.
  Destruct s; Intros.
  Constructor 2; Assumption.
  Intros; Constructor 1; Simpl; Split; Assumption.
Qed.


Theorem Equiv2_T : (Sini:State_T) (P:SPath_T->Prop)
          (Inevitable_T Sini P) <-> (FA_Until_bound Sini ([_:SPath_T] True) P).
Proof.
  Unfold iff Inevitable_T FA_Until_bound; Intros; Split; Intros.
  Elim (H x H0); Destruct x.
  Destruct p; Intros.
  Elim H1; Intros.
  Constructor 2; Assumption.
  Constructor 1; Trivial.
  Elim (H x H0); Intros.
  Constructor 2; Assumption.
  Constructor 1; Split; Assumption.
Qed.
```

```
Lemma ConsTrace_T : (s1,s2:State_T) (x,z:SPath_T)
        (isTraceFrom_T s2 z) -> (isTraceFrom_T s1 s1^s2^x) -> (isTraceFrom_T s1 s1^z).
Proof.
  Unfold isTraceFrom_T; Simpl.
  Destruct z; Destruct 1; Destruct 3; Simpl; Intros.
  Compute in H0; Rewrite H0 in H4.
  Inversion_clear H4 in H1.
  Split; [ Trivial | Apply (isTraceTick H5 H1) ].
  Split; [ Trivial | Apply (isTraceDisc H5 H6 H1) ].
Qed.

Lemma notPosible_T : (P:SPath_T->Prop) (s1:State_T) ~(Posible_T s1 P) ->
        (z:SPath_T) (s2:State_T) (isTraceFrom_T s1 s1^s2^z) -> ~(Posible_T s2 P).
Proof.
  Unfold 2 not; Intros.
  Elim H1; Intros.
  Apply H; Cut (isTraceFrom_T s1 (Cons s1 x)).
  Intro H4; Apply (posible_T 2!P H4).
  Apply Further; Assumption.
  Apply ConsTrace_T with 1:=H2 2:=H0; Assumption.
Qed.

Theorem Equiv3_T : (Sini:State_T) (P:SPath_T->Prop)
        (Always_T Sini P) <-> ~(Posible_T Sini ([s:SPath_T]~(P s))).
Proof.
  Unfold iff Always_T not; Intros; Split.
  Intros H H0; Inversion_clear H0.
  Generalize (H x H1); Elim H2; Intros.
  Inversion_clear H3 in H0.
  Elim H0; Intros.
  Elim (H6 (H4 H3)).
  Inversion_clear H4.
  Apply (H3 H6).
  Generalize Sini; Cofix u.
  Destruct x; Intros; Constructor.
  Intro; (Elim (classic (P p^s))); [ Trivial | Intros ].
  Absurd (Posible_T Sini0 [s:SPath_T]~(P s)).
  Assumption.
  Apply posible_T with 1:=H0.
  Constructor 1; Split; Assumption.
  Elim H0; Simpl; Intros.
  Apply (u (hd s)); Intros.
  Generalize H0; Clear H0; Generalize H3; Clear H3.
  Rewrite Case s; Simpl; Intros.
  Apply (notPosible_T 1!([s:SPath_T]~(P s)) H H0); Assumption.
  Unfold isTraceFrom_T; Inversion_clear H2; Simpl.
  Split; Trivial.
  Split; Trivial.
Qed.
```

```
Theorem Equiv4_T : (Sini:State_T) (P:SPath_T->Prop)
          (SafePath_T Sini P) <-> ~(Inevitable_T Sini ([s:SPath_T]~(P s))).
Proof.
  Unfold iff Inevitable_T not; Intros; Split.
  Intro sp; Inversion sp; Intros.
  Generalize H0; Elim (H1 x H); Intros.
  Inversion_clear H3 in H2.
  Elim H2; Intros.
  Apply (H6 (H4 H3)).
  Apply H3; Inversion_clear H4; Assumption.
  Intro H; Elim (not_all_ex_not SPath_T [x:SPath_T] (isTraceFrom_T Sini x)->
    (ExistsS [s:SPath_T](bound (Snd (hd s)))/\~(P s) x) H).
  Intros.
  Generalize (not_imply_elim2 (isTraceFrom_T Sini x)
    (ExistsS [s:SPath_T](bound (Snd (hd s)))/\~(P s) x) H0).
  Generalize (not_imply_elim (isTraceFrom_T Sini x)
    (ExistsS [s:SPath_T](bound (Snd (hd s)))/\~(P s) x) H0); Intros.
  Apply safePath_T with 1:=H1.
  Generalize H1; Clear H1; Generalize H2; Clear H2.
  Generalize x; Generalize Sini; Cofix u.
  Destruct x; Intros; Constructor.
  Elim (classic (P p^s)); [ Trivial | Intros ].
  Cut (ExistsS [s0:(Stream S*Instant)](bound (Snd (hd s0)))/\~(P s0) p^s).
  Intro ex; Elim (H2 ex).
  Apply Here with P:=[s:(Stream S*Instant)](bound (Snd (hd s)))/\~(P s).
  Split; Assumption.
  Apply u with Sini:=(hd s).
  Generalize H2; Clear H2; Case s; Unfold not; Intros.
  Apply (not_EX H2 H3).
  Unfold isTraceFrom_T; Elim H1; Intros ig trace; Inversion_clear trace.
  Split; Trivial.
  Split; Trivial.
Qed.


End TemporalOperators_TCTL.
```

Apéndice C

Especificación y Análisis del Sistema TCG

SystemTCG.v (disponible en **http://www.fing.edu.uy/~cluna**)

Versión de *Coq*: 6.2.2. Archivo generado con el utilitario *coq2html*.

C.1 Especificación

```
Implicit Arguments On.

Require time_clocks.  (* nociones temporales para tiempo discreto *)

Section Automatas_TCG.


 (* Definición del conjunto de etiquetas de los autómatas, en un único tipo *)

 Inductive Label : Set :=   Approach : Label | In : Label | Exit: Label | Lower : Label
                          | Raise : Label | Up : Label | Down : Label | Tick : Label.

 (* Tick es un label distinguido que representa a las transiciones temporales *)


 (* Locaciones del tren ST, del controlador SC y de la barrera SG *)

 Inductive ST : Set  := Far : ST | Near : ST | Inside : ST.
 Inductive SC : Set  := Sc1 : SC | Sc2 : SC | Sc3 : SC | Sc4 : SC.
 Inductive SG : Set  := Open : SG | Lowering : SG | Closed : SG | Raising: SG.


 (* Tipo de los estados con un reloj *)

 Definition S_Ck := [S:Set] S * Clock.


 (* Constantes del sistema *)

 Definition kt1  : Instant  := (8).
 Definition kt2  : Instant  := (20).
 Definition kc1  : Instant  := (4).
 Definition kc2  : Instant  := (4).
 Definition kg1  : Instant  := (4).
 Definition kg2  : Instant  := (4).
 Definition kg3  : Instant  := (8).
```

```
(* Invariantes de las locaciones del tren *)

Inductive InvT : (S_Ck ST) -> Prop :=
   Ifar    :  (x:Clock) (InvT (Far,x))
 | Inear   :  (x:Clock) (lt_Ck x kt2) -> (InvT (Near,x))
 | Iinside :  (x:Clock) (lt_Ck x kt2) -> (InvT (Inside,x)).


(* Invariantes de las locaciones del controlador *)

Inductive InvC : (S_Ck SC) -> Prop :=
   Isc1  :  (y:Clock) (InvC (Sc1,y))
 | Isc2  :  (y:Clock) (le_Ck y kc1) -> (InvC (Sc2,y))
 | Isc3  :  (y:Clock) (InvC (Sc3,y))
 | Isc4  :  (y:Clock) (lt_Ck y kc2) -> (InvC (Sc4,y)).


(* Invariantes de las locaciones de la barrera *)

Inductive InvG : (S_Ck SG) -> Prop :=
   Iopen     : (z:Clock) (InvG (Open,z))
 | Ilowering: (z:Clock) (lt_Ck z kg1) -> (InvG (Lowering,z))
 | Iclosed   : (z:Clock) (InvG (Closed,z))
 | Iraising  : (z:Clock) (lt_Ck z kg3) -> (InvG (Raising,z)).


(* Transiciones del tren *)

Inductive TrT : (S_Ck ST) -> Label -> (S_Ck ST) -> Prop :=
   ttApproach : (x:Clock) (TrT (Far,x) Approach (Near, Reset))
 | ttIn       : (x:Clock) (gt_Ck x kt1) -> (lt_Ck x kt2) -> (TrT (Near,x) In (Inside,x))
 | ttExit     : (x:Clock) (lt_Ck x kt2) -> (TrT (Inside,x) Exit (Far,x))
 | ttTick     : (x:Clock) (s:ST) (InvT (s, (Inc x))) -> (TrT (s,x) Tick (s, (Inc x))).


(* Transiciones del controlador *)

Inductive TrC : (S_Ck SC) -> Label -> (S_Ck SC) -> Prop :=
   tcApproach: (y:Clock) (TrC (Sc1,y) Approach (Sc2,Reset))
 | tcLower   : (y:Clock) (eq_Ck y kc1) -> (TrC (Sc2,y) Lower (Sc3,y))
 | tcExit    : (y:Clock) (TrC (Sc3,y) Exit (Sc4,Reset))
 | tcRaise   : (y:Clock) (lt_Ck y kc2) -> (TrC (Sc4,y) Raise (Sc1,y))
 | tcTick    : (y:Clock) (s: SC) (InvC (s, (Inc y))) -> (TrC (s,y) Tick (s, (Inc y))).


(* Transiciones de la barrera *)

Inductive TrG : (S_Ck SG) -> Label -> (S_Ck SG) -> Prop :=
   tgLower  : (z:Clock) (TrG (Open,z) Lower (Lowering,Reset))
 | tgDown   : (z:Clock) (lt_Ck z kg1) -> (TrG (Lowering,z) Down (Closed,z))
 | tgRaise  : (z:Clock) (TrG (Closed,z) Raise (Raising,Reset))
 | tgUp     : (z:Clock) (ge_Ck z kg2) -> (lt_Ck z kg3) -> (TrG (Raising,z) Up (Open,z))
 | tgTick   : (z:Clock) (s: SG) (InvG (s, (Inc z))) -> (TrG (s,z) Tick (s, (Inc z))).


(* Tipo de los estados del sistema TCG *)

Definition StGlobal := (S_Ck ST) * (S_Ck SC) * (S_Ck SG).
```

```
(* Transiciones del sistema compuesto train || controller || gate *)

Inductive TrGlobal : StGlobal -> Label -> StGlobal -> Prop :=

 | tGl_In       : (st1,st2:(S_Ck ST)) (TrT st1 In st2) -> (sc:(S_Ck SC)) (sg:(S_Ck SG))
                  (TrGlobal (st1,(sc,sg)) In (st2,(sc,sg)))
 | tGl_Down     : (sg1,sg2:(S_Ck SG)) (TrG sg1 Down sg2) -> (st:(S_Ck ST)) (sc:(S_Ck SC))
                  (TrGlobal (st,(sc,sg1)) Down (st,(sc,sg2)))
 | tGl_Up       : (sg1,sg2:(S_Ck SG)) (TrG sg1 Up sg2) -> (st:(S_Ck ST)) (sc:(S_Ck SC))
                  (TrGlobal (st,(sc,sg1)) Up (st,(sc,sg2)))
 | tGl_Approach: (st1,st2:(S_Ck ST)) (sc1,sc2:(S_Ck SC))
                  (TrT st1 Approach st2) -> (TrC sc1 Approach sc2) -> (sg:(S_Ck SG))
                  (TrGlobal (st1,(sc1,sg)) Approach (st2,(sc2,sg)))
 | tGl_Exit     : (st1,st2:(S_Ck ST)) (sc1,sc2:(S_Ck SC))
                  (TrT st1 Exit st2) -> (TrC sc1 Exit sc2) -> (sg:(S_Ck SG))
                  (TrGlobal (st1,(sc1,sg)) Exit (st2,(sc2,sg)))
 | tGl_Lower    : (sc1,sc2:(S_Ck SC)) (sg1,sg2:(S_Ck SG))
                  (TrC sc1 Lower sc2) -> (TrG sg1 Lower sg2) -> (st:(S_Ck ST))
                  (TrGlobal (st,(sc1,sg1)) Lower (st,(sc2,sg2)))
 | tGl_Raise    : (sc1,sc2:(S_Ck SC)) (sg1,sg2:(S_Ck SG))
                  (TrC sc1 Raise sc2) -> (TrG sg1 Raise sg2) -> (st:(S_Ck ST))
                  (TrGlobal (st,(sc1,sg1)) Raise (st,(sc2,sg2)))
 | tGl_Tick     : (st1,st2:(S_Ck ST)) (sc1,sc2:(S_Ck SC)) (sg1,sg2:(S_Ck SG))
                  (TrT st1 Tick st2) -> (TrC sc1 Tick sc2) -> (TrG sg1 Tick sg2)->
                  (TrGlobal (st1,(sc1,sg1)) Tick (st2,(sc2,sg2))).


(* Estado inicial para el sistema TCG y para sus componentes *)

Definition ini_CkT := O.
Definition ini_CkC := O.
Definition ini_CkG := O.

Definition SiniT  : (S_Ck ST)  := (Far, ini_CkT).
Definition SiniC  : (S_Ck SC)  := (Sc1, ini_CkC).
Definition SiniG  : (S_Ck SG)  := (Open, ini_CkG).

Definition SiniTCG : StGlobal := (SiniT, (SiniC, SiniG)).

End Automatas_TCG.


Hint Ifar Inear Iinside Isc1 Isc2 Isc3 Isc4 Iopen Ilowering Iclosed Iraising.
Hint ttApproach ttIn ttExit ttTick tcApproach tcLower tcExit tcRaise tcTick.
Hint tgLower tgDown tgRaise tgUp tgTick tGl_In tGl_Down tGl_Up.
Hint tGl_Approach tGl_Exit tGl_Lower tGl_Raise tGl_Tick.
```

C.2 Análisis

```
Load TemporalOperators.  (* biblioteca de operadores temporales *)
```

C.2.1 Propiedades de los los Relojes y los Parámetros del Sistema

```
Lemma Trivial1 : (eq_Ck ini_CkT ini_CkC).
Proof.
  Unfold eq_Ck ini_CkT ini_CkC; Trivial.
Qed.
```

```
Lemma Trivial2 : (x:Clock) (eq_Ck x x).
Proof.
  Unfold eq_Ck; Trivial.
Qed.

Lemma Trivial3 : (x,y:Clock) (eq_Ck x y) -> (eq_Ck (Inc x) (Inc y)).
Proof.
  Unfold eq_Ck Inc; Intros x y eq_x_y; Rewrite eq_x_y; Trivial.
Qed.

Lemma Trivial4 : (x,y,z:Clock) (eq_Ck x (plus_Ck y z)) -> (eq_Ck (Inc x) (plus_Ck (Inc y) z)).
Proof.
  Unfold eq_Ck Inc plus_Ck; Intros.
  Rewrite H; Rewrite (plus_assoc_r y z tick); Rewrite (plus_sym z tick); Trivial.
Qed.

Lemma Trivial5 : (x:Clock) (ge_Ck x Reset).
Proof.
  Unfold Clock ge_Ck Reset; Auto.
Qed.

Lemma Trivial6 : (x,y:Clock) (ge_Ck x y) -> (ge_Ck (Inc x) (Inc y)).
Proof.
  Unfold Clock ge_Ck Inc plus_Ck; Auto.
Qed.

Lemma Trivial7 : (x,y:Clock) (eq_Ck x y) -> (ge_Ck x y).
Proof.
  Unfold Clock ge_Ck eq_Ck; Intros.
  Rewrite H; Auto.
Qed.

Lemma Trivial8 : (x,y:Clock) (ge_Ck x y) -> (ge_Ck (Inc x) y).
Proof.
  Unfold Clock ge_Ck Inc plus_Ck tick plus; Auto.
Qed.

Lemma Trivial9 : (x,y:Clock) (gt_Ck x y) -> (gt_Ck (Inc x) y).
Proof.
  Unfold Clock gt_Ck Inc plus_Ck; Auto.
Qed.

Lemma Trivial10 : (x:Clock) (gt_Ck x kt1) -> (gt_Ck x kc1).
Proof.
  Unfold Clock gt_Ck kt1 kc1; Intros.
  Apply (gt_trans x (8) (4) H).
  Unfold gt; Repeat (Apply lt_n_S); Apply lt_O_Sn.
Qed.

Lemma Trivial11 : (x:Clock) ~(gt_Ck Reset x).
Proof.
  Unfold Clock gt_Ck Reset; Auto.
Qed.
```

```
Lemma Trivial12 : (x,y:Clock) (gt_Ck (Inc x) y) -> (eq_Ck x y) \/ (gt_Ck x y).
Proof.
  Unfold Clock Inc eq_Ck gt_Ck plus_Ck tick; Intros x y.
  Rewrite (plus_sym x (1)); Simpl; Intros.
  Elim (le_lt_or_eq y x (gt_S_le y x H)); Auto.
Qed.

Lemma Trivial13 : (x,y,z:Clock) (gt_Ck (Inc x) z) -> (eq_Ck x y) -> ~(le_Ck (Inc y) z).
Proof.
  Unfold Clock Inc plus_Ck tick eq_Ck le_Ck gt_Ck; Intros x y z.
  Rewrite (plus_sym x (1)); Rewrite (plus_sym y (1)); Simpl; Intros.
  Rewrite
Qed.

Lemma Trivial14 : (x:Clock) (gt_Ck x kt1) \/ (eq_Ck x kt1) -> (gt_Ck x kc1).
Proof.
  Unfold Clock kt1 kc1 tick eq_Ck gt_Ck; Intros.
  Elim H; Intro H1; [ Apply (gt_trans x (8) (4) H1) | Rewrite H1 ];
    Unfold gt; Repeat (Apply lt_n_S); Apply lt_O_Sn .
Qed.

Lemma Trivial15 : (x,y,z:Clock) (eq_Ck x kt1) -> (eq_Ck x y) -> (eq_Ck y (plus_Ck z kc1)) ->
            ~(lt_Ck (Inc z) kg1).
Proof.
  Unfold Clock kt1 kc1 kg1 Inc plus_Ck tick eq_Ck lt_Ck; Intros.
  Rewrite H in H0; Rewrite in H1.
  Apply lt_not_sym.
  Rewrite (plus_sym z (4)) in H1; Rewrite (plus_sym z (1)); Simpl.
  Rewrite (plus_minus (8) (4) z H1); Simpl.
  Repeat (Apply lt_n_S); Apply lt_O_Sn.
Qed.

Lemma Trivial16: (x,y:Clock) (lt_Ck x kc2) -> ~(eq_Ck x (plus_Ck y kc1)).
Proof.
  Unfold Clock kc1 kc2 eq_Ck plus_Ck tick lt_Ck not; Intros.
  Rewrite H0 in H; Elim (le_not_lt (4) (plus y (4)) (le_plus_r y (4)) H).
Qed.

Lemma Trivial17: (x,y,z:Clock) (eq_Ck x y) -> (eq_Ck x z) -> ~(le_Ck (Inc z) y).
Proof.
  Unfold Clock eq_Ck Inc plus_Ck tick le_Ck; Intros.
  Rewrite
  Rewrite plus_sym; Simpl; Unfold not; Intro.
  Elim (n_Sn x (le_antisym x (S x) (le_n_Sn x) H1)).
Qed.

Lemma Trivial18: (lt_Ck (Inc Reset) kc2).
Proof.
  Unfold lt_Ck Inc tick plus_Ck Reset kc2.
  Rewrite (plus_sym (0) (1)); Simpl.
  Apply lt_n_S; Apply lt_O_Sn.
Qed.
```

```
Lemma Trivial19: (x,y,z:Clock) (lt_Ck x kg1) -> (eq_Ck y (plus_Ck x kc1)) -> (eq_Ck z y) ->
        (lt_Ck (Inc z) kt2).
Proof.
  Unfold Clock lt_Ck eq_Ck Inc tick plus_Ck kg1 kc1 kt2; Intros.
  Rewrite H0 in H1; Rewrite H1.
  Rewrite (plus_sym x (4)); Rewrite (plus_sym (plus (4) x) (1)); Simpl.
  Repeat (Apply lt_n_S).
  Apply (lt_le_trans x (4) (15) H).
  Repeat (Apply le_n_S); Apply le_O_n.
Qed.

Lemma Trivial20: (x,y,z:Clock) (lt_Ck x kg1) -> (eq_Ck y (plus_Ck x kc1)) -> (eq_Ck z y) ->
        ~(gt_Ck z kt1).
Proof.
  Unfold Clock gt_Ck lt_Ck eq_Ck plus_Ck kg1 kc1 kt1 gt; Intros.
  Apply le_not_lt.
  Rewrite in H0; Rewrite H0.
  Generalize (lt_reg_r x (4) (4) H); Simpl; Intro.
  Apply (lt_le_weak (plus x (4)) (8) (lt_reg_r x (4) (4) H)).
Qed.

Lemma Trivial21: (lt_Ck (Inc Reset) kg3).
Proof.
  Unfold lt_Ck Inc tick plus_Ck Reset kg3.
  Rewrite (plus_sym (0) (1)); Simpl.
  Apply lt_n_S; Apply lt_O_Sn.
Qed.

Lemma Trivial22: (x,y:Clock) (le_Ck x y) -> (lt_Ck x y) \/ (eq_Ck x y).
Proof.
  Unfold Clock le_Ck lt_Ck eq_Ck; Intros.
  Apply (le_lt_or_eq x y H).
Qed.

Lemma Trivial23: (x,y:Clock) (lt_Ck x kc1) -> (eq_Ck y x) -> (lt (Inc y) kt2).
Proof.
  Unfold Clock lt_Ck eq_Ck Inc tick plus_Ck kc1 kt2; Intros.
  Rewrite in H; Rewrite (plus_sym y (1)); Simpl.
  Apply lt_n_S.
  Apply (lt_le_trans y (4) (19) H).
  Repeat (Apply le_n_S); Apply le_O_n.
Qed.

Lemma Trivial24: (x,y:Clock) (lt_Ck x y) -> (le_Ck (Inc x) y).
Proof.
  Unfold Clock lt_Ck le_Ck Inc tick plus_Ck; Intros.
  Rewrite (plus_sym x (1)); Simpl.
  Apply (lt_le_S x y H).
Qed.

Lemma Trivial25: (x,y:Clock) (eq_Ck y kc1) -> (eq_Ck x y) -> (lt_Ck (Inc x) kt2).
Proof.
  Unfold Clock eq_Ck lt_Ck Inc tick plus_Ck kc1 kt2; Intros.
  Rewrite (plus_sym x (1)); Simpl.
  Apply lt_n_S.
  Rewrite H in H0; Rewrite H0.
  Repeat (Apply lt_n_S); Apply lt_O_Sn.
Qed.
```

```
Lemma Trivial26: (lt_Ck (Inc Reset) kg1).
Proof.
  Unfold lt_Ck Inc plus_Ck tick kg1 Reset.
  Rewrite (plus_sym (0) (1)); Simpl.
  Apply lt_n_S; Apply lt_O_Sn.
Qed.

Lemma Trivial27: (lt_Ck (Inc Reset) kt2).
Proof.
  Unfold lt_Ck Inc plus_Ck tick kt2 Reset.
  Rewrite (plus_sym (0) (1)); Simpl.
  Apply lt_n_S; Apply lt_O_Sn.
Qed.

Lemma Trivial28: (le_Ck (Inc Reset) kc1).
Proof.
  Unfold le_Ck Inc plus_Ck tick kc1 Reset.
  Rewrite (plus_sym (0) (1)); Simpl.
  Apply le_n_S; Apply le_O_n.
Qed.

Lemma Trivial29: (x:Clock) (lt_Ck x kg3) -> (lt_Ck x kg2) \/ ((ge_Ck x kg2) /\ (lt_Ck x kg3)).
Proof.
  Unfold Clock lt_Ck ge_Ck kg2 kg3 ge; Intros.
  Elim (le_or_lt (4) x); Auto.
Qed.

Lemma Trivial30: (x:Clock) (lt_Ck x kg2) -> (lt_Ck (Inc x) kg3).
Proof.
  Unfold Clock lt_Ck Inc plus_Ck tick kg2 kg3; Intros.
  Rewrite (plus_sym x (1)); Simpl.
  Apply lt_n_S.
  Apply (lt_le_trans x (4) (7) H).
  Repeat (Apply le_n_S); Apply le_O_n.
Qed.

Lemma Trivial31: (x,y:Clock) (eq_Ck x kc1) -> (ge_Ck y x) -> (ge_Ck y kg2).
Proof.
  Unfold Clock ge_Ck eq_Ck kc1 kg2 ge; Intros.
  Rewrite H in H0; Assumption.
Qed.

Lemma not_lt_le: (x,y:nat) ~(lt x y) -> (le y x).
Proof.
  Unfold not; Intros x y no_lt_x_y.
  Elim (le_or_lt y x);
    [ Trivial | (Intro lt_x_y; Elim (no_lt_x_y lt_x_y)) ].
Qed.

Lemma Trivial32: (x:Clock) (lt_Ck x kt2) -> ~(lt_Ck (Inc x) kt2) -> (gt_Ck x kt1).
Proof.
  Unfold Clock lt_Ck gt_Ck Inc plus_Ck tick kt1 kt2 gt; Intros.
  Generalize (not_lt_le H0); Rewrite (plus_sym x (1)); Simpl; Intro.
  Generalize (le_antisym (19) x (le_S_n (19) x H1) (lt_n_Sm_le x (19) H)); Intro H4; Rewrite
  Repeat (Apply lt_n_S); Apply lt_O_Sn.
Qed.

Lemma not_le_lt: (x,y:nat) ~(le x y) -> (lt y x).
```

```
Lemma Trivial33: (x:Clock) (le_Ck x kc1) -> ~(le_Ck (Inc x) kc1) -> (eq_Ck x kc1).
Proof.
  Unfold Clock le_Ck eq_Ck Inc plus_Ck tick kc1; Intros.
  Generalize (not_le_lt H0); Rewrite (plus_sym x (1)); Simpl; Intro.
  Apply (le_antisym x (4) H (lt_n_Sm_le (4) x H1)).
Qed.

Lemma Trivial34: (x:Clock) (lt_Ck x kg3) -> ~(lt_Ck (Inc x) kg3) -> (ge_Ck x kg2).
Proof.
  Unfold Clock lt_Ck ge_Ck Inc plus_Ck tick kg3 kg2; Intros.
  Generalize (not_lt_le H0); Rewrite (plus_sym x (1)); Simpl; Intro.
  Generalize (le_antisym (7) x (le_S_n (7) x H1) (lt_n_Sm_le x (7) H)); Intro H4; Rewrite
  Unfold ge; Repeat (Apply le_n_S); Apply le_O_n.
Qed.

Lemma trivial_inv_1: (lt_Ck Reset kt2).
Proof.
  Unfold lt_Ck Reset kt2; Auto.
Qed.

Lemma trivial_inv_2: (x:Clock) (le_Ck Reset x).
Proof.
  Unfold Clock le_Ck Reset; Auto.
Qed.

Lemma trivial_inv_3: (lt_Ck Reset kc2).
  Unfold lt_Ck Reset kc2; Auto.
Qed.

Lemma trivial_inv_4: (lt_Ck Reset kg1).
Proof.
  Unfold lt_Ck Reset kg1; Auto.
Qed.

Lemma trivial_inv_5: (lt_Ck Reset kg3).
Proof.
  Unfold lt_Ck Reset kg3; Auto.
Qed.

(* NOTA: Los lemas previos pueden ser demostrados también con la táctica automática Omega, de la
biblioteca Omega. Deben hacerse todos los Unfold de las constantes, luego Intros y entonces Omega *)
```

C.2.2 Lemas de Inversión

```
(* Lemas de inversión para las transiciones y los invariantes de las locaciones *)

Derive Inversion_clear cl_TRT_APPROACH with (st1,st2:(S_Ck ST)) (TrT st1 Approach st2) Sort Prop.
Derive Inversion_clear cl_TRT_IN with (st1,st2:(S_Ck ST)) (TrT st1 In st2) Sort Prop.
Derive Inversion_clear cl_TRT_EXIT with (st1,st2:(S_Ck ST)) (TrT st1 Exit st2) Sort Prop.
Derive Inversion_clear cl_TRT_INC_TIME with (st1,st2:(S_Ck ST)) (TrT st1 Tick st2) Sort Prop.

Derive Inversion_clear cl_TRC_APPROACH with (sc1,sc2:(S_Ck SC)) (TrC sc1 Approach sc2) Sort Prop.
Derive Inversion_clear cl_TRC_LOWER with (sc1,sc2:(S_Ck SC)) (TrC sc1 Lower sc2) Sort Prop.
Derive Inversion_clear cl_TRC_EXIT with (sc1,sc2:(S_Ck SC)) (TrC sc1 Exit sc2) Sort Prop.
Derive Inversion_clear cl_TRC_RAISE with (sc1,sc2:(S_Ck SC)) (TrC sc1 Raise sc2) Sort Prop.
Derive Inversion_clear cl_TRC_INC_TIME with (sc1,sc2:(S_Ck SC)) (TrC sc1 Tick sc2) Sort Prop.

Derive Inversion_clear cl_TRG_LOWER with (sg1,sg2:(S_Ck SG)) (TrG sg1 Lower sg2) Sort Prop.
```

```
Derive Inversion_clear cl_TRG_DOWN with (sg1,sg2:(S_Ck SG)) (TrG sg1 Down sg2) Sort Prop.
Derive Inversion_clear cl_TRG_RAISE with (sg1,sg2:(S_Ck SG)) (TrG sg1 Raise sg2) Sort Prop.
Derive Inversion_clear cl_TRG_UP with (sg1,sg2:(S_Ck SG)) (TrG sg1 Up sg2) Sort Prop.
Derive Inversion_clear cl_TRG_INC_TIME with (sg1,sg2:(S_Ck SG)) (TrG sg1 Tick sg2) Sort Prop.

Derive Inversion_clear cl_Ifar with (x:Clock) (InvT (Far,x)) Sort Prop.
Derive Inversion_clear cl_Inear with (x:Clock) (InvT (Near,x)) Sort Prop.
Derive Inversion_clear cl_Iinside with (x:Clock) (InvT (Inside,x)) Sort Prop.

Derive Inversion_clear cl_Isc1 with (y:Clock) (InvC (Sc1,y)) Sort Prop.
Derive Inversion_clear cl_Isc2 with (y:Clock) (InvC (Sc2,y)) Sort Prop.
Derive Inversion_clear cl_Isc3 with (y:Clock) (InvC (Sc3,y)) Sort Prop.
Derive Inversion_clear cl_Isc4 with (y:Clock) (InvC (Sc4,y)) Sort Prop.

Derive Inversion_clear cl_Iopen with (z:Clock) (InvG (Open,z)) Sort Prop.
Derive Inversion_clear cl_Ilowering with (z:Clock) (InvG (Lowering,z)) Sort Prop.
Derive Inversion_clear cl_Iclosed with (z:Clock) (InvG (Closed,z)) Sort Prop.
Derive Inversion_clear cl_Iraising with (z:Clock) (InvG (Raising,z)) Sort Prop.
```

C.2.3 Algunas Tácticas Útiles

```
Tactic Definition Easy :=
[<:tactic:<
  Try Discriminate; Auto
>>].

Tactic Definition Easy2 :=
[<:tactic:<
  Intros; Try Discriminate; Auto
>>].

Tactic Definition Simpl_or [$eq] :=
[<:tactic:<
  Elim $eq; Intro; Easy
>>].

Tactic Definition Simpl_and [$eq] :=
[<:tactic:<
  Elim $eq; Intros; Easy
>>].

Tactic Definition Generalize_Easy [$eq] :=
[<:tactic:<
  Generalize $eq; Intro; Easy
>>].

Tactic Definition Split3_Trivial :=
[<:tactic:<
  Split; [ Auto | Split; Auto ]
>>].

Tactic Definition BeginForAll :=
[<:tactic:<
  Unfold ForAll; Induction 1; [ Simpl; Intros; Easy | Idtac ]
>>].
```

```
Tactic Definition SplitTrans :=
[<:tactic:<
  Intros s1 s2 l rs_s1 p_s1 tr_gl;
  Generalize rs_s1; Clear rs_s1; Generalize p_s1; Clear p_s1;
  Elim tr_gl
>>].

Tactic Definition SplitTrans_Simpl :=
[<:tactic:<
  SplitTrans; [ Simpl_In | Simpl_Down | Simpl_Up | Simpl_Approach | Simpl_Exit
              | Simpl_Lower | Simpl_Raise | Simpl_Tick]
>>].

Tactic Definition Simpl_In :=
[<:tactic:<
  Intros st1 st2 trt_In;
  Inversion trt_In using cl_TRT_IN;
  Intros x gt_x_kt1 lt_x_kt2 sc sg;
  Elim sc; Elim sg;
  Simpl; Intros; Easy
>>].

Tactic Definition Simpl_Down :=
[<:tactic:<
  Intros sg1 sg2 trg_Down;
  Inversion trg_Down using cl_TRG_DOWN;
  Intros z lt_z_kg1 st sc;
  Elim st; Elim sc;
  Simpl; Intros; Easy
>>].

Tactic Definition Simpl_Up :=
[<:tactic:<
  Intros sg1 sg2 trg_Up;
  Inversion trg_Up using cl_TRG_UP;
  Intros z ge_z_kg2 lt_z_kg3 st sc;
  Elim st; Elim sc;
  Simpl; Intros; Easy
>>].

Tactic Definition Simpl_Approach :=
[<:tactic:<
  Intros st1 st2 sc1 sc2 trt_Approach trc_Approach;
  Inversion trt_Approach using cl_TRT_APPROACH;
  Inversion trc_Approach using cl_TRC_APPROACH;
  Intros y x sg;
  Elim sg;
  Simpl; Intros; Easy
>>].

Tactic Definition Simpl_Exit :=
[<:tactic:<
  Intros st1 st2 sc1 sc2 trt_Exit trc_Exit;
  Inversion trt_Exit using cl_TRT_EXIT;
  Inversion trc_Exit using cl_TRC_EXIT;
  Intros y x lt_x_kt2 sg;
  Elim sg;
  Simpl; Intros; Easy
>>].
```

```
Tactic Definition Simpl_Lower :=
[<:tactic:<
  Intros sc1 sc2 sg1 sg2 trc_Lower trg_Lower;
  Inversion trc_Lower using cl_TRC_LOWER;
  Inversion trg_Lower using cl_TRG_LOWER;
  Intros z y eq_y_kc1 st;
  Elim st;
  Simpl; Intros; Easy
>>].

Tactic Definition Simpl_Raise :=
[<:tactic:<
  Intros sc1 sc2 sg1 sg2 trc_Raise trg_Raise;
  Inversion trc_Raise using cl_TRC_RAISE;
  Inversion trg_Raise using cl_TRG_RAISE;
  Intros z y lt_y_kc2 st;
  Elim st;
  Simpl; Intros; Easy
>>].

Tactic Definition Simpl_Tick :=
[<:tactic:<
  Intros st1 st2 sc1 sc2 sg1 sg2 trt_Tick trc_Tick trg_Tick;
  Inversion trt_Tick using cl_TRT_INC_TIME;
  Inversion trc_Tick using cl_TRC_INC_TIME;
  Inversion trg_Tick using cl_TRG_INC_TIME;
  Simpl; Intros; Easy
>>].
```

C.2.4 Invariantes y Propiedad de *Safety*: Demostraciones

```
Syntactic Definition InvTCG := [s:StGlobal] Cases s of (st_x,(sc_y,sg_z)) =
  (InvT st_x) /\ (InvC sc_y) /\ (InvG sg_z) end.

Syntactic Definition ForAll_TCG := (ForAll TrGlobal).

Syntactic Definition Exists_TCG := (Exists TrGlobal).


(* El estado inicial del sistema TCG satisface el invariante de las locaciones *)

Lemma Inv_SiniTCG : (InvTCG SiniTCG).
Proof.
  Simpl; Unfold SiniT SiniC SiniG; Split3_Trivial.
Qed.


Definition Inv1 := [s:StGlobal] Cases s of ((st,_),((sc,_),(sg,_))) =
  st=Near\/st=Inside -> sc=Sc3 -> sg=Closed \/ sg=Lowering end.

Lemma lema_Inv1 : (ForAll_TCG SiniTCG Inv1).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Simpl_or (p_s1 H0 H1).
Qed.
Hint lema_Inv1.
```

```
Definition Inv2 := [s:StGlobal] Cases s of ((st,_),((sc,_),(sg,_))) =
  st=Far -> sg=Open\/sg=Raising -> sc=Sc1 end.

Lemma lema_Inv2 : (ForAll_TCG SiniTCG Inv2).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Simpl_or H1.
  Elim (lema_Inv1 rs_s1 (or_intror Inside=Near Inside=Inside (refl_equal ST Inside)) (refl_equal SC Sc3));
  Intro H2; Rewrite H2 in H1; Simpl_or H1.
  Simpl_or H1.
Qed.
Hint lema_Inv2.


Definition Inv3 := [s:StGlobal] Cases s of ((st,x),((_,y),_)) =
  st=Near\/st=Inside -> (eq_Ck x y) end.

Lemma lema_Inv3 : (ForAll_TCG SiniTCG Inv3).
Proof.
  BeginForAll.
  Apply Trivial1.
  SplitTrans_Simpl.
  Apply (Trivial2 Reset).
  Simpl_or H0.
  Apply (Trivial3 (p_s1 H3)).
Qed.
Hint lema_Inv3.


Definition Inv4 := [s:StGlobal] Cases s of ((st,_),((sc,_),_)) =
  st=Near -> sc=Sc2\/sc=Sc3 end.

Lemma lema_Inv4 : (ForAll_TCG SiniTCG Inv4).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Simpl_or (p_s1 H0).
Qed.
Hint lema_Inv4.


Definition Inv5 := [s:StGlobal] Cases s of ((st,_),((sc,y),(_,z))) =
  st=Near -> sc=Sc3 -> (eq_Ck y (plus_Ck z kc1)) end.

Lemma lema_Inv5 : (ForAll_TCG SiniTCG Inv5).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Apply (Trivial4 (p_s1 H3 H4)).
Qed.
Hint lema_Inv5.


Definition Inv6 := [s:StGlobal] Cases s of ((st,x),((sc,_),_)) =
  st=Near/\(gt_Ck x kc1) \/ st=Inside -> sc=Sc3 end.
```

```
Lemma lema_Inv6 : (ForAll_TCG SiniTCG Inv6).
Proof.
  BeginForAll.
  Decompose [and or] H0; Discriminate.
  SplitTrans_Simpl.
  Apply p_s1; Left; Split; [ Auto | Apply (Trivial10 gt_x_kt1) ].
  Decompose [and or] H0; [ Elim (Trivial11 H3) | Discriminate ].
  Decompose [and or] H0; Discriminate.
  Generalize_Easy (p_s1 H0).
  Decompose [and or] H3; [ Elim (Trivial12 H6) | Auto ].
  Elim (lema_Inv4 rs_s1 H4); [Generalize (lema_Inv3 rs_s1 (or_introl s4=Near s4=Inside H4)); Intros | Auto].
  Rewrite H7 in H1; Inversion H1 using cl_Isc2; Intro H9; Elim (Trivial13 H6 H5 H9).
  Intro H7; Apply p_s1; Left; Split; [ Assumption | Generalize_Easy (Trivial9 H7) ].
Qed.
Hint lema_Inv6.


Definition Inv7 := [s:StGlobal] Cases s of ((st,_),((sc,_),(sg,_))) =
  st=Near\/st=Inside -> sc=Sc3 -> sg=Lowering\/sg=Closed end.

Lemma lema_Inv7 : (ForAll_TCG SiniTCG Inv7).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Simpl_or (p_s1 H0 H1).
Qed.
Hint lema_Inv7.


Definition Inv8 := [s:StGlobal] Cases s of ((st,x),((sc,_),(sg,_))) =
  st=Near/\(gt_Ck x kt1) \/ st=Inside -> sg=Closed end.

Lemma lema_Inv8 : (ForAll_TCG SiniTCG Inv8).
Proof.
  BeginForAll.
  Decompose [and or] H0; Discriminate.
  SplitTrans_Simpl.
  Generalize_Easy (p_s1 H0).
  Decompose [and or] H0; [ Elim (Trivial11 H3) | Discriminate ].
  Generalize_Easy (p_s1 H0).
  Generalize (lema_Inv6 rs_s1); Simpl; Intro.
  Decompose [and or] H0.
  Generalize_Easy (H1 (or_introl y0=Near/\(gt_Ck y1 kc1) y0=Inside <(y0=Near),(gt_Ck y1 kc1)>
    {H2, (Trivial14 (or_introl (gt_Ck y1 kt1) (eq_Ck y1 kt1) H4))})).
  Generalize_Easy (H1 (or_intror y0=Near/\(gt_Ck y1 kc1) y0=Inside H3)).
  Decompose [and or] H3; [ Simpl_or (Trivial12 H6) | Auto ].
  Generalize (lema_Inv6 rs_s1 (or_introl s4=Near/\(gt_Ck x kc1) s4=Inside <(s4=Near),(gt_Ck x kc1)>
    {H4, (Trivial14 (or_intror (gt_Ck x kt1) (eq_Ck x kt1) H5))})); Intro.
  Simpl_or (lema_Inv1 rs_s1 (or_introl s4=Near s4=Inside H4) H7).
  Rewrite H8 in H0; Inversion H0 using cl_Ilowering; Intro.
  Elim (Trivial15 H5 (lema_Inv3 rs_s1 (or_introl s4=Near s4=Inside H4)) (lema_Inv5 rs_s1 H4 H7) H9).
Qed.
Hint lema_Inv8.


Definition safeTCG := [s:StGlobal] Cases s of ((st,_),(_,(sg,_))) =
  st=Inside -> sg=Closed end.
```

```
Lemma lema_safeTCG : (ForAll_TCG SiniTCG safeTCG).
  Proof.
  Apply Mon_I with Pg:=Inv8 Pp:=safeTCG; [ Auto | (Unfold Inv8 safeTCG; Simpl) ].
  Unfold Inv8 safeTCG; Simpl; Intro s; Elim s.
  Intros y y0; Elim y; Elim y0;
  Intros y1 y2 y3 y4; Elim y1; Elim y2; Auto.
Qed.
Hint lema_safeTCG.


Definition Inv9 := [s:StGlobal] Cases s of ((st,x),((_,y),(_,z))) =
  st=Near/\(gt_Ck x kc1) \/ st=Inside -> (eq_Ck y (plus_Ck z kc1)) end.

Lemma lema_Inv9 : (ForAll_TCG SiniTCG Inv9).
Proof.
  BeginForAll.
  Decompose [and or] H0; Discriminate.
  SplitTrans_Simpl.
  Apply p_s1; Left; Split; [ Auto | Apply (Trivial10 gt_x_kt1) ].
  Decompose [and or] H0; [ Elim (Trivial11 H3) | Discriminate ].
  Decompose [and or] H0; Discriminate.
  Elim (Trivial16 lt_y_kc2 (p_s1 H0)).
  Decompose [and or] H3.
  Elim (Trivial12 H6); Intro.
  Elim (lema_Inv4 rs_s1 H4); Intro.
  Rewrite H7 in H1; Inversion H1 using cl_Isc2; Intro.
  Generalize (lema_Inv3 rs_s1 (or_introl s4=Near s4=Inside H4)); Intro.
  Elim (Trivial17 H5 H9 H8).
  Apply (Trivial4 (lema_Inv5 rs_s1 H4 H7)).
  Apply Trivial4; Apply p_s1; Left; Split; Assumption.
  Apply (Trivial4 (p_s1 (or_intror s4=Near/\(gt_Ck x kc1) s4=Inside H5))).
Qed.
Hint lema_Inv9.


Definition Inv10 := [s:StGlobal] Cases s of (_,((sc,_),(sg,_))) =
  sc=Sc1\/sc=Sc2 -> sg=Open\/sg=Raising end.

Lemma lema_Inv10 : (ForAll_TCG SiniTCG Inv10).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Simpl_or (p_s1 H0).
  Simpl_or H0.
  Simpl_or H0.
Qed.
Hint lema_Inv10.


Definition Inv11 := [s:StGlobal] Cases s of (_,((sc,_),(sg,_))) =
  sg=Lowering\/sg=Closed -> sc=Sc3\/sc=Sc4 end.
```

```
Lemma lema_Inv11 : (ForAll_TCG SiniTCG Inv11).
Proof.
  Generalize (Mon_I 2!TrGlobal 3!SiniTCG 4!Inv10 5!Inv11); Unfold ForAll; Intros.
  Apply H; [ Auto | Idtac | Assumption ].
  Unfold Inv10 Inv11; Simpl; Intro.
  Elim s0; Do 2 Intro; Elim y; Elim y0; Do 4 Intro; Elim y1; Elim y2.
  Intros sg ck sc; Elim sg.
  Intros ck1 H1 H2; Simpl_or H2.
  Elim sc; Intros ck1 H1; [ (Elim H1; Easy2 ) | (Elim H1; Easy2 ) | (Left; Auto) | (Right; Auto) ].
  Elim sc; Intros ck1 H1; [ (Elim H1; Easy2 ) | (Elim H1; Easy2 ) | (Left; Auto) | (Right; Auto) ].
  Intros ck1 H1 H2; Simpl_or H2.
Qed.
Hint lema_Inv11.


Definition Inv12 := [s:StGlobal] Cases s of (_,((sc,y),(_,z))) =
  sc=Sc2 -> (ge_Ck z y) end.

Lemma lema_Inv12:(ForAll_TCG SiniTCG Inv12).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Apply Trivial5.
  Apply (Trivial6 (p_s1 H3)).
Qed.
Hint lema_Inv12.


Definition Inv13 := [s:StGlobal] Cases s of ((st,_),((sc,_),_)) =
  sc=Sc2 -> st=Near end.

Lemma lema_Inv13 : (ForAll_TCG SiniTCG Inv13).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Generalize (lema_Inv6 rs_s1); Simpl.
  Rewrite H0.
  Intro H1; Generalize_Easy (H1 (or_introl Near=Near/\(gt_Ck x kc1) Near=Inside
    <(Near=Near),(gt_Ck x kc1)> {(refl_equal ST Near),(Trivial10 gt_x_kt1)})).
Qed.
Hint lema_Inv13.


Definition Inv14 := [s:StGlobal] Cases s of ((st,_),((sc,_),(sg,_))) =
  sc=Sc4 -> st=Far/\sg=Closed end.

Lemma lema_Inv14 : (ForAll_TCG SiniTCG Inv14).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Simpl_and (p_s1 H0).
  Simpl_and (p_s1 H0).
  Simpl_and (p_s1 H0).
  Split; [ Auto | Apply (lema_safeTCG rs_s1 (refl_equal ST Inside)) ].
Qed.
Hint lema_Inv14.
```

```
Definition InvSc3 := [s:StGlobal] Cases s of (_,((sc,y),_)) =
  sc=Sc3 -> (ge_Ck y kc1) end.

Lemma lema_InvSc3 : (ForAll_TCG SiniTCG InvSc3).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Apply (Trivial7 eq_y_kc1).
  Apply (Trivial8 (p_s1 H3)).
Qed.
Hint lema_InvSc3.


Definition InvInside := [s:StGlobal] Cases s of ((st,x),_) =
  st=Inside -> (gt_Ck x kt1) end.

Lemma lema_InvInside : (ForAll_TCG SiniTCG InvInside).
Proof.
  BeginForAll.
  SplitTrans_Simpl.
  Apply (Trivial9 (p_s1 H3)).
Qed.
Hint lema_InvInside.


Lemma INV_TCG_general : (s:StGlobal) (InvTCG s) -> (ForAll_TCG s InvTCG).
Proof.
  Intros.
  BeginForAll.
  SplitTrans_Simpl ; Elim p_s1; Intros H1 H2; Elim H2; Intros; Auto; (Split; [ Auto | Split; Auto ]).
  Apply Inear; Apply trivial_inv_1.
  Apply Isc2; Apply trivial_inv_2.
  Apply Isc4; Apply trivial_inv_3.
  Apply Ilowering; Apply trivial_inv_4.
  Apply Iraising; Apply trivial_inv_5.
Qed.
Hint INV_TCG_general.


Lemma INV_T_general : (s1,s2:StGlobal) (InvTCG s1) -> (RState TrGlobal s1 s2) -> (InvT (Fst s2)).
Proof.
  Intros.
  Generalize (INV_TCG_general H H0).
  Elim s2; Simpl.
  Intros st sc_sg.
  Elim sc_sg; Intros.
  Elim H1; Auto.
Qed.


Lemma INV_C_general : (s1,s2:StGlobal) (InvTCG s1) -> (RState TrGlobal s1 s2) -> (InvC (Fst (Snd s2))).
Proof.
  Intros.
  Generalize (INV_TCG_general H H0).
  Elim s2; Simpl.
  Intros st sc_sg.
  Elim sc_sg; Simpl; Intros.
  Elim H1; Intros inv_T inv_C_G; Elim inv_C_G; Auto.
Qed.
```

```
Lemma INV_G_general : (s1,s2:StGlobal) (InvTCG s1) -> (RState TrGlobal s1 s2) -> (InvG (Snd (Snd s2))).
Proof.
  Intros.
  Generalize (INV_TCG_general H H0).
  Elim s2; Simpl.
  Intros st sc_sg.
  Elim sc_sg; Simpl; Intros.
  Elim H1; Intros inv_T inv_C_G; Elim inv_C_G; Auto.
Qed.
```

C.2.5 Propiedad de *Liveness Non-Zeno*

Lemas, Definiciones y Tácticas Auxiliares

```
Lemma InvT' : (s:(S_Ck ST)) Cases s of (st,x) =
  st=Far \/ st=Near/\(lt_Ck x kt2) \/ st=Inside/\(lt_Ck x kt2) -> (InvT s) end.
Proof.
  Intro s; Elim s.
  Intros.
  Elim H; [ Intro st; Rewrite st; Auto | Intro n_i; Elim n_i; Intro H1; Elim H1; Intros st cond; Rewrite st;
    [ Apply Inear | Apply Iinside ]; Auto ].
Qed.
```

```
Lemma InvC' : (s:(S_Ck SC)) Cases s of (sc,y) =
  sc=Sc1 \/ sc=Sc2/\(le_Ck y kc1) \/ sc=Sc3 \/ sc=Sc4/\(lt_Ck y kc2) -> (InvC s) end.
Proof.
  Intro s; Elim s.
  Intros.
  Elim H; [ Intro sc; Rewrite sc; Auto | Intro sc2_4; Elim sc2_4;
    [ Intro H1; Elim H1; Intros sc cond; Rewrite sc; Apply Isc2; Auto | Intro sc3_4; Elim sc3_4;
      [ Intro sc; Rewrite sc; Auto | Intro H1; Elim H1; Intros sc cond; Rewrite sc; Apply Isc4; Auto ] ] ].
Qed.
```

```
Lemma InvG' : (s:(S_Ck SG)) Cases s of (sg,z) =
  sg=Open \/ sg=Lowering/\(lt_Ck z kg1) \/ sg=Closed \/ sg=Raising/\(lt_Ck z kg3) -> (InvG s) end.
Proof.
  Intro s; Elim s.
  Intros.
  Elim H; [ Intro sg; Rewrite sg; Auto | Intro sg2_4; Elim sg2_4;
    [ Intro H1; Elim H1; Intros sg cond; Rewrite sg; Apply Ilowering; Auto | Intro sg3_4; Elim sg3_4;
      [ Intro sg; Rewrite sg; Auto | Intro H1; Elim H1; Intros sg cond; Rewrite sg; Apply Iraising; Auto ] ] ].
Qed.
```

```
Lemma NoImpl: (A,B:Prop) (A->B) -> ~B -> ~A.
Proof.
  Unfold not; Auto.
Qed.
```

```
Lemma not_3_and: (A,B,C:Prop) ~(A/\B/\C) -> ~A \/ A/\~B \/ A/\B/\~C.
Proof.
  Intros.
  Elim (not_and_or ? ? H); Intro H2; [ Idtac | Elim (not_and_or ? ? H2); Intro ].
  Auto.
  Elim (classic A); Auto.
  Elim (classic A); Elim (classic B); Auto.
Qed.


Syntactic Definition no_invT := [s:(S_Ck ST)] Cases s of (st,x) =
  st=Near/\~(lt_Ck x kt2) \/ st=Inside/\~(lt_Ck x kt2) end.

Lemma No_invT : (s:(S_Ck ST)) ~(InvT s) -> (no_invT s).
Proof.
  Intro s; Elim s.
  Intros st x inv.
  Generalize (not_or_and ? ? (NoImpl (InvT' (st,x)) inv)); Elim st; Intro H0;
    Elim H0; Intros H1 H2; Elim (not_or_and ? ? H2); Intros.
  Absurd Far=Far; Auto.
  Elim (not_and_or ? ? H); Intros; [ Absurd Near=Near; Auto | Auto ].
  Elim (not_and_or ? ? H3); Intros; [ Absurd Inside=Inside; Auto | Auto ].
Qed.


Syntactic Definition no_invC := [s:(S_Ck SC)] Cases s of (sc,y) =
  sc=Sc2/\~(le_Ck y kc1) \/ sc=Sc4/\~(lt_Ck y kc2) end.

Lemma No_invC : (s:(S_Ck SC)) ~(InvC s) -> (no_invC s).
Proof.
  Intro s; Elim s.
  Intros sc y inv.
  Generalize (not_or_and ? ? (NoImpl (InvC' (sc,y)) inv)); Elim sc; Intro H0;
    Elim H0; Intros H1 H2; Elim (not_or_and ? ? H2); Intros.
  Absurd Sc1=Sc1; Auto.
  Elim (not_and_or ? ? H); Intros; [ Absurd Sc2=Sc2; Auto | Auto ].
  Elim (not_or_and ? ? H3); Intros; Absurd Sc3=Sc3; Auto.
  Elim (not_or_and ? ? H3); Intros H4 H5; Elim (not_and_or ? ? H5); Intros;
    [ Absurd Sc4=Sc4; Auto | Auto ].
Qed.


Syntactic Definition no_invG := [s:(S_Ck SG)] Cases s of (sg,z) =
  sg=Lowering/\~(lt_Ck z kg1) \/ sg=Raising/\~(lt_Ck z kg3) end.

Lemma No_invG : (s:(S_Ck SG)) ~(InvG s) -> (no_invG s).
Proof.
  Intro s; Elim s.
  Intros sc y inv.
  Generalize (not_or_and ? ? (NoImpl (InvG' (sc,y)) inv)); Elim sc; Intro H0;
    Elim H0; Intros H1 H2; Elim (not_or_and ? ? H2); Intros.
  Absurd Open=Open; Auto.
  Elim (not_and_or ? ? H); Intros; [ Absurd Lowering=Lowering; Auto | Auto ].
  Elim (not_or_and ? ? H3); Intros; Absurd Closed=Closed; Auto.
  Elim (not_or_and ? ? H3); Intros H4 H5; Elim (not_and_or ? ? H5); Intros;
    [ Absurd Raising=Raising; Auto | Auto ].
Qed.
```

```
Definition InvTick := [s:StGlobal] Cases s of ((st,x),((sc,y),(sg,z))) =
  (InvTCG ((st,(Inc x)),((sc,(Inc y)),(sg,(Inc z))))) end.

Definition noInvTick := [s:StGlobal] Cases s of ((st,x),((sc,y),(sg,z))) =
  ~(InvTick ((st,x),((sc,y),(sg,z)))) -> (RState TrGlobal SiniTCG ((st,x),((sc,y),(sg,z)))) ->
  ((st=Near/\(gt_Ck x kt1) \/ st=Inside)
  \/ ((InvT (st,(Inc x))) /\ (sc=Sc2/\(eq_Ck y kc1) \/ sc=Sc4))
  \/ ((InvT (st,(Inc x))) /\ (InvC (sc,(Inc y))) /\ (sg=Lowering \/ sg=Raising/\(ge_Ck z kg2)))) end.


Lemma NoInvTick : (s:StGlobal) (noInvTick s).
Proof.
  Intro s; Elim s.
  Intros st sc_sg; Elim sc_sg; Intros sc sg; Elim st; Elim sc; Elim sg; Intros.
  Simpl; Intros.
  Elim (not_3_and H); [ Intro no_invt | Intro H1; Decompose [and or] H1 ].
  Elim (No_invT no_invt); Intro H1; Elim H1; Intros st' x; Rewrite st'; Rewrite st' in H0.
  Generalize (INV_T_general 1!SiniTCG Inv_SiniTCG H0); Simpl; Intro inv_near;
    Inversion inv_near using cl_Inear; Intro lt_x_kt2.
  Left; Left; Split; [ Auto | Apply (Trivial32 lt_x_kt2 x) ].
  Auto.
  Elim (No_invC H4); Intro H5; Elim H5; Intros sc' y; Rewrite sc'; Rewrite sc' in H0.
  Generalize (INV_C_general 1!SiniTCG Inv_SiniTCG H0); Simpl; Intro inv_sc2;
    Inversion inv_sc2 using cl_Isc2; Intro le_y_kc1.
  Generalize (Trivial33 le_y_kc1 y5); Intro.
  Right; Left; Auto.
  Auto.
  Elim (No_invG H5); Intro H6; Elim H6; Intros sg' z; Rewrite sg'; Rewrite sg' in H0.
  Right; Right; Auto.
  Generalize (INV_G_general 1!SiniTCG Inv_SiniTCG H0); Simpl; Intro inv_raising;
    Inversion inv_raising using cl_Iraising; Intro lt_z_kg3.
  Generalize (Trivial34 lt_z_kg3 z); Intro.
  Right; Right; Auto.
Qed.


Syntactic Definition rsNext_In :=
  [x:Clock] [gt_x_kt1:(gt_Ck x kt1)] [lt_x_kt2:(lt_Ck x kt2)] [sc_y:(S_Ck SC)] [sg_z:(S_Ck SG)]
  (rsNext (rsIni TrGlobal ((Near,x),(sc_y,sg_z)))
    (tGl_In (ttIn 1!x gt_x_kt1 lt_x_kt2) sc_y sg_z)).

Syntactic Definition rsNext_Down :=
  [z:Clock] [lt_z_kg1:(lt_Ck z kg1)] [st_x:(S_Ck ST)] [sc_y:(S_Ck SC)]
  (rsNext (rsIni TrGlobal (st_x,(sc_y,(Lowering,z))))
    (tGl_Down (tgDown lt_z_kg1) st_x sc_y)).

Syntactic Definition rsNext_Up :=
  [z:Clock] [ge_z_kg2:(ge_Ck z kg2)] [lt_z_kg3:(lt_Ck z kg3)] [st_x:(S_Ck ST)] [sc_y:(S_Ck SC)]
  (rsNext (rsIni TrGlobal (st_x,(sc_y,(Raising,z))))
    (tGl_Up (tgUp ge_z_kg2 lt_z_kg3 ) st_x sc_y)).

Syntactic Definition rs_Next_Approach :=
  [x,y:Clock] [sg_z:(S_Ck SG)]
  (rsNext StGlobal (rsIni StGlobal TrGlobal ((Far,x),((Sc1,y),sg_z)))
    (tGl_Approach (ttApproach x) (tcApproach y) sg_z)).
```

```
Syntactic Definition rsNext_Exit :=
  [x,y:Clock] [lt_x_kt2:(lt_Ck x kt2)] [sg_z:(S_Ck SG)]
  (rsNext (rsIni TrGlobal ((Inside,x),((Sc3,y),sg_z)))
    (tGl_Exit (ttExit lt_x_kt2) (tcExit y) sg_z)).

Syntactic Definition rsNext_Lower :=
  [y,z:Clock] [eq_y_kc1:(eq_Ck y kc1)] [st_x:(S_Ck ST)]
  (rsNext (rsIni TrGlobal (st_x,((Sc2,y),(Open,z))))
    (tGl_Lower (tcLower eq_y_kc1) (tgLower z) st_x)).

Syntactic Definition rsNext_Raise :=
  [y,z:Clock] [lt_y_kc2:(lt_Ck y kc2)] [st_x:(S_Ck ST)]
  (rsNext (rsIni TrGlobal (st_x,((Sc4,y),(Closed,z))))
    (tGl_Raise (tcRaise lt_y_kc2) (tgRaise z) st_x)).

Tactic Definition SplitTrans' :=
[<:tactic:<
  Intros s1 s2 l rs_s1 p_s1 tr_gl;
  Generalize (rsNext rs_s1 tr_gl);
  Generalize rs_s1; Clear rs_s1;
  Generalize p_s1; Clear p_s1;
  Elim tr_gl
>>].

Tactic Definition SplitTrans'_Simpl :=
[<:tactic:<
  SplitTrans'; [Simpl_In | Simpl_Down | Simpl_Up | Simpl_Approach | Simpl_Exit
    | Simpl_Lower | Simpl_Raise | Simpl_Tick]
>>].

Tactic Definition ExistsHere_ITick [$s] :=
[<:tactic:<
  Apply exists with tr:=TrGlobal P:=InvTick 1:=(rsIni TrGlobal $s)
>>].
```

Demostración de *Non-Zeno*

```
Lemma NonZeno : (ForAll_TCG SiniTCG ([s:StGlobal] (Exists_TCG s InvTick))).
Proof.
  BeginForAll.
  ExistsHere_ITick SiniTCG.
  Split3_Trivial.
  SplitTrans'_Simpl ; Generalize H0.
  Rewrite (lema_Inv6 H0 (or_intror Inside=Near/\(gt_Ck x kc1) Inside=Inside (refl_equal ST Inside))).
  Rewrite (lema_safeTCG H0 (refl_equal ST Inside)); Intro rs_s2.
  Apply StepsEX with s2:=((Far,x),((Sc4,Reset),(Closed,y0))).
  Apply (rsNext_Exit x y2 lt_x_kt2 (Closed,y0)).
  ExistsHere_ITick ((Far,x),((Sc4,Reset),(Closed,y0))).
  Split3_Trivial ; (Apply Isc4; Apply Trivial18).
  Elim (lema_Inv11 H0 (or_intror Closed=Lowering Closed=Closed (refl_equal SG Closed)));
    Intro sc_closed; Rewrite sc_closed.
  Elim y1; Intro rs_s2.
  ExistsHere_ITick ((Far,y2),((Sc3,y0),(Closed,z))).
  Split3_Trivial.
  ExistsHere_ITick ((Near,y2),((Sc3,y0),(Closed,z))).
  Split3_Trivial.
  Apply Inear.
```

```
Apply (Trivial19 lt_z_kg1
  (lema_Inv5 rs_s2 (refl_equal ST Near) (refl_equal SC Sc3))
  (lema_Inv3 rs_s2 (or_introl Near=Near Near=Inside (refl_equal ST Near)))).
Elim (Trivial20 lt_z_kg1
  (lema_Inv9 rs_s2 (or_intror Inside=Near/\(gt_Ck y2 kc1) Inside=Inside (refl_equal ST Inside)))
  (lema_Inv3 rs_s2 (or_intror Inside=Near Inside=Inside (refl_equal ST Inside)))
  (lema_InvInside rs_s2 (refl_equal ST Inside))).
Elim y1; Intro rs_s2.
Generalize (INV_C_general 1!SiniTCG Inv_SiniTCG rs_s2); Simpl;
  Intro inv_sc4; Inversion inv_sc4 using cl_Isc4; Intro le_y0_kc2.
Apply StepsEX with s2:=((Far,y2),((Sc1,y0),(Raising,Reset))).
Apply (rsNext_Raise y0 z le_y0_kc2 (Far,y2)).
ExistsHere_ITick ((Far,y2),((Sc1,y0),(Raising,Reset))).
Split3_Trivial ; (Apply Iraising; Apply Trivial21).
Elim (lema_Inv4 rs_s2 (refl_equal ST Near));
  Intro sc_near; Rewrite sc_near in sc_closed; Discriminate.
Elim (Trivial20 lt_z_kg1
  (lema_Inv9 rs_s2 (or_intror Inside=Near/\(gt_Ck y2 kc1) Inside=Inside (refl_equal ST Inside)))
  (lema_Inv3 rs_s2 (or_intror Inside=Near Inside=Inside (refl_equal ST Inside)))
  (lema_InvInside rs_s2 (refl_equal ST Inside))).
Elim y1.
Intro rs_s2; Generalize rs_s2; Rewrite (lema_Inv2 rs_s2 (refl_equal ST Far)
  (or_introl Open=Open Open=Raising (refl_equal SG Open))); Intro rs_s2fin.
ExistsHere_ITick ((Far,y2),((Sc1,y0),(Open,z))).
Split3_Trivial.
Intro rs_s2; Generalize rs_s2; Elim (lema_Inv4 rs_s2 (refl_equal ST Near)); Intro sc_near; Rewrite sc_near.
Intro rs_s2'; Generalize (INV_C_general 1!SiniTCG Inv_SiniTCG rs_s2'); Simpl;
  Intro inv_sc2; Inversion inv_sc2 using cl_Isc2; Intro le_y0_kc1.
Elim (Trivial22 le_y0_kc1); Intro.
ExistsHere_ITick ((Near,y2),((Sc2,y0),(Open,z))).
Split; [ Apply Inear | Split; [ Apply Isc2 | Auto ] ].
Apply (Trivial23 H1 (lema_Inv3 rs_s2' (or_introl Near=Near Near=Inside (refl_equal ST Near)))).
Apply (Trivial24 H1).
Apply StepsEX with s2:=((Near,y2),((Sc3,y0),(Lowering,Reset))).
Apply (rsNext_Lower y0 z H1 (Near,y2)).
ExistsHere_ITick ((Near,y2),((Sc3,y0),(Lowering,Reset))).
Split; [ Apply Inear | Split; [ Auto | Apply Ilowering ] ].
Apply (Trivial25 H1 (lema_Inv3 rs_s2' (or_introl Near=Near Near=Inside (refl_equal ST Near)))).
Apply Trivial26.
Intro rs_s2fin; Elim (lema_Inv1 rs_s2fin (or_introl Near=Near Near=Inside (refl_equal ST Near))
  (refl_equal SC Sc3)); Intro; Discriminate.
Intro rs_s2; Generalize_Easy (lema_safeTCG rs_s2 (refl_equal ST Inside)).
Elim (lema_Inv10 H0 (or_intror Sc2=Sc1 Sc2=Sc2 (refl_equal SC Sc2)));
  Intros H1; Rewrite H1; Intro rs_s2.
ExistsHere_ITick ((Near,Reset),((Sc2,Reset),(Open,y1))).
Split; [ Apply Inear; Apply Trivial27 | Split; [ Apply Isc2; Apply Trivial28 | Auto ] ].
Generalize (INV_G_general 1!SiniTCG Inv_SiniTCG rs_s2); Simpl;
  Intro inv_raising; Inversion inv_raising using cl_Iraising; Intro lt_y1_kg3.
Elim (Trivial29 lt_y1_kg3); Intro.
ExistsHere_ITick ((Near,Reset),((Sc2,Reset),(Raising,y1))).
Split; [ Apply Inear; Apply Trivial27 |
  Split; [ Apply Isc2; Apply Trivial28 | Apply Iraising; Apply (Trivial30 H2) ] ].
Apply StepsEX with s2:=((Near,Reset),((Sc2,Reset),(Open,y1))).
Elim H2; Intros H3 H4.
Apply (rsNext_Up y1 H3 H4 (Near,Reset) (Sc2,Reset)).
ExistsHere_ITick ((Near,Reset),((Sc2,Reset),(Open,y1))).
Split; [ Apply Inear; Apply Trivial27 | Split; [ Apply Isc2; Apply Trivial28 | Auto ] ].
Elim y0; Intro rs_s2.
ExistsHere_ITick ((Far,x),((Sc4,Reset),(Open,y1))).
```

```
Split3_Trivial .
Apply Isc4; Apply Trivial18.
Generalize (INV_G_general 1!SiniTCG Inv_SiniTCG rs_s2); Simpl;
  Intro inv_lowering; Inversion inv_lowering using cl_Ilowering; Intro lt_y1_kg1.
Apply StepsEX with s2:=((Far,x),((Sc4,Reset),(Closed,y1))).
Apply (rsNext_Down y1 lt_y1_kg1 (Far,x) (Sc4,Reset)).
ExistsHere_ITick ((Far,x),((Sc4,Reset),(Closed,y1))).
Split3_Trivial.
Apply Isc4; Apply Trivial18.
ExistsHere_ITick ((Far,x),((Sc4,Reset),(Closed,y1))).
Split3_Trivial.
Apply Isc4; Apply Trivial18.
Generalize_Easy (lema_Inv2 rs_s2 (refl_equal ST Far)
  (or_intror Raising=Open Raising=Raising (refl_equal SG Raising))).
Intro rs_s2; ExistsHere_ITick ((y0,y1),((Sc3,y),(Lowering,Reset))).
Split; [ Generalize rs_s2; Elim y0; Intro rs_s2' | Split; [ Auto | Apply Ilowering; Apply Trivial26 ] ].
Auto.
Apply Inear; Apply (Trivial25 eq_y_kc1 (lema_Inv3 rs_s2'
  (or_introl Near=Near Near=Inside (refl_equal ST Near)))).
Generalize_Easy (lema_safeTCG rs_s2' (refl_equal ST Inside)).
Intro rs_s2; ExistsHere_ITick ((y0,y1),((Sc1,y),(Raising,Reset))).
Split; [ Generalize rs_s2; Elim y0; Intro rs_s2' | Split; [ Auto | Apply Iraising; Apply Trivial21 ] ].
Auto.
Simpl_or (lema_Inv4 rs_s2' (refl_equal ST Near)).
Generalize_Easy (lema_safeTCG rs_s2' (refl_equal ST Inside)).
Intro; Elim (classic (InvTick ((s4,(Inc x)),((s3,(Inc y)),(s0,(Inc z)))))); Intro inv_tick.
ExistsHere_ITick ((s4,(Inc x)),((s3,(Inc y)),(s0,(Inc z)))); Auto.
Generalize H3; Elim (NoInvTick ((s4,(Inc x)),((s3,(Inc y)),(s0,(Inc z)))) inv_tick H3).
Intro no_st; Decompose [and or] no_st; [ Rewrite H5 | Rewrite H6 ].
Intro rs_near; Generalize (INV_T_general 1!SiniTCG Inv_SiniTCG rs_near); Simpl;
  Intro inv_near; Inversion inv_near using cl_Inear; Intro lt_incx_kt2.
Generalize (rsNext_In (Inc x) H7 lt_incx_kt2 (s3,(Inc y)) (s0,(Inc z))); Intro rs_inside.
Generalize rs_near; Generalize rs_inside; Rewrite (lema_Inv6 (RState_Trans rs_near rs_inside)
  (or_intror Inside=Near/\(gt_Ck (Inc x) kc1) Inside=Inside (refl_equal ST Inside))).
Rewrite (lema_safeTCG (RState_Trans rs_near rs_inside) (refl_equal ST Inside)).
Intros; Apply StepsEX with s2:=((Far,(Inc x)),((Sc4,Reset),(Closed,(Inc z)))).
Apply (RState_Trans rs_inside0 (rsNext_Exit (Inc x) (Inc y) lt_incx_kt2 (Closed,(Inc z)))).
ExistsHere_ITick ((Far,(Inc x)),((Sc4,Reset),(Closed,(Inc z)))).
Split3_Trivial ; Apply Isc4; Apply Trivial18.
Intro rs_inside; Generalize rs_inside;
  Rewrite (lema_safeTCG rs_inside (refl_equal ST Inside)); Intro rs_inside0.
Generalize rs_inside0; Rewrite (lema_Inv6 rs_inside0 (or_intror Inside=Near/\(gt_Ck (Inc x) kc1)
  Inside=Inside (refl_equal ST Inside))); Intro rs_inside1.
Apply StepsEX with s2:=((Far,(Inc x)),((Sc4,Reset),(Closed,(Inc z)))).
Generalize (INV_T_general 1!SiniTCG Inv_SiniTCG rs_inside0); Simpl;
  Intro inv_inside; Inversion inv_inside using cl_Iinside; Intro lt_incx_kt2.
Apply (rsNext_Exit (Inc x) (Inc y) lt_incx_kt2 (Closed,(Inc z))).
ExistsHere_ITick ((Far,(Inc x)),((Sc4,Reset),(Closed,(Inc z)))).
Split3_Trivial ; Apply Isc4; Apply Trivial18.
Intro no_sc_sg; Elim no_sc_sg; [ Intro st_no_sc; Decompose [and or] st_no_sc | Intro no_sg ].
Rewrite H7; Rewrite H7 in rs_s1.
Elim (lema_Inv10 rs_s1 (or_intror Sc2=Sc1 Sc2=Sc2 (refl_equal SC Sc2))); Intro sg'; Rewrite sg'.
Intro rs_s2; Apply StepsEX with s2:=((s4,(Inc x)),((Sc3,(Inc y)),(Lowering,Reset))).
Apply (rsNext_Lower (Inc y) (Inc z) H8 (s4,(Inc x))).
ExistsHere_ITick ((s4,(Inc x)),((Sc3,(Inc y)),(Lowering,Reset))).
Split3_Trivial; Apply Ilowering; Apply Trivial26.
Intro rs_s2; Apply StepsEX with s2:=((s4,(Inc x)),((Sc2,(Inc y)),(Open,(Inc z)))).
Generalize (lema_Inv12 rs_s2 (refl_equal SC Sc2)); Intro ge_incz_incy.
Generalize (INV_G_general 1!SiniTCG Inv_SiniTCG rs_s2); Simpl; Intro inv_raising;
```

```
  Inversion inv_raising using cl_Iraising; Intro lt_incz_kg3.
 Apply (rsNext_Up (Inc z) (Trivial31 H8 ge_incz_incy) lt_incz_kg3 (s4,(Inc x)) (Sc2,(Inc y))).
 Apply StepsEX with s2:=((s4,(Inc x)),((Sc3,(Inc y)),(Lowering,Reset))).
 Apply (rsNext_Lower (Inc y) (Inc z) H8 (s4,(Inc x))).
 ExistsHere_ITick ((s4,(Inc x)),((Sc3,(Inc y)),(Lowering,Reset))).
 Split3_Trivial; Apply llowering; Apply Trivial26.
 Rewrite H5; Intro rs_s2; Generalize rs_s2; Generalize (lema_Inv14 rs_s2 (refl_equal SC Sc4));
  Intro far_closed; Elim far_closed; Intros far closed; Rewrite closed.
 Intro rs_s2'; Apply StepsEX with s2:=((s4,(Inc x)),((Sc1,(Inc y)),(Raising,Reset))).
 Generalize (INV_C_general 1!SiniTCG Inv_SiniTCG rs_s2'); Simpl; Intro inv_sc4;
  Inversion inv_sc4 using cl_Isc4; Intro le_incy_kc2.
 Apply (rsNext_Raise (Inc y) (Inc z) le_incy_kc2 (s4,(Inc x))).
 ExistsHere_ITick ((s4,(Inc x)),((Sc1,(Inc y)),(Raising,Reset))).
 Split3_Trivial; Apply Iraising; Apply Trivial21.
 Decompose [and or] no_sg.
 Rewrite H7; Intro rs_s2; Generalize (INV_G_general 1!SiniTCG Inv_SiniTCG rs_s2); Simpl;
  Intro inv_lowering; Inversion inv_lowering using cl_Ilowering; Intro lt_incz_kg1.
 Apply StepsEX with s2:=((s4,(Inc x)),((s3,(Inc y)),(Closed,(Inc z)))).
 Apply (rsNext_Down (Inc z) lt_incz_kg1 (s4,(Inc x)) (s3,(Inc y))).
 ExistsHere_ITick ((s4,(Inc x)),((s3,(Inc y)),(Closed,(Inc z)))).
 Split3_Trivial.
 Rewrite H8; Intro rs_s2; Apply StepsEX with s2:=((s4,(Inc x)),((s3,(Inc y)),(Open,(Inc z)))).
 Generalize (INV_G_general 1!SiniTCG Inv_SiniTCG rs_s2); Simpl; Intro inv_raising;
  Inversion inv_raising using cl_Iraising; Intro lt_incz_kg3.
 Apply (rsNext_Up (Inc z) H9 lt_incz_kg3 (s4,(Inc x)) (s3,(Inc y))).
 ExistsHere_ITick ((s4,(Inc x)),((s3,(Inc y)),(Open,(Inc z)))).
 Split3_Trivial.
Qed.


(*
 Especificación alternativa de NonZeno, usando operadores temporales acotados:

Definition S_true := [_:StGlobal] True.
Lemma NonZeno : (ForAll_TCG SiniTCG ([s:StGlobal] (Exists_TCG s S_true (eq_Ck tick)))).
*)
```