Universidad Nacional de Córdoba

Facultad de Matemática, Astronomía y Física

Trabajo Especial

Model checking cuantitativo de propiedades LTL en PRISM

Carlos Sergio Bederián

Director: Dr. Pedro R. D'Argenio

Índice general

Capítulo 1

Introducción

1.1. Motivación

Los avances tecnológicos han rodeado nuestras vidas de dispositivos y sistemas informáticos de toda clase y complejidad, desde pequeños aparatos operados por microcontroladores como un reloj digital hasta aplicaciones distribuídas de gran escala que corren en cientos y miles de nodos como los motores de búsqueda de Internet. Nuestra interacción con los mismos se ha vuelto natural y generalmente pasa desapercibida hasta que nos topamos con defectos, de los cuales la mayoría no pasan de ser errores molestos. Sin embargo, las fallas en sistemas críticos pueden tener graves consecuencias económicas, como por ejemplo los 327 millones de dólares perdidos en el Mars Climate Orbiter porque una componente reportaba datos en unidades imperiales en vez de métricas; o en el peor de los casos, poner gente en peligro como en el caso del sistema de misiles Patriot, que falló en derribar un misil Scud por un desfasaje de su reloj interno que resultó en 28 muertos. En estos sistemas es imprescindible la satisfacción de la especificación del sistema y además debe mantenerse el funcionamiento correcto, incluso en situaciones imprevistas, por períodos prolongados de tiempo.

Para encontrar los posibles fallos de un sistema existe una variedad de técnicas, aglomeradas en dos grupos: el de *testing*, en el que se realizan ensayos sobre distintas partes del sistema o sobre el sistema en su conjunto, y el de *verificación formal*, en el que se prueba la corrección del sistema utilizando métodos formales. Las técnicas de testing, aplicables desde el comienzo de la implementación del sistema con un bajo costo computacional, no son exhaustivas y por lo tanto pueden encontrar errores pero no garantizar su ausencia, particularmente cuando existen elementos no deterministas en el sistema, lo que las hace insuficientes para el desarrollo de sistemas críticos. Los métodos de verificación formal complementan al testing asegurando resultados a cambio de un mayor costo computacional que crece junto a la complejidad del sistema, y son aplicables durante la fase de diseño, previniendo la posible necesidad de revisar el diseño y reimplementarlo desde cero, que es altamente costosa en cuanto a tiempo y recursos desperdiciados. En este trabajo nos concentraremos en una de las técnicas de verificación formal conocida como *model checking*.

1.2. Model checking

El model checking consiste en decidir si un *modelo* de un sistema satisface una *propiedad* y proveer opcionalmente un contraejemplo en el caso de que no se satisfaga la propiedad.

El modelo es una descripción abstracta del sistema, que puede realizarse en base a su diseño previo a la implementación o bien puede generarse a partir del código del sistema. Aunque la abstracción reduce considerablemente el espacio de estados a verificar, la necesidad de considerar todas las ejecuciones posibles en el mismo lleva a una *explosión de estados* que dificulta la verificación de grandes sistemas. Uno de los enfoques para atacar este problema consiste en utilizar *diagramas de decisión binarios* [JEK+90], estructuras de datos eficientes que comprimen la gran redundancia contenida en los modelos, a cambio de requerir que se opere simultáneamente sobre conjuntos de estados del modelo. Esta técnica es conocida como *model checking simbólico*.

La propiedad es una especificación formal de algún requerimiento del sistema, que describe comportamientos entre los que tienen que estar incluídos los comportamientos del sistema para que se satisfaga el requerimiento. Las propiedades se expresan utilizando *lógicas temporales*, que permiten describir ocurrencias de eventos a través del tiempo. Si tenemos una propiedad que especifica el funcionamiento normal del sistema a través del tiempo y un algoritmo que permite evaluar la satisfacción de esta propiedad en cada estado del modelo del sistema, entonces nos basta con ver si los estados iniciales del modelo satisfacen la propiedad para resolver el problema del model checking.

Cuando en los modelos se dan eventos que no son certeros sino que tienen distintas probabilidades de ocurrir, desde lo extremamente improbable como que un rayo cósmico altere un bit de memoria hasta eventos con distribuciones uniformes de probabilidad como es el resultado de arrojar un dado o una moneda, el model checking pasa de tener una respuesta afirmativa o negativa a una probabilidad de satisfacción de la propiedad, o las propiedades deben ser extendidas para poder describir comportamientos probabilistas. En ambos casos, estamos realizando *model checking cuantitativo*. En otras técnicas de verificación formal, la verificación de sistemas probabilistas es muy dificultosa o imposible.

1.3. Lógicas temporales

Entre las lógicas temporales se distinguen las lógicas temporales de tiempo lineal como LTL [Pnu77], donde cada momento tiene un único sucesor, y las lógicas temporales en las que el tiempo se ramifica como CTL [EL87], donde el futuro en cada instante aún no ha sido determinado y los distintos caminos posibles forman un árbol.

Para entender sus diferencias necesitamos distinguir las *fórmulas de camino* que describen *ejecuciones*, los caminos dentro del modelo que transita el sistema en su funcionamiento, de las *fórmulas de estado* que hablan sobre las propiedades particulares de cada estado, incluyendo las ejecuciones que parten del mismo que pueden ser cuantificadas.

Como en la lógica de tiempo lineal el sucesor de cada instante ya ha sido determinado, las propiedades en LTL hablan sobre *ejecuciones*. Cada estado satisface una propiedad LTL cuando todas las ejecuciones que parten del mismo

satisfacen la propiedad.

Cuando el tiempo se ramifica, el énfasis se pone en las propiedades que satisfacen los distintos sucesores de cada estado, cuantificándolos. En el caso particular de CTL cada operador temporal, que establece una fórmula de camino, debe estar acompañado por un cuantificador existencial o universal, lo que simplifica la verificación. Por otra parte, las propiedades LTL pueden ser vistas como casos particulares de una lógica con tiempo ramificado en los que se cuantifican universalmente fórmulas de camino. La limitación impuesta a los operadores temporales de CTL resulta en propiedades expresables en LTL que no pueden ser expresadas en CTL y viceversa, por lo que estas lógicas no son comparables. Lo ideal sería implementar model checkers que verifiquen propiedades de la lógica con tiempo ramificado CTL*, que no posee las restricciones de CTL y por lo tanto incluye tanto a CTL como a LTL. La verificación de fórmulas CTL* utiliza el algoritmo de model checking de CTL para fórmulas de estado, y el de LTL para las fórmulas de camino.

Ejemplo 1.3.1. *Para un sistema simple como el de la figura 1.1 donde el sistema funciona hasta que transiciona a un estado de terminación, la propiedad LTL* F *"terminó", o "eventualmente se alcanza el estado de terminación", no se satisface para el estado inicial, señalado por la flecha, porque existe un camino en el que se permanece infinitamente en el primer estado. Tampoco se da para la propiedad CTL equivalente* AF *"terminó", o "para todos los sucesores se da que eventualmente se alcanza el estado de terminación", pero sí se satisface la propiedad* EF *"terminó" que pide que exista al menos un camino por el que se alcanza el estado de terminación.*

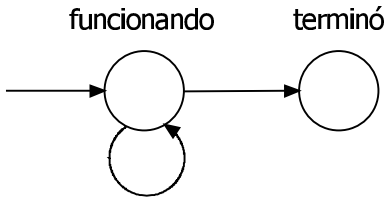


Figura 1.1: Modelo de un sistema simple que no siempre termina.

Los modelos probabilistas hacen que la cuantificación existencial o universal de las fórmulas de camino sea reemplazada por cotas mínimas y máximas de la sumatoria de las probabilidades de satisfacer la propiedad a través de cada camino, resultando en las nuevas lógicas PCTL [HJ94] y PCTL*. Sin embargo, en el caso de PCTL esto significa que debemos establecer cotas de probabilidad para cada operador temporal que ocurre en la fórmula de la propiedad, lo que lo hace poco práctico.

1.4. Objetivos

Este trabajo se enfocará la herramienta de verificación PRISM [HKNP06], que realiza simulaciones utilizando métodos de Monte Carlo y verifica propiedades PCTL para modelos probabilistas y no deterministas, y propiedades CSL para modelos estocásticos.

Modificaremos la herramienta para que soporte propiedades de la lógica LTL para modelos no deterministas utilizando el algoritmo descrito en [dA97], luego de adaptarlo a métodos simbólicos para poder operar con las estructuras internas de la herramienta. Este algoritmo toma el modelo a verificar y un autómata que describe las ejecuciones aceptadas por la propiedad, y sincroniza ambas estructuras generando un nuevo modelo. Dentro de este nuevo modelo, buscaremos los conjuntos de estados a los que pueden quedar confinadas las ejecuciones del sistema y seleccionaremos de estos conjuntos los que satisfacen la propiedad, con lo que el problema queda reducido a calcular la probabilidad de alcanzar estos estados particulares.

1.5. Organización de este trabajo

En el capítulo 2 hablaremos sobre modelos y en particular sobre *sistemas probabilistas no deterministas*, o SNP, que son una representación de modelos concurrentes con probabilidades, y las propiedades de los comportamientos en los mismos. Los resultados de este capítulo fueron tomados de [dA97].

En el capítulo 3 especificaremos propiedades en la lógica LTL, veremos los lenguajes generados por estas fórmulas y cómo se construyen ω-*autómatas* que aceptan estos lenguajes.

En el capítulo 4 presentaremos los diagramas de decisión binarios, o BDD, como una alternativa eficiente entre las distintas representaciones de funciones booleanas y veremos cómo manipularlos, siguiendo lo expuesto en [HR00]. A continuación veremos cómo esta estructura puede ser modificada para representar cualquier función cuya imagen es finita, obteniendo una nueva estructura denominada MTBDD.

En el capítulo 5 verificaremos propiedades LTL utilizando el algoritmo de model checking cuantitativo descrito en [dA97], que toma un SNP y un ω-autómata determinista y reduce el problema a un cálculo de probabilidad de alcanzar un estado dentro de un nuevo SNP. Luego mostraremos cómo almacenar un sistema de transición en un BDD o MTBDD, y adaptaremos el algoritmo de model checking para hacer uso exclusivo de estas estructuras.

En el capítulo 6 hablaremos brevemente sobre la herramienta PRISM, su lenguaje de especificación de modelos y su arquitectura interna.

En el capítulo 7 detallaremos los cambios realizados a PRISM para implementar el soporte de propiedades LTL.

En el capítulo 8 describiremos los casos de estudio realizados utilizando la herramienta modificada.

Finalmente en el capítulo 9 presentaremos las conclusiones de este trabajo.

Capítulo 2

Modelos

Comenzaremos este trabajo hablando sobre *modelos*, representaciones de los sistemas sobre los que se quiere verificar la satisfacción de una propiedad. Empezaremos definiendo formalmente una representación particular de sistemas concurrentes con eventos probabilistas. Luego definiremos el funcionamiento del sistema en términos de esta representación, y analizaremos propiedades importantes del mismo sobre las que se basa el resto del trabajo.

2.1. Sistemas no deterministas probabilistas

Veamos cómo definir formalmente la representación de los sistemas que queremos verificar. Podemos ver a un sistema como un conjunto de variables que describen su estado en un instante de tiempo, y expresar cada acción que realiza en términos de la transformación aplicada a las variables entre el antes (llamado *precondición*) y el después (*postcondición*) de la acción.

En vez de listar exhaustivamente todas las variables del sistema y modelar cada acción individualmente, haremos una abstracción del sistema hasta un nivel que nos sea conveniente. Para describir el estado del sistema usaremos conjuntos de *proposiciones atómicas*, enunciados sobre algún aspecto del sistema que en un momento determinado sólo podrán ser verdaderos o falsos.

Ejemplo 2.1.1. *Para modelar un automóvil tenemos cientos de variables como la temperatura del aceite, la carga de la batería, la posición de cada pistón, las revoluciones por minuto del motor, la presión de los neumáticos, etc. Sin embargo para describir un sistema que advierte a los pasajeros cuando hay alguna puerta abierta mientras el vehículo está en movimiento, nos basta con 3 proposiciones atómicas: "vehículo en movimiento", "puerta abierta" y "alarma encendida".*

Además, nos interesará tener una representación precisa de la elección del estado siguiente del sistema, ya sea *determinista*, que es habitual en sistemas no afectados por factores externos con un sólo hilo de ejecución, *probabilista*, que se utiliza para modelar alternativas dadas por eventos con probabilidad conocida, o *no determinista*, que se usa para modelar las elecciones realizadas por agentes sobre los que no tenemos control, como por ejemplo los sistemas concurrentes en los que el scheduler decide qué proceso se corre a continuación. Para proveer

la flexibilidad necesaria elegiremos en cada estado la acción a realizar de forma no determinista, donde cada acción tiene distintas probabilidades de alcanzar ciertos estados. Notemos que esto permite representar una elección determinista cuando el conjunto de acciones para el estado tiene un único elemento con un solo sucesor de probabilidad 1.

A esta representación, que definiremos formalmente a continuación, la llamaremos *sistema no determinista probabilista*:

Definición 2.1.1. Un *sistema no determinista probabilista*(SNP) es una 6-upla $(PA, S, A, \kappa, p, s_{in})$ donde:

- PA es un conjunto de proposiciones atómicas.
- S es un conjunto de *estados*. Para todo estado $s \in S$ y toda proposición atómica x, $s[\![x]\!]$ es el valor de verdad de x en s.
- A es un conjunto de *acciones*.
- $\kappa : S \to 2^A - \{\emptyset\}$ es una función que asocia cada estado con un conjunto no vacío de acciones que pueden realizarse en el mismo.
- $p : S \times A \times S \to [0.,1]$ es una función de distribución de probabilidad donde para todo $s, t \in S$ y $a \in \kappa(s)$, $p(t|s,a)$ es la probabilidad de realizar una transición de s a t a través de la acción a. Para todo $s \in S$ y $a \in \kappa(s)$, pedimos que $\sum_{t \in S} p(t|s,a) = 1$.
- $s_{in} \in S$ es un estado inicial.

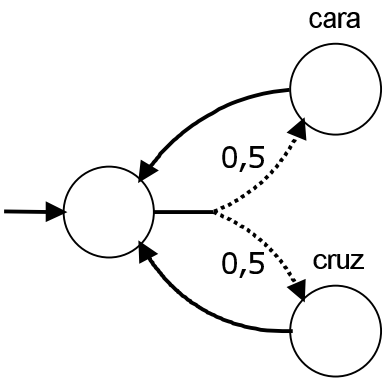


Figura 2.1: Modelo de una moneda al arrojarse. Omitiremos las proposiciones atómicas cuando ocurren negadas.

Ejemplo 2.1.2. *Modelemos lanzamientos de una moneda. Utilizaremos dos proposiciones atómicas, "cara" y "cruz" para los estados en los que la moneda salió cara y cruz respectivamente. Partiremos del momento previo al arrojarla, al que le asignaremos un estado en el que no valen "cara" ni "cruz". Alternativamente, podríamos utilizar una proposición atómica para codificar el valor de la moneda y otra para expresar que el lanzamiento fue realizado.*

Desde el estado inicial arrojaremos la moneda a través de una acción que tiene igual probabilidad de llegar a un estado en el que salió cara como en uno

en el que salió cruz. Una vez obtenido el resultado podemos retornar al estado inicial para realizar un nuevo lanzamiento. La representación gráfica del SNP de la moneda es mostrado en la figura 2.1.

Si tomamos otra moneda y operamos ambas en un orden arbitrario, el modelo del sistema (figura 2.2) debe tener en cuenta todas las combinaciones de las acciones posibles de cada moneda, o sus interleavings. *En esta situación de concurrencia, la elección del próximo estado no es sólo probabilista como al arrojar una sola moneda, sino que también hay no determinismo en la elección de la moneda que va a actuar a continuación.*

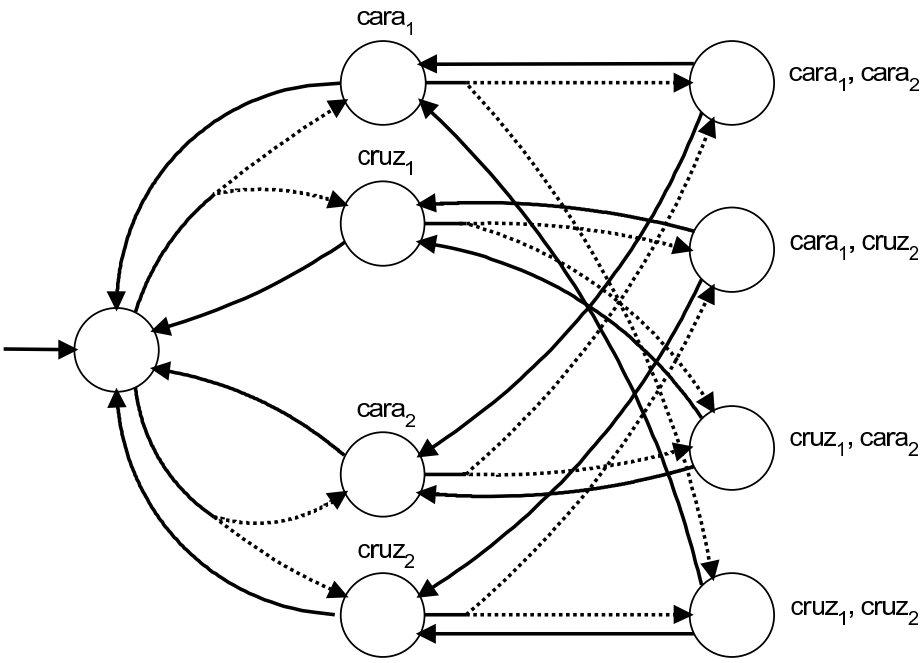


Figura 2.2: Composición paralela de dos monedas. Las probabilidades de transición omitidas son iguales a 0,5.

2.2. Ejecuciones y comportamientos

El funcionamiento del sistema modelado está representado por las secuencias de estados que nos permite realizar la relación de transición, a las que llamaremos *ejecuciones*:

Definición 2.2.1. Una *ejecución* es una secuencia infinita de estados $s_0 s_1 s_2 \ldots$ tal que $\exists a \in \kappa(s_i).p(s_{i+1}|s_i, a) > 0$ para todo $i \geq 0$.

Dada una ejecución $\sigma = s_0 s_1 s_2 \ldots$, usaremos σ_i para denotar el estado i+1-ésimo s_i, (σ, i) para denotar su sufijo infinito $s_i s_{i+1} s_{i+2} \ldots$ que comienza en s_i, e $\mathit{inft}(\sigma)$ para denotar el conjunto de estados que se repiten infinitamente en σ.

Siendo una variante de los *procesos de decisión de Markov*, los SNP satisfacen la *propiedad de Markov*, que dice que los estados anteriores de una ejecución no

afectan la elección de un nuevo estado, o sea que el sistema sólo conoce su posición actual y no cómo llegó al mismo.

Las ejecuciones nos sirven para describir un camino en un SNP, pero para saber la probabilidad de alcanzar un estado del sistema no nos basta con saber la secuencia de estados, sino que también necesitamos conocer las acciones elegidas en cada uno de los mismos, para lo que introduciremos los *comportamientos*:

Definición 2.2.2. Un *comportamiento* de un SNP es una secuencia infinita de estados y acciones $\omega = s_0 a_0 s_1 a_1 \ldots$ tal que para todo $i \geq 0$ y $s_i \in S$ se da que $a_i \in \kappa(s_i)$ y $p(s_{i+1}|s_i, a_i) > 0$. Dado un estado $s \in S$, llamaremos Ω_s al conjunto de comportamientos que comienzan en s.

Para determinar la probabilidad de realizar un comportamiento particular, lo haremos de la manera clásica de [JKK66]. Para cada estado $s \in S$, sea $\mathcal{B}_s \subseteq 2^{\Omega_s}$ la σ-álgebra más pequeña de subconjuntos medibles de Ω_s que contiene todos los conjuntos de cilindros básicos $C_{\omega_{fin}} = \{\omega \in \Omega_s \mid \omega_{fin} \text{ es prefijo de } \omega\}$ para cada prefijo finito ω_{fin} de los comportamientos que parten de s. No podemos asociar cada $\Delta \in \mathcal{B}_s$ con su medida de probabilidad $Pr(\Delta)$ debido a que la probabilidad de que un comportamiento pertenezca a Δ depende de cómo se realizaron las elecciones no deterministas, por lo que introducimos las *estrategias*, que determinan la probabilidad de elegir una acción:

Definición 2.2.3. Una *estrategia* η es un conjunto de probabilidades condicionales $Q_\eta(a \mid s_0 a_0 s_1 a_1 \ldots s_n)$ para $a \in \kappa(s_n)$, que expresa la probabilidad de elegir una acción dado el comportamiento hasta el momento.

Luego podemos definir la probabilidad de distintos eventos bajo una estrategia particular:

Definición 2.2.4. Denotaremos con $Pr_s^\eta(\mathcal{A})$ a la probabilidad de un evento $\mathcal{A}$ en $\mathcal{B}_s$ bajo la estrategia η.

La probabilidad de una transición a $t \in S$ luego de $s_0 \ldots s_n$ está dada por:

$$Pr_{s_0}^\eta(t \mid s_0 a_0 s_1 a_1 \ldots s_n) = \sum_{a \in \kappa(s_n)} p(t|s_n, a) Q_\eta(a \mid s_0 a_0 s_1 a_1 \ldots s_n)$$

La probabilidad de seguir un prefijo finito de un comportamiento en Ω_{s_0} bajo la estrategia η está dada por:

$$Pr_{s_0}^\eta(s_0 a_0 s_1 a_1 \ldots s_{n+1}) = \prod_{i=0}^{n} Pr_{s_0}^\eta(s_{i+1} \mid s_0 a 0 \ldots s_i)$$

Luego por el teorema de Carathéodory[Doo94] podemos extender esta medida a $\mathcal{B}_{s_0}$, y esta extensión es única:

$$Pr_{s_0}^\eta(C_{\omega_{fin}}) = Pr_{s_0}^\eta(\omega_{fin})$$

2.3. Conjuntos estables

Como tenemos ejecuciones infinitas dentro de un sistema con una cantidad finita de estados, la ejecuciones eventualmente se limitan a hacer ciclos en conjuntos reducidos de estados. Para hallarlos, empezaremos buscando los conjuntos de estados para los que existe una estrategia que hace que las ejecuciones nunca escapen de los mismos. A estos conjuntos los llamaremos *conjuntos estables*.

Definición 2.3.1. Sea $s \in S$ un estado y $a \in \kappa(s)$ una acción. Llamaremos *sucesores* a los estados alcanzables desde s a través de a, o sea:

$$Suc(s,a) = \{t \in S \mid p(t|s,a) > 0\}$$

Diremos que un conjunto $B \subseteq S$ es *estable* si para cada estado $s \in B$ existe una acción $a \in \kappa(s)$ tal que $Suc(s,a) \subseteq B$.

Dado un conjunto $C \subseteq S$, diremos que $B \subseteq C$ es un *conjunto estable maximal* si B es estable y no existe otro conjunto estable $B' \subseteq C$ que contenga a B.

Como para todo estado en el conjunto estable existe una acción para la que con probabilidad 1 se llega a otro estado dentro del conjunto estable, podemos formar una estrategia que siempre elige estas acciones llegando al siguiente resultado que dejaremos sin demostrar:

Lema 2.3.1. *Sea B un conjunto estable. Entonces existe una estrategia tal que cualquier comportamiento que ingresa a B permanece en B con probabilidad 1.*

Para obtener el conjunto estable maximal E_C dentro de un conjunto C, eliminamos todos los estados para los que no haya una acción a través de la cual todos los sucesores estén en C. Luego repetimos la operación para el conjunto reducido, y así sucesivamente hasta que no se eliminan más estados. La figura 2.3 muestra cómo se reduce un conjunto hasta obtener el conjunto estable maximal dentro del mismo.

$$E_C^0 = C$$
$$E_C^{i+1} = \{s \in E_C^i \mid \exists a \in \kappa(s).Suc(s,a) \subseteq E_C^i\}$$
$$E_C = lim_{i \to \infty} E_C^i$$

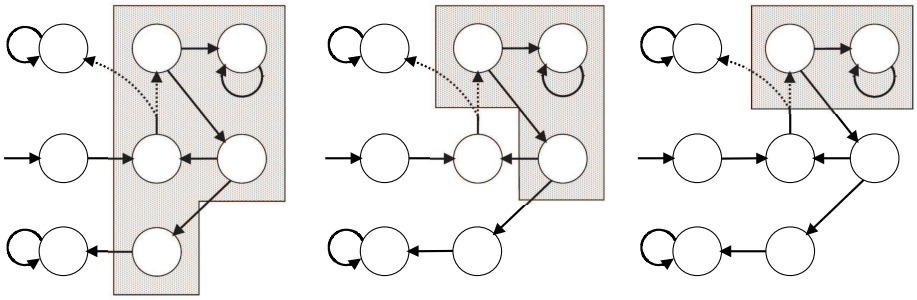


Figura 2.3: Ejecución del algoritmo de búsqueda de conjuntos estables.

2.4. Componentes terminales

Dentro de un conjunto estable puede haber estados que sólo son visitados una cantidad finita de veces por las ejecuciones atrapadas en un conjunto estable, porque no hay forma de volver a los mismos sin salir del conjunto estable. Como nos interesan los estados a los que se confinan las ejecuciones, nos concentraremos en los estados visitados infinitamente, que forman parte de ciclos dentro del conjunto estable. Estos conjuntos son conocidos en la teoría de grafos como *componentes fuertemente conexas*:

Definición 2.4.1. Sean $s, t \in S$ dos estados. Un *camino* de s a t es una secuencia finita de estados $s_1 s_2 \dots s_{n+1}$ tal que $\forall 0 < i \leq n. \exists a \in \kappa(s_i). s_{i+1} \in Suc(s_i, a)$ donde $s_1 = s$ y $s_n = t$.

Dado un estado $s \in S$ diremos que $t \in S$ es *alcanzable* desde s, o $s \rightsquigarrow t$, si existe un camino de s a t.

Sea $C \subseteq S$ un conjunto de estados. Si para todo par de estados $s, t \in C$ se da que s es alcanzable desde t y viceversa, diremos que C es *fuertemente conexo* o que es una *componente fuertemente conexa*. (CFC)

Como queremos que las ejecuciones permanezcan dentro de un conjunto estable, pediremos que los estados de una CFC dentro de un conjunto estable sean alcanzables entre sí utilizando transiciones de acciones que no salen del conjunto estable. Para ello restringiremos la relación de transición a:

$$\rho_B = \{(s,t) \in B \times B \mid \exists a \in \kappa(s).(Suc(s,a) \subseteq B \wedge t \in Suc(s,a))\}$$

Cuando el grafo construido con un conjunto de estados estable B y relación de transición ρ_B es fuertemente conexo, diremos que B es una *componente terminal*.

Definición 2.4.2. Sea B un conjunto estable. B es una *componente terminal* (CT) si el grafo (B, ρ_B) es fuertemente conexo.

Dado un conjunto $C \subseteq S$, diremos que B es una *componente terminal maximal* si B es una CT y no existe otra CT $B' \subseteq C$ que contenga a B.

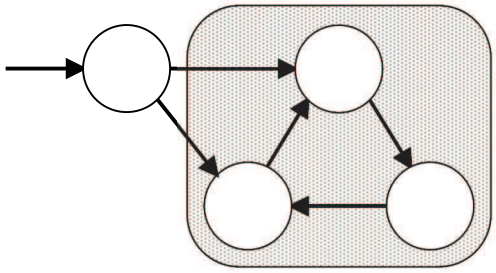


Figura 2.4: Una CFC dentro de un SNP.

Las componentes terminales poseen nuevas propiedades provenientes de combinar la conectividad de una CFC con la estabilidad de un conjunto estable:

Lema 2.4.1. *Sea B una CT. Entonces existe una estrategia tal que cualquier comportamiento que ingresa a B se mantendrá en B con probabilidad 1 y visitará todos los estados de B infinitamente a menudo.*

Veamos por qué se las llama componentes terminales. Anteriormente dijimos que para un sistema con una cantidad finita de estados, una ejecución infinita debe confinarse a un conjunto de estados. El siguiente lema nos dice que dicho conjunto es una CT:

Lema 2.4.2. *Para todo estado $s \in S$ y política η se da que*

$$Pr_s^{\eta}(\{\omega \in \Omega_s | inft(\omega) \text{ es una } CT\}) = 1.$$

Prueba. Asumamos por el absurdo que $Pr_s^\eta(\{\omega \in \Omega_s | \mathit{inft}(\omega) \text{ es una CT}\}) < 1$. Luego existe un conjunto $B \subseteq S$ que no es una CT y cuya probabilidad de ser igual a $\mathit{inft}(\omega)$ es mayor a cero. Definamos el conjunto de comportamientos confinados a B, $\Omega_s^B = \{\omega \in \Omega_s \mid \mathit{inft}(\omega) = B\}$, y la relación de transición ρ_B de la misma forma que para una CT. Como B no es una CT, existen estados $s_1, s_2 \in B$ tales que no existe un camino de s_1 a s_2 en el grafo (B, ρ_B), o sea que en los caminos de s_1 a s_2 debe usarse alguna transición cuya probabilidad de permanecer en B no es 1. Tomemos la probabilidad mínima de escapar de B a través de cualquier acción:

$$q = min\left\{\sum_{t' \notin B} p(t'|t,a) \mid t \in B \wedge a \in \kappa(t) \wedge \mathit{Suc}(t,a) \nsubseteq B\right\}$$

Por lo dicho anteriormente, $q > 0$. Luego a lo sumo una fracción $1-q$ de los comportamientos que pasan por s_1 y luego por s_2 lo hace sin salir de B. Pero como los comportamientos en Ω_s^B contienen infinitas secuencias disjuntas de s_1 a s_2, entonces $Pr_s^\eta(\omega \in \Omega_s^B) \leq (1-q)^k$ para todo $k > 0$, luego $Pr_s^\eta(\mathit{inft}(\omega) = B) = 0$, absurdo.

Como corolario tenemos que cuando una ejecución permanece infinitamente dentro de un conjunto de estados C en realidad lo hace en la unión de todos las CT maximales de C, o equivalentemente que la probabilidad de visitar infinitas veces un estado que no pertenece a ninguna CT maximal es de 0.

Corolario 2.4.3. *Sea $C \subseteq S$ un conjunto de estados de un SNP, $D_1, \ldots, D_n$ las CT maximales en C, y sea $D = \bigcup_{i=1}^n D_i$. Para toda ejecución ω, $s \in S$ y estrategia η se da que $Pr_s^\eta(\mathit{inft}(\omega) \subseteq C \wedge \mathit{inft}(\omega) \nsubseteq D) = 0$.*

Prueba. Asumamos por el absurdo que $Pr_s^\eta(\mathit{inft}(\omega) \subseteq C \wedge \mathit{inft}(\omega) \nsubseteq D) > 0$. Entonces por el lema anterior, existe una CT $B \subseteq C$ tal que $B \nsubseteq D$ y $Pr_s^\eta(\mathit{inft}(\omega) = B) > 0$. Pero D contiene todos las CT maximales en C, luego $B \subseteq D_i$ para algún i y por lo tanto $B \subseteq D$. Absurdo.

Para encontrar las CT maximales en un conjunto C, primero buscamos el conjunto estable maximal B de C. Para un conjunto de estados D, sea $\mathit{CFC}(D)$ el conjunto de componentes fuertemente conexas maximales de (D, ρ_D) obtenido con algún algoritmo de descomposición en CFC. Aunque B sea estable, los elementos de $\mathit{CFC}(B)$ pueden no serlo. Notemos que como ρ_D elimina las transiciones que pueden llevar a un estado externo a D, entonces si D no es estable existen estados sin transiciones salientes en (D, ρ_D). Luego si $D = \mathit{CFC}(D)$, todos los estados de D tienen transiciones salientes, por lo que D es estable. Por lo tanto volvemos a aplicar la búsqueda en cada elemento de $\mathit{CFC}(B)$ hasta que el resultado no cambia, obteniendo todos las CT maximales en C. La operación de este algoritmo es mostrada en la figura 2.5.

En el resto de este trabajo veremos cómo distinguir las CT en las que se confinan las ejecuciones que satisfacen la propiedad a verificar y la probabilidad de que se den estas ejecuciones.

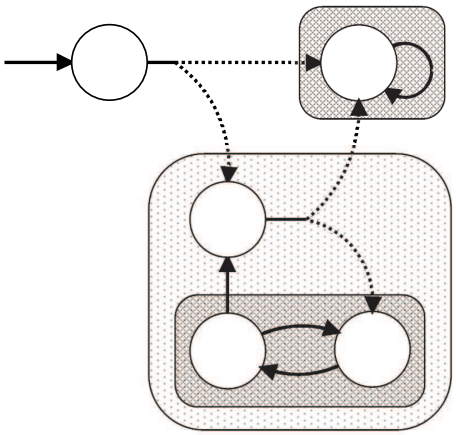


Figura 2.5: Ejecución del algoritmo de búsqueda de CT. La primera búsqueda de CFC encuentra una componente que no es estable, marcada por el sombreado fino. El sombreado grueso indica las CT definitivas.

Capítulo 3

Propiedades

Dado un modelo, representado generalmente por un sistema de transiciones de algún tipo, queremos ver si cumple alguna propiedad como por ejemplo que un sistema crítico siempre se recupera de una situación anómala. En este capítulo veremos una forma de especificar una propiedad formalmente, y cómo se puede expresar la propiedad de forma tal que podamos verificarla.

3.1. Lógica proposicional

Empezaremos por describir propiedades de un estado de un sistema. Para ello usaremos la *lógica proposicional*:

Definición 3.1.1. Una fórmula es de la *lógica proposicional* si es de la forma:

- p, donde p es una proposición atómica
- $\neg\phi$, donde ϕ es una fórmula de la lógica proposicional
- $\phi \vee \psi$, donde ϕ y ψ son fórmulas de la lógica proposicional

El operador $\neg$ es la negación, mientras que $\vee$ es la disyunción. El resto de los operadores lógicos $\wedge$ (conjunción), $\rightarrow$ (implicación) y $\leftrightarrow$ (equivalencia), y las constantes $true$ y $false$ se definen en términos de $\neg$ y $\vee$:

$$\begin{array}{lcl} \phi \wedge \psi & \equiv & \neg(\neg\phi \wedge \neg\psi) \\ \phi \rightarrow \psi & \equiv & \neg\phi \wedge \psi \\ \phi \leftrightarrow \psi & \equiv & (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi) \\ true & \equiv & \phi \vee \neg\phi \\ false & \equiv & \neg\ true \end{array}$$

La semántica de una fórmula está definida por la relación de *satisfacción* entre los estados de un modelo y la fórmula:

Definición 3.1.2. Sea s un estado, p una proposición atómica, y ϕ, ψ dos fórmulas de la lógica proposicional. La relación de *satisfacción* $\vDash$ está definida como:

$s \vDash p$	si y solo si $s[\![p]\!]$
$s \vDash \neg\phi$	si y solo si no se da $s \vDash \phi$
$s \vDash \phi \vee \psi$	si y solo si $s \vDash \phi$ o $s \vDash \psi$

3.2. LTL

Para expresar propiedades sobre ejecuciones de los sistemas recurrimos a las *lógicas temporales*. Las lógicas temporales son lógicas proposicionales a las que se les agregan operadores modales para expresar propiedades que ocurren a lo largo de la ejecución.

En nuestro trabajo nos concentraremos en la *lógica temporal lineal* (LTL), introducida por Amir Pnueli[Pnu77], sobre la que se basan otros lenguajes de especificación como IEEE 1850 PSL. En LTL el tiempo es *discreto* en cuanto a que cada momento se corresponde a un estado del sistema, y *lineal*, donde cada momento tiene un único sucesor. LTL extiende la lógica proposicional agregando dos operadores: X (del inglés *next*, "siguiente") y U. (del inglés *until*, "hasta")

Definición 3.2.1. Las fórmulas LTL son de la forma:

1. p, donde p es una proposición atómica.
2. $\neg\phi$, donde ϕ es una fórmula LTL
3. $\phi \vee \psi$, donde ϕ y ψ son fórmulas LTL
4. $\mathsf{X}\phi$, donde ϕ es una fórmula LTL
5. $\phi\mathsf{U}\psi$, donde ϕ y ψ son fórmulas LTL

A diferencia de la lógica proposicional donde la relación de satisfacción es sobre estados, la semántica de las fórmulas LTL se define sobre ejecuciones, incluso en el caso de las proposiciones atómicas para las que pediremos que el primer estado de la ejecución las satisfaga.

Definición 3.2.2. Sean σ una ejecución, $p \in PA$ una proposición atómica y ϕ, ψ fórmulas LTL. La relación de satisfacción $\vDash$ se define como:

$\sigma \vDash p$	si y solo si $\sigma_0[\![p]\!]$
$\sigma \vDash \neg\phi$	si y solo si no se da $\sigma \vDash \phi$
$\sigma \vDash \phi \vee \psi$	si y solo si $\sigma \vDash \phi$ o $\sigma \vDash \psi$
$\sigma \vDash \mathsf{X}\phi$	si y solo si $\sigma_1 \vDash \phi$
$\sigma \vDash \phi\mathsf{U}\psi$	si y solo si $\exists j \geq 0.\sigma_j \vDash \psi \wedge (\forall 0 \leq k < j.\sigma_k \vDash \phi)$

Intuitivamente, la fórmula $\mathsf{X}\phi$ es satisfecha si ϕ es satisfecha en el estado siguiente de la ejecución, mientras que $\phi\mathsf{U}\psi$ es satisfecha cuando existe un estado que satisface ψ previo al cual se mantiene ϕ. La forma de las ejecuciones que satisfacen cada fórmula se pueden ver en la figura 3.1.

Además de los operadores básicos antes mencionados, se definen varios operadores auxiliares en función de los mismos, mostrados en la figura 3.2:

- Los operadores lógicos $\wedge$, $\rightarrow$, $\leftrightarrow$, definidos de la forma tradicional en función de $\neg$ y $\vee$
- El cuantificador temporal existencial F (del inglés *finally*, "Finalmente"), definido como $\mathsf{F}\phi \equiv \mathit{true}\ \mathsf{U}\ \phi$.
- El cuantificador temporal universal G (del inglés *globally*, "Globalmente"), definido como $\mathsf{G}\phi \equiv \neg\mathsf{F}\neg\phi$.

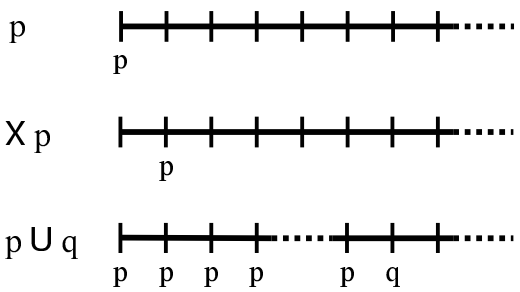


Figura 3.1: Ejecuciones de fórmulas LTL

- El operador temporal $\bar{\mathsf{U}}$ (del inglés *unless*, "a menos que"), definido como $\phi\bar{\mathsf{U}}\psi \equiv \phi\mathsf{U}\psi \vee \mathsf{G}\phi$.
- El operador temporal R (del inglés *releases*, "libera"), definido como $\phi\mathsf{R}\psi \equiv \neg(\neg\phi\mathsf{U}\neg\psi)$.

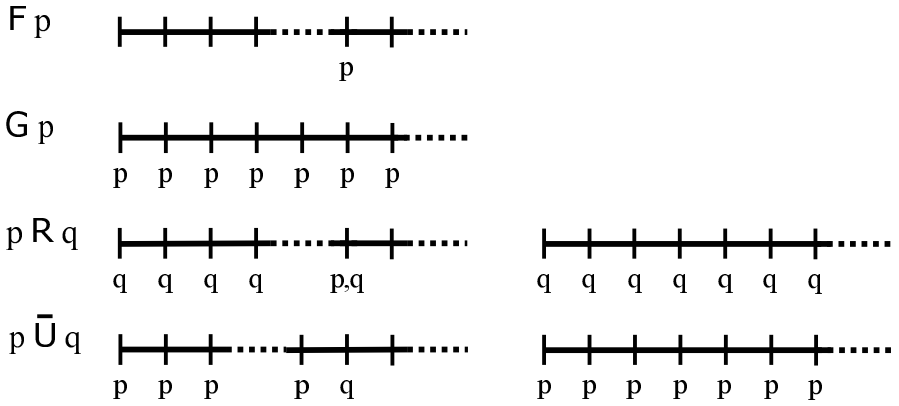


Figura 3.2: Operadores auxiliares de LTL

$\mathsf{F}\psi$ se satisface si ψ vale en algún estado de la ejecución, $\mathsf{G}\psi$ se satisface si ψ vale a lo largo de toda la ejecución, $\phi\bar{\mathsf{U}}\psi$ se satisface si se mantiene ϕ a menos que ocurra ψ, y $\phi\mathsf{R}\psi$ se satisface si ψ debe mantenerse hasta una ocurrencia de ϕ inclusive.

3.3. Equivalencia con autómatas

Para verificar si un modelo satisface una fórmula LTL nos es necesario expresarla de una manera que se adecue mejor a un algoritmo. Empezaremos definiendo formalmente el lenguaje generado por una fórmula y luego veremos distintas estructuras que pueden generar el mismo lenguaje y cómo construirlas.

3.3.1. Lenguajes regulares

Sea Σ un conjunto de símbolos al que llamaremos *alfabeto*. Una *palabra* es una secuencia finita de elementos del alfabeto. Por ejemplo, la palabra vacía ϵ,

aa, *ab*, *ba* y *ababbba* son palabras del alfabeto $\{a, b\}$. Dadas dos palabras σ y σ' podemos obtener su *concatenación* uniéndolas en una sola palabra $\sigma'' = \sigma\sigma'$.

Un *lenguaje* es un conjunto de palabras de un alfabeto, sobre el que también definiremos operaciones. Aparte de la unión de conjuntos, podemos extender la concatenación a lenguajes. Denotaremos con $\mathcal{L}.\mathcal{L}'$ a la concatenación de las palabras de los lenguajes $\mathcal{L}$ y $\mathcal{L}'$, definida como $\{\sigma\sigma' \mid \sigma \in \mathcal{L}, \sigma' \in \mathcal{L}'\}$. También podemos obtener nuevos lenguajes a través de la repetición de las palabras del lenguaje. La *clausura de Kleene* de $\mathcal{L}$, o $\mathcal{L}^*$, es el lenguaje que contiene todas las repeticiones finitas de las palabras de $\mathcal{L}$, o formalmente $\mathcal{L}^* = \{\epsilon\} \cup \{\alpha_1 \dots \alpha_n \mid \alpha_1, \dots, \alpha_n \in \mathcal{L}, n > 0\}$.

El conjunto de todos los lenguajes construídos con estos operadores es el de los *lenguajes regulares*.

Definición 3.3.1. Sea Σ un alfabeto. Diremos que un lenguaje $\mathcal{L}$ sobre Σ es *regular* si:

- $\mathcal{L} = \emptyset$ o $\mathcal{L} = \{\epsilon\}$
- $\mathcal{L} = \{a\}$ para algún $a \in \Sigma$
- $\mathcal{L} = \mathcal{L}'.\mathcal{L}''$ o $\mathcal{L} = \mathcal{L}' \cup \mathcal{L}''$ donde $\mathcal{L}'$ y $\mathcal{L}''$ son lenguajes regulares sobre Σ
- $\mathcal{L} = \mathcal{L}'^*$ donde $\mathcal{L}'$ es un lenguaje regular sobre Σ

3.3.2. Autómatas finitos

Los lenguajes regulares pueden ser generados con *autómatas finitos*:

Definición 3.3.2. Un *autómata finito* $\mathcal{A}$ es una 5-upla $(\Sigma, Q, I, \delta, F)$ donde:

- Σ es un *alfabeto* finito
- Q es un conjunto finito de *estados*
- $I \subseteq S$ es un conjunto de *estados iniciales*
- $\delta : (S \times \Sigma) \to S$ es una *relación de transición* etiquetada
- $F \subseteq S$ es un conjunto de *estados finales*

Dada una palabra finita ω compuesta de elementos de Σ y partiendo de alguno de los estados iniciales, el autómata opera consumiendo sucesivamente el primer símbolo de la palabra y tomando alguna de las transiciones que salen del estado actual que tienen como etiqueta el símbolo consumido. Al igual que para los SNPs llamaremos a este recorrido de estados del autómata *ejecución*, excepto que en este caso el recorrido es finito. Cuando una ejecución termina en uno de los estados finales, diremos que la ejecución y la palabra consumida son *aceptadas* por el autómata.

Definición 3.3.3. Una *ejecución* en $\mathcal{A}$ es una secuencia de estados en Q^* no vacía $\sigma = q_0 q_1 \dots q_n$ tal que $q_0 \in I$ y para todo $0 \leq i < n$ existe un símbolo $a_i \in \Sigma$ tal que $q_{i+1} \in \delta(q_i, a_i)$. Si además se da que $q_n \in F$, la ejecución σ y la palabra $\omega = a_0 a_1 \dots a_n$ son *aceptadas* por $\mathcal{A}$.

Cuando tomamos el conjunto de todas las palabras aceptadas por un autómata obtenemos el lenguaje que acepta. Dos autómatas serán *equivalentes* si aceptan el mismo lenguaje.

Definición 3.3.4. El conjunto de estados alcanzables desde q consumiendo $\omega = a_0 a_1 \dots a_n$ es:

$$\hat{\delta}(q, \omega) = \{p \in Q \mid \exists q_0 \dots q_n \in Q. \forall 0 \leq i < n. q_{i+1} \in \delta(q_i, a_i) \wedge q_0 = q \wedge q_n = p\}$$

El lenguaje aceptado por un autómata finito $\mathcal{A}$ está definido como:

$$\mathcal{L}(\mathcal{A}) = \{\omega \in \Sigma^* \mid \exists q_{in} \in I. \hat{\delta}(q_{in}, \omega) \in F\}$$

Dos autómatas finitos $\mathcal{A}$, $\mathcal{A}'$ son *equivalentes*, o $\mathcal{A} \equiv \mathcal{A}'$, si y solo si $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

El conjunto de lenguajes que se pueden generar con autómatas finitos, es decir, su *expresividad*, es igual a la de los lenguajes regulares. Esto también nos dice que un lenguaje es regular si existe un autómata finito que acepte el mismo lenguaje y viceversa.

Una distinción importante entre los autómatas es el *determinismo*. En los autómatas deterministas sólo existe un camino por el que se puede consumir una palabra determinada, mientras que los autómatas no deterministas pueden poseer más de una transición saliente de un estado con la misma etiqueta.

Definición 3.3.5. Un autómata de estado finito es *determinista* si y solo si $|I| = 1$ y para todo estado s y toda etiqueta a, $|\delta(s, a)| \leq 1$.

El determinismo no afecta la expresividad de los autómatas para palabras finitas. Dado un autómata no determinista se puede construír uno determinista equivalente uniendo todos los sucesores de un estado a través de la misma etiqueta en un solo estado:

Teorema 3.3.1. *Sea $\mathcal{A} = (\Sigma, Q, I, \delta, F)$ un autómata finito no determinista. Si definimos el autómata determinista $\mathcal{A}' = (\Sigma, Q', I', \delta', F')$ donde:*

- $Q' = \{\hat{\delta}(q_{in}, \omega) \mid q_{in} \in I, \omega \in \mathcal{L}(\mathcal{A})\}$
- $I' = \{I\}$
- $\delta'(q', a) = \bigcup_{q \in q'} \delta(q, a)$
- $F' = \{q_f \in Q' \mid q_f \cap F \neq \emptyset\}$

Entonces $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

3.3.3. Lenguajes ω-regulares

Los lenguajes regulares sólo contienen palabras de longitud finita, mientras que a nosotros nos interesan los lenguajes generados por las ejecuciones de un sistema que corre continuamente, que sólo contienen palabras infinitas y que son llamados ω-lenguajes. Este requerimiento no introduce una pérdida de generalidad puesto que si el sistema termina podemos agregar bucles en los estados de terminación.

Para trabajar con lenguajes infinitos definimos el operador ω, que dada una palabra finita devuelve su repetición infinita $\sigma^\omega = \sigma\sigma\sigma\sigma\ldots$. Luego lo extendemos a lenguajes aplicándole ω a cada palabra, de modo que $\mathcal{L}^\omega = \{\sigma^\omega \mid \sigma \in \mathcal{L}\}$.

Cuando concatenamos lenguajes regulares con otros lenguajes regulares a los que les aplicamos este nuevo operador obtenemos un tipo de ω-lenguaje, los lenguajes *ω-regulares*.

Definición 3.3.6. Un ω-lenguaje es *ω-regular* si se lo puede expresar como una unión finita de lenguajes $\mathcal{L}_i.(\mathcal{L}'_i)^\omega$, donde $\mathcal{L}_i$ y $\mathcal{L}'_i$ son lenguajes regulares para todo i.

En particular, los lenguajes generados por las fórmulas LTL, que definiremos a continuación, son un subconjunto de los lenguajes ω-regulares:

Definición 3.3.7. El *lenguaje* de una fórmula LTL ϕ se define como el conjunto de ejecuciones que satisfacen la fórmula, o sea:

$$\mathcal{L}(\phi) = \{\sigma \in S^\omega \mid \sigma \vDash \phi\}$$

Como la relación de satisfacción está definida sobre las proposiciones atómicas que valen en cada estado de la ejecución, podemos definir una operación para extraer las proposiciones atómicas que valen en un estado o en toda una ejecución:

$$\begin{array}{ll} L(s \in S) & = \{p \in PA \mid s[\![p]\!]\} \\ L'(s_0 s_1 \ldots \in S^\omega) & = L(s_0)L(s_1)\ldots \end{array}$$

Aplicando L' a los miembros de $\mathcal{L}(\phi)$, obtenemos un nuevo lenguaje con todas las secuencias de 2^{PA^ω} generadas por la fórmula LTL que es independiente de los estados:

$$\{L'(\sigma) \mid \sigma \in \mathcal{L}(\phi)\}$$

En el resto de este capítulo veremos distintos autómatas que son tan expresivos como los lenguajes ω-regulares. Gracias a esta propiedad, estos autómatas pueden aceptar el mismo lenguaje que el generado por una fórmula LTL y serán utilizados para realizar la verificación.

3.3.4. ω-autómatas

La no terminación de las palabras de los lenguajes ω-regulares hace que sea necesario reemplazar los estados finales de las máquinas de estados finitos por algún criterio que se debe cumplir para que una palabra infinita sea aceptada. Estos autómatas sobre palabras infinitas son llamados *ω-autómatas*.

Definición 3.3.8. Un *ω-autómata* $\mathcal{A}$ es una 5-upla $(\Sigma, Q, I, \delta, F)$ donde:

- Σ es un *alfabeto* finito
- Q es un conjunto finito de *estados*
- $I \subseteq Q$ es un conjunto de *estados iniciales*
- $\delta : (Q \times \Sigma) \to Q$ es una *relación de transición* etiquetada

- F es un *criterio de aceptación*

Una *ejecución* en $\mathcal{A}$ es una secuencia de estados $\sigma = q_0q_1q_2\ldots$ tal que $q_0 \in I$, y para todo $i \geq 0$ existe un símbolo $a_i \in \Sigma$ tal que $q_{i+1} \in \delta(q_i, a_i)$.

Una palabra infinita $\omega = a_0a_1a_2\ldots$ es *aceptada* por $\mathcal{A}$ si y solo si existe una ejecución $\sigma = q_0q_1q_2\ldots$ tal que $\forall 0 \leq i.q_{i+1} \in \delta(q_i, a_i)$ y cumple con el criterio de aceptación.

El ω-lenguaje aceptado por $\mathcal{A}$, $\mathcal{L}_\omega(\mathcal{A})$, es el conjunto de palabras infinitas aceptadas por $\mathcal{A}$.

A continuación veremos cómo distintos criterios y conjuntos de aceptación generan distintos tipos de ω-autómatas a los que puede ser traducida una fórmula LTL.

Autómatas de Büchi

Los *autómatas de Büchi*(AB) son ω-autómatas en los que el criterio de aceptación es la *condición de Büchi*:

Definición 3.3.9. Una ejecución σ es aceptada por un autómata de Büchi si y solo si $inft(\sigma) \cap F \neq \emptyset$, donde $F \subseteq Q$ es un conjunto de estados de aceptación.

Intuitivamente, una palabra es aceptada por un autómata de Büchi si la ejecución inducida por la palabra visita infinitamente a menudo alguno de los estados de aceptación.

Como dijimos anteriormente, los autómatas de Büchi son tan expresivos como los lenguajes ω-regulares:

Teorema 3.3.2. *Un lenguaje $\mathcal{L}$ es ω-regular si y solo si existe un autómata de Büchi $\mathcal{A}$ tal que $\mathcal{L} = \mathcal{L}_\omega(\mathcal{A})$.*

La dificultad que presentan los autómatas de Büchi deterministas es que son menos expresivos que los no deterministas, o sea, existen ω-lenguajes generados por autómatas de Büchi no deterministas(ABN) para los que no existen autómatas de Büchi deterministas equivalentes.

Ejemplo 3.3.1. *El lenguaje generado por la expresión ω-regular $(a|b)^*b^\omega$ es aceptado por el autómata de Büchi no determinista de la figura 3.3.5, pero no puede ser aceptado por ningún autómata de Büchi determinista. Supongamos que tal autómata existe, luego existe un punto en la ejecución luego de consumir $(a|b)^n$ para algún n en el que el autómata al consumir b entra en un ciclo que posee un estado de aceptación en el que sólo ocurre b para poder aceptar b^ω, dejando de consumir a. Pero entonces el autómata no puede aceptar $(a|b)^nba(b)^\omega$ que está contenido en $(a|b)^*b^\omega$, y al ser determinista este camino es el único por el que se puede consumir $(a|b)^n$.*

Autómatas de Büchi generalizados

Otro tipo de ω-autómata que usaremos es el de los *autómatas de Büchi generalizados*(ABG). Los autómatas de Büchi generalizados poseen un conjunto de conjuntos de aceptación de Büchi, y una ejecución es aceptada cuando satisface cada uno de los conjuntos de aceptación bajo la condición de Büchi.

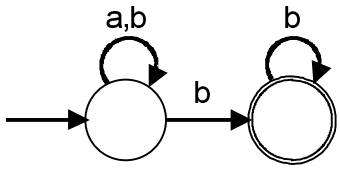


Figura 3.3: Autómata de Büchi no determinista de $(a|b)^*b^\omega$. El estado con doble borde está en el conjunto de aceptación.

Definición 3.3.10. Una ejecución σ es aceptada por un autómata de Büchi generalizado si y solo si $\forall 0 < i \leq k. inft(\sigma) \cap F_i \neq \emptyset$, donde $F = \{F_1, \ldots, F_k\} \subseteq 2^Q$ es un conjunto de conjuntos de aceptación.

También existe una variante de los autómatas de Büchi generalizados donde los conjuntos de aceptación están definidos sobre transiciones en vez de estados.

Los autómatas de Büchi generalizados son tan expresivos como los autómatas de Büchi no deterministas, y la conversión de un ABG a un ABN equivalente (llamada *degeneralización*) es sencilla:

Teorema 3.3.3. *Sea $\mathcal{A} = (\Sigma, Q, I, \delta, \{F_1, \ldots, F_k\})$ un ABG. Si construimos un autómata de Büchi $\mathcal{A}' = (\Sigma, Q', I', \delta', F')$ donde:*

- $Q' = Q \times \{i \mid 0 < i \leq k\}$
- $I' = I \times \{i\}$ *para algún* $0 < i \leq k$
- $\delta'((s,i),a) = \begin{cases} (\delta(s,a), (i \bmod k) + 1) & si\ s \in F_i \\ (\delta(s,a), i) & si\ s \notin F_i \end{cases}$
- $F' = F \times \{i\}$ *para algún* $0 < i \leq k$

Entonces $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{A}')$.

El autómata resultante es uno donde la estructura del ABG original está copiada tantas veces como su cantidad de conjuntos de aceptación. Cada copia se encarga de un conjunto de aceptación particular, y cuando se llega a un estado de aceptación se transiciona a la siguiente copia. El nuevo AB pide que se visiten los estados de aceptación de cualquiera de las copias infinitamente a menudo, lo que sólo es posible realizando un ciclo que visita todos los niveles en orden, por lo que aceptan los mismos lenguajes. La construcción de este autómata puede dejar estados no alcanzables en los distintos niveles, que se remueven o directamente no se crean dependiendo de la implementación.

3.3.5. Conversión de una fórmula LTL a un ABN

Para convertir una fórmula LTL a un autómata de Büchi tradicionalmente se usa alguna variante del algoritmo de Gerth, Peled, Vardi y Wolper[GPVW95]. Describiremos brevemente este proceso de la manera explicada en [SB00].

La construcción de un AB desde una fórmula LTL es exponencial respecto al tamaño de la fórmula, por lo que inicialmente se intenta reducirla a una fórmula equivalente de acuerdo a un conjunto de *reglas de reescritura*, algunas de las cuales mencionaremos a continuación:

$(\mathsf{X}\phi)\ \mathsf{U}\ (\mathsf{X}\psi) \equiv \mathsf{X}(\phi\mathsf{U}\psi)$
$(\mathsf{X}\phi) \wedge (\mathsf{X}\psi) \equiv \mathsf{X}(\phi \wedge \psi)$
$(\phi\mathsf{R}\psi) \wedge (\psi\mathsf{R}\tau) \equiv \phi\ \mathsf{R}\ (\psi \wedge \tau)$
$(\phi\mathsf{R}\tau) \wedge (\psi\mathsf{R}\tau) \equiv (\phi \vee \psi)\ \mathsf{R}\ \tau$
$\mathsf{X}\mathit{true} \equiv \mathit{true}$
$\phi\ \mathsf{U}\ \mathit{false} \equiv \mathit{false}$
$\mathsf{FX}\phi \equiv \mathsf{XF}\phi$
$\mathsf{GGF}\phi \equiv \mathsf{GF}\phi$
$\mathsf{FGF}\phi \equiv \mathsf{GF}\phi$
$\mathsf{XGF}\phi \equiv \mathsf{GF}\phi$
$\mathsf{F}(\phi \wedge \mathsf{GF}\psi) \equiv (\mathsf{F}\phi) \wedge (\mathsf{GF}\psi)$
$\mathsf{G}(\phi \vee \mathsf{GF}\psi) \equiv (\mathsf{G}\phi) \vee (\mathsf{GF}\psi)$
$\mathsf{X}(\phi \wedge \mathsf{GF}\psi) \equiv (\mathsf{X}\phi) \wedge (\mathsf{GF}\psi)$
$\mathsf{X}(\phi \vee \mathsf{GF}\psi) \equiv (\mathsf{X}\phi) \vee (\mathsf{GF}\psi)$

La fórmula reescrita se convierte a la *forma normal negativa*, donde sólo están permitidos los operadores U, R, X, $\wedge$, $\vee$ y el operador $\neg$ que sólo puede aplicarse a proposiciones atómicas, haciendo uso de las *reglas de dualidad* de los operadores:

$\neg\mathsf{X}\phi \equiv \mathsf{X}\neg\phi$
$\neg\mathsf{F}\phi \equiv \mathsf{G}\neg\phi$
$\neg\mathsf{G}\phi \equiv \mathsf{F}\neg\phi$
$\neg(\phi\mathsf{U}\psi) \equiv (\neg\phi)\ \mathsf{R}\ (\neg\psi)$
$\neg(\phi\mathsf{R}\psi) \equiv (\neg\phi)\ \mathsf{U}\ (\neg\psi)$

Una vez en la forma normal negativa, queremos obtener una *cobertura* de la fórmula:

Definición 3.3.11. Una fórmula LTL es *elemental* si es una constante, una proposición atómica o comienza con X.

La *cobertura elemental* de un conjunto de fórmulas LTL $\{\phi_1, \ldots, \phi_i\}$ es un conjunto de conjuntos de fórmulas elementales $\{\{\psi_{1,1}, \ldots, \psi_{1,k_1}\}, \ldots, \{\psi_{j,1}, \ldots, \psi_{j,k_j}\}\}$ tales que $\bigwedge_i \phi_i \Leftrightarrow \bigvee_j \bigwedge_k \psi_{j,k}$.

Para obtener la cobertura, la formula se debe volver a reescribir usando las *reglas de expansión* de los operadores U y R:

$$\phi\mathsf{U}\psi \equiv \psi \vee (\phi \wedge \mathsf{X}(\phi\mathsf{U}\psi)) \qquad \phi\mathsf{R}\psi \equiv \psi \wedge (\phi \vee \mathsf{X}(\phi\mathsf{R}\psi))$$

Luego de obtener la cobertura de la fórmula, procedemos recursivamente buscando las coberturas de las subfórmulas elementales X obtenidas hasta que no se encuentran más conjuntos. Con esto se obtiene un conjunto cerrado de coberturas, donde la cobertura de cada subfórmula X está incluída en el conjunto. Con este conjunto podemos construir el autómata, donde:

- Cada conjunto dentro de una cobertura será un estado en el autómata resultante cuya etiqueta es la conjunción de sus proposiciones atómicas que ocurren en el conjunto, junto con un estado inicial auxiliar *init*.
- La relación de transición se obtiene conectando *init* a cada estado de la cobertura de la fórmula original, y a cada estado con los estados de las coberturas de sus subfórmulas X, donde la etiqueta de cada transición es la del estado de destino.
- Los conjuntos de aceptación estan definidos por cada fórmula elemental de la forma $\mathsf{X}(\phi\mathsf{U}\psi)$ en el conjunto cerrado de coberturas, y contienen todos los estados cuyas etiquetas implican ψ o no implican $\phi\mathsf{U}\psi$.

Finalmente el autómata se degeneraliza obteniendo un autómata de Büchi no determinista.

Ejemplo 3.3.2. *Veamos cómo funciona el método para la fórmula* $(a \vee b)\ \mathsf{U}\ \mathsf{G}\ b$. *Reescribimos la fórmula usando las reglas de expansión:*

$$\begin{aligned}(a \vee b)\ \mathsf{U}\ \mathsf{G}\ b &\equiv (a \vee b)\ \mathsf{U}\ (\mathit{false}\ \mathsf{R}\ b) \\ &\equiv (\mathit{false}\ \mathsf{R}\ b) \vee ((a \vee b) \wedge \mathsf{X}((a \vee b)\ \mathsf{U}\ (\mathit{false}\ \mathsf{R}\ b))) \\ &\equiv (b \wedge \mathsf{X}(\mathit{false}\ \mathsf{R}\ b)) \\ &\quad \vee (a \wedge \mathsf{X}((a \vee b)\ \mathsf{U}\ (\mathit{false}\ \mathsf{R}\ b))) \\ &\quad \vee (b \wedge \mathsf{X}((a \vee b)\ \mathsf{U}\ (\mathit{false}\ \mathsf{R}\ b)))\end{aligned}$$

La cobertura elemental es entonces el conjunto $\{\{b, \mathsf{X}(\mathit{false}\ \mathsf{R}\ b)\}, \{a, \mathsf{X}((a \vee b)\ \mathsf{U}\ (\mathit{false}\ \mathsf{R}\ b))\}, \{b, \mathsf{X}((a \vee b)\ \mathsf{U}\ (\mathit{false}\ \mathsf{R}\ b))\}\}$. *El conjunto es cerrado porque las coberturas de las subfórmulas elementales* $\mathsf{X}(\mathit{false}\ \mathsf{R}\ b)$ *y* $\mathsf{X}((a \vee b)\ \mathsf{U}\ (\mathit{false}\ \mathsf{R}\ b))$ *ya están en el mismo. La única subfórmula de tipo* $\mathsf{X}(\phi \mathsf{U} \psi)$ *en la cobertura,* $\mathsf{X}((a \vee b)\ \mathsf{U}\ (\mathit{false}\ \mathsf{R}\ b))$, *genera el único conjunto de aceptación, compuesto por los estados con etiqueta* b *porque* $b \Rightarrow (\mathit{false}\ \mathsf{R}\ b)$.

El autómata resultante, mostrado en la figura 3.4, es $(2^{\{a,b\}}, Q, I, \delta, F)$ *donde:*

- $Q = \{\ \mathit{init},$
  $q_1 : \{b, \mathsf{X}(\mathit{false}\ \mathsf{R}\ b)\},$
  $q_2 : \{a, \mathsf{X}((a \vee b)\mathsf{U}(\mathit{false}\ \mathsf{R}\ b))\},$
  $q_3 : \{b, \mathsf{X}((a \vee b)\mathsf{U}(\mathit{false}\ \mathsf{R}\ b))\}$
  $\}$
- $I = \{\mathit{init}\}$
- $\begin{aligned}\delta(\mathit{init}, a) &= \{q_2\} \\ \delta(\mathit{init}, b) &= \{q_1, q_3\} \\ \delta(q_1, b) &= \{q_1\} \\ \delta(q_2, b) &= \{q_1, q_3\} \\ \delta(q_2, a) &= \{q_2\} \\ \delta(q_2, b) &= \{q_1, q_3\} \\ \delta(q_3, b) &= \{q_1, q_3\} \\ \delta(q_3, a) &= \{q_2\}\end{aligned}$
- $F = \{\{q_1, q_3\}\}$

Aparte de la reescritura de la fórmula original, existen varias técnicas aplicables en distintas fases del proceso para reducir el tamaño del autómata resultante, como la optimización booleana de las coberturas, técnicas de bisimulación, y detección de componentes fuertemente conexas para reducir los conjuntos de aceptación[SB00]. También existen algoritmos alternativos donde la fórmula se convierte en un autómata alternante y luego en un autómata de Büchi generalizado sobre transiciones[GO01][Tau06].

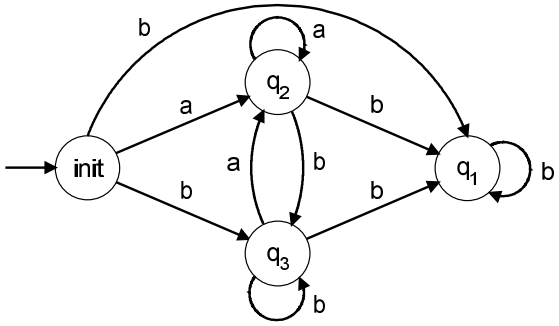


Figura 3.4: Resultado de la conversión de $(a \vee b)\mathsf{U}\ \mathsf{G}\ b$ a un autómata de Büchi generalizado

Autómatas de Rabin

La última clase de ω-autómatas que veremos son los *autómatas de Rabin*. La expresividad de los autómatas de Rabin deterministas(ARD) es igual a la de los lenguajes ω-regulares al igual que los autómatas de Büchi y los autómatas de Büchi generalizados. Estos autómatas son una parte esencial de nuestro trabajo puesto que los métodos de verificación de propiedades basados en autómatas para modelos concurrentes requieren que los autómatas de la propiedad sean deterministas[CY95].

Definición 3.3.12. Una ejecución σ es aceptada por un autómata de Rabin $\mathcal{A}$ si y solo si existe un par $(L_i, H_i) \subseteq F$ tal que $\mathit{inft}(\sigma) \cap L_i \neq \emptyset$ e $\mathit{inft}(\sigma) \cap H_i = \emptyset$, donde $F = \{(L_i, H_i) \mid L_i, H_i \subseteq Q\}$ es un conjunto de pares de aceptación de Rabin.

Intuitivamente, una palabra es aceptada si induce una ejecución σ que sólo visita finitas veces los estados en el primer conjunto de algún par de aceptación, y visita infinitamente a menudo algún estado del segundo conjunto del mismo par. Alternativamente, si complementamos el primer conjunto de un par de aceptación obtenemos los estados a los que debe estar confinada $\mathit{inft}(\sigma)$ para dicho par.

Ejemplo 3.3.3. *Vimos anteriormente que el lenguaje generado por la expresión ω-regular $(a|b)^*b^\omega$ no puede ser generado por un autómata de Büchi determinista. Los autómatas de Rabin, que nos permiten especificar qué estados no pueden ser visitados infinitas veces para que una palabra sea aceptada, pueden generar este lenguaje sin recurrir al no determinismo, como podemos ver en la figura 3.5.*

3.3.6. Conversión de un ABN a un ARD

El procedimiento de conversión de un autómata de Büchi no determinista a un autómata de Rabin se basa en el algoritmo de determinización de autómatas para palabras finitas. El problema de aplicar esta construcción a un ω-autómata no determinista es que existen palabras no aceptadas que al recorrer el autómata pasan por un estado de aceptación y luego *abortan* en una cantidad finita de

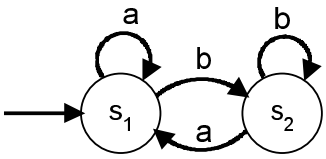


Figura 3.5: Autómata de Rabin determinista de $(a|b)^*b^\omega$. El conjunto de aceptación es $\{(\{s_1\},\{s_2\})\}$.

pasos, o sea, llegan a un estado en el que no pueden consumir el siguiente símbolo. Esto no es detectado por el algoritmo de determinización, y resulta en ejecuciones que pueden proseguir haciendo que se acepten palabras por error.

Ejemplo 3.3.4. *En la figura 3.6 podemos ver el ABN de la fórmula* FGa*, que no es expresable con un autómata de Büchi determinista, y el resultado de aplicarle el algoritmo de determinización para autómatas finitos. El autómata determinista resultante acepta las palabras de la forma* $(a(\neg a))^\omega$*, pero estas palabras no son aceptadas por el ABN puesto que en el momento que la ejecución ingresa en* s_2 *el autómata deja de poder consumir* $\neg a$ *y la ejecución debe abortar.*

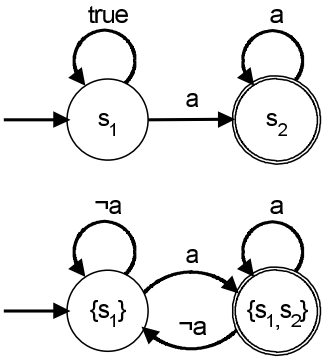


Figura 3.6: Determinización fallida del autómata de Büchi no determinista de la fórmula FGa usando el algoritmo para autómatas sobre palabras finitas.

La solución de Safra[Saf89] a este problema consiste en usar este algoritmo, pero al llegar a un estado de aceptación se siguen todas las ejecuciones que surgen del mismo para poder detectar las que abortan. Esta información es almacenada en lo que se denominan *árboles de Safra*:

Definición 3.3.13. Sea Q un conjunto finito. Un *árbol de Safra* sobre Q es un árbol finito ordenado $(N, h, <, l, m, h)$ donde:

- N es un conjunto de *nodos* con nombres distintos en $\{1,\ldots,2|Q|\}$
- $h : N \to 2^N$ es una función de *descendencia*
- $<$ es un orden total sobre los hijos de cada nodo al que llamaremos *edad*
- $m : N \to \mathbb{B}$ es una función de *marcado*
- $l : N \to 2^Q - \{\emptyset\}$ es una función de *etiquetado* donde para todo nodo n:

- $l(n) \supseteq \bigcup_{j \in h(n)} l(j)$
- $\forall i, j \in h(n). i \neq j \Rightarrow l(i) \cap l(j) = \emptyset$

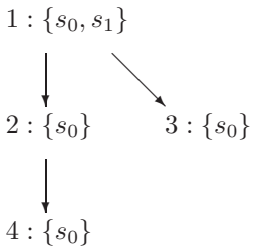
El siguiente teorema de [Kle05], provee un algoritmo de construcción de un autómata de Rabin determinista equivalente a un autómata de Büchi dado utilizando árboles de Safra.

Teorema 3.3.4. *Sea $(\Sigma, Q, \{q_0\}, \delta, F)$ un autómata de Büchi no determinista. Podemos construir un autómata de Rabin determinista equivalente $(\Sigma, Q', \{q_0'\}, \delta', F')$ donde:*

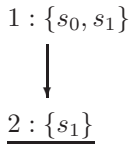
- *Q' es el conjunto de todos los árboles de Safra sobre Q.*
- *q_0' es el árbol de Safra con un único nodo desmarcado de nombre 1 y etiqueta $\{q_0\}$.*
- *Dado un árbol de Safra t y un símbolo $a \in \Sigma$, $\delta'(t, a)$ es el resultado de:*
  1. *Desmarcar todos los nodos de t.*
  2. *Para cada nodo n de t tal que $l(n) \cap F \neq \emptyset$, crear un nuevo nodo hijo de n que será el más joven si posee hermanos, con etiqueta $l(n) \cap F$ y un nombre que no esté en uso.*
  3. *Para cada nodo n de t, reemplazar $l(n)$ por $\bigcup_{q \in l(n)} \delta(q, a)$.*
  4. *Para cada etiqueta $q \in Q$ compartida entre nodos hermanos, removerla de todos los nodos más jóvenes y sus respectivas descendencias.*
  5. *Remover todos los nodos cuya etiqueta es vacía.*
  6. *Si la unión de las etiquetas de los descendientes de un nodo es igual a la etiqueta del nodo, marcarlo y remover sus descendientes.*
- *$F' = \{(L_1, U_1), \ldots, (L_{2n}, U_{2n})\}$ donde L_i contiene todos los árboles de Safra cuyo nodo i está marcado y U_i contiene todos los árboles de Safra sin un nodo i*

Ejemplo 3.3.5. *Convirtamos el ABN de la figura a un ARD. El algoritmo comienza con el árbol de Safra inicial T_0 de un nodo desmarcado con nombre 1 y etiqueta $\{s_0\}$, que denotaremos con $1 : \{s_0\}$. Consumiendo a desde este nodo obtenemos el mismo árbol luego de aplicar el procedimiento. Al consumir b, obtenemos un nuevo árbol de Safra T_1 con un nodo $1 : \{s_0, s_1\}$.*

Como este nuevo árbol posee un nodo con estado en el conjunto de aceptación, en las próximas transiciones el árbol resultante T_2 adquiere un nuevo nodo hijo en el paso número 2 con el que se seguirán las ejecuciones que parten del estado de aceptación s_1, $2 : \{s_1\}$. Al realizar una transición desde este árbol, en el paso 2 se crea un tercer nodo gemelo del segundo y un cuarto nodo hijo del segundo que también es idéntico, obteniendo el siguiente árbol:



Al realizar el paso 3 consumiendo a en este árbol nos quedan 3 nodos con etiquetas iguales vacías, por lo que los nodos se eliminan y obtenemos nuevamente el árbol inicial. De consumir b, los 3 nodos pasan a tener etiqueta $\{s_1\}$*, por lo que la etiqueta del hermano menor del segundo nivel se elimina en el paso 4 y luego se elimina el nodo en el paso 5, seguido por el marcado del nodo 2 y la eliminación de su hijo por tener las mismas etiquetas. De esta manera nos queda el siguiente árbol de Safra* T_3*:*



Aplicando nuevamente la sucesión de pasos a este estado llegamos a sí mismo a través de b y nuevamente al árbol T_0 *a través de a, culminando la conversión.*

El conjunto de aceptación resultante es $\{(\emptyset,\emptyset),(\{T_3\},\{T_0,T_1\}),(\emptyset,\{T_1,T_2,T_3,T_4\}),(\emptyset,\{T_1,T_2,T_3,T_4\})\}$*. Como un par de aceptación no acepta ninguna palabra infinita si su primer conjunto es vacío, el conjunto de aceptación final es* $\{(\{T_3\},\{T_0,T_1\})\}$*.*

El resultado de este algoritmo, mostrado en la figura 3.7, no es óptimo como el de la figura 3.5, pero se le puede aplicar una serie de optimizaciones [Kle05] sobre las que no hablaremos en este trabajo.

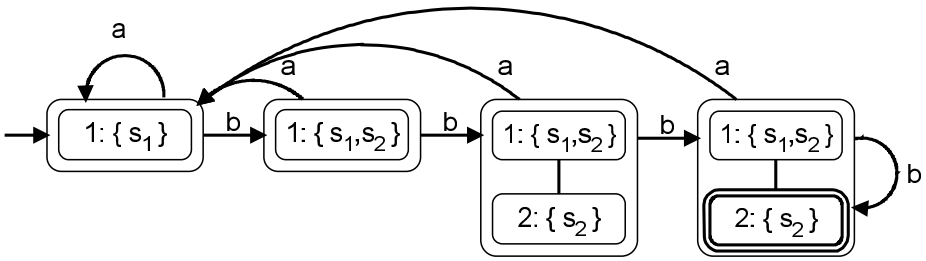


Figura 3.7: Autómata de Rabin determinista de $(a|b)^*b^\omega$ resultante de aplicar la construcción de Safra al autómata de la figura 3.3.5.

Capítulo 4

Diagramas de decisión binarios

Los *diagramas de decisión binarios* son las estructuras de datos sobre las que se construyen los algoritmos de model checking simbólico cualitativo, y junto con algunas modificaciones que veremos al final de este capítulo también son aplicables al model checking cuantitativo sobre el que trata este trabajo.

4.1. Funciones booleanas

Definición 4.1.1. Una *variable booleana* x es una variable que contiene un valor de $\mathbb{B} = \{0, 1\}$.

Dado un conjunto de variables booleanas, podemos definir funciones sobre los valores que almacenan:

Definición 4.1.2. Una *función booleana* f de n argumentos es una función de $\mathbb{B}^n$ en $\mathbb{B}$. Con $f(x_1, x_2, \ldots, x_n)$ denotaremos la función booleana f donde reemplazamos las ocurrencias de las variables booleanas $x_1, x_2, \ldots, x_n$ por sus respectivos valores.

El cálculo del valor de una función booleana puede dividirse fácilmente tomando una variable y calculando la disyunción de las funciones que resultan de asignarle 0 y 1 a la variable en la función original, lo que se conoce como *expansión de Shannon*. A estas funciones donde el valor de una variable ha sido fijado las llamaremos *cofactores*:

Definición 4.1.3. Los *cofactores* $f_{\bar{x}}, f_x$ de una función booleana f son las funciones que resultan de evaluar parcialmente f reemplazando las ocurrencias de la variable x en f por 0 y 1 respectivamente.

Lema 4.1.1. *Para toda variable booleana x y función booleana f tenemos que:*

$$f \equiv \bar{x} \cdot f_{\bar{x}} + x \cdot f_x$$

donde $\bar{x}$ es el complemento de x.

4.2. Almacenamiento de funciones booleanas

En esta sección estudiaremos distintas estructuras para almacenar funciones booleanas. Analizaremos las distintas ventajas y desventajas en la representación de cada una, dando énfasis al tamaño de la estructura y la complejidad de las operaciones sobre la misma.

4.2.1. Fórmulas

La fórmula proposicional misma, que es la representación más común en un escrito, es compacta como deseamos. Componer funciones es simple, pero se hace difícil ver si la función es satisfacible, o sea que existen valores para los que la función es verdadera, o si es válida, que se da cuando la función es verdadera para cualquier valor, o si es equivalente a otra fórmula. De hecho, probar que una función es satisfacible es un problema NP-Completo[Coo71]. Para mitigar este problema, las fórmulas se suelen expresar en la *forma normal disyuntiva* donde la fórmula es de la forma $\bigvee_{j=1}^{l} \left(\bigwedge_{i=1}^{k_j} x_j^i \right)$, o en la *forma normal conjuntiva* de la forma $\bigwedge_{j=1}^{l} \left(\bigvee_{i=1}^{k_j} x_j^i \right)$. En la forma disyuntiva probar satisfacción toma tiempo polinomial, obtenido a cambio de una prueba de validez cara, mientras que la situación se invierte para la forma conjuntiva. Como la conversión entre ambas formas es de complejidad exponencial, no son una alternativa viable.

4.2.2. Matrices

También pueden usarse matrices, dividiendo las variables en cada eje según algún criterio y guardando cada resultado de la función en una celda de la matriz. Un caso común es el de la tabla de verdad, donde no se dividen las variables resultando en un vector de valores. Veamos la tabla de verdad de la función $(x + y) \cdot z$:

x	y	z	$(x + y) \cdot z$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Al ser exhaustiva esta representación es poco eficiente, teniendo 2^n celdas para n variables, pero la simplicidad de la representación hace que las operaciones sobre la misma sean veloces. Cuando la matriz contiene un elemento dominante (generalmente ceros) se le pueden aplicar técnicas de compresión que reducen significativamente el tamaño de la estructura sin sacrificar la velocidad de las operaciones, obteniendo *matrices ralas*.

4.2.3. Árboles

Aplicando la expansión de Shannon sucesivamente para cada variable booleana obtenemos un *árbol de decisión binario* o *árbol de Shannon* como el de la figura 4.1, donde cada nodo interno tiene de nombre una variable y como sub-árboles ambos cofactores sobre dicha variable, hasta llegar finalmente a las hojas donde están los valores de la función. Usaremos una línea punteada para unir el sub-árbol donde la variable toma el valor 0 y una línea sólida para el valor 1.

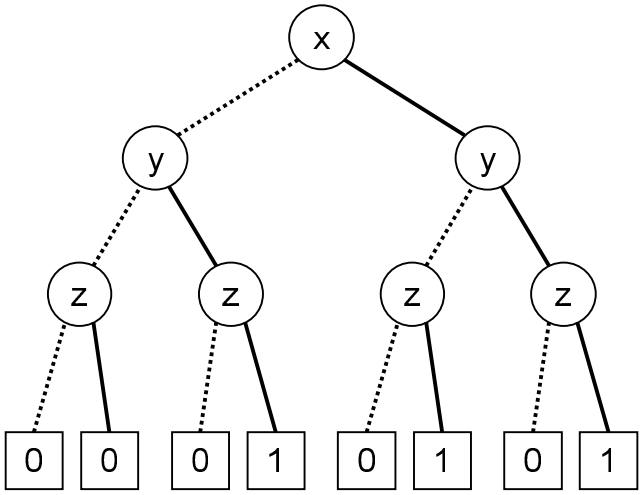


Figura 4.1: Arbol de decisión binario de la funcion $(x + y) \cdot z$

Para ver el valor de la función simplemente partimos desde la raíz y elegimos las líneas punteadas o enteras según el valor de la variable que ocurre en el nodo hasta llegar a una hoja.

Los árboles de decisión binarios requieren $2^{n+1} - 1$ nodos para representar funciones de n variables. Sin embargo poseen mucha información redundante que podríamos eliminar si relajáramos los requerimientos de la estructura.

4.2.4. BDD

Al ver un árbol de Shannon como un grafo dirigido acíclico podemos aplicarle distintas optimizaciones. Así surgen los *diagramas de decisión binarios*.

Definición 4.2.1. Un *diagrama de decisión binario* (BDD) es un grafo dirigido acíclico con una raíz cuyos nodos internos, que siempre poseen 2 aristas salientes, tienen como etiqueta una variable booleana y cuyos nodos terminales tienen etiqueta 0 o 1.

4.3. Reducción de BDD

Ahora que relajamos nuestros requerimientos podemos eliminar los datos redundantes:

4.3.1. Unión de subgrafos isomorfos

Notemos que todas las hojas tienen los valores 0 o 1, así que podemos redirigir todos las flechas que van a un nodo 0 a un único nodo 0 y todas las flechas que van a un 1 a un único nodo 1 y eliminar el resto. El resultado de aplicar esta reducción al árbol de Shannon de $(x + y) \cdot z$ se puede ver en la figura 4.2.

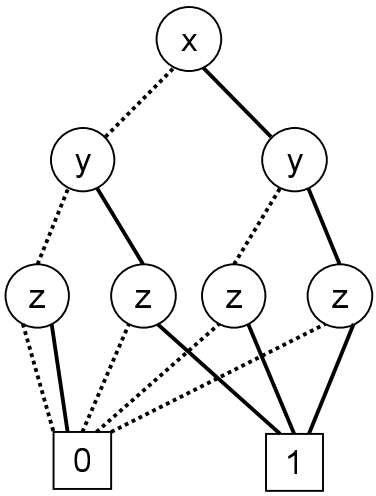


Figura 4.2: Primera reducción sobre el árbol de decisión binario de $(x + y) \cdot z$

Podemos generalizar esta optimización a todos los subgrafos que son estructuralmente equivalentes, dejando una sola copia de cada uno y obteniendo el grafo de la figura 4.3.

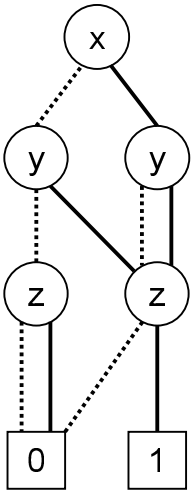


Figura 4.3: Segunda reducción sobre $(x + y) \cdot z$

4.3.2. Eliminación de nodos cuyos hijos son isomorfos entre sí

Además tenemos nodos que llegan al mismo destino o a subgrafos con la misma estructura por ambas salidas. Esto significa que la función que representa el subgrafo es independiente de la variable booleana de la etiqueta del nodo, por lo que podemos redirigir todas las flechas entrantes al nodo directamente al destino y eliminarlo. La aplicación de esta reducción al grafo de la figura anterior resulta en el de la figura 4.4.

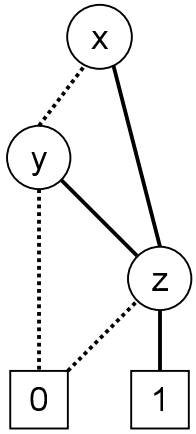


Figura 4.4: Tercera reducción sobre $(x + y) \cdot z$. Este BDD es reducido.

Cuando a un BDD no se le pueden eliminar más nodos aplicando las optimizaciones anteriores, diremos que es *reducido*.

4.4. Ordenando BDD

Si bien los BDDs reducidos son compactos, todavía podemos mejorar la situación. Es posible que una variable booleana ocurra más de una vez en un camino del BDD, y nos es difícil ver si dos BDD son equivalentes. Por ejemplo, los BDD de la figura 4.5 representan la misma función de la figura anterior pero la estructura es distinta por cómo están dispuestas las variables.

Para resolver esto pediremos que exista un *orden* sobre las variables.

Definición 4.4.1. Sea $[x_1, \ldots, x_n]$ una lista ordenada de variables donde no hay elementos duplicados. Un BDD tiene el *orden* $[x_1, \ldots, x_n]$ si todas las variables que ocurren en sus nodos están en la lista, y para todas las variables x_j que siguen a x_i en el BDD por cualquier camino se da $i < j$. Un BDD es *ordenado* si tiene un orden para alguna lista de variables. Dos BDD B1 y B2 tienen *órdenes compatibles* si para cualquier par de variables x, y de B1 tal que x ocurre antes que y, entonces también lo hace en B2 y viceversa.

Así, podemos ver que el primer BDD de la figura 4.5 no tiene orden, el segundo tiene orden $[x, y, z]$, el tercero $[y, z, x]$ y el cuarto $[z, x, y]$.

Cabe aclarar que el tamaño de un BDD puede variar entre lineal y exponencial respecto a la cantidad de variables según el orden elegido, como po-

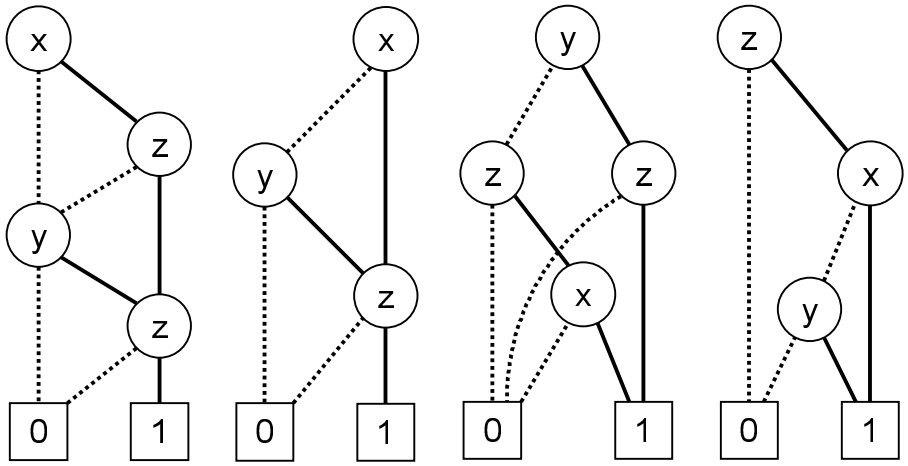


Figura 4.5: Cuatro BDD de $(x + y) \cdot z$

demos ver en la figura 4.6. Encontrar un orden óptimo es un problema NP-completo[BW96], pero existen heurísticas [Par02] con las que se obtienen buenos resultados.

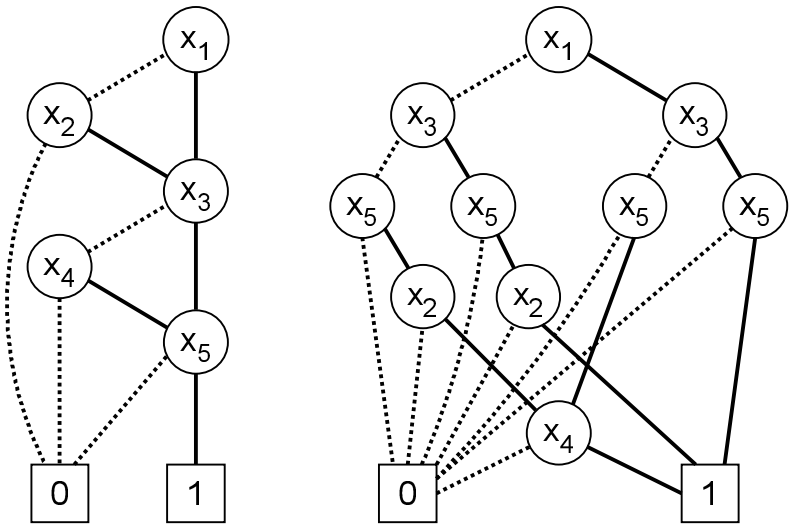


Figura 4.6: BDD reducidos de $(x_1 + x_2) \cdot (x_3 + x_4) \cdot x_5$, con órdenes $[x1, x2, x3, x4, x5]$ y $[x1, x3, x5, x2, x4]$

4.5. ROBDD

Los BDD ordenados y reducidos (ROBDD) poseen una propiedad importante que los hace más convenientes:

Teorema 4.5.1. *El BDD ordenado y reducido que representa a una función booleana es único para todos los órdenes compatibles.*

Esto hace que algunas operaciones sean muy baratas en costo computacional si elegimos un orden a priori, por ejemplo:

- Para probar la validez de una función, basta con ver que su ROBDD es el de la constante 1.
- Para probar satisfacibilidad, basta con ver que su ROBDD no es el de la constante 0.
- Para probar equivalencia con otra función, basta con comparar la estructura de los dos ROBDD.
- Para encontrar las variables redundantes en una fórmula basta con ver qué variables no ocurren en su ROBDD.

Esto también permite almacenar una sola copia de cada subgrafo y reusarla en todos los BDD con órdenes compatibles que lo contienen, reduciendo la memoria requerida para almacenar más de un BDD.

En adelante asumiremos que se ha elegido un orden y nos referiremos a los ROBDD como BDD.

4.6. Algoritmos sobre BDD

A continuación llamaremos $lo(n)$ al nodo apuntado por la flecha punteada que parte del nodo n y $hi(n)$ al apuntado por la flecha sólida. Además, abreviaremos con $\hat{x}$ la secuencia finita $x_1, \ldots, x_n$, que contiene a todas las variables de la forma x_i.

4.6.1. Reducción

El algoritmo de reducción `reduce` se encarga de realizar las optimizaciones que mencionamos anteriormente para obtener un BDD reducido, de complejidad $\mathcal{O}(|B|log|B|)$. La reducción se hace en dos pasadas: una que etiqueta los nodos de modo que si dos nodos poseen la misma etiqueta computan la misma función booleana, y la segunda que une estos nodos que computan las mismas funciones y las aristas hacia los mismos para reducir el BDD.

1. Asignamos inicialmente etiquetas 0 y 1 a B_0 y B_1, que representan las funciones constantes *false* y *true* respectivamente.
2. Recorremos el arbol nivel por nivel desde las hojas hacia la raíz, etiquetando cada nodo n de la siguiente forma:
   - Si la etiqueta de $lo(n)$) es igual a la etiqueta de $hi(n)$, entonces ambos hijos de n son isomorfos y por lo tanto el nodo n es redundante, por lo que le asignamos esta etiqueta a n.
   - Si existe otro nodo m con la misma variable booleana de n tal que las etiquetas de $hi(m)$ y $lo(m)$ son iguales a $hi(n)$ y $lo(n)$ respectivamente, entonces los subgrafos con raíz en n y m son isomorfos, por lo que le asignamos la etiqueta de m a n.
   - Si no se da ninguno de los casos anteriores, se le asigna una etiqueta que no esté en uso.

3. Realizamos un recorrido *bottom-up* del BDD en el que se redirigen las aristas a un nodo único por etiqueta y se elimina el resto.

4.6.2. Complementación

El complemento de un BDD B, $\texttt{not}(B)$, se obtiene intercambiando las etiquetas de los terminales B_0 y B_1 o bien redirigiendo cada transición hacia B_0 a B_1 y viceversa, dependiendo de la implementación.

4.6.3. Composición con operadores binarios

Uno de los algoritmos para BDD de uso más común es `apply`, que dados dos BDD B_f y B_g representando funciones f y g respectivamente y un operador booleano binario *op*, se encarga de computar el BDD de $f\ op\ g$. Su funcionamiento se basa en la expansión de Shannon de $f\ op\ g$:

$$\begin{aligned} f\ op\ g &\equiv \bar{x_i} \cdot (f\ op\ g)_{\bar{x_i}} + x_i \cdot (f\ op\ g)_{x_i} \\ &\equiv \bar{x_i} \cdot (f_{\bar{x_i}}\ op\ g_{\bar{x_i}}) + x_i \cdot (f_{x_i}\ op\ g_{x_i}) \end{aligned}$$

O sea que para cada variable x_i podemos dividir el problema en dos, uno donde x_i vale 0 y otro donde vale 1, y repetir esto recursivamente hasta tener dos terminales donde aplicar *op* directamente. Como podemos elegir cualquier variable en cada división, elegimos seguir el orden existente en B_f y B_g para obtener un OBDD del mismo orden. Este método produce un BDD con muchos subgrafos duplicados, por lo que el BDD resultante no es reducido.

Calculemos $\texttt{apply}(op, B_f, B_g)$, donde r_f y r_g son los nodos raíz de B_f y B_g respectivamente:

1. Si r_f y r_g son ambos terminales con etiquetas e_f y e_g, devolvemos $B_{e_f\ op\ e_g}$.

2. Si r_f y r_g tienen como etiqueta la misma variable booleana x_i, devolvemos un BDD cuya raíz tiene etiqueta x_i, y está unido al resultado de $\texttt{apply}(op, lo(r_f), lo(r_g))$ por la línea de puntos y al resultado de $\texttt{apply}(op, hi(r_f), hi(r_g))$ por la línea sólida.

3. Si r_f tiene como etiqueta x_i y r_g es terminal o tiene como etiqueta x_j tal que $i < j$, entonces B_g es independiente de x_i. Como $g \equiv g_{\bar{x_i}} \equiv g_{x_i}$, devolvemos un BDD cuya raíz tiene etiqueta x_i, y está unido al resultado de $\texttt{apply}(op, lo(r_f), r_g)$ por la línea de puntos y al resultado de $\texttt{apply}(op, hi(r_f), r_g)$ por la línea sólida.

4. Si r_g tiene como etiqueta x_i y r_f es terminal o tiene como etiqueta x_j tal que $i < j$, nos encontramos en el caso simétrico al anterior.

El BDD resultante puede no estar reducido, por lo que finalmente corremos `reduce` sobre el mismo.

La figura 4.7 muestra cómo `apply` opera sobre dos BDD. En la misma los nodos de los argumentos a la llamada original han sido etiquetados para poder referenciarlos en el resultado, donde la etiqueta de cada nodo contiene el par de argumentos con los que se llama recursivamente a la función. De esta manera se puede ver que en el transcurso de la composición se producen llamadas repetidas a `apply` con los mismos argumentos. Si utilizamos técnicas de programación

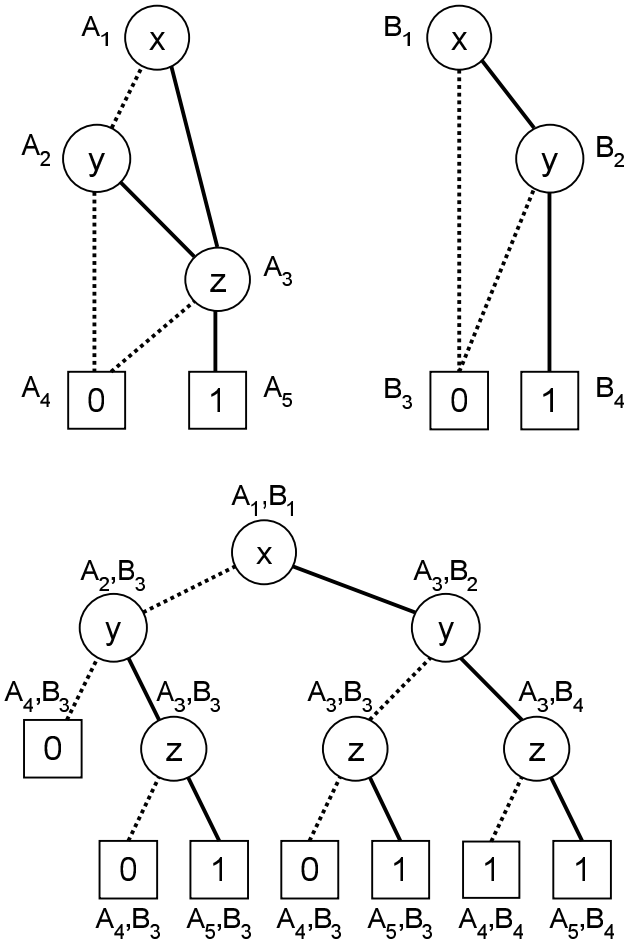


Figura 4.7: `apply`$(+, B_{(x+y)\cdot z}, B_{x\cdot y})$

dinámica y guardamos los resultados de cada llamada para evitar recomputarlos, el algoritmo pasa de tener complejidad exponencial (cada nodo interno produce 2 llamadas recursivas a `apply`) a tener una cota de $2|B_f||B_g|$ además de ahorrarnos la posterior reducción de los resultados duplicados. El BDD resultante tiene menos de $|B_f||B_g|$ nodos[Bry86].

4.6.4. Cofactores

`restrict`(b, x, B) computa el cofactor de f donde la variable x toma el valor b y f es la función representada por el BDD B. Simplemente consiste en redirigir todas las flechas que llegan a cada nodo n con variable x a $lo(n)$ si $i = 0$ o a $hi(n)$ si $i = 1$.

4.6.5. Cuantificación

`exists`(x, B) computa la cuantificación existencial $\exists x.f(\hat{x})$ donde f es la función representada por el BDD B. Como $\exists x.f(\hat{x}) = f_{\bar{x}}(\hat{x}) + f_x(\hat{x})$, esto se po-

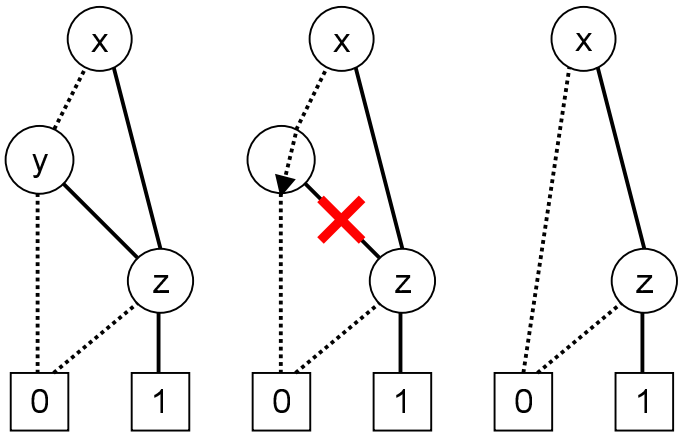


Figura 4.8: `restrict`$(0, y, B_{(x+y)\cdot z})$

dría resolver haciendo `apply`$(+, $`restrict`$(0, x, B), $`restrict`$(1, x, B))$. Sin embargo esta llamada a `apply` también opera antes de la ocurrencia de un nodo con variable x, donde ambos cofactores son idénticos. Una solución más rápida consiste en buscar los nodos n con variable x en el BDD y reemplazarlos por `apply`$(+, lo(n), hi(n))$. La cuantificación universal `forall`, al ser el operador dual de `exists`, es implementada de la misma manera usando $\cdot$ en vez de $+$.

La cuantificación sobre múltiples variables se obtiene realizando llamadas sucesivas para cada variable, y es NP-Completo.

4.6.6. Permutación de variables

`permute`$(B, \langle\hat{x}\rangle, \langle\hat{y}\rangle)$ devuelve para un BDD B el resultado de permutar simultáneamente todas las ocurrencias de $x_i \in \hat{x}$ en la función representada por B por $y_i \in \hat{y}$ y viceversa.

Ejemplo 4.6.1. *El resultado de* `permute`$(B, \langle x\rangle, \langle y\rangle)$ *donde* B *representa a* $x \cdot y + \bar{x} \cdot z$ *es un BDD que representa a la función* $y \cdot x + \bar{y} \cdot z$.

Esto se puede calcular recorriendo la estructura del BDD intercambiando las etiquetas de cada nodo. Este BDD tiene distinto orden al original, por lo que se le debe aplicar un algoritmo de reordenamiento de variables como el de [Som99], que realiza el intercambio velozmente operando *in situ* en el BDD.

4.7. MTBDD

Al ser una estructura que sólo almacena funciones booleanas, la aplicabilidad de los BDD a distintos problemas es limitada. De querer almacenar funciones con imágenes de más de dos elementos en BDD, necesitaríamos codificar los elementos del rango utilizando números binarios y utilizar un BDD para cada bit del resultado de la función. Una solución a este problema son los *multi-terminal BDD* (MTBDD), BDD donde la función representada no tiene imagen en $\mathbb{B}$ sino en cualquier conjunto finito, y cuyo grafo tiene tantas hojas como la imagen de la función tiene elementos. En particular nos concentraremos en los

MTBDD que representan funciones cuya imagen es un conjunto finito arbitrario de números reales en el intervalo $[0, 1]$.

Los algoritmos sobre MTBDD requieren pocos cambios para poder aplicarse a los nuevos nodos terminales. Las cuantificaciones universal y existencial pierden sentido sobre números reales, pero cambiando el operador aplicado sobre los cofactores en las mismas obtenemos algoritmos que calculan sumatorias, productorias, mínimos y máximos sobre la variable elegida, que llamaremos `sumAbstract`, `prodAbstract`, `minAbstract` y `maxAbstract` respectivamente.

Además se agregan nuevos algoritmos a los que ya teníamos, incluídos los de operaciones sobre matrices de números reales almacenadas en MTBDD. Uno importante es `threshold`, que dado un operador de comparación, un número real y un MTBDD nos devuelve un BDD tal que:

$$\texttt{threshold}(\bowtie, t, f)(\hat{x}) = \begin{cases} 1 & \text{si } f(\hat{x}) \bowtie t \\ 0 & \text{caso contrario} \end{cases}$$

`threshold` se puede implementar redirigiendo cada lado que llega a un terminal a 1 si el terminal satisface la comparación y a 0 si no lo hace, y luego reduciendo el BDD resultante.

Los MTBDD continúan siendo una estructura de datos eficiente mientras que la imagen tenga una cantidad pequeña de elementos. De lo contrario, la reducción no es posible y obtenemos algo próximo a un árbol binario de decisión, que es más grande que una matriz.

Capítulo 5

Model Checking

El problema de *model checking* consiste en decidir si el sistema satisface una propiedad, o sea, que todas las ejecuciones posibles en el sistema están contenidas en el lenguaje de la fórmula de la propiedad. En el *model checking cuantitativo* los modelos contienen transiciones con probabilidades, por lo que es necesario ajustar el problema. Para ello se pueden agregar nuevos operadores probabilísticos a la lógica temporal, como por ejemplo $\mathsf{P}_{\geq p}\ \phi$ que es satisfecho si la probabilidad de satisfacer ϕ es mayor o igual que p, o bien se calcula directamente la probabilidad de satisfacción de la propiedad bajo alguna estrategia. En este trabajo optaremos por lo segundo y nos concentraremos en las estrategias que resultan en las probabilidades máximas y mínimas de satisfacer la propiedad, que son las de mayor interés.

5.1. Model checking cuantitativo de propiedades LTL

En los capítulos anteriores vimos cómo representar el funcionamiento de un sistema mediante ejecuciones en sistemas no deterministas probabilísticos, y cómo una propiedad LTL puede ser vista como un ω-autómata que acepta distintas ejecuciones. Para verificar una propiedad utilizaremos el algoritmo descrito por Luca de Alfaro en [dA98], en el que se construye un nuevo SNP, producto del SNP original y el autómata de la propiedad, donde las ejecuciones recorren ambas estructuras simultáneamente.

5.1.1. Construcción del producto

Definición 5.1.1. Sea $(PA, S, A, \kappa, p, s_{in})$ un SNP, y $(2^{PA}, Q, \delta, \{q_{in}\}, F)$ un autómata de Rabin determinista. El *SNP producto* de ambos es un SNP $(PA, S', A, \kappa', p', s'_{in})$ tal que:

- $S' = S \times Q$, donde para todo estado $(s, q) \in S'$ se da $(s, q)[\![x]\!] = s[\![x]\!]$
- $\kappa'(s, q) = \kappa(s)$ para todo estado $(s, q) \in S'$
- $p((s', q') \mid (s, q), a) = \begin{cases} p(s' \mid s, a) & \text{si } \delta(q, L(s')) = q' \\ 0 & \text{caso contrario} \end{cases}$

- $s'_{in} = (s_{in}, \delta(q_{in}, L(s_{in})))$

Notemos que por la construcción de la relación de transición, todas las transiciones en el SNP que puedan llegar a formar parte de ejecuciones aceptadas por la fórmula permanecen en el SNP producto con la misma probabilidad.

Por los lemas vistos en el capítulo 2 sabemos que las ejecuciones eventualmente se confinan a las CT dentro del SNP. Por otro lado, los pares de aceptación del ARD de la fórmula nos dicen a qué conjuntos de estados se debe confinar una ejecución y cuáles estados deben visitarse infinitamente a menudo para que se satisfaga la fórmula. Por lo tanto extenderemos los pares de aceptación para que se ajusten a los estados del SNP, y buscaremos las CT que satisfagan este nuevo criterio de aceptación:

Definición 5.1.2. Sea S el conjunto de estados del SNP original y $F = \{(H_1, L_1), \ldots, (H_m, L_m)\}$ el criterio de aceptación del autómata de Rabin determinista de la propiedad. Definimos los pares de aceptación sobre el SNP producto como $(H'_i, L'_i) = (S \times H_i, S \times L_i)$.

Para cada par de aceptación (H'_i, L'_i), llamaremos *CT aceptadas* a las CT $B^i_1, \ldots, B^i_n$ que son maximales en H'_i y contienen algún estado en L'_i.

Dado un SNP producto, denotaremos con $T = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} B^i_j$ al conjunto de todos los estados en CT aceptadas.

Veamos cómo se obtiene la probabilidad máxima de satisfacer ϕ, $Pr^+_{s_{in}}(\omega \vDash \phi) = Pr^{\eta^+}_{s_{in}}(\omega \vDash \phi)$ donde η^+ es una estrategia óptima que maximiza dicha probabilidad. La probabilidad mínima se puede obtener usando el mismo método sobre la negación de la fórmula y sustrayendo el resultado de 1, dado que $Pr^+_{s_{in}}(\omega \vDash \phi) = 1 - Pr^-_{s_{in}}(\omega \vDash \neg\phi)$, o con el método directo descrito en [CL06].

Como la probabilidad máxima de permanecer en una CT es de 1, el cálculo de la probabilidad máxima de satisfacer la fórmula se reduce a encontrar la probabilidad máxima de alcanzar cualquiera de las CT aceptadas.

Teorema 5.1.1. $Pr^+_{s_{in}}(\omega \vDash \phi) = sup_\eta Pr^\eta_{s'_{in}}(\exists k.\omega_k \in T)$

5.2. Cálculo de alcanzabilidad máxima

Para calcular la probabilidad máxima de alcanzar T desde s_{in} dividiremos el problema en partes. Primero buscaremos el conjunto de estados cuya probabilidad de alcanzar T es de 0 bajo toda estrategia al que llamaremos S^{no}. S^{no} puede obtenerse restándole a S' los estados que pueden alcanzar T con probabilidad no nula, usando el siguiente método iterativo que parte de T y en la i-ésima iteración agrega los estados a i transiciones de T:

$$\begin{array}{ll} B^0 & = T \\ B^{i+1} & = B^i \cup \{s \in S' \mid \exists a \in \kappa(s), t \in B^i . p(t|s,a) > 0\} \\ S^{no} & = S' - lim_{k \leftarrow \infty} B^k \end{array}$$

Una vez calculado S^{no}, nos resta averiguar la probabilidad máxima de alcanzar T desde cada estado s en $S^? = S' - (T \cup S^{no})$, a la que llamaremos x_s, resolviendo el siguiente problema de optimización lineal:

$$\begin{array}{ll} \text{Minimizar} & \sum_{t \in S^?} x_s \\ \text{Sujeto a} & x_s \geq \sum_{t \in S^?} s'.p(t|s,a) + \sum_{t \in T} p(t|s,a) \\ & \text{para todo } s \in S^?, a \in \kappa(s). \end{array}$$

5.3. Model checking simbólico

Un problema intrínseco a la verificación de modelos es la *explosión de estados*, el crecimiento exponencial de un modelo respecto al tamaño del sistema a representar. Además, la generación del SNP producto puede llegar a multiplicar la cantidad de estados por el tamaño del autómata de la propiedad a verificar, que a su vez también es exponencial respecto a la longitud de la fórmula. Esto limita la aplicabilidad del model checking de fórmulas LTL a sistemas medianos o pequeños y a fórmulas que se traducen a autómatas pequeños.

Para aliviar este problema hay una variedad de técnicas que se aplican en cada paso del proceso, y otras que directamente atacan la representación del sistema mismo. A continuación veremos una forma de realizar lo segundo utilizando BDD, pero que introduce la dificultad de deber operar sobre todo el conjunto simultáneamente (en lo que se llaman *métodos implícitos*) en vez de poder tratar estado por estado. Cuando aplicamos métodos implícitos al model checking, estamos realizando *model checking simbólico*.

5.3.1. Representando sistemas de transiciones con funciones booleanas

Veamos cómo un modelo se puede representar como una función booleana, empezando por los estados. Cada estado $s \in S$ puede ser visto como un vector de los valores de verdad de cada proposición atómica en el mismo, $\langle s[\![p_1]\!], \ldots, s[\![p_n]\!]\rangle$. Si dos o más estados tienen la misma etiqueta, agregaremos las proposiciones atómicas auxiliares que sean necesarias para poder distinguirlos.

A cada proposición atómica $p_1, \ldots, p_n$ le asignaremos una variable booleana $b_1, \ldots, b_n$. Luego cada estado posee una secuencia de valores booleanos única que lo representa, donde cada elemento de la secuencia es almacenado en una variable booleana. Podemos usar esta codificación para representar cualquier subconjunto A de S usando la *función característica* de A, que dado un elemento nos dice si está contenido o no en el conjunto y está definida como:

$$1_A(x) = \begin{cases} 1 & \text{si } x \in A \\ 0 & \text{si } x \notin A \end{cases}$$

La función característica devuelve un valor booleano, por lo que, con la codificación de estados descrita anteriormente como entrada, es una función booleana. En el caso del conjunto vacío, la función característica es la función constante 0 pues no existe elemento que esté contenido en el conjunto. Para un conjunto de un elemento, necesitamos que la función devuelva 1 sólo si se ingresa el elemento deseado, lo que se logra tomando la conjunción de todas las variables booleanas en las que codificamos los estados, negándolas si su valor es 0 para que la conjunción sea 1. De esta manera un elemento con codificación $\langle 0, 1, 0, 1\rangle$ pasa a ser la función booleana $f(x_1, x_2, x_3, x_4) = \bar{x_1} \cdot x_2 \cdot \bar{x_3} \cdot x_4$.

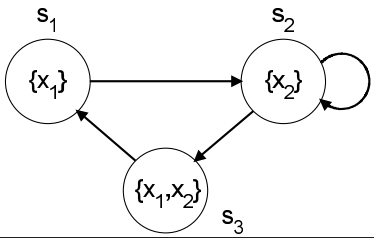
Notemos que para esta forma de almacenar conjuntos, la disyunción de funciones características resulta en la unión de los conjuntos representados, o su intersección en el caso de la conjunción. Luego el conjunto $\{\langle 0, 0, 0, 1\rangle, \langle 0, 1, 0, 1\rangle\}$ pasa a ser la función booleana $f(x_1, x_2, x_3, x_4) = (\bar{x_1} \cdot \bar{x_2} \cdot \bar{x_3} \cdot x_4) + (\bar{x_1} \cdot x_2 \cdot \bar{x_3} \cdot x_4)$.

Esta representación también almacena los valores de las proposiciones atómicas en cada estado. Supongamos que tenemos un conjunto con un estado s en

el que se da $p_1 \wedge \neg p_2$. Su función característica es $1_{\{s\}} = x_1 \cdot \bar{x_2}$. Para obtener $s[\![p_2]\!]$, calculamos el resultado de $1_{\{s\}} \cdot x_2 = x_1 \cdot \bar{x_2} \cdot x_2 = false$, que implica que la ocurrencia de x_2 en $1_{\{s\}}$ está negada, dándonos el valor deseado. En caso contrario obtenemos la misma función característica inicial, pues la conjunción es idempotente. En el caso general, dado un conjunto de estados S, la distributividad de la conjunción respecto a la disyunción implica que al calcular la conjunción de la función característica del conjunto con una variable booleana x (o su negación) obtenemos la función característica del conjunto de estados en S que satisfacen la proposición atómica p (o su negación) representada por la variable x.

Ahora veamos cómo representar transiciones, ignorando por el momento las probabilidades y acciones. Utilizaremos la función característica de la misma manera que para los estados, pero almacenando los pares ordenados que componen la relación de transición, representando el par (s, s') de la transición $s \rightarrow s'$ con el vector booleano $\langle \hat{s}, \hat{s}' \rangle$, donde $\langle \hat{s} \rangle$ y $\langle \hat{s}' \rangle$ son las codificaciones de las proposiciones atómicas que valen en s y s', respectivamente. En la figura 5.1 se puede ver cómo un sistema de transiciones sencillo se traduce a funciones booleanas.

Si a esta función característica la escribimos en una grilla donde en un eje tenemos los 2^n valores posibles de $\hat{s}$ y en el otro la misma cantidad de valores posibles de $\hat{s}'$, tenemos una matriz cuadrada con valores booleanos. Esta representación *explícita* produce generalmente la mejor performance a cambio de un uso elevado de memoria, incluso utilizando matrices ralas.



Estado	Codificación	Función booleana
s_1	$\langle 1, 0 \rangle$	$x_1 \cdot \bar{x_2}$
s_2	$\langle 0, 1 \rangle$	$\bar{x_1} \cdot x_2$
s_3	$\langle 1, 1 \rangle$	$x_1 \cdot x_2$

Transición	Codificación	Función booleana
$s_1 \rightarrow s_2$	$\langle 1, 0, 0, 1 \rangle$	$x_1 \cdot \bar{x_2} \cdot \bar{x'_1} \cdot x'_2$
$s_2 \rightarrow s_2$	$\langle 0, 1, 0, 1 \rangle$	$\bar{x_1} \cdot x_2 \cdot \bar{x'_1} \cdot x'_2$
$s_2 \rightarrow s_3$	$\langle 0, 1, 1, 1 \rangle$	$\bar{x_1} \cdot x_2 \cdot \bar{x'_1} \cdot x'_2$
$s_3 \rightarrow s_1$	$\langle 1, 1, 1, 0 \rangle$	$x_1 \cdot x_2 \cdot \bar{x'_1} \cdot x'_2$

Figura 5.1: Un modelo y la codificación de sus estados y transiciones

5.3.2. Representando un SNP con MTBDD

Para representar un modelo probabilístico con MTBDD, mantenemos la misma codificación de estados que mencionamos anteriormente, pero en vez de representar la función característica de la relación de transición usaremos su probabilidad. Luego, dados estados $s, s' \in S$ codificados como $\langle \hat{s} \rangle, \langle \hat{s}' \rangle$, el MTBDD

de transiciónes almacena la función:

$$f(\hat{s}, \hat{s}') = p(s' \mid s)$$

De este MTBDD se puede extraer un BDD que contiene los sucesores de cada estado y acción calculando $\texttt{threshold}(>, 0, B_f)$.

Finalmente, para representar un SNP necesitamos agregar acciones. Simplemente basta con codificar cada acción con una secuencia única de valores booleanos como hemos hecho anteriormente, y agregar esta secuencia a cada transición que se realiza a través de dicha acción. Si una acción $a \in \kappa(s)$ se codifica como $\langle a_1, \ldots, a_m \rangle$, entonces:

$$f(\hat{a}, \hat{s}, \hat{s}') = p(s' \mid s, a)$$

5.3.3. Construcción del SNP producto

Teniendo ahora funciones booleanas que representan el SNP, veamos cómo se construyen los BDDs que componen el SNP producto para la verificación de una propiedad. Supongamos que utilizamos las variables booleanas $s_1, \ldots, s_n$ para representar el conjunto S de estados en un BDD, y tenemos un autómata de Rabin determinista con conjunto de estados Q.

Creamos tantas variables booleanas $q_1, \ldots, q_m$ como nos sean convenientes para codificar cada estado del autómata con una secuencia única, donde $m \geq \lceil log_2 |Q| \rceil$. Por el sólo hecho de agregar estas variables booleanas y sin alterar de ninguna forma los BDD existentes, podemos ver el BDD del conjunto de estados S como $S \times Q$.[1] Esto se debe a que cuando una variable no ocurre en un BDD significa que la función representada es independiente de la misma, de modo que:

$$\forall \hat{q} : 1_S(\hat{s}) = 1_{S \times Q}(\hat{s}, \hat{q})$$

Esto también satisface la propiedad deseada $\forall s \in S, q \in Q : (s, q)[\![x]\!] = s[\![x]\!]$ porque los valores de las proposiciones atómicas en s siguen almacenados en las variables $\hat{s}$, que no son modificadas.

Además necesitamos crear variables booleanas $q'_1, \ldots, q'_m$ para los destinos de la relación de transición del ARD. Agregando estas variables, y junto a las variables agregadas anteriormente, podemos ver a la función de probabilidad de transición original de tipo $A \times S \times S \to [0, 1]$ como la función p' en $A \times (S \times Q) \times (S \times Q) \to [0, 1]$, donde:

$$\forall s \in S, q \in Q : \kappa'(s, q) = \kappa(s)$$
$$\forall q, q' \in Q : p'((s', q') \mid (s, q), a) = p(s' \mid s, a)$$

Lo que significa que el SNP original ha sido copiado 2^m veces, y se puede hacer una transición desde una copia a cualquier copia a través de la misma acción original con la probabilidad original. Esto difiere de la función de probabilidad deseada, donde las transiciones sólo ocurren si el estado de destino del SNP satisface la etiqueta de la transición en el ARD:

$$p((s', q') \mid (s, q), a) = \begin{cases} p(s' \mid s, a) & \text{si } \delta(q, L(s')) = q' \\ 0 & \text{caso contrario} \end{cases}$$

[1] En realidad esto también agrega $2^m - |Q|$ estados inexistentes en el ARD, que serán eliminados luego por ser inalcanzables.

Para solucionar este problema, construiremos un BDD que filtre las transiciones no deseadas. Esto se puede realizar obteniendo el producto del BDD de p' con el de una función que vale 1 para las transiciones que satisfacen la propiedad deseada y 0 en caso contrario. Como representamos al estado s' por sus proposiciones atómicas, la función es:

$$f(\hat{s}', \hat{q}, \hat{q}') = \begin{cases} 1 & \text{si } \delta(\langle\hat{q}\rangle, \langle\hat{s}'\rangle) = \langle\hat{q}'\rangle \\ 0 & \text{caso contrario} \end{cases}$$

Luego reexpresamos la guarda:

$$f(\hat{s}', \hat{q}, \hat{q}') = \begin{cases} 1 & \text{si } (\langle\hat{q}\rangle, \langle\hat{s}'\rangle, \langle\hat{q}'\rangle) \in \delta \\ 0 & \text{caso contrario} \end{cases}$$

Pero esto no es más que la función característica de δ con las etiquetas de las transiciones permutadas por las variables de s', que se puede calcular mediante $\texttt{permute}(1_\delta, \langle\hat{l}\rangle, \langle\hat{s}'\rangle)$ para algún conjunto de variables $\hat{l}$ con el que se guardan las etiquetas en el BDD de 1_δ.

Nos queda elegir el nuevo estado inicial. Por haber agregado las variables tenemos un BDD de $s_{in} \times Q$, y queremos elegir el estado $(s_{in}, \delta(q_{in}, L(s_{in})))$. Dado el BDD de δ, calculamos $1_\delta \cdot 1_{q_{in}}$ para obtener las transiciones que parten de q_{in}, y luego $1_\delta \cdot 1_{q_{in}} \cdot s_{in}$ para obtener la 3-upla $(q_{in}, s_{in}, q') \in \delta$ donde $q' = \delta(q_{in}, L(s_{in}))$ puesto que $L(s_{in})$ y s_{in} son iguales en esta representación. Finalmente calculamos la proyección sobre el segundo y tercer elemento de la 3-upla utilizando el operador de cuantificación $\texttt{exists}(\langle\hat{q}\rangle, B_{(q_{in}, s_{in}, q')})$ obteniendo la tupla (s_{in}, q'). Como q' está almacenado en las variables de destino de las transiciones, para convertir esta tupla en un estado legítimo de $S \times Q$ permutamos las variables $\hat{q}'$ con $\hat{q}$, con lo que conseguimos un BDD de $(s_{in}, \delta(q_{in}, L(s_{in})))$.

5.3.4. Eliminación de estados no alcanzables

La construcción del SNP producto genera algunos estados y transiciones que no son alcanzables desde el estado inicial. Los mismos pueden provocar problemas en pasos siguientes, además de agrandar los BDD y requerir mayor tiempo de procesamiento ya que los algoritmos implícitos no pueden distinguirlos de estados legítimos.

Para eliminarlos, computamos el conjunto de estados alcanzables desde s_{in}, denominado *forward set* de s_{in} o $Fwd(s_{in})$, usando el siguiente método iterativo que partiendo de un conjunto con el estado inicial agrega a todos sus sucesores y repite esto hasta no agregar más estados:

$$\begin{array}{ll} C_0 & = \{s_{in}\} \\ C_{i+1} & = img(C_i) \\ Fwd(s_{in}) & = \bigcup_{k=0}^{\infty} C_k \end{array}$$

$img(C) = \bigcup_{c \in C, a \in \kappa(c)} Suc(c, a)$ es el conjunto de todos los sucesores de los estados de C, al que llamaremos *imagen* de C. La imagen es calculada seleccionando las transiciones que parten de C a través de su intersección con el conjunto de sucesores (obtenida usando `threshold` como vimos anteriormente), abstrayéndonos de las variables de acción y estados de origen para obtener un conjunto de estados de destino, y finalmente permutando las variables de destino por las de origen para tener el mismo conjunto de variables de S':

$$
\begin{array}{ll}
trans & = \texttt{threshold}(>, 0, p') \\
trans_C & = \texttt{apply}(\cdot, trans, C) \\
dest_C & = \texttt{exists}(trans_C, \langle \hat{a}, \hat{s}, \hat{q} \rangle) \\
img(C) & = \texttt{permute}(dest_C, \langle \hat{s}, \hat{q} \rangle, \langle \hat{s}', \hat{q}' \rangle)
\end{array}
$$

Con este conjunto de estados alcanzables podemos limpiar nuestra relación de transición, eliminando todas las transiciones que parten de elementos fuera del conjunto, y luego eliminando las transiciones que llegan a elementos fuera del conjunto:

$$
\begin{array}{l}
p' = \texttt{apply}(\cdot, p', C) \\
p' = \texttt{apply}(\cdot, p', \texttt{permute}(C, \langle \hat{s}, \hat{q} \rangle, \langle \hat{s}', \hat{q}' \rangle))
\end{array}
$$

5.3.5. Búsqueda simbólica de CFC maximales

Antes de ver el algoritmo de búsqueda de CT maximales, veremos cómo se realiza una parte crucial del mismo: la búsqueda de las componentes fuertemente conexas maximales en un conjunto de estados. Este problema tiene una solución bien conocida y veloz en los métodos explícitos, el algoritmo de Tarjan[Tar72], que detecta las CFC a través del etiquetado de los nodos a medida que se realiza una búsqueda en profundidad (DFS) en el grafo. Este algoritmo es de complejidad lineal en la cantidad de lados del grafo.

Lamentablemente este algoritmo no es viable para BDD debido al alto costo de extraer estados y transiciones individuales de la estructura. Como la búsqueda de CFC es generalmente una pieza clave de los algoritmos de model checking, y debido a la popularidad del model checking simbólico para verificar sistemas con mayor cantidad de estados, se han diseñado una variedad de algoritmos simbólicos de búsqueda de CFC [EL86][BGS00][FFK+01][XB99][GPP03] que intentan aproximarse a la eficiencia del algoritmo de Tarjan. Explicaremos cómo funciona el algoritmo de Xie y Beerel[XB99], y cómo el algoritmo *lockstep*[BGS00] optimiza esta búsqueda.

Los algoritmos simbólicos operan buscando CFC dentro de conjuntos *CFC-cerrados*, que son conjuntos de estados donde cada CFC o bien está incluída en el conjunto o está fuera del conjunto. La unión e intersección de conjuntos CFC-cerrados también es CFC-cerrada, al igual que el complemento respecto del conjunto S de todos los estados del grafo.

Una forma de encontrar conjuntos CFC-cerrados es tomar un estado cualquiera $s \in S$ y calcular su *forward set*, o bien su *backward set* $Bwd(s)$, que es el conjunto de estados que contienen al nodo en su forward set.

Lema 5.3.1. *Sea $s \in S$ un estado. $Fwd(s)$ y $Bwd(s)$ son CFC-cerrados.*

Prueba. Supongamos que no lo son. Luego existe una CFC que posee elementos dentro y fuera de $Fwd(s)$ o $Bwd(s)$. Por la definición de CFC todos sus elementos son alcanzables desde cualquier otro elemento, entonces los elementos fuera de $Fwd(s)$ son alcanzables desde los elementos dentro del mismo y los elementos dentro de $Bwd(s)$ son alcanzables desde los de afuera, por lo que deben estar incluídos en $Fwd(s)$ y $Bwd(s)$ respectivamente. Absurdo.

La implementación de *Bwd* es similar a la de *Fwd*, pero en vez de calcular la imagen del conjunto de estados actual C_i en cada iteración, calcularemos su *preimagen* $pre(C_i)$ que es el conjunto de estados que llega en una transición a

C_i. Esto se realiza de manera similar a *img* pero seleccionando las transiciones por su destino en vez de por su origen:

$$\begin{aligned} trans &= \texttt{threshold}(>, 0, p') \\ dest_C &= \texttt{permute}(C, \langle \hat{s}, \hat{q} \rangle, \langle \hat{s}', \hat{q}' \rangle) \\ trans_C &= \texttt{apply}(\cdot, trans, dest_C) \\ pre(C) &= \texttt{exists}(trans_C, \langle \hat{a}, \hat{s}', \hat{q}' \rangle) \end{aligned}$$

Algoritmo de Xie-Beerel

Dado un estado y su respectivo forward set y backward set, el siguiente lema nos dice que su intersección es una CFC:

Lema 5.3.2. *Sea $s \in S$ un estado. $Fwd(s) \cap Bwd(s)$ es una CFC maximal en S.*

Prueba. Supongamos que $Fwd(s) \cap Bwd(s) \neq \emptyset$. Luego para todo par de estados $t, r \in Fwd(s) \cap Bwd(s)$ tenemos $t \rightsquigarrow s$ y $r \rightsquigarrow s$ por estar en $Bwd(s)$, y $s \rightsquigarrow t$ y $s \rightsquigarrow r$ por estar en $Fwd(s)$. Luego podemos construír caminos de t a r pasando por s y viceversa, por lo que $t \rightsquigarrow r$ y $r \rightsquigarrow t$ y por lo tanto $Fwd(s) \cap Bwd(s)$ es fuertemente conexo.

Para ver que es maximal, supongamos por el absurdo que existe una CFC $C \subseteq S$ tal que $Fwd(s) \cap Bwd(s) \subseteq C$. Sea $D = C - (Fwd(s) \cap Bwd(s))$ el conjunto de estados en C pero que no están en $Fwd(s) \cap Bwd(s)$. Como C es fuertemente conexo, entonces existen caminos desde todos los elementos en $Fwd(s) \cap Bwd(s)$ a los elementos de D, lo que nos dice que $D \subseteq Fwd(s)$, y también hay caminos desde todos los elementos de $Fwd(s) \cap Bwd(s)$ a los elementos de D, por lo que $D \subseteq Bwd(s)$. Luego $D = \emptyset$, absurdo.

De este resultado deriva el siguiente algoritmo recursivo, donde luego de encontrar una CFC se procede a buscar CFCs en el complemento del forward set o backward set obtenido, y en el resto del mismo una vez eliminada la CFC encontrada pues ambos conjuntos son CFC-cerrados:

Algorithm 1 BúsquedaXB ($C \subseteq S$)

$s := TomarElemento(C)$
$CFC := Fwd(s) \cap Bwd(s)$
return $\{CFC\} \cup BúsquedaXB(Bwd(s) - CFC) \cup BúsquedaXB(C - Bwd(s))$

Para poder usar este algoritmo necesitaremos implementar *TomarElemento*, que dado el BDD de un conjunto devuelve un BDD de un conjunto unitario que contiene un solo elemento del conjunto original. Para lograr esto, construiremos el elemento variable por variable. Para cada variable x de 1_C, evaluamos el valor de cualquiera de los cofactores de 1_C. Si el BDD del cofactor no es el de la función constante 0, entonces existe un elemento del conjunto cuyo valor de x es el de la restricción aplicada. Luego tomamos otra variable y repetimos el procedimiento sobre el cofactor. Notemos que si este procedimiento se realiza eligiendo las variables de acuerdo al orden del BDD, el algoritmo simplemente desciende por la estructura del BDD hasta un nodo terminal 1 y devuelve las elecciones tomadas en cada nodo.

Algorithm 2 TomarElemento ($C \subseteq S, \langle \hat{s} \rangle$)

```
elemento := B_1
for all var ∈ ⟨ŝ⟩ do
  if C_var ≠ B_0 then
    elemento := elemento · var
    C := C_var
  else
    elemento := elemento · var̄
    C := C_var̄
  end if
end for
return elemento
```

Como las operaciones de costo computacional significativo son el cálculo de la imagen o preimagen de un conjunto, la complejidad de los algoritmos de búsqueda de CFC generalmente se expresa en términos de la cantidad de imágenes y preimágenes computadas, también llamadas *pasos simbólicos*. Este algoritmo tiene complejidad $\mathcal{O}(|S|^2)$.

Algoritmo de lockstep

El algoritmo de lockstep mejora la cota del algoritmo anterior computando parcialmente el forward set o el backward set. Recibe este nombre porque en cada iteración se realiza un solo paso en el cálculo del forward set y del backward set, hasta que uno de estos dos conjuntos converge. Una vez que esto sucede, se sigue calculando el conjunto que todavía no ha convergido sólo hasta que su intersección con el conjunto convergido deja de crecer. Esta intersección que es la CFC que contiene a v. Obtenida la CFC, la recursión se hace particionando el espacio de estados de la misma manera que el algoritmo anterior, pero usando el conjunto que convergió en vez de elegir uno arbitrariamente.

Veamos que el criterio de terminación es suficiente. Supongamos que $Bwd(s)$ ha convergido, que hemos llegado a un paso del cómputo de $Fwd(s)$ en el que se encontró un conjunto de estados nuevos A de intersección vacía con $Bwd(s)$ con lo que el algoritmo termina la búsqueda de la CFC, y supongamos por el absurdo que existe algún paso siguiente en el cómputo de $Fwd(s)$ que agrega un conjunto de estados B a $Fwd(s) \cap Bwd(s)$. Luego los estados de B son alcanzables desde algún conjunto de estados $A' \subseteq A$, pero entonces $A' \subseteq Bwd(s)$ y por lo tanto $A \cap Bwd(s) \neq \emptyset$. Absurdo.

Esta optimización resulta en una complejidad de $\mathcal{O}\left(|S| log \frac{dN}{|S|}\right)$, donde d es la distancia máxima entre dos estados y N es la cantidad de CFC maximales en S.

Algorithm 3 Lockstep ($C \subseteq S$)

$s := TomarElemento(C)$
$F, dF, B, dB := \{s\}, \{s\}, \{s\}, \{s\}$
while $dF \neq \emptyset \wedge dB \neq \emptyset$ **do**
  $dF := img(F) \cap F$
  $dB := pre(B) \cap B$
  $F := F \cup dF$
  $B := B \cup dB$
end while
if $dF = \emptyset$ **then**
  $Convergido := F$
  while $F \cap dB \neq \emptyset$ **do**
    $dB := pre(B) \cap B$
    $B := B \cup dB$
  end while
else
  $Convergido := B$
  while $B \cap dF \neq \emptyset$ **do**
    $dF := img(F) \cap F$
    $F := F \cup dF$
  end while
end if
$CFC := F \cap B$
return $\{CFC\} \cup Lockstep(Convergido - CFC) \cup Lockstep(C - Convergido)$

5.3.6. Búsqueda simbólica de conjuntos estables maximales

La otra parte del algoritmo de búsqueda de CT maximales es la búsqueda de conjuntos estables maximales. Repasando el algoritmo de la sección 2.3 vemos que necesitamos implementar una función que calcule las iteraciones $E_C^{i+1} = \{s \in E_C^i \mid \exists a \in \kappa(s).Suc(s,a) \subseteq E_C^i\}$, los conjuntos de estados para los que existe una acción que sólo lleva a estados en E_C^i, que convergen al conjunto estable maximal en C.

Comenzaremos seleccionando el conjunto de transiciones que parten de E_C^i y llegan a E_C^i:

$$\begin{aligned} DesdeE_C^i &= \texttt{apply}(\cdot, p', E_C^i) \\ DesdeYHaciaE_C^i &= \texttt{apply}(\cdot, DesdeE_C^i, \texttt{permute}(E_C^i, \langle \hat{s}, \hat{q} \rangle, \langle \hat{s}', \hat{q}' \rangle)) \end{aligned}$$

Luego para cada estado de E_C^i calculamos la suma de las probabilidades de las transiciones que permanecen en E_C^i a través de cada acción.

$$ProbPermanecerEnE_C^i = \texttt{sumAbstract}(DesdeYHaciaE_C^i, \langle \hat{s}', \hat{q}' \rangle)$$

Finalmente seleccionamos los estados y acciones para los que esta suma de probabilidades es igual a 1, o sea que todos sus sucesores están contenidos en E_C^i, y nos quedamos con los estados que poseen al menos una acción que cumple con este requisito.

$$\begin{array}{ll} SiemprePermaneceEnE_C^i & = \texttt{threshold}(=, 1, ProbPermanecerEnE_C^i) \\ E_C^{i+1} & = \texttt{exists}(SiemprePermaneceEnE_C^i, \langle \hat{a} \rangle) \end{array}$$

5.3.7. Cálculo de alcanzabilidad

La implementación de un algoritmo de programación lineal para el cálculo de la probabilidad de alcanzar las CTs aceptadas es directa usando operaciones de matrices sobre MTBDD. Sin embargo estos algoritmos requieren matrices más grandes que la de la función de probabilidad de transición para operar, lo que va en contra de nuestra meta de verificar sistemas de gran cantidad de estados, y las matrices utilizadas son irregulares con lo que la eficiencia de la representación de los MTBDD se pierde.

Por esta razón existe un método alternativo para calcular aproximaciones de las probabilidades, mostrado en [Bai98], que se resuelve con métodos numéricos iterativos:

$Pr_s^+ = lim_{n\to\infty} Pr_{s(n)}^+$ donde:

$$Pr_{s(n)}^+ = \begin{cases} 0 & \text{si } s \in S^{no} \\ 0 & \text{si } s \in S^? \wedge n = 0 \\ max_{a\in\kappa(s)} \left\{ \sum_{t\in S} p(t|s,a).Pr_{s(n-1)}^+ \right\} & \text{si } s \in S^? \wedge n > 0 \end{cases}$$

Este método solo requiere memoria adicional para el vector resultado. El vector se hace más irregular a medida que converge por lo que su almacenamiento en MTBDD resulta lento e ineficiente, y el desempeño varía mucho según el modelo. Sin embargo el menor tamaño del vector motiva la creación de métodos híbridos donde el vector es almacenado convencionalmente en una matriz y el resto del sistema se almacena en MTBDD. Los métodos híbridos [Par02] están situados generalmente a mitad de camino entre los métodos puramente simbólicos y los métodos explícitos en cuanto a consumo de recursos y tiempo de procesamiento.

Capítulo 6

PRISM

PRISM [HKNP06], del inglés *Probabilistic Symbolic Model Checker*, es una herramienta de simulación y model checking simbólico de propiedades PCTL (Probabilistic Computation Tree Logic) para cadenas de Markov de tiempo discreto y procesos de decisión de Markov, y de propiedades CSL (Continuous Stochastic Logic) para cadenas de Markov de tiempo contínuo, sobre la que se ha verificado una gran variedad de casos de estudio y permanece en constante desarrollo.

La herramienta, que es software libre, está escrita en su mayoría en Java, salvo las rutinas de mayor costo computacional que están escritas en C y son llamadas a través de la Java Native Interface para mejor desempeño. Esto incluye a CUDD [Som99], el paquete de MTBDD utilizado.

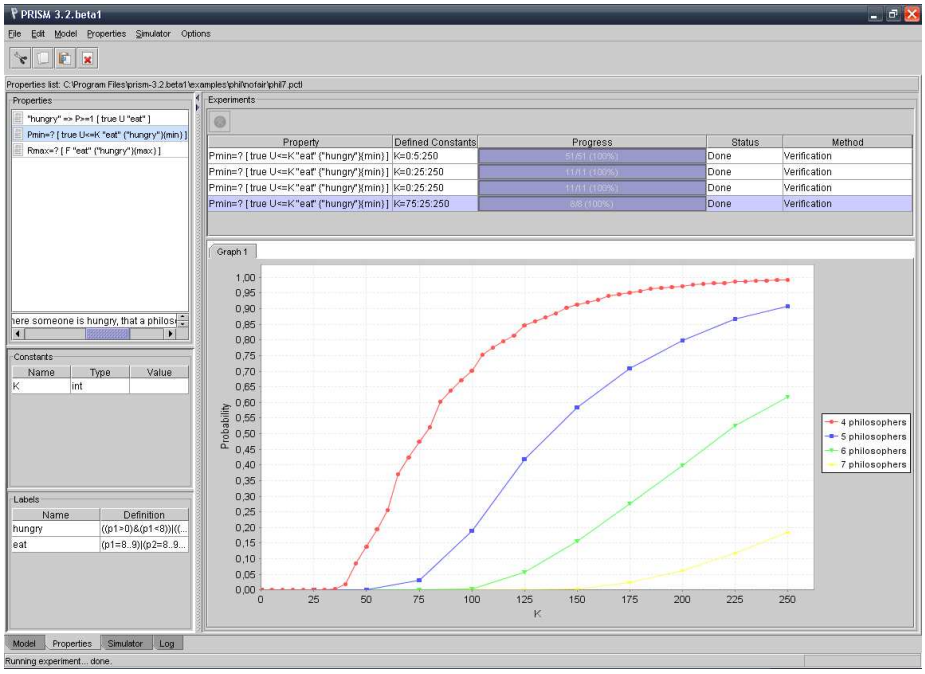


Figura 6.1: PRISM en acción.

6.1. Arquitectura de PRISM

Para verificar un sistema, uno provee su especificación escrita en el lenguaje de PRISM y una lista de propiedades a verificar. La especificación es parseada y se crea un modelo del tipo que corresponda. Luego se parsea la lista de propiedades creando una lista de objetos de tipo PCTLFormula (que incluye fórmulas CSL), y se las procesa una por una. Para cada propiedad se verifica que la lógica en la que está escrita sea la adecuada para el tipo de modelo en cuestión y que sea aplicable al modelo. Una vez que se verificó que la propiedad se adecua al modelo se invoca al model checker propiamente dicho de ese tipo de modelo, que dado un modelo y una propiedad calcula la probabilidad de satisfacción. Esto se repite hasta acabar con la lista de propiedades, y los resultados son mostrados en la consola de PRISM o en su interfaz gráfica, vista en la figura 6.1.

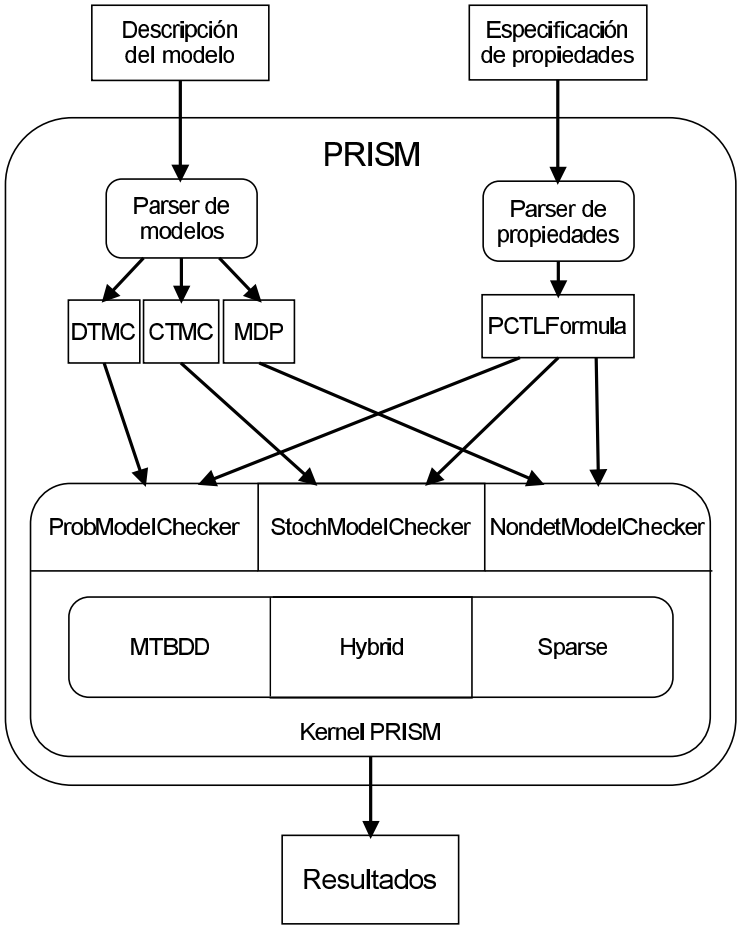


Figura 6.2: Arquitectura de PRISM.

PRISM posee tres motores distintos para el cálculo numérico de probabilidades: uno puramente simbólico que sólo utiliza MTBDD, uno híbrido que usa una matriz rala para el vector resultado, y uno que usa solamente matrices ralas. Como el resto de PRISM solo trabaja con MTBDD, el BDD de transiciones se

vuelca a una matriz realizando una búsqueda DFS en la estructura del BDD previo al cálculo de probabilidades. Estos tres motores le confieren a PRISM una gran versatilidad para la verificación de modelos de todo tamaño. La figura 6.2 muestra la arquitectura general de la herramienta.

6.2. El lenguaje PRISM

Los modelos de PRISM se especifican como una composición de *módulos*, componentes del sistema que corren bajo alguna forma de concurrencia. El sistema y cada módulo en particular poseen un conjunto de *constantes* y *variables* cuyos rangos de valores son especificados por el usuario:

```
const int N = 5;
x : [0..N] init 0;
```

Cuando las variables no pertenecen a ningún módulo las llamaremos *variables globales*. Cada combinación distinta de los valores de todas las variables representa un estado del sistema.

Aplicando distintos operadores aritméticos y de comparación a variables y valores constantes se obtienen *expresiones*, que son funciones de las variables del sistema con imagen en los números reales, o en los booleanos cuando se realiza una comparación entre subexpresiones:

```
formula incr = x < N ? x + 1 : x;
```

Para referirnos a un conjunto de estados utilizaremos su función característica escrita como expresión booleana.

Cada variable puede ser inicializada con un valor estableciendo un estado inicial único que consiste de los valores iniciales de todas las variables, o bien caracterizando los estados iniciales con una expresión.

Las transiciones dentro de cada módulo se especifican con *comandos*, que consisten de una *acción* utilizada para sincronización que puede ser vacía, una *guarda* que es una expresión booleana que establece los estados a los que se puede aplicar el comando, y un conjunto de *actualizaciones*, que son asignaciones a variables globales o pertenecientes al módulo de nuevos valores junto con probabilidad de elegir cada actualización. Se requiere que la variable a actualizar ocurra primada en la actualización y que la suma de las probabilidades de las actualizaciones de un comando sea igual a 1.

Para especificar elecciones no determinísticas se utilizan comandos cuyas guardas describen conjuntos superpuestos. Cuando el sistema alcanza un estado en la interseccion de dos o más guardas, el comando a ejecutar es elegido de forma no determinística y luego se elige la actualización de forma probabilista.

Ejemplo 6.2.1. *El siguiente modelo representa un balde que se va llenando progresivamente y puede ser vaciado completamente siempre que contenga agua. La elección de agregar agua o vaciarlo es no determinista para nivel* $\in [1..CAPACIDAD - 1]$.

```
mdp

const int CAPACIDAD = 4;

module balde
  nivel: [0..CAPACIDAD] init 0;

  [] nivel < CAPACIDAD -> 1: nivel' = nivel+1;
  [] nivel > 0         -> 1: nivel' = 0;
endmodule;
```

La composición de módulos es tomada del lenguaje de especificación CSP. El operador de composición más común es ||, y es el usado por defecto para componer módulos. Este operador permite todos los *interleavings* posibles de los comandos seleccionables de cada módulo, salvo cuando dos o más módulos poseen comandos con la misma acción, en cuyo caso dichos comandos deben ser ejecutados simultáneamente en todos los módulos que poseen la acción. El operador ||| elimina esta restricción. Además se incluyen operadores de renombre, ocultación y sincronización sobre conjuntos de acciones definidos por el usuario.

Ejemplo 6.2.2. *Veamos un modelo que hace uso intensivo de la sincronización de acciones entre módulos. Este modelo describe un sistema en el que un bombero hace repetidos viajes entre un tanque y un incendio trasladando agua en un balde, que a veces es volcada parcialmente en el camino por accidente. Estas 4 entidades son modeladas individualmente en módulos separados y deben sincronizarse cada vez que una acción involucra el estado de dos o más de ellas, como por ejemplo al llenar el balde con agua del tanque.*

```
mdp

const int CAPTANQUE = 100;
const int CAPBALDE  = 2;
const int LLAMAS = 30;
const int MAXVIAJES = CAPTANQUE;

module balde
  b : [0..CAPBALDE] init 0;

  [llenarbalde] b < CAPBALDE -> 1:   (b'=b+func(min,CAPBALDE-b,t));
  [correr]      b > 0        -> 0.5: (b'=b)
                             + 0.5: (b'=b-1);
  [apagarfuego] true         -> 1:   (b'=b-func(min,1,b));
endmodule

module tanque
  t : [0..CAPTANQUE] init CAPTANQUE;

  [llenarbalde] t > 0 -> 1: (t'=t-func(min,t,CAPBALDE-b));
endmodule
```

```
module incendio
  l : [0..LLAMAS] init LLAMAS;

  [apagarfuego] l > 0 -> 1: (l'=l-func(min,l,b));
endmodule

module bombero
  estado : [0..2] init 0;
  viajes : [0..MAXVIAJES] init 0;

  [llenarbalde] estado=0                    -> 1: (estado'=1);
  [correr]      estado=1 & viajes<MAXVIAJES -> 1: (estado'=2) & (viajes'=viajes+1);
  [apagarfuego] estado=2                    -> 1: (estado'=0);
endmodule
```

También se le pueden asignar recompensas a los distintos estados o transiciones. Esto se realiza con bloques delimitados por las directivas `rewards` y `endrewards`. Para asignarle una recompensa a un conjunto de estados, usamos una expresión booleana para identificar los estados y una expresión para el valor de la recompensa, separados por ":". Para asignarle recompensas a transiciones, identificamos la transición con su acción y guarda y le asignamos una expresión para el valor. Cuando un estado o una transición está incluida en más de una de estas asignaciones, se le asigna la suma de todas las recompesas aplicables.

Ejemplo 6.2.3. *Siguiendo con el ejemplo anterior, podríamos llevar rastro de la cantidad de viajes sin llevar la cuenta dentro del modelo del bombero sino con una estructura de recompensas aparte que aumenta cada vez que el bombero termina de correr, llegando a* `estado=2` *que no puede repetirse hasta que el bombero volvió a llenar el balde:*

```
rewards
estado=2 : 1;
endrewards
```

O bien podemos asignarle la recompensa a la acción de correr directamente:

```
rewards
[correr] true : 1;
endrewards
```

6.3. Modelos de PRISM

La representación interna de un modelo de PRISM es muy similar a la explicada en el capítulo anterior. Por cada variable declarada en el modelo con rango $[m..M]$, se almacena el rango para poder convertir las expresiones a BDD y se crean $2 \cdot \lceil log_2(M-m) \rceil$ variables booleanas en el paquete de BDD con las que se codifican los valores $0 \ldots M-m$ para las filas y columnas de la matriz de transiciones. Además según su tipo se agrupan referencias a las mismas en vectores para rápido acceso a la hora de operar con BDD, `allDDRowVars` en el caso de las variables de fila y `allDDColVars` para las de columna. También

se crean variables booleanas por cada comando de cada módulo para resolver el nodeterminismo local, referenciadas en `allDDChoiceVars`, variables por cada módulo para resolver el nodeterminismo en el interleaving de los mismos, referenciadas en `allDDSchedVars`, y variables por cada acción que debe ejecutarse en sincronía, referenciadas en `allDDSynchVars`. Estos últimos tres conjuntos de variables que contienen todas las opciones de nodeterminismo para una transición son agrupados en `allDDNondetVars`.

Cada comando es convertido a una matriz de probabilidad de transiciones. La acción del comando y el módulo al que pertenece definen las filas seleccionadas en `allDDNondetVars`. La guarda se traduce a un BDD y con éste se seleccionan los estados origen en `allDDRowVars`. Luego para cada actualización se seleccionan las columnas en `allDDColVars`, y a estas celdas seleccionadas de la matriz se les asigna la probabilidad de la actualización. El BDD de transiciones `trans` es la suma de las matrices de transiciones de cada comando.

El BDD de estados iniciales `start` es creado a partir de los valores iniciales de cada variable o traduciendo la expresión de estados iniciales a un BDD. El BDD del conjunto de estados alcanzables `reach` es calculado como vimos anteriormente computando el forward set de los estados iniciales.

Finalmente se incluyen BDD para cada declaración de recompensas agrupados según su tipo.

Capítulo 7

Implementación

Para agregar soporte de la lógica LTL a modelos no deterministas en PRISM, necesitamos hacer que reconozca las fórmulas en esta lógica como propiedades y proveer un model checker para las mismas, resultando en la arquitectura de la figura 7.1.

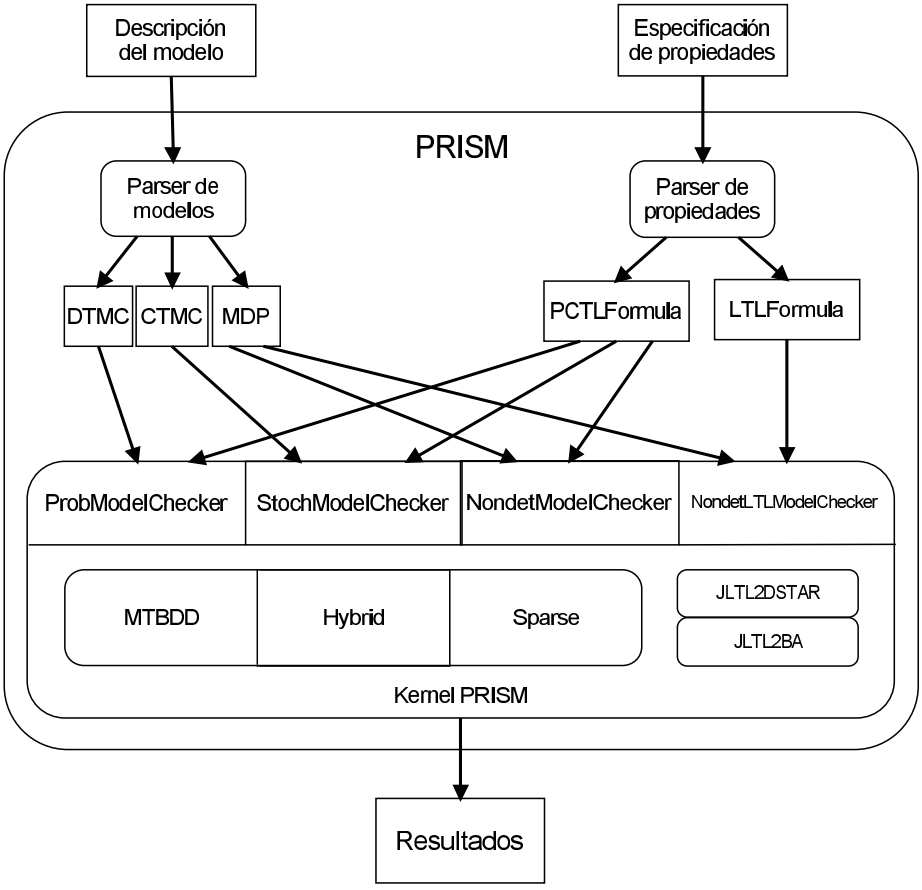


Figura 7.1: Cambios realizados a la arquitectura de PRISM.

Para conseguir lo primero definimos una superclase `Formula` de `PCTLFormula`, que provee métodos comunes para poder introducir `LTLFormula` sin mayores cambios al resto del programa. `LTLFormula` almacena fórmulas LTL donde se permite definir opcionalmente si se quiere calcular la probabilidad máxima o mínima y las proposiciones atómicas son reemplazadas por expresiones booleanas de PRISM. Al igual que las proposiciones atómicas, las expresiones booleanas deben valer o no en cada estado y este valor se puede calcular a priori, por lo que la modificación no afecta los algoritmos vistos. Los objetos de tipo `LTLFormula` son generados por el parser de PRISM, modificado para que reconozca la sintaxis de LTL además de PCTL y CSL de la siguiente forma:

```
Formula     = PCTLFormula
            | LTLProb
LTLProb     = ('Pmin' | 'Pmax')? LTLFormula
LTLFormula  = LTLAndOr ('=>' LTLAndOr)?
LTLAndOr    = LTLUntil ('|' LTLUntil | '&' LTLUntil)?
LTLUntil    = LTLUnary ('U' LTLUnary)?
LTLUnary    = '!' LTLUnary
            | 'X' LTLUnary
            | 'F' LTLUnary
            | 'G' LTLUnary
            | LTLLabel
LTLLabel    = '"' Identifier '"'
            | '(' LTLFormula ')'
            | Expression
```

Una vez creado el modelo y elegida la propiedad a verificar, se despacha la fórmula al model checker adecuado. En el caso de las fórmulas LTL este es nuestro propio model checker, `NondetLTLModelChecker`, que implementa la misma interfaz del resto de los model checkers para fórmulas de tipo `LTLFormula` y modelos no deterministas, cuyo flujo de ejecución se ve en la figura 7.2. El mismo genera un nuevo modelo no determinista del SNP producto en el que se calcula la probabilidad de alcanzar las componentes terminales maximales que son aceptadas.

La propiedad recibida debe ser convertida a un autómata de Rabin determinista para poder generar el SNP producto. Para evitar introducir modificaciones innecesarias a las herramientas de traducción, que operan sobre fórmulas LTL sin modificaciones, eliminamos el operador de probabilidad y el resto de la fórmula es negada en el caso del mínimo. También debemos reemplazar las expresiones booleanas que ocurren en la fórmula por proposiciones atómicas y llevar rastro de estos reemplazos. Como queremos obtener un autómata lo más chico posible, intentaremos utilizar la menor cantidad de proposiciones atómicas posibles para facilitar la aplicación de técnicas de reducción de la cantidad de estados del autómata resultante como las reglas de reescritura. Esto se logra fácilmente gracias a la prueba de equivalencia de orden constante de los BDD: si cada expresión se traduce a un BDD de todos los estados que la satisfacen y calculamos el BDD de la intersección de estos estados con los estados alcanzables del modelo, todas las expresiones equivalentes en este modelo tendrán el mismo BDD de intersección puesto que la representación es canónica. Además, si la intersección es vacía entonces la propiedad es falsa para este modelo, y de la misma forma si

la intersección es igual al conjunto de estados alcanzables entonces la propiedad equivale a *true*. Por lo tanto cada expresión encontrada en la fórmula es traducida a un BDD e intersecada con los estados alcanzables para obtener un BDD B, que es comparado con el BDD de la función constante 0, los estados alcanzables, y los BDDs de todas las expresiones traducidas anteriormente. De ser distinta a todas las anteriores, se crea una proposición atómica nueva p y el par (p, B) es almacenado en un diccionario para rápida búsqueda. En caso contrario la expresión es reemplazada por *false*, *true* o la proposición atómica asociada a B en el diccionario, respectivamente. Obtenida una fórmula LTL propiamente dicha, se la traduce a un autómata de Rabin determinista con un puerto a Java de la herramienta `ltl2dstar` [Kle05], que a su vez utiliza un puerto a Java de la herramienta `LTL2BA` [GO01] para la conversión a un autómata de Büchi.

Una vez generado el autómata de Rabin determinista de la fórmula, procedemos a crear variables booleanas en el paquete de BDDs para codificar los estados del mismo en la forma vista en el capítulo 4. Como queremos que el model checking de fórmulas LTL no afecte la operación normal de PRISM no podemos alterar el orden de estas variables en el BDD, situadas al final del mismo. Las referencias a dichas variables son agregadas a copias locales de `allDDRowVars` y `allDDColVars` según su tipo.

Con el modelo, el autómata de la fórmula y todas las variables necesarias creadas, estamos en condiciones de generar el SNP producto siguiendo el proceso descrito en el capítulo 5. La única consideración que debemos hacer es que las proposiciones atómicas que etiquetan las transiciones del autómata de Rabin deben ser convertidas a los estados que representan en el modelo a través de la intersección de los BDD de cada proposición en la etiqueta o sus complementos en caso de ocurrir negados, información que está disponible en el diccionario creado anteriormente. Los nuevos BDDs de transición, estados iniciales y estados alcanzables junto con los nuevos conjuntos de variables son utilizados para crear un nuevo modelo no determinista.

Dentro del nuevo modelo, realizamos la búsqueda de componentes terminales usando lockstep para la descomposición en CFC, y utilizando el motor de PRISM seleccionado por el usuario calculamos la probabilidad máxima de alcanzarlas desde los estados iniciales, que son restados a 1 de haberse pedido el cálculo de probabilidad mínima. El resultado es devuelto y el modelo del SNP producto es borrado, finalizando la verificación de la propiedad.

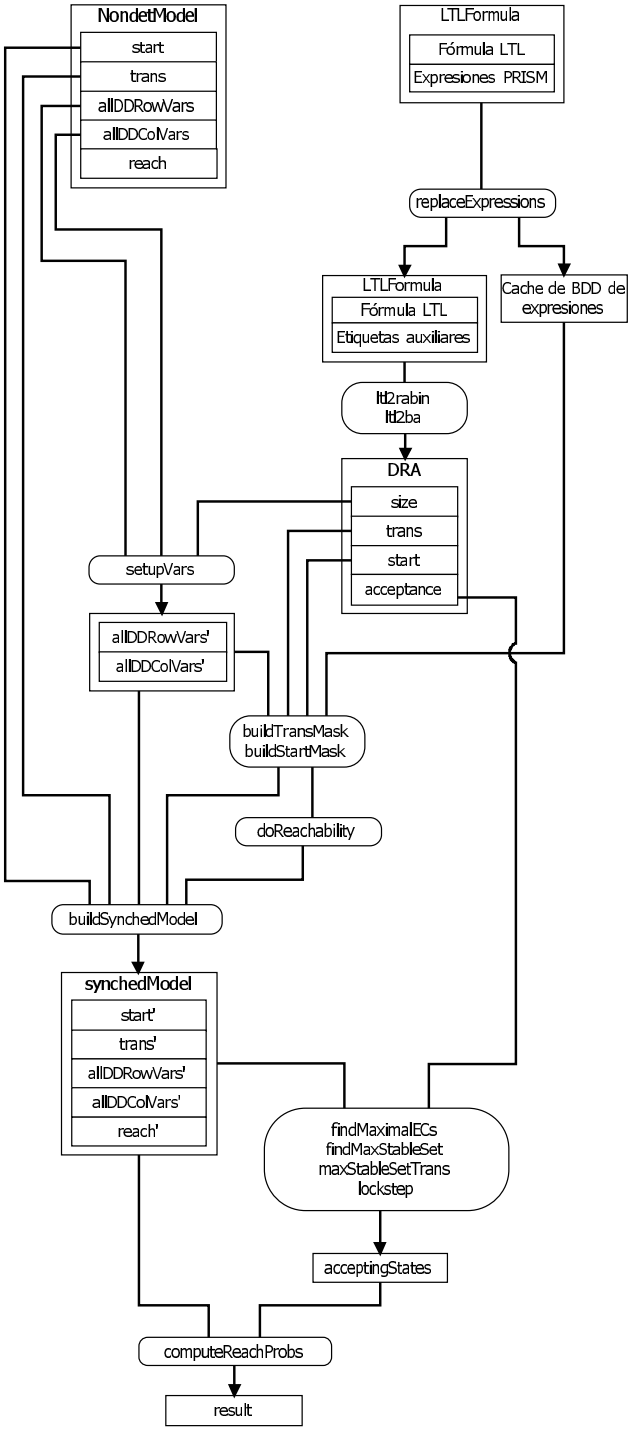


Figura 7.2: El módulo `NondetLTLModelChecker`.

Capítulo 8

Casos de estudio

Los siguientes casos de estudio fueron realizados sobre Ubuntu Linux 7.10 con un procesador Athlon XP de 2,2GHz y 1GiB de RAM.

8.1. Autoestabilización

Los algoritmos de autoestabilización para redes de procesos sirven para regularizar automáticamente situaciones ilegales en la red en una cantidad finita de pasos. Estudiaremos el modelo presentado en [IJ90], donde una cantidad de hosts están conectados en una topología de anillo por la que se transfieren *tokens* entre los hosts. El objetivo del algoritmo de autoestabilización en este caso consiste en dejar un solo token en circulación, partiendo de una situación inicial en la que existe una cantidad arbitraria de tokens en la red.

El algoritmo consigue su objetivo haciendo que cada host que posee un token descarte todos los tokens entrantes, y circula su token con igual probabilidad hacia cualquiera de sus dos pares. El modelo PRISM de una red de 3 hosts está dado por:

```
mdp

global int token1: [0..1];
global int token2: [0..1];
global int token3: [0..1];

module host1
    [] token1=1 -> 0.5: (token1'=0) & (token2'=1)
                 + 0.5: (token1'=0) & (token3'=1);
endmodule

module host2=host1[token1=token2,token2=token3,token3=token1] endmodule
module host3=host1[token1=token3,token2=token1,token3=token2] endmodule

formula ntokens = token1 + token2 + token3;

init
    ntokens > 1
endinit
```

Previo a este trabajo era posible probar la convergencia del modelo verificando la propiedad $\mathsf{F}\ (ntokens = 1)$ expresada en la lógica PCTL en PRISM, pero no era posible probar que en esta solución el token pasa infinitamente a menudo por todos los hosts de la red. Utilizando la lógica LTL, esta propiedad es $\mathsf{GF}\ (token_i = 1)$, y su verificación dió los siguientes resultados:

Hosts	Modelo original			Producto		
	Estados	Transiciones	Nodos	Estados	Transiciones	Nodos
5	5	10	68	9	18	92
10	1023	8960	369	1533	13109	753
15	32767	430080	729	49149	634864	1508
20	1048575	18350080	1189	1572861	27197419	2488
25	33554431	734003200	1749	50331645	1090519014	3693
30	1073741823	28185722880	2409	1610612733	41943039969	5123

Hosts	Tiempo en segundos					Prob.
	Rabin	Producto	CT	Alcanzabilidad	Total	mínima
5	0.066	0.002	0.002	0.001	0.078	1
10	0.062	0.006	0.011	0.001	0.081	1
15	0.062	0.012	0.078	0.001	0.154	1
20	0.061	0.020	0.537	0.002	0.621	1
25	0.066	0.030	4.321	0.002	4.419	1
30	0.067	0.047	55.689	0.001	55.805	1

8.2. Dining philosophers

Un problema común de concurrencia es el de los *dining philosophers* de Dijkstra. N filósofos se encuentran sentados en una mesa pensando. En la mesa hay un tenedor entre cada par de filósofos y un plato de spaghettis para cada uno. Cuando un filósofo tiene hambre toma los dos tenedores próximos en algún orden y cuando obtiene ambos procede a comer de su plato spaghettis hasta saciarse, para luego seguir pensando. La escasez de tenedores y la necesidad de obtener ambos para comer[1] convierte a esta mesa en un problema de asignación de recursos compartidos donde existe riesgo de *deadlock* cuando todos los filósofos toman un tenedor y por lo tanto no pueden tomar el otro, y de *starvation* cuando un filósofo con menor prioridad en la estrategia intenta tomar los cubiertos en un mal momento repetidamente o directamente no recibe un turno para actuar. Como el algoritmo de model checking implementado busca la probabilidad máxima sin importarle la distribución equitativa de las elecciones no deterministas, las situaciones de starvation de componentes individuales son triviales de conseguir y por lo tanto analizaremos el progreso de todo el conjunto de filósofos. El modelo del filósofo puede verse en la figura 8.1.

Analicemos la probabilidad mínima de que no se de una situación en la que ningún filósofo pueda comer, dada por la propiedad $\neg\mathsf{FG}(\forall i : \neg comiendo_i)$, equivalente a $\mathsf{GF}(\exists i : comiendo_i)$. PRISM nos advierte que el modelo contiene estados de deadlock de los que no parten transiciones, por lo que se le agregan bucles de estos estados a sí mismos utilizando el parámetro `-fixdl`. La verificación de la propiedad nos devuelve los siguientes resultados:

[1] Las versiones más modernas y verosímiles del problema utilizan palillos chinos y distintos platos.

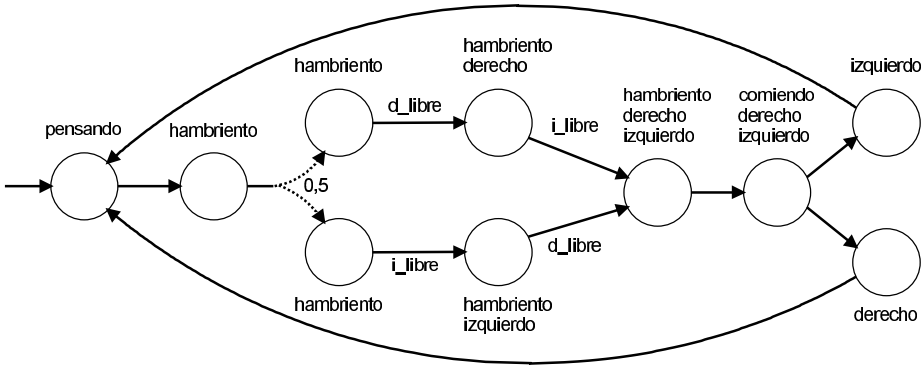


Figura 8.1: SNP de un filósofo.

Filósofos	Modelo original			Producto		
	Estados	Transiciones	Nodos	Estados	Transiciones	Nodos
3	1655	5144	637	2196	6605	1380
4	19591	81166	1179	27152	108858	3023
5	231791	1200367	1832	327082	1640492	5278
6	2742319	17041820	2594	3884156	23412038	7989
7	32444255	235224495	3465	45743710	322190402	11110
8	383846431	3180491130	4041	535919264	4320818130	14622
9	4541268671	$4,23 \times 10^{10}$	5534	6257847438	$5,68 \times 10^{10}$	18519
10	$5,37 \times 10^{10}$	$5,56 \times 10^{11}$	6732	$7,29 \times 10^{10}$	$7,37 \times 10^{11}$	22801
11	$6,35 \times 10^{11}$	$7,24 \times 10^{12}$	8039	$8,48 \times 10^{11}$	$9,45 \times 10^{12}$	27468
12	$7,52 \times 10^{12}$	$9,34 \times 10^{13}$	9455	$9,87 \times 10^{12}$	$1,20 \times 10^{14}$	32520
13	$8,89 \times 10^{13}$	$1,19 \times 10^{15}$	10246	$1,14 \times 10^{14}$	$1,51 \times 10^{15}$	37957
14	$1,05 \times 10^{15}$	$1,52 \times 10^{16}$	12614	$1,33 \times 10^{15}$	$1,90 \times 10^{16}$	43779

Filósofos	Tiempo en segundos					Prob.
	Rabin	Producto	CT	Alcanzabilidad	Total	mínima
3	0.062	0.031	0.027	0.069	0.191	0
4	0.059	0.110	0.113	0.341	0.624	0
5	0.061	0.264	0.262	2.161	2.751	0
6	0.062	0.584	0.579	7.154	8.385	0
7	0.067	1.096	1.301	20.967	23.432	0
8	0.066	1.604	2.369	46.054	50.095	0
9	0.068	2.819	3.784	93.935	100.608	0
10	0.067	3.701	6.070	188.747	198.586	0
11	0.068	4.929	9.374	325.216	339.589	0
12	0.175	6.730	13.419	514.240	534.566	0
13	0.163	9.534	19.486	1088.579	1117.763	0
14	0.214	12.382	26.014	3364.620	3403.233	0

La solucion de Duot, Fribourg y Claudine[DFC02] a este problema, basada en una solución original de Lehmann y Rabin[RL94], consiste en dejar el cubierto obtenido si no se puede obtener el segundo para evitar deadlocks. Esta modificación generalmente introduce un nuevo problema, el *livelock*, en el que todos los filósofos toman un cubierto cada uno al mismo tiempo y, al no poder obtener el segundo, también los liberan simultáneamente con lo que ninguno consigue comer. Sin embargo nuestra elección del primer cubierto a tomar

es probabilista, lo que previene una repetición infinita de malas elecciones. El modelo modificado se muestra en la figura 8.2.

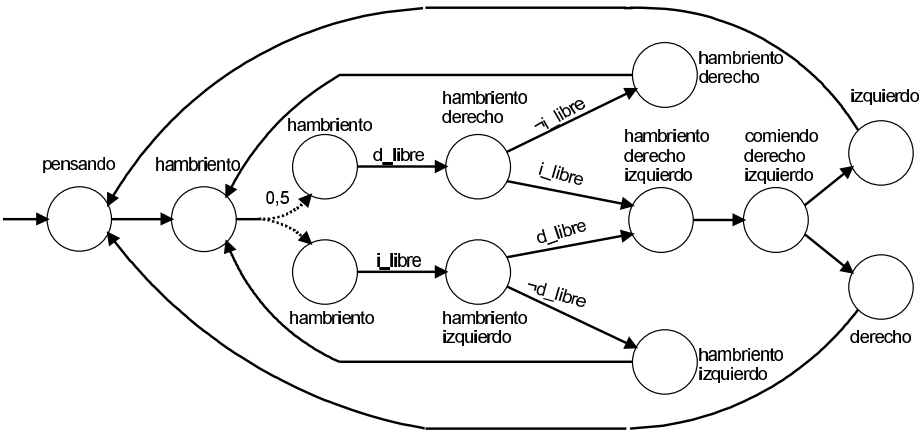


Figura 8.2: SNP de un filósofo libre de deadlock.

Verificando la propiedad anterior para este nuevo modelo, en el que no hay estados de deadlock como en el caso anterior, obtenemos:

Filósofos	Modelo original			Producto		
	Estados	Transiciones	Nodos	Estados	Transiciones	Nodos
3	956	3048	765	1170	3699	1603
4	9440	40120	1650	12161	51204	4005
5	93068	494420	2854	123914	651935	7782
6	917424	5848524	4339	1250306	7892280	13172
7	9043420	67259808	6101	12534636	92300873	19822
8	89144512	757721264	8150	125037881	1052244328	27574
9	878732012	8402796252	10486	1242307383	11761616484	36603
10	8662001936	92032909540	13109	12302615498	129425665440	46760

Filósofos	Tiempo en segundos					Prob.
	Rabin	Producto	CT	Alcanzabilidad	Total	mínima
3	0.062	0.032	0.042	0.002	0.139	1
4	0.064	0.136	0.384	0.001	0.595	1
5	0.062	0.466	2.086	0.002	2.618	1
6	0.067	1.193	9.778	0.002	11.042	1
7	0.068	2.602	41.766	0.002	44.441	1
8	0.068	5.378	169.970	0.002	175.421	1
9	0.068	9.686	615.893	0.002	625.652	1
10	0.074	14.884	5970.809	0.001	5985.771	1

8.3. Binary exponential backoff

El algoritmo de *binary exponential backoff* es una parte crucial del protocolo CSMA/CD (*Carrier Sense, Multiple Access with Collision Detection*) utilizado inicialmente en los estándares IEEE 802.3, también conocidos en su conjunto como *Ethernet*. CSMA/CD es utilizado para resolver la contención entre varios hosts para enviar mensajes a través de un canal único común.

Para enviar un mensaje, un host escucha el canal hasta que éste se desocupe y luego transmite su mensaje. Si otro host también transmite se produce una *colisión* que destruye los contenidos del mensaje, y por lo tanto los hosts deben encontrar una forma de enviar satisfactoriamente sus mensajes sin más información que la detección de colisiones. El algoritmo ataca este problema haciendo que ambos hosts esperen una cantidad aleatoria acotada de tiempo, múltiplo del tiempo necesario para propagar una señal en todo el medio, antes de reintentar el envío. De darse nuevamente una colisión, se duplica la cota máxima de tiempo y toma otro número al azar de este intervalo. Esto se repite hasta que se alcanza una cantidad máxima de intentos o hasta que el mensaje comienza a ser enviado satisfactoriamente, con lo que el host toma control del canal y puede enviar el resto del mensaje mientras que el resto de los hosts espera hasta que se desocupe el canal. El nombre del algoritmo proviene de las cotas crecientes para la elección del tiempo de espera, que son potencias de 2. Generalmente el algoritmo es *truncado*, de modo que al cabo de una cantidad determinada de intentos la cota deja de incrementarse.

Modelamos una versión simplificada de este problema donde una cantidad de hosts intenta enviar un mensaje al mismo tiempo con una cota máxima de 8 intervalos, y verificamos la probabilidad de satisfacción de las siguientes propiedades:

- $\mathsf{F}\ (\exists i.\acute{e}xito_i)$, o sea, que alguno de los hosts en contención consigue enviar satisfactoriamente su mensaje.

Intentos	Hosts	Modelo original			Producto		
		Estados	Transiciones	Nodos	Estados	Transiciones	Nodos
4	2	457	844	946	457	844	1042
	3	11721	25850	3823	11721	25850	4101
	4	166385	450336	9315	166385	450336	9870
	5	2319849	7436402	18220	2319849	7436402	19052
	6	32285417	119845992	29995	32285417	119845992	31104
5	2	777	1450	1240	777	1450	1363
	3	41209	91504	6156	41209	91504	6628
	4	897409	2353904	17341	897409	2353904	18524
	5	19113453	57971074	36640	19113453	57971074	38618
6	2	1097	2056	1377	1097	2056	1518
	3	99545	227154	8165	99545	227154	8727
	4	3032993	8285980	26267	3032993	8285980	27771

Intentos	Hosts	Tiempo en segundos					Prob.
		Rabin	Producto	CT	Alcanzabilidad	Total	mínima
4	2	0.068	0.020	0.022	0.087	0.191	0.98438
	3	0.062	0.110	0.138	0.725	1.036	0.99655
	4	0.062	0.309	0.526	6.086	6.987	0.99473
	5	0.068	0.643	1.937	40.271	42.925	0.99382
	6	0.065	1.108	6.320	340.850	348.459	0.99247
5	2	0.062	0.030	0.028	0.156	0.279	0.99805
	3	0.062	0.226	0.204	2.559	3.055	0.99972
	4	0.068	0.754	1.043	51.227	53.102	0.99962
	5	0.065	1.707	3.955	498.113	503.990	0.99955
6	2	0.075	0.038	0.032	0.231	0.363	0.99976
	3	0.062	0.377	0.238	7.900	8.580	0.99998
	4	0.069	1.386	1.362	199.657	202.685	0.99997

- $\mathsf{G}\ (\neg((\exists i : \mathit{fallo}_i) \wedge (\mathsf{F}\ \exists j : \mathit{éxito}_j)))$, o que no se dé que un host se apodera del canal al mismo tiempo o luego de que otro host se haya dado por vencido por error al suponer que el medio está roto.

Intentos	Hosts	Modelo original			Producto		
		Estados	Transiciones	Nodos	Estados	Transiciones	Nodos
4	2	457	844	946	457	844	979
	3	11721	25850	3823	11721	25850	4315
	4	166385	450336	9315	166385	450336	10171
	5	2319849	7436402	18220	2319849	7436402	19483
	6	32285417	119845992	29995	32285417	119845992	31617
5	2	777	1450	1240	777	1450	1277
	3	41209	91504	6156	41209	91504	6912
	4	897409	2353904	17341	897409	2353904	18908
	5	19113453	57971074	36640	19113453	57971074	39096
6	2	1097	2056	1377	1097	2056	1414
	3	99545	227154	8165	99545	227154	9030
	4	3032993	8285980	26267	3032993	8285980	28229

Intentos	Hosts	Tiempo en segundos					Prob.
		Rabin	Producto	CT	Alcanzabilidad	Total	mínima
4	2	0.072	0.020	0.067	0.001	0.162	1.00000
	3	0.065	0.119	1.016	1.335	2.539	0.95724
	4	0.072	0.328	17.769	12.388	30.616	0.88596
	5	0.076	0.690	323.786	87.500	412.049	0.79696
	6	0.080	1.136	5574.260	575.652	6151.185	0.69869
5	2	0.065	0.031	0.114	0.001	0.212	1.00000
	3	0.066	0.278	3.153	5.222	8.736	0.99313
	4	0.075	0.791	101.370	82.417	185.150	0.97758
	5	0.082	1.794	3238.850	901.554	4142.287	0.95190
6	2	0.064	0.037	0.157	0.010	0.259	1.00000
	3	0.066	0.394	6.341	11.583	18.370	0.99897
	4	0.077	1.424	311.844	298.170	611.517	0.99603

En ambos casos los tamaños de los productos fueron iguales a los de los modelos originales, puesto que los estados del modelo se corresponden directamente con los estados de los autómatas de Rabin de las propiedades.

Capítulo 9

Conclusiones

Se implementó con éxito un model checker de propiedades LTL sobre la herramienta PRISM. El uso de métodos simbólicos a lo largo de todo el proceso de verificación resulta en un consumo de memoria que crece linealmente respecto al tamaño del sistema, incluso cuando la cantidad de estados y transiciones crece exponencialmente.

Este código, con algunos agregados y modificaciones, ha sido integrado al árbol de desarrollo de PRISM y formará parte de una versión futura. Los model checkers de PRISM y su lenguaje de especificación de propiedades han sido extendidos a PCTL*, con el algoritmo implementado en este trabajo a cargo de la verificación de las fórmulas de camino para los procesos de decisión de Markov vistos en este trabajo, y próximamente también para las cadenas de Markov de tiempo discreto en los que no hay no determinismo. Además se le ha agregado la verificación de propiedades LTL bajo la hipótesis de *fairness* de [Bai98], para la que la construcción del producto del modelo y el autómata de la propiedad es la misma pero cambia levemente la definición de estados aceptados y su búsqueda.

Adicionalmente se han implementado otras optimizaciones al proceso de model checking. Como la búsqueda de componentes fuertemente conexas consume junto con el cálculo de alcanzabilidad la mayor parte del tiempo de verificación, se ha incluído un algoritmo adicional [GPP02] que tiene una complejidad lineal respecto de la cantidad de nodos del grafo, aunque en la práctica tiene una performance similar al algoritmo de *lockstep*. Al poder realizarse modificaciones sobre el resto de PRISM, también se ha podido alterar la posición de las variables agregadas en la verificación en el orden de los BDD, resultando en un consumo de memoria considerablemente menor para el SNP producto debido al uso de una menor cantidad de nodos para almacenar la función de transición, y una mayor velocidad en el resto del algoritmo que opera sobre el mismo.

Bibliografía

[Bai98] Christel Baier, *On the algorithmic verification of probabilistic systems*, Habilitation, Universität Mannheim, 1998.

[BGS00] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi, *An algorithm for strongly connected component analysis in n log n symbolic steps*, Form. Methods Syst. Des. **28** (2000), no. 1, 37–56.

[Bry86] Randal E. Bryant, *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers **35** (1986), no. 8, 677–691.

[BW96] Beate Bollig and Ingo Wegener, *Improving the variable ordering of OBDDs is NP-Complete*, IEEE Trans. Comput. **45** (1996), no. 9, 993–1002.

[CL06] Maximiliano Combina and Matías Lee, *Model checking cuantitativo con enfoque en la teoría de autómatas*, Trabajo especial, Universidad Nacional de Córdoba, 2006.

[Coo71] Stephen A. Cook, *The complexity of theorem-proving procedures*, STOC '71: Proceedings of the third annual ACM symposium on Theory of computing (New York, NY, USA), ACM, 1971, pp. 151–158.

[CY95] Costas Courcoubetis and Mihalis Yannakakis, *The complexity of probabilistic verification*, J. ACM **42** (1995), no. 4, 857–907.

[dA97] Luca de Alfaro, *Temporal logics for the specification of performance and reliability*, Symposium on Theoretical Aspects of Computer Science, 1997, pp. 165–176.

[dA98] ———, *Formal verification of probabilistic systems*, Ph.D. thesis, Stanford, CA, USA, 1998, Adviser-Zohar Manna.

[DFC02] Marie Duot, Laurent Fribourg, and Claudine, *Randomized dining philosophers without fairness assumption*, 2002.

[Doo94] J.L. Doob, *Measure theory*, Springer-Verlag, 1994.

[EL86] E. A. Emerson and C. L. Lei, *Efficient model checking in fragments of the propositional mu-calculus*, Proceedings of the 1st Symp. on Logic in Computer Science, 1986.

[EL87] E. Allen Emerson and Chin-Laung Lei, *Modalities for model checking: branching time logic strikes back*, Sci. Comput. Program. **8** (1987), no. 3, 275–306.

[FFK+01] Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, and Zijiang Yang, *Is there a best symbolic cycle-detection algorithm?*, TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (London, UK), Springer-Verlag, 2001, pp. 420–434.

[GO01] Paul Gastin and Denis Oddoux, *Fast LTL to Büchi automata translation*, 2001.

[GPP02] R. Gentilini, C. Piazza, and A. Policriti, *Computing strongly connected components in a linear number of symbolic steps*, 2002.

[GPP03] Raffaella Gentilini, Carla Piazza, and Alberto Policriti, *Computing strongly connected components in a linear number of symbolic steps*, SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 2003, pp. 573–582.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper, *Simple on-the-fly automatic verification of linear temporal logic*, Protocol Specification Testing and Verification (Warsaw, Poland), Chapman & Hall, 1995, pp. 3–18.

[HJ94] Hans Hansson and Bengt Jonsson, *A logic for reasoning about time and reliability*, Formal Aspects of Computing **6** (1994), no. 5, 512–535.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, *PRISM: A tool for automatic verification of probabilistic systems*, Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06) (H. Hermanns and J. Palsberg, eds.), LNCS, vol. 3920, Springer, 2006, pp. 441–444.

[HR00] Michael R. A. Huth and Mark D. Ryan, *Logic in computer science: Modelling and reasoning about systems*, Cambridge University Press, Cambridge, England, 2000.

[IJ90] A. Israeli and M. Jalfon, *Token management schemes and random walks yield self-stabilizating mutual exclusion*, Proc. ACM Symposium on Principles of Distributed Computing, 1990, pp. 119–131.

[JEK+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, *Symbolic Model Checking:* 10^{20} *States and Beyond*, Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (Washington, D.C.), IEEE Computer Society Press, 1990, pp. 1–33.

[JKK66] J. Snell J. Kemeny and A. Knapp, *Denumerable markov chains*, D. Van Nostrand Company, 1966.

[Kle05] Joachim Klein, *Linear time logic and deterministic omega-automata*, January 2005.

[Par02] D. Parker, *Implementation of symbolic model checking for probabilistic systems*, Ph.D. thesis, University of Birmingham, 2002.

[Pnu77] Amir Pnueli, *The temporal logic of programs*, FOCS, 1977, pp. 46–57.

[RL94] Michael O. Rabin and Daniel Lehmann, *The advantages of free choice: a symmetric and fully distributed solution for the dining philosophers problem*, 333–352.

[Saf89] Shmuel Safra, *Complexity of automata on infinite objects*, Ph.D. thesis, Rehovot, Israel, 1989.

[SB00] Fabio Somenzi and Roderick Bloem, *Efficient Büchi automata from LTL formulae*, July 2000, LNCS 1855, pp. 248–263.

[Som99] Fabio Somenzi, *Binary decision diagrams*, 1999.

[Tar72] Robert Tarjan, *Depth-first search and linear graph algorithms*, SIAM Journal on Computing **1** (1972), no. 2, 146–160.

[Tau06] Heikki Tauriainen, *Automata and linear temporal logic: Translations with transition-based acceptance*, Research Report A104, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, September 2006, Doctoral dissertation.

[XB99] Aiguo Xie and Peter A. Beerel, *Implicit enumeration of strongly connected components*, ICCAD, 1999, pp. 37–40.