FACULTAD DE MATEMÁTICAS

Trabajo fin de Grado

Grado en Matemáticas

Librería Haskell sobre Model Checking

Realizado por
Pablo Domínguez Balbás

Dirigido por
Francisco Jesús Martín Mateos

Departamento
Ciencias de la Computación
e Inteligencia Artificial

Sevilla, Junio de 2020

Abstract

In this project we establish the basic theoretical concepts about *Finite State Machines* and *Model Checking* so we can implement those concepts in the form of a fully functional *Haskell* package, that can be used by anyone interested in an introduction to these fields.

More specifically, we present the ground definitions to formally define the mathematical object of *Automata* and the alphabet it accepts, whereas also presenting the two basic temporal logics of LTL and CTL, and the model checking algorithm for CTL.

Índice general

Índice de figuras

Índice de código

CAPÍTULO 1

Introducción

La primera mención que se hace sobre los autómatas, o las máquinas de estados finitos, surge de Warren McCulloch y Walter Pitts, dos neurofisiólogos quienes hacia el año 1943 elaboraban una primera descripción de este concepto mientras formaban parte de un equipo de investigación multidisciplinar. Su objetivo era poder modelar a través de conceptos abstractos el pensamiento humano. Cabe destacar que este mismo origen también ha tenido impacto en la teoría de redes neuronales y en otras áreas de las ciencias de la computación. Más tarde, este concepto fue generalizado por parte de George H. Mealy y Edward F. Moore, quienes construyeron una definición mucho más formal, estricta y rica y a partir de los cuales fueron nombrados respectivamente los autómatas de Moore y los autómatas de Mealy.

Por otra parte y de forma paralela, también durante mediados del siglo XX, se desarrollaba lo que más tarde se conocería como *Lógica temporal*. Esta nueva rama de la lógica, iniciada por Arthur Prior, buscaba suplir la aparente carencia de los sistemas lógicos desarrollados hasta el momento de poder realizar afirmaciones que no necesariamente son ciertas en el instante presente, pero que pueden llegar a serlo en un futuro, y lo que es más, de poder describir aportando un sistema lógico el tiempo inherente a estas afirmaciones.

Por último, hacia finales de la década de los noventa, gracias a un aumento del interés en lenguajes de evaluación perezosa y a la intención de crear un lenguaje puramente funcional y abierto, nace el lenguaje de programación que hoy conocemos por *Haskell*.

En nuestro caso, en este trabajo vamos a emplear los conceptos de la teoría de autómatas finitos deterministas junto con herramientas de la lógica temporal con el fin de llegar a desarrollar una librería en Haskell donde se implementen estos conocimientos, contribuyendo así al conjunto de librerías abiertas ya desarrolladas en este lenguaje de programación.

Para ello, hemos distribuido principalmente el trabajo en cuatro capítulos, donde se recoge la información que luego se emplea en la librería. Destacar que la finalidad en última instancia de este proyecto es la librería en si, que puede encontrarse en el repositorio oficial *Hackage*.

- **Capítulo 1**: En este capítulo presentaremos las definiciones básicas tanto de lenguajes regulares como de autómatas finitos deterministas, ambos conceptos íntimamente relacionados.
- **Capítulo 2**: En este segundo capítulo, desarrollamos la base de la lógica temporal

dividida en dos ramas principales, por una parte *CTL* y por otra parte *PLTL*, para luego combinarlas en un sistema lógico más rico conocido como *CTL**.

- **Capítulo 3**: En este tercer capítulo, unificamos todos los conceptos de los dos capítulos anteriores para presentar el algoritmo de model checking para *CTL*. Este algoritmo nos permite extraer propiedades y conclusiones de un objeto, autómata, bajo un sistema lógico, el de la lógica temporal.
- **Capítulo 4**: En este último capítulo, detallamos las especificaciones tanto de las herramientas utilizadas para la construcción de la librería, como de la librería per sé.

Personalmente, sugiero al lector que, si dispone de tiempo para ello, pruebe personalmente la librería, instalándola en su sistema con el comando `cabal install FSM`, y al mismo tiempo, dado que es totalmente de código abierto, le invito a realizar los cambios, experimentos y pruebas que considere oportunos sobre la misma.

CAPÍTULO 2

Autómatas y Lenguajes Regulares

En este primer capítulo recogeremos los conceptos básicos sobre las herramientas matemáticas que más tarde utilizaremos para realizar comprobaciones sobre modelos de lógica temporal, muchos de los cuales han sido extraidos de [2] y de [3]. Aunque los conceptos son extensibles a su versión no finita, salvo que se especifique lo contrario todos los elementos que intervendrán en este capítulo se considerarán finitos.

2.1– Lenguajes regulares

Definición 2.1. Llamamos **alfabeto** a cualquier colección finita E de caracteres. Se define una palabra o **cadena** como a cualquier secuencia finita de elementos de E.

Definición 2.2. Sea E un alfabeto, se define E^* como el conjunto de todas las posibles cadenas de E, incluyendo la cadena vacía. Así, definimos sobre E^* la siguiente operación $\circ$ llamada **concatenación** de la siguiente manera: Sean $a_1a_2\cdots a_n, b_1b_2\cdots b_m \in E^*$ entonces $a_1a_2\cdots a_n \circ b_1b_2\cdots b_m = a_1a_2\cdots a_nb_1b_2\cdots b_m$.

Del mismo modo, si $S, T \subseteq E^*$ entonces $S \circ T = \{s \circ t : s \in S, t \in t\}$. Usualmente, $S \circ T$ se denota como ST.

Definición 2.3. Si denotamos λ a la cadena vacía y sea $B \subseteq E^*$, se define B^* como el conjunto formado a partir de concatenar cadenas de B, es decir, $B^* = \{b_1b_2\cdots : b_i \in B\} \cup \{\lambda\}$

Definición 2.4. Sea E un alfabeto, se define un **lenguaje** sobre E a cualquier $L \subseteq E^*$.

Definición 2.5. Sea E un alfabeto, consideremos los operadores $+, \circ, *, (,)$ y los símbolos $\emptyset, \lambda$ entonces:

- $a \in E, \emptyset, \lambda$ son **expresiones regulares**.
- Si r_1 y r_2 son expresiones regulares entonces $r_1 + r_2, r_1 \circ r_2, r_1^*$ y (r_1) también lo son.
- No hay expresiones regulares que no estén generadas por los dos anteriones puntos.

Definición 2.6. Sea r una expresión regular, se define el lenguaje $L(r)$ como:

- $\emptyset$ es una expresión regular que denota el conjunto vacío.

- λ es una expresión regular que denota $\{\lambda\}$
- $\forall\, a \in E, a$ es una expresión regular que denota $\{a\}$
- $L(r_1 + r_2) = L(r_1) + L(r_2)$
- $L(r_1 \circ r_2) = L(r_1) \circ L(r_2)$
- $L((r_1)) = L(r_1)$
- $L(r_1^*) = (L(r_1))^*$

Siendo r_1, r_2 expresiones regulares. Para evitar confusiones en las expresiones, se toma por convenio que * precede a $\circ$ y $\circ$ precede a $+$.

Teorema 2.1. *Sean L_1 y L_2 lenguajes:*

- *Si L_1 es finito, entonces es regular.*
- *Si L_1 es regular, entonces L_1^* es regular.*
- *Si L_1 y L_2 son regulares, entonces $L_1 \cup L_2$ es regular.*
- *Si L_1 y L_2 son regulares, entonces $L_1 \circ L_2$ es regular.*
- *Si L_1 es regular, entonces L_1^c es regular.*
- *Si L_1 y L_2 son regulares, entonces $L_1 \cap L_2$ es regular.*

2.2– Autómatas

En cuanto al concepto de autómata, existen distintas formas de abordar ésta definición. La elección entre una u otra forma dependerá de si para nuestro caso resulta de mayor relevancia lo que está ocurriendo en cada momento, o si por otra parte es más importante destacar lo que hace que las cosas cambien. Así, un autómata puede quedar definido desde el punto de vista de los estados (*state-based*) o bien desde el punto de vista de sus transiciones (*action-based*). En todo nuestro desarrollo nos basaremos en aquellos sistemas del tipo *action-based*, siguiendo la definición que se presenta en el desarrollo teórico de autómatas en [4].

Definición 2.7. Se define un **autómata** o sistema de transición como a la tupla $\mathcal{A} = (Q, E, T, q_0, F)$, siendo estos:

- Q es el conjunto de posibles estados de $\mathcal{A}$
- E es un alfabeto.
- $T \subseteq Q \times E \times Q$, es el conjunto de transiciones.
- q_0 es el estado inicial del autómata.
- $F \subseteq Q$, es el conjunto de estados de aceptación.

También podemos encontrar definiciones donde en lugar de aportar F, encontraremos l como la aplicación que asocia cada estado de Q con el conjunto finito de las propiedades elementales que se tienen sobre dicho estado.

2.2.1. Diagrama de transición

Usualmente, aunque en esta sección nos referiremos al concepto más puramente formal, los sistemas de transición se pueden representar de una manera **gráfica** visualmente más atractiva empleando grafos, donde los estados serán los vértices de nuestro grafo y las transiciones representan las aristas. Por ejemplo si una arista une los estados (vértices) 2 y 4, y tiene asociada un carácter b, la transición es $\{2, b, 4\}$. Por otra parte, a los estados de aceptación los representaremos con un doble círculo. Así, en el siguiente ejemplo sencillo se muestra un autómata que aceptará aquellas cadenas de caracteres en las que la primera ocurrencia del carácter a, tenga dos apariciones consecutivas de éste carácter. Formalmente, nuestro autómata sería la tupla

$$\begin{gathered}(Q = \{1, 2, 3, 4\},\\ E = \{a, b\},\\ T = \{\{1, a, 2\}, \{1, b, 1\}, \{2, a, 3\}, \{2, b, 4\}, \{3, a, 3\}, \{3, b, 3\}, \{4, a, 4\}, \{4, b, 4\}\},\\ q0 = 1,\\ F = \{3\})\end{gathered}$$

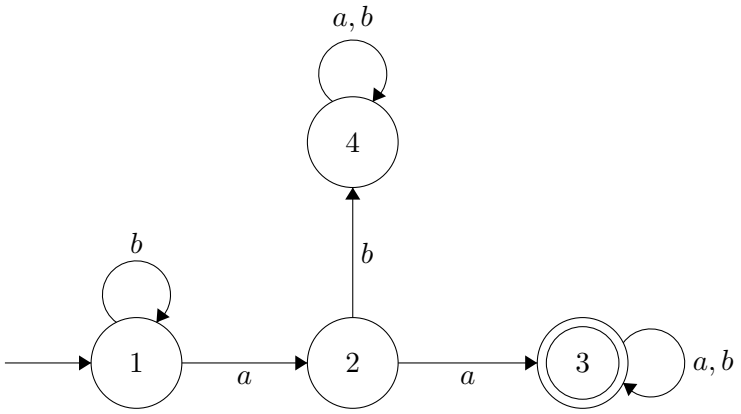

Figura 2.1: Diagrama de transición

Definición 2.8. Sea $\mathcal{A}$ un autómata, se define un **camino** sobre $\mathcal{A}$ como a aquella secuencia σ finita o infinita de transiciones de $\mathcal{A}$. Además, se define la **longitud** de un camino σ, denotada por $|\sigma|$ como al número de transiciones de dicho camino. Cabe destacar que dicho número puede ser o bien natural o bien infinito. El estado i-ésimo de un camino es el estado alcanzado tras haberse sucedido i transiciones. Si el camino σ es finito, y q_n es el último estado que alcanza, decimos que σ tiene éxito (es aceptado por $\mathcal{A}$) si $q_n \in F$.

Definición 2.9. Sea $\mathcal{A}$ un autómata, se define una **ejecución parcial** como a aquél camino que parte del estado inicial q_0.

Definición 2.10. Sea $\mathcal{A}$ un autómata, se define una **ejecución completa** como a aquella ejecución parcial maximal, es decir, que no puede ser extendida. Toda ejecución completa o bien es infinita o bien desemboca en un estado q_n a partir del cual no existe ninguna transición posible, en cuyo caso nos referimos a éste como punto muerto o *deadlock*.

Definición 2.11. Sea $\mathcal{A}$ un autómata, $q \in Q$ se dice alcanzable si existe al menos una ejecución completa en la que dicho estado aparece al menos una vez.

2.2.2. Variables de estado

Cuando estemos trabajando con sistemas de transición del tipo *state-based*, usualmente es recomendable incluir en nuestras consideraciones lo que llamaremos *variables de estado*, que no serán más que acumuladores que cambian con el autómata. Éste puede hacer que dichas variables cambien de dos maneras:

- Mediante asignaciones, esto es, una transición puede modificar el valor de una variable de estado.
- Mediante guardas, usualmente añadiendo condicionales podemos prevenir o modificar las transiciones de un estado a otro en función del valor de la variable de estado.

Si queremos aplicar técnicas de *model-checking* es conveniente desarrollar el comportamiento entre el autómata y las variables de estado, sus posibles valores y únicamente las posibles transiciones empleando un grafo de estados. Al autómata que se genera a partir del desarrollo de este otro si se le considera puramente un sistema de transición, ya que en él no hay guardas, condicionales o asignaciones pues consideramos todos los posibles casos individualmente.

2.3– Autómatas y lenguajes regulares

Definición 2.12. Sea $\mathcal{A}$ un autómata, definimos el **lenguaje** reconocido por $\mathcal{A}$ como

$$L(\mathcal{A}) = \{w \in E^* : w \text{ es aceptado por } \mathcal{A}\}$$

Definición 2.13. Se dice que el autómata $\mathcal{A}$ es **determinista** si $\forall\, q \in Q, w \in E\, \exists!\, q' \in Q \,/\, (q, w, q') \in T$.

Si $\mathcal{A}$ es determinista, podemos definir la función de transición $\delta : Q \times E \to Q$ tal que $\delta(q, w) = q'$

A su vez, podemos extender δ a una aplicación $\delta : Q \times E^* \to Q$ de manera recursiva tal que:

$$\delta(q, -) = q$$
$$\delta(q, wu) = \delta(\delta(q, w), u) \text{ con } w \in E^*, u \in E.$$

Donde $\{-\}$ representa la cadena vacía, o "no hacer nada". La idea es que si $\mathcal{A}$ está en un estado q y evalúa la cadena w entonces estará en un estado $\delta(q, w)$. De este modo, si $q_0, w_1, q_1, \cdots, w_n, q_n$ es una ejecución parcial, entonces $q_n = \delta(q_0, a_1, \cdots, a_n)$ y por tanto

$$L(\mathcal{A}) = \{w \in E^* : \delta(q_0, w) \in F\}$$

Definición 2.14. Un autómata $\mathcal{A}$ se dice generalizado o **no determinista** si transiciones de la forma $(q, -, q')$ están permitidas. Cabe destacar que el tipo de lenguaje reconocido por un autómata, sea o no determinista, es el mismo.

Teorema 2.2. *Sea E un alfabeto y $L \subseteq E^*$ un lenguaje, las siguientes afirmaciones son equivalentes:*

I) L es reconocido por un autómata determinista.

II) L es reconocido por un autómata.

III) L es reconocido por un autómata no determinista.

Demostración:
Es trivial que I $\Rightarrow$ II y que II $\Rightarrow$ III.
Veamos que III $\Rightarrow$ I. Supongamos que L es reconocido por $\mathcal{A} = (Q, E, T, q_0, F)$ no determinista. Sea $X \subseteq Q$, consideremos el conjunto $\overline{X}$ de todos los puntos a los que se llega en el diagrama de transición partiendo de puntos de X a partir de la transición $-$. Se tiene entonces por un lado que $X \subseteq \overline{X}$ y que $\overline{\overline{X}} = \overline{X}$. Definimos así el autómata determinista $\mathcal{A}' = (Q', E, T', q_0', F')$ de la siguiente manera:

$$Q' := \{X \subseteq Q : X = \overline{X}\}$$
$$\delta(X, w) = \overline{\{q' \in Q : (q, -, q') \text{ es una transición con } q \in X\}}$$
$$T' := \{(X, w, \delta(X, w)) : X \in Q', w \in E\}$$
$$q_0' = \overline{\{q_0\}}$$
$$F' := \{X \in Q' : q \in X \text{ para algún } q \in F\}$$

Se tiene entonces que $L(\mathcal{A}') = L(\mathcal{A}) = L$.

□

Definición 2.15. Si se tiene cualquiera de las situaciones del teorema anterior, decimos que L es reconocido por un autómata.

Teorema 2.3. *Sea L un lenguaje, decimos que L es un **lenguaje regular** si y sólo si L es reconocido por un autómata.*

2.4– Producto sincronizado

Cuando queramos modelar algún sistema de la vida real, una buena estrategia es dividir el proceso en subsistemas, modelar estos y luego unirlos para construir el sistema original. A este proceso, en términos de sistemas de transición, se le conoce como **sincronización**. Dedicaremos esta sección a estudiar los métodos más comunes de sincronización. Cabe destacar que un autómata que represente el sistema global, si lo intentásemos construir directamente, tendría tal número de estados que se volvería impracticable su construcción. Nos referimos a este fenómeno como una explosión de estados.

Definición 2.16. Sean los autómatas $\mathcal{A}_1, \cdots, \mathcal{A}_n$, se define su producto cartesiano $\mathcal{A}_1 \times \cdots \times \mathcal{A}_n$ como al autómata $\mathcal{A} = (Q, E, T, q_0, F)$ de manera que:

- $Q = Q_1 \times \cdots \times Q_n$
- $E = \prod_{i=1}^{n} (E_i \cup \{-\})$, donde aquí nos referimos al producto cartesiano y $\{-\}$ representa la cadena vacía, o dicho de otro modo, a "no hacer nada".
- $T = Q \times E \times Q$ cumpliendo que si $e_i = -$ entonces $q_i = q_i'$
- $q_0 = (q_0^1, \cdots, q_0^n)$

- $F = F_1 \times \cdots \times F_n$

Dentro de este producto cartesiano, cada uno de los autómatas puede o bien no hacer nada, como indica la transición ficticia $-$ o bien realizar una transición 'local'. No obstante, este concepto no es suficiente para sincronizar dichos autómatas ya que por ejemplo, una de las posibles transiciones es $((q_i, \cdots, q_i), (-, \cdots, -), (q_i, \cdots, q_i))$ que es no hacer nada, cosa que no contemplábamos en nuestra definición anterior de autómata.

Para poder sincronizar un autómata, deberemos sustituir E por un subconjunto $S \subseteq E$ de manera que las transiciones construidas a partir de este conjunto se comporten como el sistema global. Cuando las partes del sistema no interactúan unas con otras se cumple que $S = E$. En el resto de casos, habremos de estudiar el sistema para poder construir S.

La estrategia para sincronizar sistemas de transición dependerá fundamentalmente del comportamiento de éstos y de cómo queramos que el sistema global se comporte. Así, uno de los posibles métodos para sincronizar el sistema es el siguiente:

2.4.1. Sincronización por transmisión de mensaje

Este método de sincronización es comúnmente empleado cuando estamos modelando un sistema que emite y recibe mensajes. La notación usual, siendo $a \in E$ es poner $!a$ cuando se emite este carácter, y $?a$ cuando se recibe. A la hora de construir el producto sincronizado, únicamente están permitidas las transiciones en las que cada emisión tiene acompañada su correspondiente recepción del mensaje. En función de si la emisión y la recepción del mensaje ocurren o no de manera simultánea, hablaremos de comunicación síncrona o asíncrona respectivamente.

En el segundo caso, los mensajes no son recibidos de manera inmediata sino que quedan en espera en uno o más canales de comunicación, llamados buffers. El orden de entrada de los mensajes será FIFO (del inglés *first in first out*).

CAPÍTULO 3

Lógica Temporal

3.1– Introducción y motivación de la lógica temporal

La lógica temporal, a diferencia de la lógica de primer orden, aspira a poder describir el comportamiento dinámico de diversos sistemas, es decir, aquellas propiedades que cambian con el tiempo, añadiendo a ésta conceptos como "siempre", "algunas veces" o "nunca" entre otros. Aunque ciertamente muchas de las proposiciones también podrían escribirse a la manera de la lógica de primer orden, para ello tendríamos la necesidad de crear numerosas variables y cuantificadores adicionales para denotar los distintos instantes de tiempo, mientras que la lógica temporal proporciona herramientas más potentes al tiempo que sencillas para abstraer y afrontar problemas que evolucionan en el tiempo. Un ejemplo de ello puede ser el querer expresar la sentencia "I es siempre mayor que 3 y a veces menor que 6" usando lógica de primer orden lo que tendríamos sería:

$$\forall\ t\ \ \big([I > 3] \wedge [\ \exists\ t' : [I < 6]]\big)$$

Mientras que si usasemos las herramientas que la lógica temporal, como veremos en este capítulo, esta sentencia queda mucho más simplificada. Existen dos lógicas que usualmente se aplican en sistemas dinámicos, que son CTL (*Computation Tree Logic*) y LTL (*Linear Temporal Logic*). Según la situación es posible que sea más útil una u otra, aunque en este capítulo desarrollaremos una combinación de ambas denotada por CTL^*.

3.2– El lenguage de la lógica temporal

Al igual que otras lógicas temporales, CTL^* sirve para expresar formalmente las propiedades que tienen las ejecuciones de un sistema. En esta sección vamos a ver cuáles son las herramientas básicas empleadas para ello.

- Como vimos en el capítulo anterior, una ejecución se define como una secuencia de estados. La lógica temporal usa proposiciones atómicas para realizar afirmaciones sobre los estados del sistema. Estas proposiciones son afirmaciones elementales que para un estado del sistema tienen un valor de verdad bien definido. Podemos considerar como proposiciones atómicas "buen tiempo", "ocupado", etc...

- Además, en la sintaxis de la lógica temporal incluimos también parte de los combinadores booleanos clásicos, que son T, F, $\neg$, $\wedge$, $\vee$, $\Rightarrow$ y $\Leftrightarrow$. Estos combinadores nos permiten construir afirmaciones más complejas a partir de subfórmulas más simples. Nos referimos a una fórmula proposicional cuando tenemos una mezcla de proposiciones y combinadores booleanos.

- Los combinadores temporales nos permiten expresar propiedades del sistema que evolucionan con el tiempo, y es lo que en esencia distingue a la lógica temporal de la lógica de primer orden. En primer lugar, si P es una propiedad que se tiene en un determinado estado, los combinadores más básicos son:

  - $\mathbb{X}P$ expresa que el "siguiente" estado satisface P
  - $\mathbb{F}P$ expresa que un estado futuro satisfará P, sin especificar cual. Este puede leerse como P *ocurrirá algún día.*
  - $\mathbb{G}P$ expresa que todos los estados futuros satisfacen P. Este puede leerse como P *ocurrirá siempre.*

  Se puede observar que existe una fuerte relación entre $\mathbb{F}$ y $\mathbb{G}$, y no es coincidencia pues uno es dual del otro en el sentido formal, es decir, para cualquier fórmula ϕ se tiene que $\mathbb{G}\phi$ y $\neg\mathbb{F}\neg\phi$ son equivalentes.

- Es la capacidad de anidar de manera arbitraria los combinadores temporales lo que aporta a la lógica temporal propiedades realmente interesantes, pues a partir de sub-fórmulas más simples podemos construir nuevas fórmulas.

  Debido a que los combinadores $\mathbb{F}$ y $\mathbb{G}$ suelen ser empleados conjuntamente para expresar propiedades relacionadas con la repetición, se abrevia el uso de $\mathbb{GF}\phi$, que literalmente debe ser leido como ϕ *siempre ocurrirá algún dia futuro*, y en su lugar usaremos $\overset{\infty}{\mathbb{F}}$. Su dual es $\overset{\infty}{\mathbb{G}}$, abreviatura de $\mathbb{FG}\phi$, leido como ϕ *ocurrirá siempre a partir de cierto punto.*

- El combinador $\mathbb{U}$ para indicar *hasta que*, es un combinador más rico y complicado si lo comparamos con los presentados anteriormente. Por ejemplo $\phi_1\mathbb{U}\phi_2$ significa que ϕ_1 se verifica *hasta que* se verifique ϕ_2, o visto de otro modo, ϕ_2 se verificará *algún dia*, y mientras tanto se tendrá ϕ_1.

  Visto de este modo, el combinador $\mathbb{F}$ es un caso particular de $\mathbb{U}$ en el sentido de que $\mathbb{F}\phi$ y $\texttt{true}\mathbb{U}\phi$ son equivalentes.

  Existe también un *hasta que* más débil, denotado por $\mathbb{W}$. La afirmación $\phi_1\mathbb{W}\phi_2$ sigue expresando ϕ_1 *hasta que* ϕ_2, pero eliminando la suposición de que ϕ_2 necesariamente ocurrirá. De esta manera, lo anterior también puede ser expresado como ϕ_1 mientras no ocurra ϕ_2. Cabe destacar que $\mathbb{W}$ puede ser expresado en términos de $\mathbb{U}$ teniendo en cuenta la siguiente equivalencia:

  $$\phi_1\mathbb{W}\phi_2 \equiv (\phi_1\mathbb{U}\phi_2) \vee \mathbb{G}\phi_1$$

- A pesar de la enorme utilidad de las herramientas arriba presentadas, éstas únicamente nos permiten expresar propiedades relativas a una ejecución. Es por esto que necesitamos aún herramientas para reflejar el comportamiento tipo *árbol* que

nuestro sistema pueda tener. Para esto usaremos los cuantificadores $\mathbb{A}$ y $\mathbb{E}$, también llamados *cuantificadores de camino.*

La fórmula $\mathbb{A}\phi$ expresa que *todas las ejecuciones* a partir del estado actual safisfacen ϕ, mientras que $\mathbb{E}\phi$ lo que expresa es que *existe una ejecución* que satisface ϕ. Es importante clarificar la diferencia entre $\mathbb{A}$ y $\mathbb{G}$: $\mathbb{A}\phi$ expresa que todas las posibles ejecuciones satisfacen ϕ, mientras que $\mathbb{G}\phi$ expresa que en la ejecución que estamos considerando ϕ se satisfará siempre.

En la siguiente figura podemos ver gráficamente la representación de las combinaciones de los operadores $\mathbb{A}$ y $\mathbb{E}$, con los operadores $\mathbb{X}$, $\mathbb{F}$, $\mathbb{G}$ y $\mathbb{U}$:

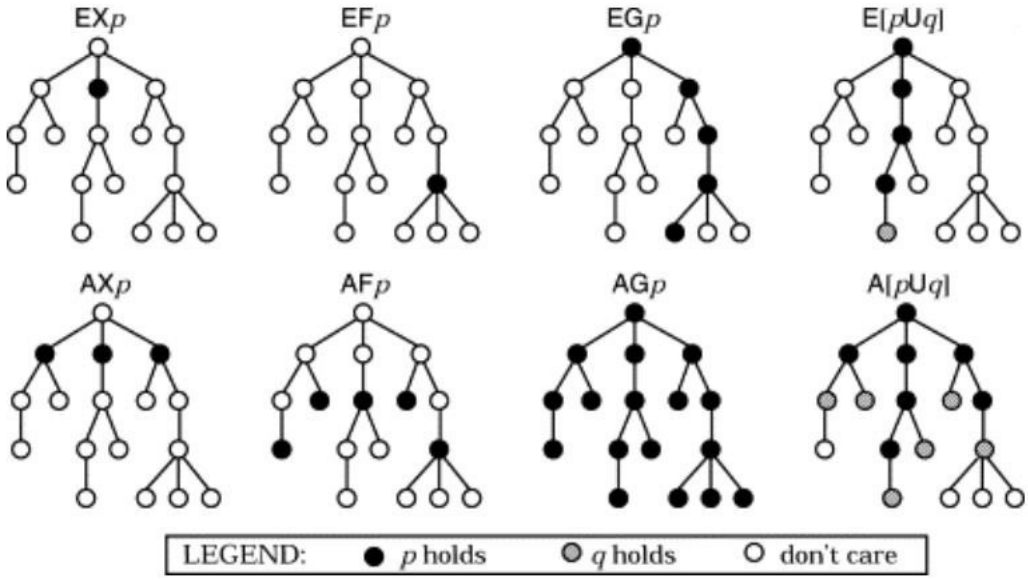

Figura 3.1: CTL[1]

En CTL^*, $\mathbb{A}$ y $\mathbb{E}$ son duales el uno de el otro, como es común entre los cuantificadores existenciales y universales: si $\mathbb{A}\phi$ no se verifica, entonces existe una ejecución que no satisface ϕ, y por tanto satisface $\neg\phi$, ergo $\mathbb{A}\phi$ y $\neg\mathbb{E}\neg\phi$ son equivalentes.

3.3– Sintaxis formal

Los conceptos presentados en la sección anterior nos llevan de forma natural a presentar la gramática formal de CTL^*, descrita similarmente a [7]pag224. Sean ϕ y ψ dos fórmulas:

$$
\begin{array}{lll}
\phi, \psi \quad ::= & P_1 \mid P_2 \mid \ldots & \text{(proposiciones atómicas)} \\
& \mid \neg\phi \mid \phi \wedge \psi \mid \phi \Rightarrow \psi \mid \ldots & \text{(combinadores booleanos)} \\
& \mid \mathbb{X}\phi \mid \mathbb{F}\phi \mid \mathbb{G}\phi \mid \phi\mathbb{U}\psi \mid \ldots & \text{(combinadores temporales)} \\
& \mid \mathbb{E}\phi \mid \mathbb{A}\phi & \text{(cuantificadores de caminos)}
\end{array}
$$

Presentamos esta gramática de forma abstracta. En la práctica, cada herramienta allí donde se implementen fórmulas temporales permitirá el uso de paréntesis y tendrá sus

[1]Esta figura se ha tomado de [8]

propias convenciones en las prioridades de los operadores, y además tendrá su propio conjunto completo de conectivas y de proposiciones atómicas, de forma que aunque la lógica temporal empleada sea la misma, ésta puede definirse de la forma más oportuna para cada caso.

3.4– Semántica y reglas de la lógica temporal

Si la lógica en la que construimos nuestro sistema la establecemos basándonos en autómatas del tipo *state-based*, a éstos modelos los denominaremos *Estructuras de Kripke*, mientras que si nos basamos en autómatas del tipo *action-based* los llamaremos *sistemas de transición etiquetados*, y son en éstos últimos en los que vamos a desarrollar la semántica de la lógica CTL^*. No obstante, un desarrollo alternativo sería muy similar pues ambos sistemas lógicos son distintos puntos de vista del mismo problema.

Sean $\mathcal{A}$ un autómata, σ una ejecución sobre dicho autómata e i un instante de dicha ejecución, y consideremos ϕ una fórmula temporal, diremos que $\mathcal{A}, \sigma, i \models \phi$, y se leerá como *en el instante i de la ejecución σ, se cumple ϕ*, donde σ es una ejecución de $\mathcal{A}$ no necesariamente empezando en el estado inicial.

Es usual omitir el autómata al que nos referimos si estamos trabajando dentro del mismo contexto. Así mismo, escribiremos $\sigma, i \not\models \phi$ si ϕ no se satisface en el instante i de σ. A continuación presentamos las reglas básicas de la semántica de la lógica temporal

- $\sigma, i \models P \Leftrightarrow P \in l(\sigma(i))$
- $\sigma, i \models \neg\phi \Leftrightarrow$ no se tiene que $\sigma, i \models \phi$
- $\sigma, i \models \mathbb{X}\phi \Leftrightarrow i < |\sigma| \wedge \sigma, i+1 \models \phi$
- $\sigma, i \models \mathbb{F}\phi \Leftrightarrow \exists\, j \;/\; i \leq j \leq |\sigma| \wedge \sigma, j \models \phi$
- $\sigma, i \models \mathbb{G}\phi \Leftrightarrow \forall\, j \;/\; i \leq j \leq |\sigma|,\ \sigma, j \models \phi$
- $\sigma, i \models \phi \mathbb{U} \psi \Leftrightarrow \exists\, j \;/\; i \leq j \leq |\sigma|$ con $\sigma, j \models \psi \wedge\ \forall k \;/\; i \leq k \leq j$ se tiene que $\sigma, k \models \phi$
- $\sigma, i \models \mathbb{E}\phi \Leftrightarrow \exists\, \sigma' \;/\; \sigma(0) \cdots \sigma(i) = \sigma'(0) \cdots \sigma'(i) \wedge \sigma', i \models \phi$
- $\sigma, i \models \mathbb{A}\phi \Leftrightarrow\ \forall\, \sigma' \;/\; \sigma(0) \cdots \sigma(i) = \sigma'(0) \cdots \sigma'(i)$ set tiene que $\sigma', i \models \phi$

Partiendo de ésta semántica, decimos que *el autómata $\mathcal{A}$ satisface ϕ*, y se denota por $\mathcal{A} \models \phi \Leftrightarrow \sigma, 0 \models \phi$ para cualquier σ ejecución de $\mathcal{A}$. En CTL^*, el tiempo se tratará como discreto, y consideramos que nada ocurre ni existe entre el instante i y el $i+1$

3.5– PLTL vs CTL

PLTL (*Propositional Linear Temporal Logic*) y *CTL* (*Computation Tree Logic*) son las dos lógicas temporales más extendidas en lo que a herramientas de model checking se refiere, aunque sus orígenes difieren.

Por un lado, *PLTL* es el fragmento de CTL^* que se obtiene al eliminar los cuantificadores $\mathbb{A}$ y $\mathbb{E}$. De este modo, una fórmula ϕ en *PLTL* sobre el contexto de una ejecución dada carece de la capacidad para examinar ejecuciones alternativas que se separen de la ejecución inicial en cada paso. *PLTL* únicamente se centra en el conjunto de ejecuciones

y no en la manera en la que éstas están organizadas en un árbol. Por ejemplo, $PLTL$ no puede expresar que en algunos instantes a lo largo de una ejecución sería posible extender la ejecución de este u otro modo. En la siguiente figura mostramos dos autómatas que desde el punto de vista de este tipo de lógica temporal resultan indistinguibles, de manera que si una fórmula ϕ se tiene para uno, se tendrá también para el otro.

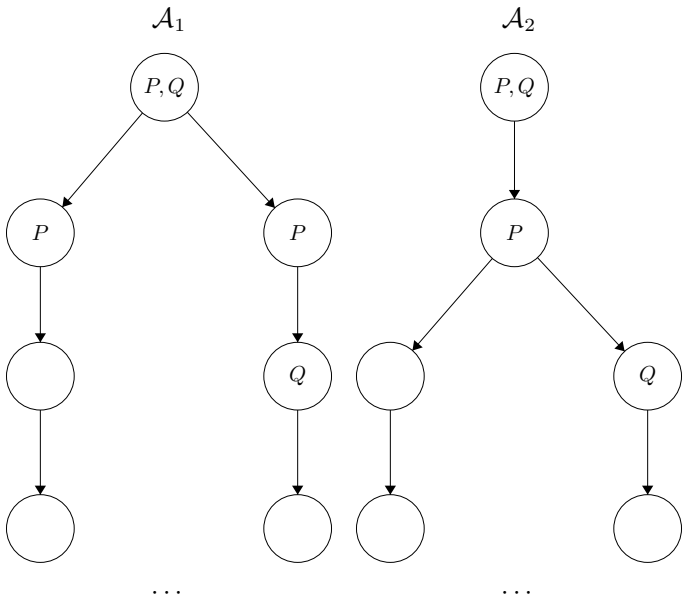

Figura 3.2: Autómatas no distinguibles en $PLTL$

Por útlimo, CTL es el fragmento de CTL^* que se obtiene al exigir que cada combinador temporal ($\mathbb{X}, \mathbb{F}, \mathbb{U}, ..$) se encuentre inmediatamente precedido de un cuantificador de camino, ya sea $\mathbb{A}$ o $\mathbb{E}$.

La necesidad de cuantificar sobre los posibles futuros limita susceptiblemente la expresividad de CTL. En este lenguaje no es posible anidar varios combinadores temporales refiriéndonos a la misma ejecución. Una consecuencia de esto es que las fórmulas CTL son fórmulas de estado, es decir, la fórmula $\sigma, i \models \phi$ únicamente depende del estado $\sigma(i)$ y no depende de la ejecución actual. Por otra parte, esta misma limitación también tiene sus beneficios, como veremos en el último capítulo.

La elección de uno u otro lenguaje dependerá del ámbito sobre el que estemos trabajando. En la práctica, si nuestro objetivo es describir el comportamiento de un sistema, es probable que encontremos mayor dificultad para hacerlo con CTL que con $PLTL$. No obstante, la mayoría de desventajas que CTL pueda tener sobre $PLTL$ se ven compensadas por el hecho de que las técnicas de *model checking* son mucho más eficientes expresadas en CTL que en $PLTL$. En resumen, si nuestro objetivo es enunciar propiedades sobre el sistema, lo recomendable sería emplear $PLTL$, mientras que si lo que buscamos es la verificación exhaustiva del sistema, usaremos CTL.

CAPÍTULO 4

Model Checking

En este capítulo nos centraremos en describir los principios subyacentes de los algoritmos empleados en *model checking*, es decir, aquellos que nos permiten averiguar si un autómata dado satisface una fórmula temporal. Consideraremos este planteamiento de manera separada para CTL y para $PLTL$, dado que en uno u otro caso obtendremos respuestas de carácter distinto.

4.1– *Model Checking* en *CTL*

El algoritmo de *model checking* para CTL se desarrolló inicialmente por Queille, Sifakis, Clarke, Emerson y Sistla. Este algoritmo, descrito en [1], juega un papel muy relevante en el área de la verificación de sistemas, debido a que el algoritmo se puede ejecutar en tiempo lineal en términos de cada una de sus componentes, que son por una parte el autómata y por otra la fórmula de CTL. Este algoritmo se basa en el hecho de que CTL únicamente puede expresar fórmulas de estado.

La componente principal de este algoritmo es un procedimiento que irá marcando sobre el autómata $\mathcal{A}$, partiendo de la fórmula ϕ, para cada uno de los estados q del autómata y para cada una de las subfórmulas ψ de ϕ si dicha subfórmula se verifica en ese estado. Utilizaremos variables de la forma `q.phi` para indicar si en el autómata se cumple la propiedad ϕ a partir del estado q. Al final, obtendremos que la variable `q.phi` tiene el valor `true` si $q \models \phi$, y tendrá el valor `false` en otro caso.

Usaremos el término *marcado* para indicar que el valor de `q.phi` es primero computado y luego memorizado. Esto es importante porque el *marcado* de `q.phi` usa los valores de `q'.psi` para subfórmulas ψ de ϕ y para los estados `q'` alcanzables desde `q`. Cuando el *marcado* de `phi` (ϕ) se complete, resulta sencillo decidir si $\mathcal{A} \models \phi$ simplemente a partir del valor de `q0.phi`, siendo $q0$ el estado inicial de $\mathcal{A}$. A continuación, pasamos a presentar el esqueleto del algoritmo:

```
1  metodo marcado(phi):
2      caso 1: phi = P
3          para todo q en Q,
4              si p en l(p) entonces q.phi := true,
5              en otro caso q.phi := false.
```

```
6
7       caso 2: phi = not psi
8           marcado(psi);
9           para todo q en Q, q.phi := not(q.psi).
10
11      caso 3: phi = psi1 ∧ psi2
12          marcado(psi1); marcado(psi2);
13          para todo q en Q, q.phi := and(q.psi1, q.psi2).
14
15      caso 4: phi = 𝔼𝕏 psi
16          marcado(psi);
17          para todo q en Q, inicializamos q.phi := false;
18          para todo (q,q') en T,
19              si q'.psi = true entonces q.phi := true.
20
21      caso 5: phi = 𝔼 psi1 𝕌 psi2
22          marcado(psi1); marcado(psi2);
23          para todo q en Q,
24              inicializamos q.phi := false; q.previo := false;
25              inicializamos L := {};
26          para todo q en Q,
27              si q.psi2 = true entonces L := L + {q};
28          while L no vacio {
29              quitamos q de L;
30              L := L - {q};
31              q.phi := true;
32              para todo (q',q) en T {
33                  si q'.previo = false entonces {
34                      q'.previo := true;
35                      si q'.psi = true entonces L := L + {q'};
36                  }
37              }
38          }
39
40      caso 6: phi = 𝔸 psi1 𝕌 psi2
41          marcado(psi1); marcado(psi2);
42          L := {}
43          para todo q en Q,
44              inicializamos q.nb := grado(q); q.phi := false;
45          para todo q en Q,
46              si q.psi2 = true entonces L := L + {q};
47          while L no vacio {
48              quitamos q de L;
49              L := L - {q};
50              q.phi := true;
51              para todo (q',q) en T {
52              q'.nb := q'.nb - 1;
53              si (q'.nb = 0) ∧ (q'.psi1 = true) ∧ (q'.phi = false)
54                  L := L + {q'};
55              }
56          }
```

Código 4.1: Marcado de CTL

Cuando ϕ es una proposición atómica (casos 1, 2 y 3) su marcado resulta sencillo, y puede ejecutarse en tiempo $O(|Q|)$ más el tiempo para marcar las subfórmulas de ϕ. Por otra parte, en el caso 4 el marcado requiere que recorramos el conjunto T de las transiciones del autómata $\mathcal{A}$, lo que nos requerirá esta vez un tiempo $O(|T|)$ más la inicialización y el marcado de ψ. No hemos tratado el caso $\mathbb{A}\mathbb{X}\psi$ por ser equivalente a $\neg\mathbb{E}\mathbb{X}\neg\psi$, es decir, que se podría tratar como caso particular de 2 y 4. En el caso 5, cuando la fórmula es de la forma $\mathbb{E}\psi_1\mathbb{U}\psi_2$ el marcado de ϕ comienza por los marcados de ψ_1 y ψ_2 usando un algoritmo estándar para estudiar la alcanzabilidad controlada en un grafo. En este caso, el algoritmo se podrá ejecutar con $O(|Q| + |T|)$.

Por último, el algoritmo de marcado para el caso 6, $\mathbb{A}\psi_1\mathbb{U}\psi_2$ resulta algo más complejo debido a que se basa en que un estado q verifica $\mathbb{A}\psi_1\mathbb{U}\psi_2$ si y solo si se tienen alguna de las siguientes condiciones:

- q verifica ψ_2
- q verifica ψ_1, q tiene al menos un estado posterior y todos sus estados posteriores satisfacen $\mathbb{A}\psi_1\mathbb{U}\psi_2$.

En este algoritmo llevaremos un contador `nb` asociado a cada estado el cual inicializamos con el número de estados sucesores, para después disminuir en una unidad el contador cada vez que el estado es marcado. Una vez este contador llegue a cero, sabremos que todos los estados alcanzables desde q satisfacen $\mathbb{A}\psi_1\mathbb{U}\psi_2$. Si además q satisface ψ_1, sabremos que satisface ϕ.

No obstante, nos queda por verificar la veracidad de este método. El caso base está claro, cada estado únicamente puede ser añadido una vez a L. Dado que los otros casos son más sencillos, nos limitaremos a analizar el caso donde ϕ es de la forma $\mathbb{A}\psi_1\mathbb{U}\psi_2$. En primer lugar asumimos por inducción que el marcado de las subfórmulas ψ_1 y ψ_2 es correcto. Podemos observar que si `q.phi` se establece como `true` en algún momento del marcado entonces q satisface ϕ. En el otro sentido, supongamos que `q.phi` es `false` al final del recorrido, entonces se tiene necesariamente que $q \not\models \psi_2$. Si además $q \not\models \psi_1$, entonces $q \not\models \phi$ y el valor de `q.phi` es correcto. Si por otro lado $q \models \psi_1$ y el grado de q es no nulo entonces necesariamente `q.nb` es estrictamente positivo al final del marcado, pues de otro modo q habría sido insertado en L. Por lo tanto, sea q_1 uno de los estados que suceden a q, tendrá como valor de `q1.phi` en `false`, y el mismo razonamiento puede ser aplicado de forma inductiva a q_1. Si no satisface ψ_2 y sí satisface ψ_1 entonces tiene un estado sucesor q_2 que no está marcado para ϕ, y así sucesivamente. Por este razonamiento llegaríamos a la conclusión de que $\psi_1\mathbb{U}\psi_2$ no se verifica, lo que demuestra que $q \not\models \phi$ y por tanto el marcado de q es correcto.

Complejidad del algoritmo: Cada paso en el algoritmo está asociado o bien con el marcado de $q \in Q$ o con el procesamiento de la transición (q', q) de T. Por lo tanto, una vez los marcados de ψ_1 y ψ_2 se hayan completado, el tiempo requerido para marcar ϕ será del orden de $O(|Q| + |T|)$, es decir $O(\mathcal{A})$. Finalmente, el marcado de una fórmula arbitraria ϕ requiere del marcado de aquellas sub-fórmulas de ϕ seguidas, por lo que el tiempo total necesario para el marcado será de $O(|\mathcal{A}| \times |\phi|)$

4.2– *Model Checking* en *PLTL*

Para explicar el principio que subyace en el algoritmo de *model checking* para *PLTL* requeriría de la introducción de autómatas de Büchi, los cuales quedan fuera del alcance

de este proyecto, por los que nos ceñiremos a una introducción sobre dicho algoritmo. Para el caso de $PLTL$ ya no nos enfrentamos a fórmulas de estado si no a fórmulas de camino, y un autómata finito podrá generar un número infinito de ejecuciones, cada una de ellas finitas por si misma en lo que a longitud se refiere. Consideremos por ejemplo la fórmula $\phi : \overset{\infty}{\mathbb{F}} P$, una ejecución $q_0, q_1, q_2, \cdots$ donde se satisface que ϕ debe contener infinitas posiciones $q_{n_1}, q_{n_2}, \cdots$ en las que se tiene P. Entre cada una de estas posiciones puede haber un número arbitrario de estados que satisfagan $\neg P$. De este modo, diremos que la ejecución es de la forma $((\neg P)^*.P)^\omega$.

Siguiendo esta línea, una ejecución que no satisface ϕ deberá necesariamente, desde cierto punto en adelante, contener únicamente estados que satisfacen $\neg P$. Se dice que una ejecución de este tipo es de la forma $(P + \neg P)^*.(\neg P)^\omega$. Las dos notaciones que acabamos de emplear para expresar la forma que una ejecución que satisfaga ϕ y que no la satisfaga se dicen $\omega - expresiones\ regulares$, las cuales extienden el concepto de expresión regular, y se emplean en el campo de lenguajes con un número infinito de palabras. En las $\omega - expresiones\ regulares$ añadimos al clásico exponente $*$, que indica una cantidad finita arbitraria de repeticiones, un nuevo exponente ω para indicar una cantidad infinita de repeticiones.

Debemos tener en cuenta cuando tratamos $Model\ checking$ en lógicas $PLTL$ que las técnicas que se tratan se basan en la posibilidad de asociar con cada fórmula ϕ de $PLTL$ una *ω-expresión regular* ε_ϕ que describa la forma que una ejecución ha de tener para satisfacer ϕ. La pregunta de *¿Se tiene que $\mathcal{A} \models \phi$?* se reduce a *¿Son todas las ejecuciones de $\mathcal{A}$ de la forma que describe ε_ϕ ?*. En la práctica, los algoritmos no razonan a partir de lenguajes regulares si no a partir de autómatas, de ahí que un el mecanismo de $Model\ checking$ para $PLTL$, dada la fórmula ϕ, sea construir el autómata $\mathcal{B}_{\neg\phi}$ que reconozca las ejecuciones que no satisfacen ϕ. A partir de este, deberemos combinarlo con $\mathcal{A}$ para obtener el autómata $\mathcal{A} \otimes \mathcal{B}_{\neg\phi}$ de manera que este acepte las ejecuciones de $\mathcal{A}$ que no satisfacen ϕ. De esta manera, la pregunta de si se tiene que $\mathcal{A} \models \phi$ se reduce a comprobar si el lenguaje reconocido por dicho autómata combinado es vacío.

Veamos a continuación un ejemplo: Sea ϕ la fórmula que afirma que tras cada ocurrencia de P debe aparecer una ocurrencia de Q, lo cual en $PLTL$ se escribe como $\mathbb{G}(P \Rightarrow \mathbb{X}\mathbb{F}Q)$. Por tanto, $\neg\phi$ significa que existe una ocurrencia de P tras la cual nunca aparecerá Q, y podemos representar $\mathcal{B}_{\neg\phi}$ del siguiente modo:

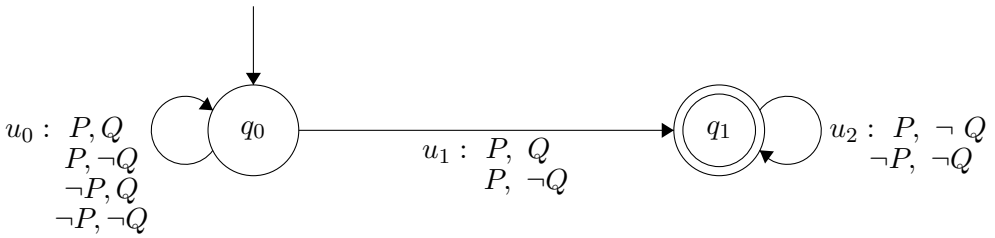

Figura 4.1: Autómata de comprobación en PLTL

Cabe destacar que este autómata no es determinista, pues en cualquier momento, partiendo de q_0 y de manera arbitraria, si se tiene P puede cambiar del estado q_0 al estado q_1, estado donde Q no se tiene nunca más. De esta manera observamos que $\mathcal{B}_{\neg\phi}$ caracteriza nuestra fórmula de $PLTL$. Consideremos ahora el autómata $\mathcal{A}$ de la fig. 4.2.

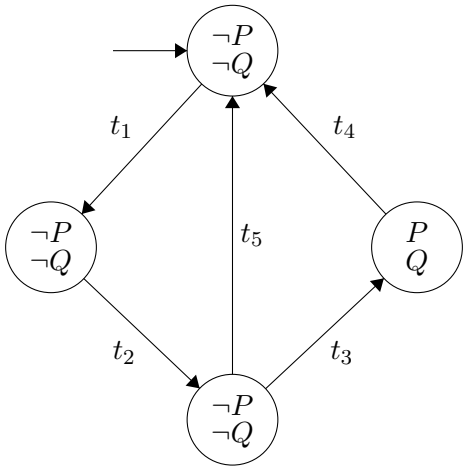

Figura 4.2: Autómata para PLTL

Si quisiéramos comprobar si $\mathcal{A} \models \mathbb{G}(P \Rightarrow \mathbb{X}\mathbb{F}Q)$, tendríamos que considerar el autómata $\mathcal{A} \otimes \mathcal{B}_{\neg\phi}$ resultante de la sincronización de los dos, teniendo en cuenta que las proposiciones representan los estados de $\mathcal{A}$ por una parte, y las transiciones de $\mathcal{B}_{\neg\phi}$ por otra. En lo referente a la parte de $\mathcal{B}_{\neg\phi}$, la transición $t \otimes u_1$ ocurrirá únicamente si t parte de un estado donde se satisface P, mientras que una transición de la forma $t \otimes u_2$ sólo se tendrá si t partiera de un estado donde se satisficiera $\neg Q$. Podemos observar la sincronización completa en la fig. 4.3.

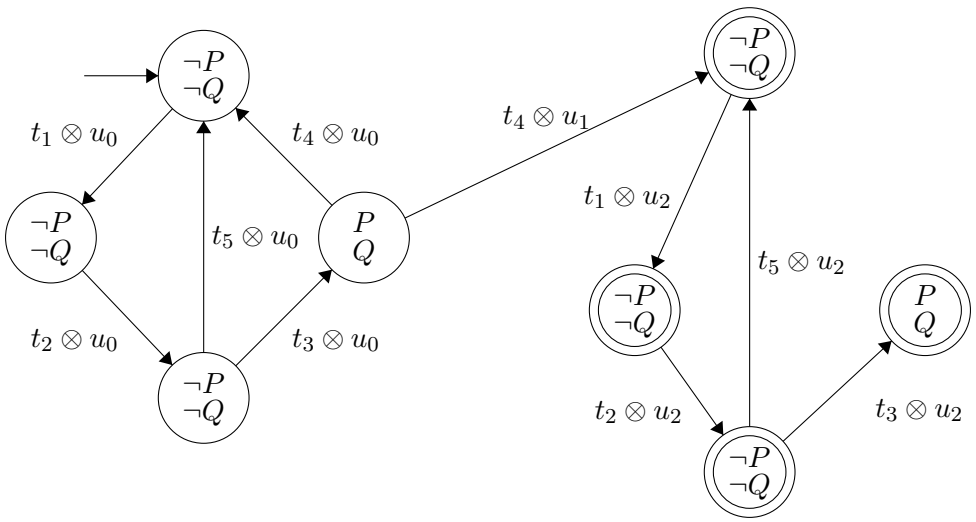

Figura 4.3: Sincronización de ambos autómatas

Observemos que hay comportamientos de $\mathcal{A}$ que son aceptados por $\mathcal{A} \otimes \mathcal{B}_{\neg\phi}$, es decir, que existen ejecuciones que o bien una vez se completen o bien de manera indefinida a partir de una de sus transiciones, en las que el estado final sobre el autómata será uno de los estados finales (aquellos marcados con doble círculo). De esta manera podemos comprobar que el lenguaje que el autómata sincronizado reconoce es no vacío y por tanto podemos afirmar que $\mathcal{A} \not\models \phi$.

Hemos querido reducir el algoritmo de *Model checking* para $PLTL$ a un ejemplo por

su naturaleza compleja, especialmente en lo referente a la parte de construir $\mathcal{B}_{\neg\phi}$. De todas maneras cabe destacar que de forma general dicho autómata deberá reconocer palabras no finitas, como por ejémplo los autómatas de *Büchi*, los cuales son un tipo de $\omega-$autómatas que extienden el concepto de autómata finito determinista a ejecuciones (*inputs*) de tipo infinito. En cualquier caso, desde el punto de vista de la complejidad, los principales puntos a tener en cuenta sobre el algoritmo son:

- En el peor de los casos, $\mathcal{B}_{\neg\phi}$ tendrá a lo sumo $O(2^{|\phi|})$
- El producto $\mathcal{A} \otimes \mathcal{B}_{\neg\phi}$ tiene tamaño $O(|\mathcal{A}| \times |\mathcal{B}_{\neg\phi}|)$
- Si $\mathcal{A} \otimes \mathcal{B}_{\neg\phi}$ cabe en memoria, podemos determinar si acepta un lenguaje no vacío en tiempo $O(|\mathcal{A}| \times 2^{|\phi|})$

En cierto sentido, podemos decir que el autómata $\mathcal{B}_{\neg\phi}$ *observa* el comportamiento del autómata $\mathcal{A}$ una vez estos están sincronizados. En conclusión, sabemos que la pregunta "$\mathcal{A} \models \phi$?" puede ser respondida en tiempo $O(|\mathcal{A}| \times 2^{|\phi|})$.

4.3– El problema de explosión de estados

El principal inconveniente a la hora de construir algoritmos de *model checking* viene derivado del hecho de que dichos algoritmos se basan en la construcción explícita del autómata $\mathcal{A}$ que tenemos como objetivo a verificar, el cuál en el caso de CTL debemos someter a una serie de transformaciones y marcados, y en el caso de $PLTL$ debemos sincronizar con $\mathcal{B}_{\neg\phi}$ y estudiar los estados alcanzables. En la práctica, el número de estados de $\mathcal{A}$ suele ser de un orden bastante alto, de manera que si $\mathcal{A}$ proviene por ejemplo de la sincronización de los autómatas $\mathcal{A}_1, \ldots, \mathcal{A}_n$, el tamaño de $\mathcal{A}$ será del orden de $|\mathcal{A}_1| \times \cdots \times |\mathcal{A}_n|$, lo cuál representa potencialmente un crecimiento exponencial sobre el número de estados de un autómata. Para este tipo de casos, existen métodos que son capaces de realizar técnicas de *model checking* sobre algunos de estos autómatas de forma automática, sin necesidad de su construcción explícita. No obstante, dichos métodos quedan fuera de nuestro alcance en lo que al objeto de este estudio se refiere.

CAPÍTULO 5

Implementación

El objetivo de este último capítulo es reflejar y explicar las funcionalidades del trabajo que se ha venido realizando para la implementación de los conceptos de los capítulos anteriores, además de poder servir de guía al lector en caso de que quisiera también desarrollar una librería en *Haskell*.

Es importante aclarar que la intención desde el primer momento fue crear algo útil y funcional, y por ello una de las dificultades del desarrollo ha sido mantener un equilibrio entre funcionalidades que generalicen y que puedan servir siempre de utilidad al usuario, y funcionalidades suficientemente específicas como para sernos útiles durante el desarrollo de los distintos módulos. Una consecuencia de esto ha sido la decisión de mantener los módulos lo más independientes posible entre sí, de forma que si un usuario encuentra útil para su trabajo uno de ellos, no se vea forzado ni condicionado a usar el resto de módulos si no le es necesario. Para poder usar la librería, una vez instalado nuestro entorno de trabajo en *Haskell*, únicamente necesitaremos ejecutar la orden de `cabal install FSM` y la librería que en este capítulo estamos presentando quedará instalada en el sistema.

5.1– Creación de la librería

Uno de los primeros retos a la hora de programar en cualquier lenguaje es definir apropiadamente un entorno y metodología de trabajo, de forma que el desarrollador se sienta cómodo y ágil a la hora de crear código. En mi experiencia y opinión personal, en el caso concreto de *Haskell*, los *Interface Development Environment* o *IDE* que hay disponibles en ocasiones resultan difíciles de configurar apropiadamente, por lo que mi decisión personal fue hacer uso de un editor de texto sencillo para luego realizar las pruebas sobre la terminal. En concreto, el trabajo fue realizado por completo sobre una máquina con Manjaro Linux 4.19.

5.1.1. Instalación y gestión del entorno

Aunque quizás la herramienta más conocida como gestor de paquetes en *Haskell* sea *cabal*, encontré que en ciertos sistemas operativos creaba una serie de dificultades relacionadas con los vínculos simbólicos o *symlinks*, por lo que mi decisión final fue hacer

uso del gestor de paquetes *stack*. En la documentación del mismo se puede encontrar una guía paso a paso de cómo construir y gestionar un paquete.

5.1.2. Integración contínua

La herramienta por excelencia para la gestión de código, y de la que se hizo uso en este proyecto es GitHub. Aunque no ha sido un proyecto en el que hayan intervenido en el desarrollo varias personas, sí que ha sido realmente útil a la hora de llevar un control de versiones del código, gestionar copias de seguridad en caso de pérdida de archivos y finalmente permitir la posibilidad de que cualquier persona pueda contribuir o modificar el trabajo realizado.

5.1.3. Subida al repositorio

Hackage es el principal repositorio de paquetes de *Haskell*. En él se almacenan todos los paquetes de forma que con un simple *cabal install nombre del paquete* (o *stack* en su caso), el paquete queda instalado en nuestro sistema y podemos hacer uso del mismo. Cabe destacar que para poder subir paquetes al repositorio principal, es necesario que solicitemos ser un desarrollador verificado, para lo que necesitaremos que nuestro paquete cumpla ciertos requisitos de calidad de forma que el repositorio siga siendo estable.

Además, resulta muy recomendable que a nuestro paquete añadamos la mayor cantidad de documentación posible. Para incluir dicha documentación, basta con que añadamos comentarios en el propio código siguiendo la estructura del lenguaje de marcado *Haddock*, de forma que una vez hayamos completado la guía de nuestro paquete bastará con que ejecutemos la orden de *stack haddock* para que se genere la documentación.

Por último, y quizás más importante, quería destacar la posibilidad de *Hackage* de crear *paquetes candidatos*. Esta funcionalidad nos permite desarrollar nuestro paquete en un entorno más tipo pre-repositorio oficial, de forma que podemos borrar o modificar el paquete sin problema, mientras que una vez subido al repositorio principal, no puede ser modificado. Como regla general, recomiendo realizar todo el desarrollo en la sección de candidatos, hasta llegar a una versión funcional del desarrollo donde ya nos sintamos suficientemente cómodos como para llevar nuestro paquete al repositorio principal. Además, esta funcionalidad nos permitirá familiarizarnos con *Hackage*, de forma que disminuye los errores que podamos cometer sobre el repositorio principal.

5.1.4. Guía práctica

Veamos un ejemplo práctico donde crearemos una librería sencilla desde cero. Para ello, es necesario que el lector realice algunos cambios acorde al sistema en el que esté trabajando. Un requerimiento básico durante toda la guía será el uso de un sistema `UNIX`, como por ejemplo MacOS, Linux Ubuntu, Archlinux, etc. Es probable que cualquier usuario de linux tenga alguna experiencia previa con el gestor de paquetes del sistema, pero dado que en MacOS es menos frecuente la necesidad de usar el mismo, recomiendo al usuario que si está usando este OS, utilice el gestor de paquetes `brew`.

El primer paso para poder crear una librería es instalar en tu sistema, haciendo uso de la terminal y tu gestor de paquetes (`brew` en MacOS, `apt` en Ubuntu, `pacman` en Arch...) `ghc`, que es el compilador de *Haskell* y `stack`, que será el gestor de paquetes que usaremos. Alternativamente, puedes usar `cabal-install` como gestor de paquetes, si se

adecua mejor a tus necesidades. Para comprobar que la instalación ha tenido éxito, puedes ejecutar `stack ghci` en la terminal para activar el modo interactivo, y si todo ha ido bien aparecerá un *REPL* o *Read, Evaluate, Print and Loopback*, donde puedes ejecutar código de *Haskell* línea a línea, pero no escribir código propiamente dicho.

Una vez llegado a este punto, que seguramente sea el que traiga más quebraderos de cabeza en cuanto a instalación, lo siguiente es crear el paquete. Para ello, basta con que creemos un directorio donde vayamos a almacenar todo el código fuente de nuestra librería, nos movamos desde la terminal hasta ese directorio haciendo uso del comando `cd` y ejecutemos `stack new nombre-de-tu-paquete`. Este comando creará un sistema de carpetas y archivos, donde los más relevantes son los siguientes:

- `src/`: Dentro de esta carpeta, almacenaremos los módulos de nuestra librería que queramos crear. Por ejemplo, si nuestra librería es sobre funciones matemáticas y queremos crear dos "ramas" de la librería una para funciones reales y otra para funciones complejas; dentro de las cuales queremos crear módulos para funciones polinomiales y no polinomiales, entonces dentro de `src` crearíamos dos carpetas, una llamada `RealFuncts` y otra `ComplexFuncts`, y dentro de cada una crearíamos `Poly.hs` y `NoPoly.hs`, de forma que cuando los usuarios hagan uso de la librería, puedan hacer `import RealFuncts.Poly` para importar uno de los módulos de una rama.

- `app/Main.hs`: Es el lugar donde creamos nuestro fichero principal para que luego la librería se pueda instalar correctamente. En este fichero bastará con que modifiquemos `Lib` con cada uno de los paquetes que queramos que los usuarios de nuestra librería puedan usar activamente, y si no queremos que nada sea ejecutable, podemos cambiar `somefun` por `return ()`. Cabe destacar que es posible que creemos módulos que únicamente se usen internamente dentro de la librería, que a priori no queremos que sean con los que el usuario interaccione. Estos módulos no los incluiríamos en `Main.hs`

- `nombre-de-tu-paquete.cabal`: Por último, y quizás más importante, se genera en el directorio raíz de tu paquete el fichero de configuración. Es en este fichero donde se almacenan los parámetros que definen correctamente la licencia, versión, dependencias, autoría, etc, de tu paquete, y es importante que dediquemos tiempo a configurarlo adecuadamente según nuestras necesidades.

Para la elaboración de los módulos, bastará con que creemos archivos terminados en `.hs` con nuestro editor de texto de preferencia, y realicemos las pruebas de las funciones que construyamos ejecutando el entorno interactivo `stack ghci`. Es importante que los módulos empiecen siempre con la directiva `module`. Por ejemplo, si seguimos el ejemplo anterior, `module RealFuncts.Poly where` (cambiando `RealFuncts.Poly` por tu rama y módulo concreto). Es esta sentencia la que hace que el módulo pueda ser importado e instalado posteriormente. Además, es probable que en el módulo definamos una serie de estructuras, tipos abstractos de datos o funciones que queramos que puedan ser utilizados. Para ello, debemos incluir el nombre de estas estructuras entre paréntesis de la siguiente manera:

```
module FSM.Logic (

    -- * Implementation of the model checking algorithm for CTL
    checkCTL,
    modelsCTL,
    checkFormulas

) where
```

Código 5.1: ejemplo Module

A partir de esta sentencia es donde importaremos las librerías que necesitemos en nuestro código, crearemos las funcionalidades que consideremos, etc. Las líneas que comienzan con doble guión, además de un comentario típico de *Haskell*, supone una indicación para la construcción de una sección en *Haddock*, que veremos más adelante.

Una vez nos sintamos conformes con una primera versión de nuestra librería, habiendo realizado una configuración inicial, deberemos hacer dos cosas. La primera es ejecutar la orden `stack sdist`, que creará una versión comprimida de toda la librería. Es importante ejecutar esta opción porque además de ello se hacen una serie de comprobaciones sobre el funcionamiento y la estructura de la librería, y si no pasa estos filtros, no podremos subir nuestra librería a *Hackage*. La segunda de las cosas que debemos hacer es llevar un correcto control de versiones de nuestra librería. Para ello, la mejor opción sin duda es usar *GitHub* como repositorio personal de nuestro código. Para ello, lo único que necesitamos es ir a *GitHub*, crearnos una cuenta e iniciar un nuevo repositorio. Es importante que, al menos por simplicidad, el repositorio no sea privado.

Una vez hecho esto, volvemos a la terminal, y desde el directorio raíz de nuestra librería ejecutamos sucesivamente los siguientes comandos:

- `git init`: Esto nos inicializa un repositorio vacío en nuestro sistema de carpetas de la librería. Solamente necesitaremos ejecutar este comando la primera vez que creemos el repositorio.

- `git add .`: Este comando es la orden para añadir ficheros al repositorio. El punto indica que queremos añadir todos, no obstante si quisiéramos podríamos indicar diferentes archivos, carpetas, ...

- `git commit -m "comentario"`: Este comando añade un comentario a los cambios que vamos a realizar en nuestro repositorio. Podemos sustituir `"comentario"` por algo más descriptivo como `"cambio en la funcionalidad x"`. Aunque sencillo, este paso es imprescindible.

- `git remote add origin git@github.com:username/new-repo`: con este comando, vinculamos el repositorio que hemos creado en la web con el que tenemos en nuestro ordenador. Deberemos cambiar `username` y `new-repo` con nuestro nombre de usuario y nuestro repositorio. Al igual que con `git init`, este comando solamente necesitaremos ejecutarlo la primera vez que estemos subiendo archivos al repositorio.

- `git push -u origin master`: Este es el comando final que necesitaremos para subir los cambios a *GitHub*.

Idealmente, cuando se trabaja con este método de control de versiones, la mejor práctica es crear una rama a partir de la rama principal de nuestro código donde desarrollemos una nueva funcionalidad y una vez esta funcionalidad se pueda implementar de forma estable y segura con el código principal, crear un *pull request* para unir esta rama secundaria con la rama principal o *master*, no obstante con los comandos anteriormente mencionados nos será suficiente.

Como nota adicional, todos los repositorios de *GitHub* se inicializan por defecto con un archivo que se llama `README.md`. Este archivo en formato *Markdown* es el que se espera que rellenemos con una información genérica sobre el contenido del repositorio, cómo comprender la estructura de su contenido, etc. Algo para entender qué es el repositorio, pero sin entrar totalmente en todos los detalles.

El último paso para que nuestra librería esté totalmente funcional y poder subirla a *Hackage* es añadir descripciones y comentarios a modo de documentación y guía para que el usuario de nuestra librería sepa hacer uso de ella apropiadamente. Para ello, usaremos el lenguaje de marcado conocido como *Haddock*. Los lenguajes de marcado son lenguajes que no son de programación propiamente dichos que puedan generar funciones o aplicaciones, si no que se utilizan para tomar notas dentro o en torno al código que desarrollemos de forma que sirvan de apoyo al mismo.

Si alguna vez has programado en *Haskell*, sabrás que el doble guión sirve para "ignorar" o comentar una línea. Pues bien, para generar la documentación bastará con que tras estos dos guiones añadamos un símbolo apropiado, para que *Haddock* pueda construir secciones y descripciones a partir de ellos. En mi caso, me bastó con las dos instrucciones siguientes.

- `-- *`: Como se pudo ver en el *ejemplo Module*, este inicio de comentario marca una sección, que deberemos incluir al comiendo de nuestro fichero. Con ello, *Haddock* construirá un índice dentro de nuestro código.

- `-- |`: Este inicio de línea marca un párrafo. Mi recomendación es que añadamos un párrafo antes de cada función o funcionalidad de nuestro código para explicar su uso y funcionamiento.

Y con esto debería bastar. Si tienes dudas sobre cómo usar *Haddock*, o cómo organizar tu librería, mi recomendación es que explores *Hackage* y visualices el código fuente de alguna librería (clickando en `source` arriba a la derecha). Ahí podrás ver ejemplos de cómo está construida la documentación y estructurada esa librería en concreto.

Para terminar, generaremos nuestra documentación usando `stack haddock` y subiremos nuestro código fuente a los candidatos de *Hackage*. Para subir nuestro candidato, bastará con que subamos el archivo comprimido que se genera al ejecutar `stack sdist`. No obstante, en los candidatos no podremos visualizar la documentación de nuestra librería a no ser que la subamos manualmente, para lo que deberemos navegar en la terminal hasta el directorio donde se ha almacenado la documentación (que se nos indica al ejecutar `stack haddock`) y ejecutar los siguientes comandos:

```
tar -c -v -z --format=ustar -f PACKAGE-VERSION-docs.tar.gz PACKAGE-VERSION-docs

curl -X PUT -H 'Content-Type: application/x-tar' -H 'Content-Encoding: gzip' --
    data-binary "@PACKAGE-VERSION-docs.tar.gz" "https://USERNAME:PASSWORD@hackage
    .haskell.org/package/PACKAGE-VERSION/candidate/docs"
```

Código 5.2: subir Docs Hackage candidates

Cambiando `PACKAGE-VERSION` por el nombre de nuestra librería y su versión y `USERNAME` y `PASSWORD` por nuestras credenciales en *Hackage*. De esta forma, deberíamos poder visualizar nuestra librería en el apartado *candidates* de *Hackage*, y con esto finalizaría el proceso de creación de la primera versión de la librería.

5.2– Primer módulo - Automata

En este módulo, el cual en cierto modo establece una base conceptual para el trabajo posterior, desarrollamos los conceptos básicos en torno al capítulo primero. En él, se presentan funciones para crear, modificar y obtener información del autómata.

El módulo comienza creando el tipo abstracto de dato *Automata*, el cuál consiste, al igual que en la definición formal del capítulo primero, en una tupla donde se almacenan las distintas estructuras que componen el autómata. El código es como sigue:

```
data Automata = A (Set Int,Set Char,Int,M.Matrix Int,Set Int)
```

Código 5.3: Creación del tipo Automata

Siendo A el constructor, y los elementos de la tupla el conjunto de estados, el alfabeto que acepta el autómata, el estado inicial, la matriz de transición de estados y el conjunto de estados de aceptación respectivamente. Es de observar que el tipo `Automata` no tiene asociado en su construcción una instancia para mostrar dicho tipo, sino que hemos creado una representación específica para el tipo de forma que el usuario pueda ver la información del mismo de manera más cómoda.

5.2.1. Funciones de creación

La única función de creación que hemos desarrollado ha sido la de *createAutomata*. Para mayor sencillez, hemos hecho que la función en lugar de tomar un conjunto de estados para construir los estados del autómata, tome simplemente un entero positivo que representa la cantidad de estados del autómata y construya a partir de éste el conjunto de estados. La función es como sigue:

```

createAutomata :: Int -> String -> Int -> M.Matrix Int -> [Int] -> Automata
createAutomata s i s0 m a
    | s < 1 =
        error "Number of states must be greater than 1"
    | not (member s0 s') =
        error "Not valid initial state"
    | not ((M.nrows m,M.ncols m) == (size s',size i')) =
        error "Not valid matrix size"
    | not (isSubsetOf (delete 0 (fromList (M.toList m))) s') =
```

```
11          error "Not valid matrix elems"
12      | not (isSubsetOf a' s') =
13          error "Not valid accepting states"
14      | otherwise = A (s',i',s0,m,a')
15        where s' = fromList [1..s]
16              i' = fromList (L.sort i)
17              a' = fromList (L.sort a)
```

Código 5.4: función createAutomata

5.2.2. Funciones de acceso

En esta parte hemos creado numerosas funciones que, haciendo uso intensivo de la capacidad de *Haskell* de comparación de patrones o *pattern matching*, permiten obtener información relativa al autómata. Para comodidad de lectura de esta memoria, no incluiré el código salvo que sea de especial interés en el desarrollo de la misma, si no que éste puede encontrarse en los anexos I, II, III. También puede encontrarse completo tanto en el repositorio de GitHub como en Hackage.

- getStates: Esta función nos devuelve el conjunto de estados del autómata.
- getAcceptingStates: Esta función nos devuelve la lista de los estados de aceptación del autómata.
- getInitialState: Esta función nos devuelve el estado inicial del autómata.
- getInputs: Esta función nos devuelve el alfabeto o *inputs* que el autómata acepta.
- getAssociations: Esta función nos devuelve la matriz de transición de estados del autómata. Esta matriz se construye de forma que tiene tantas filas como estados tenga el autómata, y tantas columnas como caracteres tenga el alfabeto del autómata. Cada fila de la matriz representa a qué estado se mueve el autómata partiendo del "estado fila" si recibe el carácter de su correspondiente columna. En caso de que en ese estado no se acepte la transición de algún carácter, por convención pondremos un 0.
- getTransitions: Esta función toma como entrada un objeto tipo Automata y un estado, y nos devuelve los posibles caracteres o *inputs* que ese estado acepta.
- getOutgoingStates: Esta función, a partir de un estado y un Automata, nos devuelve los posibles estados que son alcanzables en el autómata a partir del estado dado (en un paso).
- getIncomingStates: Esta función nos devuelve los estados que alcanzan en un paso a otro estado dado.
- getDeadlocks: Esta función nos devuelve los estados que una vez alcanzados, el autómata se "bloquea" o llega a un punto muerto a partir del cual no puede transicionar a ningún otro estado.
- getIsolated: Esta función nos devuelve los estados aislados del autómata, es decir, que no pueden ser alcanzados por ningún otro estado.

Veamos un ejemplo de uso de estas funciones. Para ello, consideremos el siguiente autómata:

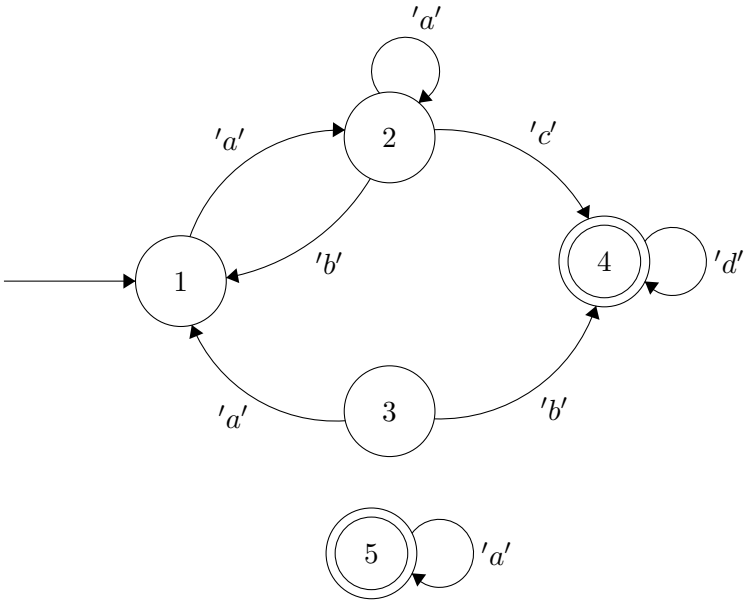

Figura 5.1: Autómata de ejemplo

El código para crear dicho autómata es como sigue:

```
import qualified Data.Matrix as M
import FSM.Automata

mat = M.fromLists [[2,0,0,0],[2,1,4,0],[1,4,0,0],[0,0,0,4],[5,0,0,0]]
tom = createAutomata 5 ['a', 'b', 'c', 'd'] 1 mat [4,5]

getOutgoingStates tom 2 --output: fromList [1,2,4]
getIncomingStates tom 4 --output: fromList [2,3,4]
getDeadlocks tom --output: fromList [4,5]
getIsolated tom --output: fromList [3,5]
```

Código 5.5: función createAutomata

5.2.3. Funciones de comprobación

En esta parte, únicamente hemos definido una función, llamada validInput, que dada una cadena finita de caracteres y un autómata, comprueba si tras haber realizado las transiciones a lo largo de la cadena terminamos dichas transiciones en un estado de aceptación del autómata. Cabe destacar que es necesario que la cadena siga una progresión lógica por los estados. Veamos su funcionamiento con una serie de ejemplos, en los que nos apoyaremos en el autómata definido en la sección anterior:

```
validInput "aabac" tom --output: True
validInput "c" tom --output: Error (debido a que en el estado incial no se
    considera como input el caracter 'c')
validInput "abacd" tom --output: True
```

```
validInput "e" tom --output: Error (debido a que no esta en el alfabeto)
validInput "aba" tom --output: False
```

Código 5.6: ejemplo validInput

5.2.4. Funciones de edición

En esta última parte del primer módulo, podemos encontrar una serie de funciones que nos permiten modificar la información que se almacena en el autómata. Las funciones son:

- addState: A partir de un autómata y una lista de enteros que representa la fila de la matriz de transición de estados correspondiente. No es necesario aportar el número del estado pues la función ordenará automáticamente los estados.
- deleteState: Esta función elimina un estado dado, modificando la matriz de transición de estados de forma que se renombran los estados que fueran mayor que el mismo.
- changeInitialState: Esta función nos modifica el estado inicial de un autómata.
- addAcceptingState: Añade un estado al conjunto de estados de aceptación de un autómata.

Veamos un ejemplo del funcionamiento de la tercera función, *deleteState*. La matriz del autómata de ejemplo de la figura 5.1 es de la forma

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 2 & 1 & 4 & 0 \\ 1 & 4 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 5 & 0 & 0 & 0 \end{pmatrix}$$

Mientras que si ejecutamos el siguiente código:

```
deleteState tom 2
```

Código 5.7: ejemplo deleteState

La matriz que nos queda es la siguiente:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & 0 & 0 & 3 \\ 4 & 0 & 0 & 0 \end{pmatrix}$$

Se puede observar que la primera fila, que representa al estado 1 y que es la única que estaba "por debajo de" el estado 2, simplemente ha perdido la asociación con el estado 2. Por otra parte, el resto de filas han perdido las asociaciones que pudieran tener con el estado 2, y aquellos estados mayores que 2 se han reasignado apropiadamente.

5.3– Segundo módulo - States

En este segundo módulo se desarrolla la estructura necesaria para almacenar información en los estados del autómata. Aunque la idea intuitiva sería crear el par (`Automata`, `AutomataInfo`), de nuevo esto nos obligaría a hacer uso del primer módulo, por lo que decidí no hacerlo para dar mayor libertad al usuario. La idea intuitiva de este módulo gira en torno a crear una estructura tipo `clave:valor`, similar a los diccionarios de `Python`, para almacenar la información que nos interese sobre los estados. Al igual que en el módulo anterior, en este también hemos añadido funciones específicas para visualidar las estructuras que se definen en este módulo.

5.3.1. Consideraciones previas

En torno a este módulo, se nos han planteado dos dificultades principalmente. Por un lado, queríamos que el usuario pudiese almacenar cualquier tipo de estructura en los estados, ya fueran números, cadenas de caracteres, matrices, etc, mientras que por otro lado también queríamos añadir la funcionalidad de que se pudiera acceder fácilmente a esta información y modificarla.

Para solventar el primer problema, en la propia documentación de la librería se insta al usuario a que cree su propio tipo abstracto de dato, de forma que asocie cada constructor de ese tipo con los posibles tipos de datos que quiera almacenar en los estados. Por ejemplo, si quisiéramos almacenar en los estados cadenas y listas de números, construiríamos el siguiente TAD:

```
1 data CustomData = Type1 String | Type2 [Int] deriving (Show,Eq)
```

Código 5.8: custom TAD

Es de vital importancia que el TAD que construyamos incluya `deriving (Show,Eq)`, pues es necesario para el correcto funcionamiendo de las funciones que se definen en el módulo.

5.3.2. La estructura de AutomataInfo

La estructura de `AutomataInfo` nace de la intención de solventar la segunda dificultad mencionada en la sección anterior. Así, con el fin de añadir la posibilidad de acceder y modificar la información que almacenaremos en los estados de la forma más flexible posible, hemos construido `AutomataInfo` de forma que se apoya fuertemente en la estructura de la librería `Data.Map`. De esta forma, hemos creado un `Map` de dos niveles de profundidad, donde el primer nivel tiene parejas de (`State`,`StateInfo`). Es decir, este primer nivel es una estructura que nos asocia cada estado con la información que se almacena en dicho estado. A su vez, `StateInfo` es una estructura también de parejas de la forma (`Tag`,`a`), donde `Tag` es una etiqueta de tipo `String` y `a` sería el TAD que creamos en la sección anterior. Aunque para nuestro uso esta estructura no es imprescindible, pues no necesitamos ni de las etiquetas ni del TAD que nos incluya nuestra información, si un usuario quisiera almacenar en un estado el conjunto {3,5,“a”}, siendo por ejemplo 3 un contador que almacena éxitos, 5 un contador que almacena fracasos y “a” una orden arbitraria, y luego quisiera modificar estos valores, no podría hacerlo sin almacenar constantemente el valor previo de cada contador.

Sin embargo, con la estructura que hemos creado, la creación del TAD nos permite almacenar distintas estructuras dentro del mismo sitio, ya sea en una lista, en un conjunto como el ejemplo anterior o en un `Map` como es en nuestro caso. Además, gracias a las etiquetas, podemos acceder y modificar dicha información de forma intuitiva y directa.

5.3.3. Funciones de creación

Para la creación de la estructura `AutomataInfo`, ofrecemos dos funciones básicas.

- createStateInfo: Esta función crea la mínima expresión de la estructura `AutomataInfo`, partiendo de un estado, una etiqueta y un valor nos crea la información de un único estado. El principal uso de esta función es para la recursión de la siguiente función.
- fromlsStateInfo: A partir de un estado, una lista de pares de etiquetas y valores y o bien `Nothing` o bien `Just info`, siendo `info` nuestro `AutomataInfo`, nos creará el objeto final. En el primer caso, se crea desde cero mientras que en el segundo caso o bien añade la información relativa a un estado si este no existía previamente, o bien la actualiza si existía, realizando la correspondencia con las etiquetas.

Veamos un ejemplo sencillo a partir de `myCustomData`:

```
1 import FSM.States
2
3 info1 = fromlsStateInfo 1 [("tag1", Type1 "r")] Nothing
4 info2 = fromlsStateInfo 2 [("tag1", Type1 "r")] (Just info1) -- incluye la info
      al estado 2
5 info1 = fromlsStateInfo 1 [("tag1", Type2 [3]),("tag2",Type1 "hi")] (Just info2)
      -- nos modifica la informacion del estado 1
```

Código 5.9: ejemplo fromlsStateInfo

5.3.4. Funciones de acceso

Con motivación similar a las funciones creadas en el módulo anterior, estas funciones nos dan información sobre nuestra estructura `AutomataInfo`.

- getStateInfo: Esta función sencilla nos devuelve la estructura `Map` almacenada en un estado.
- getStatesWithInfo: Nos devuelve aquellos estados que actualmente contienen algún tipo de información.
- getTagsInState: Nos devuelve las etiquetas que se encuentran en un estado dado.
- getInfoInState: Esta función toma como valores un `AutomataInfo`, un estado y o bien una etiqueta o bien `Nothing`, de forma que nos devuelve la información dentro de dicha etiqueta o toda la información dentro del estado, respectivamente.

Veamos su funcionamiento con un ejemplo, haciendo uso de `info1` del ejemplo anterior:

```
getInfoInState info2 1 (Just "tag1") --output: Type1 "r"
getInfoInState info1 1 Nothing -- output: Type1 "hi" y Type2 [3]
```

Código 5.10: ejemplo getInfoInState

5.3.5. Funciones de edición

Para la edición de la información que se almacena, hemos definido dos funciones principales:

- alterStateInfo: Esta función parte de un estado, bien una etiqueta o bien `Nothing`, una nueva información y una estructura de `AutomataInfo` y, en el caso de que hayamos dado una etiqueta modifica el valor de la etiqueta si existía en el estado, o la añade si no existía, mientras que si recibe `Nothing` elimina toda la información del estado.
- unionStateInfo: Por último, esta función combina los diccionarios almacenados en dos `AutomataInfo` de una forma sesgada por la izquierda, dando preferencia a la información almacenada en el `AutomataInfo` que procede del primer argumento.

Veamos un ejemplo de estas funciones:

```
info2 = alterStateInfo 1 Nothing (Type1 "ho") info1 --output: elimina el estado 1
info3 = alterStateInfo 2 (Just "tag1") (Type1 "ho") info1 --ouput: modifica el
    estado 2
info4 = alterStateInfo 3 (Just "tag1") (Type1 "ho") info3 --output: crea un nuevo
     estado, y pone una nueva etiqueta

unionStateInfo info1 info4 --output: en el estado 2, por estar en conflicto, se
    queda con info1, e incluye el estado 3
```

Código 5.11: ejemplo funciones edición States

5.4– Tercer módulo - Logic

En este tercer y último módulo hemos creado el tipo abstracto de dato `CTL` en el que implementamos el lenguaje temporal del mismo nombre, y del mismo modo el tipo abstracto de dato `LTL`. El objetivo de este módulo es hacer uso conjunto de las funcionalidades de los dos módulos anteriores para implementar el algoritmo de model checking para `CTL` que desarrollamos en el capítulo 4, añadiendo el resto de casos que no se consideran entre los seis casos base del algoritmo a través de equivalencias. Para comprobar que el algoritmo funciona apropiadamente, plantearemos una serie de ejemplos.

5.4.1. Ejemplo 1

Sobre este autómata, vamos a comprobar las siguientes fórmulas:

- $p \wedge q$
- $\neg r$

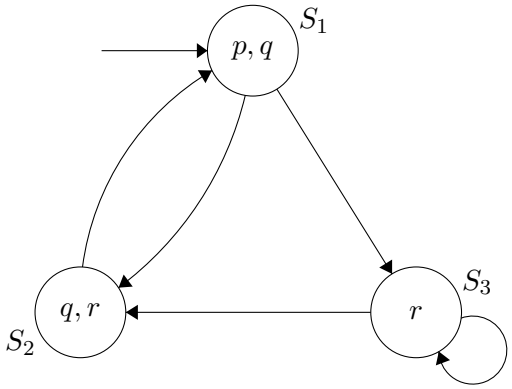

Figura 5.2: Ejemplo primero checkCTL

- $\mathbb{EX}(q \wedge r)$: Que puede leerse como "en alguno de los siguientes se tiene $q \wedge r$".
- $\neg\mathbb{AX}(q \wedge r)$: Que puede leerse como "no siempre en los siguientes se tiene que $q \wedge r$".
- $\neg\mathbb{EF}(p \wedge r)$: Que puede leerse como "en ningún estado futuro se tiene $q \wedge r$".
- $\mathbb{EG}r$: Que puede leerse como "existe un camino donde siempre se tiene r".
- $\mathbb{AF}r$: Que puede leerse como "en todos los caminos, en algún punto se tiene r".
- $\mathbb{E}(p \wedge q)\mathbb{U}r$: Que puede leerse como "existe un camino donde se tiene $p \wedge q$ hasta que se tiene r".
- $\mathbb{A}p\mathbb{U}r$: Que puede leerse como "en todos los caminos se tiene p hasta que se tiene r".
- $\mathbb{AG}(p \vee q \vee r \longrightarrow \mathbb{EF}\ \mathbb{EG}r)$: Que puede leerse como "en todos los caminos se tiene siempre que si se tiene $p \vee q \vee r$ entonces a partir de ese momento en algún punto se va a tener que siempre r".

Para ello, construimos el autómata y la información de los estados, y luego aplicamos la función de este módulo `modelsCTL` que nos dirá si el autómata modela la fórmula temporal.

```
1  import qualified Data.Matrix as M
2  import FSM.Automata
3  import FSM.States
4  import FSM.Logic
5
6  --Creamos el automata
7  mat = M.fromLists [[2,3],[1,3],[0,3]]
8  tom = createAutomata 3 ['a', 'b'] 1 mat []
9
10 --Creamos la informacion de los estados
11 info_state1 = fromlsStateInfo 1 [("atom1", Atom "p"),("atom2", Atom "q")] Nothing
12 info_state2 = fromlsStateInfo 2 [("atom1", Atom "q"),("atom2", Atom "r")] (Just
       info_state1)
13 info_state3 = fromlsStateInfo 3 [("atom1", Atom "r")] (Just info_state2)
14
15 --Usamos modelsCTL
16 modelsCTL (And (Atom "p") (Atom "q")) tom info_state3 --output: True
```

```
modelsCTL (Not (Atom "r")) tom info_state3 --output: True
modelsCTL (EX (And (Atom "q") (Atom "r"))) tom info_state3 --output: True
modelsCTL (Not (AX (And (Atom "q") (Atom "r")))) tom info_state3 --output: True
modelsCTL (Not (EF (And (Atom "p") (Atom "r")))) tom info_state3 --output: True
modelsCTL (EG (Atom "r")) tom info_state3 --output: False
modelsCTL (AF (Atom "r")) tom info_state3 --output: True
modelsCTL (EU (And (Atom "p") (Atom "q")) (Atom "r")) tom info_state3 --output:
    True
modelsCTL (AU (Atom "p") (Atom "r")) tom info_state3 --output: True
modelsCTL (AG (RArrow (Or (Or (Atom "p") (Atom "q")) (Atom "r")) (EF (EG (Atom "r
    ")))))) tom info_state3 --output: True
```

Código 5.12: ejemplo uno modelsCTL

5.4.2. Ejemplo 2

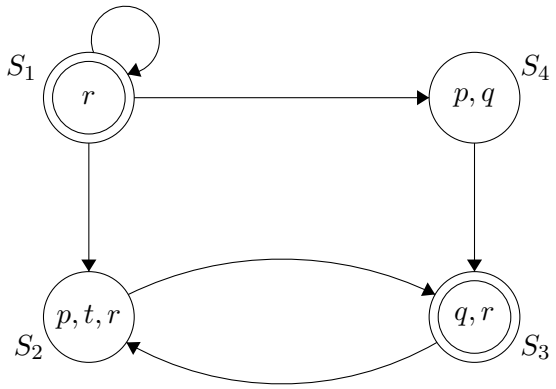

Figura 5.3: Ejemplo segundo checkCTL

Sobre este autómata, vamos a comprobar las siguientes fórmulas para el caso donde el estado inicial es S_1 y para el caso donde el estado inicial es S_3.

- $\neg p \longrightarrow r$
- $\mathbb{A}\mathbb{F}t$: Que puede leerse como "en todos los caminos, en algún punto se tiene t".
- $\neg\mathbb{E}\mathbb{G}r$: Que puede leerse como "no existe un camino donde siempre se tiene r".
- $\mathbb{E}\, t\mathbb{U}q$: Que puede leerse como "existe un camino donde se tiene t hasta que se tiene q".
- $\mathbb{A}\mathbb{F}q$: Que puede leerse como "en todos los caminos, en algún punto se tiene q".
- $\mathbb{E}\mathbb{F}q$: Que puede leerse como "en algún estado futuro se tiene q".
- $\mathbb{E}\mathbb{G}r$: Que puede leerse como "existe un camino donde siempre se tiene q".
- $\mathbb{E}\mathbb{G}(r \vee q)$: Que puede leerse como "en algún estado futuro se tiene $r \vee q$".

Para los casos donde queramos comprobar múltiples fórmulas, tenemos disponible el uso de la función recursiva `checkFormulas`.

```haskell
import qualified Data.Matrix as M
import FSM.Automata
import FSM.States
import FSM.Logic

--Creamos el automata
mat = M.fromLists [[1,2,4],[3,0,0],[2,0,0],[3,0,0]]
tom = createAutomata 4 ['a', 'b','c'] 1 mat [1,3]

--Creamos la informacion de los estados
info_state1 = fromlsStateInfo 1 [("atom1", Atom "r")] Nothing
info_state2 = fromlsStateInfo 2 [("atom1", Atom "p"),("atom2", Atom "t"), ("atom3
    ", Atom "r")] (Just info_state1)
info_state3 = fromlsStateInfo 3 [("atom1", Atom "q"),("atom2", Atom "r")] (Just
    info_state2)
info_state4 = fromlsStateInfo 4 [("atom1", Atom "p"),("atom2", Atom "q")] (Just
    info_state3)

--creamos la lista de formulas
formulas = [(RArrow (Not (Atom "p")) (Atom "r")),(AF (Atom "t")),
(Not (EG (Atom "r"))), (EU (Atom "t") (Atom "q")), (AF (Atom "q")),
(EF (Atom "q")), (EG (Atom "r")), (EG (Or (Atom "r") (Atom "q")))]

checkFormulas tom info_state4 formulas [] -- output: [True,False,False,False,
    False,True,True,True]
checkFormulas (changeInitialState tom 3) info_state4 formulas [] --output: [True,
    False,False,True,True,True,True,True]
```

Código 5.13: ejemplo dos modelsCTL

El motivo de los casos donde obtenemos falso puede verse a partir de los siguientes contrajemeplos:

- Para $\mathbb{AF}t$, al igual que para $\mathbb{AF}q$ tenemos el contraejemplo del camino que siempre permanece en el estado S_1
- Para $\neg\mathbb{EG}r$ tenemos el contraejemplo del camino S_1, S_2, S_3
- Para $\mathbb{E}t\mathbb{U}q$ vemos que no se tiene porque de primeras en el estado S_1 no se tiene t.

5.4.3. Ejemplo 3

Al igual que en el ejemplo anterior, vamos a comprobar las siguientes fórmulas para el caso donde el estado inicial es S_1 y para el caso donde el estado inicial es S_3.

- $\mathbb{AF}q$: Que puede leerse como "en todos los caminos, en algún punto se tiene q".
- $\mathbb{AG}\,\mathbb{EF}(p \vee r)$: Que puede leerse como "en todos los caminos se tiene siempre que en algún punto futuro se tiene $p \vee r$".
- $\mathbb{EX}\,\mathbb{EX}r$: Que puede leerse como "en alguno de los siguientes estados se tiene que en alguno de sus siguientes estados se tiene r".

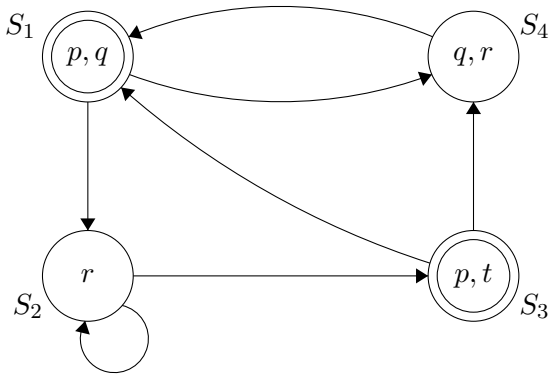

Figura 5.4: Ejemplo tercero checkCTL

- $\mathbb{A}\mathbb{G}\,\mathbb{A}\mathbb{F}q$: Que puede leerse como "en todos los caminos se tiene siempre que en en algún punto futuro en todos los caminos se tiene q".

```
1  import qualified Data.Matrix as M
2  import FSM.Automata
3  import FSM.States
4  import FSM.Logic
5
6  --Creamos el automata
7  mat = M.fromLists [[2,4],[2,3],[1,4],[1,0]]
8  tom = createAutomata 4 ['a', 'b'] 1 mat [1,3]
9
10 --Creamos la informacion de los estados
11 info_state1 = fromlsStateInfo 1 [("atom1", Atom "p"),("atom2", Atom "q")] Nothing
12 info_state2 = fromlsStateInfo 2 [("atom1", Atom "r")] (Just info_state1)
13 info_state3 = fromlsStateInfo 3 [("atom1", Atom "p"),("atom2", Atom "t")] (Just
       info_state2)
14 info_state4 = fromlsStateInfo 4 [("atom1", Atom "q"),("atom2", Atom "r")] (Just
       info_state3)
15
16 --creamos la lista de formulas
17 formulas = [(AF (Atom "q")),(AG (EF (Or (Atom "p") (Atom "r")))),
18 (EX (EX (Atom "r"))),(AG (AF (Atom "q")))]
19 checkFormulas tom info_state4 formulas [] -- output: [True,True,True,False]
20
21 checkFormulas (changeInitialState tom 3) info_state4 formulas [] --output: [True,
       True,True,False]
```

Código 5.14: ejemplo tres modelsCTL

5.4.4. Ejemplo 4

Por completitud, planteemos este cuarto y último ejemplo, donde vamos a comprobar las siguientes fórmulas para todos los posibles estados iniciales.

Las fórmulas que vamos a comprobar son las siguientes:

- $\mathbb{E}\mathbb{G}p$: Que puede leerse como "existe un camino donde siempre se tiene p".

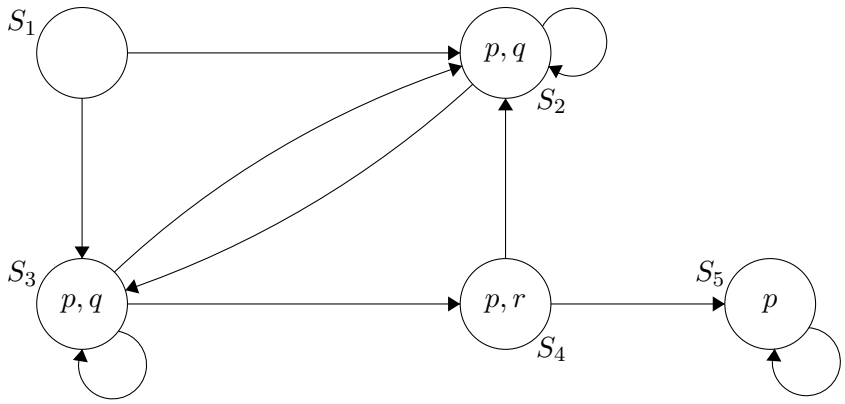

Figura 5.5: Ejemplo cuarto checkCTL

- $\mathbb{AG}p$: Que puede leerse como "en todos los caminos se tiene siempre p".
- $\mathbb{EF}\,\mathbb{AG}p$: Que puede leerse como "existe un camino donde en el futuro se tiene siempre en todos los caminos p".
- $\mathbb{A}[p\mathbb{U}(\mathbb{EG}(p \longrightarrow q))]$: Que puede leerse como "en todos los caminos se tiene siempre p hasta que en algún punto se tiene que en algún camino se tiene siempre que $(p \longrightarrow q)$".
- $\mathbb{E}[((p \wedge q) \vee r)\mathbb{U}(\mathbb{E}[r\mathbb{U}\,\mathbb{AG}p])]$: Que puede leerse como "existe un camino donde se tiene $((p \wedge q) \vee r)$ hasta que en algún punto se tiene que existe un camino donde se tiene r hasta que en algún punto se tiene que en todos los caminos se tiene siempre p".

```
1  import qualified Data.Matrix as M
2  import FSM.Automata
3  import FSM.States
4  import FSM.Logic
5
6  --Creamos los automatas
7  mat = M.fromLists [[2,3,0],[3,2,0],[3,2,4],[2,5,0],[5,0,0]]
8  tom1 = createAutomata 5 ['a', 'b','c'] 1 mat []
9  tom2 = createAutomata 5 ['a', 'b','c'] 2 mat []
10 tom3 = createAutomata 5 ['a', 'b','c'] 3 mat []
11 tom4 = createAutomata 5 ['a', 'b','c'] 4 mat []
12 tom5 = createAutomata 5 ['a', 'b','c'] 5 mat []
13
14 --Creamos la informacion de los estados
15 info_state1 = fromlsStateInfo 1 [] Nothing
16 info_state2 = fromlsStateInfo 2 [("atom1", Atom "p"),("atom2", Atom "q")] (Just
       info_state1)
17 info_state3 = fromlsStateInfo 3 [("atom1", Atom "p"),("atom2", Atom "q")] (Just
       info_state2)
18 info_state4 = fromlsStateInfo 4 [("atom1", Atom "p"),("atom2", Atom "r")] (Just
       info_state3)
19 info_state5 = fromlsStateInfo 5 [("atom1", Atom "p")] (Just info_state4)
20
```

```
21 --creamos la lista de formulas
22 formulas = [EG (Atom "p"),AG (Atom "p"), EF (AG (Atom "p")),AU (Atom "p") (EG (
       RArrow (Atom "p") (Atom "q"))), EU (Or (And (Atom "p") (Atom "q")) (Atom "r")
       ) (EU (Atom "r") (AG (Atom "p")))]
23
24 checkFormulas tom1 info_state5 formulas [] --output: [False,False,True,True,False
       ]
25 checkFormulas tom2 info_state5 formulas [] --output: [True,True,True,True,True]
26 checkFormulas tom3 info_state5 formulas [] --output: [True,True,True,True,True]
27 checkFormulas tom4 info_state5 formulas [] --output: [True,True,True,False,True]
28 checkFormulas tom5 info_state5 formulas [] --output: [True,True,True,False,True]
```

Código 5.15: ejemplo cuatro modelsCTL

CAPÍTULO 6

Conclusiones

El desarrollo de esta memoria junto con la librería en *Haskell* nos ha permitido establecer una introducción tanto a los conceptos básicos sobre *Model Checking* como a los procedimientos para el desarrollo de código en *Haskell*, y de forma más genérica, en lenguajes de programación funcional.

Además, se dejan algunas vías hacia las cuales la librería podría avanzar, las cuales pueden ir o bien profundizando en las posibles propiedades de un sistema, o bien implementando sistemas lógicos más complejos, tales como los que giran en torno al *Model Checking* simbólico o en torno a autómatas finitos **no** deterministas.

En uno u otro caso, consideramos el objetivo de este trabajo cumplido pues el objetivo final era poder aprovechar todo el desarrollo teórico para poder transformarlo en una herramienta real de la que se pueda hacer provecho para introducirse tanto en estos conceptos como en el uso de *Haskell* como lenguaje de programación.

Anexos

ANEXO I

Código Fuente FSM.Automata

```
1
2  -- Import packages ----------------------------------------
3
4  module FSM.Automata (
5
6      Automata,
7      -- * Creating functions
8      createAutomata,
9
10     -- * Accessing functions
11     getStates,
12     getAcceptingStates,
13     getInitialState,
14     getInputs,
15     getAssociations,
16     getTransitions,
17     getOutgoingStates,
18     getIncomingStates,
19     getDeadlocks,
20     getIsolated,
21
22     -- * Checking functions
23     validInput,
24
25     -- * Editing functions
26     addState,
27     deleteState,
28     changeInitialState,
29     addAcceptingState
30
31
32 ) where
33
34 import Data.Set
35 import qualified Data.List as L
36 import qualified Data.Matrix as M
```

```
37 import qualified Data.Vector as V
38
39 -- Custom show functions ---------------------------------------
40
41 showIntSet :: [Int] -> Int -> String
42 showIntSet [] 0 = id "{" ++ id "}"
43 showIntSet [] _ = id "}"
44 showIntSet (l:ls) 0 = id "{" ++ show l ++ showIntSet ls 1
45 showIntSet (l:ls) k = id "," ++ show l ++ showIntSet ls k
46
47 showCharSet :: [Char] -> Int -> String
48 showCharSet [] 0 = id "{" ++ id "}"
49 showCharSet [] _ = id "}"
50 showCharSet (l:ls) 0 = id "{" ++ show l ++ showCharSet ls 1
51 showCharSet (l:ls) k = id "," ++ show l ++ showCharSet ls k
52
53 -- Create data types ---------------------------------------
54
55 data Automata = A (Set Int,Set Char,Int,M.Matrix Int,Set Int)
56               --deriving Show
57
58 instance Show Automata where
59     show (A (s,i,s0,m,a)) =
60         (id "\n" ++ id "Set of states:" ++ id "\n" ++ s' ++ id "\n" ) ++
61         (id "\n" ++ id "Set of inputs (alphabet):" ++ id "\n" ++ i' ++ id "\n" )
              ++
62         (id "\n" ++ id "Initial state:" ++ id "\n" ++ show s0 ++ id "\n" ) ++
63         (id "\n" ++ id "Matrix of associations:" ++ id "\n" ++ show m ++ id "\n" )
               ++
64         (id "\n" ++ id "Set of accepting states:" ++ id "\n" ++ a')
65         --(id "\n" ++ id "Current state:" ++ id "\n" ++ show c ++ id "\n") ++ id
               "\n"
66         where s' = showIntSet (toList s) 0
67               i' = showCharSet (toList i) 0
68               a' = showIntSet (toList a) 0
69
70 -- Creating functions ---------------------------------------
71
72 -- | This is the main function for creating the Automata abstract data type. By
       default, the inital state and the current state of the automata are the same.
73 --
74 --  Please pay attention to how the object is built. E.g.,
75 --
76 -- > createAutomata s i s0 m a
77 --  where:
78 --
79 -- -s is the number of states of the automata.
80 -- -i is the alphabet the automata accepts.
81 -- -s0 is the initial state of the automata.
82 -- -m is the matrix of associations of the automata. (Details here: '
       getAssociations')
83 -- -a is the list of accepting states of the automata.
84 --
```

```haskell
-- More specifically you could
--
-- > import qualified Data.Matrix as M
-- > mat = M.fromLists [[2,0,0,0],[2,1,4,0],[1,4,0,0],[0,0,0,3]]
-- > tom = createAutomata 4 ['a', 'b', 'c', 'd'] 1 mat [4]


createAutomata :: Int -> String -> Int -> M.Matrix Int -> [Int] -> Automata
createAutomata s i s0 m a
    | s < 1 =
       error "Number of states must be greater than 1"
    | not (member s0 s') =
       error "Not valid initial state"
    | not ((M.nrows m,M.ncols m) == (size s',size i')) =
       error "Not valid matrix size"
    | not (isSubsetOf (delete 0 (fromList (M.toList m))) s') =
       error "Not valid matrix elems"
    | not (isSubsetOf a' s') =
       error "Not valid accepting states"
    | otherwise = A (s',i',s0,m,a')
      where s' = fromList [1..s]
            i' = fromList (L.sort i)
            a' = fromList (L.sort a)



-- Accessing functions ----------------------------------------


-- | This function returns the set of states of the automata. It is really of not
     much use since the generation of the automata only needs the number of
    states and not the whole set of them, but just in case you want to check
    which states does the current automata have.
getStates :: Automata -> Set Int
getStates t = s
    where A (s,i,s0,m,a) = t

-- | This function returns the list of accepting states of the automata. It is a
    list and not a set for coherence purpouses. When you build the automata you
    have to give a list of accepting states so I though it made sense to also
    return a list of accepting states as the accessing function.
getAcceptingStates :: Automata -> [Int]
getAcceptingStates t = a'
    where A (s,i,s0,m,a) = t
          a' = toList a

-- | This function returns the current initial state of the automata.
getInitialState :: Automata -> Int
getInitialState t = s0
    where A (s,i,s0,m,a) = t

-- | This function returns the string of inputs (alphabet) that the automata
    accepts.
```

```haskell
getInputs :: Automata -> String
getInputs t = toList i
    where A (s,i,s0,m,a) = t

-- | This function returns the associations matrix of the automata. This matrix
    is built according to the following rules:
--
-- 1. The columns of the matrix represent the inputs of the alphabet that the
    automata accepts in <https://en.wikipedia.org/wiki/Lexicographical_order
    lexicographical order>.
-- 2. The rows of the matrix represent the states of the automata in ascending
    order.
-- 3. The element \(a_{ij} = k \) means that the state \(i\) is connected to the
    state \(k\) thanks to the input that the column \(j\) of the matrix
    represents.
--
-- More info can be found here: <https://en.wikipedia.org/wiki/State-
    transition_table Wikipedia: State-transition table>
--
-- Continuing with the previous example, the following matrix correspongs to the
    automata in the figure.
--
-- > mat = M.fromLists [[2,0,0,0],[2,1,4,0],[1,4,0,0],[0,0,0,3]]
-- > tom = createAutomata 4 ['a', 'b', 'c', 'd'] 1 mat [4] 1
--
-- The code above represent this matrix:
--
-- >    'a' 'b' 'c' 'd'      <= inputs
-- >  ------------------
-- > 1 | 2   0   0   0
-- > 2 | 2   1   4   0
-- > 3 | 1   4   0   0
-- > 4 | 0   0   0   3
-- >
-- > ^
-- > |
-- > states
--
-- And the matrix above represents the transitions in the following automata:
--
-- <<https://i.imgur.com/ymWLlsb.png Tom automata figure>>

getAssociations :: Automata -> M.Matrix Int
getAssociations t = m
    where A (s,i,s0,m,a) = t

-- | This function returns the inputs that a state accepts for transitioning into
    another state.
--
getTransitions :: Automata -> Int -> Set Char
getTransitions t k
    | not (member k s) = error "Not a valid state"
    | otherwise = fromList l
```

```
175     where m = getAssociations t
176           i = getInputs t
177           s = getStates t
178           row = V.toList (M.getRow k m)
179           l = [ a | (a,k) <- zip i row, k /= 0]
180
181
182 -- -- | This function returns the current state in which the automata currently
        is.
183 -- --
184 -- getCurrentState :: Automata -> Int
185 -- getCurrentState t = c
186 --     where A (s,i,s0,m,a,c) = t
187
188 -- | This function returns the states you can possibly reach from a given state.
189 --
190 getOutgoingStates :: Automata -> Int -> Set Int
191 getOutgoingStates t k
192     | not (member k s) = error "Not a valid state"
193     | otherwise = fromList l
194     where m = getAssociations t
195           s = getStates t
196           row = V.toList (M.getRow k m)
197           l = [ p | p <- row, p /= 0]
198
199 -- | This function returns the states that can possibly reach a given state.
200 --
201 getIncomingStates :: Automata -> Int -> Set Int
202 getIncomingStates t k
203     | not (member k s) = error "Not a valid state"
204     | otherwise = fromList l
205     where m = getAssociations t
206           s = getStates t
207           rows = [(n,member k (fromList (V.toList (M.getRow n m)))) | n <- [1..(M.
                nrows m)]]
208           l = [n | (n,bool) <- rows, bool == True]
209
210 -- | This function returns those states of the automata that do not have any
        input to any other state (except for itself), i.e., once that a 'deadlock'
        state is reached, none of the rest of state can be reached anymore for the
        current execution.
211 getDeadlocks :: Automata -> Set Int
212 getDeadlocks t = Data.Set.filter (\ p -> (Data.Set.null (getOutgoingStates t p)
        ||
213                                       (getOutgoingStates t p) == Data.Set.fromList
                                            [p])) s
214     where A (s,i,s0,m,a) = t
215 -- getDeadlocks :: Automata -> Set Int
216 -- getDeadlocks t = fromList hs
217 --     where A (s,i,s0,m,a,c) = t
218 --           hs = [n | n <- toList s,
219 --                and [n == (M.getRow n m)V.!k || (M.getRow n m)V.!k == 0 | k <-
        [0..((size i)-1)]]]
```

```
220
221 -- | This function returns the states of the given automata that cannot be
        reached.
222 --
223 getIsolated :: Automata -> Set Int
224 getIsolated t = Data.Set.filter (\ p -> (Data.Set.null (getIncomingStates t p) ||
225                                 (getIncomingStates t p) == Data.Set.fromList [p])) s
226   where A (s,i,s0,m,a) = t
227 {-
228 getIsolated :: Automata -> Set Int
229 getIsolated t = fromList l
230     where s = getStates t
231           l = [ p | p <- toList s, getIncomingStates t p == empty]-}
232
233
234 -- Checking functions ----------------------------------------
235
236 validInputAux :: String -> Automata -> Int -> Bool
237 validInputAux str a k
238     | not (isSubsetOf (fromList str) i) = error "Invalid input"
239     | L.null str = member k ac
240     | not (member st (getTransitions a k)) = error ("Not valid input " ++ (show
            st) ++ " for state " ++ (show k) )
241     | otherwise = validInputAux (tail str) a k'
242     where s = getStates a
243           i = fromList (getInputs a)
244           s0 = getInitialState a
245           m = getAssociations a
246           ac = fromList (getAcceptingStates a)
247           st = head str
248           k' = M.getElem k ((findIndex st i)+1) m
249
250 -- | This function test if a string is @/valid/@, i.e., if when the automata
        receives the string, ends in one of the accepting states.
251 validInput :: String -> Automata -> Bool
252 validInput str a = validInputAux str a s0
253     where s0 = getInitialState a
254
255 -- Editing functions ----------------------------------------
256
257 -- | Function for adding a state to an Automata with the list of associations to
        the other states. If you would want to add a non-connected state, simply
        enter the list [0,..,0], with as many zeros as possible inputs.
258 addState :: Automata -> [Int] -> Automata
259 addState a ls
260     | L.length ls /= L.length (getInputs a) = error ( "Not a valid list of
            associations" )
261     | otherwise = createAutomata s i s0 m t
262     where s = (M.nrows (getAssociations a)) +1
263           i = getInputs a
264           s0 = getInitialState a
265           t = getAcceptingStates a
266           m = M.fromLists ((M.toLists (getAssociations a))++[ls])
```

```
267
268 dropElemAtIndex :: Int -> [[Int]] -> [[Int]]
269 dropElemAtIndex i ls = L.take (i-1) ls ++ L.drop i ls
270
271 -- | This function deletes a state and all the connections it has with any other
    state. Please note that this function automatically reassigns new numbers for
     the remaining states, so the states and the associations matrix change
    accordingly. E.g. if you delete in the previous automata the 3rd state, then
    since the new automata has just 3 states, the old 4th state becomes the new 3
    rd state.
272 deleteState :: Automata -> Int ->Automata
273 deleteState a i
274     | not (elem i (getStates a)) = a
275        --error ( "This state is not one of the states of the automata." )
276     | (getInitialState a) == i = error ( "You are trying to delete the initial
        state. If you want to perform this action, first change the initial state
         and then delete the old one.")
277     | elem i (fromList (getAcceptingStates a)) && L.length (getAcceptingStates a)
         == 1 = error ("You are trying to delete the only accepting state.")
278     | otherwise = createAutomata s i' s0' m t
279     where s = (M.nrows (getAssociations a)) -1
280           i' = getInputs a
281           s0 = getInitialState a
282           s0' = if s0 < i then s0 else s0-1
283           t = [if l < i then l else l-1 | l <- toList ((fromList (
                getAcceptingStates a)) `difference` singleton i)]
284           rows = M.toLists (getAssociations a)
285           rows_deleted = dropElemAtIndex i ([[if l < i
286                                               then l
287                                               else if l > i
288                                                    then l-1
289                                                    else 0 | l <- ls] | ls <- rows])
290           m = M.fromLists rows_deleted
291
292 -- | This function changes the initial state.
293 changeInitialState :: Automata -> Int -> Automata
294 changeInitialState t s0'
295     | not (elem s0' (getStates t)) = error ( "This state is not one of the states
         of the automata." )
296     | (getInitialState t) == s0' = error ( "State " ++ show s0' ++ " is already
        the initial state.")
297     | otherwise = createAutomata s' i' s0' m a
298        where a = getAcceptingStates t
299              s' = size (getStates t)
300              i' = getInputs t
301              m = getAssociations t
302
303
304 -- | This function adds one accepting state
305 --
306 addAcceptingState :: Automata -> Int -> Automata
307 addAcceptingState t a0
```

```
308     | not (elem a0 (getStates t)) = error ( "This state is not one of the states
            of the automata." )
309     | elem a0 (getAcceptingStates t) = error ( "State " ++ show a0 ++ " is
            already one of the accepting states.")
310     | otherwise = createAutomata s' i' s0 m a'
311     where a = getAcceptingStates t
312           a' = a ++ [a0]
313           s' = size (getStates t)
314           i' = getInputs t
315           m = getAssociations t
316           s0 = getInitialState t
```

Código I.1: Código fuente `FSM.Automata.hs`

ANEXO II

Código Fuente FSM.States

```
1
2  -- |
3  --
4  -- = Considerations
5  --
6  -- One caveat you should always take into account when using this package is that
       without some data creation from the user, the use of this package is a bit
      restricted. This happens because the way it is built the package forbids you
      to use more than one type of information between states (or inside one), so
      to work around this, if you want to have multiple types of information inside
       states, do as follows:
7  --
8  -- @
9  --  data CustomData = Type1 String | Type2 Int deriving (Show,Eq)
10 -- @
11 --
12 -- dont forget about the deriving because otherwise it will conflict with the
      functions in the package.
13
14
15 module FSM.States (
16     State,
17     Tag,
18     StateInfo,
19     AutomataInfo,
20
21     -- * Creating functions
22     createStateInfo,
23     fromlsStateInfo,
24
25     -- * Accessing functions
26     getStateInfo,
27     getStatesWithInfo,
28     getTagsInState,
29     getInfoInState,
30
```

```
31      -- * Editing functions
32      alterStateInfo,
33      unionStateInfo
34
35
36  ) where
37
38  import qualified Data.Map as Map
39  import qualified Data.List as L
40
41  type State = Int
42  type Tag = String
43
44
45  newtype StateInfo a = StateInfo {tagMap :: Map.Map Tag a} deriving Eq
46  newtype AutomataInfo a = AutomataInfo { toMap :: Map.Map State (StateInfo a)}
        deriving Eq
47
48  --data UserStateInfo = UserStateInfo { rate :: Float } deriving Show
49
50  verboseShowStateInfo :: Show a => Map.Map Tag a -> String
51  verboseShowStateInfo = concatMap formatter . Map.toAscList
52    where formatter (k, v) = concat ["--> [tag] ",k, ": ","\n", show v, "\n"]
53
54  instance Show a => Show (StateInfo a) where
55    show = verboseShowStateInfo . tagMap
56
57  verboseShow :: Show a => Map.Map State (StateInfo a) -> String
58  verboseShow = concatMap formatter . Map.toAscList
59    where formatter (s, i) = concat ["=> The elements in state ", show s, " are:\n"
          , verboseShowStateInfo (tagMap i), "\n"]
60
61
62  instance Show a => Show (AutomataInfo a) where
63    show = verboseShow . toMap
64
65
66  -- Creating functions -----------------------
67
68  -- | This function takes a State, a Tag and a value and creates an AutomataInfo
        object containing only the given State with the value and the tag associated
        to it.
69  -- E.g.:
70  --
71  -- > createStateInfo 4 "tag" 25
72  --
73  -- If you created your own data type, you can do as follows:
74  --
75  -- > my_info = createStateInfo 4 "tag" (Type2 25)
76  --
77  createStateInfo :: State -> Tag -> a -> AutomataInfo a
78  createStateInfo state tag k = AutomataInfo {toMap = Map.singleton state (
        StateInfo {tagMap = Map.singleton tag k})}
```

```
79
80  -- | This function takes a State, a list of (Tag,value) and Maybe AutomataInfo
        and returns the AutomataInfo updated with the list of tags given. Please
        notice that if Nothing is given, it will return the created AutomataInfo
        while if a (Just AutomataInfo) object is given, it will update the tags in
        the given state.
81  -- E.g. (notice that we are using @my_info@ from the previous example)
82  --
83  -- > fromlsStateInfo 4 [("foo", Type1 "on"),("bar", Type2 0)] Nothing
84  -- > fromlsStateInfo 4 [("foo", Type1 "on"),("bar", Type2 0)] (Just my_info)
85  --
86  fromlsStateInfo :: Eq a => State -> [(Tag,a)] -> Maybe (AutomataInfo a) ->
        AutomataInfo a
87  fromlsStateInfo state [] Nothing = AutomataInfo {toMap = Map.empty}
88  fromlsStateInfo state (l:ls) Nothing
89      | ls /= [] = fromlsStateInfo state ls (Just first_info)
90      | otherwise = first_info
91      where (tag,value) = l
92            first_info = createStateInfo state tag value
93  fromlsStateInfo state [] (Just info) = info
94  fromlsStateInfo state (l:ls) (Just (AutomataInfo info))
95      | ls /= [] = fromlsStateInfo state ls (Just new_aut_info)
96      | otherwise = new_aut_info
97      where (tag,value) = l
98            new_info = createStateInfo state tag value
99            new_aut_info = unionStateInfo new_info (AutomataInfo info)
100
101
102 -- Accessing functions ------------------------
103
104 getStateInfo :: StateInfo a -> Map.Map Tag a
105 getStateInfo (StateInfo k) = k
106
107
108 -- | This function returns the states of the given AutomataInfo that currently
        contain some information
109 --
110 getStatesWithInfo :: AutomataInfo a -> [State]
111 getStatesWithInfo (AutomataInfo k) = Map.keys k
112
113 -- | This function returns the tags that a given state contains inside the
        AutomataInfo
114 --
115 getTagsInState :: AutomataInfo a -> State -> [Tag]
116 getTagsInState (AutomataInfo k) n
117     | not (elem n (getStatesWithInfo (AutomataInfo k))) = error ("This state does
            not contain info.")
118     | otherwise = Map.keys (tagMap state_map)
119     where (Just state_map) = Map.lookup n k
120
121
122 -- | This function returns the information contained in the given state. If
        @Nothing@ is given, then it returns all the information in the state while if
```

```
     @Just tag@ is given, it will return only the information inside the given
    tag.
-- E.g:
--
-- > getInfoInState my_info 4 Nothing
-- > getInfoInState my_info 4 (Just "foo")
--
getInfoInState :: AutomataInfo a -> State -> (Maybe Tag) -> StateInfo a
getInfoInState (AutomataInfo k) n Nothing
    | not (elem n (getStatesWithInfo (AutomataInfo k))) = StateInfo Map.empty
       --error ("This state does not contain info.")
    | otherwise = state_map
    where (Just state_map) = Map.lookup n k
getInfoInState (AutomataInfo k) n (Just tag)
    | not (elem n (getStatesWithInfo (AutomataInfo k))) = StateInfo Map.empty
       -- error ("This state does not contain info.")
    | not (elem tag (getTagsInState (AutomataInfo k) n)) = StateInfo Map.empty
       --error ("This state does not contain the given tag.")
    | otherwise = output
    where (Just state_map) = Map.lookup n k
          tag_map = tagMap state_map
          Just tag_info = Map.lookup tag tag_map
          output = StateInfo {tagMap = Map.singleton tag tag_info}

-- Editing functions ---------------

-- | This function takes a State, Maybe Tag, a value and an AutomataInfo object
    and updates the value of the Tag in the given State. Please note that if if
    Nothing is given, it will delete the State.
-- E.g:
--
-- > alterStateInfo 3 (Just "foo") (Type2 45) my_info
--
alterStateInfo :: State -> Maybe Tag -> a -> AutomataInfo a -> AutomataInfo a
alterStateInfo state Nothing _ (AutomataInfo info)
    | not (elem state (getStatesWithInfo (AutomataInfo info))) = (AutomataInfo
       info)
       --error ("This state does not contain info.")
    | otherwise = AutomataInfo {toMap = Map.delete state info}
alterStateInfo state (Just tag) sinf (AutomataInfo info)
    | elem state (getStatesWithInfo (AutomataInfo info)) =
       let f _ = Just sinf
           (Just state_map) = Map.lookup state info
           tag_map = tagMap state_map
           new_tag_map = Map.alter f tag tag_map
           g _ = Just (StateInfo {tagMap = new_tag_map})
           new_state_map = Map.alter g state info
       in AutomataInfo {toMap = new_state_map}
    | otherwise =
       let new_tag_map = StateInfo {tagMap = Map.singleton tag sinf}
           g _ = Just new_tag_map
           new_state_map = Map.alter g state info
       in AutomataInfo {toMap = new_state_map}
```

```
171
172 -- | This function takes the left-biased union of t1 and t2. It prefers t1 when
        duplicate keys are encountered. Works similarly to <http://hackage.haskell.
        org/package/containers-0.6.2.1/docs/Data-Map-Strict.html#g:12 Data.Map.union
        >.
173 --
174
175 unionStateInfo :: AutomataInfo a -> AutomataInfo a -> AutomataInfo a
176 unionStateInfo (AutomataInfo info1) (AutomataInfo info2) =
177   AutomataInfo (Map.unionWith (\ (StateInfo sti1) (StateInfo sti2) ->
178                                (StateInfo (Map.union sti1 sti2)))
179                info1 info2)
```

Código II.1: Código fuente `FSM.States.hs`

ANEXO III

Código Fuente FSM.Logic

```haskell

module FSM.Logic (
    -- * Definition of the CTL language
    CTL (..),

    -- * Implementation of the model checking algorithm for CTL
    checkCTL,
    modelsCTL,
    checkFormulas

) where

import FSM.States
import FSM.Automata

import Data.Set
import qualified Data.Map as Map

-- | This is the definition of the CTL language. More info about this language
    can be found <https://en.wikipedia.org/wiki/Computation_tree_logic here>. You
     can find the details of the constructors in the definition of the 'CTL' data
    .
--
--
-- Here you can find a visual explanation of the constructors defined below.
-- <<https://i.imgur.com/e4tPiOY.jpg CTL examples>>
--
-- <https://www.researchgate.net/figure/CTL-tree-logic-1_fig6_257343964 Source>:
    A SAFE COTS-BASED DESIGN FLOW OF EMBEDDED SYSTEMS by Salam Hajjar
--

data CTL a = CTrue | CFalse -- ^ Basic bools.
       | RArrow (CTL a) (CTL a) -- ^ Basic imply.
       | DArrow (CTL a) (CTL a) -- ^ Basic double imply.
       | Atom a -- ^ It defines an atomic statement. E.g.: 'Atom' @"The plants
           look great."@
```

```
32          | Not (CTL a) -- 'Not' negates a 'CTL' formula.
33          | And (CTL a) (CTL a) -- 'And'
34          | Or (CTL a) (CTL a)
35          | EX (CTL a) -- ^ 'EX' means that the 'CTL' formula holds in at least one
              of the inmediate successors states.
36          | EF (CTL a) -- ^ 'EF' means that the 'CTL' formula holds in at least one
              of the future states.
37          | EG (CTL a) -- ^ 'EG' means that the 'CTL' formula holds always from one
              of the future states.
38          | AX (CTL a) -- ^ 'AX' means that the 'CTL' formula holds in every one of
              the inmediate successors states.
39          | AF (CTL a) -- ^ 'AF' means that the 'CTL' formula holds in at least one
              state of every possible path.
40          | AG (CTL a) -- ^ 'AG' means that the 'CTL' formula holds in the current
              states and all the successors in all paths. (It is true globally)
41          | EU (CTL a) (CTL a) -- ^ 'EU' means that exists a path from the current
              state that satisfies the first 'CTL' formula /until/ it reaches a
              state in that path that satisfies the second 'CTL' formula.
42          | AU (CTL a) (CTL a) -- ^ 'AU' means that every path from the current
              state satisfies the first 'CTL' formula /until/ it reaches a state
              that satisfies the second 'CTL' formula.
43        deriving (Ord,Show)
44
45  instance Eq a => Eq (CTL a) where
46      (CTrue) == Not (CFalse) = True
47      (Atom a) == (Atom b) = a == b
48      (Not a) == (Not b) = a == b
49      (And a b) == (And c d) = (a == c) && (b == d)
50      (Or a b) == (Or c d) = (a == c) && (b == d)
51      (RArrow a b) == (Or (Not c) d) = (a == c) && (b == d)
52      (EX a) == (EX b) = a == b
53      (EF a) == (EF b) = a == b
54      (EG a) == (EG b) = a == b
55      (AX a) == (AX b) = a == b
56      (AF a) == (AF b) = a == b
57      (AG a) == (AG b) = a == b
58      (EU a b) == (EU c d) = (a == c) && (b == d)
59      (AU a b) == (AU c d) = (a == c) && (b == d)
60      -- Aditional equivalences
61      (Or a b) == (Not (And (Not c) (Not d))) = (a == c) && (b == d)
62      (And a b) == (Not (Or (Not c) (Not d))) = (a == c) && (b == d)
63      Not (Not a) == b = a == b
64
65      (AX a) == (Not (EX (Not b))) = a == b
66      (AF a) == (Not (EX (Not b))) = a == b
67      (AG a) == (Not (EX (Not b))) = a == b
68
69      (EX a) == (Not (AX (Not b))) = a == b
70      (EF a) == (Not (AX (Not b))) = a == b
71      (EG a) == (Not (AX (Not b))) = a == b
72
73      (AU a b) == Not (Or (EU (Not c1) (Not (Or d c2))) (EG (Not c3))) = (a == d)
            && (b == c1) && (c1 == c2) && (c2 == c3)
```

```
74
75  data LTL a = LTL_Atom a -- ^ It defines an atomic statement. E.g.: 'Atom' @"The
        plants look great."@
76          | LTL_Not (LTL a) -- 'Not' negates a 'CTL' formula.
77          | LTL_And (LTL a) (LTL a) -- 'And'
78          | LTL_Or (LTL a) (LTL a)
79          | LTL_X (LTL a) -- ^ 'EX' means that the 'CTL' formula holds in at least
                one of the inmediate successors states.
80          | LTL_F (LTL a) -- ^ 'EF' means that the 'CTL' formula holds in at least
                one of the future states.
81          | LTL_G (LTL a) -- ^ 'EG' means that the 'CTL' formula holds always from
                one of the future states.
82          | LTL_U (LTL a) (LTL a) -- ^ 'EU' means that exists a path from the
                current state that satisfies the first 'CTL' formula /until/ it
                reaches a state in that path that satisfies the second 'CTL' formula.
83          deriving (Ord,Show)
84
85  instance Eq a => Eq (LTL a) where
86      (LTL_Atom a) == (LTL_Atom b) = a == b
87      (LTL_Not a) == (LTL_Not b) = a == b
88      (LTL_And a b) == (LTL_And c d) = (a == c) && (b == d)
89      (LTL_Or a b) == (LTL_Or c d) = (a == c) && (b == d)
90      (LTL_X a) == (LTL_X b) = a == b
91      (LTL_F a) == (LTL_F b) = a == b
92      (LTL_G a) == (LTL_G b) = a == b
93      (LTL_U a b) == (LTL_U c d) = (a == c) && (b == d)
94      -- Aditional equivalences
95      (LTL_Or a b) == (LTL_Not (LTL_And (LTL_Not c) (LTL_Not d))) = (a == c) && (b
            == d)
96      (LTL_And a b) == (LTL_Not (LTL_Or (LTL_Not c) (LTL_Not d))) = (a == c) && (b
            == d)
97
98
99
100 -- | This is the function that implements the model checking algorithm for CTL as
        defined by Queille, Sifakis, Clarke, Emerson and Sistla <https://dl.acm.org/
        doi/abs/10.1145/5397.5399 here> and that was later improved.
101 --
102 -- This function takes as an argument a 'CTL' formula, an 'Automata' and
        information about the states as defined in "FSM.Automata" and "FSM.States"
        respectively and checks whether the 'Automata' implies the 'CTL' formula.
        Once the algorithm has finished, you just need to look at the value in the
        initial state of the automata to know if it does, for example with:
103 --
104 -- @
105 -- Map.lookup (getInitialState 'Automata') (checkCTL 'CTL' 'Automata' '
        AutomataInfo')
106 -- @
107 --
108 checkCTL :: Eq a => CTL a -> Automata -> AutomataInfo (CTL a) -> Map.Map Int Bool
109 checkCTL CTrue tom info =
110     let states = (toList (getStates tom))
111     in Map.fromList [(x,True) | x <- states]
```

```
112 checkCTL (Atom a) tom info =
113     let states = (toList (getStates tom))
114     in checkCTLauxAtom (Atom a) info tom states Map.empty
115 checkCTL (Not a) tom info = checkCTLauxNot (checkCTL a tom info)
116 checkCTL (And a b) tom info = checkCTLauxAnd (checkCTL a tom info) (checkCTL b
        tom info)
117 checkCTL (Or a b) tom info = checkCTL (Not (And (Not a) (Not b))) tom info
118 checkCTL (RArrow a b) tom info = checkCTL (Or (Not a) b) tom info
119 checkCTL (DArrow a b) tom info = checkCTL (Or (And a b) (And (Not a) (Not b)))
        tom info
120 checkCTL (EX a) tom info =
121   let states = (toList (getStates tom))
122       sublabel = checkCTL a tom info
123   in checkCTLauxEX tom states sublabel (Map.fromList [(x,False) | x <- states])
124 checkCTL (AX a) tom info = checkCTL (Not (EX (Not a))) tom info
125 checkCTL (EU a b) tom info =
126     let sublabel1 = checkCTL a tom info
127         sublabel2 = checkCTL b tom info
128         states = (toList (getStates tom))
129         init_list = [x | (x,k) <- (Map.toList sublabel2), k == True]
130     in checkCTLauxEU tom (Map.fromList [(x,False) | x <- states]) (Map.fromList
            [(x,False) | x <- states]) init_list sublabel1
131 checkCTL (EF a) tom info = checkCTL (EU CTrue a) tom info
132 checkCTL (AU a b) tom info =
133     let sublabel1 = checkCTL a tom info
134         sublabel2 = checkCTL b tom info
135         states = (toList (getStates tom))
136         degree_map = Map.fromList [(x,length (toList (getOutgoingStates tom x))) |
                x <- states]
137         label_map = (Map.fromList [(x,False) | x <- states])
138         init_list = [x | (x,k) <- (Map.toList sublabel2), k == True]
139     in checkCTLauxAU tom label_map degree_map init_list sublabel1
140 checkCTL (AF a) tom info = checkCTL (AU CTrue a) tom info
141 checkCTL (EG a) tom info = checkCTL (Not (AF (Not a))) tom info
142 checkCTL (AG a) tom info = checkCTL (Not (EF (Not a))) tom info
143
144
145 checkCTLauxAtom :: Eq a => CTL a -> AutomataInfo (CTL a) -> Automata -> [State]
        -> Map.Map Int Bool -> Map.Map Int Bool
146 checkCTLauxAtom (Atom a) info tom [] label_map = label_map
147 checkCTLauxAtom (Atom a) info tom (l:ls) label_map =
148   let content_info = getStateInfo (getInfoInState info l Nothing)
149       content = Map.elems content_info
150       new_bool = elem (Atom a) content
151       f _ = Just new_bool
152       new_map = Map.alter f l label_map
153   in checkCTLauxAtom (Atom a) info tom ls new_map
154
155
156 checkCTLauxNot :: Map.Map Int Bool -> Map.Map Int Bool
157 checkCTLauxNot label_map = notMapAux label_map (Map.keys label_map)
158
159 notMapAux :: Map.Map Int Bool -> [Int] -> Map.Map Int Bool
```

```
notMapAux label_map [] = label_map
notMapAux label_map (l:ls) =
    let Just old_bool = Map.lookup l label_map
        f _ = Just (not old_bool)
        new_map = Map.update f l label_map
    in notMapAux new_map ls

checkCTLauxAnd :: Map.Map Int Bool -> Map.Map Int Bool -> Map.Map Int Bool
checkCTLauxAnd label_map1 label_map2 = andMapAux label_map1 label_map2 (Map.keys
    label_map1)

andMapAux :: Map.Map Int Bool -> Map.Map Int Bool -> [Int] -> Map.Map Int Bool
andMapAux label_map1 label_map2 [] = label_map1
andMapAux label_map1 label_map2 (l:ls) =
    let Just bool1 = Map.lookup l label_map1
        Just bool2 = Map.lookup l label_map2
        f _ = Just (bool1 && bool2)
        new_map = Map.update f l label_map1
    in andMapAux new_map label_map2 ls

checkCTLauxEX :: Automata -> [State] -> Map.Map Int Bool -> Map.Map Int Bool ->
    Map.Map Int Bool
checkCTLauxEX tom [] label_map marked_map = marked_map
checkCTLauxEX tom (l:ls) label_map marked_map =
    let connected = toList (getOutgoingStates tom l)
        connected_map = Map.filterWithKey (\k _ -> (elem k connected)) label_map
        new_bool = or (Map.elems connected_map)
        f _ = Just new_bool
        new_map = Map.update f l marked_map -- es alter?
    in checkCTLauxEX tom ls label_map new_map

checkCTLauxEU :: Automata -> Map.Map Int Bool -> Map.Map Int Bool -> [State] ->
    Map.Map Int Bool -> Map.Map Int Bool
checkCTLauxEU tom label_map seenbefore_map [] sublabel = label_map
checkCTLauxEU tom label_map seenbefore_map (k:ks) sublabel =
    let previous_states = toList (getIncomingStates tom k)
        f _ = Just True
        new_map = Map.update f k label_map
        (added_previous,new_seenbefore_map) = checkEUprevious seenbefore_map
            previous_states sublabel ks
    in checkCTLauxEU tom new_map new_seenbefore_map added_previous sublabel


checkEUprevious :: Map.Map Int Bool -> [State] -> Map.Map Int Bool -> [State] ->
    ([State],Map.Map Int Bool)
checkEUprevious seenbefore_map [] sublabel ls = (ls,seenbefore_map)
checkEUprevious seenbefore_map (p:ps) sublabel ls
    | previous_bool == False =
        let f _ = Just True
            new_seenbefore_map = Map.update f p seenbefore_map
        in if previous_marked == True
            then checkEUprevious new_seenbefore_map ps sublabel (toList (fromList
                (ls++[p])))
```

```
207             else checkEUprevious new_seenbefore_map ps sublabel ls
208     | otherwise = checkEUprevious seenbefore_map ps sublabel ls
209     where Just previous_bool = Map.lookup p seenbefore_map
210           Just previous_marked = Map.lookup p sublabel
211
212 checkCTLauxAU :: Automata -> Map.Map Int Bool -> Map.Map Int Int -> [State] ->
        Map.Map Int Bool -> Map.Map Int Bool
213 checkCTLauxAU tom label_map degree_map [] sublabel = label_map
214 checkCTLauxAU tom label_map degree_map (l:ls) sublabel =
215     let previous_states = toList (getIncomingStates tom l)
216         f _ = Just True
217         new_map = Map.update f l label_map
218         (added_previous,new_degree_map) = checkAUprevious new_map degree_map
                previous_states sublabel ls
219     in checkCTLauxAU tom new_map new_degree_map added_previous sublabel
220
221 checkAUprevious :: Map.Map Int Bool -> Map.Map Int Int -> [State] -> Map.Map Int
        Bool -> [State] -> ([State], Map.Map Int Int)
222 checkAUprevious label_map degree_map [] sublabel ls = (ls,degree_map)
223 checkAUprevious label_map degree_map (p:ps) sublabel ls =
224     if (new_degree == 0) && (previous_marked == True) && (label == False)
225     then checkAUprevious label_map new_degree_map ps sublabel (toList (fromList (
            ls++[p])))
226     else checkAUprevious label_map new_degree_map ps sublabel ls
227     where Just previous_degree = Map.lookup p degree_map
228           new_degree = previous_degree -1
229           f _ = Just new_degree
230           new_degree_map = Map.update f p degree_map
231           Just previous_marked = Map.lookup p sublabel
232           Just label = Map.lookup p label_map
233
234 -- | This function returns the result of \(automata \models formula \).
235 --
236 modelsCTL :: Eq a => CTL a -> Automata -> AutomataInfo (CTL a) -> Bool
237 modelsCTL a tom info = model
238     where (Just model) = Map.lookup (getInitialState tom) (checkCTL a tom info)
239
240
241
242 -- | This function loops over multiple formulas and tells you if the automata
        models each formula.
243 --
244 checkFormulas :: Eq a => Automata -> AutomataInfo (CTL a) -> [CTL a] -> [Bool] ->
        [Bool]
245 checkFormulas tom info [] bs = bs
246 checkFormulas tom info (l:ls) bs = checkFormulas tom info ls (bs++[modelsCTL l
        tom info])
```

Código III.1: Código fuente `FSM.Logic.hs`

Bibliografía

[1] B. Bérard, M. Bidoit, A. Finkel, F. Laroussine, A. Petit, L. Petrucci and Ph. Schnoebelen *Systems and Software Verification.* Springer, 2001.

[2] Ian Chiswell *A Course in Formal Languages, Automata and Groups.* Springer, 2009.

[3] James A. Anderson *Automata Theory with Modern Applications.* Cambridge University Press, 2006.

[4] Jeffrey Shallit *A Second Course in Formal Languages and Automata Theory.* Cambridge University Press, 2008.

[5] Peter Linz *An Introduction to Formal Languages and Automata.* Jones and Barlett Publishers, 2000.

[6] F. Kröger and S. Merz *Temporal Logic and State Systems.* Springer, 2008.

[7] M. Huth and M. Ryan *Logic in Computer Science.* Cambridge University Press, 2004.

[8] Salam Hajjar *Logic in Computer Science.* A Safe COTS-Based Design Flow Of Embedded Systems.