

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Licenciatura en Ciencias de la Computación

Traducción de UML State Machines en Autómatas Temporizados

Pablo Demidoff – 04/91

Daniela Galigniana – 352/91

Director: Fernando Schapachnik

Índice

Índice.....	2
Índice de Figuras y Tablas	3
1. Introducción.....	5
2. Problema.....	6
3. UML - State Machines.....	8
Estados y Estados Compuestos	10
Estado final.....	13
Transiciones.....	13
Eventos	13
Eventos de tiempo – after	13
Semántica de la state machine.....	14
Semántica de los eventos.....	14
Semántica de los estados	15
Procesamiento de eventos: run-to-completion.....	15
Transiciones en Conflicto	18
Prioridad de Disparo.....	18
Algoritmo de selección de Transición	18
4. Autómata Temporizado	20
Relojes	20
Autómata Temporizado.....	21
Semántica	22
Ejemplo de autómata temporizado.....	23
5. Reglas de traducción.....	25
6. Primera versión: De state machines a autómatas finitos	29
Algoritmo principal.....	29
Cuerpo principal del algoritmo.....	30
Función: ObtenerTransiciones	33
Función: ObtenerTransicionesEstadoSimple.....	34

Función: ObtenerTransicionesEstadoCompuestoNoConcurrente	34
Función: ObtenerTransicionesEstadoCompuestoConcurrente.....	34
Función: crearEstadoConfiguracionDestino.....	35
7. Demostración: de state machines a autómatas finitos	41
8. Segunda versión: De state machines a autómatas temporizados	46
9. Justificación de la transformación de state machines a autómatas temporizados.....	49
10. Ejemplo	55
Otras pruebas realizadas.....	61
11. Conclusiones	65
12. Trabajos Futuros	66
13. Referencias.....	67

Índice de Figuras y Tablas

Figura 1: State Machine - Principal.....	9
Figura 3: Estado Simple	11
Figura 4: Estado compuesto no concurrente	11
Figura 5: Estado Compuesto Concurrente con 2 regiones	12
Figura 6: <i>State Machine</i> bien formada	12
Figura 7: Ejemplo de una <i>state machine</i> con estados concurrentes.....	17
Figura 8: Estado de configuración de la state machine de la Figura 7	18
Figura 9: <i>Autómata temporizado</i> representando una Expendedora de bebidas.....	24
Figura 10: Estado de configuración inicial de la state machine de la Figura 7.....	26
Figura 11: Estado de Configuración cuando se proceso el evento E1 desde el estado de configuración inicial	26
Figura 12: Estado de Configuración cuando se proceso el evento E4 desde el estado de configuración inicial	27
Figura 13: <i>Autómata</i> tras el procesamiento de los eventos E1 y E4.....	27
Figura 14: <i>Autómata</i> resultante de la traducción de la <i>state machine</i> de la Figura 7	28
Figura 15: <i>State machine</i> para ejemplificar la creación del <i>estado de configuración</i> destino.....	36
Figura 16: Un <i>estado de configuración</i> válido de la <i>state machine</i> de la Figura 15.....	37

Figura 17: Dos relojes activos con diferencias en las inicializaciones y mediciones	46
Figura 18: <i>State machine</i> para ejemplificar el uso de relojes	50
Figura 19: <i>State machine</i> modelando un sistema de monitoreo de pacientes.....	57
Figura 20: <i>Autómata temporizado</i> generado por el algoritmo	60
Figura 21: Ejemplo de <i>state machine</i> , caso 3 de la tabla de más abajo.....	62
Figura 22: Ejemplo de <i>state machine</i> , caso 4 de la tabla de más abajo.....	63
Tabla con los resultados de las corridas realizadas.....	63
Figura 23: Comportamiento del algoritmo.....	64

1. Introducción

Los *autómatas temporizados*^{1,9} han sido utilizados extensivamente para analizar y modelar características de los sistemas de tiempo real, esto es sistemas cuyo correcto funcionamiento debe asegurar el cumplimiento de estrictas restricciones de tiempo tales como tiempo de respuesta, período de tareas, demoras de transmisión, etc. La representación de los sistemas a través de los *autómatas temporizados* permite realizar chequeos formales sobre los mismos utilizando *model checkers* para asegurar y/o validar las restricciones de tiempo que deben cumplirse en el sistema.

La semántica de los *autómatas temporizados* está definida formalmente, lo que hace que se requiera un conocimiento detallado de la misma para poder utilizarlos. Esta es una limitante a la hora de usarlos masivamente.

Por otro lado, el lenguaje *UML*² (*Unified Modeling Language*) se ha convertido en el estándar de documentación de cualquier sistema, incluidos los de tiempo real. Permite modelar un sistema completo desde su etapa de análisis hasta los detalles de implementación y distribución de los componentes en el hardware. El lenguaje provee una serie de artefactos que permiten modelar tanto la visión estática del sistema como la dinámica, con una semántica semi-formal.

Dado que *UML* se ha convertido en un estándar, de amplia repercusión y uso en la industria del desarrollo de sistemas, con una amplia gama de herramientas que permiten modelar sistemas utilizándolo, surge la motivación de este trabajo para utilizarlo como lenguaje de modelado de los sistemas de tiempo real, sin perder la posibilidad de realizar los chequeos requeridos para garantizar el correcto funcionamiento de los mismos.

Para ello, a lo largo de este trabajo nos focalizaremos en los artefactos *UML* que permiten describir las partes dinámicas de los sistemas, y más precisamente en un artefacto: la *state machine*. Dentro de la especificación de *UML*, la definición de la *state machine* provee un lenguaje jerárquico altamente expresivo con una sintaxis bien definida, pero desafortunadamente, una semántica que no está bien formalizada. El foco de este trabajo está basado en la utilización de este artefacto para modelar el comportamiento de sistemas de tiempo real y poder utilizarlos como entrada a un *model checker*.

Para llevar a cabo esta tarea se presentará un algoritmo que permite traducir una *state machine* en un *autómata temporizado*, que a su vez pueda usarse como entrada para un *model checker*. La traducción está basada en el comportamiento de la *state machine*, y como ésta reacciona a los distintos eventos posibles.

El informe contiene en la sección 2 una descripción detallada del problema que queremos resolver. En la sección 3 se presentan las *state machines* y en la sección 4 los *autómatas temporizados* que se utilizan en este trabajo. En la sección 5 se detallan las reglas de traducción y los puntos básicos de la traducción propuesta. A lo largo de este trabajo se mostrarán dos algoritmos; el primero de ellos se describe en la sección 6 y traduce las *state machine* a *autómatas finitos*. Este algoritmo posee una demostración de su funcionamiento en la sección 7. El segundo algoritmo, detallado en la sección 8 y justificado en la sección 9, agrega el manejo de los eventos de tiempo e incorpora relojes al *autómata*, llevando la traducción de *state machines* a *autómatas temporizados*. Para finalizar, en la sección 10 se presenta un ejemplo y en las secciones 11 y 12 las conclusiones y futuros trabajos.

2. Problema

Los métodos formales han demostrado ser aplicables con eficiencia en el desarrollo industrial de sistemas críticos de tiempo real^{14, 15}. Los *model checkers* son una de las herramientas que permiten realizar verificaciones formales sobre los sistemas de tiempo real, ofreciendo simulaciones y validaciones formales de propiedades sobre los mismos. Estos sistemas son generalmente modelados a través de *autómatas temporizados*, que son la entrada para las herramientas de validación. Zeus¹³, UPPAL³, Kronos⁴ son ejemplos de herramientas que permiten realizar estas validaciones.

Sin embargo, los métodos formales no son muy usados en la industria. El problema es que mientras la demanda por software de tiempo real se incrementa muy rápido, la disponibilidad de desarrolladores que puedan dominarlos se mantiene pequeña. La consecuencia es que los métodos formales son generalmente considerados muy difíciles o muy caros para ser usados en desarrollos de software de tiempo real ordinarios.

Por el contrario, *UML* ha logrado una gran popularidad, esencialmente porque es una notación semi-formal relativamente fácil de usar y bien soportada por herramientas de desarrollo. En el último tiempo, *UML* también está ganando popularidad en los desarrollos de tiempo real. Sin ir más lejos, la OMG (Object Management Group) definió una especificación para ser usada en el modelado de sistemas de tiempo real, conocida como *UML for Real Time*¹² (*UML RT*). *UML RT* fue definida partiendo de la base de ROOM⁵ y ha sido adoptada rápidamente por varios desarrolladores. Sin embargo, la aplicación de *UML RT* para el campo de tiempo real todavía sufre del mismo problema que *UML*, no está formalmente bien definida. Esta es una limitación relevante, dado que generalmente las aplicaciones de tiempo real también son críticas en cuanto a la seguridad, y por lo tanto requiere de actividades como la verificación formal de cumplimiento de varias propiedades como safety, utility, liveness, etc.; la simulación del sistema que se está modelando, la generación de casos de prueba, entre muchas otras. Es muy difícil, llevar a cabo estas actividades cuando las especificaciones están escritas en una notación semi-formal como *UML* o *UML RT*.

Ante esta falta de formalismo sobre *UML RT*, se han presentado diferentes trabajos y análisis en la búsqueda de poder utilizarlo como lenguaje de modelado de sistemas de tiempo real, dando formalidad a los artefactos de *UML*.

Vieru Del Bianco, Luigi Lavazza, Marco Mauri, Giuseppe Occorso en su trabajo "*Towards UML-based formal specifications of component-based real-time software*"⁶ proponen un formalismo llamado *UML+* que es una extensión de *UML* y que no posee la restricción de *UML RT* de no tener una definición formal. Igualmente, este nuevo formalismo no soporta el diseño y desarrollo directamente sobre *UML+*, de la manera que lo permiten las herramientas que implementan *UML*. Este formalismo intenta unir las ventajas de cada uno para paliar sus diferencias. La idea general es desarrollar una especificación en *UML+*, luego traducirlo automáticamente a *TRIO temporal logic formulas* o *autómatas temporizados* para poder hacer la verificación formal de propiedades. Para el caso de *autómatas temporizados* proponen usar *Kronos model checker*. Una vez validado el modelo, se lo traduce de *UML+* a *UML RT* para empezar la etapa de diseño y desarrollo. Recién en este punto, y con la traducción realizada, se pueden utilizar herramientas, tales como el Rational Rose RealTime.

En forma más completa Alexander Knapp, Stephan Merz y Christopher Rauh en su trabajo "*Model Checking Timed UML State Machines and Collaborations*"⁷ presentan un prototipo de una

herramienta a la que llamaron HUGO/RT que automáticamente verifica si una *state machine* con anotaciones de tiempo interactúa acorde a escenarios especificados por diagramas de colaboración con anotaciones de tiempo. Los diagramas de *UML* son traducidos automáticamente a *autómatas temporizados*, y luego se usa el *model checker UPPAAL* para la verificación de las propiedades. En forma general, se cuenta con un conjunto de *state machines* que representan al sistema. Por otro lado, se cuenta con un conjunto de diagramas de colaboración que representan las propiedades que queremos verificar en nuestro sistema. Las *state machines* serán traducidas en un conjunto de *autómatas temporizados*. Cada diagrama de colaboración sería traducido a un "observer *UPPAAL timed autómatas*". Luego, se utiliza el *model checker UPPAAL* para verificar la propiedad deseada.

En estos trabajos mencionados, así como en algunos otros, si bien se presentan traducciones, no se presenta un algoritmo formal que realice las traducciones mencionadas. Por lo tanto, lo que se busca con este trabajo es presentar un algoritmo que traduzca las *state machine* de *UML* en *autómatas temporizados* que puedan ser utilizados por las herramientas de *model checking*. De esta forma se podrá utilizar las herramientas brindadas por *UML* pero sin perder la rigurosidad presentada por los métodos formales.

Nuestro objetivo es que el algoritmo aquí presentado facilite las tareas de análisis y diseño de sistemas de tiempo real, formando parte de un proceso de desarrollo que contemple:

1. Definir la/s *state machine/s* que modelan el sistema que se quiere desarrollar
2. Ejecutar nuestro algoritmo para transformar la/s *state machine/s* en un/os *autómata temporizado/s*
3. Escribir el observador con la propiedad que se quiera verificar
4. Ejecutar el *model checker*
5. En caso de que las propiedades no se verifiquen, modificar la/s *state machine/s* y volver a realizar el paso 2.

3. UML - State Machines

El paquete de *state machines* forma parte de los *Elementos de Comportamiento* (Behavioral Elements) de la especificación de UML. Este paquete especifica una serie de conceptos que se pueden usar para modelar comportamiento discreto a través de sistemas finitos de transición de estados.

La especificación de las *state machines* realizada por UML es una variante orientada a objetos de los *statecharts* de Harel⁸. Los *statecharts* de Harel son a su vez una extensión de los diagramas convencionales de estados y transiciones con tres elementos que permiten manejar jerarquías, concurrencia y comunicaciones. El corazón de este enfoque se basa en agregar a los diagramas de estados y transiciones convencionales descomposiciones AND/OR de estados en conjunto con transiciones entre los niveles, y un mecanismo de broadcast para la comunicación entre componentes concurrentes. Las ideas esenciales detrás de esta extensión son la provisión de descripciones en profundidad y la noción de ortogonalidad.

Las principales diferencias que presentan las *state machines* sobre los *statecharts* de Harel son las siguientes:

- Los eventos pueden llevar parámetros, no sólo ser señales primitivas
- La *state machine* soporta *call events* para modelar comportamientos de tipos
- No existe la noción de procesos. Más aún, todas las acciones y eventos predefinidos que se relacionan con actividades no están definidos, así como la relación entre actividad y estado

La Figura 1 expresa gráficamente la sintaxis abstracta de las *state machines* y la Figura 2 expresa la sintaxis de los eventos que pueden disparar el comportamiento de la *state machine*.

El modelado usando una *state machine* parte de un estado top, que es el que contiene los distintos estados y transiciones que definen el comportamiento del objeto que se está modelando, conformando una estructura jerárquica entre los distintos estados. En las próximas secciones detallaremos estos conceptos.

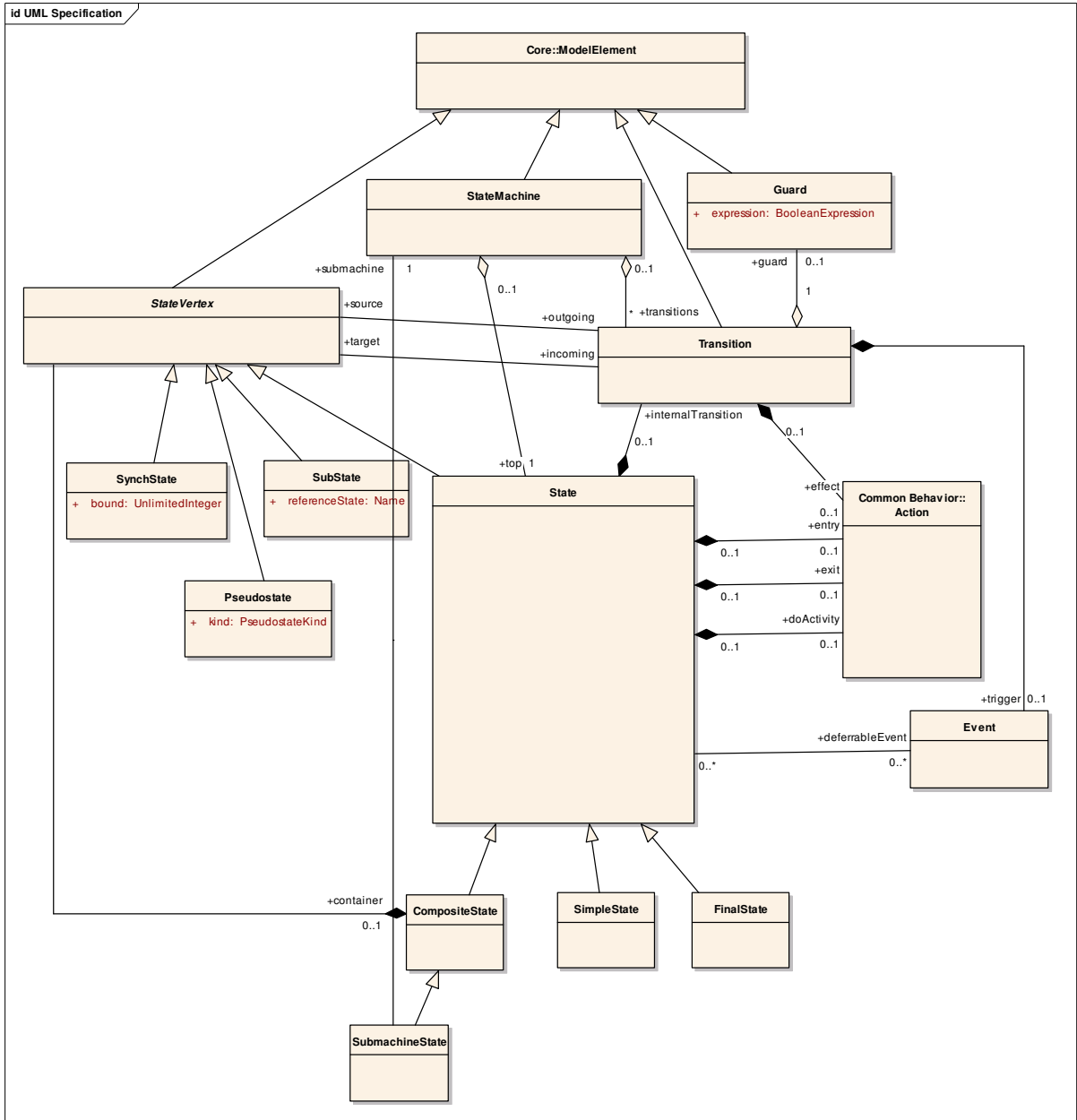


Figura 1: State Machine - Principal

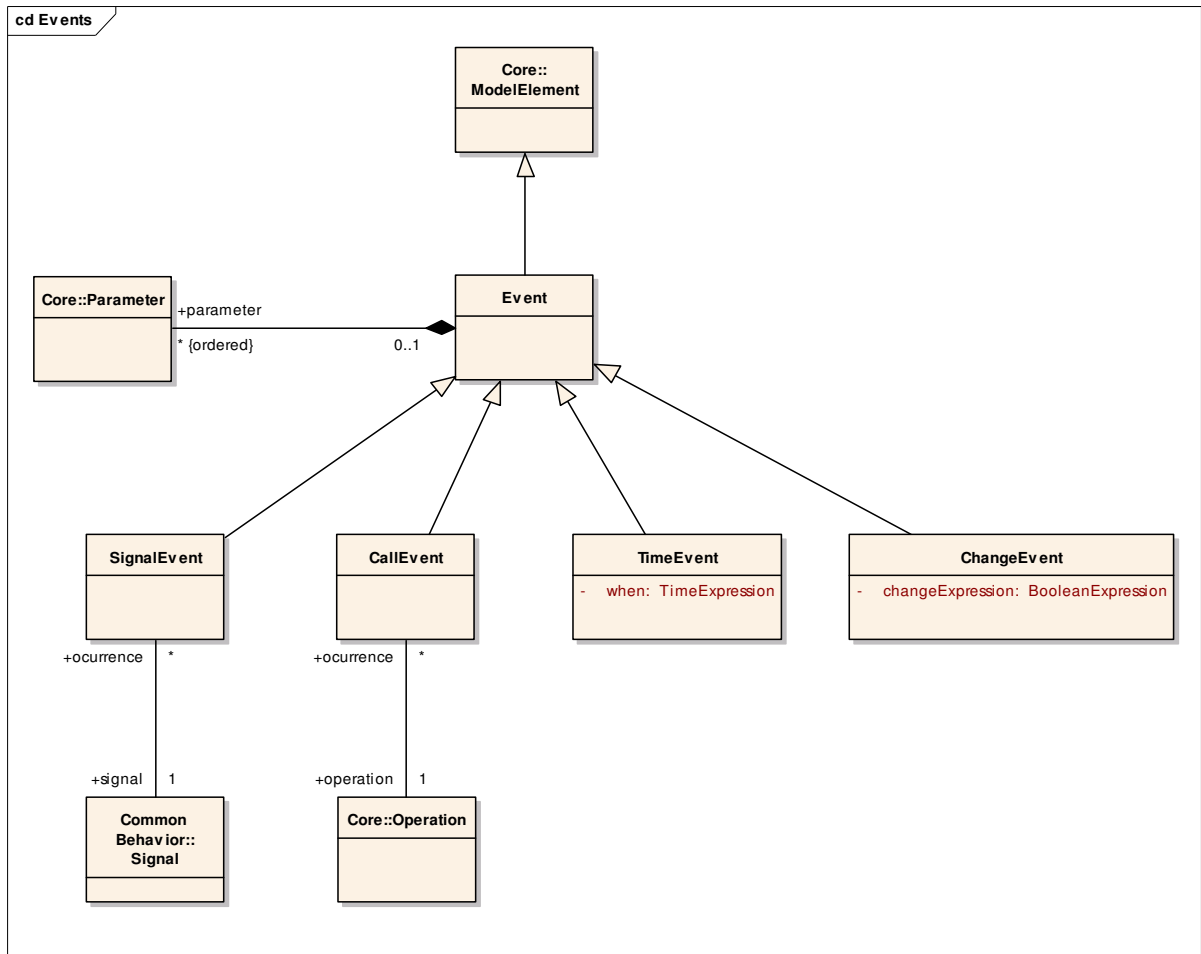


Figura 2: State Machine - Eventos

Estados y Estados Compuestos

Los estados son meta-classes abstractas que modelan situaciones durante las cuales el sistema que se está modelando cumple un invariante. El invariante puede representar situaciones estáticas como por ejemplo, un objeto esperando que un evento externo ocurra; o situaciones dinámicas como el procesamiento de algunas actividades.

Sintácticamente, se los representa con un rectángulo con los vértices redondeados y cada uno tiene un nombre que lo identifica.

Los estados tienen tres asociaciones opcionales. A través de estas asociaciones se pueden especificar actividades que se ejecutan cuando se ingresa al estado, cuando el estado está activo y cuando se sale del estado. Comúnmente se las conoce como "entry", "doActivity" y "exit". La Figura 3 muestra un ejemplo de un estado simple.

Los estados compuestos son estados que contienen a otros estados. La relación es de composición entre el estado compuesto y los estados que contiene. Los estados englobados dentro de un estado compuesto también son llamados sub-estados. En la Figura 4 se puede ver gráficamente un estado compuesto.

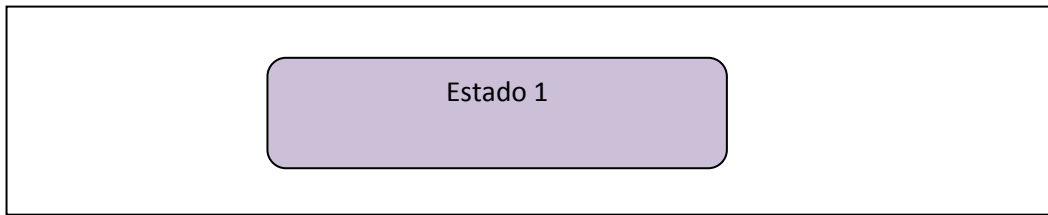


Figura 3: Estado Simple

Un estado compuesto puede ser de dos tipos: concurrente o no concurrente. En el primer caso, el estado compuesto se descompone directamente en dos o más componentes conjuntivos ortogonales conocidos como regiones. Las regiones en general están asociadas a ejecuciones concurrentes. En el segundo caso, el estado compuesto no contiene componentes ortogonales directos en la composición. El estado compuesto mostrado en la Figura 4 es un ejemplo de estado compuesto no concurrente, mientras que en la Figura 5 se puede ver un estado compuesto concurrente.

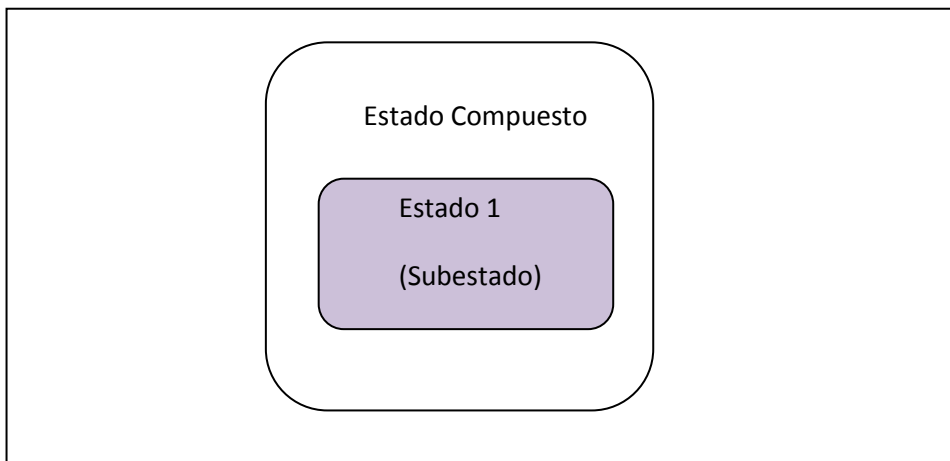


Figura 4: Estado compuesto no concurrente

Además de estos tipos de estados, la especificación de *UML* define algunos *pseudo-estados* que se pueden utilizar para modelar otros tipos de situaciones, como decisiones o combinación de eventos. Entre ellos se pueden mencionar a la historia, tanto superficial como profunda (deep and swallow history), decisión (choice), fork y joins. Estos *pseudo-estados* están fuera del alcance de este trabajo ya que las situaciones que permiten modelar no son fáciles ni naturales de representar con *autómatas temporizados*.

El único pseudo-estado contemplado es el inicial, ya que para una correcta definición de la *state machine* es necesario que cada estado compuesto tenga un estado inicial que es el que permite inicializarlo. En el caso de estados compuestos concurrentes, habrá un estado inicial por región. Un uso particular es cuando se trata del estado *top* que representa a la *state machine* en sí misma y cuyo estado inicial es el que permite inicializarla. Esta propiedad de poder inicializar al estado compuesto se debe a que el estado inicial sólo posee una transición de salida, caso contrario la *state machine* está mal formada. En la Figura 6 se puede ver una *state machine* completa y bien formada.

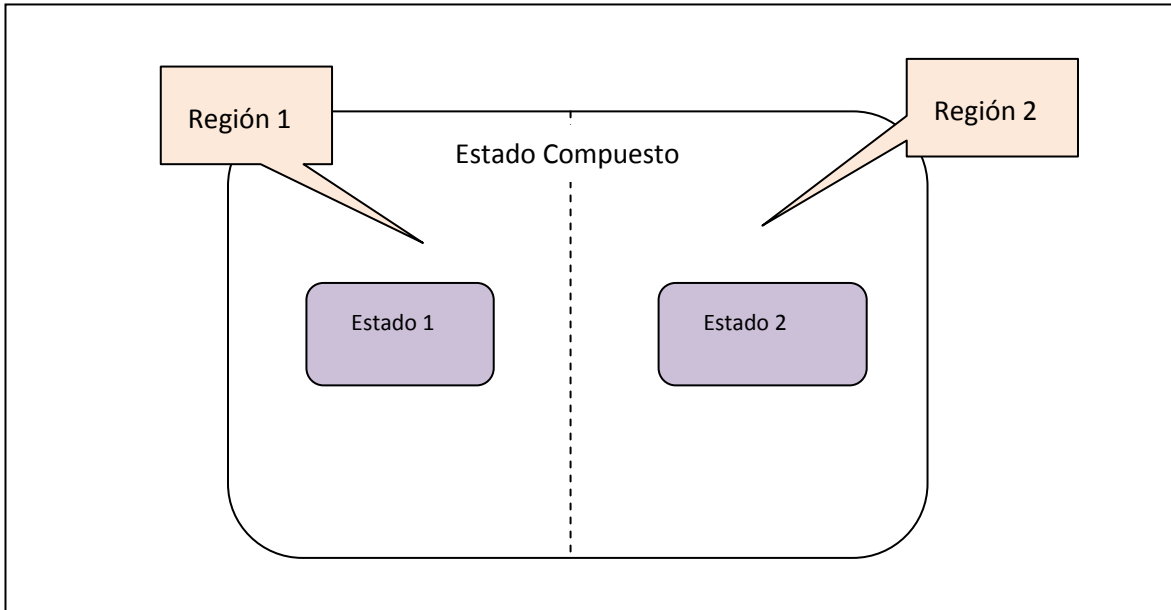


Figura 5: Estado Compuesto Concurrente con 2 regiones

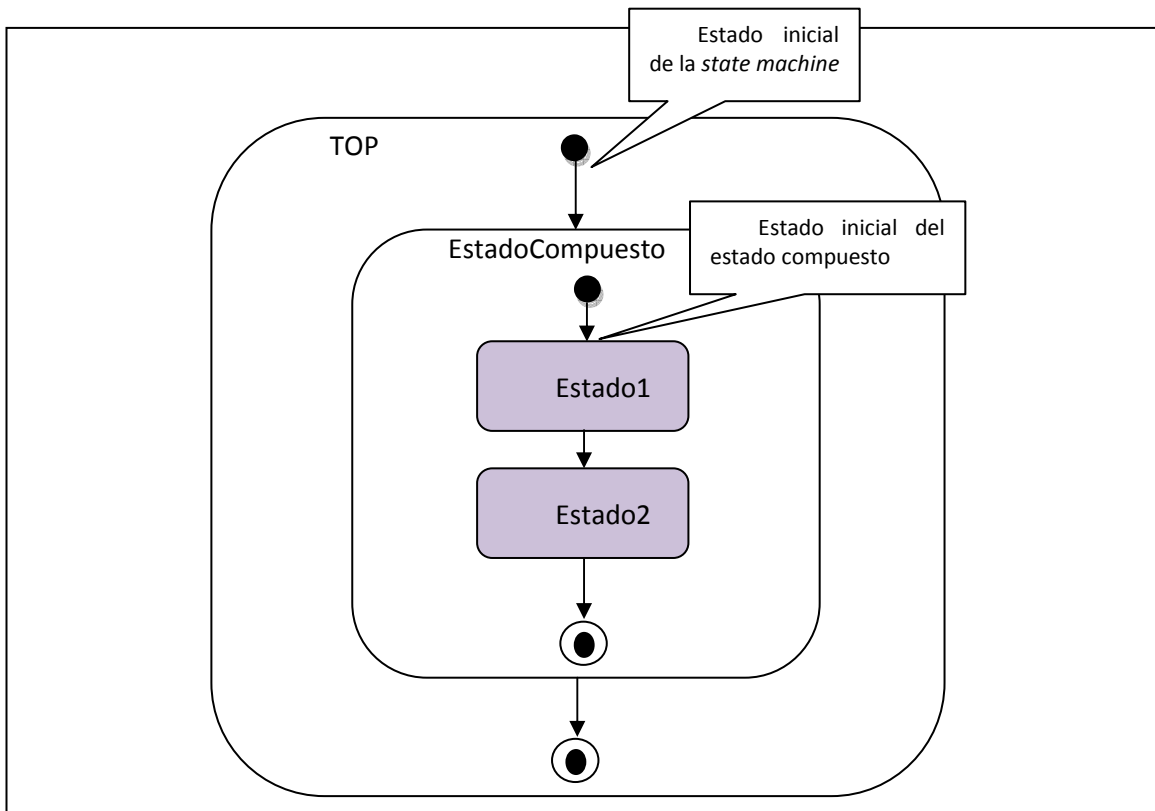


Figura 6: State Machine bien formada

Estado final

Es un estado que indica que el estado compuesto que lo contiene se completó. Si el estado que lo contiene es el estado “top” significa que la *state machine* se completó. Para el caso de estados compuestos no concurrentes, cuando se alcanza el estado final, se lanza en forma automática la transición de completitud. La transición de completitud es una transición de salida del estado compuesto, que no contiene evento disparador ya que se activa al alcanzar el estado final.

Un caso particular son los estados compuestos concurrentes, ya que en este caso la transición de completitud se lanza cuando se alcanza el estado final en todas las regiones. Si una región llega al estado final pero las otras siguen en otros estados, la región se queda en el estado final a la espera de la finalización de las restantes regiones ortogonales.

En la sección “Semántica de la *state machine*” se explica más en detalle el concepto de estado “top” y los estados finales en los distintos estados compuestos.

Transiciones

Una transición es una relación directa entre un estado origen y un estado destino. La transición es la que provee la dinámica a la *state machine*, permitiendo que la misma cambie de un estado a otro. Este cambio de estado representa la respuesta completa de la *state machine* a una instancia particular de un evento.

Una transición contiene:

- Disparador (*Trigger*): evento que activa la transición
- Guarda (*Guard*): un predicado lógico que provee un control de grano fino sobre el disparo de la transición
- Acción (*Effect*): especifica una acción opcional que se ejecuta al disparar la transición

Eventos

Los eventos representan la ocurrencia de alguna situación en el dominio de ejecución de la *state machine*. Como se mencionó anteriormente, se usan como disparadores de las transiciones, que permiten ir de un estado a otro.

Eventos de tiempo – after

Un caso particular pero de gran importancia en el contexto de este trabajo, son los eventos de tiempo. En la especificación de *UML* se identifica un tipo de evento de tiempo conocido comúnmente con la cláusula *after*. A los eventos de tiempo se les permite definir un período de tiempo por el cual se permanece en un estado. Agotado el mismo, se dispara la transición de salida cuyo disparador es el *after*.

Dado que la semántica de este tipo de eventos no está bien definida en la especificación, en el contexto se considera que se utilizan relojes individuales para cada estado que tenga entre sus

transiciones de salida alguna que se active por un evento de tipo “*after*”. Este reloj se inicializa al ingresar al estado y genera el evento de tiempo una vez alcanzado el tiempo especificado.

Semántica de la state machine

La semántica de la *state machine* detallada por *UML 1.4.2* está descrita en términos de la operación de una máquina de estados hipotética que implementa su especificación.

Los componentes claves de esta máquina hipotética son:

- 1) Una cola de eventos que mantiene las instancias de los eventos de entrada hasta que son procesados
- 2) Un mecanismo de disparo de eventos que selecciona los eventos de la cola para su procesamiento
- 3) Un procesador de eventos que procesa los eventos disparados de acuerdo con la semántica general de las *state machines* y la forma específica de la *state machine* en cuestión

Básicamente una *state machine* se utiliza para modelar el comportamiento de un objeto ante el arribo de eventos. En las próximas secciones se detalla cómo se procesan los eventos, pero es importante destacar en este punto que la *state machine* se encuentra en dos posibles estados:

- 1) Procesando un evento, lo que deriva en la ejecución de una transición
- 2) En uno o más estados (dependiendo de que haya o no regiones concurrentes)

En el contexto de este trabajo, la semántica de la *state machine* la analizaremos como un conjunto de ejecuciones de transiciones ya que no modelaremos actividades dentro de los estados. Este conjunto de ejecuciones se determina generando todas las posibles secuencias de arribo de eventos y como se comporta la *state machine* ante cada una de ellas. En la sección “*Procesamiento de eventos: run to completion*” se explica con mayor detalle qué significa una ejecución y la interpretación que usamos de la *state machine* para el desarrollo del algoritmo de traducción.

Semántica de los eventos

Si bien la semántica de UML describe algunos conceptos sobre los eventos, en nuestro caso vamos a trabajar evaluando todas las posibles combinaciones de arribo de eventos, dado que queremos generar un *autómata temporizado* que responda a la ocurrencia de eventos de la misma forma que lo haría la *state machine*, sea cual fuese el orden de ocurrencia de los mismos. Esto conlleva analizar en cada *estado de configuración* válido de la *state machine* como reaccionaría la misma ante la llegada de cada evento posible. En nuestro caso, tomamos un evento por vez, y hasta no completar su procesamiento no continuamos con el siguiente.

Al trabajar con todas las combinaciones posibles de arribo de eventos, no implementamos algunos de los conceptos que detalla la especificación de UML y que comentamos a continuación.

Las instancias de los eventos son generadas como resultado de alguna acción en el sistema o en el ambiente que lo rodea (dominio). Las mismas son transportadas a sus destinos dependiendo del tipo de acción, el destino en sí mismo, las propiedades del medio de comunicación y otros factores. En algunos casos es instantáneo y completamente confiable mientras que en otros puede involucrar

demoras variables de transmisión, pérdida de eventos, re-ordenamiento, o duplicación. *UML* no hace asunciones específicas sobre este punto.

Un evento es recibido cuando se encola en la cola de eventos de su destino, en nuestro caso la cola de eventos de la *state machine*. Un evento es despachado cuando se lo saca de la cola de eventos y es entregado a la *state machine* para su procesamiento. En este punto se lo referencia como evento actual. Finalmente, es consumido cuando se completa el procesamiento del evento. Un evento consumido no está más disponible para su procesamiento. La especificación de *UML* no detalla nada acerca del intervalo de tiempo entre la recepción del evento, el disparo y el consumo, es decir el tiempo que pasa en la cola de eventos, así como en procesarse. Esto permite el modelado de diferentes modelos semánticos.

Semántica de los estados

Los estados pueden estar activos o inactivos, siendo las transiciones las que producen el cambio de situación de los estados. Un estado se convierte en activo cuando se ingresa al mismo como resultado de una transición. Asimismo, un estado se convierte en inactivo cuando se sale del mismo como resultado de una transición.

En una *state machine* que contiene estados compuestos tanto concurrentes como no concurrentes, más de un estado está activo en el mismo momento, el padre y los hijos. Si la *state machine* está en un estado simple contenido en un estado compuesto, entonces todos los estados compuestos que directa o transitivamente contienen al estado simple también están activos.

Como alguno de los estados compuestos puede contener estados concurrentes, el estado activo actual se representa como un árbol de estados comenzando con el estado “top” como el *root* del árbol, hacia estados simples individuales en las hojas. A este árbol que representa los estados activos en un momento dado del tiempo de vida de la *state machine* se lo conoce como *estado de configuración*. Excepto durante la ejecución de las transiciones, los siguientes invariantes se aplican a los *estados de configuración*:

- Si un estado compuesto está activo y no hay concurrencia exactamente uno de sus sub-estados está activo. Por ejemplo en la *state machine* de la figura 6, cuando el estado “EstadoCompuesto” está activo, solo uno de sus sub-estados lo está también, o el estado “Estado1” o el estado “Estado2”
- Si un estado compuesto está activo y tiene concurrencia todas sus regiones están activas. Por ejemplo, en la *state machine* de la Figura 7 cuando está activo el estado “ECompuesto1”, también lo están tanto la región “Region1” como la región “Region2”; y dentro de las regiones, un estado de cada una de ellas está activo

Procesamiento de eventos: run-to-completion

Como se mencionó anteriormente, el algoritmo que se presenta en este trabajo se basa en evaluar todas las combinaciones posibles de llegada de eventos. Por tal motivo, algunas de las definiciones de la especificación de *UML* no fueron implementadas porque aplican a la ejecución del objeto que modela la *state machine*. Sin embargo, sí se tomaron para aplicar en cada paso del algoritmo como se verá más adelante.

La semántica definida en la especificación de *UML* para el procesamiento de los eventos se basa en las siguientes premisas:

- Los eventos se procesan uno a la vez
- No está definido el orden para desencolar los eventos, para poder modelar diferentes esquemas basados en prioridades. En nuestro caso no importa el orden ya que estaremos trabajando con todas las posibles combinaciones de eventos, por lo que soporta cualquier implementación
- Un evento puede ser desencolado y despachado sólo si el evento anterior estuviese completo. En nuestro caso, trabajaremos con un evento por vez, respetando este concepto
- Antes de comenzar un paso, la *state machine* está en una configuración estable con todas las acciones completas. En nuestro caso, al trabajar con un evento por vez, hasta que no se terminó su procesamiento no se continúa con el siguiente

La ejecución de una *state machine* según el estándar UML 1.4.2 se realiza procesando los eventos, uno por vez, aplicando lo que se conoce como “Run-to-completion step”, que implica procesar un evento solamente cuando la *state machine* se encuentra en un estado válido, y luego de procesar un evento, pasa a otro estado válido.

Una *state machine* se dice que está en un estado válido cuando se están ejecutando las actividades de uno o más estados, dependiendo que haya o no regiones concurrentes. Cada estado válido de una *state machine* se representa con un árbol de *estado de configuración*, descrito anteriormente. En las Figuras 7 y 8 se muestra una *state machine* y el árbol de *estado de configuración* que correspondiente al momento en que esta activo el estado *ECompuesto1*, y los estados *Estado2* y *Estado3*. Cabe notar que en general el estado *top* no es dibujado por las herramientas de diseño, sino que se lo supone como el contenedor de la *state machine*.

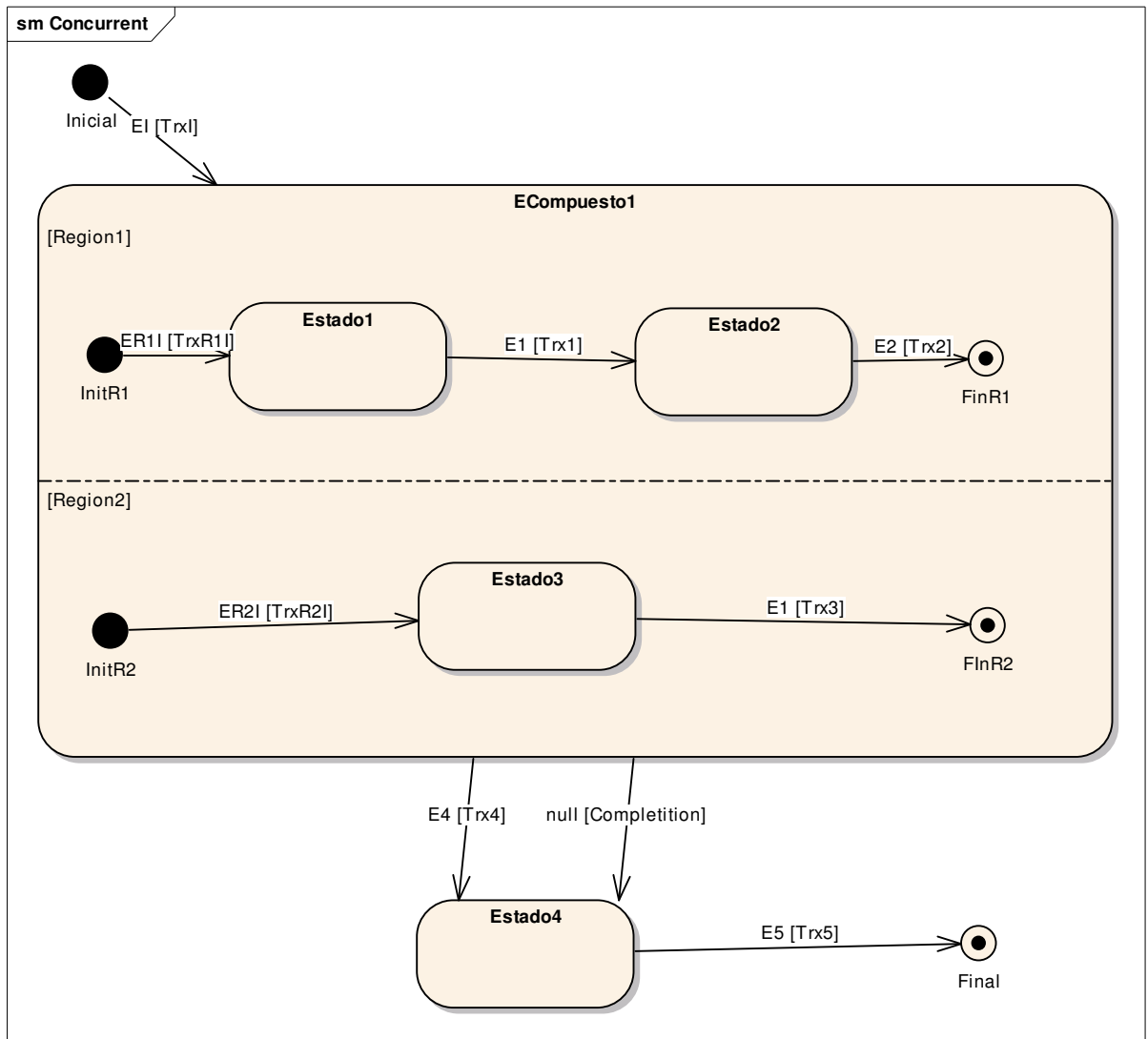


Figura 7: Ejemplo de una *state machine* con estados concurrentes

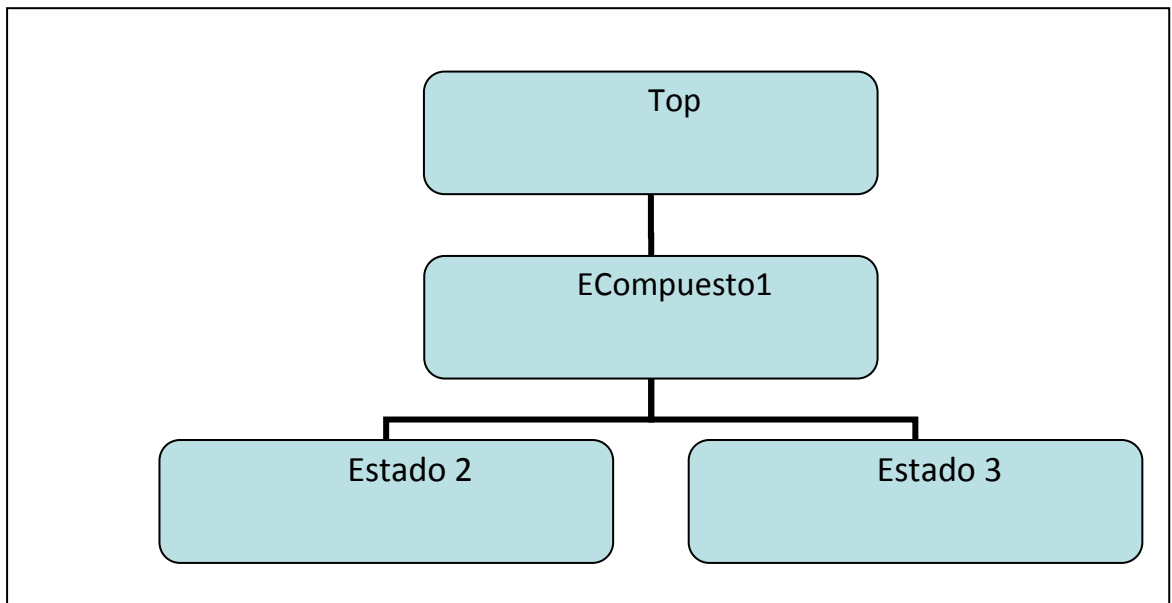


Figura 8: Estado de configuración de la state machine de la Figura 7

Transiciones en Conflicto

Se ha mencionado anteriormente que más de una transición puede ser habilitada por la llegada de un evento en una *state machine*. Si esto ocurre, entonces puede ser que dichas transiciones estén en conflicto entre sí. Por ejemplo, considerar el caso en que dos transiciones se originan en el mismo estado, son disparadas por el mismo evento, pero tienen diferentes guardas. Si el evento ocurre y ambas guardas son verdaderas, entonces sólo una transición se va a disparar. En otras palabras, en el caso de transiciones en conflicto, una sola de ellas será disparada en un único paso del *run-to-completion*, según el manejo de prioridades descrito más abajo.

Dos transiciones se dicen en conflicto si las dos salen del mismo estado, o, más precisamente, que la intersección del conjunto de estados que originan las mismas no es vacío. Solamente transiciones que ocurren en regiones ortogonales pueden ser disparadas simultáneamente. Esta restricción garantiza que el nuevo *estado de configuración* resultante de ejecutar el conjunto de transiciones esté bien formado.

Prioridad de Disparo

En la situación en que hay transiciones en conflicto, la selección de qué transiciones se van a disparar está basada en parte en una prioridad implícita. Esta prioridad resuelve algunos conflictos, pero no todos. La prioridad se basa en la posición relativa en la jerarquía de estados. Por definición, una transición que se origina en un sub-estado tiene más prioridad que una transición en conflicto originada en alguno de los estados que la contienen.

Algoritmo de selección de Transición

El conjunto de transiciones que se van a disparar es el conjunto maximal de transiciones que satisfacen las condiciones siguientes:

- Todas las transiciones en el conjunto están habilitadas

- No hay transiciones en conflicto dentro del conjunto
- No hay una transición fuera del conjunto con prioridad más alta que una transición en el conjunto (esto es, transiciones habilitadas con prioridades más altas están en el conjunto mientras que las transiciones conflictivas con menor prioridad son dejadas fuera del mismo)

La especificación de *UML* sugiere que esto puede implementarse fácilmente con un algoritmo de selección goloso, con una búsqueda desde las hojas del árbol que representa el estado de configuración hacia la raíz del mismo, en forma transversal. Los estados en el *estado de configuración activo* son recorridos comenzando con los más internamente anidados y moviéndose hacia el estado *top*. Para cada estado en un nivel dado, todas las transiciones de salida son evaluadas para determinar si alguna se activa con el evento. Este recorrido garantiza que no se viole ninguno de los principios que asignan la prioridad a las transiciones. El único punto no trivial a resolver es el conflicto de transiciones entre estados ortogonales en todos los niveles. Esto se resuelve terminando la búsqueda en cada estado ortogonal una vez que alguna transición dentro de algunas de sus regiones es disparada.

4. Autómata Temporizado

Un *autómata temporizado*⁹ es esencialmente una máquina de estados finitos no determinística, extendida por un conjunto finito de relojes con valores reales. Este *autómata* puede ser considerado como un modelo abstracto de un sistema temporizado. Las variables modelan los relojes lógicos en el sistema, que son inicializados en cero cuando el sistema comienza, y luego se incrementan sincrónicamente al mismo ritmo. Los relojes son utilizados para medir, por ejemplo, el tiempo transcurrido entre dos eventos, el tiempo de espera o la demora de una comunicación.

Para restringir el comportamiento de un autómata se usan las restricciones de relojes, vistas como guardas en los ejes. Un eje representa una transición que puede ser tomada cuando los valores de los relojes satisfacen la guarda etiquetada en el eje. Los relojes pueden volver a cero cuando una transición es tomada, dependiendo de que el eje contenga el *reset* del mismo.

Los nodos pueden ser asociados con invariantes de la forma $x \sim c$, donde x es un reloj, c es una constante entera, y $\sim \in \{<, \leq\}$.

Los ejes entre los nodos son etiquetados con tuplas (gd, ac) donde:

- *gd*: representa la guarda del eje, expresado como una conjunción de restricciones temporales $x \sim c$ o $x - y \sim c$ donde x e y son relojes, c es una constante entera, y $\sim \in \{<, \leq, =, \geq, >\}$ es una relación binaria.
- *ac*: es un conjunto de operaciones “reset” $x := c$ sobre los relojes.

Relojes

A continuación introducimos varias definiciones asociadas con los relojes.

Definición 4-1

Un **conjunto de relojes** X es un conjunto de variables $\{x_1, x_2, \dots, x_n\}$ cuyos valores pertenecen a un dominio D de reales no negativos.

Una **valuación** $v: X \rightarrow D$ es una función total que denota los valores de los relojes, tal que $v(x_i)$ es el valor asociado al reloj x_i . Llamamos V_X al conjunto de valuaciones posibles de X .

Dado $v \in V_X$ y $t \in D$, $v + t$ es la valuación que asigna a cada reloj $x \in X$ el valor $v(x) + t$.

Definición 4-2

Dado un conjunto de relojes X , un subconjunto $\alpha \subseteq X$ y una valuación v , definimos “reseteo de relojes de α ” $Reset_\alpha(v)$ como la valuación que asigna cero a todos los relojes en α y mantiene los mismos valores que v para el resto de los relojes. En particular, $\mathbf{0} \in V_X$ es la función que establece el valor 0 para todos los relojes de X .

Definición 4-3

Dado un conjunto de relojes X , definimos una **restricción sobre los relojes** φ de acuerdo con la siguiente gramática:

$$\varphi ::= x \# c \mid x - y \# c \mid x + c \# y + d \mid \neg \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

donde:

$$x, y \in X$$

c, d son números naturales

$$\# \in \{ <, \leq, >, \geq, = \}$$

$\varphi \in \Phi$ es un elemento del conjunto de restricciones

Notaremos Φ a un conjunto de restricciones sobre X o **restricciones temporales**.

Una valuación $v \in V_X$ **satisface** una restricción $\varphi \in \Phi$ (notado $v \models \varphi$) si y sólo si:

$$\begin{aligned} v \not\models x \# c & \Leftrightarrow v(x) \# c \\ v \not\models x - y \# c & \Leftrightarrow v(x) - v(y) \# c \\ v \not\models x + c \# y + d & \Leftrightarrow v(x) + c \# v(y) + d \\ v \not\models \neg \varphi & \Leftrightarrow \neg(v \models \varphi) \\ v \not\models \varphi_1 \wedge \varphi_2 & \Leftrightarrow v \not\models \varphi_1 \wedge v \not\models \varphi_2 \\ v \not\models \varphi_1 \vee \varphi_2 & \Leftrightarrow v \not\models \varphi_1 \vee v \not\models \varphi_2 \end{aligned}$$

Autómata Temporizado

Definición 4-4

Un *autómata temporizado* o TA es una tupla $A = \langle S, X, \Sigma, E, I, s_0 \rangle$ dónde:

S es un conjunto finito de nodos

X es un conjunto finito de relojes

Σ es un conjunto de etiquetas

E es un conjunto finito de ejes o transiciones

$I: S \rightarrow \Phi$ es una función total llamada “invariante del nodo” que asocia a cada nodo una restricción temporal

s_0 es el nodo inicial

Un eje o transición $e \in E$ es una tupla $\langle s, a, \varphi, \alpha, s' \rangle$ dónde:

$s \in S$ es el nodo origen

$s' \in S$ es el nodo destino

$a \in \Sigma$ es la etiqueta

φ es una restricción que llamaremos “guarda”

$\alpha \subseteq X$ es el subconjunto de relojes que se “resetean” (vuelven a cero) en el eje

Es necesario aclarar que no utilizamos la noción de nodos finales, ya que las ejecuciones con las que trataremos en el presente trabajo son infinitas.

Dado un *autómata temporizado* $A = \langle S, X, \Sigma, E, I, s_0 \rangle$ notamos

$$Locs(A) = S,$$

$$Clocks(A) = X,$$

$$Labels(A) = \Sigma,$$

$$Edges(A) = E,$$

$$Iv(A) = I,$$

$$Init(A) = s_0.$$

Dado un eje $e = \langle s, a, \varphi, \alpha, s' \rangle \in E$ notamos

$$Src(e) = s,$$

$$Label(e) = a,$$

$$Guard(e) = \varphi,$$

$$Reset(e) = \alpha,$$

$$Tgt(e) = s'.$$

Semántica

El modelo semántico detrás de los *autómatas temporizados* aquí descritos es un modelo de eventos instantáneos. Los ejes modelan la ocurrencia de estos eventos, mientras que los relojes se utilizan para medir el tiempo que transcurre entre estas ocurrencias. Cada eje puede tener asociado un conjunto de relojes, notados entre llaves $\{\}$, que son los relojes que se resetean al realizar la transición.

Se puede asociar a un eje una pre-condición temporal. Este hecho se representa a través de una guarda que restringe el valor de uno o más relojes. Por lo tanto, la transición puede efectuarse sólo

si la guarda es verdadera. En este caso, la transición se ejecuta instantáneamente, reseteando los valores de los relojes correspondientes.

Con respecto a los nodos, también es posible asociar condiciones temporales a los mismos, a través de los invariantes, restringiendo los valores de relojes permitidos en cada nodo. En general, se utiliza el concepto de invariante para expresar el máximo tiempo que se puede permanecer en un nodo, o sea su *deadline*.

Un *autómata temporizado* comienza en el nodo s_0 con todos sus relojes inicializados en cero. Puede moverse a través de un eje $\langle s, a, \varphi, \alpha, s' \rangle$ sin que pase el tiempo, por la instantaneidad de ejecución de las transiciones que se explicó anteriormente. Como veremos más adelante, el tiempo sólo transcurre en los nodos.

Los valores de todos los relojes se incrementan uniformemente con el tiempo y, en cualquier instante, el valor de un reloj es igual al tiempo que pasó desde la última vez que se lo reseteó. El *autómata temporizado* puede ejecutar una transición solamente si la restricción temporal φ asociada a su eje se satisface con los valores actuales de los relojes. Los *autómatas temporizados* pueden permanecer en un nodo dado s , pero no pueden estar más tiempo que el límite impuesto por la restricción de tiempo $I(s)$ asociada a él.

En cualquier instante, el estado del sistema puede ser descrito totalmente por el nodo actual del *autómata temporizado* y los valores de sus relojes. Por lo tanto, cuando el sistema se encuentra en un estado $s = (l, v)$, éste puede permanecer en el nodo l dejando pasar el tiempo, mientras la valuación corriente de los relojes satisfaga el invariante.

Ejemplo de autómata temporizado

La Figura 9 muestra un *autómata temporizado* que representa el comportamiento de una expendedora de bebidas. La máquina espera una moneda. Luego permite la selección de la opción té o café durante 15 seg. Si a los 15 seg, no se seleccionó ninguna opción, retorna la moneda. Si se seleccionó una bebida, ésta se prepara en a lo sumo 3 seg.

La llegada de la moneda es un evento sin restricciones, que resetea el reloj x . Este reloj permite contar el tiempo que tiene el usuario para elegir la bebida. Una vez seleccionada, por cualquiera de las opciones se resetea el reloj z , que permite modelar el tiempo de preparación de la bebida. Cuando la bebida está lista, el sistema vuelve a su estado inicial.

El invariante del nodo "opción" modela la espera de hasta 15 seg para que el usuario seleccione la bebida. El invariante del nodo "preparar" representa el hecho de que, una vez elegida, la bebida se prepara en a lo sumo 3 seg.

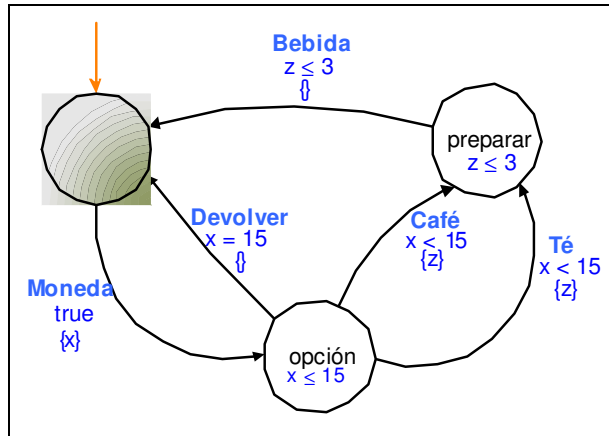


Figura 9: *Autómata temporizado* representando una Expendedora de bebidas

5. Reglas de traducción

Los algoritmos que se presentarán en las próximas secciones realizan la traducción de la *state machine* basándose en la ejecución de la misma. El *autómata* resultante de la traducción representa todas las posibles ejecuciones de la *state machine*, ante cualquier combinación de eventos.

Se presentarán dos algoritmos. En el primero se realiza la traducción de una *state machine* a un *autómata finito* (sin relojes). En el segundo, se agrega el manejo de los relojes, para lograr como resultado un *autómata temporizado*. Esta división en etapas para la elaboración del algoritmo se debió a los dos problemas más importantes a enfrentar para lograr la traducción:

- Aplanamiento de la estructura jerárquica de la *state machine* sin perder semántica
- Utilización de los relojes, tratando de minimizar el uso de los mismos

El primer punto está relacionado con los estados compuestos y la flexibilidad que brindan las *state machines* para modelar situaciones complejas, dividiendo esa complejidad en sub-estados. Para los *autómatas* fue necesario “aplanar” estos estados compuestos, sin modificar el comportamiento.

El segundo punto se relaciona a la utilización de herramientas de *model checker*. Estas herramientas⁹ tienen una complejidad exponencial en función de la cantidad de relojes, por lo que es necesario minimizar el uso de los mismos.

Dado que la primera versión del algoritmo genera un *autómata finito* y la segunda versión un *autómata temporizado*, pero el segundo usa el mismo concepto para la creación de los nodos del *autómata finito*, en la explicación siguiente haremos uso del término *autómata* a secas, ya que los conceptos de alto nivel explicados a continuación aplican a ambas versiones del algoritmo.

El concepto básico detrás de la traducción es lograr un algoritmo que crea el *autómata* de una forma tal que cada nodo representa a un *estado de configuración* válido de la *state machine*. Por lo tanto, el *autómata* resultante representa la evolución de la *state machine*, desde el estado de configuración inicial al final, ante toda posible combinación de eventos.

Para mostrar en qué se basa nuestra traducción, tomaremos la *state machine* de la Figura 7 para mostrar cómo se realiza la traducción. En dicha *state machine*, el estado de configuración inicial es el árbol cuyas hojas son los estados Estado1 y Estado3, representado en la Figura 10. Esto se debe a que, como se explicó anteriormente, la *state machine* cuenta con un sólo estado inicial, que a su vez tiene una sola transición de salida que permite inicializarla. De igual manera los estados compuestos tienen esta misma propiedad, salvo en el caso de estados compuestos concurrentes que cuentan con más de un estado inicial, en particular uno por cada región ortogonal.

En el *autómata* que se genera se agrega el primer nodo que representa este *estado de configuración inicial*. Por la semántica explicada en la sección “Procesamiento de eventos: run-to-completion”, la *state machine* solo avanzará ante el arribo de dos eventos: E1 o E4. Cualquier otro evento distinto a estos será descartado ya que no activa ninguna transición de salida.

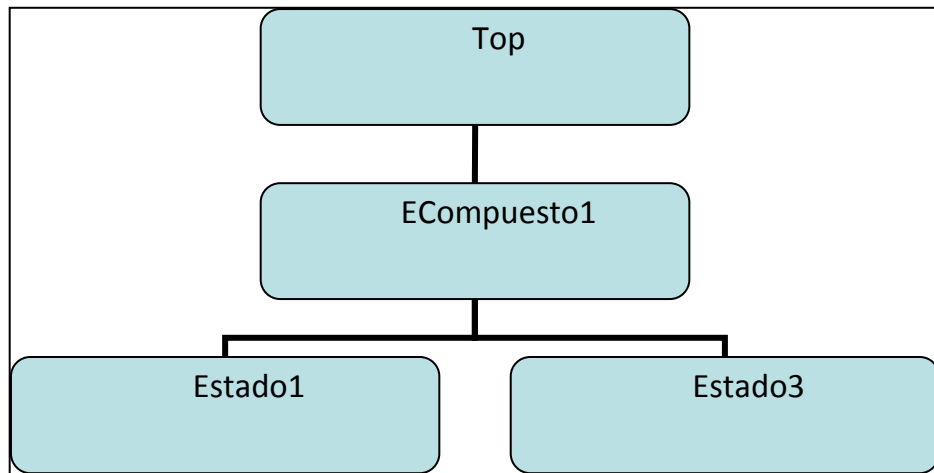


Figura 10: Estado de configuración inicial de la state machine de la Figura 7

Volviendo al *estado de configuración inicial* y suponiendo que arriba el evento E1, éste activa las transiciones Trx1 y Trx3 de la *state machine*, quedando ésta en el *estado de configuración* mostrado en la Figura 11, cuyas hojas son Estado2 y FinR2. Si en lugar de llegar el evento E1, hubiera llegado el evento E4 estando en el *estado de configuración inicial*, la transición que se activa es la Trx4 llevando a la *state machine* al *estado de configuración* mostrado en la Figura 12, con el estado Estado4 como hijo. El estado de configuración está conformado por las hojas del árbol que lo representa, pero cada estado sabe cuál es su padre, con lo cual ante el arribo del evento E4, que no dispara transiciones en las hojas, se busca en sus padres y allí aparece la transición Trx4.

Estos dos posibles avances de la *state machine* hacen que se agreguen al *autómata* dos nodos, representando a cada uno de los posibles avances y los respectivos ejes con el evento como guarda. La Figura 13 muestra cómo va quedando el *autómata* hasta este punto.

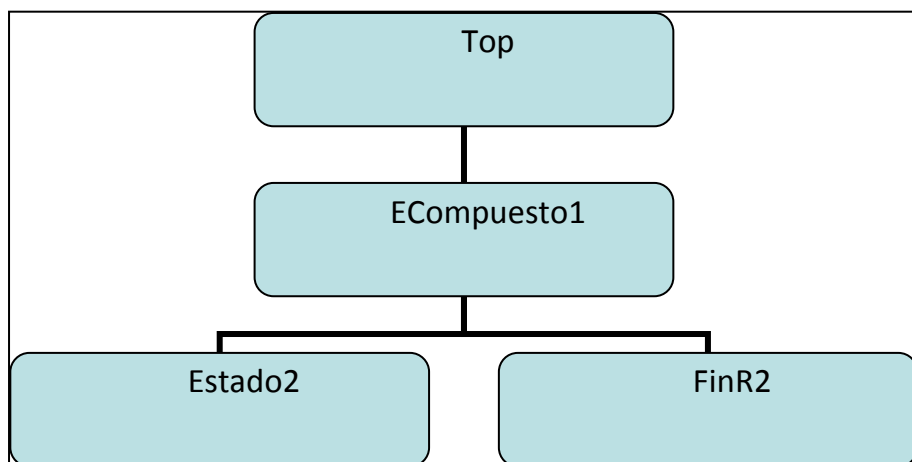


Figura 11: Estado de Configuración cuando se procesa el evento E1 desde el estado de configuración inicial

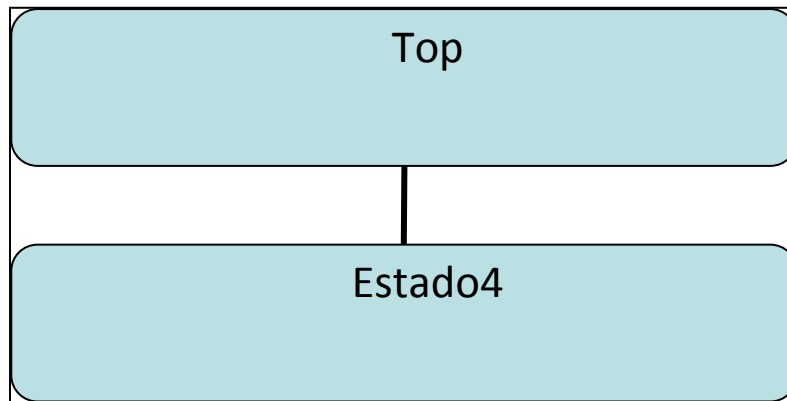


Figura 12: Estado de Configuración cuando se proceso el evento E4 desde el estado de configuración inicial

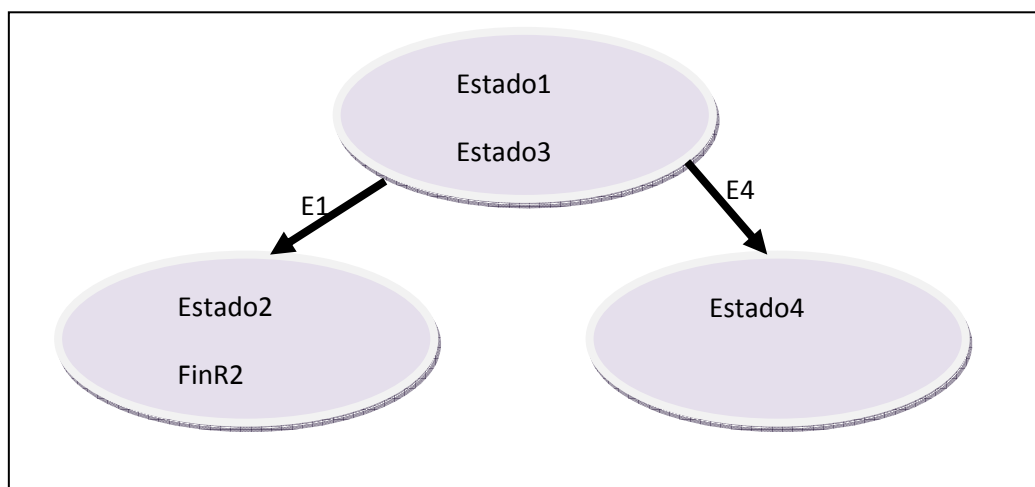


Figura 13: Autómata tras el procesamiento de los eventos E1 y E4

Los nodos que se van agregando al *autómata* quedan pendientes de análisis, ya que para continuar trabajando en la evolución de la *state machine* hay que analizar desde cada posible *estado de configuración* válido (representado por los nodos del *autómata*) cómo reacciona ante los eventos. Si tomamos el estado de configuración de la Figura 11, los eventos a los cuales puede reaccionar son E2 (activa la transición Trx2) y E4 (activa la transición Trx4). En el caso de que llegara E4, el *estado de configuración* resultante de procesar el evento es el mismo que se muestra en la Figura 12, por lo que no se agrega un nodo al *autómata* pero si se agrega el eje desde el nodo que representa el *estado de configuración* con hojas Estado2 y FinR2 al nodo que representa el *estado de configuración* con hoja Estado4. Si el evento que llega es el evento E2, tras procesarlo, como se alcanzan los finales de las dos regiones, se dispara la transición de completitud del estado compuesto (ver sección “Estado Final”) en forma atómica, llevándonos al mismo nodo que con el evento E4. Por lo tanto sólo se agrega un nuevo eje entre los mismos nodos pero con el evento E2 como guarda.

De esta forma se continúa con el resto de los nodos hasta que todos sean analizados. El *autómata* resultante de la traducción es el que se muestra en la Figura 14.

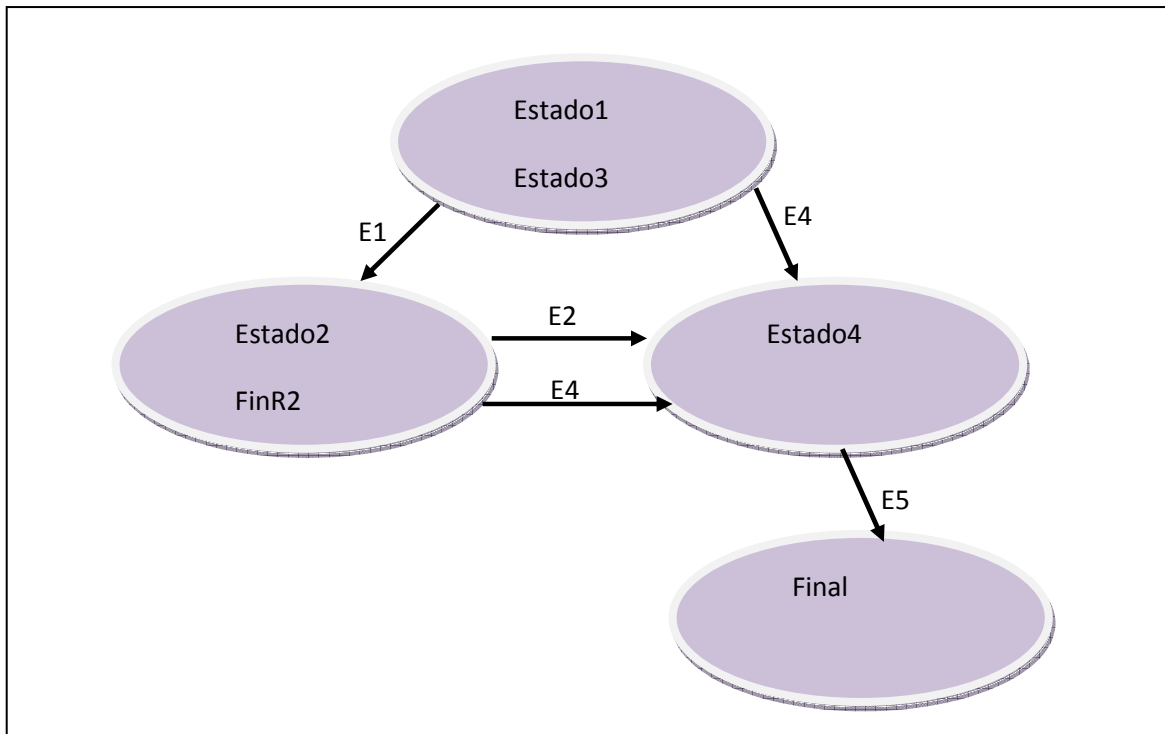


Figura 14: *Autómata* resultante de la traducción de la *state machine* de la Figura 7

6. Primera versión: De state machines a autómatas finitos

El trabajo de elaboración del algoritmo se realizó en forma incremental. En primer lugar se resolvió la problemática de aplanar la *state machine* e identificar todos los posibles estados de configuración válidos.

Para un mejor entendimiento de los algoritmos y el mapeo entre la *state machine* y el *autómata finito / temporizado*, se define la siguiente nomenclatura:

- Cuando se habla de **estados** y **transiciones**, se hace referencia a los componentes de la *state machine*.
- Cuando se habla de **nodos** y **ejes**, se hace referencia a los componentes del *autómata finito / temporizado*.

La forma de operación del algoritmo es iterar por todos los *estados de configuración* válidos, y analizar hacia dónde avanza cada uno, ante cada posible evento de la *state machine*. Cada nodo del *autómata* resultante representa un posible *estado de configuración* válido de la *state machine*. El resultado de este “avance” puede ser:

- El agregado de un nuevo nodo al *autómata*, con su respectivo eje que vincula el nodo en análisis con el nuevo o,
- El agregado de un nuevo eje entre el nodo que se analiza y uno ya existente dentro del *autómata* (si existe es porque se creó en alguna otra iteración como resultado del avance de algún otro estado de configuración).

Como se mencionó en la sección “UML State Machine”, cada uno de los *estados de configuración* se representa como un árbol de estados activos. El árbol se conforma en base al anidamiento de estados y regiones concurrentes. Las hojas del árbol son siempre estados simples.

Algoritmo principal

A continuación se detallan algunas referencias usadas a lo largo del algoritmo:

- AF (*Autómata Finito*): Es el *autómata* que genera el algoritmo. Compuesto de **nodos** y **ejes**.
- SM (*State Machine*): Es la *state machine* a procesar. Compuesta de **estados** y **transiciones**.

La estructura de la *state machine* es una clase que contiene:

- Lista de estados, pertenecientes al nivel más alto de la jerarquía.
- Lista de transiciones, pertenecientes al nivel más alto de la jerarquía.
- Los sub-estados figuran en la lista de hijos del estado padre.

Tanto los estados, como las transiciones tienen un ID único que los identifica. Los estados saben cuál es su padre, salvo el TOP que no tiene padre.

A continuación se detalla el significado de algunas de las variables utilizadas en el pseudo-código:

- **ASC** (Active State Configuration): Estado de configuración ⁽¹⁾ que se analiza.
- **NSC** (New State Configuration): Estado de configuración ⁽¹⁾ al cuál avanzo después de ejecutar las transiciones desde el **ASC**.
- **NA** (Nodo Activo): Nodo del AF que está en proceso. El mismo representa al **ASC** que estoy evaluando.
- **NN** (Nuevo Nodo): Nodo que se va a crear en el AF correspondiente al nuevo estado de configuración (**NSC**).
- **CEV** (Conjunto Eventos): Conjunto de todos los eventos posibles que pueden ocurrir en la *state machine* que estamos traduciendo.
- **EA** (Evento Actual): Evento que se procesa en un momento dado.
- **CTX** (Conjunto de Transiciones): Transiciones que se activan por la ocurrencia del evento que está en proceso (**EA**).
- **NE** (Nuevo Eje): Eje que se crea para unir al Nodo Activo (**NA**) con el Nuevo Nodo (**NN**) en el AF.

⁽¹⁾ Los estados de configuración se representan sólo almacenando los estados hojas del árbol de configuración, dado que sus padres pueden ser calculados.

Cuerpo principal del algoritmo

El algoritmo toma como entrada una *state machine* (SM) y retorna un *autómata finito* (AF), que modela todas las posibles ejecuciones de la *state machine*. En esta versión, tomamos como entrada una *state machine* que no posee *TimeEvents*, con lo cual no es necesario contemplar el uso de relojes para modelar el paso del tiempo.

Como se mencionó anteriormente, el procesamiento de un evento respeta el *run-to-completion step*, que implica procesar un evento solamente cuando la *state machine* se encuentra en un *estado de configuración* válido. Al procesar el evento se pasa a otro *estado de configuración* válido, con lo cual, en el *autómata* que representa la ejecución de la *state machine*, lo que se hace es simbolizar en cada nodo un estado válido de la ejecución de la *state machine*, y que por cada uno de los posibles eventos que se puedan recibir estando en dicho estado de configuración, se represente cuál es el *estado de configuración* destino resultante de ejecutar el evento E en el estado original. Al tener todas las combinaciones de estados de configuración / eventos, se infieren todas las posibles ejecuciones.

En la transcripción del pseudo-código (ver más adelante), se muestra la función principal del algoritmo, cuya primera acción es obtener el *estado de configuración inicial (línea 1)*, es decir, calcular cuál es el primer *estado de configuración* válido de la *state machine* cuando se inicia la

ejecución. Éste es siempre el mismo, ya que por reglas de formación sólo puede existir un estado inicial y sólo una transición puede salir del mismo.

La *línea 2* crea el nodo del *autómata* que representa a dicho *estado de configuración*, ya que cada nodo del *autómata* referencia sólo a uno de dichos estados.

En la *línea 3* se marca como pendiente al nodo del *autómata* para poder procesarlo a su ingreso en el ciclo principal. Posteriormente, en la *línea 4* se lo marca como nodo inicial y por último, en la *línea 5* se agrega al *autómata*.

El ciclo principal del algoritmo (*línea 6*) toma progresivamente todos los *estados de configuración* pendientes y, por cada uno de ellos (representado el actual por la variable ASC), itera sobre cada evento posible de la *state machine* (*línea 7*), representando EA al evento actual (*línea 8*). Para cada EA, se calcula el *estado de configuración* destino.

Dado que no todos los estados de configuración reaccionan a todos los eventos, en la *línea 9* se obtiene el conjunto de transiciones que son disparadas por el evento EA en los estados activos de ASC.

En la *línea 10* se verifica que exista alguna transición, de no ser así, se vuelve a tomar otro evento de la lista, ya que significa que el evento actual no activa ninguna transición de salida del *estado de configuración* que se está analizando. En caso afirmativo, se crea el *estado de configuración destino* (*línea 11*), que es el resultado de ejecutar las transiciones que están en el conjunto CTX desde el *estado de configuración* activo ASC.

En virtud del procedimiento utilizado para el análisis de cada uno de los nodos (*estados de configuración*) podría ocurrir que el *estado de configuración* resultante ya hubiera sido creado con anterioridad, por ende, la primera acción llevada a cabo en la *línea 12*, será buscar en el AF (*Autómata Finito*) si el nodo que representa al *estado de configuración* calculado como destino, ya existe. En caso de existir previamente, no es necesario agregar el nodo al AF, lo único que hay que hacer es crear el eje que une al nodo que identifica al estado de configuración ASC, con el nuevo estado de configuración NSC, representado por el nodo NN (Nuevo Nodo) -*líneas 21 y 22*- y continuar con el procesamiento de eventos.

Dado el caso de que no existiera el nodo que representa al nuevo *estado de configuración* destino (NSC), se lo creará en la *línea 14*. En la *línea 15* es marcado como pendiente, y en las *líneas 16, 17 y 18* se verifica si es el nodo final de la *state machine* y posteriormente se lo marca como pendiente. Por último, en la *línea 19*, se agrega al nuevo nodo NN (Nuevo Nodo) al AF (*Autómata Finito*). Como en el caso anterior, en las *líneas 21 y 22* se crea el eje que une a los dos nodos y se lo agrega al AF.

Cuando se termina de procesar todos los eventos de CEV (Conjunto Eventos), se marca el nodo como procesado (*línea 25*) y después tomamos otro nodo sin procesar del AF (que representa un *estado de configuración*), para seguir evaluando todas las posibilidades. Estas tareas se ven reflejadas en las *líneas 26 y 27*.

A continuación se encuentra el pseudocódigo descrito anteriormente:

```
-- ASC: Active State Configuration
-- NA: Nodo Activo
-- CEV: Conjunto Eventos
-- EA: Evento Actual
-- CTX: Conjunto de Transiciones
-- NSC: New State Configuration
-- NN: Nuevo Nodo
-- NE: Nuevo Eje
Línea 1     ASC = obtenerEstadoConfiguracionInicial()
Línea 2     NA = crearNodo(ASC)
Línea 3     marcarNodoPendiente(NA)
Línea 4     marcarNodoInicial(NA)
Línea 5     agregarNodo(NA)
Línea 6     Mientras ASC != vacío Hacer
Línea 7         Para cada Evento en CEV Hacer
Línea 8             EA = CEV.Proximo()
Línea 9             CTX = obtenerTransiciones(ASC, EA)
Línea 10            Si CTX != vacío Entonces
Línea 11                NSC = crearEstadoConfiguracionDestino(ASC, CTX)
Línea 12                NN = obtenerNodoEquivalente(NSC)
Línea 13                Si NN == vacío Entonces
Línea 14                    NN = crearNodo(NSC)
Línea 15                    marcarNodoPendiente(NN)
Línea 16                    Si estadoFinal(NCS) Entonces
Línea 17                        marcarNodoFinal(NN)
Línea 18                    Fin Si
Línea 19                    agregarNodo(NN)
Línea 20                Fin Si
Línea 21                NE = crearEje(NA, NN, EA)
Línea 22                agregarEje(NE)
Línea 23            Fin Si
Línea 24        Fin Para
Línea 25        marcarNodoProcesado(NA)
Línea 26        NA = obtenerNodoPendiente
Línea 27        ASC = obtenerEstadoConfiguracion(NA)
Línea 28        Fin Mientras
```

Como puede observarse, se utilizan algunas funciones auxiliares que son importantes para que la traducción propuesta sea más clara. A continuación se explican las más significativas.

Función: ObtenerTransiciones

Dado un estado de configuración EC y un evento E (parámetros de entrada), esta función devuelve el conjunto de transiciones que se activan por el arribo de dicho evento.

Para analizar las transiciones que se activan, se deben considerar 3 casos dependiendo del tipo de *estado de configuración* que se evalúa. El más sencillo (caso 1), es un estado simple descendiente del estado TOP. Esto lo verificamos con la condición evaluada en la *línea 4*, que dice que si el padre es el TOP y la cantidad de estados activos es 1, entonces, llama a la función que calcula las transiciones desde un estado simple (*línea 5*).

El caso 2, hace referencia a la situación en que el *estado de configuración* evaluado representa a un estado compuesto no concurrente. En este contexto, identificado porque el EC (Estado de Configuración) contiene un solo estado pero el padre no es el TOP, se llama a la función que calcula las transiciones desde un estado compuesto no concurrente (*línea 8*).

El último caso (caso 3), se da cuando la cantidad de estados es mayor que 1, evidenciando que estamos en presencia de un estado compuesto concurrente.

obtenerTransiciones (EstadoConfiguracion EC¹, Evento E): Conj.Transiciones

```
Línea 1      TRXS = vacío
Línea 2      Si length(EC) == 1 Entonces
Línea 3          /* CASO 1 */
Línea 4          Si EC(0).getContainer() == TOP Entonces
Línea 5              TRXS = obtenerTransicionesEstadoSimple(EC, E)
Línea 6          Sino
Línea 7              /* CASO 2 */
Línea 8              TRXS = obtenerTransicionesEstadoCompuestoNoConcurrente(EC, E)
Línea 9          Fin Si
Línea 10     Sino
Línea 11         /* CASO 3 */
Línea 12         TRXS = obtenerTransicionesEstadoCompuestoConcurrente(EC, E)
Línea 13     Fin si
Línea 14     Retornar TRXS
```

¹ Cabe recordar que el estado de configuración está representado por las hojas del árbol de configuración, es decir, los estados más profundos activos de la *state machine*. El único caso donde la longitud es mayor a 1 es cuando hay regiones concurrentes.

Función: ObtenerTransicionesEstadoSimple

Esta función calcula la transición activada por el evento E en el único estado que hay en el estado de configuración.

obtenerTransicionesEstadoSimple (EstadoConfiguracion EC, Evento E): Conj.Transiciones

```
Línea 1      TRXS = vacío;  
Línea 2      TRXS.add(obtenerTransicion (EC(0), E));  
Línea 3      Retornar TRXS;
```

Función: ObtenerTransicionesEstadoCompuestoNoConcurrente

En este caso, si el estado que está activo en la hoja del árbol de estado de configuración responde al evento E, se devuelve la transición que se dispara (líneas 2 y 4). En el caso de que la hoja (estado más profundo en el estado de configuración) no responda a dicho evento, se debe empezar a recorrer el árbol hacia arriba buscando qué estado padre responde a dicho evento, hasta llegar al TOP.

obtenerTransicionesEstadoCompuestoNoConcurrente (EstadoConfiguracion EC, Evento E): Conj.Transiciones

```
Línea 1      TRXS = vacío;  
Línea 2      T = obtenerTransicion (EC(0), E)  
Línea 3      Si T != vacío Entonces  
Línea 4          TRXS.add(T)  
Línea 5      Sino  
Línea 6          S = EC(0).getContainer()  
Línea 7          T = obtenerTransicion (S, E)  
Línea 8          Mientras T == vacío and S.getContainer() != TOP Entonces  
Línea 9              S = S.getContainer()  
Línea 10             T = obtenerTransicion (S, E)  
Línea 11          Fin Mientras  
Línea 12          Si T != vacío Entonces  
Línea 13              TRXS.add(T)  
Línea 14          Fin Si  
Línea 15      Fin Si  
Línea 16      Retornar TRXS
```

Función: ObtenerTransicionesEstadoCompuestoConcurrente

Este caso es parecido al anterior, pero como existen estados concurrentes, cuando se busca los estados que se activan hay que hacerlo mirando a cada uno de los estados activos (son más de uno). Primero se verifica que el evento dispare alguna transición en las hojas (líneas 2 a 7) y si no, hay que buscar en los padres, subiendo nivel por nivel en el árbol de configuración. Si el árbol no está balanceado, la función obtenerNivelSuperior sólo sube por las ramas de máximo nivel.

De esta forma se asciende hasta encontrar algún padre en el que se dispare el evento E. Como se puede ver en las *líneas 11 y 14* la búsqueda es recursiva.

obtenerTransicionesEstadoCompuestoConcurrente (EstadoConfiguracion EC, Evento E): Conj.Transiciones

```
Línea 1      TRXS = vacío
Línea 2      Para cada S en EC Hacer
Línea 3          T = obtenerTransicion(S, E)
Línea 4          Si T != vacío Entonces
Línea 5              TRXS.add (T)
Línea 6          Fin si
Línea 7      Fin Para
Línea 8      Si TRXS == vacío Entonces
/* No encontré transiciones en las hojas, tengo que recorrer el árbol hacia arriba */
Línea 10     EFC = obtenerNivelSuperior(EC)
Línea 11     TRXS = obtenerTransiciones(EFC, E)
Línea 12     Mientras TRXS.length == 0 and EFC != TOP Hacer
Línea 13         EFC = obtenerNivelSuperior(EFC)
Línea 14         TRXS = obtenerTransiciones(EFC, E)
Línea 15     Fin Mientras
Línea 16     Fin Si
Línea 17     Retornar TRXS
```

Función: crearEstadoConfiguracionDestino

Dado un *estado de configuración* EC y un conjunto de transiciones ctx (parámetros de entrada), la función devuelve el *estado de configuración* destino luego de procesar las transiciones que están en ctx. Esto corresponde al *run-to-completion step*.

Para lograrlo, se recorre el árbol por niveles, comenzando por el más profundo, para ir buscando los estados que se activan por las transiciones que estamos analizando. Por cada nivel que se procesa, se separan los estados en 2 grupos, a los cuales se los identifica como conTrx y sinTrx. En el primero, se mantienen los estados que poseen alguna transición del conjunto de transiciones recibido por parámetro. En el segundo, los que no.

Dado que un estado compuesto que contiene varias regiones, puede tener un estado en una región que responda a una transición, pero los hermanos (es decir, estados activos en las regiones ortogonales) de éste no, en este caso se debe avanzar por el estado que se activa por la transición, en el *estado de configuración* destino, pero hay que mantener a sus hermanos en la misma posición. Dado que uno de los hijos del estado compuesto que contiene las regiones respondió a la transición, se corta la búsqueda por la rama del árbol que contiene al estado compuesto analizado.

En el caso de que ninguna región de un estado compuesto responda a una transición, se avanzará hacia el padre, para ver si algún estado en el árbol de configuración tiene una transición que responda al evento. Pero hay que tener en cuenta que si ningún padre responde al evento, en el *estado de configuración* hay que dejar a los hijos que analizamos en un primer momento. Esto es, dado que el evento puede activar una región concurrente ortogonal al padre de las hojas analizadas, lo que hace que las hojas tienen que mantenerse en el nuevo *estado de configuración*. Para poder

hacer esto, lo que se hace es mantener un conjunto llamado *pendings*, en donde vamos dejando a los estados que no respondieron a ninguna transición, que no son hermanos de algún estado que si respondió, y además que ningún padre de los mismos va a responder a alguna transición.

Veamos un ejemplo para clarificar la idea de esta función. Supongamos que tenemos la *state machine* de la Figura 15, y que el ASC (*estado de configuración* que se está analizando) es el que tiene como hojas del árbol a los estados State4, State5, State9 y State7, representado en la Figura 16. Ante el arribo del evento E8, se recorren las hojas más profundas del *árbol de configuración* sin encontrar una transición que se active. Al ir subiendo hacia los padres, encontramos la transacción Trx2 que responde a dicho evento y que forma parte de una región ortogonal a la región que contiene el estado compuesto al cual pertenecen el resto de de las hojas de ASC. En el *estado de configuración* destino entonces es necesario mantener a los estados State5, State9 y State7 (que se mantuvieron en la lista de *pendings*) y solo avanzar del estado State4 al estado EndR1.

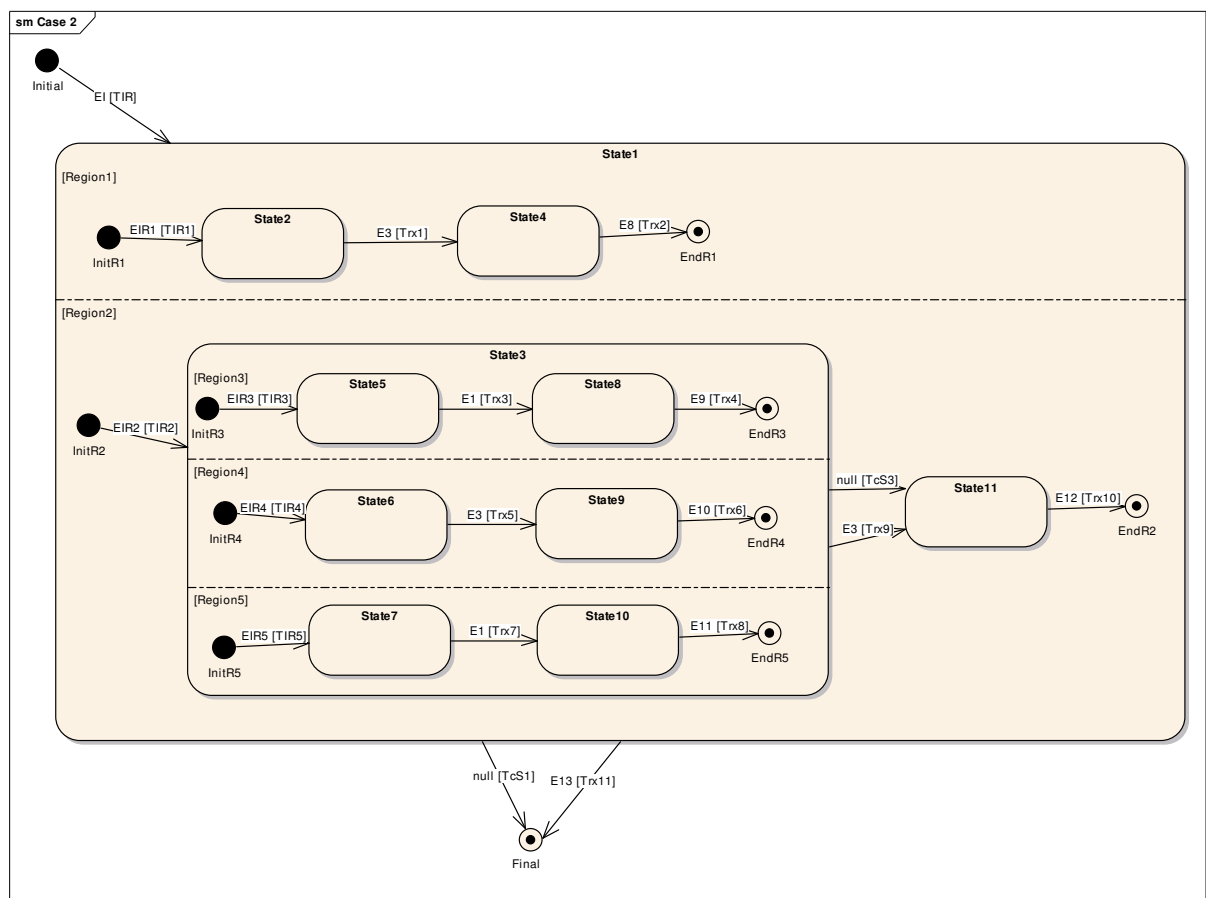


Figura 15: *State machine* para ejemplificar la creación del *estado de configuración* destino

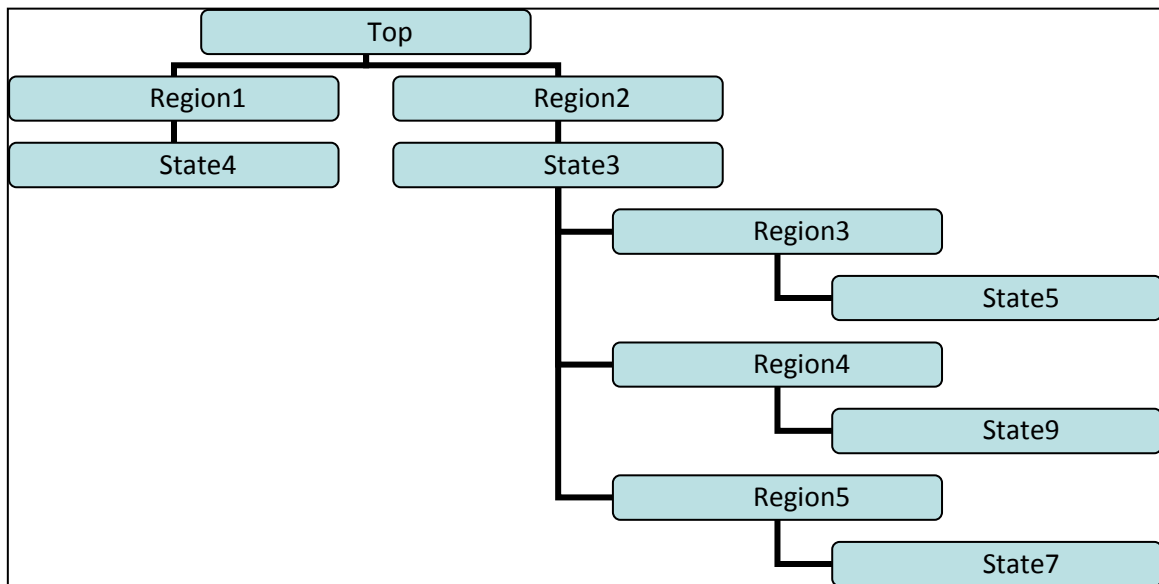


Figura 16: Un estado de configuración válido de la state machine de la Figura 15

A continuación se muestra el pseudocódigo y se explican cada una de las líneas.

crearEstadoConfiguracionDestino (EstadoConfiguracion EC, Conj.Transiciones ctx): EstadoConfiguracion

```

Línea 1      NewSC = vacío
Línea 2      nivel = EC.maximoNivel()
Línea 3      sc = EC      -- Me hago una copia del estado de configuración original
Línea 4      Mientras sc != vacío Hacer
Línea 5          sinTrx = vacío
Línea 6          conTrx = vacío
Línea 7          Para cada estado s en sc Hacer
                  -- voy recorriendo el árbol desde el nivel más profundo
Línea 8              Si s.nivel = nivel Entonces
Línea 9                  tx = buscarTransicion(ctx, s)
Línea 10                 Si tx != vacío Entonces
Línea 11                     Si ! tieneHijo(s, NewSC) Entonces
Línea 12                         conTrx.add(s)
Línea 13                         agregarEstadoDestino(NewSC, tx)
Línea 14                     Fin Si
Línea 15                 Sino
Línea 16                     sinTrx.add(s)
Línea 17                 Fin Si
Línea 18                 sc.del(s)
Línea 19             Fin Si
Línea 20     Fin Para
  
```

```

-- Corto el recorrido en la rama eliminando a los hermanos de
-- los estados que sí tienen a la transición
Línea 21 Si conTrx != vacío Entonces
        -- borro de pendings los hijos de los estados que se agregan a conTrx
Línea 22 Para cada estado k en conTrx Hacer
Línea 23     eliminarHijos(k, pendings)
Línea 24 Fin Para

        -- los hermanos son eliminados de sinTrx
Línea 25 h = eliminarHermanos(conTrx, sinTrx)
Línea 26 Para cada estado k en h Hacer
Línea 27     Si k es compuesto Entonces
        -- tengo que buscar los hijos de h en EC para agregarlos a NewSC
Línea 28         agregarEstadoCompuesto(NewSC, EC, k)
Línea 29     Sino
Línea 30         NewSC.add(k)
Línea 31     Fin Si
Línea 32 Fin Para
Línea 33 Fin Si

-- Agrego a los padres de los que no tienen la transición para
-- seguir subiendo por la rama del árbol
Línea 34 Si sinTrx != vacío Entonces
Línea 35     Para cada estado k en sinTrx Hacer
Línea 36         Si ! tieneHijo(k, pending) Entonces
Línea 37             pending.add(k)
Línea 38         Fin Si
Línea 39         Si k.getContainer() != TOP Entonces
Línea 40             sc.add(k.getContainer())
Línea 41         Fin Si
Línea 42     Fin Para
Línea 43 Fin Si
Línea 44     nivel = nivel -1
Línea 45 Fin Mientras

-- Si hay estados pendientes los agrego al NewSC
Línea 46 Si pending != vacío Entonces
Línea 47     Para cada estado k en pending Hacer
Línea 48         NewSC.add(k)
Línea 49     Fin Para
Línea 50 Fin Si

-- Ejecutar transiciones de completitud.
Línea 51 s = obtenerEstadoFinalOrtogonal (NewSC);
Línea 52 Si s != vacío Entonces

```

```

Línea 53      tx = obtenerTransicionComplejidad(s)
Línea 54      eliminarHijos(s, NewSC)
Línea 55      agregarEstadoDestino(NewSC, tx)
Línea 56      Fin Si

Línea 57      Retornar NewSC

```

Lo primero que se hace es crear el *estado de configuración destino*, llamado NewSC, obtener la máxima profundidad del árbol, y realizar una copia del estado de configuración actual que vamos a procesar, dado que vamos a ir eliminando del *estado de configuración* los estados que ya fueron procesados. Estas acciones están cubiertas por las *líneas 1, 2 y 3*.

El ciclo de la *línea 4* tiene como condición que el *estado de configuración* sc (la copia) no esté vacía. Luego se inicializan los dos conjuntos de estados conTrx y sinTrx en las *líneas 5 y 6*. Recordar que un *estado de configuración* es una lista de las hojas del árbol, por eso podemos ver a estos 2 conjuntos como listas.

Con las condiciones de las *líneas 7 y 8* lo que se hace es sólo recorrer las hojas del árbol del nivel que se está analizando. En la *línea 9* se busca si el estado s contiene en su conjunto de transiciones de salida alguna de las transiciones del conjunto que se activaron (ctx), y si existe alguna (*línea 10*), lo que se hace es controlar que el estado s no sea un padre de algún estado ya agregado en el *estado de configuración destino* (NewSC) (*línea 11*). El motivo de este control, es que se puede dar el caso en árboles de estados de configuración desbalanceados, donde ya algún estado de una región respondió a una transición, pero en la otra región tuvimos que subir por el padre dado que no hubo estado que respondió. En este caso, notar que las hojas de dicha región quedaron en la lista de pendientes, con lo cual, al final serán tomadas en cuenta.

Las *líneas 12 y 13* agregan el estado al conjunto conTrx y el destino de ejecutar la transición al *estado de configuración destino* NewSC.

En el caso de no haber encontrado transición, se agrega dicho estado al conjunto sinTrx (*línea 16*). En la *línea 18*, se elimina al estado analizado de la lista sc.

Entre las *líneas 21 y 33*, lo que se hace por cada nivel analizado es ver que pasó con el conjunto conTrx. Primero se eliminan los hijos de cada estado en conTrx que esté en pending (*líneas 22-24*), luego, se eliminan los hermanos de cada estado en conTrx que está en sinTrx (*líneas 25-33*). Esto último se hace, porque como se mencionó anteriormente, si un estado en una región responde a una transición, los hermanos del mismo estado compuesto deben quedar en el *estado de configuración destino*.

Entre las *líneas 34 y 43*, se analiza el conjunto sinTrx. En este caso, para cada uno de los estados en sinTrx, si no posee un hijo suyo en pending, hay que agregarlo a dicha lista (*línea 37*). En la *línea 40* se agrega el padre del estado en cuestión a sc, para que el mismo sea procesado en un próximo paso del ciclo (cuando se suba de nivel).

En la *línea 44* se decrementa la variable nivel para ir subiendo por el árbol de configuración.

Una vez fuera del ciclo principal, se analiza si quedó algo en la lista de pending, en cuyo caso hay que agregarlo al *estado de configuración destino* NewSC.

Y por último, entre las *líneas 51 y 56* se verifica si algún estado compuesto tiene a todas sus regiones en un estado final, con lo cual, debería ser disparada automáticamente la transición de completitud. En este caso, se reemplaza a los estados finales (hijos) por el destino de la ejecución de dicha transición.

7. Demostración: de state machines a autómatas finitos

En esta sección presentaremos la demostración de que el algoritmo explicado en el punto anterior transforma una *state machine* en un *autómata finito* y que este *autómata finito* tiene en sus caminos válidos las secuencias de ejecuciones válidas de la *state machine*.

Primero mostraremos nuevamente el pseudocódigo, en forma reducida (para detalles ir a la sección 6), para tener referencia a las líneas explicadas, luego algunas definiciones y por último la demostración en sí misma.

Algoritmo (State Machine SM): Autómata A

```
Línea 1     ASC = obtenerEstadoConfiguracionInicial()
Línea 2     NA = crearNodo(ASC)
Línea 3     marcarNodoPendiente(NA)
Línea 4     marcarNodoInicial(NA)
Línea 5     agregarNodo(NA)
Línea 6     Mientras ASC != vacío Hacer
Línea 7         Para cada Evento en CEV Hacer
Línea 8             EA = CEV.Proximo()
Línea 9             CTX = obtenerTransiciones(ASC, EA)
Línea 10            Si CTX no vacío Entonces
Línea 11                NSC = crearEstadoConfiguracionDestino(ASC, CTX)
Línea 12                NN = obtenerNodoEquivalente(NSC)
Línea 13                Si NN es nulo Entonces
Línea 14                    NN = crearNodo(ASC, CTX)
Línea 15                    marcarNodoPendiente(NN)
Línea 16                    Si estadoFinal(NNS) Entonces
Línea 17                        marcarNodoFinal(NN)
Línea 18                    Fin Si
Línea 19                    agregarNodo(NN)
Línea 20                Fin Si
Línea 21                NE = crearEje(NA, NN, EA)
Línea 22                agregarEje(NE)
Línea 23                Fin Si
Línea 24            Fin Para
Línea 25            marcarNodoProcesado(NA)
Línea 26            NA = obtenerNodoPendiente()
Línea 27            ASC = obtenerEstadoConfiguracion(NA)
Línea 28        Fin Mientras
```

En el contexto de este trabajo, y en base a lo explicado hasta el momento, se da la particularidad de que la semántica de una *state machine* se expresa como un *autómata finito*. Este *autómata finito* es el que acepta a todas las ejecuciones de la *state machine*. Lo que vamos a demostrar en esta

sección es que el algoritmo propuesto en la sección “Primera versión: de state machine a autómata finito” dada una *state machine* devuelve un *autómata finito* que acepta sus ejecuciones. Este *autómata finito* tiene como nodos a los distintos *estados de configuración* y como ejes a los eventos de la *state machine*.

Antes de avanzar con la demostración, es necesario presentar algunas definiciones y lemas que serán usados posteriormente.

Definición 1: State Machine

Una *state machine* (SM) es una tupla $SM = \langle \text{Estado}^*, \text{Transición}^* \rangle$, donde las Transiciones están etiquetadas por eventos.

Un *estado de configuración* (STC) es una lista de estados hojas del árbol de configuración de la SM en un momento dado.

Una ejecución de una *state machine* es una cadena de la forma:

$$STC_0 \rightarrow t_{e_0+} \rightarrow STC_1 \rightarrow t_{e_1+} \rightarrow \dots \rightarrow t_{e_n+} \rightarrow STC_n$$

donde STC_0, \dots, STC_n son *estados de configuración* válidos en la *state machine*.

Por válido se entiende que existe una secuencia de eventos que soporta la *state machine* y que activa las transiciones en el orden indicado. Una transición se activa cuando el evento que se está procesando es su etiqueta.

Lema (creación de estado de configuración destino)

La función *crearEstadoConfiguracionDestino* devuelve, dado un *estado de configuración* válido STC_i en la *state machine* SM y un conjunto de transiciones T que se activan ante un evento e, el *estado de configuración* STC_j resultante de procesar el evento e en la *state machine* SM. Dicho en otras palabras, la función

crearEstadoConfiguracionDestino(STC_i, SM, T, e) devuelve STC_j tal que en las corridas de SM existe una subsecuencia $STC_i \rightarrow T_e \rightarrow STC_j$,

o sea, que la función *crearEstadoConfiguracionDestino* respeta la semántica de la *state machine* SM, y T respeta las condiciones de selección de transiciones descritas en la sección 3, sub-sección “Algoritmo de selección de transiciones”.

De la semántica de la *state machine*, explicada en la sección 3, se detectan tres posibles casos de activación de transiciones ante la llegada de un evento. Estos casos, en relación a la implementación aquí presentada, se identifican por cómo está compuesto el estado de configuración, árbol que representa los estados activos en un momento dado en una *state machine*. Los casos son los siguientes:

1. Una sola hoja cuyo padre es el "top" (Estado Simple).

En este caso y dado que por definición una *state machine* es determinística, el procesamiento del evento se reduce a identificar la transición que se activa con el evento y crear el siguiente *estado de configuración* con el estado target de dicha transición.

2. Una sola hoja cuyo padre es un estado compuesto (Estado Compuesto No Concurrente). En este caso, dado que sólo hay una hoja, implica que el estado compuesto no es concurrente, con lo cual, el evento puede activar una transición del estado hoja o de algún estado en toda la rama desde el top. Esto produce dos posibles casos:
 - a. La transición que se activa tiene como "source" a la hoja, en cuyo caso se crea el *estado de configuración destino* con el estado "target" de la transición.
 - b. Desde la hoja no se activa ninguna transición, por lo que se realiza una búsqueda recursiva en el padre del estado hasta encontrar un estado en la rama del cual se active una transición. Al encontrar la transición, se crea el estado de configuración destino, con el "target" de la transición activada.
3. Más de una hoja en el árbol (Estado Compuesto Concurrente).

En este caso el padre de las hojas es un estado compuesto concurrente, con lo cual, es necesario recorrer las hojas empezando por los estados simples más anidados (profundos) e ir recorriendo hacia el estado "top". Para cada estado en un nivel determinado, todas las transiciones son evaluadas para determinar cuáles se activan, produciéndose dos posibles casos:

- a. De una o más hojas se activan transiciones, en cuyo caso se crea el *estado de configuración destino* con los target de dichas transiciones y se agregan las hojas que no activaron transiciones. La búsqueda en esta rama del árbol se debe terminar.
- b. Desde ninguna hoja se activan transiciones, lo que implica continuar la búsqueda recursiva en el árbol por nivel hasta identificar una o más transiciones que se activan. Al llegar a este punto, el estado de configuración se crea con los estados destinos de dichas transiciones, sumando los estados del nivel que no dispararon transiciones.

En los casos 2 y 3 por tratarse de estados compuestos, una vez creado el *estado de configuración destino*, hay que identificar si el mismo representa la finalización del compuesto. De ser positiva esta validación hay que procesar la transición de completitud del estado compuesto (obligatoria por definición de la *state machine*) creando el *estado de configuración* con el target de esta transición de completitud y eliminando lo anterior.

Así mismo, y en los 3 casos, al ser el target de la/s transición/es que se activan un estado compuesto, se debe "inicializar" el mismo. Esto significa que el *estado de configuración* se forma con los target de todas las transiciones iniciales de las regiones del compuesto. Si no es concurrente será sólo 1.

Lema (obtener estado de configuración inicial)

La función *obtenerEstadoConfiguracionInicial* devuelve el *estado de configuración* STC_0 resultante de procesar el evento inicial del estado TOP en la *state machine* SM. Por definición de la *state machine*, todos los estados compuestos tienen un pseudo-estado (llamado inicial) el cual permite inicializarlo.

Esta función toma este caso particular dado que sólo existe una sola transición de salida del estado inicial. Con lo cual, no hay que estar buscando que transiciones se disparan ante un evento determinado.

Demostración del algoritmo

Como se mencionó anteriormente la semántica de una *state machine SM* se expresa como un *autómata finito AF*. Este *autómata finito AF* es el que acepta como lenguaje a todas las ejecuciones de la *state machine SM*. Lo que se va a demostrar a continuación es que el algoritmo presentado en la sección 6, dada una *state machine SM* devuelve un *autómata finito AF* que acepta su ejecuciones. Es decir,

$$\text{traducir}(SM) \rightarrow AF$$

Este *autómata finito* tiene como nodos a los *estados de configuración* STC_i y como ejes a los eventos de la *state machine*.

Lo que se va a demostrar es que dada una *state machine SM*, el lenguaje aceptado por el *autómata finito AF*, generado por el algoritmo *traducir*, es el conjunto de todas las posibles ejecuciones de *SM*. Ante esto se demostrarán los siguientes enunciados

1. Todo camino posible del *autómata finito AF* es una ejecución válida en la *state machine SM*
2. Toda ejecución válida de la *state machine SM* es aceptada por el *autómata finito AF*

Enunciado 1: se demostrará por inducción en la longitud de los prefijos de los caminos del *autómata finito AF*. Llamaremos camino válido del *autómata finito* a un camino que o bien lleva a un estado de aceptación, o bien es un prefijo de un camino que lleva a un estado de aceptación.

Caso base: prefijo de un camino válido que tiene sólo el nodo inicial de *AF*, llamado STC_0

STC_0 (*Estado de Configuración Inicial*) es posible en la *SM* ya que por definición una *SM* tiene un estado de inicio, y una sola transición de inicio y por lema *obtener estado de configuración inicial*, STC_0 es válido. En las líneas 1 y 2 se crea el nodo que representa este estado de configuración, agregándose el mismo al *autómata finito* en la línea 5.

Paso inductivo (por absurdo): supongamos que *AF* tiene un prefijo de un camino de longitud n que es válido y termina en el nodo STC_n , y una transición t_e que se activa por el evento e que lo lleva a STC_{n+1} y STC_{n+1} no es un estado de configuración válido en *SM*. Es decir, la secuencia

$STC_0 \rightarrow t_{e0} \rightarrow \dots \rightarrow t_{en} \rightarrow STC_n \rightarrow t_e \rightarrow STC_{n+1}$ es un prefijo de un camino posible en *AF*, pero no es un prefijo de una ejecución de *SM*, mientras que $STC_0 \rightarrow t_{e0} \rightarrow \dots \rightarrow t_{en} \rightarrow STC_n$ sí lo es por hipótesis inductiva.

Existen dos motivos por los cuales se puede dar esta situación:

- a) e no activa ninguna transición de salida desde STC_n
- b) $STC_n + t_e \neq STC_{n+1}$

Caso a)

Por la línea 19 sólo se agregan nodos al *autómata* si se cumple la condición de la línea 10, que por las líneas 9 y 10 del algoritmo sólo se crean *estados de configuración* de un STC_n a otro STC_{n+1} con los eventos válidos y que activan una transición

Caso b)

Por la línea 21, el eje entre STC_n y STC_{n+1} ($STC_n \rightarrow STC_{n+1}$) está sólo si STC_{n+1} salió de la línea 11, es decir que fue creado con la función explicada en el lema anterior.

Además en la línea 11, STC_{n+1} se crea a partir de $STC_n + t_e$. Es decir, la función *crearEstadoConfiguracionDestino*, recibe el *estado de configuración* actual y el conjunto de transacciones que se activa con el evento e y a partir de estos conjuntos, crea el nuevo *estado de configuración* resultado de la ejecución de las transacciones activas. Por lo tanto, si no existe el prefijo en la SM, tampoco estará en el AF.

Enunciado 2: se demostrará por inducción en la longitud de los prefijos de las ejecuciones de la *state machine* SM

Caso base: el prefijo más corto de la *state machine* SM es el *estado de configuración* inicial STC_0 , que es un prefijo de un camino del *autómata finito* AF ya que este nodo se crea de acuerdo a lo explicado en el lema *obtener estado de configuración inicial* y se agrega a AF en las líneas 1, 2 y 5 del algoritmo.

Paso inductivo (por absurdo): supongamos que tenemos un prefijo de una ejecución válida de la *state machine* SM que termina en STC_n y que es un prefijo de un camino aceptado por el *autómata finito* AF y que al extender la ejecución con la transición t_e que se activa con el evento e , sigue siendo válida en SM, pero no es aceptada por AF, es decir,

$STC_0 \rightarrow t_{e0} \rightarrow \dots \rightarrow t_{en} \rightarrow STC_n \rightarrow t_e \rightarrow STC_{n+1}$ es válida en SM pero no lo es en AF, mientras que $STC_0 \rightarrow t_{e0} \rightarrow \dots \rightarrow t_{en} \rightarrow STC_n$ sí lo es por hipótesis inductiva

Por las líneas 7 y 9 el evento e va a ser analizado, y al procesarse por la línea 11, se va a crear con la función de la línea 11 STC_{n+1} (función explicada en el lema anterior). Por la línea 12 (en caso que el nodo equivalente a STC_{n+1}) o por las líneas 14 y 19 el *autómata finito* incorpora como nodo a STC_{n+1} . Además por la línea 21, quedan STC_n y STC_{n+1} conectados en AF, por lo tanto $STC_n \rightarrow t_e \rightarrow STC_{n+1}$ es un prefijo de un camino válido en AF.

De acuerdo con el algoritmo, un nodo es de aceptación en el *autómata finito* AF, si y solo si representa al estado final de la *state machine* (por las líneas 16 y 17). Ambos puntos prueban que todas las extensiones válidas en el *autómata finito* AF son válidas en la *state machine* SM y viceversa. En particular, las extensiones que agregan el *estado de configuración final* y el nodo de aceptación. Por ende, los prefijos que se extienden eventualmente se transforman en caminos/corridas completas.

8. Segunda versión: De state machines a autómatas temporizados

Esta versión del algoritmo toma como entrada una *state machine* (SM) y retorna un *autómata temporizado* (TA) que modela todas las posibles ejecuciones de la *state machine*. En esta versión se soportan los eventos de tiempo (TimeEvents) en la *state machine*, es por ello que es necesario incorporar el uso de relojes para contabilizar el transcurso del tiempo al momento de ingresar a un estado que tiene transiciones de salida que se activan por eventos de tiempo.

Visto que los estados pueden ser compuestos, puede ocurrir que un estado padre posea un reloj y alguno de sus estados hijos posea otro, computando cada uno, tiempos independientes. En la figura 17 se muestra una porción de una *state machine* donde se ejemplifica esta situación.

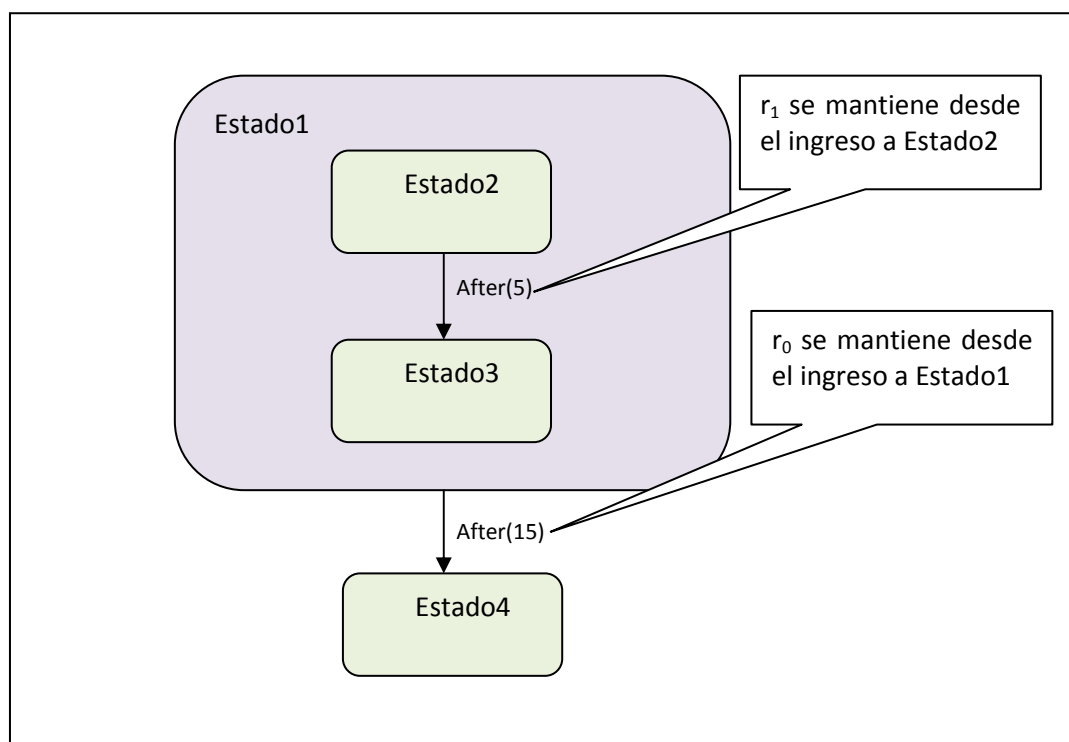


Figura 17: Dos relojes activos con diferencias en las inicializaciones y mediciones

Para poder agregar esta nueva funcionalidad, si bien el esquema del algoritmo es básicamente igual que la versión anterior, ahora en cada nodo del *autómata temporizado*, además del *estado de configuración* que lo originó, se guarda el conjunto de relojes que se encuentran activos. Para minimizar la cantidad de relojes que se utilizan en el *autómata temporizado*, se usa un pool de relojes. Con lo cual, si un reloj fue utilizado, pero actualmente no está en uso, se puede reutilizar.

Cuando se crea el nodo inicial, se verifica si existen relojes que se activan con la inicialización de la *state machine* (esta acción se lleva a cabo en la *línea 2*). La variable R, representada como un vector, contiene el pool de relojes, donde cada índice representa una variable del reloj, y el

contenido de la posición que se corresponde con el estado que está utilizando ese reloj. Ej, si $r(0) = s1$ y $r(1) = s2$ indica que el estado $s1$ posee el reloj $r0$, y el estado $s2$ posee el reloj $r1$. No está de más aclarar que cada estado puede poseer un solo reloj, dado que cada estado computa el tiempo desde que se ingresa en él hasta que se sale de él debido a alguna transición de TimeEvent.

Puesto que el ciclo principal va tomando cada uno de los *estados de configuración* no procesados, y que los mismos guardan el estado de sus relojes, en la *línea 10* se obtiene el conjunto de relojes correspondientes al nodo analizado.

Al avanzar de un estado de configuración a otro, se pueden presentar dos situaciones con respecto a los relojes:

1. *Eliminación de relojes*: si en el *estado de configuración destino* un estado que poseía un reloj no se encuentra más activo, tampoco seguirán vigentes sus relojes. Esto se resuelve en la *línea 15* llamando a la función *eliminarRelojes*, y en *Rnew* está el pool de relojes activos en el nuevo *estado de configuración*.
2. *Agregado de relojes*: si aparece un nuevo estado que posee una transición de salida por evento de tiempo en el *estado de configuración destino*. Esto se resuelve en la *línea 16* llamando a la función *calcularRelojes*. Cabe aclarar que esta función además de agregar en *Rnew* los nuevos relojes, los devuelve para que sean asignados a la variable *NR* (Nuevo Reloj), ya que es necesario saber cuáles son los nuevos relojes en el momento de inicializarlos en el *autómata temporizado* cuando se agreguen los ejes.

Ahora veamos cómo se utiliza el pool de relojes para generar las expresiones temporales en los ejes del *autómata temporizado*.

En la *línea 19*, cuando se crea al nuevo nodo, se calcula la expresión *Reset*, que es la que inicializa los nuevos relojes (*NR*) y ésta se utiliza en el eje que va del *Nodo Actual (NA)* al *Nuevo Nodo (NN)*.

En la *línea 26* y posteriores, se pregunta si el evento analizado es un evento de tiempo, en el caso de sea así, se calcula la expresión de *restricción temporal*, que va a ir en el eje que une el *Nodo Actual (NA)* con el nuevo nodo (*NN*). La función *calcularExpTemporal* devuelve para cada estado origen del conjunto de transiciones que recibe cuál es el reloj que utiliza y el tiempo asignado en ese evento, generando expresiones del tipo $r1>5$, $r2>10$.

A continuación se encuentra el pseudocódigo descrito anteriormente:

```
-- ASC: Active State Configuration
-- R: Pool de Relojes
-- NA: Nodo Activo
-- CEV: Conjunto Eventos
-- EA: Evento Actual
-- CTX: Conjunto de Transiciones
-- NSC: New State Configuration
-- NN: Nuevo Nodo
-- Rnew: Relojes del nuevo estado de configuración
-- NR: Nuevos Relojes
-- ER: Expresión Reset
```

```

-- ET: Expresión Temporal
-- NE: Nuevo Eje
Línea 1     ASC = crearEstadoConfiguracionInicial
Línea 2     R = calcularRelojes(ASC)
Línea 3     NA = crearNodo(ASC, R)
Línea 4     marcarNodoPendiente(NA)
Línea 5     marcarNodoInicial(NA)
Línea 6     agregarNodo(NA)
Línea 7     Mientras ASC != vacío Hacer
Línea 8         Para cada Evento en CEV Hacer
Línea 9             EA = CEV.Proximo()
Línea 10            R = obtenerRelojes(NA)
Línea 11            CTX = obtenerTransiciones(ASC, EA)
Línea 12            Si CTX != vacío Entonces
Línea 13                NSC = crearEstadoConfiguracionDestino(ASC, CTX)
Línea 14                NN = obtenerNodoEquivalente(NSC)
Línea 15                Rnew = eliminarRelojes(NSC, R)
Línea 16                NR = calcularRelojes(NSC, Rnew)
Línea 17                Si NN == vacío Entonces
Línea 18                    NN = crearNodo(NSC, Rnew)
Línea 19                    ER = calcularExpReset(NR)
Línea 20                    marcarNodoPendiente(NN)
Línea 21                    Si estadoFinal(NCS) Entonces
Línea 22                        marcarNodoFinal(NN)
Línea 23                    Fin Si
Línea 24                    agregarNodo(NN)
Línea 25                Fin Si
Línea 26                Si esTimeEvent(EA) Entonces
Línea 27                    ET = calcularExpTemporal(CTX, R)
Línea 28                    NE = crearEje(NA, NN, EA.nombre, ER, ET)
Línea 29                Sino
Línea 30                    NE = crearEje(NA, NN, EA.nombre, ER)
Línea 31                Fin Si
Línea 32                agregarEje(NE)
Línea 33            Fin Si
Línea 34        Fin Para
Línea 35        marcarNodoProcesado(NA)
Línea 36        NA = obtenerNodoPendiente()
Línea 37        ASC = obtenerEstadoConfiguracion(NA)
Línea 38    Fin Mientras

```

9. *Justificación de la transformación de state machines a autómatas temporizados*

A diferencia de la primera versión del algoritmo presentada en la sección 6, en este caso daremos una justificación más informal de que el algoritmo presentado anteriormente es razonable en la traducción a *autómatas temporizados*. Primero se exponen algunas de las definiciones que presenta UML sobre los eventos de tiempo y la semántica de los mismos, donde se muestra una definición semi-formal de los mismos.

En la especificación de UML 1.4.2 ² dice:

“Un evento de tiempo (TimedEvent) modela la expiración de un tiempo límite (deadline) específico. Notar que el tiempo de ocurrencia de un evento de tiempo es el mismo que el tiempo de su recepción. Sin embargo, es importante notar que puede haber un demora variable entre el tiempo de recepción y el tiempo de despacho (por ejemplo, a causa de demoras en las colas)”

Y cuando se hace referencia a la semántica de los eventos menciona:

“Un evento de tiempo es el paso de un período de tiempo determinado desde el procesamiento de un evento (a menudo la entrada al estado actual) o la ocurrencia de una fecha / hora determinada”

Y en la especificación de UML 2.1.2 ¹¹ aclara:

“Un evento de tiempo especifica un punto en el tiempo representado por una expresión. La expresión puede ser absoluta o relativa a algún otro punto en el tiempo. Los eventos de tiempo relativos deben ser siempre usados dentro de un contexto, y el punto de partida es el inicio del estado en donde se origina la transición que es disparada por dicho evento”

Por eso, en el contexto de este trabajo se considera que las expresiones de los eventos de tiempo son relativas, y cuando se ingresa a un estado que posee una transición de salida con un evento de tiempo, hay que empezar a contar el paso del tiempo desde el ingreso al mismo, a los efectos de poder iniciar el citado evento cuando se cumple el tiempo especificado en el evento de tiempo.

Recordemos también que el *estado de configuración* tiene una representación arbórea, por lo que varios estados (anidados o concurrentes) pueden estar activos simultáneamente, lo que implica que el paso del tiempo no es necesariamente el mismo para cada uno de estos estados, porque los mismos pudieron activarse en distintos momentos.

Con lo cual, en la *state machine*, cada estado tiene que controlar el paso de su tiempo, pero no alcanza con que cada nodo del *autómata temporizado* generado controle el paso de su tiempo, dado que un estado activo puede estarlo en varios *estados de configuración*, lo que implica que el mismo estado va a estar representado en varios nodos del *autómata temporizado*. Por tal motivo, usamos el concepto de relojes incorporados en los *autómatas temporizados*, utilizando uno por cada estado, para poder controlar el paso del tiempo de cada uno de ellos.

Este reloj debe ser asignado cuando el estado se activa (es decir, no estaba activo en el *estado de configuración* anterior) y cada vez que el estado deja de estar activo (es decir, no se encuentra activo en el *estado de configuración* destino) es necesario agregar la condición de terminación por transcurso de tiempo. Cabe recordar que el evento de tiempo es un evento más, por lo que el algoritmo va a considerarlo en la selección de transiciones a procesar.

Veamos un ejemplo:

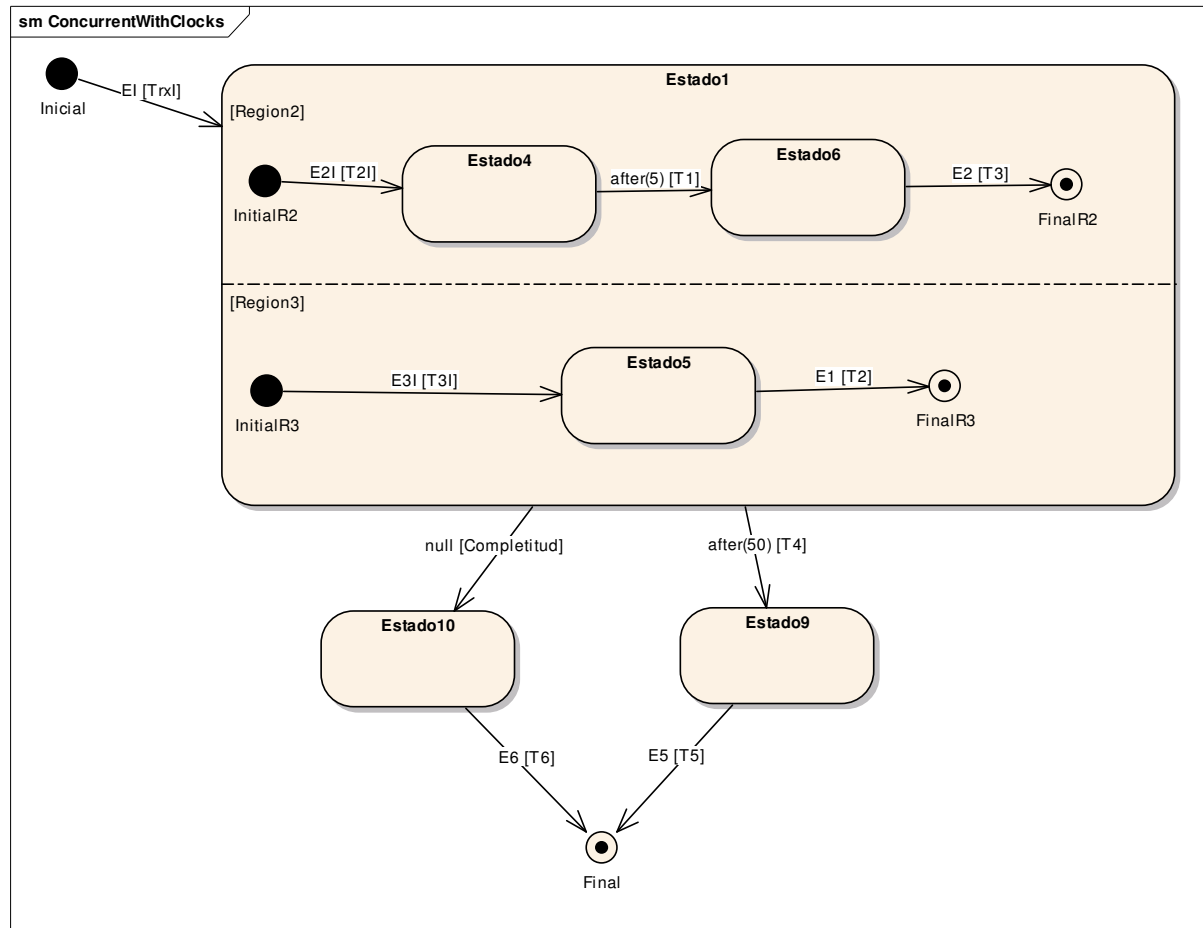


Figura 18: State machine para ejemplificar el uso de relojes

Haciendo un seguimiento de una situación posible:

- Se procesa el evento E1 (evento inicial).

Se activa el Estado 1 y sus hijos Estado4 y Estado5. Dado que Estado1 y Estado4 tienen transiciones con *after* (evento de tiempo como se explicó en la sección 3), se debe asignar un reloj para cada uno de ellos. Llamemos r1 al reloj de Estado1 y r2 al reloj de Estado4.

- Se procesa el evento E1.

El *estado de configuración* siguiente es Estado1 – Estado4 – FinalR3, y los relojes r1 y r2 tienen que seguir activos.

- Se cumple el tiempo de la transición T1 (*after(5)*).

Se pasa al *estado de configuración* Estado1 – Estado6 – FinalR3, pero el reloj r2 ya no es necesario, dado que Estado4 dejó de estar activo.

En todos los cambios de *estados de configuración* descritos anteriormente, se debe contemplar la transición T4 (salida del Estado1 cuando el tiempo es 50), y agregar en el *autómata temporizado* el eje con la condición " $r1 \geq 50$ ", para representar la salida del Estado1 cuando se cumplió el tiempo especificado. Se utiliza el operador \geq para no dejar lugar a dudas que es a partir del momento en que se cumple el tiempo especificado, debido a que en la especificación de *UML* se aclara que puede haber demoras debido a la implementación de la ejecución de las *state machines*.

En base a esto, se realizaron las modificaciones necesarias al algoritmo para soportar el paso del tiempo.

Las premisas que se tienen en cuenta son:

1. En el algoritmo inicial dado una *state machine* sin eventos de tiempo se genera un *autómata finito*, el cual representa la ejecución de la misma.
2. Cuando una *state machine* posee eventos de tiempo, hay que considerar de una manera especial a los estados que poseen transiciones de salida por dichos eventos.
3. Los estados identificados en el punto 2 tienen que llevar un registro del avance del tiempo para poder ejecutar la transición de salida cuando se cumple el tiempo especificado en el evento de tiempo. A esto se lo puede pensar como la activación de un reloj interno haciendo una inicialización (*reset*) cuando se ingresa en uno de estos estados, y el disparo del evento de salida relacionado al paso del tiempo, se lo indica con la condición en la transición correspondiente.
4. Cada nodo del *autómata temporizado* representa un árbol de configuración de estados de la ejecución de la *state machine*, por lo cual, es posible que varios estados estén activos simultáneamente, con evoluciones del tiempo independientes, lo que implica, que deben existir varios relojes para representar el paso del tiempo por cada uno de los estados que estén activos.
5. El algoritmo presentado en este trabajo no necesita representar el paso del tiempo de la ejecución de la *state machine*, sino que debe agregar relojes al *autómata finito* creado en la primera versión del algoritmo (sección 6), convirtiéndolo en un *autómata temporizado*, de tal manera que la ejecución del *autómata temporizado* sea equivalente a la ejecución de la *state machine* (con eventos de tiempo).

Por lo tanto, lo que se hace es aumentar la información que poseemos en cada nodo del *autómata finito*, con la información de cuáles estados están contando el paso del tiempo, por medio de una estructura que llamamos "pool de relojes", a fin de reutilizar los relojes que ya no están en uso. De esta manera, se busca minimizar la cantidad de relojes para mejorar la performance del *model checker* a utilizar para validar la implementación aquí presentada, dado que la complejidad del mismo es exponencial en la cantidad de relojes.

Además, cada vez que se crea un eje entre dos nodos, basándose en los relojes activos en los nodos origen y destino, es necesario analizar qué restricciones de relojes aplican en los ejes. Estas restricciones pueden ser de inicialización (*reset*) de un reloj que empieza a ser utilizado por primera

vez en el nodo destino; o una restricción temporal, esto implica el agregado de la condición que permite el avance de un nodo al otro, sólo cuando transcurre un tiempo determinado. Cabe aclarar que este tiempo que se determina es igual al tiempo que figura en el evento de tiempo modelado en la *state machine*.

Por consiguiente, llamemos:

- ECS (Estado de Configuración Source)
- ECT (Estado de Configuración Target)

Que están siendo representados en el *autómata temporizado* por los nodos:

- NS (Nodo Source)
- NT (Nodo Target)

Los casos que se pueden presentar son:

- A. ECT posee un estado E con una transición de salida con un evento de tiempo que no está en ECS, lo que implica que hay que crear un reloj para que cuente el tiempo que transcurre mientras está activo el estado E, e inicializar el reloj en el eje que va entre NS y NT. Este nuevo reloj, estará representado en el “pool de relojes” que está en NT.
- B. ECS posee un estado E con una transición de salida con un evento de tiempo, el cual está representado por un reloj R en el pool de relojes que está en NS, pero dicho estado no se encuentra más activo en ECT, lo que implica que hay que eliminar dicho reloj del pool de relojes de NT, dado que el estado no está más activo. Además, hay que agregar la expresión condicional que indique la salida del estado E por medio del paso del tiempo, en el eje que va entre NS y NT, si la transición que generó el cambio de estado de configuración se activó con el evento de tiempo.

No puede suceder que:

- C. Sea necesario agregar un reloj para un estado E que ya estaba en ECS (estado de configuración source), porque el paso del tiempo se cuenta a partir del ingreso al mismo. Esto se debe a que el algoritmo toma a cada *estado de configuración* una sola vez (se los marca una vez que fueron evaluados). No importa el orden en el cual se procesan, sólo que se lo haga una sola vez.

Las modificaciones realizadas al algoritmo son:

- Se modificó el método *crearNodo* para que reciba ahora dos parámetros, que son el estado de configuración y el pool de relojes. Ver su uso en las líneas 3 y 18.
- Ya se contaba con el método *obtenerEstadoConfiguracion* para obtener el *estado de configuración* de un nodo. Ahora se agregó el método *obtenerRelojes* para obtener el pool de relojes de un nodo. Ver su uso en la línea 10, dentro del ciclo en que se van procesando cada uno de los nodos del *autómata temporizado*.
- Se agrego el método *calcularRelojes*, el cual posee 2 versiones:

- i. La primera, calcula los relojes necesarios para un *estado de configuración* dado, y se lo utiliza en la línea 2, para calcular los relojes del *estado de configuración* inicial.
 - ii. La segunda se utiliza en la línea 16, cuando se crea el nodo destino. Aquí además se pasa como parámetro el pool de relojes del *estado de configuración origen*, para que sólo se agreguen los nuevos relojes correspondientes a estados que no estaban en el estado de configuración origen (CASO A).
- Se agregó el método *eliminarRelojes*, usado en la línea 15, cuya función es eliminar del pool de relojes que recibe los correspondientes a estados que no están en el *estado de configuración* destino (CASO B).

Para agregar las restricciones temporales y/o inicializaciones en los ejes, se crearon los siguientes métodos:

1. *calcularExpReset* (línea 19): Arma la expresión para inicializar en cero los relojes que recibe por parámetro. Notar que en el algoritmo, se usa el pool retornado por el método *calcularRelojes*, el cual retorna sólo los relojes que se agregaron en el nuevo *estado de configuración*.
2. *calcularExpTemporal* (línea 27): Arma la expresión temporal para salir del nodo cuando transcurre el tiempo indicado por el evento que recibió como parámetro.

A continuación se presenta el pseudocódigo, con las modificaciones detalladas anteriormente:

```

-- ASC: Active State Configuration
-- R: Pool de Relojes
-- NA: Nodo Activo
-- CEV: Conjunto Eventos
-- EA: Evento Actual
-- CTX: Conjunto de Transiciones
-- NSC: New State Configuration
-- NN: Nuevo Nodo
-- Rnew: Relojes del nuevo estado de configuración
-- NR: Nuevos Relojes
-- ER: Expresión Reset
-- ET: Expresión Temporal
-- NE: Nuevo Eje
Línea 1     ASC = crearEstadoConfiguracionInicial()
Línea 2     R = calcularRelojes(ASC)
Línea 3     NA = crearNodo(ASC, R)
Línea 4     marcarNodoPendiente(NA)
Línea 5     marcarNodoInicial(NA)
Línea 6     agregarNodo(NA)
Línea 7     Mientras ASC!= vacío Hacer
Línea 8         Para cada Evento en CEV Hacer

```

```

Línea 9      EA = CEV.Proximo()
Línea 10     R = obtenerRelojes(NA)
Línea 11     CTX = obtenerTransiciones(ASC, EA)
Línea 12     Si CTX != vacío Entonces
Línea 13         NSC = crearEstadoConfiguracionDestino(ASC, CTX)
Línea 14         NN = obtenerNodoEquivalente(NSC)
Línea 15         Rnew = eliminarRelojes(NSC, R)
Línea 16         NR = calcularRelojes(NSC, Rnew)
Línea 17         Si NN == vacío Entonces
Línea 18             NN = crearNodo(NSC, Rnew)
Línea 19             ER = calcularExpReset(NR)
Línea 20             marcarNodoPendiente(NN)
Línea 21             Si estadoFinal(NCS) Entonces
Línea 22                 marcarNodoFinal(NN)
Línea 23             Fin Si
Línea 24             agregarNodo(NN)
Línea 25         Fin Si
Línea 26     Si esTimeEvent(EA) Entonces
Línea 27         ET = calcularExpTemporal(CTX, R)
Línea 28         NE = crearEje(NA, NN, EA.nombre, ER, ET)
Línea 29     Sino
Línea 30         NE = crearEje(NA, NN, EA.nombre, ER)
Línea 31     Fin Si
Línea 32     agregarEje(NE)
Línea 33     Fin Si
Línea 34     Fin Para
Línea 35     marcarNodoProcesado(NA)
Línea 36     NA = obtenerNodoPendiente()
Línea 37     ASC = obtenerEstadoConfiguracion(NA)
Línea 38     Fin Mientras

```

10. Ejemplo

A lo largo de las secciones anteriores describimos los algoritmos desarrollados en el contexto de este trabajo, así como las justificaciones sobre lo que realizan. Para completar la validación del mismo, se modeló un sistema de monitoreo de pacientes con una *state machine* y se lo transformó a *autómata temporizado* con el algoritmo. Sobre el *autómata temporizado* resultante se probaron dos propiedades, una que fuera falsa y otra que se cumpliera y se validaron con el *model checker* Zeus.

El sistema de monitores del ejemplo, mide en forma paralela presión y pulso. Una vez encendido e inflado el soporte sobre el brazo, se pueden producir dos situaciones:

1. Pudo medir ambos indicadores, desinfla e informa valores
2. Si en 20 segundos no obtuvo las mediciones, desinfla e informa el error

En la figura 19 se muestra la *state machine* que lo representa.

El *autómata temporizado* resultante se representa en un archivo *ascii*, que es el que toma el *model checker* Zeus como entrada junto con el *autómata observador* que representa la propiedad a probar. En nuestro caso las propiedades probadas son las siguientes:

1. ¿Siempre se puede alcanzar el estado final en 25 segundos o más? Esta propiedad es verdadera
2. ¿Siempre se puede alcanzar el estado final en menos de 25 segundos? Esta propiedad es falsa

El observador generado para la propiedad 1 se detalla a continuación:

```
/* Medidor de Presion */
#states 4

#trans 18

#clocks 1 x

state: 0 /* Inflando */
invar: ( true )
trans:
true => off ; reset { } ; goto 1
true => tope ; reset { } ; goto 0
true => Infla ; reset { } ; goto 0
true => okPresion ; reset { } ; goto 0
true => okPulso ; reset { } ; goto 0
true => Desinf ; reset { } ; goto 0
true => Error ; reset { } ; goto 0
true => Mostrar ; reset { } ; goto 0
```

```
state: 1
invar: ( true )
trans:
x>=25 => fin; reset { } ; goto 2
x<25 => fin; reset { } ; goto 3

state: 2
prop: ERROR
invar: ( true )
trans:

state: 3
prop: OK
invar: ( true )
trans:
true => off ; reset { } ; goto 3
true => tope ; reset { } ; goto 3
true => Infla ; reset { } ; goto 3
true => okPresion ; reset { } ; goto 3
true => okPulso ; reset { } ; goto 3
true => Desinf ; reset { } ; goto 3
true => Error ; reset { } ; goto 3
true => Mostrar ; reset { } ; goto 3
```

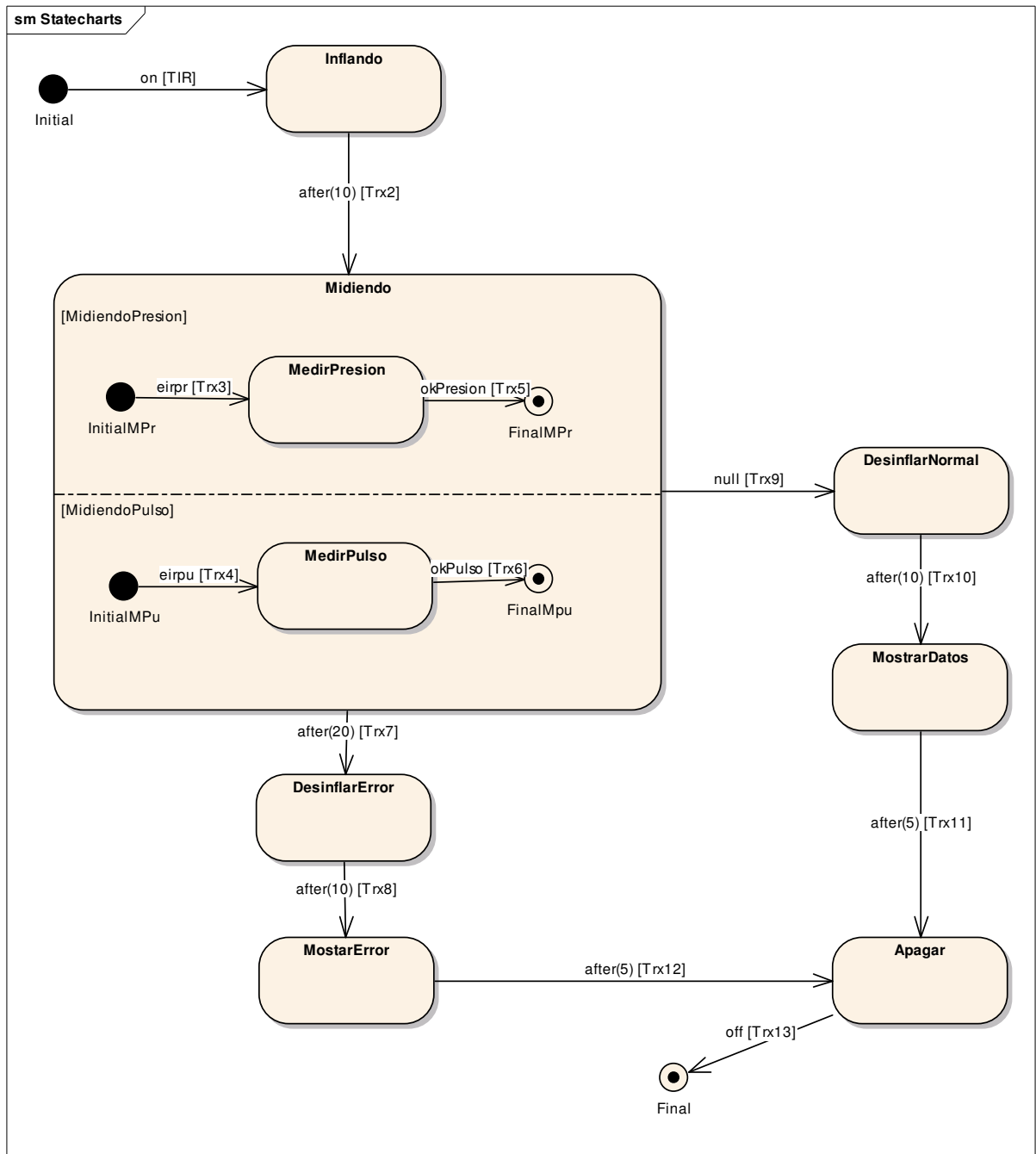


Figura 19: State machine modelando un sistema de monitoreo de pacientes

El *autómata temporizado* generado contiene 10 estados y 13 transiciones. En la figura 21 se puede ver su representación gráfica, mientras que a continuación se encuentra la estructura del mismo en el formato esperado por el *model checker*. Cabe aclarar que los comentarios al lado de cada nodo son una referencia para poder hacer la analogía con los estados activos que cada uno representa de la *state machine*.

```

/* Monitor de Pacientes */
#states 10
#trans 13
#clocks 1 r0

state: 0 /* Inflando */
invar: ( true )
trans:
r0>10 => O:Infla ; reset { r0 } ; goto 1

state: 1 /* Midiendo.MidiendoPresion.MedirPresion + Midiendo.MidiendoPulso.MedirPulso */
invar: ( true )
trans:
r0>20 => O:Error ; reset { r0 } ; goto 2
true => O:okPresion ; reset { } ; goto 3
true => O:okPulso ; reset { } ; goto 4

state: 2 /* DesinflarError */
invar: ( true )
trans:
r0>10 => O:Desinf ; reset { r0 } ; goto 5

state: 3 /* Midiendo.MidiendoPresion.FinPr + Midiendo.MidiendoPulso.MedirPulso */
invar: ( true )
trans:
r0>20 => O:Error ; reset { r0 } ; goto 2
true => O:okPulso ; reset { r0 } ; goto 6

state: 4 /* Midiendo.MidiendoPulso.FinPu + Midiendo.MidiendoPresion.MedirPresion */
invar: ( true )
trans:
r0>20 => O:Error ; reset { r0 } ; goto 2
true => O:okPresion ; reset { r0 } ; goto 6

state: 5 /* MostrarError */
invar: ( true )
trans:
r0>5 => O:Mostrar ; reset { } ; goto 7

state: 6 /* DesinflarNormal */
invar: ( true )
trans:
r0>10 => O:Desinf ; reset { r0 } ; goto 8

```

```
state: 7 /* Apagar */  
invar: ( true )  
trans:  
true => 0:off ; reset { } ; goto 9
```

```
state: 8 /* MostrarDatos */  
invar: ( true )  
trans:  
r0>5 => 0:Mostrar ; reset { } ; goto 7
```

```
state: 9 /* Fin */  
invar: ( true )  
trans:
```

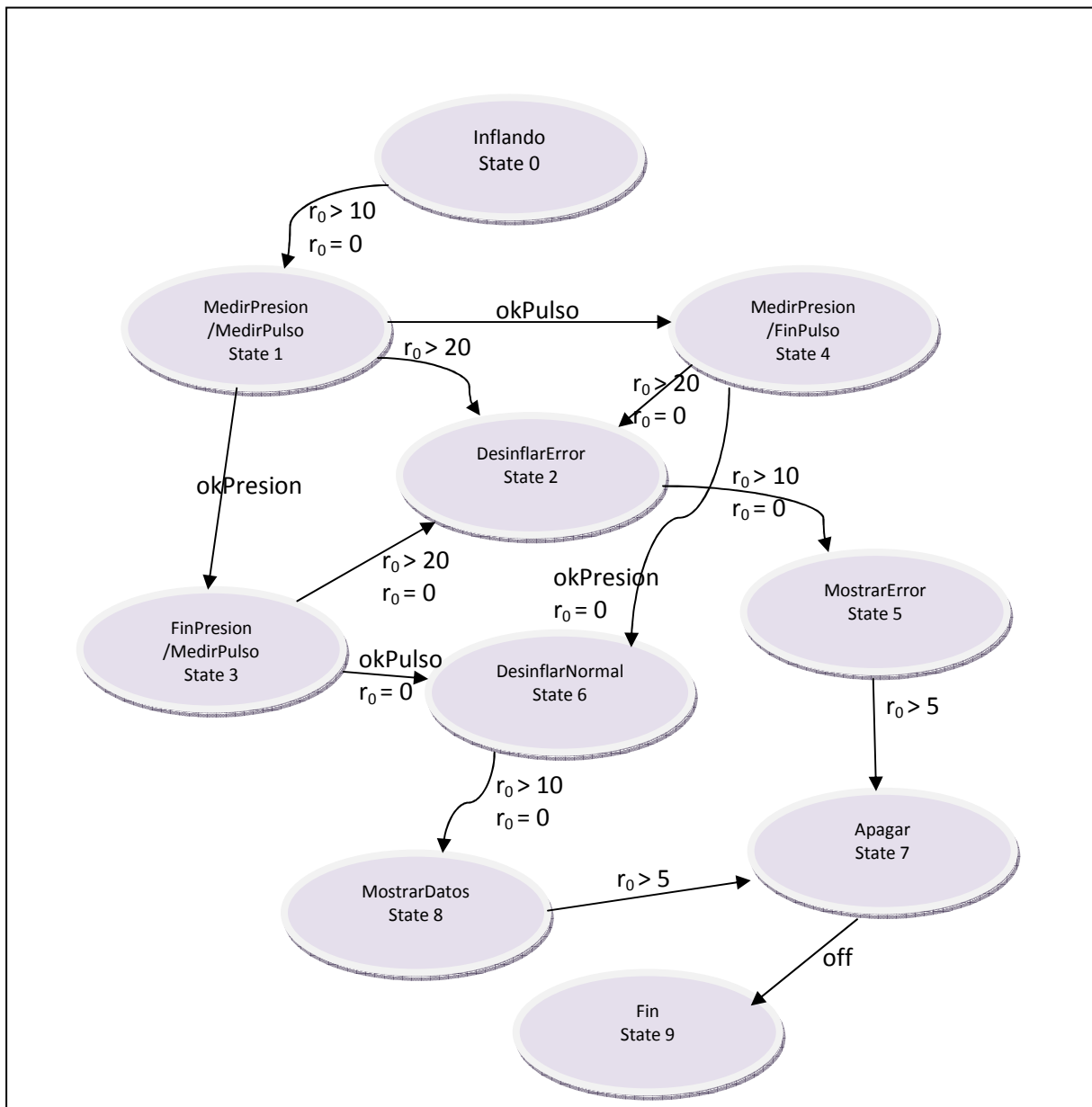


Figura 20: Autómata temporizado generado por el algoritmo

La traza generada por el *model checker* Zeus ante la propiedad 1 es la siguiente:

```

$ ./zeus2 med.tg obs-true.tg -p error
Parseando med.tg...
Parseando obs-true.tg...
Alcanzando 'error' (con trazas)...
Encontré prop.
Traza:
<0, 0> (r0<=10 and r0=x)
-- infla -->
<1, 0> (r0<=20 and r0+10=x)
-- error -->
<2, 0> (r0<=10 and r0+30=x)

```

```

-- desinf -->
<5, 0> (r0<=5 and r0+40<=x)
-- mostrar -->
<7, 0> (45<=x)
-- off -->
<9, 1> (45<=x)
-- fin -->
<9, 2> (true)
Propiedad alcanzada.

```

Donde,

-- evento -->	Indica el evento que se proceso, por ejemplo -- infla -->
<n ₁ , n ₂ >	Indica el par de nodos de la composición entre el <i>autómata temporizado</i> y el observador. n ₁ es un nodo del <i>autómata temporizado</i> y n ₂ es un nodo del observador
(invariante)	Indica el invariante que se cumple en cada nodo

Otras pruebas realizadas

Además del ejemplo presentado anteriormente, se realizaron una serie de ejecuciones con *state machines* tentativas que representaran los distintos posibles casos de modelado, con regiones concurrentes en diferentes niveles y distintas cantidades de estados y transiciones.

En la siguiente tabla se muestran comparativamente los tamaños de las *state machines* y *autómatas temporizados*, así como el tiempo que demandó la traducción. En el caso de las *state machine* se cuentan todos los estados (incluidas las regiones) menos los iniciales y el top. Con respecto a las transiciones de igual manera se cuentan todas (incluidas las transiciones de completitud en caso que las haya) pero no se contemplan las transiciones de inicio, tanto de la *state machine* como de los distintos estados compuestos.

Cabe aclarar que en los casos de *state machines* con igual cantidad de estados y transiciones, los *autómatas temporizados* generados varían en su conformación a raíz de que las transiciones se activan por distintos eventos, generando mayor o menor cantidad de *estados de configuración* válidos.

En referencia a los análisis realizados notamos que el tiempo que demanda en realizar la traducción depende del tamaño de los *autómatas temporizados* generados, y esto varía según la combinación de transiciones concurrentes que se lanzan en cada caso. Es decir, si vemos los ejemplos de las figuras 21 y 22, que representan a los casos 3 y 4 de la tabla, se nota que si bien las *state machines* son similares en cuanto a cantidad de estados y transiciones, como la combinación de eventos posibles activan distintas cantidad de transiciones, entonces varía la cantidad nodos y ejes en el *autómata temporizado*.

Si se ve en el caso 3, ante la llegada del evento e1, estando activos los estados Estado5 y Estado7, siempre van a avanzar en paralelo las dos regiones que los contienen. Sin embargo en el caso 4, van a avanzar en forma separada, generando mayor combinación de *estados de configuración* válidos, por ende mayor cantidad de nodos en el *autómata temporizado* resultante.

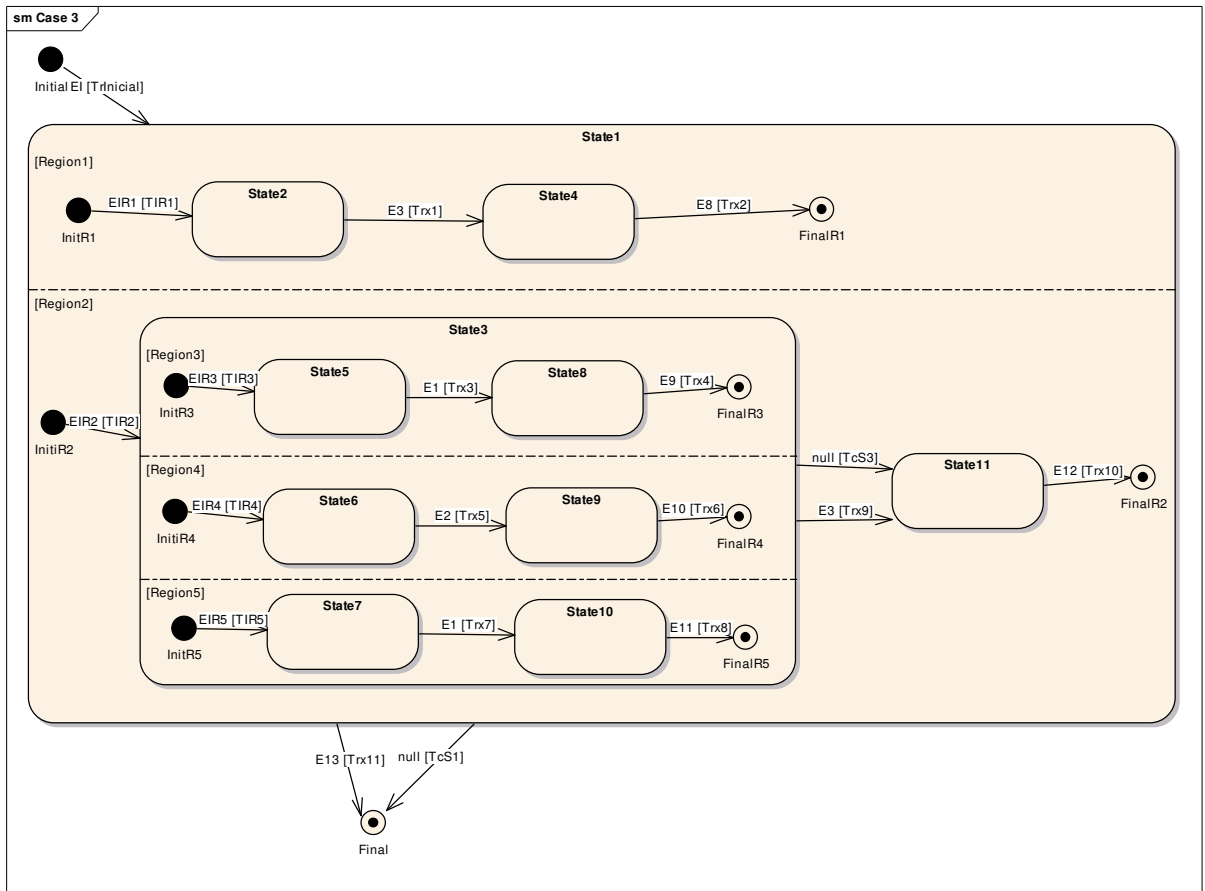


Figura 21: Ejemplo de *state machine*, caso 3 de la tabla de más abajo

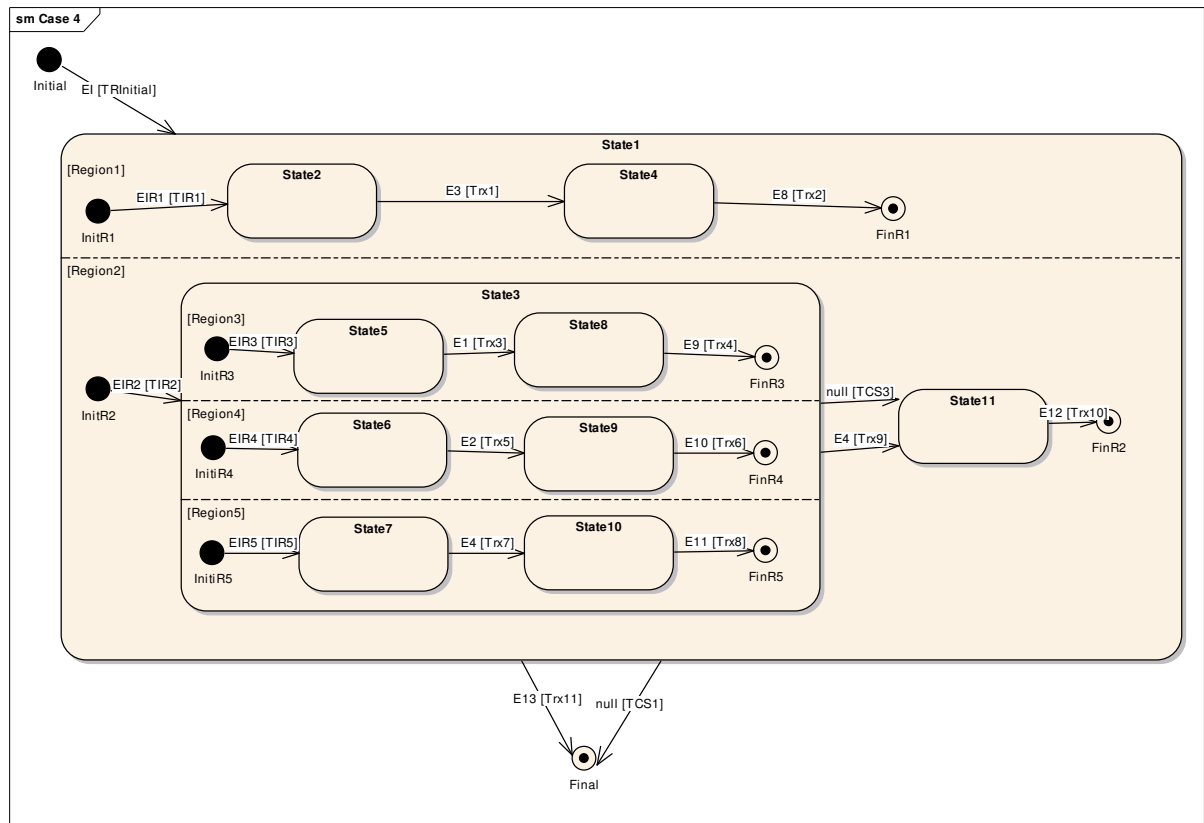


Figura 22: Ejemplo de *state machine*, caso 4 de la tabla de más abajo

Nro Caso	State machine		Estados de configuración analizados	Autómata temporizado		Tiempo	Tiempo por Estado de Configuración
	Estados	Transiciones		Nodos	Ejes		
1	10	6	5	4	5	47 ms	9,4
2	19	13	97	27	97	141 ms	1,453608247
3	19	13	65	20	65	110 ms	1,692307692
4	19	13	355	84	355	563 ms	1,585915493
5	21	14	90	33	90	156 ms	1,733333333
6	24	16	90	26	90	141 ms	1,566666667
7	6	8	8	6	8	31 ms	3,875
8	23	23	1484	297	1484	2735 ms	1,842991914

Tabla con los resultados de las corridas realizadas

En la tabla se puede observar que el tiempo sube de manera aproximadamente lineal con la cantidad de *estados de configuración* analizados, y la cantidad de estos es dominada principalmente por el nivel de concurrencia de la SM, aún en el caso 4, que consideramos representativo de un componente de tamaño medio que podría formar parte de un sistema real, la traducción toma apenas más que medio segundo, tiempo que es despreciable en comparación con los tiempos tradicionales de los *model checkers* que se usarán en la etapa siguiente del proceso de análisis. El gráfico de la Figura 23 muestra este comportamiento aproximadamente lineal. En este caso se eliminaron los dos casos de menor cantidad de estados de configuración, ya que por el tamaño tan chico los tiempos están influenciados por el overhead de la virtual machine.

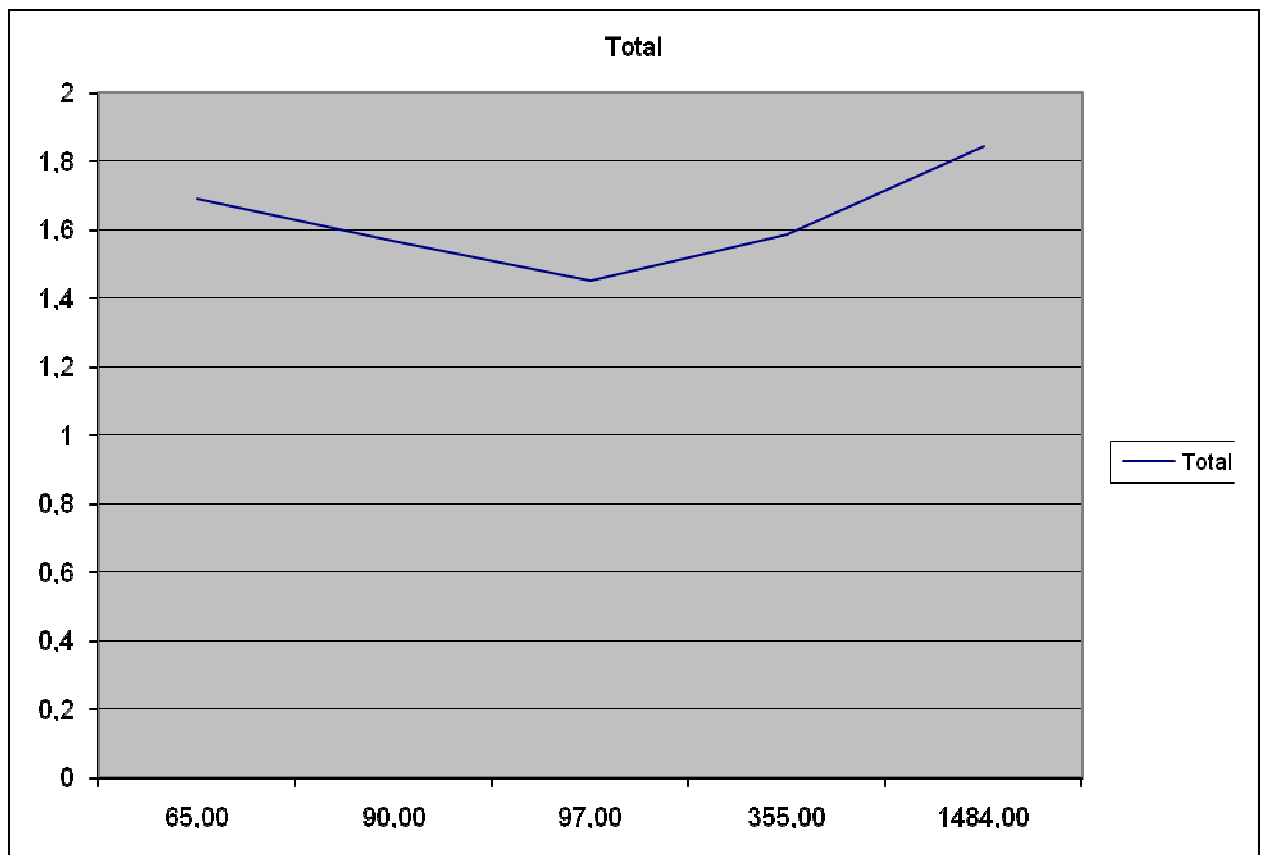


Figura 23: Comportamiento del algoritmo

11. Conclusiones

En el marco de esta tesis hemos desarrollado un algoritmo que traduce *state machines* conteniendo eventos de tiempo a *autómatas temporizados*. Estos pueden ser utilizados como entrada de un *model checker* para validar propiedades sobre el diseño expresado en la *state machine*.

Hay otros trabajos que mencionan haber realizado un algoritmo que hace esto pero sin presentarlo. En nuestro caso mostramos el algoritmo, explicamos en detalle el razonamiento de cómo fue creado, demostramos la versión del algoritmo sin relojes, justificamos el agregado de relojes y explicamos por qué en este caso no pudimos realizar una demostración. Se realizó el código ejecutable en Java, y se mostró en un ejemplo, como debería ser utilizado.

Con este ejemplo, pudimos demostrar que los siguientes pasos son una forma de trabajo en el diseño y desarrollo de sistemas de tiempo real:

1. Definir la *state machine*
2. Ejecutar nuestro algoritmo para transformar la *state machine* en un *autómata temporizado*
3. Escribir el observador con la propiedad que se quiera verificar
4. Ejecutar el *model checker*
5. En caso de que las propiedades no se verifiquen, modificar la *state machine* y volver a realizar el paso 2.

La construcción y prueba del algoritmo presentado nos permiten argumentar que es factible la utilización de *UML* para el modelado de sistemas de tiempo real, y que se no se pierde en este paso la posibilidad de realizar los chequeos formales necesarios.

12. Trabajos Futuros

Consideramos que luego de la realización de este trabajo hay algunos puntos que sería interesante profundizar y otros que podrían mejorar la herramienta.

Los puntos a profundizar son:

1. Soportar los componentes de la *state machine* que dejamos afuera por simplicidad o limitaciones del *model checker*. Ver sección “Reglas de Traducción”.
2. Soportar otros tipos de diagramas de *UML* para hacer más completa la especificación del sistema que se está modelando.
3. Traducir las trazas generadas por el *model checker* a diagramas de secuencia.

Las mejoras posibles a la herramienta serían:

1. Tomar como entrada del algoritmo un diagrama *UML*, una opción sería tomar el diagrama de la *state machine* generado por una herramienta comercial de diseño a través del formato XMI.
2. Integrar el algoritmo al *model checker*, con lo cual, no hay que estar generando una salida que después será tomada como entrada por el *model checker*.

13. Referencias

1. [Yov97] Sergio Yovine, Model Checking Timed Automata, Verimag Centre Equation, 1997.
2. [OMG05] OMG, Unified Modeling Language Specification, version 1.4.2, January 2005.
3. [LPY97] K. Larsen, P. Petterson, W. Yi, UPPAL in a nutshell, Springer International Journal of Software Tools for Technology Transfer, 1, 1997.
4. [DOT95] C. Daws, A. Olivero, S. Tripakis, S. Yovine, The Tool Kronos, Verimag, 1995.
5. [SGW94] Bran Selic, Garth Gullekson, Paul T. Ward, Real-Time Object-Oriented Modeling, Wiley, 1994.
6. [BLM03] Vieru Del Bianco, Luigi Lavazza, Marco Mauri, Giuseppe Occorso, Towards UML-based formal specifications of component-based real-time software, 134 Lecture Notes in Computer Science Publisher: Springer-Verlag Heidelberg Volume: Volume 2621 / 2003.
7. [KMR02] Alexander Knapp, Stephan Merz and Christopher Rauh, Model Checking Timed UML State Machines and Collaborations, 7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT). 2002
8. [Har87] Harel, D.: Statecharts, A Visual Formalism for Complex Systems, Science of Computer Programming, pp. 231-274, 1987.
9. [AD94] R. Alur, D. Dill, A Theory of Timed Automata, Theoretical Computer Science, vol. 126, pp. 183-225, 1994.
10. [BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1999.
11. [OMG07] OMG, Unified Modeling Language Superstructure, version 2.1.2, November 2007.
12. [OMG05] OMG, UML Profile for Schedulability, Performance, and Time Specification, version 1.1, January 2005.
13. [Scha07] Fernando Schapachnik, Verificación de autómatas temporizados en arquitecturas monoprocesador y multiprocesador, Ph D. Thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2007
14. [Guru06] R. Gurulingesh, Formal Methods for Safety-Critical Embedded Real Time System: A Case Study. 2006
15. [GNKM06] R. Gurulingesh, S. Neera, R. Krithi and M. Malewar, Efficient Real-Time Support for Automotive Applications: A Case Study, 12th IEEE 12th IEEE International Conference on Embedded and Real-Time Computing and Applications - RTCSA'2006, Aug 16-18, 2006, Sydney, Australia.