UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

FACULTAD DE INGENIERÍA

Proyecto de grado:

Construcción de un verificador de modelos

Juan Machado

Tutor:
Luis Sierra

Resumen

Este proyecto consiste en la construcción de un verificador de modelos que implemente los principales algoritmos de verificación. Para esto se escogieron dos lenguajes de propiedades que requieren distintos algoritmos de verificación. *LTL* requiere un algoritmo basado en autómatas que reconocen las ejecuciones de un sistema, tratando las mismas como palabras. Mientras que *CTL* requiere un algoritmo basado en el análisis de la satisfacibilidad de las fórmulas por los estados del sistema de transiciones.

Este verificador debe ser fácil de utilizar y de mantener, además de flexible de forma que sea posible agregar nuevos lenguajes para describir propiedades. Para lograr esto se realizó un diseño orientado a objetos y con módulos bien definidos apuntando a su mantenimiento y extensión.

También pensando en estos objetivos se escogió el lenguaje de programación *Python 2.7* debido a que cumple con los requisitos del diseño. Además es un lenguaje que mezcla programación funcional con programación imperativa, lo que permite implementar algoritmos complejos en pocas instrucciones generando un código fácil de comprender por otras personas.

Por otro lado se utilizó el formato de archivo *GraphML* para representar los sistemas de transiciones. Este formato es muy útil para representar grafos ya que existen varios editores para el mismo y al ser un lenguaje basado en *XML* existen muchas herramientas para su manipulación.

Además se utilizó la herramienta *Python Lex-Yacc* para parsear las fórmulas. Esta herramienta requirió la definición de las gramáticas correspondientes para cada lenguaje.

De esta forma se desarrolló un verificador fácil de utilizar, fácil de mantener, modularizado y flexible.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Este documento desarrolla la construcción de una herramienta para verificar propiedades sobre distintos sistemas. Estas propiedades son expresadas en distintas lógicas basadas en proposiciones mientras que los sistemas son representados por grafos etiquetados a los que llamaremos sistemas de transiciones. Esta herramienta es llamada verificador de modelos.

Este verificador es construído con fines académicos. Su objetivo es que sea fácil de utilizar, utilizando herramientas conocidas para modelar sistemas y lenguajes intuitivos para expresar propiedades. A su vez debe ser fácil de mantener y flexible para agregar otros tipos de lógica para expresar propiedades.

En los siguientes capítulos se introducen herramientas para modelar sistemas reactivos así como distintos lenguajes para expresar propiedades sobre estos sistemas.

Una vez que se tienen tanto el modelo del sistema representado por un sistema de transiciones TS, como la propiedad φ especificada formalmente en alguno de los lenguajes vistos, se utiliza un verificador de modelos para resolver el problema

$$TS \models \varphi$$

Esto significa que el sistema de transiciones TS cumple con la propiedad φ.

Los lenguajes seleccionados en este documento para especificar las propiedades utilizan distintos paradigmas de verificación. El algoritmo de verificación para LTL, uno de los lenguajes utilizados, construye un autómata que representa la propiedad y se basa en el reconocimiento de palabras para los autómatas. En este caso se realizan operaciones tanto sobre los autómatas como sobre el sistema de transiciones que representa el sistema reactivo. Por

otro lado el algoritmo utilizado para CTL, el otro lenguaje utilizado en este proyecto, se basa en el análisis de satisfacibilidad de la propiedad por cada estado del sistema de transiciones. En este caso no se realiza ninguna operación sobre el sistema de transiciones, por lo que no se modifica el modelo original del sistema.

El verificador de modelos implementado contiene la sintaxis de cada lenguaje así como la implementación de sus respectivos algoritmos de verificación de forma modular. De esta forma se mantiene cierta flexibilidad para agregar nuevos lengujes utilizando los algoritmos de verificación ya implementados.

Para finalizar se tratan algunos casos de estudio completos. Estos consisten en el modelado de un sistema, la especificación de propiedades y el análisis de su satisfacibilidad.

Capítulo 2

Sistemas de transiciones

Se quiere modelar propiedades sobre sistemas reactivos, es decir sistemas que interaccionan con el entorno. Los sistemas de transiciones son una herramienta que permite modelar el comportamiento de un sistema reactivo. Para esto debe ser capaz de representar los estados y acciones que integran el sistema reactivo a modelar.

En este capítulo se detalla como están compuestos los sistemas de transiciones. También se especifican las principales operaciones, las cuales permiten modelar la interacción entre distintos sistemas y por lo tanto representar sistemas reactivos más complejos.

2.1. Sistemas de transiciones

Los sistemas de transiciones son grafos dirigidos en los cuales los estados modelan los estados del sistema reactivo mientras que las transiciones representan los cambios estado.

Cada estado del sistema de transiciones tiene asociado un conjunto de proposiciones atómicas, las cuales son verdaderas en el estado representado por este nodo. Las transiciones tienen asociado un conjunto de acciones, las cuales permiten la sincronización entre distintos sistemas.

Definición 1. *Sistema de transiciones.*
Un sistema de transiciones consiste en una tupla $TS = (S, Act, \rightarrow, I, AP, L)$*, donde:*

- S *es el conjunto de estados*
- Act *es el conjunto de acciones*
- $\rightarrow \subseteq S \times Act \times S$ *es la relación de transición*

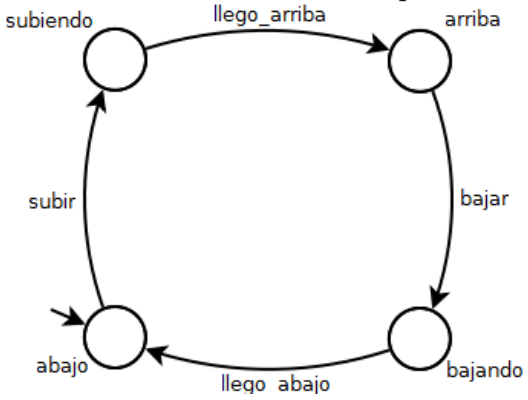
- $I \subseteq S$ *es el conjunto de estados iniciales*
- AP *es el conjunto de proposiciones atómicas*
- $L : S \to 2^{AP}$ *es la función de etiquetado*

A continuación se muestra un ejemplo de sistema de transiciones.

Ejemplo.

Se considera el sistema de transiciones de la figura 2.1 para modelar un ascensor.

Figura 2.1: Sistema de transiciones para un ascensor



Este sistema está definido de la siguiente forma:

- $S = \{abajo, subiendo, arriba, bajando\}$
- $Act = \{subir, llego\_arriba, bajar, llego\_abajo\}$
- $\to = \{(abajo, subir, subiendo), (subiendo, llego\_arriba, arriba), (arriba, bajar, bajando), (bajando, llego\_abajo, abajo)\}$
- $I = \{abajo\}$

Una posible elección para las proposiciones es:

- $AP = \{abajo, subiendo, arriba, bajando\}$

En este caso se pueden etiquitar los estados de la siguiente forma:

- $L(abajo) = \{abajo\}$

- $L(subiendo) = \{subiendo\}$
- $L(arriba) = \{arriba\}$
- $L(bajando) = \{bajando\}$

2.1.1. Ejecuciones

Para las siguientes definiciones se considera un sistema de transiciones $\mathrm{TS} = (\mathrm{S}, \mathrm{Act}, \rightarrow, \mathrm{I}, \mathrm{AP}, \mathrm{L})$.

Definición 2. *Predecesores y Sucesores.*
Sea $s \in S$ y $\alpha \in Act$ se define el conjunto de estados $\alpha-$sucesores como:

$$Post(s, \alpha) = \{s' \in S \mid s \xrightarrow{\alpha} s'\}$$

y el conjunto de estados sucesores como:

$$Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$$

De la misma forma se define el conjunto de estados $\alpha-$predecesores como:

$$Pre(s, \alpha) = \{s' \in S \mid s' \xrightarrow{\alpha} s\}$$

y el conjunto de estados predecesores como:

$$Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha)$$

Definición 3. *Fragmento de camino.*
Un fragmento de camino π es una secuencia de estados $s_0 s_1 s_2 ...$ donde $s_i \in Post(s_{i-1})$ para todo $i > 0$.

Definición 4. *Camino.*
Un camino π es un fragmento de camino infinito $\pi = s_0 s_1 s_2 ...$ con $s_0 \in I$, siendo I el conjunto de estados iniciales de TS.

Dados estos modelos de sistemas reactivos, las ejecuciones de los mismos se representan mediante caminos del sistema de transiciones. Los sistemas que se tratan en este proyecto no tienen estados finales, por lo que las ejecuciones nunca terminan.

2.2. Comunicación entre sistemas de transiciones

En la sección anterior se vieron los conceptos básicos de los sistemas de transiciones. En la práctica lo más común es que los sistemas a modelar sean complejos. Los sistemas que se tratan en este documento son en general sistemas en paralelo.

Para poder modelar este tipo de sistemas se necesita que los sistemas de transiciones puedan comunicarse. Para esto se definen las siguientes operaciones entre sistemas de transiciones. Estas operaciones permiten modelar la comunicación entre distintos sistemas y así poder modelar sistemas más complejos.

2.2.1. Intercalado

La primera de las formas de interacción es el intercalado. Este método consiste en ejecutar en cada paso una acción de uno de los sistemas independientemente de las acciones de los otros sistemas. Esto quiere decir que el sistema en ejecución en ese momento cambia de estado, pero el resto de los sistemas quedan inalterados.

El siguiente ejemplo ilustra la idea del intercalado de sistemas.

Ejemplo. Puertas del ascensor

Se consideran los sistemas de transiciones *PuertaAbajo* y *PuertaArriba* definidos en la figura 2.2 para modelar dos puertas de un ascensor.

Estos semáforos se encuentran en pisos distintos, por lo que el comportamiento de cada una de ellas es independiente de la otra.

El sistema de transiciones resultante $PuertaAbajo|||PuertaArriba$ se muestra en la figura 2.3.

Más formalmente el intercalado de dos sistemas de transiciones se define de la siguiente manera.

Figura 2.2: Puertas del ascensor

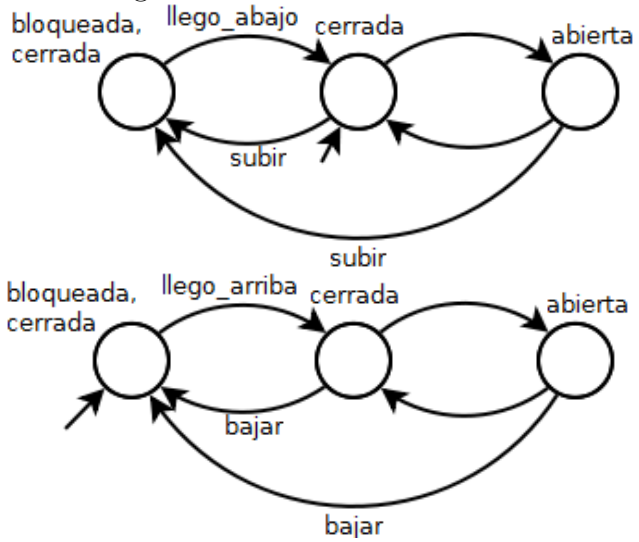
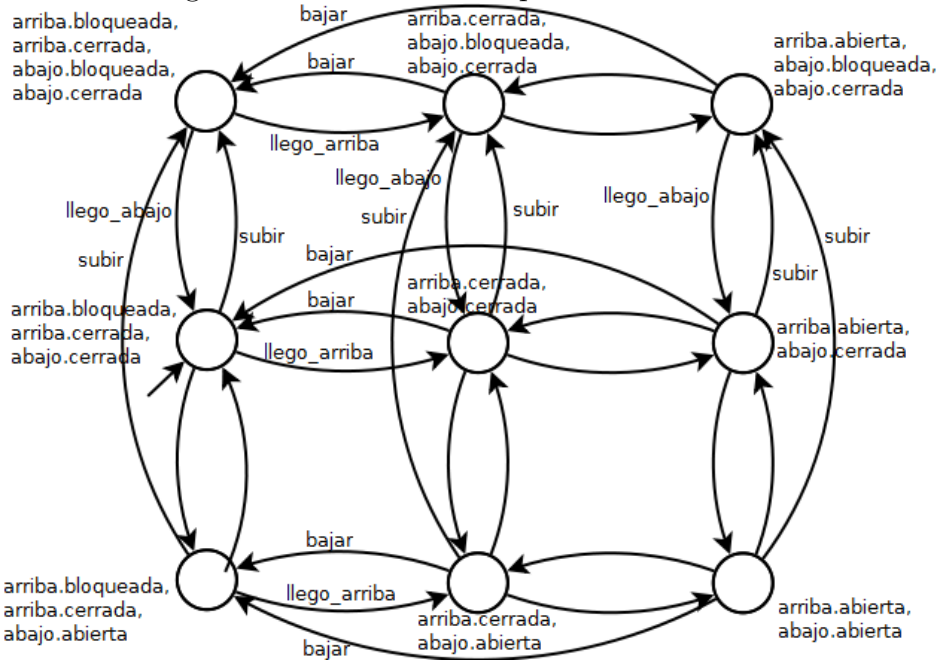


Figura 2.3: Intercalado de puertas del ascensor



Definición 5. *Intercalado de sistema de transiciones.*
Dados dos sistemas de transiciones

- $TS_1 = (S_1, Act_1, \rightarrow_1, I_1, AP_1, L_1)$
- $TS_2 = (S_2, Act_2, \rightarrow_2, I_2, AP_2, L_2)$

El sistema de transiciones $TS = TS_1 ||| TS_2$ *se define como*

$$TS = (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

Donde $L(s_1, s_2) = L_1(s_1) \cup L_2(s_2)$ *y la relación de transición* $\rightarrow$ *queda definida por las siguientes reglas*

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)} \qquad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{(s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')}$$

Se puede apreciar en el ejemplo de las luces de tránsito que el sistema de transiciones resultante cumple con esta definición.

2.2.2. Handshaking

En el método visto anteriormente los sistemas son independientes y no interactúan entre sí.

En el *Handshaking* los sistemas interactúan entre sí realizando acciones sincronizadas. Cada sistema tiene su conjunto de acciones propio, pero se toma un conjunto con acciones pertenecientes a ambos sistemas con el fin de sincronizarlos.

Definición 6. *Handshaking.*
Dados dos sistemas de transiciones

- $TS_1 = (S_1, Act_1, \rightarrow_1, I_1, AP_1, L_1)$
- $TS_2 = (S_2, Act_2, \rightarrow_2, I_2, AP_2, L_2)$

Y el conjunto $H \subseteq Act_1 \cap Act_2$*, el sistema de transiciones* $TS = TS_1 ||_H TS_2$ *se define como*

$$TS = (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

Donde $L(s_1, s_2) = L_1(s_1) \cup L_2(s_2)$ *y la relación de transición* $\rightarrow$ *queda definida por las siguientes reglas:*

- $\alpha \notin H$

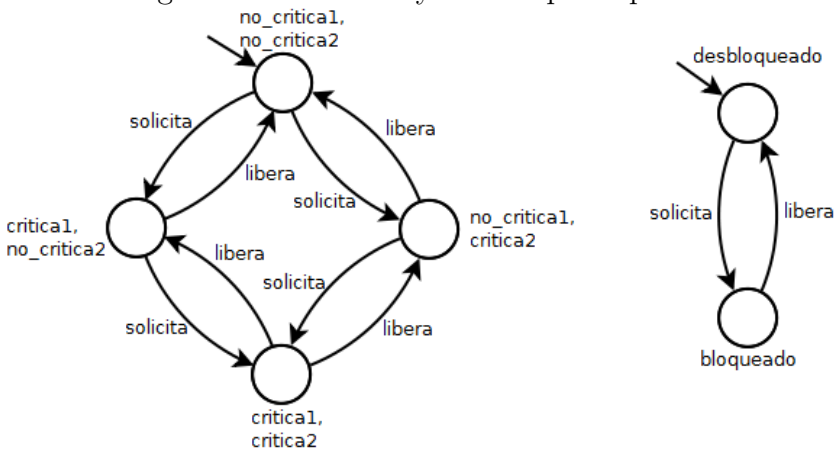
$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)} \qquad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{(s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')}$$

- $\alpha \in H$

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \quad s_2 \xrightarrow{\alpha}_2 s_2'}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')}$$

Ejemplo. Mutua exclusión con árbitro
Para modelar el problema de mutua exclusión se necesitan dos procesos con estados sección crítica y sección no crítica. Además se necesitan las acciones *solicita* y *libera* como se muestra en la figura 2.4.

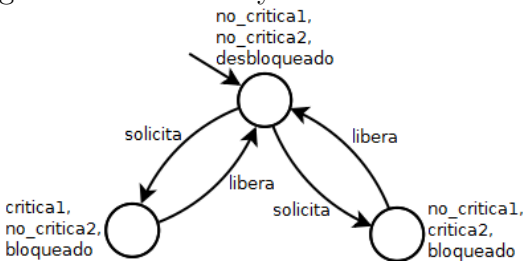
Figura 2.4: Procesos y árbitro por separado



Ahora para evitar que ambos procesos ingresen simultáneamente a la sección crítica se necesita un árbitro. Este árbitro ejecuta las acciones *solicita* y *libera* de forma que luego de cada *solicita* se ejecute un *libera*. Esto se logra sincronizando los sistemas de transiciones con el árbitro mediante el conjunto de $H = \{solicita, libera\}$. El sistema resultante se puede ver en la figura 2.5.

Continuando con el ejemplo del ascensor, se utiliza el mismo para mostrar el *Handshaking* de los sistemas antes mostrados.

Figura 2.5: Procesos y árbitro sincronizados

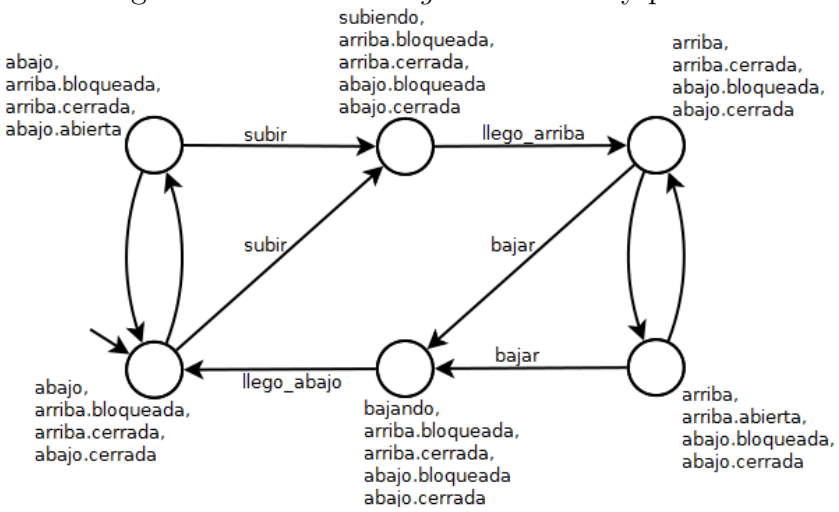


Ejemplo. *Handshaking* del ascensor y puertas
En la figura 2.6 se muestra el sistema

$$Ascensor||_H PuertaAbajo|||PuertaArriba$$

Donde $H = \{subir, llego\_arriba, bajar, llego\_abajo\}$

Figura 2.6: *Handshaking* del ascensor y puertas



Capítulo 3

Autómatas de Büchi

Los autómatas regulares fueron introducidos por Huffman [8] como herramientas para reconocer palabras de largo finito. Pero para representar las ejecuciones de un sistema se necesita trabajar con palabras de largo infinito. Para esto Büchi introdujo el concepto de Autómatas de Büchi (NBA) [9]. Estos son una variante de los autómatas regulares utilizados para reconocer palabras de largo infinito.

3.1. Lenguajes *omega*-regulares

Llamamos lenguajes regulares a los lenguajes que se pueden representar mediante expresiones regulares. Las palabras de estos lenguajes tienen largo finito. En cambio si queremos tratar con palabras de largo infinito necesitaremos una clase distinta de lenguajes, asociados con una clase distinta de expresiones. Estos lenguajes se denominan lenguajes *omega*-regulares.

Utilizaremos la letra griega ω (*omega*) para denotar la repetición infinita, por ejemplo a^ω es la palabra infinita que contiene sólo símbolos a.[1]

Llamamos Σ^ω al conjunto de todas las palabras infinitas sobre Σ. Cualquier subconjunto de Σ^ω es un lenguaje de palabras infinitas.

Definición 7. *Expresión ω-regular.*
Una expresión ω-regular G sobre el alfabeto Σ tiene la siguiente forma

$$G = E_1.F_1^\omega + ... + E_n.F_n^\omega$$

donde $n \geq 1$ y $E_1, ..., E_n, F_1, ..., F_n$ son expresiones regulares sobre Σ tales que $\varepsilon \notin L(F_i)$, siendo $L(F_i)$ el lenguaje representado por la expresión regular F_i.

[1] Notar que a^* representa un lenguaje, es el conjunto de todas las palabras finitas que contienen únicamente el símbolo a, mientras que a^ω representa solamente una palabra.

Un lenguaje ω-regular, análogamente a los regulares, es un lenguaje que se puede representar mediante una expresión ω-regular.

Definición 8. *Lenguaje ω-regular.*
Un lenguaje $\mathcal{L}$ es ω-regular si $\mathcal{L} = L_\omega(G)$, donde $L_\omega(G)$ es el conjunto de palabras representado por la expresión ω-regular G sobre Σ.

En particular trataremos con ciertos lenguajes ω-regulares, a los que llamaremos propiedades ω-regulares.

Definición 9. *Propiedad ω-regular.*
Dado un conjunto AP de proposiciones atómicas. Se dice que una propiedad P sobre AP es ω-regular si P es un lenguaje ω-regular sobre el alfabeto 2^{AP}.

3.2. Autómatas de Büchi

Los Autómatas de Büchi (NBA) están compuestos por un conjunto de estados Q, un conjunto de estados iniciales $Q_0 \subseteq Q$, un alfabeto Σ sobre el cual se definen las acciones a tomar en cada estado, una función de transición $\delta : Q \times \Sigma \to 2^Q$, y un conjunto de estados de aceptación $F \subseteq Q$.

Los NBA se diferencian de los autómatas regulares por su comportamiento, más precisamente en el criterio de aceptación. El lenguaje reconocido por un NBA es el conjunto que contiene todas las palabras cuya ejecución pasa infinitas veces por alguno de los estados del conjunto de aceptación.

A continuación se define la estructura de los NBA.

Definición 10. *Autómatas de Büchi.*
Los Autómatas de Büchi (NBA) consisten en una tupla $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$. Donde:

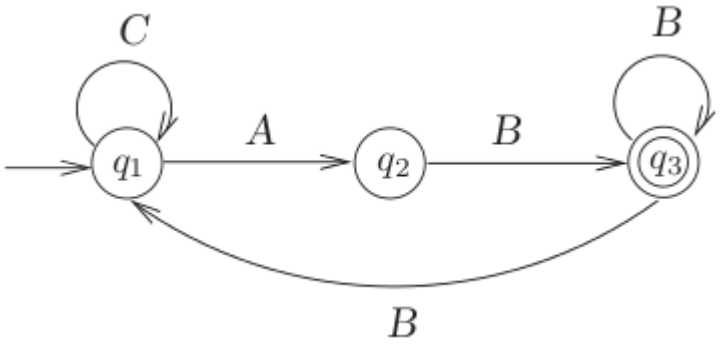
- *Q es un conjunto finito de estados*
- *Σ es un alfabeto*
- *$\delta : Q \times \Sigma \to 2^Q$ es una función de transición*
- *$Q_0 \subseteq Q$ es un conjunto de estados iniciales*
- *$F \subseteq Q$ es un conjunto de aceptación*

Ejemplo. Considerando el NBA de la figura 3.1 sobre el alfabeto $\{A, B, C\}$.

El lenguaje aceptado por el NBA está dado por la expresión regular

$$C^*AB(B^+ + BC^*AB)^\omega$$

Figura 3.1: Ejemplo de Autómata de Büchi



3.3. Autómata de Büchi Generalizado

En este trabajo se utiliza una variante de los NBA, el Autómata de Buchi Generalizado (GNBA). La estructura de los GNBAs es muy similar a la de los NBAs, varía sólo en el conjunto de aceptación. En lugar de este contiene un conjunto F cuyos elementos son los conjuntos de aceptación F_i. El criterio de aceptación en estos autómatas es visitar infinitas veces todos los conjuntos de aceptación F_i.

Definición 11. *Autómatas de Büchi Generalizado.*
Los Autómatas de Büchi Generalizado (GNBA) consisten en una tupla $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$*, donde:* Q*,* Σ*,* δ *y* Q_0 *se definen igual que en los NBA mientras que* $\mathcal{F} \subseteq 2^Q$ *es el conjunto de aceptación.*

El lenguaje aceptado por un GNBA $\mathcal{G}$ consiste en todas las palabras que pasan infinitas veces por todos los conjuntos de aceptación.

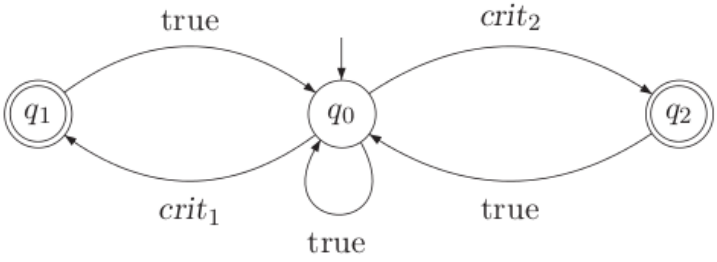
Más formalmente, dado un GNBA $\mathcal{G} = (\mathrm{Q}, \Sigma, \delta, \mathrm{Q}_0, \mathcal{F})$ y una secuencia infinita de estados $q_0q_1q_2q_3...$, esta es aceptada por $\mathcal{G}$ si

$$\forall \mathrm{F} \in \mathcal{F}, \exists \text{ infinitas } j \in \mathbb{N} \text{ tal que } q_j \in F$$

Ejemplo. En la figura 3.2 se muestra un GNBA sobre el alfabeto 2^{AP} donde $AP = \{crit_1, crit_2\}$ y los conjuntos de aceptación $F_1 = \{q_1\}$ y $F_2 = \{q_2\}$.

El lenguaje aceptado es la propiedad que contiene todas las palabras infinitas tal que $crit_1$ y $crit_2$ ocurren infinitas veces. Este GNBA representa la propiedad de que ninguno de los procesos quede infinitamente esperando para ingresar a la sección crítica.

Figura 3.2: GNBA para sección crítica



Capítulo 4

Lógica Temporal Lineal

En este capítulo se realiza una introducción a la Lógica Temporal Lineal, como una extensión de Lógica Proposicional para poder expresar propiedades en sistemas reactivos. Con esta finalidad, este lenguaje fue introducido por Pnueli en [10].

La Lógica Lineal Temporal permite expresar propiedades sobre los sistemas en cuestón, ya que se agregan operadores que hacen referencia al tiempo y permite representar los distintos estados en distintos momentos durante la ejecución del sistema.

4.1. Sintaxis y semántica

4.1.1. Sintaxis de LTL

Las fórmulas de LTL son construídas según la siguiente gramática

$$\varphi ::= true \mid a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \bigcirc\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi \cup \varphi$$

Los operadores temporales permiten establecer relaciones entre las etiquetas de los estados del sistema en una ejecución. Estos nuevos operadores son:

- $\Diamond p \longrightarrow$ eventually p : inevitablemente en el futuro se cumplirá p
- $\Box p \longrightarrow$ always p : ahora y en el futuro se cumple p
- $\bigcirc p \longrightarrow$ next p : en el siguiente paso se cumple p
- $p \cup q \longrightarrow p$ until q : q se cumplirá inevitablemente en el futuro, y mientras tanto se cumple p

Ejemplo. Tomando como ejemplo el problema de mutua exclusión entre dos procesos P_1 y P_2, donde los procesos son modelados por tres estados:

(1) sección no crítica

(2) espera para entrar a la sección crítica

(3) sección crítica

Sean las proposiciones $wait_i$ y $crit_i$ que representan los estados de espera y sección crítica respectivamente para el proceso P_i, se pueden representar las siguientes propiedades con LTL.

- P_1 y P_2 nunca acceden simultaneamente a la sección crítica

$$\Box(\neg crit_1 \vee \neg crit_2)$$

- P_1 y P_2 acceden infinitas veces a la sección crítica

$$(\Box\Diamond crit_1) \wedge (\Box\Diamond crit_2)$$

4.1.2. Semántica de LTL

La satisfacibilidad en LTL se basa en la satisfacibilidad de trazas, o más precisamente de fragmentos de traza. A continuación se definen los conceptos de *traza* y *satisfacibilidad* para una fórmula LTL.

Definición 12. *Trazas.*
Sea un sistema de transiciones $TS = (S, Act, \rightarrow, I, AP, L)$ *y un fragmento de camino* $\pi = s_0 s_1 s_2 ....$ *La traza correspondiente a* π *es* $Traza(\pi) = L(s_0)L(s_1)L(s_2)...$
Además se definen las trazas de un sistema de transiciones como:

$$Trazas(TS) = \{\, Traza(\pi) \mid \pi \text{ es un camino de } TS \}$$

Definición 13. *Satisfacibilidad en LTL para trazas.*
Dada una traza $\sigma = A_0 A_1 A_2 ...$ *y dos fórmulas LTL* φ, ψ.

- $\sigma \models true$
- $\sigma \models a$ *si y solo si* $a \in A_0$
- $\sigma \models \neg\varphi$ *si y solo si* $\sigma \not\models \varphi$
- $\sigma \models \varphi \wedge \psi$ *si y solo si* $\sigma \models \varphi$ *y* $\sigma \models \psi$
- $\sigma \models \bigcirc\varphi$ *si y solo si* $\sigma[1...] \models \varphi$

- $\sigma \models \Diamond\varphi$ *si y solo si* $\exists j \geq 0, \sigma[j...] \models \varphi$
- $\sigma \models \Box\varphi$ *si y solo si* $\forall j \geq 0, \sigma[j...] \models \varphi$
- $\sigma \models \varphi \cup \psi$ *si y solo si* $\exists j \geq 0, \sigma[j...] \models \psi$ *y* $\forall 0 \leq i < j, \sigma[i...] \models \varphi$

Una vez definida la satisfacibilidad para trazas, se define la satisfacibilidad para un sistema de transiciones.

Definición 14. *Satisfacibilidad en LTL para un sistema de trasiciones.*
Dado un sistema se transiciones $TS = (S, Act, \rightarrow, I, AP, L)$ *y una fórmula LTL* φ*.*

La relación de satisfacibilidad para un sistema de transiciones se define como

$$TS \models \varphi \text{ si y sólo si } \sigma \models \varphi, \forall \sigma \in \textit{Trazas}(TS)$$

4.1.3. Propiedades

Las siguientes propiedades son necesarias para poder expresar cualquier fórmula de LTL utilizando un conjunto reducido de operadores ($true, \neg, \wedge, \bigcirc, \cup$).

- $p \vee q = \neg(\neg p \wedge \neg q)$
- $p \rightarrow q = \neg(p \wedge \neg q)$
- $p \leftrightarrow q = \neg(p \wedge \neg q) \wedge \neg(\neg p \wedge q)$
- $\Diamond p = true \cup p$
- $\Box p = \neg(true \cup \neg p)$

Definición 15. *Conjunto funcionalmente completo de conectivos.*
Sea un conjunto de conectivos C*. Decimos que* C *es funcionalmente completo si todas las fórmulas tienen una fórmula equivalente que utiliza únicamente conectivos de* C*.*

A partir de las propiedades anteriores se puede afirmar que el conjunto $\{true, \neg, \wedge, \bigcirc, \cup\}$ es un conjunto funcionalmente completo.

De esta forma se normaliza cada fórmula LTL a una equivalente pero con un conjunto reducido de operadores a fin de facilitar los prosesos posteriores.

4.2. Algoritmo de Verificación

Dado un sistema de transiciones T que modela nuestro sistema, se quiere determinar si todas sus ejecuciones cumplen con una propiedad dada φ.

$$ejecuciones(T) \subseteq \mathcal{L}(\varphi)$$

Este problema se puede ver como determinar si existe una ejecución del sistema que no cumple φ, o dicho de otra forma, que cumple $\neg\varphi$.

$$ejecuciones(T) \cap \mathcal{L}(\neg\varphi) = \emptyset$$

El procedimiento para verificarlo consta de tres pasos. Cada uno de ellos se puede tratar de forma individual.

El primer paso es construir un NBA $\mathcal{A}$ que reconozca los malos comportamientos, esto es que reconozca las palabras que cumplen con $\neg\varphi$. Este autómata es útil para encontrar las ejecuciones que no cumplen con la propiedad deseada.

A continuación se debe realizar el producto $T \otimes \mathcal{A}$, donde T es el modelo de nuestro sistema y $\mathcal{A}$ el autómata obtenido en el paso anterior. De esta forma intersectamos las ejecuciones de nuestro sistema con las palabras que no cumplen la propiedad a verificar.

Por último debemos verificar que el lenguaje representado por el autómata resultante es vacío, esto significa que no hay ninguna ejecución del sistema que cumple con la propiedad $\neg\varphi$. En caso de no ser vacío, cualquiera de las palabras contenidas en él sirve como contraejemplo, o en otras palabras, como ejemplo de comportamiento no deseado.

4.2.1. Construcción de un GNBA a partir de una fórmula LTL

Veremos como construir un GNBA a partir de una fórmula φ.

Lo primero es calcular el conjunto $clausura(\varphi)$ que se define a continuación.

Definición 16. *Clausura.*
La clausura de una fórmula LTL es el conjunto formado por todas sus subfórmulas y las negaciones de las mismas.

Ahora se necesita construir los conjuntos elementales con respecto a $clausura(\varphi)$.

Definición 17. *Conjunto elemental.*
Sea B un subconjunto de la clausura φ. B es un conjunto elemental de φ si cumple:

- *es consistente con respecto a la lógica proposicional:*
  - $\varphi_1 \wedge \varphi_2 \in B \Longleftrightarrow \varphi_1 \in B$ *y* $\varphi_2 \in B$
  - $\neg\psi \in B \Longrightarrow \psi \notin B$
  - $true \in clausura(\varphi) \Longrightarrow true \in B$
- *es maximal:*
  - $\varphi_2 \in B \Longrightarrow \varphi_1 \cup \varphi_2 \in B$
  - $\varphi_1 \cup \varphi_2 \in B$ *y* $\varphi_2 \notin B \Longrightarrow \varphi_1 \in B$
- *es localmente consistente con respecto al operador until:*
  - $\psi \notin B \Longrightarrow \neg\psi \in B$

Una vez definidos los conceptos anteriores se puede comenzar a construir el GNBA para una fórmula LTL de la siguiente manera:

1. El conjunto de estado está formado por todos los conjuntos de fórmulas elementales de φ
2. $Q_0 = \{B \in Q \mid \varphi \in B\}$
   El conjunto de estados iniciales está formado por todos los estados que contienen a φ
3. $F = \{F_{\varphi_1 \cup \varphi_2} \mid \varphi_1 \cup \varphi_2 \in clausura(\varphi)\}$ donde $F_{\varphi_1 \cup \varphi_2} = \{B \in Q \mid \varphi_1 \cup \varphi_2 \notin B \text{ o } \varphi_2 \in B\}$ es el conjunto de aceptación
4. La relación de trasición $\delta : Q \times 2^{AP} \to 2^Q$ queda definida por:
   - si $A \neq B \cap AP$ entonces $\delta(B, A) = \phi$
   - si $A = B \cap AP$ entonces $\delta(B, A) = B'$, donde B' es el conjunto elemental de fórmulas que cumple:
     i para cada $\bigcirc\psi \in clausura(\varphi) : \bigcirc\psi \in B \Leftrightarrow \psi \in B'$ y
     ii para cada $\varphi_1 \cup \varphi_2 \in clausura(\varphi)$:
     $\varphi_1 \cup \varphi_2 \in B \Leftrightarrow (\varphi_2 \in B \vee (\varphi_1 \in B \wedge \varphi_1 \cup \varphi_2 \in B'))$

Con este mecanismo podemos generar un GNBA para cualquier fórmula LTL, asumiendo que estas fórmulas LTL contienen únicamente los conectivos *true*, $\neg$, $\wedge$, $\bigcirc$ y $\cup$. En caso de que esto no se cumpla, se puede generar un GNBA para una fórmula equivalente que utilice sólo estos conectivos con las propiedades vistas anteriormente.

Ejemplo. Considerando la fórmula $\varphi = a \cup b$.
Los conjuntos de fórmulas elementales de φ son:

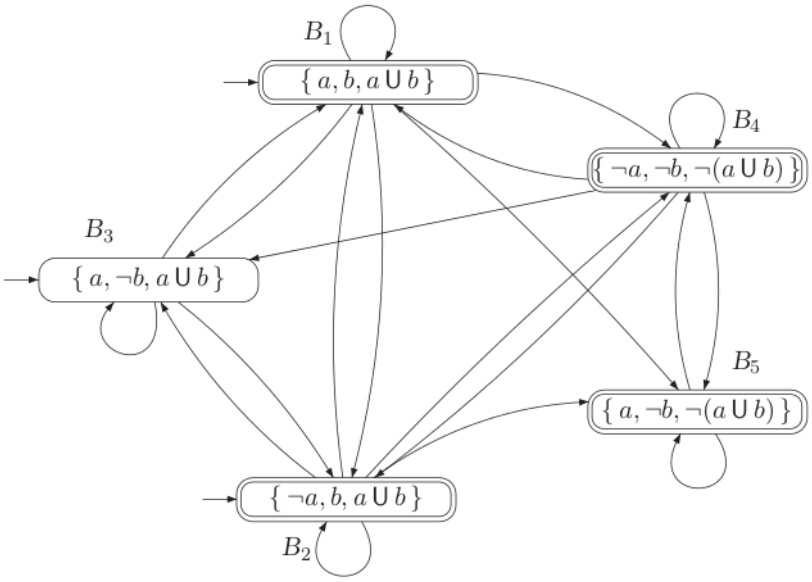
$$B_1 = \{a, b, \varphi\}, B_2 = \{\neg a, b, \varphi\}, B_3 = \{a, \neg b, \varphi\}, B_4 = \{\neg a, \neg b, \neg\varphi\}, B_5 = \{a, \neg b, \neg\varphi\}$$

Los estados iniciales son los conjuntos B_i tal que $\varphi \in B_i$. $Q_0 = \{B_1, B_2, B_3\}$.
El conjunto de conjuntos de aceptación es $F = \{F_\varphi\}$, donde

$$F_\varphi = \{B \in Q \mid \varphi \in B \text{ o } q \in B\} = \{B_1, B_2, B_4, B_5\}$$

El GNBA resultante se muestra en la figura 4.1.

Figura 4.1: GNBA para la fórmula $a \cup b$ [a]



[a] Imagen tomada de [1]

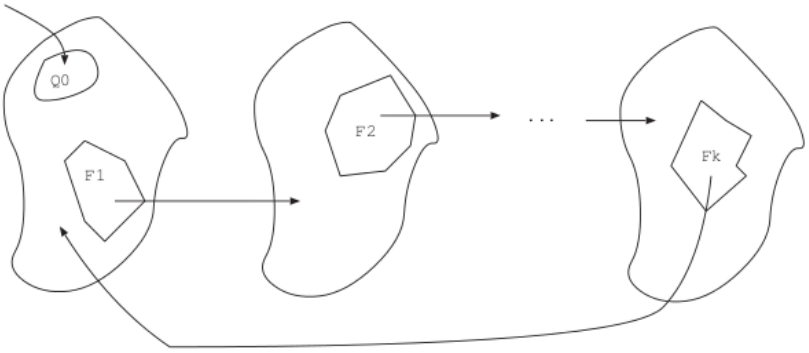
Una vez obtenido el GNBA para una fórmula φ debemos transformar este en un NBA equivalente. Para esto primero debemos entender la principal dieferencia entre ambos. Esta radica en su criterio de aceptación. Una palabra es reconocida (o aceptada) por un NBA si esta pasa infinitas veces por al menos un estado de su conjunto de aceptación. Mientras que una palabra es reconocida por un GNBA si esta pasa infinitas veces por al menos un estado de cada conjunto de aceptación.

Sabiendo esto se puede encontrar un método sencillo para transformar un GNBA en un NBA equivalente.

Sea (G) un GNBA y $(F) = \{F_1, ..., F_k\}$ el conjunto formado por todos sus estados de aceptación. El método consiste en crear k copias de (G) tales que

cada estado del conjunto F_i está conectado con los correspondientes estados en la copia $i + 1$ como se muestra en la figura 4.2.

Figura 4.2: Generación de un NBA a partir de un GNBA [a]



[a]Imagen tomada de [1]

El conjunto de aceptación del NBA resultante es el conjunto F_1 de la copia 1.

De esta forma nos aseguramos de que toda palabra que pase infinitas veces por alguno de los estados de F_1 pasa infitas veces por al menos un estado de cada conjunto de aceptación de $\mathcal{G}$.

4.2.2. Producto

En segundo lugar se realiza el producto $T \otimes \mathcal{A}$, siendo $\mathcal{A}$ el NBA resultante de la parte anterior.

El lenguaje reconocido por el producto de dos autómatas es la intersección de los lenguajes reconocidos por cada uno de ellos.

El producto $\mathcal{A} = \mathcal{A}_1 \otimes \mathcal{A}_2$ queda definido de la siguiente manera

- El conjunto de estados $Q = Q_1 \times Q_2$ siendo Q_1 y Q_2 los conjuntos de estados de $\mathcal{A}_1$ y $\mathcal{A}_2$ respectivamente.
- El conjunto de estados iniciles $Q_0 = Q_{0_1} \times Q_{0_2}$ siendo Q_{0_1} y Q_{0_2} los conjuntos de estados iniciales de $\mathcal{A}_1$ y $\mathcal{A}_2$ respectivamente.
- La función de transición δ cumple

$$\delta_1(q_1, A) = q_1' \;\; y \;\; \delta_2(q_2, A) = q_2' \;\; entonces \;\; \delta((q_1, q_2), A) = (q_1', q_2')$$

- En este proyecto se trabaja con sistemas reactivos, y al estos no tener estados finales, el conjunto de aceptación es $F = \{(q_1, q_2) \mid q_1 \in F_1\}$ siendo F_1 el conjunto de aceptación de $\mathcal{A}_1$.

4.2.3. Algoritmo de verificación

Por último se debe verificar si las trazas del sistema de transiciones T no contienen ninguna palabra que sea reconocida por $\mathcal{A}$, o sea un mal comportamiento de nuestro sistema. Esto significa que ninguna ejecución de T es reconocida por el autómata $\mathcal{A}$.

Lo anterior es equivalente a controlar que $T \otimes \mathcal{A} \models \Diamond F$ siendo F un estado de aceptación. Esto se logra verificando si existe algún estado de aceptación alcanzable y que al mismo tiempo pertenezca a un ciclo.

Capítulo 5

Lógica de Árbol Computacional

En los capítulos anteriores se analiza la verificación de un sistema mediante propiedades expresadas en LTL. LTL es una lógica lineal, esto se debe a que la noción de tiempo se basa en caminos. Las distintas ejecuciones se tratan como secuencias de estados, en donde cada estado tiene un único sucesor posible.

En este capítulo se trata la lógica arborecente CTL, introducida en [11] por Clarke y Emerson. A diferencia de LTL, en CTL el conjunto de ejecuciones se representa con un único árbol en donde los nodos representan los estados y las aristas representan las distintas decisiones posibles en cada estado.

Debido a esto, LTL y CTL son lógicas incompatibles entre si. Esto significa que existen fórmulas en LTL para las que no existe una fórmula equivalente en CTL y viceversa.

En este capítulo se introducirán los conceptos básicos de CTL, su sintaxis, semántica, y propiedades básicas que serán de utilidad en los capítulos posteriores. Además se mostrará su incompatibilidad con LTL.

Finalmente se construirá un algoritmo que permite verificar si el modelo del sistema en cuestión satisface la propiedad planteada en términos de CTL mostrando ejemplos y contraejemplos según corresponda.

5.1. Sintáxis y semántica

En CTL se mantienen los conectivos de LTL. Sus significados son los mismos, pero a diferencia de LTL, los conectivos temporales están sujetos a cuantificadores que hacen referencia a las posibles ejecuciones desde el estado actual. Esto es, mirando el árbol de ejecuciones, todos los posibles caminos a partir del estado actual.

5.1.1. Sintaxis de CTL

Las fórmulas de CTL son construídas según la siguiente gramática

$$\varphi ::= true \mid a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists\bigcirc\varphi \mid \exists\Box\varphi \mid \exists\Diamond\varphi \mid \exists\varphi\cup\varphi \mid \forall\bigcirc\varphi \mid \forall\Box\varphi \mid \forall\Diamond\varphi \mid \forall\varphi\cup\varphi$$

Donde a es una proposición atómica.

Ejemplos de fórmulas correctas. Dado el conjunto de proposiciones $\{p, q\}$, las siguientes son fórmulas CTL correctas:

- $\exists \bigcirc p$
- $\forall p \cup q$
- $\exists\Diamond(\forall((\exists\Box p) \cup q))$

Ejemplo de la mutua exculsión. Considerando el ejemplo visto en el capítulo anterior para verificar la mutua exclusión: La siguiente fórmula indica que los dos procesos no pueden acceder a la sección crítica simultáneamente

$$\forall\Box(\neg crit_1 \vee \neg crit_2)$$

Y la siguiente fórmula expresa que ambos procesos accederán infinitas veces a la sección crítica

$$(\forall\Box\forall\Diamond crit_1) \wedge (\forall\Box\forall\Diamond crit_2)$$

5.1.2. Semántica de CTL

La satisfacibilidad en CTL para un sistema de transiciones se basa en la satisfacibilidad de estados del mismo, a diferencia de LTL que se basa en la satisfacibilidad de trazas.

A continuanción se presenta el concepto de satisfacibilidad de una fórmula CTL por un estado del sistema. Este es el concepto de partida para analizar la satisfacibilidad para los sistemas de transiciones.

Definición 18. *Satisfacibilidad en CTL.*
Dado un sistema de transiciones con conjunto de estados S. Dados $s \in S$ y φ, ψ fórmulas CTL.

- $s \models a$ *si y sólo si* $a \in L(s)$
- $s \models \neg\varphi$ *si y sólo si* $s \not\models \varphi$

- $s \models \varphi \wedge \psi$ *si y sólo si* $s \models \varphi$ *y* $s \models \psi$
- $s \models \exists \bigcirc \varphi$ *si y sólo si* $\pi \models \bigcirc\varphi$ *para algún* $\pi \in Caminos(s)$
- $s \models \exists\varphi \cup \psi$ *si y sólo si* $\pi \models \varphi \cup \psi$ *para algún* $\pi \in Caminos(s)$
- $s \models \forall \bigcirc \varphi$ *si y sólo si* $\pi \models \bigcirc\varphi$ *para todo* $\pi \in Caminos(s)$
- $s \models \forall\varphi \cup \psi$ *si y sólo si* $\pi \models \varphi \cup \psi$ *para todo* $\pi \in Caminos(s)$

Donde:

- $\pi \models \bigcirc\varphi$ *si y solo si* $\pi[1] \models \varphi$
- $\pi \models \varphi \cup \psi$ *si y solo si* $\exists j \geq 0 :: (\pi[j] \models \psi \ y \ (\forall k : 0 \leq k < j : \pi[k] \models \varphi))$

Ahora que está definida la satisfacibilidad para un estado cualquiera de un sistema de transiciones se tratará la satisfacibilidad para un sistema.

Definición 19. *Satisfacibilidad en CTL para un sistema de trasiciones. Dado un sistema se transiciones* $TS = (S, Act, \rightarrow, I, AP, L)$ *y una fórmula CTL* φ*.*

Sea el conjunto de satisfacibilidad $Sat(\varphi) = \{s \in S \mid s \models \varphi\}$*.*

La relación de satisfacibilidad para un sistema de transiciones se define como

$$TS \models \varphi \text{ si y sólo si } s_0 \models \varphi, \forall s_0 \in I$$

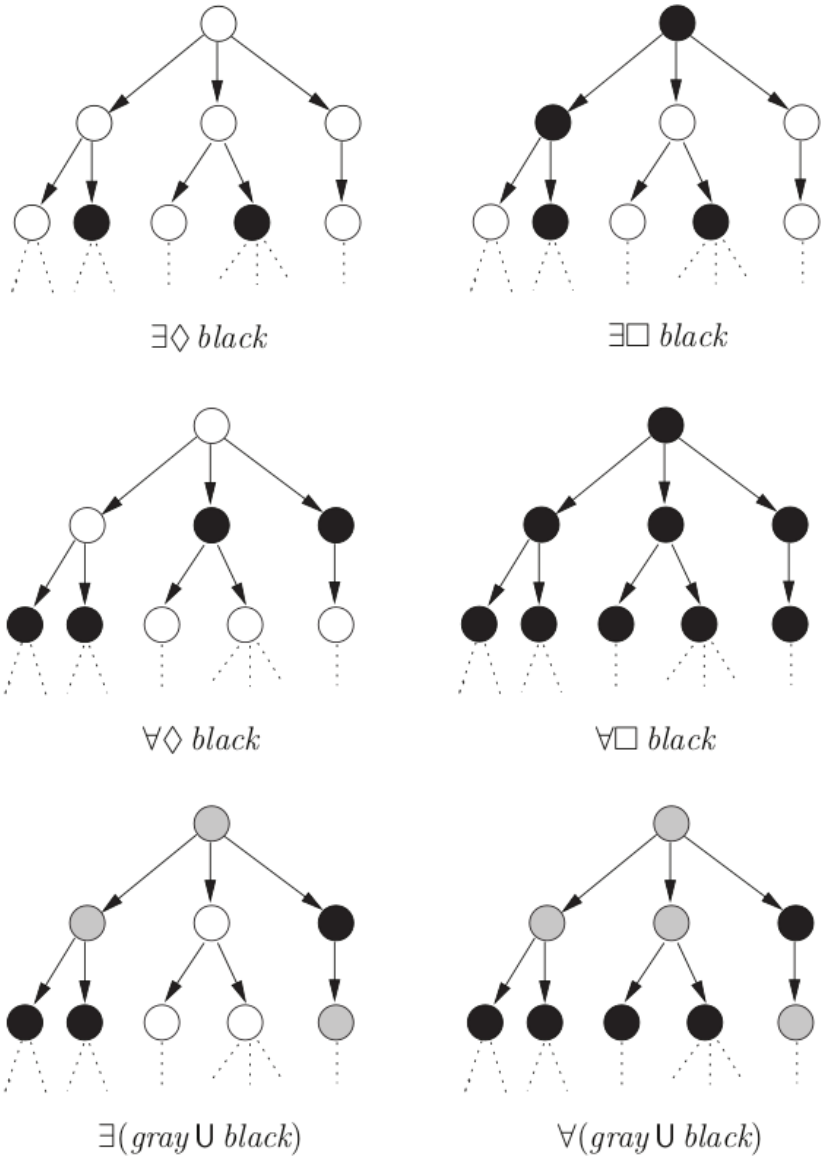
En la figura 5.1 se ilustran ejemplos de fórmulas CTL y sus ejecuciones a partir de un estado representadas en el árbol de ejecución del sistema.

5.1.3. Propiedades

En esta sección se muestran propiedades útiles para poder expresar todas las fórmulas de CTL utilizando un conjunto reducido y funcionalmente completo de conectivos.

- $\exists\Diamond\varphi = \exists(true \cup \varphi)$
- $\forall\Diamond\varphi = \forall(true \cup \varphi)$
- $\exists\Box\varphi = \neg\forall\Diamond\neg\varphi$
- $\forall\Box\varphi = \neg\exists\Diamond\neg\varphi$
- $\forall \bigcirc \varphi = \neg\exists \bigcirc \neg\varphi$

Figura 5.1: Semántica de CTL [a]



[a]Imagen tomada de [1]

- $\forall\Diamond\varphi = \neg\exists\Box\neg\varphi$

Con estas propiedades es posible expresar cualquier propiedad en CTL mediante una fórmula CTL equivalente que sólo contenga el cuantificador existencial.

5.1.4. CTL vs LTL

Tanto CTL como LTL son lenguajes que permiten expresar muchas propiedades para un sistema. Pero son lógicas incompatibles entre sí. Esto quiere decir que existen fórmulas de CTL para las cuales no hay una equivalente en LTL y viceversa.

Definición 20. *Equivalencia entre fórmulas CTL y fórmulas LTL.*
Una fórmula CTL φ y una fórmula LTL ψ son equivalentes si para todo sistema de transiciones TS:

$$TS \models \varphi \text{ si y sólo si } TS \models \psi$$

Un estado satisface una fórmula LTL φ cuando todos los caminos a partir de este estado satisfacen φ. De esto se puede observar que para obtener una fórmula LTL equivalente a una fórmula CTL dada basta con eliminar los cuantificacdores universales de la misma. Más precisamente, dada una fórmula CTL, en caso de que exista un fórmula LTL equivalente, esta se obtiene eliminando los cuantificadores tanto universales como existenciales de la misma.

5.2. Algoritmo de verificación

Para simplificar la verificación se trabajará con un conjunto reducido y funcionalmente completo de operadores. Para esto son de utilidad las propiedades del lenguaje vistas anteriormente, que nos permiten transformar cualquier fórmula CTL en una fórmula equivalente que utilice únicamente los conectivos de este conjunto.

El enfoque de verificación utilizado para CTL es distinto al que se utiliza para LTL. Su diferencia más visible es que mientras el algoritmo de verificación de LTL se basa en la construcción y análisis de autómatas, el algoritmo para CTL realiza el cálculo de satisfacibilidad de una fórmula para cada estado sin alterar el sistema de transiciones original. Este algoritmo de verificación se basa en el cálculo del conjunto de satisfacibilidad de la fórmula a verificar. Por esto también se analiza el algoritmo que permite este cálculo.

5.2.1. Cálculo del conjunto de satisfacibilidad

El conjunto de satisfacibilidad de una fórmula es el conjunto de los estados de un sistema de transiciones que satisfacen la fórmula.

Dado un sistema de transiciones $TS = (S, Act, \rightarrow, I, AP, L)$, el conjunto de satisfacibilidad se calcula como:

- $Sat(True) = S$
- $Sat(a) = \{s \in S \mid a \in L(s)\}$, con $a \in AP$
- $Sat(\varphi \wedge \psi) = Sat(\varphi) \cap Sat(\psi)$
- $Sat(\neg\varphi) = S\ Sat(\varphi)$
- $Sat(\exists \bigcirc \varphi) = \{s \in S \mid Posts(s) \cap Sat(\varphi) \neq \emptyset\}$
- $Sat(\exists(\varphi \cup \psi)) =?$
- $Sat(\exists\Box\varphi) =?$

Ahora resta definir el cálculo del conjunto de satisfacción para los operadores Until y Always.

Cálculo del conjunto de satisfacibilidad para el operador Until

Dada la siguiente ley de expansión para el operador Until

$$\exists(\varphi \cup \psi) = \psi \vee (\varphi \wedge \exists \bigcirc \exists(\varphi \cup \psi))$$

El cálculo del conjunto de satisfacibilidad se basa en la ecuación de punto fijo, a partir de la ley de expansión.

$$Sat(\exists(\varphi \cup \psi)) = Sat(\psi \vee (\varphi \wedge \exists \bigcirc \exists(\varphi \cup \psi)))$$

Aplicando la definición de la función Sat obtenemos que:

$$Sat(\exists(\varphi \cup \psi)) = Sat(\psi) \cup \{s \in Sat(\varphi) \mid Post(s) \cap Sat(\exists(\varphi \cup \psi)) \neq \emptyset\}$$

Por lo que el conjunto de satisfacibilidad de $\exists(\varphi \cup \psi)$ es el menor conjunto $T \subseteq S$ que cumple:

- $Sat(\psi) \subseteq T$
- si $s \in Sat(\psi)$ y $Post(s) \cap T \neq \emptyset$ entonces $s \in T$

Este conjunto se puede calcular mediante la siguiente recursión

- $T_0 = Sat(\psi)$
- $T_{i+1} = T_i \cup \{s \in Sat(\varphi) \mid Post(s) \cap T_i \neq \emptyset\}$

Esta sucesión converge al conjunto $Sat(\exists(\varphi \cup \psi))$.

Cálculo del conjunto de satisfacibilidad para el operador Always

Dada la siguiente ley de expansión para el operador Always

$$\exists\Box\varphi = \varphi \wedge \exists \bigcirc \exists\Box\varphi$$

El cálculo del conjunto de satisfacibilidad se basa en la ecuación de punto fijo, a partir de la ley de expansión.

$$Sat(\exists\Box\varphi) = Sat(\varphi \wedge \exists \bigcirc \exists\Box\varphi)$$

Aplicando la definición de la función Sat obtenemos que:

$$Sat(\exists\Box\varphi) = \{s \in Sat(\varphi) \mid Post(s) \cap Sat(\exists\Box\varphi) \neq \emptyset\}$$

Por lo que el conjunto de satisfacibilidad de $\exists\Box\varphi$ es el menor conjunto $T \subseteq S$ que cumple:

- $T \subseteq Sat(\varphi)$
- si $s \in T$ entonces $T \cap Post(s) \neq \emptyset$

Este conjunto se puede calcular mediante la siguiente recursión

- $T_0 = Sat(\varphi)$
- $T_{i+1} = T_i \cap \{s \in Sat(\varphi) \mid Post(s) \cap T_i \neq \emptyset\}$

Esta sucesión converge al conjunto $Sat(\exists\Box\varphi)$.

5.2.2. Algoritmo principal

El algoritmo prncipal de verificación es quién responde a la pregunta:

$$TS \models \varphi$$

Donde TS es un sistema de transiciones y φ es una fórmula CTL.

Por definición $TS \models \varphi$ si y sólo si todo estado inicial s de TS cumple $s \models \varphi$.

Por lo que una vez calculado $Sat(\varphi)$ el problema simplemente corresponde a resolver

$$I \subseteq Sat(\varphi)$$

Donde I el conjunto de estados iniciales de TS.

5.2.3. Mejora de implementación

Al calcular el conjunto se satisfacibilidad de una fórmula recursivamente es posible que se esten repitiendo cálculos ya que se aplica la recursión para cada subfórmula de forma independiente.

Ejemplo.
Sea la fórmula CTL

$$\varphi = \psi \wedge \psi$$

Entonces el cálculo del conjunto de satisfacibilidad para φ se realiza de la siguiente forma

$$Sat(\varphi) = Sat(\psi) \cap Sat(\psi)$$

En el ejemplo anterior si el conjunto de satisfacibilidad es calculado recursivamente se repetirán cálculos, ya que $Sat(\psi)$ se estará calculando dos veces. Una forma de evitar esto es implementar la solución mediante un enfoque *bottom-up*. Las recursiones vistas anteriormente se calculan de forma iterativa.

Primero se resuelven los pasos base, o sea, se calcula el conjunto de satisfacibilidad para las fórmulas atómicas. A medida que avanza el algoritmo se guardan los resultados en tablas para poder acceder a los mismo posteriormente y evitar cálculos repetidos. Luego se procede calculándolo para las fórmulas más complejas de forma tal que para el paso a resolver, sus llamadas recursivas ya fueron calculadas. Estos cálculos se encontrarán almacenados en tablas, por lo que la resolución del paso actual a partir de sus llamadas recursivas es inmediata.

Para el ejemplo visto antriormente, primero se realiza el cálculo

$$S_\psi = Sat(\psi)$$

Luego se procede a calcular

$$Sat(\varphi) = S_\psi \cap S_\psi$$

En donde S_ψ es el conjunto previamente calculado, por lo que no es necesario volver a calcularlo y simplemente se realiza la intersección.

Capítulo 6

Implementación del verificador

En este capítulo se detalla el desarrollo del verificador. Esto es, sus principales características como ser el el lenguaje de programación elegido, el lenguaje de marcas utilizado para representar grafos o librerías externas necesarias para el verificador.

También se ve la modularización del sistema implementado, que corresponde a los temas vistos en los capítulos anteriores. Se implementaron módulos para la manipulación de sistemas de transiciones y de autómatas de Büchi, con operaciones sobre los mismos y las funcionalidades necesarias para el verificador. Además se implementaron módulos para cada lógica (en este caso LTL y CTL) para poder interpretar cada una de ellas. En estos módulos se define su sintáxis, semántica y su algoritmo de verificación, de forma que para agregar otra lógica al verificador es necesario crear únicamente su módulo correspondiente.

6.1. Lenguaje de programación

El lenguaje utilizado para desarrollar el verificador fue *Python 2.7*. Esto se debe a que sus atributos se adaptaban a las necesidades del verificador.

Python es un lenguaje de alto nivel con características de lenguaje funcional, y esto ayuda a que la implementación de algoritmos requiera menos instrucciones y que a su vez estas instrucciones sean de alto nivel, generando un código simple y fácil de entender por lo que es perfectamente mantenible por otras personas.

Además es un lenguaje extremadamente flexible, lo que permite reutilizar un algoritmo implementado permitiendo aplicarlo sobre distintas estructuras sin necesidad de adaptarlo o reescribirlo. Esto es de mucha utilidad ya que se manejan estructuras similares como ser fórmulas de distintas lógicas y sus es-

tructuras quedan encapsuladas dentro de sus respectivos módulos, pudiendo aplicar el resto de los algoritmos sin necesidad de modificaciones.

Otro de los atributos importantes de este lenguaje es que soporta la programación orientada a objetos. Esto reafirma la idea de flexibilidad, modularidad y reutilización de código, por lo que fue otro de los atributos que se tuvieron en cuenta al momento de seleccionar el lenguaje de programación.

6.2. Librerías auxiliares

En la implementación del verificador se utilizaron dos librerías auxiliares. La primera de ellas es *Expat* que provee herramientas para parsear xml en objetos. La otra librería es *Python Lex-Yacc*, una implementación de *Yacc* para Python es utilizada para generar el parser correspondiente a cada lógica.

En esta sección se detallan las librerías mencionadas.

6.2.1. Expat

La librería `xml.parsers.expat` es una interfaz para el parser de XML *Expat*. Esta librería provee la clase `xmlparser`, el cual contiene entre otros los siguientes métodos

- `ParserCreate()`

  Retorna una nueva instancia de la clase `xmlparser`.

- `ParseFile(file)`

  Parsea el contenido XML desde el archivo *file*.

- `StartElementHandler(name, attributes)`

  Define la rutina que se invoca cada vez que comienza un nuevo elemento XML.

  name indica el nombre del elemento y *attributes* es un hash conteniendo los atributos del elemento con sus respectivos valores.

- `EndElementHandler(name)`

  Define la rutina a invocar cuando se cierra un elemento XML.

  name indica el nombre del elemento a cerrar.

Esta librería fue utilizada para parsear los sistemas de transiciones desde *GraphML*. Este lenguaje se describe más adelante.

6.2.2. Python Lex-Yacc

Python Lex-Yacc (PLY) es una implementación para *Python* de las herramientas *Lex* y *Yacc*.

Es un paquete implementado puramente en *Python* que permite generar parsers facilmente. Este paquete está compuesto por los módulos `lex.py` y `yacc.py`. El primero de ellos, `lex.py`, es utilizado para separar el texto a parsear en un conjunto de marcas llamados *tokens* a partir de expresiones regulares definidas para cada uno de estos *tokens*.

Una vez obtenido este conjunto de *tokens* el módulo `yacc.py` reconoce los elementos del lenguaje previamente definidos mediante una gramática libre de contexto y construye el árbol sintáctico correspondiente.

Las expresiones regulares así como las gramáticas libres de contexto utilizadas para los lenguajes de LTL y CTL se encuentran definidas en la sección 6.6.

6.3. GraphML

GraphML es un método para describir estructuras de grafos basado en el lenguaje de marcas XML. No es un lenguaje en sí mismo, sino que define elementos y atributos en XML que permiten representar varios tipos de estructuras de grafos, como ser grafos dirigidos, no dirigidos, hipergrafos entre otros.

Los elementos básicos en *GraphML* son

- `graph`

  Este elemento representa el grafo en sí, conteniendo atributos como `edgedefault` que indica si sus aristas son dirigidas o no.

- `node`

  Contiene la información de cada nodo como su identificador.

- `edge`

  Representa cada arista, indicando su origen y destino en los atributos `source` y `target` respectivamente.

Al estar definido sobre un lenguaje tan simple y utilizado como XML se convierte en un método fácilmente parseable.

Además este método es altamente flexible para agregar nuevos elementos y atributos según la necesidad de cada aplicación específicamente.

En este proyecto *GraphML* es utilizado para representar los sistemas de transiciones.

6.4. Módulos desarrollados

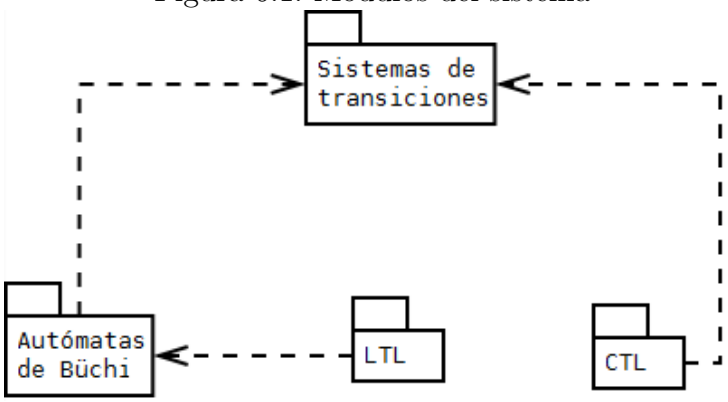
El verificador está compuesto por cuatro módulos, destacando principalmente los módulos *LTL* y *CTL* que implementan las interpretaciones de las respectivas lógicas. Esto permite agregar nuevos lenguajes para expresar propiedades, como pueden ser otros tipos de lógicas, mediante la implementación de su respectivo módulo.

En el caso de LTL se tuvo que implementar el módulo auxiliar *Autómatas de Büchi* para manipular dichos autómatas, ya que son necesarios para el algoritmo de verificación de esta lógica. Si bien este es un módulo auxiliar para la verificación de propiedades en LTL también es posible expresar propiedades directamente mediante estos autómatas para luego ser verificadas.

También se tiene un módulo destinado a los sistemas de transiciones para representar los modelos de los sistemas reactivos.

En la figura 6.1 se muestra la integración entre los distintos módulos.

Figura 6.1: Módulos del sistema



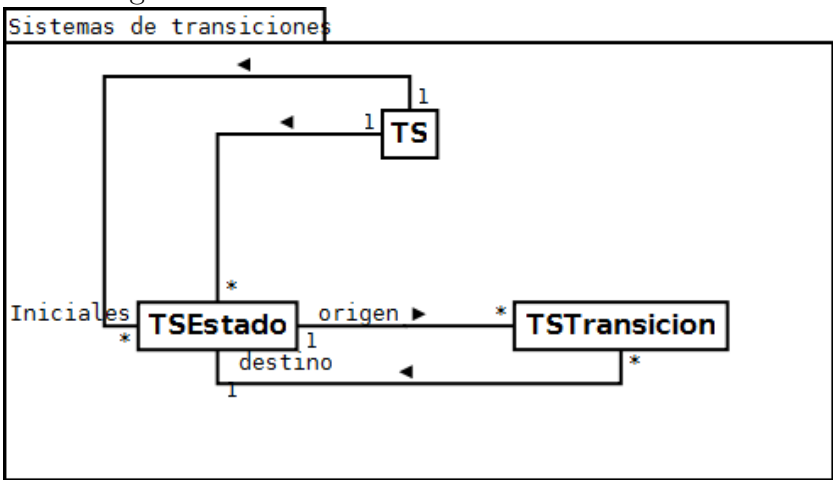
6.4.1. Sistemas de transiciones

El módulo *Sistemas de transiciones* contiene la implementación de la estructura de un sistema de transiciones. También se encuentran implementadas las operaciones básicas entre sistemas. De esta forma es posible crear sistemas de transiciones complejos a partir de sistemas más simples. Además estas operaciones permiten modelar sistemas reactivos que interactúan entre sí.

Este módulo se compone de tres clases. La clase principal del módulo es `TS`, que contiene la estructura y las operaciones necesarias para representar los sistemas de transiciones. Además se tienen las clases `TSEstado`

y `TSTransicion` para representar los estados y las transiciones del sistema respectivamente.

Figura 6.2: Módulo de Sistemas de transiciones



Operaciones entre sistemas de transiciones

Este módulo implementa dos de las operaciones principales entre sistemas de transiciones. Las operaciones implementadas fueron vistas anteriormente y son el Intercalado y el *Handshaking*. Estas operaciones se encuentran implementadas en las funciones `intercaladoTS` y `productoSynTS` de la clase `TS`.

El autómata devuelto como resultado de cada una de estas operaciones corresponde únicamente a la parte alcanzable del mismo.

6.4.2. Autómatas de Büchi

Este módulo fue implementado debido a la necesidad de tratar estos autómatas para la verificación de propiedades expresadas en lenguajes lineales, como LTL.

Como ya se vio en los capítulos anteriores el algoritmo de verificación para propiedades en LTL se basa en la construcción de un *Autómata de Büchi*. El primer paso de dicho algoritmo es traducir la propiedad a un autómata equivalente.

También es posible expresar propiedades directamente sobre estos autómatas. Se pueden expresar las ejecuciones no deseadas con un autómata de Büchi y verificar si el sistema permite alguna de estas ejecuciones. En definitiva es

esto lo que realiza el algoritmo de verificación para LTL, por lo que esto es equivalente a saltear el primer paso del mismo.

Este módulo consta de dos clases principales.

- `GNBA`

  Esta clase contiene la estructura los autómatas de Büchi. Con este fin también fue necesario implementar la clase `Nodo` para representar los nodos de estos autómatas.

- `ProductoTSNBA`

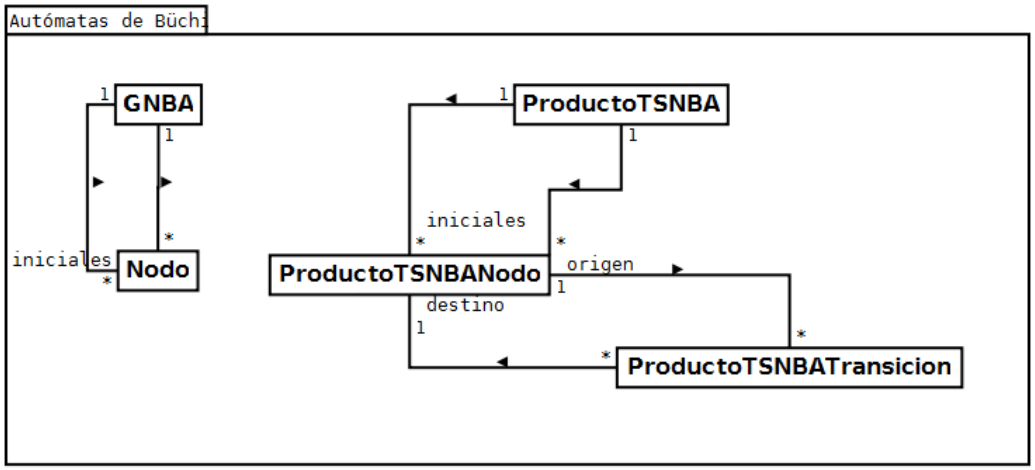
  Esta clase contiene la información del producto entre un sistema de transiciones y un autómata de Büchi.

  Este producto se implementó en una calse específica, ya que para generarse necesita un algoritmo específico. Además es la clase que implementa la parte principal del algoritmo de verificación.

  Esta clase también puede ser parte del módulo *LTL* pero finalmente se optó por mantenerla en este módulo ya que como se mencionó anteriormente las propiedades pueden ser expresadas directamente mediante autómatas y el algoritmo de verificación es el mismo.

  También se implementaron las clases `ProductoTSNBANodo` y `ProductoTSNBATransicion` para representar los nodos y transiciones del producto respectivamente.

Figura 6.3: Módulo de Autómatas de Büchi



Construcción del producto entre un sistema de transiciones y un autómata de Büchi

Este producto genera como resultado un nuevo autómata, el cuál está representado por la clase `ProductoTSNBA`.

Esta clase implementa el algoritmo de contrucción de este producto mediante la función `ProductoTSNBA(ts, nba)`, donde:

- `ts` es un sistema de transiciones.
- `nba` es un autómata de Büchi.

Esta función devuelve la parte alcanzable del producto.

Verificación de propiedadades

Una vez obtenido el producto mencionado en la sección anterior el siguiente paso es verificar si existen palabras reconocidas por este autómata.

Como se vio en capítulos anteriores, como el autómata reconoce las palabras que no cumplen la propiedad deseada, este proceso es equivalente encontrar las ejecuciones del sistema que no cumplen la propiedad. En caso de existir al menos una, esta es un contrajemplo que muestra que el sistema no cumple dicha propiedad.

Este proceso se encuentra implementado en la función `verificar()` de la clase `ProductoTSNBA`. Esta función se basa en el recorrido DFS para buscar un ciclo en el autómata. Este ciclo debe pasar por al menos un estado de cada conjunto de aceptación del autómata, como se vio en el capítulo correspondiente a Autómatas de Büchi.

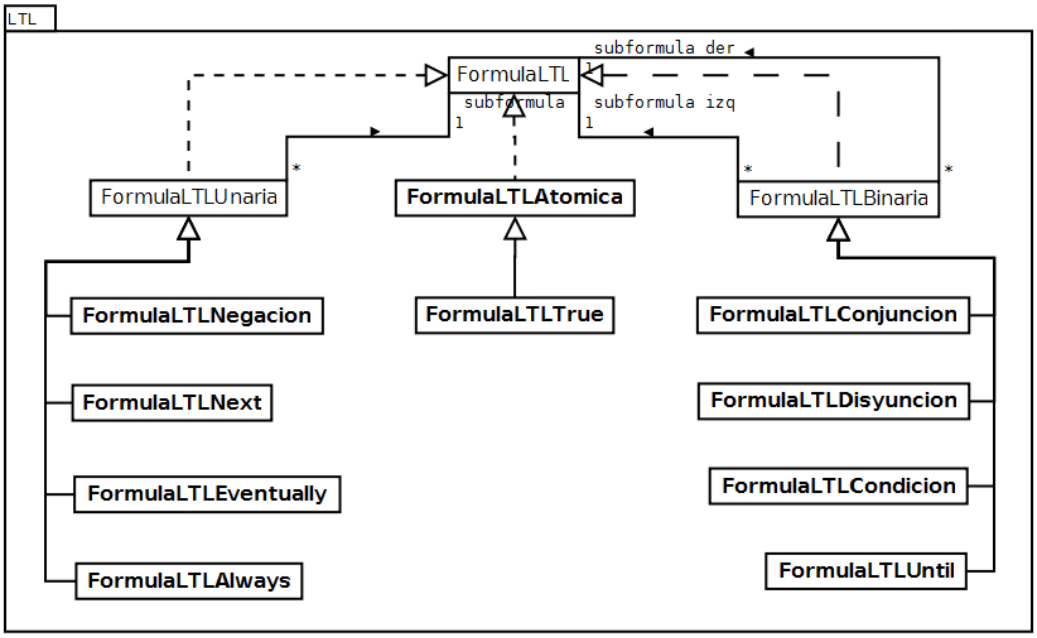
6.4.3. LTL

El módulo *LTL* está comprendido por las clases necesarias para implementar dicha lógica. En la figura 6.4 se muestran las relaciones entre la clase `FormulaLTL` y todas sus especializaciones que implementan la sintaxis y semántica de LTL.

Es importante destacar que este módulo no implementa el algoritmo de verificación para LTL. El algoritmo es implementado en su mayor parte por el módulo *Autómatas de Büchi*. El módulo *LTL* simplemente traduce la fórmula a verificar en un autómata equivalente y traslada la responsabilidad de la verificación al módulo correspondiente.

La clase que inicia el proceso de verificación es `LTLMC`. El único objetivo de esta clase es implementar la función `verificar(ts, formula)`, donde:

Figura 6.4: Módulo de LTL



- `ts` es un sistema de transiciones.
- `formula` es una fórmula LTL.

Esta función es la encargada de iniciar el proceso de verificación traduciendo la fórmula LTL al autómata correspondiente, delegar la verificación al módulo de *Autómatas de Büchi* y luego desplegar el resultado de la misma.

6.4.4. CTL

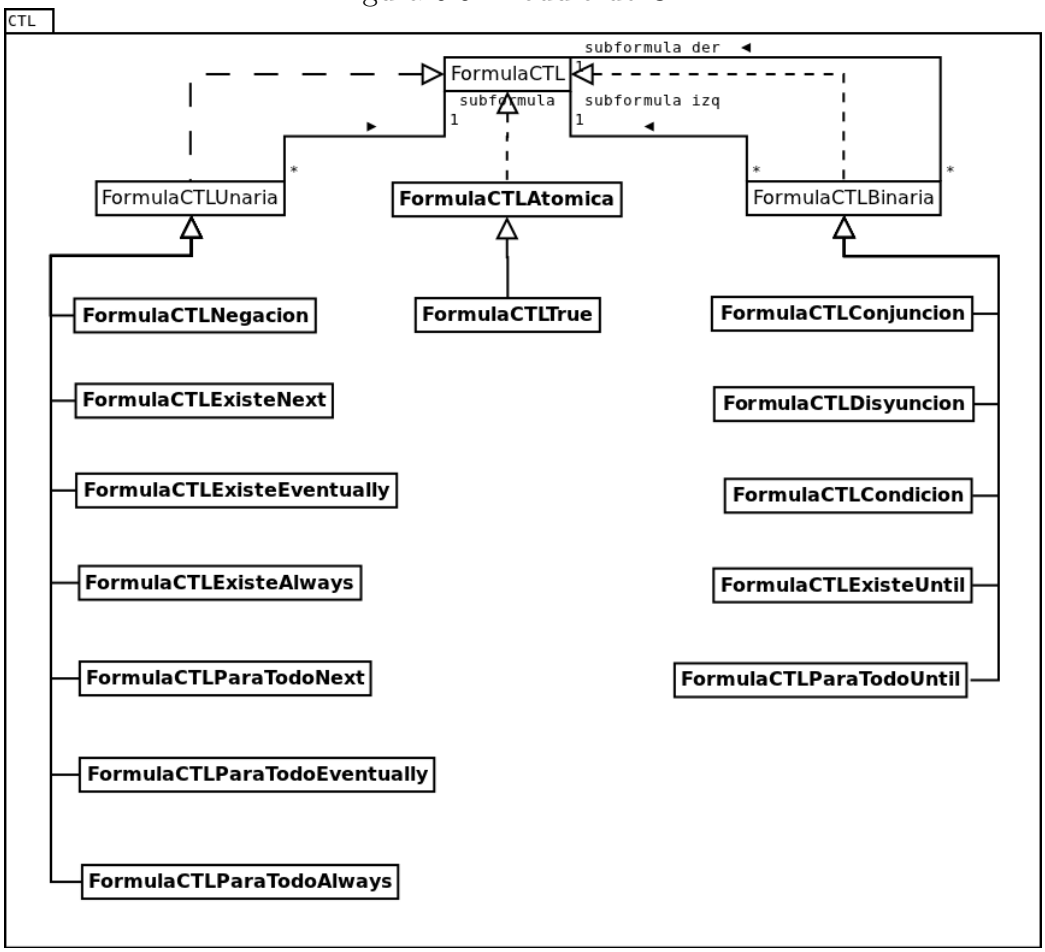
Este módulo al igual que el módulo anterior implementa la sintaxis y semántica de CTL. Pero a diferencia del módulo *LTL* este implementa el algoritmo de verificación. Esto se debe a que el algoritmo para este tipo de lógica no requiere la implementación de módulos ni estructuras auxiliares. Este algoritmo se basa en el cálculo del conjunto de satisfacibilidad de una fórmula y sus subfórmulas. Se encuentra implementado en la función `verificar(ts, formula)`, donde:

- `ts` es un sistema de transiciones.
- `formula` es una fórmula CTL.

Esta función verifica que el conjunto de los estados iniciales de `ts` esté incluído en el conjunto de satisfacibilidad de `formula`, tal como se vio en el capítulo de CTL. Además esta función se encuentra en la clase `CTLMC`, cuyo único objetivo es alojar la misma.

Por otro lado el cálculo del conjunto de satisfacibilidad se encuentra definido en la función `getSat(ts)` de la clase `FormulaCTL`, donde `ts` es un sistema de transiciones.

Figura 6.5: Módulo de CTL



6.5. GraphML para sistemas de transiciones

Como se mencionó anteriormente se utilizó el formato de archivo *GraphML* para representar los sistemas de transiciones. Los estados son representados por el elemento `node` mientras que las transiciones son representadas por el elemento `edge`.

El formato *GraphML* no establece como representar las etiquetas tanto en los nodos como en las aristas. Para esto existe el atributo `data`, que proporciona flexibilidad para agregar atributos no contemplados por el formato. Esto tiene un desventaja, y es que los atributos no contemplados por el formato no se representan de forma estándar, y por lo tanto su representación depende del editor utilizado. En este caso el editor utilizado es *yEd*.

Para los estados se guarda la siguiente información:

- Identificador

  Este valor se guarda en el atributo `id` de cada nodo. Es el identificador del estado, por lo que debe ser único.

  Cuando se genera un sistema de transiciones mediante el verificador este genera los identificadores de cada estado automáticamente.

- Proposiciones

  Representan el conjunto de las proposiciones válidas en cada estado.

  Se guardan en el atributo `y:NodeLabel`. Este atributo es se encuentra dentro del atributo `data`, ya que no se encuentra especificado en el formato.

  En caso de haber varias proposiciones, estas deben estar separadas por comas.

Además de esta información se debe indicar cuales son los estados iniciales.

Para las transiciones se debe guardar la siguiente información:

- Origen

  Representa el estado de origen de la transición. Se guarda en el atributo `source`.

- Destino

  Representa el estado de destino de la transición. Se guarda en el atributo `target`.

- Acción

  Representa la acción que corresponde al cambio de estado.

  Se guardan en el atributo `y:EdgeLabel`.

  Este atributo es se encuentra dentro del atributo `data`, ya que no se encuentra especificado en el formato.

El parser de *GraphML* a sistemas de transiciones se encuentra implementado en el objeto `ParserGraphML` del paquete *Sistemas de transiciones*.

Como se mencionó anteriormente hay atributos que no están especificados en el formato y por lo tanto su interpretación depende del editor utilizado. Estos atributos son especificados en el parser mediante sub clases. En este caso se utiliza el *yEd Graph Editor*, para el cual fue implementado el objeto `ParserGraphML_yEd`. Este objeto es un parser de *GraphML* que además interpreta la información dentro del atributo `data` como las etiquetas de los estados y transiciones.

A continuación se muestra un ejemplo de un sistema de transiciones y su correspondiente representación en *GraphML*.

Ejemplo de sistema de transiciones en *GraphML*

6.6. Analizador sintáctico de fórmulas

Para el análisis sintáctico de fórmulas se utilizó la herramienta *Python Lex-Yacc*.

Para reconocer los elementos de un lenguaje, esta herramienta requiere la definición de sus elementos mediante expresiones regulares. Luego se debe definir la sintaxis del lenguaje mediante una gramática libre de contexto.

A continuación se detallan los elementos y las gramáticas definidas para los lenguajes LTL y CTL.

6.6.1. LTL

En la siguiente tabla se muestran los operadores de LTL con su correspondiente cadena de caracteres.

Además se definió la siguiente expresión regular para las proposiciones atómicas.

$$[a - z\_][a - z0 - 9\_]*$$

Operador	Cadena de caracteres
$\neg\bot$	TRUE
$\neg$	-
$\wedge$	/\
$\vee$	\/
$\rightarrow$	->
$\bigcirc$	O
$\Box$	[]
$\Diamond$	<>
$\cup$	U

Cuadro 6.1: Cadena de caracteres correspondiente a cada operador LTL.

Una vez definidos los elementos del lenguaje se definió la sintaxis mediante la siguiente gramática.

formula ::=	formula $\cup$ formula
	| formula $\wedge$ formula
	| formula $\vee$ formula
	| formula $\rightarrow$ formula
	| $\bigcirc$ formula
	| $\Box$ formula
	| $\Diamond$ formula
	| $\neg$ formula
	| proposicion
	| TRUE

Esta información se encuentra implementada en el archivo `parserLTL.py` del módulo *LTL*.

6.6.2. CTL

En la siguiente tabla se muestran los operadores de CTL con su correspondiente cadena de caracteres.

Al igual que en LTL se utilizó la siguiente expresión regular para las proposiciones atómicas.

$$[a - z_] [a - z0 - 9_]*$$

La gramática definida para CTL es la siguiente.

Operador	Cadena de caracteres
$\neg\bot$	TRUE
$\neg$	-
$\wedge$	/\
$\vee$	\/
$\rightarrow$	->
$\exists\bigcirc$	EO
$\forall\bigcirc$	AO
$\exists\Box$	E[]
$\forall\Box$	A[]
$\exists\Diamond$	E<>
$\forall\Diamond$	A<>
$\exists\cup$	EU
$\forall\cup$	AU

Cuadro 6.2: Cadena de caracteres correspondiente a cada operador CTL.

formula ::=	formula $\exists\cup$ formula
	| formula $\forall\cup$ formula
	| formula $\wedge$ formula
	| formula $\vee$ formula
	| formula $\rightarrow$ formula
	| $\exists\bigcirc$ formula
	| $\forall\bigcirc$ formula
	| $\exists\Box$ formula
	| $\forall\Box$ formula
	| $\exists\Diamond$ formula
	| $\forall\Diamond$ formula
	| $\neg$ formula
	| proposicion
	| TRUE

Esta información se encuentra implementada en el archivo `parserCTL.py` del módulo *CTL*.

6.7. Casos de estudio

En esta sección se muestra la utilización del verificador en dos casos de estudios distintos. En estos casos de estudio se verifican propiedades expre-

sadas en LTL y en CTL para distintos sistemas modelados por sistemas de transiciones.

6.7.1. Mutua exclusión con árbitro

A continuación se muestra el ejemplo de mutua exclusión con árbitro, visto en capítulos anteriores. En la figura 2.5 se muestra el sistemas de transiciones correspondiente. El archivo en formato *GraphML* para este caso de estudio se encuentra en la sección B.1 ubicada en los apéndices.

Se desea verificar que los sistemas en cuestión cumplen con las siguientes propiedades en CTL:

- Los procesos nunca acceden simultaneamente en la sección crítica

  $$\forall\Box(\neg \text{crit}_1 \vee \neg \text{crit}_2)$$

  El resultado de la verificación de esta propiedad es:

  ```
  El sistema cumple la propiedad.
  ```

- Cada proceso accede infinitas veces a la sección crítica

  $$(\forall\Box\forall\Diamond \text{crit}_1) \wedge (\forall\Box\forall\Diamond \text{crit}_2)$$

  El resultado de la verificación de esta propiedad es:

  ```
  El sistema NO cumple la propiedad.
  Contraejemplo:
  ['noncrit_1', 'noncrit_2', 'unlock']
  ```

6.7.2. Ascensor

A continuación se muestra el ejemplo del ascensor, también visto en capítulos anteriores. En la figura 2.6 se muestra el sistemas de transiciones correspondiente. El archivo en formato *GraphML* para este caso de estudio se encuentra en la sección B.2 ubicada en los apéndices.

Se desea verificar que los sistemas en cuestión cumplen con la siguiente propiedad en LTL:

- Si el ascensor está subiendo o bajando entonces ambas puertas están cerradas

  $$\Box((\text{subiendo} \vee \text{bajando}) \rightarrow (\text{arriba.cerrada} \wedge \text{abajo.cerrada}))$$

El resultado de la verificación de esta propiedad es:

```
El sistema cumple la propiedad.
```

6.8. Ejemplo de extensión

Como ya se mencionó anteriormente, uno de los principales objetivos de este verificador es la posibilidad de extensión para permitir verificar otro tipo de propiedades sobre sistemas reactivos.

En esta sección se muestra un ejemplo de extensión para verificar propiedades de tiempo real. Para esto se introducen básicamente los principales conceptos y se describen las modificaciones necesarias para verificar este tipo de propiedades. Esto incluye implementar un lenguaje que permita expresar las propiedades deseadas, pero además se necesita modificar la estructura de los sistemas de transiciones para modelar el tiempo y así poder expresar propiedades teniendo en cuenta el mismo.

6.8.1. Sistemas de transiciones con tiempo

En capítulos anteriores se trabajó con Sistemas de transiciones, pero al momento de verificar propiedades de tiempo real estos no permiten modelar el tiempo. Para poder hacerlo se puede utilizar otro tipo de sistemas de transiciones. Los sistemas de transiciones con tiempo [12] agregan un conjunto de relojes junto con un conjunto de operaciones de comparación y una operación de reinicio.

Definición 21. *Sistema de transiciones con tiempo.*
Un sistema de transiciones con tiempo consiste en una tupla $TS = (S, I, Act, X, C, \rightarrow)$*, donde:*

- S *es el conjunto de estados*
- $I \subseteq S$ *es el conjunto de estados iniciales*
- Act *es el conjunto de acciones*
- X *es el conjunto de relojes*
- C *es el conjunto comparaciones sobre los relojes*
- $\rightarrow \subseteq S \times 2^X \times Act \times S$ *es la relación de transición*

La definición anterior introduce el conjunto X como el conjunto de las variables que modelan los relojes. También introduce el conjunto $C : S \to \Phi(X)$ de comparaciones sobre los relojes. Donde

$$\Phi(X) = x \leq c \mid x \geq c \mid x < c \mid x > c \mid \Phi_1(X) \wedge \Phi_2(X)$$

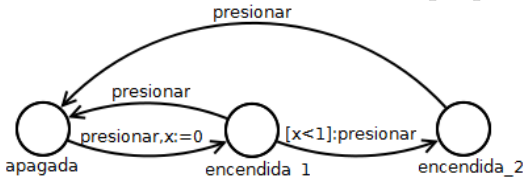
con $c \in \mathbb{R}$.

Ejemplo. Sistema de transiciones con tiempo para una lámpara
Se considera el sistema de transiciones de la figura 6.6 para modelar una lámpara. Donde:

- S = {apagada, encendida$_1$, encendida$_2$}
- I = {apagada}
- Act = {presionar}
- X = {x}
- C = {$x < 1$}
- $\to$ = {(apagada, presionar, encendida$_1$), (encendida$_1$, presionar, encendida$_2$), (encendida$_1$, presionar, apagada), (encendida$_2$, presionar, apagada)}

Esta lámpara tiene dos niveles de iluminación, 1 y 2. Para encender la lámpara en el modo 2 se debe presionar el botón de encendido dos veces en menos de un segundo.

Figura 6.6: Sistema de transiciones con tiempo para una lámpara



6.8.2. Lenguaje de tiempo

Los lenguajes vistos en capítulos anteriores no permiten expresar propiedades de tiempo real, para esto se necesita un lenguaje que incluya variables de tiempo. Una posibilidad es Lógica Temporal Proposicional de Tiempo (TPTL) [13]. Este lenguaje es una extensión de LTL que permite agregar

comparaciones sobre los relojes, las mismas que utilizan los sistemas de transiciones con tiempo. Las fórmulas de TPTL son construídas según la siguiente gramática

$$\varphi ::= true \mid a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \bigcirc\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi \cup \varphi \mid \Phi(X)$$

Donde

$$\Phi(X) = x \leq c \mid x \geq c \mid x < c \mid x > c \mid \Phi_1(X) \wedge \Phi_2(X)$$

con $c \in \mathbb{R}$.

Ejemplo. Propiedades de tiempo para una lámpara
A continuación se muestran posibles propiedades en TPTL sobre una lámpara.

- Si la lámpara está apagada y se presiona dos veces el botón de encendido en un tiempo mayor a un segundo, la lámpara se apaga

$$\neg((\text{apagada} \wedge \bigcirc \bigcirc (x \geq 1)) \wedge (\neg \bigcirc \bigcirc \text{apagada}))$$

- Si la lámpara está apagada y se presiona dos veces el botón de encendido en menos de un segundo, la lámpara se enciende en modo de iluminación 2

$$\neg((\text{apagada} \wedge \bigcirc \bigcirc (x < 1)) \wedge (\neg \bigcirc \bigcirc \text{encendida}_2))$$

6.8.3. Modificaciones en la implementación

Para implementar los sistemas de transiciones con tiempo se debe agregar una subclase de `TS` con las operaciones necesarias para manipular los relojes. Ademas se debe implementar una subclase de `TSTransicion` que incluya las guardas y los reinicios de relojes.

Por otro lado para agregar un lenguaje de propiedades es necesario implementar un nuevo módulo, incluyendo el parser con la definición de la gramática para este lenguaje. Este módulo además debe tener una función `verificar()` que implemente el algoritmo de verificación para este lenguaje[1].

[1]El algoritmo de verificación no se introcuce en este documento.

Capítulo 7

Conclusiones

Se implementó un verificador de modelos para sistemas reactivos. El sistema a verificar se representa mediante un grafo en *GraphML* y las propiedades se expresan en distintos lenguajes de especificación.

Se implementaron módulos para los lenguajes de especificación *LTL* y *CTL*. La implementación de cada uno de estos módulos contiene tanto la sintaxis como los algoritmos de verificación de los respectivos lenguajes. De esta forma podemos agregar nuevos lenguajes sin afectar el resto de los módulos. En particular en la sección 6.8 mostramos cómo extender el verificador para soportar *TPTL* y eventualmente nuevos lenguajes de especificación.

Esta implementación se basa en el paradigma de orientación a objetos. Esto es un punto a favor ya que facilita la comprensión, la mantenibilidad y la posterior extensión del verificador.

Otros puntos a favor son la modularización, flexibilidad del código y la facilidad de uso. Estos aspectos son importantes pensando en estudiantes de cursos de grado ya que permite visualizar problemas sencillos, ver en funcionamiento los algoritmos de verificación y analizar posibles mejoras en la implementación. Esto contrasta con herramientas similares como *Spin* o *UPPAL* ya que estas se enfocan en otros aspectos, principalmente en el rendimiento. Además por la misma razón, para garantizar un buen rendimiento en la verificación de propiedades, tienen ciertas limitaciones como pueden ser la profundidad de las fórmulas que expresan las propiedades a verificar.

Por último se muestra un ejemplo de utilización del verificador. Esto comprende el modelado de un sistema, la especificación de propiedades sobre el mismo en distintos lenguajes y la verificación de estas propiedades.

7.1. Trabajo futuro

El principal trabajo futuro sería la extensión del verificador agregando nuevos lenguajes de especificación de propiedades. Esta posibilidad es el principal objetivo del proyecto. Para esto se necesita agregar un nuevo módulo que implemente la sintaxis de este lenguaje además de su algoritmo de verificación. Esto se ilustrada con un ejemplo en la sección 6.8.

Otro trabajo futuro es el análisis de rendimiento. Este aspecto no es uno de los objetivos en este proyecto, por lo que no fue probado. Si bien con las pruebas realizadas se comportó adecuadamente, no se utilizaron grandes sistemas en cuanto a cantidad de estados ni propiedades complejas. Por esto el trabajo pendiente no es mejorar, sino analizar su comportamiento en estos escenarios más complejos.

Bibliografía

[1] *Principles of Model Checking.*
Christel Baier y Joost-Pieter Katoen.
The MIT Press, 2008.

[2] *Software Reliability Methods.*
Doron A. Peled.
Springer, 2001.

[3] *Handbook of Automated Reasoning. Chapter 21: Model Checking.*
Edmund M. Clarke y Bernd-Holger Schlingloff.
Elsevier Science Publishers B. V. Amsterdam, The Netherlands, 2001.

[4] Sitio web de *Python 2.7*:
`http://docs.python.org/2/`
Último acceso: 31/10/2013.

[5] Sitio web de librería *Expat para Python*:
`http://docs.python.org/2/library/pyexpat.html`
Último acceso: 31/10/2013.

[6] Sitio web de *Python Lex-Yacc*:
`http://www.dabeaz.com/ply/`
Último acceso: 31/10/2013.

[7] Sitio web de *GraphML File Format*:
`http://graphml.graphdrawing.org/`
Último acceso: 31/10/2013.

[8] *The synthesis of sequential switching circuits.*
D. A. Huffman.
Journal of the Franklin Institute, 1954.

[9] *On a decision method in restricted second order arithmetic.*
J. R. Buchi.

International Congress on Logic, Methodology and Philosophy of Science.
Stanford University Press, 1962.

[10] *The temporal logic of programs.*
A. Pnueli.
18th IEEE Symposium on Foundations of Computer Science (FOCS).
IEEE Computer Society Press, 1977.

[11] *Design and synthesis of synchronization skeletons using branching time temporal logic.*
E. M. Clarke y E. A. Emerson.
Logic of Programs, volume 131 of Lecture Notes in Computer Science.
Springer-Verlag, 1981.

[12] *Temporal proof methodologies for real-time systems.*
T. A. Henzinger, Z. Manna y A. Pnueli.
18th Annual Symposium on Principles of Programming Languages.
ACM Press, 1991.

[13] *A really temporal logic.*
R. Alur y Th.A. Henzinger.
30th Annual Symposium on Foundations of Computer Science (FOCS'89).
IEEE Computer Society Press, 1989.

Apéndices

Apéndice A

Manual de usuario

A.1. Requerimientos del sistema

Para ejecutar el verificador es necesario tener instalado *Python* versión 2.7.

A.2. Operaciones entre sistemas de transiciones

Este verificador también implementa operaciones entre sistemas de transiciones, de forma de poder trabajar con sistemas de transiciones complejos generados a partir de otros más simples. Las operaciones implementadas son el Intercalado y el *Handshaking*. A continuación se muestra la ejecución de cada una de ellas.

A.2.1. Intercalado

Para realizar el intercalado de dos sistemas de transiciones se debe ejecutar el siguiente comando:

```
> python MC.py ic <TS1.graphml> <TS2.graphml> <intercalado.graphml>
```

Donde `TS1.graphml` y `TS2.graphml` son archivos previamente generados en los cuales se representan sistemas de transiciones en formato *GraphML*, e `intercalado.graphml` indica el archivo en el cual se escribirá el sistema de transiciones resultado del intercalado.

A.2.2. *Handshaking*

Para realizar el *Handshaking* de dos sistemas de transiciones se debe ejecutar el siguiente comando:

```
> python MC.py hs <TS1.graphml> <TS2.graphml> <handshaking.graphml>
```

Donde `TS1.graphml` y `TS2.graphml` son archivos previamente generados en los cuales se representan sistemas de transiciones en formato *GraphML*, y `handshaking.graphml` indica el archivo en el cual se escribirá el sistema de transiciones resultado del *Handshaking*.

A.3. Verificación de propiedades

Una vez que se tiene el sistema de transiciones en formato *GraphML* se puede comenzar a verificar propiedades sobre dicho sistema. Para esto de debe tener en cuenta que hay dos lenguajes implementados para representar propiedades, estos son LTL y CTL. A continuación se muestra la ejecución del verificador para cada uno de estos lenguajes.

A.3.1. Propiedades en LTL

Para verificar las propiedades en LTL se debe ejecutar el verificador de la siguiente forma:

```
> python MC.py ltl <archivo.graphml> <formula LTL>
```

Donde `archivo.graphml` es el archivo donde se encuentra representado el sistema de transiciones y `formula LTL` es la fórmula LTL expresada según la sección 6.6.1.

A.3.2. Propiedades en CTL

Para verificar las propiedades en CTL se debe ejecutar el verificador de la siguiente forma:

```
> python MC.py ctl <archivo.graphml> <formula CTL>
```

Donde `archivo.graphml` es el archivo donde se encuentra representado el sistema de transiciones y `formula CTL` es la fórmula CTL expresada según la sección 6.6.2.

A.3.3. Resultados de la ejecución

Para la verificación de una propiedad existen dos posibles resultados, estas son que el sistema cumpla la propiedad, o que no la cumpla. En caso de que el sistema cumpla la propiedad se muestra el siguiente mensaje:

```
El sistema cumple la propiedad.
```

Mientras que si el sistema no cumple la propiedad se muestra el mensaje:

```
El sistema NO cumple la propiedad.
```

Testigo o contrajemplo

Además de indicar si el sistema cumple o no la propiedad, el verificador proporciona en caso de ser necesario una ejecución testigo de la respuesta dada.

Si se está verificando una propiedad LTL se proporciona una contraejemplo en caso de que el sistema no cumpla la propiedad.

Para el caso de CTL si se está verificando una propiedad cuyo cuantificador es universal se proporciona un contraejemplo en caso de que la propiedad no se cumpla, mientras que si el cuantificador de la propiedad es existencial se proporciona un testigo en caso de que la misma se cumpla.

Apéndice B

Archivos *GraphML*

B.1. Archivo *GraphML* para sistema de transiciones de mutua exclusión con árbitro

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:y="http://www.yworks.com/xml/graphml"
 xmlns:yed="http://www.yworks.com/xml/yed/3"
 xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
 http://www.yworks.com/xml/schema/graphml/1.1/ygraphml.xsd">
  <graph edgedefault="directed" id="G">
    <node id="n0">
      <data>
        <y:ShapeNode>
          <y:NodeLabel alignment="center" autoSizePolicy="content"
 fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
 hasLineColor="false" modelName="internal" visible="true">
 INI_noncrit_1,noncrit_2,unlock</y:NodeLabel>
        </y:ShapeNode>
      </data>
    </node>
    <node id="n1">
      <data>
        <y:ShapeNode>
          <y:NodeLabel alignment="center" autoSizePolicy="content"
 fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
 hasLineColor="false" modelName="internal" visible="true">
```

```
crit_1,noncrit_2,lock</y:NodeLabel>
       </y:ShapeNode>
     </data>
   </node>
   <node id="n2">
     <data>
       <y:ShapeNode>
         <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="fals
hasLineColor="false" modelName="internal" visible="true">
noncrit_1,crit_2,lock</y:NodeLabel>
         <y:Shape type="rectangle"/>
       </y:ShapeNode>
     </data>
   </node>
   <edge id="e0" source="n0" target="n1">
     <data>
       <y:PolyLineEdge>
 <y:EdgeLabel>request</y:EdgeLabel>
       </y:PolyLineEdge>
     </data>
   </edge>
   <edge id="e1" source="n0" target="n2">
     <data>
       <y:PolyLineEdge>
 <y:EdgeLabel>request</y:EdgeLabel>
       </y:PolyLineEdge>
     </data>
   </edge>
   <edge id="e2" source="n1" target="n0">
     <data>
       <y:PolyLineEdge>
 <y:EdgeLabel>release</y:EdgeLabel>
       </y:PolyLineEdge>
     </data>
   </edge>
   <edge id="e3" source="n2" target="n0">
     <data>
       <y:PolyLineEdge>
 <y:EdgeLabel>release</y:EdgeLabel>
       </y:PolyLineEdge>
```

```
      </data>
    </edge>
  </graph>
</graphml>
```

B.2. Archivo *GraphML* para sistema de transiciones de ascensor con puertas

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:y="http://www.yworks.com/xml/graphml"
xmlns:yed="http://www.yworks.com/xml/yed/3"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://www.yworks.com/xml/schema/graphml/1.1/ygraphml.xsd">
  <graph edgedefault="directed" id="G">
    <node id="n0">
      <data>
        <y:ShapeNode>
          <y:Fill color="#FFCC00" transparent="false"/>
          <y:BorderStyle color="#000000" type="line" width="1.0"/>
          <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
hasLineColor="false" modelName="internal" visible="true">
INI_abajo,arriba_bloqueada,arriba_cerrada,abajo_cerrada</y:NodeLabel>
          <y:Shape type="rectangle"/>
        </y:ShapeNode>
      </data>
    </node>
    <node id="n1">
      <data>
        <y:ShapeNode>
          <y:Fill color="#FFCC00" transparent="false"/>
          <y:BorderStyle color="#000000" type="line" width="1.0"/>
          <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
hasLineColor="false" modelName="internal" visible="true">
subiendo,arriba_bloqueada,arriba_cerrada,abajo_bloqueada,abajo_cerrada</y:NodeLabel>
          <y:Shape type="rectangle"/>
```

```
        </y:ShapeNode>
      </data>
    </node>
    <node id="n2">
      <data>
        <y:ShapeNode>
          <y:Fill color="#FFCC00" transparent="false"/>
          <y:BorderStyle color="#000000" type="line" width="1.0"/>
          <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false
hasLineColor="false" modelName="internal" visible="true">
abajo,arriba_bloqueada,arriba_cerrada,abajo_abierta</y:NodeLabel>
          <y:Shape type="rectangle"/>
        </y:ShapeNode>
      </data>
    </node>
    <node id="n3">
      <data>
        <y:ShapeNode>
          <y:Fill color="#FFCC00" transparent="false"/>
          <y:BorderStyle color="#000000" type="line" width="1.0"/>
          <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false
hasLineColor="false" modelName="internal" visible="true">
arriba,arriba_cerrada,abajo_bloqueada,abajo_cerrada</y:NodeLabel>
          <y:Shape type="rectangle"/>
        </y:ShapeNode>
      </data>
    </node>
    <node id="n4">
      <data>
        <y:ShapeNode>
          <y:Fill color="#FFCC00" transparent="false"/>
          <y:BorderStyle color="#000000" type="line" width="1.0"/>
          <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false
hasLineColor="false" modelName="internal" visible="true">
bajando,arriba_bloqueada,arriba_cerrada,abajo_bloqueada,abajo_cerrada</y:Node
          <y:Shape type="rectangle"/>
        </y:ShapeNode>
      </data>
```

```
    </node>
    <node id="n5">
      <data>
        <y:ShapeNode>
          <y:Fill color="#FFCC00" transparent="false"/>
          <y:BorderStyle color="#000000" type="line" width="1.0"/>
          <y:NodeLabel alignment="center" autoSizePolicy="content"
fontFamily="Dialog" fontSize="12" fontStyle="plain" hasBackgroundColor="false"
hasLineColor="false" modelName="internal" visible="true">
arriba,arriba_abierta,abajo_bloqueada,abajo_cerrada</y:NodeLabel>
          <y:Shape type="rectangle"/>
        </y:ShapeNode>
      </data>
    </node>
    <edge id="e0" source="n0" target="n1">
      <data>
        <y:PolyLineEdge>
          <y:LineStyle color="#000000" type="line" width="1.0"/>
          <y:Arrows source="none" target="standard"/>
          <y:EdgeLabel>subir</y:EdgeLabel>
        </y:PolyLineEdge>
      </data>
    </edge>
    <edge id="e1" source="n0" target="n2">
      <data>
        <y:PolyLineEdge>
          <y:LineStyle color="#000000" type="line" width="1.0"/>
          <y:Arrows source="none" target="standard"/>
          <y:EdgeLabel></y:EdgeLabel>
        </y:PolyLineEdge>
      </data>
    </edge>
    <edge id="e2" source="n1" target="n3">
      <data>
        <y:PolyLineEdge>
          <y:LineStyle color="#000000" type="line" width="1.0"/>
          <y:Arrows source="none" target="standard"/>
          <y:EdgeLabel>llego_arriba</y:EdgeLabel>
        </y:PolyLineEdge>
      </data>
    </edge>
```

```
<edge id="e3" source="n2" target="n1">
  <data>
    <y:PolyLineEdge>
      <y:LineStyle color="#000000" type="line" width="1.0"/>
      <y:Arrows source="none" target="standard"/>
      <y:EdgeLabel>subir</y:EdgeLabel>
    </y:PolyLineEdge>
  </data>
</edge>
<edge id="e4" source="n2" target="n0">
  <data>
    <y:PolyLineEdge>
      <y:LineStyle color="#000000" type="line" width="1.0"/>
      <y:Arrows source="none" target="standard"/>
      <y:EdgeLabel></y:EdgeLabel>
    </y:PolyLineEdge>
  </data>
</edge>
<edge id="e5" source="n3" target="n4">
  <data>
    <y:PolyLineEdge>
      <y:LineStyle color="#000000" type="line" width="1.0"/>
      <y:Arrows source="none" target="standard"/>
      <y:EdgeLabel>bajar</y:EdgeLabel>
    </y:PolyLineEdge>
  </data>
</edge>
<edge id="e6" source="n3" target="n5">
  <data>
    <y:PolyLineEdge>
      <y:LineStyle color="#000000" type="line" width="1.0"/>
      <y:Arrows source="none" target="standard"/>
      <y:EdgeLabel></y:EdgeLabel>
    </y:PolyLineEdge>
  </data>
</edge>
<edge id="e7" source="n4" target="n0">
  <data>
    <y:PolyLineEdge>
      <y:LineStyle color="#000000" type="line" width="1.0"/>
      <y:Arrows source="none" target="standard"/>
```

```
          <y:EdgeLabel>llego_abajo</y:EdgeLabel>
        </y:PolyLineEdge>
      </data>
    </edge>
    <edge id="e8" source="n5" target="n4">
      <data>
        <y:PolyLineEdge>
          <y:LineStyle color="#000000" type="line" width="1.0"/>
          <y:Arrows source="none" target="standard"/>
          <y:EdgeLabel>bajar</y:EdgeLabel>
        </y:PolyLineEdge>
      </data>
    </edge>
    <edge id="e9" source="n5" target="n3">
      <data>
        <y:PolyLineEdge>
          <y:LineStyle color="#000000" type="line" width="1.0"/>
          <y:Arrows source="none" target="standard"/>
          <y:EdgeLabel></y:EdgeLabel>
        </y:PolyLineEdge>
      </data>
    </edge>
  </graph>
</graphml>
```