Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba

Abstracción a Estados Esenciales en el Model Checker Probabilista PRISM

Nicolás H. Zandarin
Director: Dr. Pedro R. D'Argenio

Córdoba, 17 de diciembre de 2010

Resumen

En este trabajo se presenta una adaptación al model checking simbólico de un método de reducción de estados, el cual, tiene como objetivo reducir el costo de los cálculos numéricos involucrados en el model checking probabilista. El método procede eligiendo estados distinguidos, que llamamos esenciales, como representantes de los estados que convergen con probabilidad 1 a tales estados. El espacio de estados se reduce, luego, al conjunto de estados esenciales y las transiciones se adaptan apropiadamente a esta reducción. También, se presenta una implementación del mismo sobre el model checker simbólico PRISM y los resultados obtenidos al verificar propiedades cuantitativas sobre diversos casos de estudio.

Clasificación ACM 1998:

D.2.4 (Software/Program Verification/Model checking)

Palabras clave:

Métodos formales, Model checking Probabilista, Explosión del espacio de estados, MTBDD, PRISM, Estados esenciales

Índice general

1. Introducción **3**
- 1.1. Model checking . . . 4
  - 1.1.1. Model checking probabilista-no determinista . . . 4
  - 1.1.2. Lógica temporal . . . 5
  - 1.1.3. Model checking simbólico . . . 5
- 1.2. Este trabajo . . . 6
- 1.3. PRISM model checker . . . 6
- 1.4. Organización del trabajo . . . 6

2. Fundamentos **8**
- 2.1. Sistemas no deterministas probabilistas . . . 8
- 2.2. Lógica temporal probabilista . . . 12
- 2.3. Model Checking . . . 13
- 2.4. Reducción a estados esenciales . . . 16

3. Diagramas de decisión binarios multi-terminales **19**
- 3.1. MTBDD reducido y ordenado . . . 20
- 3.2. Operaciones . . . 22
- 3.3. Representación de vectores y matrices . . . 26

4. Model checking simbólico **28**
- 4.1. Representación y construcción de modelos con MTBDD . . . 28
  - 4.1.1. Codificación de estados . . . 29
  - 4.1.2. Codificación del no determinismo . . . 31
  - 4.1.3. Construcción del MTBDD . . . 32
- 4.2. Model checking probabilista sobre MTBDD . . . 34

5. Reducción a estados esenciales **39**
- 5.1. Algoritmos . . . 40
  - 5.1.1. Cálculo de los estados esenciales . . . 40
  - 5.1.2. Modificación del SNP . . . 43
- 5.2. Reducción a estados esenciales con MTBDD . . . 44

6. Casos de estudio **46**
6.1. Binary exponential backoff 46
6.1.1. MTBDD 47
6.1.2. Híbrido 49
6.2. Zeroconf 50
6.2.1. MTBDD 51
6.2.2. Híbrido 53
6.3. Bounded retransmission protocol 55
6.3.1. MTBDD 55
6.3.2. Híbrido 57

7. Conclusión **59**

Capítulo 1

Introducción

Interactuamos con sistemas computarizados la mayoría de nuestros días y cada vez dependemos más de ellos. Por esta razón, la necesidad de que funcionen apropiadamente es también mayor. Muchos de nosotros confiamos en nuestro celular para organizarnos y comunicarnos, y en caso de un fallo del mismo podemos llegar a tener un gran disgusto. Sin embargo existen otros casos en donde una falla tiene graves consecuencias y no puede ser tolerada. Tal es el caso de sistemas críticos como frenos en vehículos, corazones artificiales o controladores de trafico aéreo.

Además es importante notar que no es suficiente que para que un sistema sea considerado correcto, realice su tarea en el orden debido, si no que también es importante el tiempo transcurrido entre cada paso. Es decir, de nada sirve un sistema de frenos que, luego de presionar el pedal, demore un minuto en detener la rueda; o que luego de frenar jamás la libere. Sumado a esto, se encuentra que los sistemas son cada vez más complejos, por lo que las técnicas de *pruebas* o *testing* convencionales son cada vez más inadecuadas dado que los posibles escenarios e interacciones entre los mismos son mayores.

Una rama de la ciencia de la computación que aborda este problema es la *verificación formal*. Esta se encarga de verificar la corrección del sistema analizando sus propiedades de manera estática, posiblemente a partir del diseño del sistema, mucho antes de que el mismo haya sido desarrollado o construido. Para esto, se obtiene un modelo del sistema y a partir de ahí se deriva una prueba matemática de su corrección.

Existen varios métodos y herramientas para asistir en la creación de la prueba, pero en muchos casos, sobre todo los que tienen interacción con el mundo físico, nos encontramos con la necesidad de tolerar fallos o eventos que pueden ocurrir en cualquier orden y permanecer funcionando correctamente un tiempo indefinido, como por ejemplo, un servidor en el que uno de sus discos puede dejar de funcionar repentinamente.

En algunos de estos casos, los métodos tradicionales de verificación son imposibles de aplicar, dada la gran complejidad que se desprende de esos comportamientos. Por esto, son de particular interés las técnicas de verificación que pueden ser completamente automatizadas y, a cambio de un mayor costo computacional, verificar la corrección del sistema. Uno de estos métodos formales es el *model checking*.

1.1. Model checking

Dado un modelo de un sistema y una propiedad deseada del mismo, conocemos como *model checking* a la verificación automática de esta propiedad con respecto al modelo. El modelo es una representación de un sistema real en el que se identifican todos los estados posibles en los que el sistema se puede encontrar y las transiciones que pueden ocurrir entre ellos. Por otro lado, la propiedad es una especificación de una característica o funcionamiento deseado del sistema, en un formalismo lógico. Ejemplos son: *"El sistema nunca entra en estado de error"*, *"Nunca se entrega el café si no se introdujo una moneda antes"*. El model checker, entonces, verifica mediante un análisis sistemático del modelo si la propiedad se cumple. Si no es así, este es capaz de generar un contraejemplo que señala el o los casos en que no se satisface.

1.1.1. Model checking probabilista-no determinista

Las propiedades mencionadas anteriormente son propiedades *cualitativas*, pero en muchos casos es vital que el sistema cumpla con propiedades *cuantitativas* para que se considere correcto. Son propiedades de este tipo: *"El sistema advierte el error en menos de 150 ms."*, *"el mensaje es recibido correctamente el 80 % de las veces"*.

Verificar estas propiedades requiere un mayor costo computacional comparado con las propiedades cualitativas ya que usualmente se necesitan realizar cálculos numéricos para obtener los resultados deseados. La primera propiedad expresa requerimientos de tiempo real, mientras que la segunda expresa probabilidades sobre el comportamiento. En este trabajo trataremos con propiedades del segundo tipo, es decir, las que expresan probabilidades. Además del comportamiento probabilista del modelo, también es útil poder expresar en el mismo el comportamiento no determinista inherente a muchos sistemas, es decir, la posibilidad de que varios eventos ocurran en cualquier orden, o uno en lugar del otro. Analicemos un ejemplo:

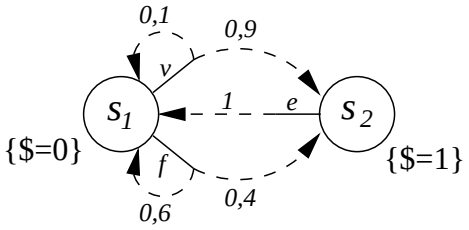


Figura 1.1: Modelo probabilista-no determinista

En la figura 1.1 tenemos el modelo una máquina que expende un producto con un valor de $1. Si se le paga con una moneda falsa, la recibe con un 40 % de probabilidad, y si se le paga con una verdadera, la acepta el 90 % de las veces. Cuando la acepta, el producto se entrega siempre (100 %), de lo contrario la máquina se "traga" la moneda. Entonces, el modelo consiste en 2 estados s_1 y s_2. Con líneas sólidas se representan los diferentes eventos que pueden ocurrir, sobre los que no se tiene control. En este

caso v denota ingresar una moneda verdadera y f una falsa, y e denota la entrega del producto. Finalmente con líneas punteadas vemos las transiciones de estados junto a sus probabilidades, donde 1 es 100 %.

1.1.2. Lógica temporal

El formalismo lógico que utilizaremos para expresar las probabilidades a verificar durante el model checking, es la *lógica temporal*.

Fue sugerida por A. Pnuelli para la especificación formal de propiedades sobre el comportamiento de un sistema. La *lógica temporal* es una extensión de lógica modal que a su vez es una extensión de la lógica proposicional. Básicamente nos provee de operadores que se refieren al funcionamiento del sistema a través del tiempo. Uno de estos operadores es el operador **F** (finally) el cuál indica que el predicado siguiente será verdadero en algún momento futuro, es decir, en algún estado futuro de nuestro modelo. Por ejemplo *"F llega el mensaje"*.

Más adelante, la lógica temporal fue extendida ante la necesidad expresar propiedades cuantitativas que involucren probabilidades. Este tipo de lógicas temporales, es el utilizado en el *model checking probabilista*.

1.1.3. Model checking simbólico

Uno de los problemas a los que se enfrenta el *model checking* es al denominado *explosión del espacio de estados*. Esto se produce debido a que es necesario representar en el modelo todos los estados posibles del sistema, lo que da lugar a modelos de miles de millones de estados.

Es de gran importancia entonces, almacenar los modelos de la manera más compacta posible, para dar lugar a la verificación de sistemas más grandes.

Con este objetivo se comenzó el desarrollo del model *checking simbólico*, en el que los modelos son almacenados utilizando *diagramas de decisión binarios* (BDD), con los que se logra reducir el uso de memoria significativamente. Debido a que los BDD sólo pueden almacenar los valores 1 y 0, para el caso del model checking probabilista utilizamos una extensión denominada *diagramas de decisión binarios multi-terminales* (MTBDD) los cuales pueden tomar cualquier valor, con los que podemos expresar los distintos valores de las probabilidades.

Los algoritmos de *model checking* tuvieron que ser adaptados para poder aplicarse sobre este tipo de almacenamiento. Esto dio lugar a una gran investigación heurística, debido a la gran sensibilidad de los BDD (y en consiguiente MTBDD), con respecto al orden en el que se almacenan los datos. Lamentablemente, este ahorro en espacio produce un aumento en el costo computacional, particularmente, durante los cálculos numéricos necesarios en el *model checking probabilista*. Para solucionar este problema, se diseñó un método *híbrido*, el cual, intenta tomar lo mejor de los métodos simbólicos que utilizan MTBDD y de los métodos explícitos más tradicionales. El *método de reducción a estados esenciales* desarrollado en este trabajo se aplica tanto al *model checking simbólico* puro, como a la variante *híbrida*.

1.2. Este trabajo

El objetivo de este trabajo es incorporar al *model checking probabilista simbólico*, un método para reducir el costo computacional del *model checking* de propiedades cuantitativas sobre modelos probabilistas.

El método se denomina *reducción a estados esenciales* y fue presentado en [DJ+02]. Básicamente, éste se aplica durante el *model checking*, en una etapa previa al cálculo de las probabilidades. Al finalizar, nuestro método elimina todos los estados del modelo, excepto un conjunto especial denominado *estados esenciales*. Luego, a partir del modelo reducido, se obtienen los mismos resultados que con el modelo completo, pero a un costo computacional menor.

En este trabajo se diseñó un método iterativo puramente simbólico para ser utilizado durante el *model checking simbólico*. El mismo fue implementado utilizando MTBDD en el *model checker simbólico* PRISM y como mencionamos, su aplicación fue extendida para el caso *híbrido*.

1.3. PRISM model checker

PRSIM es un *model checker* probabilista simbólico. Mediante su propio lenguaje (también soporta PEPA), permite expresar los modelos probabilistas *cadenas de Markov discretas* (DTMC) y *procesos de decisión de Markov* (MDP). Este último también exhibe comportamiento no determinista. Sobre ambos modelos se pueden evaluar propiedades expresadas en *lógica de computación de árboles probabilista* (PCTL/pCTL) y en las últimas versiones, también en *lógica temporal lineal* (LTL).

Además de los modelos discretos, implementa el model checking sobre *cadenas de Markov continuas* (CTMC) con propiedades expresadas en *Lógica estocástica continua* (CSL).

PRISM permite realizar el model checking de tres maneras diferentes: Almacenando los modelos de manera explícita utilizando como tipo de datos *matrices ralas*, de manera puramente simbólica mediante MTBDD, o combinando de ambos tipos de datos, en el denominado método *híbrido*.

Los estudios de la implementación de nuestro método de reducción se realizaron con modelos MDP y propiedades PCTL, por ser la naturaleza probabilista y no determinista de los MDP la más sensible a nuestra reducción. Sin embargo también se puede aplicar a modelos puramente probabilistas como CTMC y utilizando cualquier lógica temporal probabilista.

1.4. Organización del trabajo

En el capítulo 2, presentamos los fundamentos teóricos sobre los que desarrollamos este trabajo. En él definimos los modelos y la lógica temporal utilizadas, junto con los principios del *model checking*. Finalmente, presentamos nuestro método de reducción de estados

En el capítulo 3, detallamos los MTBDD, la estructura de datos sobre la que se basa el *model checking simbólico*.

En el capítulo 4, explicamos la implementación del *model checking simbólico* en PRISM. Comenzando por la construcción de los modelos, los problemas involucrados y como se efectúa la verificación de las propiedades sobre las estructuras de datos utilizadas.

En el capítulo 5, explicamos la adaptación del *método de reducción a estados esenciales* al model checking simbólico realizada en este trabajo, y su implementación sobre MTBDD.

En el capítulo 6, mostramos los casos de estudio analizados y las conclusiones sobre los casos favorables y desfavorables.

Finalmente, en el capítulo 7, expresamos las conclusiones a las que arribamos, y el trabajo futuro que se desprende de este.

Capítulo 2

Fundamentos

2.1. Sistemas no deterministas probabilistas

Estamos interesados en modelar sistemas que exhiben características aleatorias, es decir, cuyo flujo de comportamiento se puede expresar a través de probabilidades. Pero también queremos poder modelar casos en los que la probabilidad de un evento no está definida, si no que sólo tenemos conocimiento de los eventos posibles y cualquiera de estos ocurre de manera no determinista. Esto último es útil para modelar sistemas que reaccionan ante un estímulo (sistemas reactivos), o la concurrencia entre varios sistemas corriendo en paralelo.

Con ese propósito, siguiendo los lineamientos de [BdA95], utilizaremos los *sistemas no deterministas probabilistas* (SNP) que, como su nombre indica, nos permite construir un modelo con este comportamiento. En un SNP, la transición de un estado a otro consta de 2 etapas: primero, se elige una acción de manera no determinista y posteriormente, mediante una distribución de probabilidad asociada a la acción, se selecciona el estado siguiente. Cabe notar que esta expresividad no nos impide tener una transición completamente determinista (i.e. un estado con una sola acción cuya distribución se asocia al estado siguiente con probabilidad 1). Este modelo es similar al llamado *proceso de decisión de Markov* (MDP) [Par02] o *sistema de transición probabilista* (PTS) [DJ+02].

A continuación definimos un SNP, posteriormente, se hablará del concepto de *ejecución* y *estrategia*. Luego, definimos formalmente una medida de probabilidad sobre un SNP para, de esta manera, poder obtener las *probabilidades máximas y mínimas* de un conjunto de *ejecuciones* con respecto a éste.

Definición 2.1.1. *Un sistema no determinista probabilista (SNP) es una 7-upla* $(PA, S, A, V, \kappa, p, s_{in})$ *donde:*

- PA *es un conjunto de proposiciones atómicas.*
- S *es un conjunto finito de estados.*
- A *es un conjunto de acciones.*

- $V : S \to 2^{PA}$ *es la función de etiquetado que asocia a cada estado* $s \in S$ *el conjunto de proposiciones atómicas que son verdaderas en* s.
- $\kappa : S \to 2^{A} - \{\emptyset\}$ *es una función que asocia a cada estado un conjunto de acciones.*
- $p : S \times A \times S \to [0, 1]$ *es una función tal que para todo* $s \in S$ *y todo* $\mathrm{a} \in \kappa(s)$, $p(s, \mathrm{a}, \cdot)$ *es una distribución de probabilidad. (i.e.* $p(s, \mathrm{a}, \cdot) : S \to [0, 1]$ *tal que* $\sum_{t \in S} p(s, \mathrm{a}, t) = 1$*).*
- $s_{in} \in S$ *es el estado inicial.*

Intuitivamente, cada estado $s \in S$ tiene asociado un conjunto de acciones $\kappa(s)$ que representan las decisiones no deterministas que se pueden tomar en s. A su vez s y una acción $a \in \kappa(s)$ determinan la distribución de probabilidad $p(s, a, \cdot)$ que denota la probabilidad de realizar una transición a cualquier estado en S.

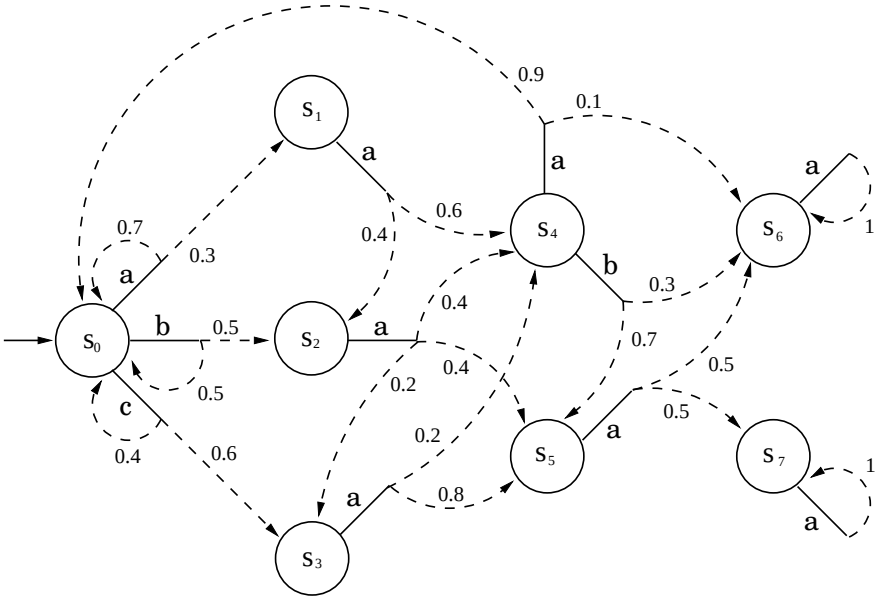


Figura 2.1: Sistema no determinista probabilista

La figura 2.1 muestra la *representación simbólica* de un ejemplo de SNP. En esta los estados se representan mediante círculos rotulados cada uno con un elemento de S. Aquí tenemos 8 estados, por lo que $S = \{s_0, \ldots, s_7\}$. El estado inicial se indica con una flecha de llegada, en nuestro ejemplo tenemos que $s_{in} = s_0$. Cada acción es representada con una línea sólida conectada al estado que pertenece. Para los nombres de las acciones utilizamos letras minúsculas. Notar que los nombres de acciones se pueden repetir en estados diferentes. En nuestro SNP, las acciones del estado s_0 son a, b y c, y las acciones del estado s_4 son a y b; por lo tanto $\kappa(s_0) = \{a, b, c\}$ y $\kappa(s_4) = \{a, b\}$. Luego para representar las transiciones dibujamos flechas con líneas entrecortadas que van desde una acción a un estado y se rotulan con la probabilidad de ser tomadas. Por ejemplo, la acción a del estado s_2 posee 3 transiciones: una hacia s_3 con probabilidad 0,2, otra

hacia s_4 con probabilidad 0,4 y finalmente, una hacia s_5 también con probabilidad 0,4; es decir que tenemos que $p(s_2, a, s_3) = 0{,}2$, $p(s_2, a, s_4) = 0{,}4$ y $p(s_2, a, s_5) = 0{,}4$. Notar cómo las probabilidades de cada acción siempre suman 1. En este caso no representamos las proposiciones atómicas, pero de ser necesario, se pueden enumerar entre llaves junto a los estados que las hacen verdaderas.

Partiendo de un estado s y tomando una decisión no determinista seguida de una probabilista podemos ir generando un camino de estados. Una serie infinita de estados obtenida de esta forma es llamada una *ejecución* del sistema.

Definición 2.1.2. *Una ejecución de un SNP* Π *es una secuencia infinita* $\omega = s_0 s_1 s_2 \dots$ *tal que para todo* $s_i \in S$ *existe* $\mathrm{a}_i \in \kappa(s_i)$ *y ocurre que* $p(s_i, \mathrm{a}_i, s_{i+1}) > 0$ *para todo* $i \geq 0$. *Una ejecución finita de un SNP* Π *es una secuencia finita de estados* σ *tal que existe una secuencia infinita de estados* ω *de tal manera que* $\sigma\omega$ *es una ejecución de* Π. *Dada una ejecución* $\omega = s_0 s_1 s_2 \dots$, *escribiremos* ω_i *para denotar a* s_i.

Por ejemplo, si tomamos el SNP de la figura 2.1 en la página anterior, podemos ver que las sucesiones de estados $(s_0)^\omega$, $(s_0 s_3 s_4)^\omega$ y $s_0 s_1 s_2 s_5 (s_6)^\omega$ son todas *ejecuciones*. De la misma manera, se puede ver que ninguna de las siguientes es una *ejecución* en el SNP: $(s_0 s_1)^\omega$, $s_0 s_1 s_2 s_5 s_6$ y $s_0 s_3 (s_6)^\omega$.

Debido a la presencia del no determinismo en los SNP, una ejecución no nos permite determinar la probabilidad con la que se alcanza cada uno de sus estados ya que no tenemos la información de las acciones tomadas. Por ejemplo, si volvemos al SNP de la figura 2.1 en la página anterior y suponemos una *ejecución* $\sigma = (s_0 s_1 s_4 s_6 \dots)$, como desconocemos si la acción tomada entre los estados s_4 y s_6 es a o b, no podemos saber si esa transición fue realizada con probabilidad 0,1 o 0,3.

Para razonar sobre las decisiones no deterministas, podemos asumir que estas fueron hechas por un ente externo, el cual determina una *estrategia*. En esta *estrategia* se define que acción seleccionar basada en los estados transitados hasta el momento. Entonces, dada una *ejecución* ω podemos saber la probabilidad con la que fue alcanzado cada estado, pero sólo bajo la *estrategia* utilizada para generar ω.

Los resultados sobre el comportamiento probabilista de un SNP para una única *estrategia* son de poca utilidad. Pero, como se explica más adelante, podemos verificar propiedades significativas de un SNP computando la probabilidad *máxima* o *mínima* de observar algún comportamiento específico sobre todas las *estrategias* posibles.

Para esto necesitamos una estructura sobre las *ejecuciones* que esté compuesta de elementos medibles. Lo hacemos de la manera convencional, según [KSK66].

Notemos que a cada estado $s \in S$ se le puede asociar el conjunto Ω_s de todas las *ejecuciones* comenzadas en s.

Luego, sea $\mathcal{B}_s \subseteq 2^{\Omega_s}$ el álgebra de Borel más pequeño generado por los conjuntos de cilindros básicos $C_\sigma = \{\omega \in \Omega_s | \sigma \text{ es prefijo de } \omega\}$ para cada *ejecución finita* σ que parte de s.

Lamentablemente, no es posible definir una medida de probabilidad sobre los elementos de $\mathcal{B}_s$ ya que la pertenencia de una *ejecución* a uno de los conjuntos $\Delta \in \mathcal{B}_s$

depende de cómo se realizaron sus decisiones no deterministas. A pesar de esto, sí es posible definir una *probabilidad máxima* y una *probabilidad mínima* sobre cada conjunto de *ejecuciones* Δ denotadas $\mu^+(\Delta)$ y $\mu^-(\Delta)$ respectivamente. Para definir estas probabilidades, primero necesitamos formalizar el concepto de *estrategia*.

Definición 2.1.3. *Una estrategia η es un conjunto de probabilidades condicionales $\mathrm{Q}_\eta(\mathrm{a}|\ s_0 s_1 \dots s_n)$ donde $n \in \mathbb{N}$, $s_i \in S$ y $\mathrm{a} \in \kappa(s_n)$*

Entonces, dada una ejecución finita, la *estrategia* determina la probabilidad de elegir la acción siguiente.

Definición 2.1.4. *La probabilidad de un evento $\mathcal{A}$ en $\mathcal{B}_s$ bajo la estrategia η, se define como $\Pr_s^\eta(\mathcal{A})$.*
La probabilidad de una transición hacia $t \in S$ luego de la ejecución finita $s_0 \dots s_n$, está dada por:

$$\Pr_{s_0}^\eta(t|\ s_0 \dots s_n) = \sum_{\mathrm{a} \in \kappa(s_n)} p(s_n, \mathrm{a}, t) Q_\eta(\mathrm{a}|\ s_0 \dots s_n) \tag{2.1}$$

La probabilidad de recorrer una ejecución finita en Ω_{s_0}, bajo la estrategia η está dada por:

$$\Pr_{s_0}^\eta(s_0 \dots s_{n+1}) = \prod_{i=0}^{n} \Pr_{s_0}^\eta(s_{i+1}|\ s_0 \dots s_i) \tag{2.2}$$

Estas probabilidades para ejecuciones finitas dan lugar a una única medida de probabilidad $\mu_{s_0,\eta}$ sobre $\mathcal{B}_{s_0}$ que asocia cada $\Delta \in \mathcal{B}_{s_0}$ con su probabilidad $\mu_{s_0,\eta}$:

$$\mu_{s_0,\eta}(C_\sigma) = \Pr_{s_0}^\eta(\sigma) \tag{2.3}$$

Ejemplo 2.1. *Tomemos el SNP de la figura 2.1 en la página 9, y sea η una estrategia tal que:*
$\mathrm{Q}_\eta(\mathrm{a}|\ s_0) = 0{,}3 \quad \mathrm{Q}_\eta(\mathrm{a}|\ s_0 s_1) = 1 \quad \mathrm{Q}_\eta(\mathrm{a}|\ s_0 s_1 s_4) = 0{,}2 \quad \mathrm{Q}_\eta(b|\ s_0 s_1 s_4) = 0{,}8$

Entonces, la probabilidad de la ejecución finita $s_0 s_1 s_4 s_6$ en el SNP bajo la estrategia η es:
$\Pr_{s_0}^\eta(s_0 s_1 s_4 s_6)$
$= \Pr_{s_0}^\eta(s_1|\ s_0) \times \Pr_{s_0}^\eta(s_4|\ s_0\, s_1) \times \Pr_{s_0}^\eta(s_6|\ s_0 s_1 s_4)$
$= \big(\mathrm{Q}_\eta(\mathrm{a}|\ s_0) \times p(s_0, \mathrm{a}, s_1)\big) \times \big(\mathrm{Q}_\eta(\mathrm{a}|\ s_0 s_1) \times p(s_1, \mathrm{a}, s_4)\big) \times \big(\mathrm{Q}_\eta(\mathrm{a}|\ s_0 s_1 s_4) \times p(s_4, \mathrm{a}, s_6) +$
$\mathrm{Q}_\eta(b|\ s_0 s_1 s_4) \times p(s_4, b, s_6)\big)$
$= \big(0{,}3 \times 0{,}3\big) \times \big(1 \times 0{,}6\big) \times \big(0{,}2 \times 0{,}1 + 0{,}8 \times 0{,}3\big) = 0{,}01404$

Utilizando la definición de $\mu_{s_0,\eta}$ podemos definir las probabilidades *máxima* y *mínima* de la siguiente manera:

Definición 2.1.5. *La probabilidad máxima $\mu_s^+(\Delta)$ y la probabilidad mínima $\mu_s^-(\Delta)$ de un conjunto de comportamientos $\Delta \in \mathcal{B}_s$ están definidas por:*

$$\mu_s^+(\Delta) = \sup_\eta \mu_{s,\eta}(\Delta) \qquad \mu_s^-(\Delta) = \inf_\eta \mu_{s,\eta}(\Delta)$$

Luego $\mu_s^+(\Delta)$ y $\mu_s^-(\Delta)$ representan las probabilidades de que el sistema siga una evolución en Δ cuando las decisiones no deterministas son respectivamente tan favorables o desfavorables como sea posible.

2.2. Lógica temporal probabilista

Para expresar las propiedades a verificar sobre un modelo durante el *model checking*, convencionalmente se utiliza la lógica temporal. Existe una variedad de lógicas temporales, pero en primera instancia, la lógica que se utilice depende del tipo de modelo que se quiera verificar. Por ejemplo, en nuestro caso necesitamos una lógica que permita expresar requerimientos sobre probabilidades, ya que el modelo que queremos analizar es probabilista. En segundo lugar, la lógica utilizada define la expresividad de las propiedades permitidas. Habitualmente, ocurre que las lógicas temporales no son comparables entre sí, i.e. una permite expresar cosas que la otra no y viceversa. Finalmente, los algoritmos utilizados para verificar las propiedades sobre el modelo (los cuales dependen tanto de la lógica como del modelo) definen el costo computacional, es decir, el orden de tiempo de los algoritmos en términos del tamaño de la propiedad y el modelo.

Podemos encontrar fundamentalmente dos categorías de lógicas temporales: las lógicas temporales de tiempo lineal como LTL [Pnu77], donde cada momento tiene un único sucesor; y las lógicas temporales donde el tiempo se ramifica como CTL [EL87], donde el futuro en cada instante aun no ha sido determinado y los distintos caminos forman un árbol. Estas lógicas son un ejemplo de lógicas incomparables. Existe una lógica temporal no perteneciente a estas categorías que suma la capacidad de expresión de LTL y CTL denominada CTL*, lamentablemente, esta expresividad está acompañada con un incremento exponencial en la complejidad del *model checking*.

Para verificar propiedades sobre los SNP utilizaremos una extensión de CTL que incluye un *operador de probabilidad* $\mathcal{P}$, denominada PCTL [HJ94]. Es esencialmente lo mismo que pCTL de [ASB+95]. La idea intuitiva del operador es que un estado s satisface $\mathcal{P}_{\bowtie p}[\psi]$ si la probabilidad de tomar un camino desde s satisfaciendo ψ está en el intervalo definido por $\bowtie p$.

Sintaxis

Formalmente, distinguimos entre dos tipos de fórmulas: fórmulas de estado ϕ y fórmulas de camino ψ, las cuales son evaluadas sobre estados y *ejecuciones*, respectivamente. Es importante destacar que las propiedades de un modelo siempre se expresan en términos de fórmulas de estado. Las fórmulas de camino sólo ocurren como parámetros del operador $\mathcal{P}$.

Definición 2.2.1. *Sintaxis de PCTL:*

$$\begin{aligned} \phi &::= \ true \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \mathcal{P}_{\bowtie p}[\psi] \\ \psi &::= \ X\phi \mid \phi \,\mathcal{U}\, \phi \end{aligned}$$

Donde a es una proposición atómica, $\bowtie \in \{\leq, <, \geq, >\}$ y $p \in [0, 1]$

En las fórmulas de camino, los operadores que aceptamos son X (*Next*) y $\mathcal{U}$ (*Until*). Intuitivamente, $X\phi$ es verdadero si ϕ se satisface en el estado siguiente; $\phi_1 \ \mathcal{U} \ \phi_2$ es verdadero si ϕ_2 se satisface en algún punto del futuro y ϕ_1 es verdadero hasta entonces. Ahora definimos formalmente la semántica.

Semántica

Para un SNP Π, estado $s \in S$ y fórmula PCTL ϕ, con $s \models \phi$ indicamos que ϕ se satisface en s. Denotamos con $Sat(\phi)$ el conjunto $\{s \in S \mid s \models \phi\}$ de todos los estados que satisfacen la fórmula ϕ. Similarmente, para un camino ω que satisface la fórmula ψ, escribimos $\omega \models \psi$.

Definición 2.2.2. *Semántica de PCTL:*
Para un camino ω:

$$\begin{array}{lcl} \omega \models X\phi & \iff & \omega_1 \models \phi \\ \omega \models \phi_1 \,\mathcal{U}\, \phi_2 & \iff & \exists k \geq 0 \,.\, \exists i \leq k \,.\, (\omega_i \models \phi_2 \wedge \omega_j \models \phi_1 \,\forall j < i) \end{array}$$

Para un estado $s \in S$:

$$\begin{array}{lcl} s \models true & & \forall s \in S \\ s \models a & \iff & a \in V(s) \\ s \models \phi_1 \wedge \phi_2 & \iff & s \models \phi_1 \wedge s \models \phi_2 \\ s \models \neg\phi & \iff & s \not\models \phi \\ s \models \mathcal{P}_{\geq p}[\psi] & \iff & p_s^{min}(\psi) \geq p \\ s \models \mathcal{P}_{\leq p}[\psi] & \iff & p_s^{max}(\psi) \leq p \end{array}$$

Donde $p_s^{min}(\psi) = \mu_s^-(\{\omega \in \Omega_s \mid \omega \models \psi\})$ y $p_s^{max}(\psi) = \mu_s^+(\{\omega \in \Omega_s \mid \omega \models \psi\})$. Las semánticas para $\mathcal{P}_{<p}[\psi]$ y $\mathcal{P}_{>p}[\psi]$ se definen de manera similar.

Para ver que la semántica está bien definida, es posible mostrar por inducción en la estructura de ψ que $\{\omega \in \Omega_s \mid \omega \models \psi\} \in \mathcal{B}_s$ para toda ψ (fórmula de secuencia) [Var85].

2.3. Model Checking

Presentamos ahora los algoritmos de *model checking* utilizados para decidir si un SNP Π satisface una especificación ϕ escrita en PCTL.

La estructura del algoritmo comparte la misma estructura básica del algoritmo original para verificar CTL presentado en [CES86]. Dada una fórmula ϕ, se evalúa recursivamente el valor de verdad de cada sub-fórmula en cada estado $s \in S$, comenzando con las proposiciones atómicas de ϕ y siguiendo la definición recursiva de las fórmulas de estado, hasta que el valor de verdad de ϕ en sí mismo es calculado en todos los estados $s \in S$.

A partir del modelo se puede obtener directamente qué estados satisfacen una proposición atómica y, dado que se conoce que conjunto de estados satisfacen las fórmulas ϕ y ψ, la intersección de estos satisface $\phi \wedge \psi$ y el complemento del conjunto que satisface ϕ, satisface $\neg\phi$. El caso no trivial es el del operador $\mathcal{P}$, en el cual es necesario calcular las probabilidades relevantes y entonces identificar los estados que satisfacen la cota dada en la fórmula. La complejidad de este cálculo es, a lo sumo, polinomial en el tamaño del modelo. Por consiguiente, la complejidad del *model checking* de PCTL sobre SNP, es lineal en el tamaño de la fórmula y polinomial en el tamaño del modelo. [BdA95]

Next

El caso del operador *next* es relativamente simple:

$$p_s^{max}(X\phi) = \max_{a\in\kappa(s)} \left\{ \sum_{t\in Sat(\phi)} p(s,a,t) \right\}$$

$$p_s^{min}(X\phi) = \min_{a\in\kappa(s)} \left\{ \sum_{t\in Sat(\phi)} p(s,a,t) \right\}$$

El algoritmo para resolver estas ecuaciones según [Par02], consiste en lo siguiente: Sea m el número total de acciones de todos los estados del SNP. i.e. $m = \sum_{s\in S} |\kappa(s)|$. Podemos representar la función $p : S \times A \times S \to [0,1]$ como una matriz $m \times |S|$ denominada **Ac**, donde cada fila corresponde a una única acción de un estado. Esto significa que hay $|\kappa(s)|$ filas que le corresponden al estado s y no sólo una. Asumimos también que tenemos un vector indexado por estados $\vec{\phi}$ tal que $\vec{\phi}(s)$ es 1 si $s \models \phi$ y 0 caso contrario. Entonces, podemos llevar a cabo el cálculo de arriba en dos pasos: una multiplicación matriz-vector $\mathbf{Ac}\cdot\vec{\phi}$ que calcula la suma de todas las acciones produciendo un vector de largo m; y luego una operación que selecciona, de este vector, el valor máximo o mínimo de cada estado, según corresponda, reduciendo el largo de m a $|S|$.

Until

En este caso se requiere calcular la probabilidad $p_s^{max}(\phi_1 \ \mathcal{U} \ \phi_2)$ o $p_s^{min}(\phi_1 \ \mathcal{U} \ \phi_2)$. Definimos $S^d = \{s \in S \mid s \models \phi_2\}$ como el conjunto de *estados destino* y a $S^p = \{s \in S \mid s \models \phi_1\}$ como el conjunto de *estados intermedios*. También es útil determinar otros conjuntos de estados antes de comenzar el cálculo. Definimos S^{yes} y S^{no} como los conjuntos conteniendo los estados con probabilidad exactamente 1 y 0, respectivamente; y a $S^?$ como el conjunto conteniendo los estados para los que su probabilidad debe ser calculada, es decir, $S^? = S \setminus (S^{yes} \cup S^{no})$. Inicialmente se puede establecer a $S^{yes} = S^d$ y a $S^{no} = S \setminus (S^d \cup S^p)$, pero luego estos conjuntos pueden ser incrementados utilizando un par de simples algoritmos de punto fijo que denominamos *algoritmos de precómputo*. Estos no tienen un costo computacional significativo y ayudan a disminuir notablemente la tarea del cálculo de las probabilidades, disminuyendo el espacio de estados a evaluar posteriormente, ya que cómo vimos, mientras mayor sean S^{yes} y S^{no}, menor es $S^?$.

Los algoritmos para mejorar S^{yes} y S^{no} son diferentes, dependiendo si se quiere calcular p^{max} o p^{min}. En [Par02] ambos casos se explican con detalle. Para nuestro trabajo alcanza con mencionar que en la mayoría de las verificaciones estos algoritmos de precómputo reducen notablemente el costo computacional, y en algunas situaciones logran que $S^? = \emptyset$ por lo que luego no es necesario el cómputo numérico.

El cálculo de p^{max} o p^{min} para los estados $S^?$ puede ser efectuado resolviendo un problema de optimización lineal sobre las variables $\{x_s \mid s \in S^?\}$. Para el caso de p^{max}, el problema es:

Minimizar $\sum_{s \in S^?} x_s$ sujeto a las restricciones

$$x_s \geq \sum_{t \in S^?} p(s, a, t) \cdot x_t + \sum_{t \in S^{yes}} p(s, a, t)$$

para todo $s \in S^?$ y todo $a \in \kappa(s)$

y para el caso p^{min}:

Maximizar $\sum_{s \in S^?} x_s$ sujeto a las restricciones

$$x_s \leq \sum_{t \in S^?} p(s, a, t) \cdot x_t + \sum_{t \in S^{yes}} p(s, a, t)$$

para todo $s \in S^?$ y todo $a \in \kappa(s)$

En cualquier caso, el problema admite una única solución óptima y podemos hacer $p_s^{max} = x_s$ o $p_s^{min} = x_s$, respectivamente. Estos resultados se pueden encontrar en [BT91] [CY90] [dA97].

Los problemas de optimización lineal pueden ser resueltos usando métodos tradicionales como el método Simplex. De cualquier manera, estos métodos no son apropiados para el tamaño de los problemas que nos interesan. Por suerte, como se explica en [Bai98], es posible utilizar métodos iterativos para aproximar las probabilidades p_s^{max} y p_s^{min}. Entonces $p_s^{max} = \lim_{n \to \infty} p_s^{max(n)}$ donde:

$$p_s^{max(n)} = \begin{cases} 0 & \text{si } s \in S^{no} \\ 1 & \text{si } s \in S^{yes} \\ 0 & \text{si } s \in S^? \text{ y } n = 0 \\ \max_{a \in \kappa(s)} \left\{ \sum_{t \in S} p(s, a, t) \cdot p_s^{max(n-1)} \right\} & \text{si } s \in S^? \text{ y } n > 0 \end{cases}$$

y $p_s^{min} = \lim_{n \to \infty} p_s^{min(n)}$ donde:

$$p_s^{min(n)} = \begin{cases} 0 & \text{si } s \in S^{no} \\ 1 & \text{si } s \in S^{yes} \\ 0 & \text{si } s \in S^? \text{ y } n = 0 \\ \min_{a \in \kappa(s)} \left\{ \sum_{t \in S} p(s, a, t) \cdot p_s^{min(n-1)} \right\} & \text{si } s \in S^? \text{ y } n > 0 \end{cases}$$

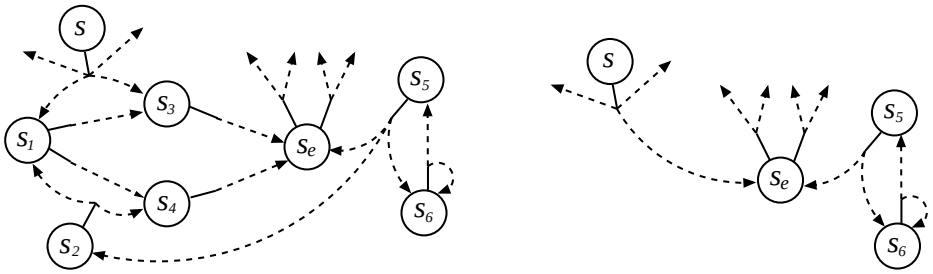
De esta manera se calculan $p_s^{max(n)}$ y $p_s^{min(n)}$ para n sucesivos, finalizando cuando se satisfaga algún criterio de convergencia. Asumiendo nuevamente que p es representada en una matriz, cada iteración puede ser realizada con una multiplicación matriz-vector y una operación de máximo o mínimo.

2.4. Reducción a estados esenciales

En este trabajo estamos interesados en reducir el costo computacional del *model checking probabilista* de propiedades cuantitativas. Como vimos en la sección anterior, la parte más costosa del proceso es la obtención de las probabilidades $p_s^{max}(\phi_1 \mathcal{U} \phi_2)$ o $p_s^{min}(\phi_1 \mathcal{U} \phi_2)$. Particularmente el cálculo numérico necesario para obtener las probabilidades de los estados en $S^?$. En esta sección explicamos un método de reducción de estados introducido en [DJ+02] para reducir el conjunto de estados $S^?$. El método opera sobre este conjunto inmediatamente después de los algoritmos de *precómputo* y justo antes del comienzo de los cálculos numéricos. Denominamos a este método: *reducción a estados esenciales*.

Dado que la reducción depende de la propiedad $\phi_1 \mathcal{U} \phi_2$ a evaluar, de ahora en adelante asumiremos que los conjunto de estados $S^?$, S^{yes} y S^{no} son los dejados por los algoritmos de precómputo durante el *model checking* de la propiedad.

Por lo tanto, el *método de reducción a estados esenciales* consiste en eliminar todos los estados de $S^?$, excepto un tipo particular de estado denominado *esencial*. Intuitivamente se puede pensar que un *estado esencial* s_e "absorbe" a todos los estados a partir de los cuales cualquier elección de transiciones genera una ejecución que pasa por s_e, por lo que no es necesario calcular las probabilidades de las transiciones entre estos estados ya que cualquier ejecución pasa por s_e con probabilidad 1. Entonces, el algoritmo procede identificando los *estados esenciales* y los estados que absorbe, luego las transiciones que llegan a los estados absorbidos se redirigen al *esencial* correspondiente y finalmente los estados absorbidos se eliminan del modelo.



(a) Situación inicial (b) Después de la reducción

Figura 2.2: Reducción de estados

Ejemplo 2.2. *Tomemos el fragmento de SNP de la figura 2.2(a) y supongamos que* ϕ_1 *es una propiedad tal que se satisface en todos los estados del fragmento y* ϕ_2 *no se se satisface en ninguno de los estados del fragmento pero posiblemente vale en algún otro. Observamos que todas las transiciones iniciadas en* s_1,s_2,s_3,s_4 *llevan al estado* s_e *con probabilidad 1 en un número finito de pasos, por lo que* $p_{s_i}^{max}(\phi_1 \mathcal{U} \phi_2) = p_{s_e}^{max}(\phi_1 \mathcal{U} \phi_2)$ *y* $p_{s_i}^{min}(\phi_1 \mathcal{U} \phi_2) = p_{s_e}^{min}(\phi_1 \mathcal{U} \phi_2)$ *para* $i = 1, 2, 3, 4$. *Entonces, podemos reducir el sistema al representar todos los estados mencionados con* s_e, *y sumar todas las transiciones probabilistas que llevaban a los estados representados por* s_e. *Quedando como resultado, el SNP de la figura 2.2(b).*

Notar que además de reducir el tamaño del SNP, el análisis puede resolver algún no determinismo en el sistema, simplificando cualquier cálculo posterior.

Consideremos los estados s_5 y s_6 de la figura 2.2(a). Estos también tienen las probabilidades *máxima* y *mínima* idénticas a s_e. Pero debido a la transición entre s_5 y s_6 esa propiedad aparece únicamente cuando tomamos en cuenta ejecuciones infinitas. El análisis correspondiente es más costoso que si únicamente se toman en cuenta ejecuciones finitas. En este caso sólo consideramos el análisis basado en ejecuciones finitas.

A continuación formalizamos las ideas expuestas más arriba.

Definición 2.4.1. *Sea* Π *un SNP y sea* $s \in S$*. Definimos el conjunto de* estados siguientes *succ(s) como:*

$$succ(s) = \{t \in S \mid \exists \mathrm{a} \in \kappa(s) \cdot p(s, \mathrm{a}, t) > 0\}$$

Definición 2.4.2. *La Relación de Dominación* $\preceq$ *es la relación más pequeña que satisface lo siguiente: Para todo* $s \in S$*,* $s \preceq s$ *y para todo* $s \in S^?$*,* $t \in S$ *tal que* $s \neq t$*,*

$$s \preceq t \;\Leftarrow\; \forall s' \in succ(s) : s' \preceq t) \tag{2.4}$$

La relación $\preceq$ *es un orden parcial.*

Definición 2.4.3. *Un estado* $e \in S$ *es llamado* ***estado esencial*** *si es maximal con respecto a* $\preceq$

Definición 2.4.4. *Denotamos con* $\sim$ *a la relación inducida por* $\preceq$ *de la siguiente manera:*

$$s_1 \sim s_2 \;\Leftrightarrow\; \exists t : s_1 \preceq t \wedge s_2 \preceq t \tag{2.5}$$

La relación $\sim$ *es una relación de equivalencia.*

Notar que los estados relacionados con $\sim$ son dominados por el mismo estado esencial.

Queremos reducir el fragmento de SNP concreto a uno abstracto que contenga sólo *estados esenciales*, de tal manera que el fragmento abstracto preserve las probabilidades *máximas* y *mínimas*. Llamaremos al SNP abstracto una *abstracción de estados esenciales*. Por definición de $\sim$ sabemos que cada clase de equivalencia posee un único estado esencial. Estos estados esenciales pueden ser tomados como estados abstractos representando a todas las clases de equivalencia. Además cualquier distribución puede ser abstraída sumando todas las probabilidades de los estados de la misma clase de equivalencia y apuntando al estado que la representa. Veamoslo más formalmente.

Definición 2.4.5. *(Abstracción de estados esenciales). Sea* $\Pi = (PA, S, A, V, \kappa, p, s_{in})$ *un SNP y sea* $S^?$ *el conjunto de estados obtenido por los algoritmos de precómputo, donde asumimos que el conjunto de estados* $S^\perp = (S \setminus S^?)$ *es absorbente. Sea* $\mathcal{E}$ *la partición de* S *asociada a* $\sim$*, y sea* $E \subseteq S$ *el conjunto de estados esenciales. Para cualquier* $e \in E$*, denotemos con* $[e]$ *la clase de equivalencia de* e *en* $\mathcal{E}$*. El SNP esencial de* Π *es el PTS* $\Pi_{\mathcal{E}} = (PA, E, A, V, \kappa_{\mathcal{E}}, p_{\mathcal{E}}, e_{in})$*, donde* $S^\perp \subseteq E$*,* $e_{in} \in E$ *tal que* $s_{in} \in [e_{in}]$ *y donde para todo* $e \in E$ $\kappa_{\mathcal{E}}(e) = \kappa(e)$ *y para todo* $e, e' \in E$*,* $\mathrm{a} \in \kappa(e)$ *tenemos* $p_{\mathcal{E}}(e, \mathrm{a}, e') = \sum_{e'' \in [e']} p(e, \mathrm{a}, e'')$*.*

Lema 2.4.1. *Sea* $\Pi = (PA, S, A, V, \kappa, p, s_{in})$ *un SNP, sea* $(\phi_1 \mathcal{U} \phi_2)$ *la propiedad que se está verificando, y sea* $\Pi_{\mathcal{E}} = (PA, E, A, V, \kappa_{\mathcal{E}}, p_{\mathcal{E}}, e_{in})$ *su* ***abstracción de estados esenciales****. Para* $s \in S$ *dominado por* $e \in E$*, tenemos:*

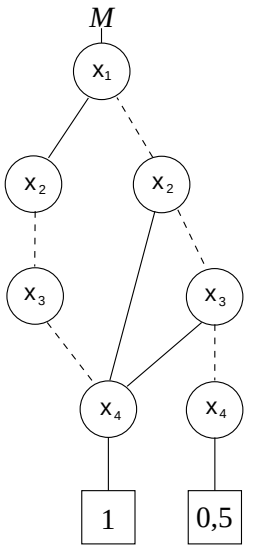
$$p^{max}_{s,\Pi}(\phi_1 \,\mathcal{U}\, \phi_2) = p^{max}_{e,\Pi_{\mathcal{E}}}(\phi_1 \,\mathcal{U}\, \phi_2) \quad y \quad p^{min}_{s,\Pi}(\phi_1 \,\mathcal{U}\, \phi_2) = p^{min}_{e,\Pi_{\mathcal{E}}}(\phi_1 \,\mathcal{U}\, \phi_2)$$

Donde $p^{max}_{s,\Pi}$ *es la función* p^{max}_{s} *interpretada en el SNP* Π*. (y similarmente para* $p^{min}_{s,\Pi}$*,* $p^{max}_{e,\Pi_{\mathcal{E}}}$ *y* $p^{min}_{e,\Pi_{\mathcal{E}}}$*).*

Capítulo 3

Diagramas de decisión binarios multi-terminales

En esta sección introducimos la estructura de datos más importante del model checking simbólico: los diagramas de decisión binarios multi-terminales (MTBDD) los cuales fueron propuestos en [CMZ+93]. Los MTBDD son una extensión de otra estructura llamada diagramas de decisión binarios (BDD) originalmente creados por Lee [Lee59] y Akers [Ake78]. Un BDD es un grafo utilizado para representar una función booleana n-aria $f : \mathbb{B}^n \to \mathbb{B}$. Los MTBDD extienden a los BDD permitiendo representar funciones que toman un valor de un conjunto D cualquiera y no sólo de $\mathbb{B}$, es decir, funciones de la forma $f : \mathbb{B}^n \to D$. Usualmente se toma a $\mathbb{R}$ como el conjunto D, y es lo que se hará en este trabajo.



(a)

x_1	x_2	x_3	x_4	f_M
0	0	0	1	0,5
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
otros casos				0

(b)

Figura 3.1: Un MTBDD M y la función que representa

Definición 3.1. *Sea $X = \{x_1, \ldots, x_n\}$ un conjunto de variables booleanas. Un diagrama de decisión binario multi-terminal de un vector $\vec{x} = (x_1, \ldots, x_n)$ en $\mathbb{R}$ es un grafo dirigido, acíclico con raíz tal que: Cada vértice es de uno de dos tipos:* ***interno*** *o* ***terminal****. Los vértices internos están etiquetados con una variable booleana $x_i \in X$ y poseen exactamente 2 hijos. Las aristas que conectan cada nodo con sus hijos están etiquetadas una con* ***0*** *y la otra con* ***1****. Los vértices terminales están etiquetados con un número real y no poseen hijos.*

Los vértices del grafo son denominados **nodos**. Representamos los **nodos internos** con círculos y los **nodos terminales** con cuadrados, cada uno con su etiqueta en el interior o con las etiquetas alineadas a la izquierda. Las aristas que conectan los nodos se representan con líneas. La línea sólida representa la arista etiquetada con **1** y la línea punteada la etiquetada con **0**.

De ahora en más asumiremos que un MTBDD toma siempre valores en $\mathbb{R}$.

Un MTBDD M sobre variables $(x_1, \ldots, x_n)$, representa la función $f_M(x_1, \ldots, x_n) : \mathbb{B}^n \to \mathbb{R}$. El valor de esta función se determina trazando un camino en M desde la raíz hasta un nodo terminal, por cada nodo interno m, tomando la arista **1** si la variable x_i de m es 1 o la arista **0** si es 0. En la figura 3.1 vemos un MTBDD y la función que representa. Por simplicidad, omitimos el nodo terminal 0 y todas las aristas hacia el mismo.

3.1. MTBDD reducido y ordenado

La razón por la que los MTBDD ocupan poco espacio de almacenamiento es porque se almacenan de manera reducida, es decir, omitiendo información redundante. La manera de llevar un MTBDD de su forma de árbol explícita a la forma reducida consta de 3 etapas:

Eliminación de nodos terminales duplicados (C1)

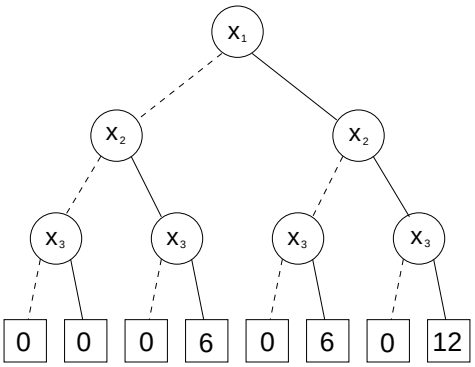
Si el MTBDD contiene más de un terminal con el mismo valor, entonces se redirigen todas las aristas que apuntan a esos nodos idénticos a un único nodo.

Eliminación de nodos redundantes (C2)

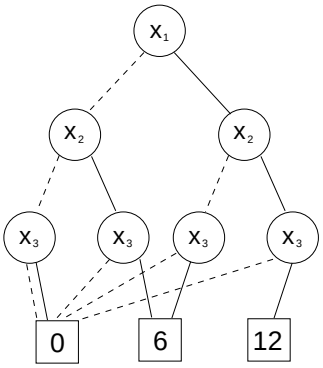
Si ambas aristas de un nodo n apuntan al mismo nodo m, entonces se elimina el nodo n y se redirigen todas sus aristas entrantes a m.

Eliminación de nodos internos duplicados (C3)

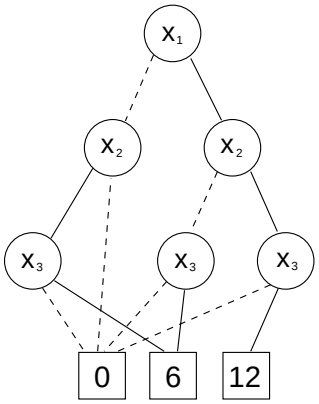
Notar que cualquier nodo interno de un MTBDD puede ser considerado como un sub-MTBDD. Entonces, si dos nodos distintos n y m son las raíces de dos sub-MTBDD estructuralmente idénticos, eliminamos uno de ellos y redirigimos todas sus aristas entrantes al otro.



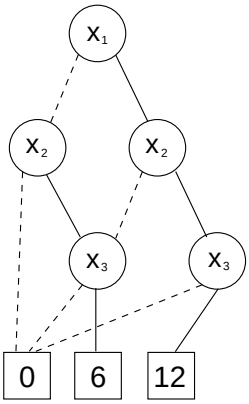
(a) El MTBDD en su forma de árbol sin reducir.



(b) Luego de aplicar **C1**



(c) Luego de aplicar **C2**



(d) El MTBDD *reducido*, luego de aplicar **C3**

Figura 3.2: Reducción de un MTBDD que representa la función $(2x_1 + 2x_2) \cdot 3x_3$

Definición 3.1.1. *Un MTBDD es un* ***MTBDD reducido*** *si no le pueden ser efectuadas las optimizaciones* ***C1,C2*** *y* ***C3****.*

En la figura 3.2 vemos el proceso completo de reducción de un MTBDD que representa la función $(2x_1 + 2x_2) \cdot 3x_3$.

Dar un orden a las variables en un MTBDD es muy importante por varias razones. Primero porque es muy difícil realizar operaciones entre MTBDD si sus variables están ordenadas de manera diferente. Además, si en un mismo camino desde la raíz hasta un nodo terminal ocurren 2 nodos rotulados con la misma variable booleana, estamos almacenando información redundante. Finalmente, lo más importante de todo: el MTBDD debe tener un orden definido, ya que es necesario para lograr que, cualquiera sea la función representada por el MTBDD, esta representación sea única.

Definición 3.1.2. *Sea $\vec{x} = (x_1, \ldots, x_n)$ una lista de variables donde no hay elementos duplicados. Y sea $<$ un orden total sobre las variables de tal manera que se cumple $x_1 < \cdots < x_n$. Un MTBDD está* ***ordenado*** *con respecto a $\vec{x}$ si todas las variables que rotulan sus nodos internos están en $\vec{x}$ y si dados 2 nodos m rotulado con x_i y w rotulado*

con x_j tal que m ocurre antes que w, entonces $x_i < x_j$.
Un MTBDD es ***ordenado*** *si tiene un orden para alguna lista de variables. Dos MTBDD M_1 y M_2 tienen* ***ordenes compatibles*** *si para cualquier par de variables x_i,x_j de M_1 tal que x_i ocurre antes que x_j, entonces también lo hace en M_2 y viceversa.*

Lema 3.1.1. *Un MTBDD ordenado y reducido que representa una función es único para todos los ordenes compatibles.*

Gracias a esto, verificar la equivalencia de 2 MTBDD se reduce a comparar sus estructuras.

El tamaño de un MTBDD representando una función dada es muy sensible tanto al orden de sus variables booleanas como a la manera de codificar la función con esas variables. Esto tiene un efecto directo en los requerimientos de espacio para la estructura de datos y en el tiempo necesario para realizar operaciones sobre ella. Lamentablemente, encontrar el orden optimo es un problema NP-Completo [BW96], pero existen análisis heurísticos [Par02] con los que se obtienen en general buenos resultados. Esto se explicará con más detalle en el capítulo 4.

El tamaño de un MTBDD también es afectado por el número de nodos terminales que contiene, es decir, el número de valores distintos que toma la función que representa. Sólo se puede obtener un MTBDD compacto maximizando la reutilización de la mayor cantidad de sub-MTBDD posible. Una gran cantidad de nodos terminales reduce la capacidad de reutilización y, por lo tanto, incrementa el número de nodos del MTBDD.

Gracias a esta gran reusabilidad, en algunas implementaciónes es posible almacenar una sola copia de un sub-MTBDD y reutilizarla en todos los MTBDD con ordenes compatibles que lo contienen, reduciendo aun más la memoria requerida para almacenar más de un MTBDD.

3.2. Operaciones

Desde su concepción, los MTBDD fueron pensados para representar funciones y, desde ese mismo momento se desarrollan adaptaciones de las operaciones más usuales, como la suma y el producto, para operar sobre los mismos. Con el consiguiente uso de MTBDD para almacenar diversos tipos de datos, otras operaciones más complejas fueron implementadas para trabajar con los nuevos objetos. Un caso particular es la librería CUDD utilizada por PRISM que, permite trabajar con MTBDD y BDD, y provee una extensa galería de operaciones.

Debido a la estructura de árbol con raíz de un MTBDD, la mayoría de las operaciones trabajan tomando la raíz y, efectuando algún cálculo recursivo sobre los sub-MTBDD de las aristas **0** y **1**. Esto continúa hasta llegar a los nodos terminales, donde la operación es propiamente efectuada. Definimos esto formalmente:

Definición 3.2.1. *Sea M un MTBDD sobre las variables $(x_1, \ldots, x_n)$, definimos su* ***cofactor*** *$M|_{x_i=b}$ tal que $b \in \{0, 1\}$ como el MTBDD sobre las variables $(x_1, .., x_{i-1}, x_{i+1}, .., x_n)$ representando la función $f_M(x_1, .., x_{i-1}, b, x_{i+1}, .., x_n)$.*

*Sea m un **nodo interno**, denotamos con $lo(m)$ y $hi(m)$ los nodos (hijos) de m alcanzados a través de las aristas **0** y **1**, respectivamente.*

Lema 3.2.1. *Para todas las funciones f_M de un MTBDD M y toda variable booleana x, ocurre:*

$$f_M = x \cdot f_{M|_{x=1}} + (1 - x) \cdot f_{M|_{x=0}}$$

A continuación describiremos todas las operaciones que necesitaremos para trabajar con los MTBDD. Trataremos a los BDD simplemente como un caso especial de MTBDD.

- `const(c)`
  Donde $c \in \mathbb{R}$, crea un nuevo MTBDD con el valor constante c, es decir, un único nodo terminal etiquetado con c.

- `apply(`op,M_1,M_2`)`
  Donde op es una operación binaria sobre los reales ($+$, $-$, $\times$, $\div$, min, max, etc.), devuelve el MTBDD que representa la función f_{M_1} op f_{M_2}. Si M_1 y M_2 son BDD, op puede ser una operación booleana ($\wedge$, $\vee$, $\rightarrow$, etc.).

  El algoritmo opera de manera recursiva en la estructura del MTBDD. Se basa en la separación en *cofactores* y en el lema 3.2.1.

  $$f_{M_1} \,\mathrm{op}\, f_{M_2} = (1 - x_i) \cdot (f_{M_1|_{x_i=0}} \,\mathrm{op}\, f_{M_2|_{x_i=0}}) + x_i \cdot (f_{M_1|_{x_i=1}} \,\mathrm{op}\, f_{M_2|_{x_i=1}})$$

  Por lo que el problema se va separando en 2 partes hasta alcanzar los nodos terminales en los que efectivamente se realiza la operación. El orden en que se van tomando las variables para realizar la separación en los cofactores es el inducido por el orden de ambos MTBDD.

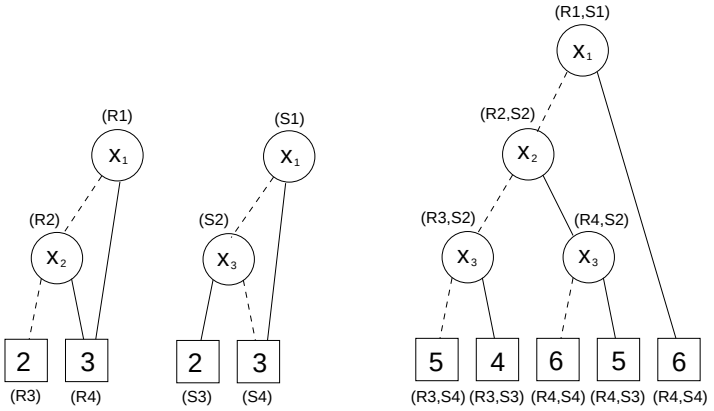
  Veamos ahora cómo es el cálculo que efectúa el algoritmo [HR00]. Supongamos que se quiere calcular `apply`(op, M_1, M_2), donde r_1 y r_2 son los nodos raíz de M_1 y M_2 respectivamente.

  1. Si ambos r_1 y r_2 son nodos terminales etiquetados con c_1 y c_2 respectivamente, entonces computamos $c_r = c_1 \,\mathrm{op}\, c_2$ y el MTBDD resultado es un nodo terminal y raíz etiquetado con c_r.
  2. Si ambos r_1 y r_2 están etiquetados con la misma variable booleana x_i, entonces el resultado es un nodo etiquetado con x_i cuya arista 0 está unida a `apply`$(\mathrm{op}, lo(r_1), lo(r_2))$ y su otra arista 1 está unida a `apply`$(\mathrm{op}, hi(r_1), hi(r_2))$.
  3. Si r_1 está etiquetado con x_i, pero r_2 es un nodo terminal o un nodo etiquetado con x_j tal que $x_j > x_i$, entonces sabemos que M_2 no posee nodo etiquetado con x_i ya que ambos MTBDD tienen orden compatible. Entonces, f_{M_2} es independiente de x_i ($f_{M_2} \equiv f_{M_2|_{x_i=0}} \equiv f_{M_2|_{x_i=1}}$). Por lo que el resultado es un nodo etiquetado con x_i cuya arista 0 está unida a `apply`$(\mathrm{op}, lo(r_1), r_2)$ y su otra arista 1 está unida a `apply`$(\mathrm{op}, hi(r_1), r_2)$.

4. Si r_2 es un nodo interno, pero r_1 es un nodo terminal o uno etiquetado con x_j con $x_j > x_i$ entonces el caso es simétrico al anterior.

El MTBDD resultante está *ordenado*, pero puede no estar *reducido* por lo que luego de la operación es necesario aplicarle las reducciones (C1, C2 y C3).

El algoritmo tiene complejidad exponencial en el tamaño de sus argumentos. Pero si notamos que `apply` es llamada varias veces con los mismos argumentos, podemos ganar eficiencia si esto fuese evaluado sólo la primera vez y el resultado fuera recordado en llamadas futuras. Esto no sólo ahorra cálculo sino que también produce un MTBDD que requiere menos reducción. Con esta técnica las llamadas a `apply` se reducen a $2 \cdot |M_1| \cdot |M_2|$ [Bry86]



(a) R_1 (b) S_1 (c) `apply`$(+, R_1, S_1)$

Figura 3.3: Algoritmo `apply`

Ejemplo 3.1. *Tomemos los MTBDD R_1 y S_1 de la figura 3.3(a) y 3.3(b) respectivamente. Estos representan las funciones f_{R_1} y f_{S_1} tal que:*

$f_{R_1}(0,0) = 2$, $f_{R_1}(0,1) = 3$, $f_{R_1}(1,0) = 3$ *y* $f_{R_1}(1,1) = 3$
$f_{S_1}(0,0) = 3$, $f_{S_1}(0,1) = 2$, $f_{S_1}(1,0) = 3$ *y* $f_{S_1}(1,1) = 3$

Ambos están ***reducidos*** *y* ***ordenados****. Además el orden es compatible ya que las variables que comparten ocurren en el mismo orden y, por lo tanto, podemos operar sobre ellos aplicando* ***apply****. Notar que R1 es un MTBDD en (x_1, x_2) y S1 es un MTBDD en (x_1, x_3). por lo que al momento de realizar la operación es necesario definir el orden entre las variables que no comparten. En este caso utilizaremos (x_1, x_2, x_3). En la figura 3.3(c) podemos ver el resultado de* ***apply***$(+, R_1, S_1)$ *(sin completar la reducción) en donde se señalan los subMTBDD con los que se llama a* ***apply*** *para obtener cada nodo. Denotamos T al MTBDD que representa la función $f_{S_1+R_1}$:*

$f_T(0,0,0) = f_{R_1}(0,0) + f_{S_1}(0,0) = 5$, $f_T(0,0,1) = f_{R_1}(0,0) + f_{S_1}(0,1) = 4$
$f_T(0,1,0) = f_{R_1}(0,1) + f_{S_1}(0,0) = 6$, $f_T(0,1,1) = f_{R_1}(0,1) + f_{S_1}(0,1) = 5$

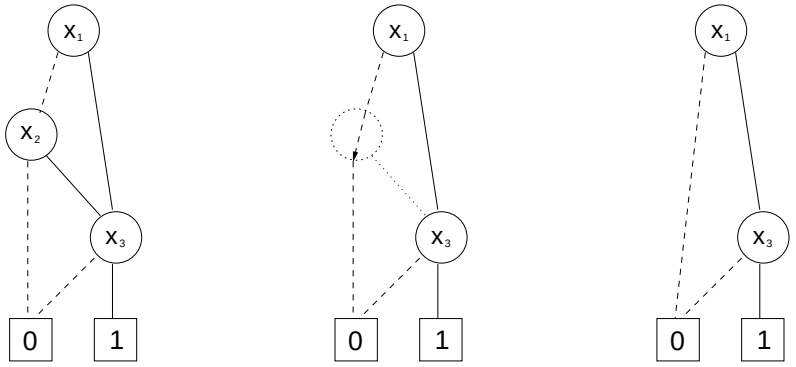
$f_T(1,0,0) = f_{R_1}(1,0) + f_{S_1}(1,0) = 6,\ f_T(1,0,1) = f_{R_1}(1,0) + f_{S_1}(1,1) = 6$
$f_T(1,1,0) = f_{R_1}(1,1) + f_{S_1}(1,0) = 6,\ f_T(1,1,1) = f_{R_1}(1,1) + f_{S_1}(1,1) = 6$

Finalmente, observamos que durante el cálculo de T, hay 2 nodos que resultan de operar con exactamente los parámetros (R_4, S_4). y otros 2 nodos que, tomando en cuenta la conmutatividad de la operación, también pueden calcularse una única vez: (R_3, R_4) y (R_4, R_3). Por lo que el MTBDD final se obtiene en 7 llamadas recursivas a ***apply****, 3 de las cuales son las sumas.*

- `not`(M)
  Donde M es un BDD, devuelve el BDD representando la función $\neg f_M$.

  El resultado se obtiene intercambiando las etiquetas de los terminales 0 y 1, o bien redirigiendo cada transición hacia 0 a 1 y viceversa, dependiendo de la implementación.

- `restrict`(b, x, M)
  Donde $b \in \{0, 1\}$ y x es una variable booleana, computa el *cofactor* de f_M donde la variable x toma el valor de b. Consiste en redirigir todas las aristas que llegan a cada nodo n etiquetado con x a $lo(n)$ si $b = 0$ o a $hi(n)$ si $b = 1$. Y finalmente reduciendo el resultado.



(a) BDD B representando la función $(x_1 \vee x_2) \wedge x_3$ (b) `restrict`$(0, x_2, B)$ (c) resultado representando $(x_1 \wedge x_3)$

Figura 3.4: Algoritmo `restrict`

- `threshold`$(M, \bowtie, c)$
  Donde $\bowtie$ es un operador relacional ($<, >, \geq, \leq$, etc.) y $c \in \mathbb{R}$, devuelve el BDD M' representando la función $f_{M'}$ igual a 1 si $f_M \bowtie c$ y 0 caso contrario.

  La manera de implementar `threshold` consiste en redirigir cada arista hacia un nodo terminal a 1 si el valor del nodo satisface la comparación y en caso contrario, redirigirla a 0. Como esto generará muchos sub-MTBDD idénticos, el MTBDD resultante debe ser reducido.

- `abstract`(op, x, M)
  Donde op es una operación binaria conmutativa y asociativa sobre $\mathbb{R}$, devuelve el resultado de abstraer x de M aplicando op sobre todos los valores posibles de la variable. Es decir el MTBDD resultante representa la función $f_{M|_{x=0}}$ op $f_{M|_{x=1}}$.

  Esto se podría implementar utilizando `restrict` y `apply` como:
  `apply`$(\text{op}, $`restrict`$(0, x, M), $`restrict`$(1, x, M))$. Pero en este caso `apply` está operando sobre los 2 cofactores que son idénticos hasta el nivel del nodo x. Para mejorar la eficiencia del algoritmo directamente buscamos en M los nodos etiquetados con x y los reemplazamos con el resultado de llamar a `apply` con sus 2 hijos.

  La abstracción sobre múltiples variables se implementa realizando llamadas sucesivas para cada variable, y es NP-Completo [HR00]

- `thereExists`(x, M)
  Donde M es un BDD, es equivalente a `abstract`$(\vee, x, M)$.

- `forAll`(x, M)
  Donde M es un BDD, es equivalente a `abstract`$(\wedge, x, M)$.

- `replaceVars`$(M, \vec{x}, \vec{y})$
  Donde $\vec{y} = (y_1, \ldots, y_n)$, devuelve un MTBDD M' sobre las variables $\vec{y}$ tal que $f_{M'}(b_1, \ldots, b_n) = f_M(b_1, \ldots, b_n)$ para cualquier $(b_1, \ldots, b_n) \in \mathbb{B}^n$.

  Esto se puede implementar recorriendo la estructura del MTBDD intercambiando las etiquetas de cada nodo. El MTBDD resultante tiene un orden incompatible con el original, por lo que luego es necesario un algoritmo de reordenamiento de variables como el de [Som99].

3.3. Representación de vectores y matrices

Para nuestros propósitos en este trabajo estamos interesados en utilizar MTBDD para la representación de vectores y matrices. La razón de esto es que tanto las transiciones del modelo como los conjuntos de estados y otros datos necesarios para el model checking son representados en matrices y vectores. Hablaremos con más precisión de esto en los siguientes capítulos.

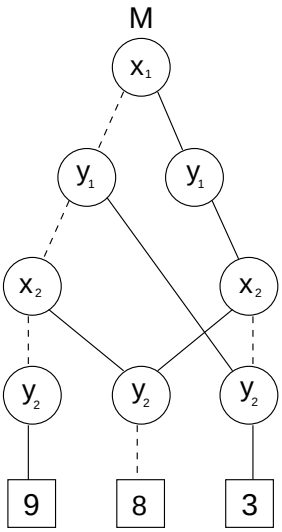
La representación de matrices y vectores utilizando MTBDD se introdujo en [CFM+93] [BFG+93].

Supongamos que $\vec{v}$ es un vector en $\mathbb{R}$ de largo 2^n. Podemos pensar a $\vec{v}$ como un mapeo de $I = \{0, \ldots, 2^n - 1\} \subset \mathbb{N}$ en $\mathbb{R}$, i.e. $\vec{v} : I \to \mathbb{R}$. Pero entonces, dada una codificación binaria de los 2^n elementos de I en n variables booleanas, i.e. una biyección enc : $I \to \mathbb{B}^n$, podemos representar $\vec{v}$ como un MTBDD V sobre las variables $(x_1, \ldots, x_n)$. Decimos que V representa $\vec{v}$ sii $f_V(\text{enc}(i)) = \vec{v}(i)$ para $0 < i < 2^n - 1$.

La misma idea se puede aplicar para representar matrices. Podemos pensar a una matriz $\mathbf{M}$ de $2^n \times 2^n$ como un mapeo de $I \times I$ en $\mathbb{R}$ donde $I = \{0, \ldots, 2^n - 1\}$. De nuevo si asumimos una codificación enc : $I \to \mathbb{B}^n$, podemos representar $\mathbf{M}$ con un MTBDD

$$\begin{pmatrix} 0 & 9 & 0 & 3 \\ 8 & 0 & 0 & 3 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 8 & 0 \end{pmatrix}$$

(a) matriz **M**



(b) MTBDD M

x_1	y_1	x_2	y_2	Matriz	f_M
0	0	0	1	(0,1)	9
0	0	1	0	(1,0)	8
0	1	0	1	(0,3)	3
0	1	1	1	(1,3)	3
1	1	1	0	(3,2)	8
1	1	0	1	(2,3)	3
otros casos				(_,_)	0

(c) coordenadas de la matriz y los valores correspondientes de las variables

Figura 3.5: Una Matriz **M** y el MTBDD M que la representa

M sombre $2n$ variables que codifican a n filas y n columnas. Entonces, decimos que M representa a **M** sii $f_M(\text{enc}(i), \text{enc}(j)) = \mathbf{M}(i, j)$ para $0 < i, j < 2^n - 1$.

Notar que también se pueden representar vectores y matrices con tamaños distintos a potencias de dos simplemente asignando 0 a las coordenadas extra de los MTBDD resultantes.

Podemos ver que en la figura 3.5(b) las variables de fila y columna están ordenadas de manera alternante. Esto es una heurística bien conocida [Par02] para minimizar el tamaño del MTBDD y se discutirá en el capítulo 4. Por claridad se omitieron las aristas hacia 0. Otra ventaja de este orden de variables es que permite una división de la matriz en sus submatrices componentes. Si la matriz **M** es representada por el MTBDD M, las mitades superior e inferior de **M** son representadas por los *cofactores* $M|_{x_1=0}$ y $M|_{x_1=1}$ respectivamente. Como x_1 es la primer variable en el orden, estos cofactores se pueden encontrar simplemente siguiendo sus aristas 1 y 0. Si repetimos este proceso en la siguiente variable, y_1, se obtienen los 4 cuadrantes de **M**. Además de proveer un acceso conveniente a las submatrices, esto es muy beneficioso para implementar operaciones sobre las matrices que pueden ser expresadas recursivamente. Esto se explica con más detalle en [BFG+93].

Capítulo 4

Model checking simbólico

En este capítulo se explican las etapas involucradas en el *model checking simbólico* de SNP sobre el que se basa este trabajo. El primer paso en este proceso es establecer una representación eficiente del modelo. En el capítulo 3 se menciona que las transiciones del modelo y otras estructuras de datos, son representado en matrices. A su vez estas estructuras se almacenan utilizando MTBDD y en algunos casos, cuando el desempeño lo amerite, también se almacenan datos de manera explícita. Debido a la sensibilidad de los MTBDD con respecto a sus variables, nos concentraremos en la manera de minimizar su tamaño. Luego hablaremos del problema de llevar a cabo el *model checking probabilista* sobre estas estructuras. En realidad, de los cómputos previos involucrados en el *model checking* luego de los cuales interviene el *método de reducción a estados esenciales* presentado en este trabajo, cuya implementación simbólica se detalla en el siguiente capítulo.

4.1. Representación y construcción de modelos con MTBDD

El tamaño de un MTBDD está definido por el número de nodos contenidos en la estructura de datos. Esto es muy importante porque afecta no sólo a la cantidad de memoria requerida para almacenamiento, sino que también al tiempo requerido para manipularlo. En general, el tiempo para realizar operaciones sobre el MTBDD es proporcional al número de nodos que contiene.

En varias oportunidades, mencionamos que los MTBDD son muy sensibles tanto al orden dado a sus variables booleanas como a la manera en que la función es codificada en esas variables. A continuación presentaremos las técnicas desarrolladas en [Par02] para atacar estos problemas.

Desde su concepción [CMZ+93] [CFM+93] [BFG+93], los MTBDD han sido usados para representar matrices. La idea general de este concepto fue explicada en la sección 3.3, allí los índices de las filas y columnas de las matrices son números enteros, los cuales al codificarlos en variables booleanas, nos permitían pensar en las matrices como funciones de booleanos a reales; que es exactamente lo que un MTBDD representa.

Lo que nos interesa ahora es poder representar la matriz de transiciones de un SNP. En este caso, los índices de la matriz son estados, y los valores son las probabilidades de las transiciones. De hecho, debido a la presencia del no determinismo la matriz a representar no es cuadrada, por lo que surge la necesidad de idear una manera para poder codificar también esta información.

Explicamos aquí la codificación utilizada en PRISM, la cual ha sido estudiada por varios autores y es en términos generales la más eficiente con respecto a la cantidad de nodos en el MTBDD que se obtiene. Remitirse a [Par02] para una explicación detallada de la heurística utilizada y comparaciones con otros esquemas.

Comenzamos con la manera de codificar los estados del SNP en variables booleanas, más adelante se mostrará como extender el esquema agregando una codificación para el no determinismo.

mdp

module $M1$
$v_1 :$ $[0..1]$ **init** 0;
$[]$ $(v_1 = 0)$ **&** $(v_2 = 2)$ $\rightarrow$ $(v_1' = 1)$;
$[]$ $(v_1 = 1)$ **&** $(v_2 = 2)$ $\rightarrow$ $(v_1' = 0)$;
endmodule

module $M2$
$v_2 :$ $[0..2]$ **init** 0;
$[]$ $(v_1 = 0)$ **&** $(v_2 = 0)$ $\rightarrow$ $0{,}4 : (v_2' = 1) + 0{,}6 : (v_2' = 2)$;
$[]$ $(v_1 = 0)$ **&** $(v_2 = 1)$ $\rightarrow$ $0{,}6 : (v_2' = 0) + 0{,}4 : (v_2' = 1)$;
endmodule

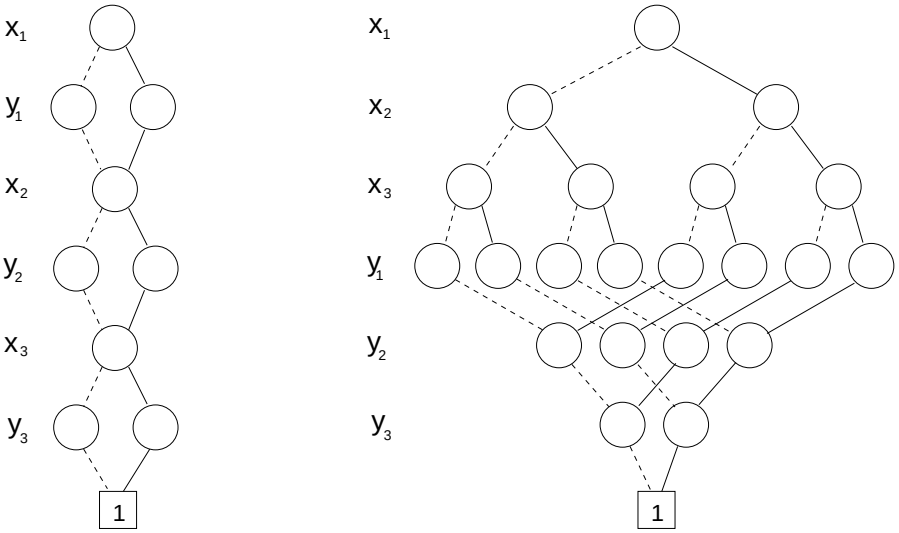
Figura 4.1: Un SNP descripto en el lenguaje de PRISM

4.1.1. Codificación de estados

Lo que haremos es mantener una correspondencia muy cercana entre las variables de PRSIM y las variables MTBDD, es decir, entre las variables del lenguaje de alto nivel de PRISM que da origen al modelo y la codificación de estas variables en las variables booleanas del MTBDD. Para lograr esto, codificamos cada variable PRISM con un conjunto propio de variables MTBDD. Para la codificación de cada una, utilizamos la representación binaria estándar de los enteros. Consideremos el modelo de la figura 4.1, más adelante explicaremos su interpretación. Por el momento notemos que este posee dos variables v_1 y v_2 con los rangos $\{0, 1\}$ y $\{0, 1, 2\}$ respectivamente. Nuestra codificación utiliza tres variables MTBDD, (x_1, x_2, x_3), con una para la variable v_1 y dos para la variable v_2, digamos x_1 para v_1; y x_2, x_3 para v_2. Entonces, por ejemplo, el estado $(1, 2)$ se codificaría como $(1, 1, 0)$ y el $(0, 1)$ como $(0, 0, 1)$. Notar que, como consecuencia de la codificación se introduce en el modelo una cantidad de estados extra no contemplados.

Esto puede ocurrir porque el rango de alguna variable no es potencia de 2 o simplemente porque el estado no es alcanzable. Por ejemplo, (0,1,1) codifica el estado (0,3) el cual no está en el rango de nuestras variables, y (1,0,1) codifica el estado (1,1) que si está en el rango, pero no está contemplado en el modelo. La solución consiste en dejar las filas y columnas de estos estados completamente en cero. De esta manera, los estados extra no interfieren ya que no tienen transiciones de ni entrantes ni salientes.

Luego de definir una codificación apropiada, pasamos a resolver el segundo problema: el orden de las variables. Obtener el orden óptimo es un problema NP-Completo [THY93], [BW96]. Por esto la manera tradicional para encontrar un buen orden se basa en análisis heurísticos. Cuando hablamos de representar matrices en MTBDD en la sección 3.3, mencionamos que intercalar los índices de fila y columna proveía un acceso conveniente a las sub-matrices, pero como puede verse en la figura 4.2 esto también reduce significativamente el tamaño del MTBDD.



(a) Intercalado (b) no intercalado

Figura 4.2: dos MTBDD representando la matriz identidad 8×8 con distinto orden en sus variables

Ejemplo 4.1. *En la figura 4.2 podemos ver 2 MTBDD representando la matriz identidad* 8×8. *En ambos casos se utilizan las variables* x_1, x_2, x_3 *para codificar las filas y las variables* y_1, y_2, y_3 *para codificar las columnas. En el MTBDD (a) las variables están intercaladas, y en el (b) se codifican primero las filas y luego las columnas. Por claridad, omitimos las transiciones hacia 0. En términos de MTBDD la función representada es* $(x_1 = y_1) \wedge (x_2 = y_2) \wedge (x_3 = y_3)$. *Consideremos que ocurre cuando recorremos el MTBDD buscando averiguar el valor de la función para alguna asignación de las variables: en el MTBDD (a), luego de dos niveles ya determinamos si* $x_1 = y_1$, *si es así continuamos descendiendo, si no, la siguiente transición nos lleva al nodo 0. De cualquier manera, a partir de este punto los valores de* x_1 *y* y_1 *son irrelevantes, ya*

que en esta función x_1 y y_1 sólo se relacionan entre sí y su relación ya fue evaluada. En cambio en el MTBDD (b), desde el momento que sabemos el valor de x_1 todavía debemos descender 3 niveles más antes de poder ***usar*** *el valor. Mientras tanto debemos ir* ***recordando*** *los valores de x_2 y x_3, causando un gran crecimiento en el número de nodos.*

En general el tamaño de un MTBDD para la matriz identidad $2^n \times 2^n$ es del orden $O(n)$ u $O(n^2)$ para los casos intercalados y no intercalados, respectivamente. Intercalar las variables de filas y columnas también es beneficioso para una matriz de transiciones ya que en una transición de estados típica, son pocas las variables que cambian y debido a la correspondencia entre las variables PRISM y las del MTBDD, esto también aplica al nivel de MTBDD. Es decir, en general, la variable y_i está estrechamente relacionada con la x_i.

Una vez elegido el esquema intercalado queda decidir la manera de agrupar las variables. Siguiendo con la idea de mantener la estructura del modelo, agrupamos juntas a las variables que representan la misma variable PRISM, a la vez también se supone que las variables PRISM pertenecientes a un mismo módulo están muy relacionadas. Por lo tanto, el orden adoptado consiste en agrupar las variables MTBDD por módulo y dentro de cada uno, por variable PRISM.

4.1.2. Codificación del no determinismo

Una matriz de transiciones que codifique el no determinismo se puede pensar como una matriz no cuadrada $m \times |S|$ donde m es la cantidad de decisiones no deterministas del todo el modelo y $|S|$ es la cantidad de estados. Esta matriz es similar a la discutida en la sección anterior, sólo que posee varias filas por estado. Podemos pensarla como una función de la forma $S \times \{1, \ldots, m\} \times S \to [0, 1]$. Cómo ya tenemos una codificación para los estados, si codificamos $\{1, \ldots, m\}$ en variables binarias tenemos nuevamente una función de números binarios en reales, por lo que podemos representarla completamente con MTBDD. Utilizaremos las variables $(z_1, \ldots, z_k)$ con $k \geq 0$ para codificar $\{1, \ldots, m\}$. Nuevamente, surge el inconveniente de que al haber estados con menos de m decisiones no deterministas estamos introduciendo índices extra que están indefinidos. Esta situación es análoga a la anterior y la solucionaremos de la misma manera, añadiendo distribuciones de probabilidad vacías.

Definición 4.1.1. *Sea M un MTBDD sobre los conjuntos de variables $\vec{x} = (x_1, \ldots x_n)$, $\vec{y} = (y_1, \ldots y_n)$, $\vec{z} = (z_1, \ldots z_k)$. Decimos que M representa la matriz de transición de un SNP Π si para alguna codificación de estados $enc : S \to \mathbb{B}^n$ y cualquier $s \in S$:*

- *si $\mathrm{a} \in \kappa(s)$, entonces existe $b \in \mathbb{B}^k$ tal que:*

  $$f_M(\vec{x} = enc(s), \vec{y} = enc(t), \vec{z} = b) = p(s, \mathrm{a}, t) \text{ para todo } t \in S$$

- *para cualquier $b \in \mathbb{B}^k$, se cumple una de las siguientes:*

  1. $f_M(\vec{x} = enc(s), \vec{y} = enc(t), \vec{z} = b) = 0$ *para todo $t \in S$*

2. *existe* $a \in \kappa(s)$ *tal que*
   $f_M(\vec{x} = enc(s), \vec{y} = enc(t), \vec{x} = b) = p(s, a, t)$ *para todo* $t \in S$

La primera parte asegura que toda acción de cada estado del SNP esté codificada en el MTBDD, y la segunda que cualquier acción codificada en el MTBDD pertenezca realmente al SNP.

Ahora nos hace falta definir efectivamente la codificación de las decisiones no deterministas en variables MTBDD. De la misma manera que al definir la codificación de los estados, intentaremos mantener la estructura original del modelo representado en el lenguaje de PRISM. A partir de esta decisión encontramos que el no determinismo se produce en dos situaciones: por la composición en paralelo de los módulos, y por la superposición de las guardas en las condiciones internas de cada módulo. En general, la mayor causa del no determinismo es producido por la composición de los módulos. Supongamos por ahora que es la única causa. Entonces, un MDP con m módulos, tiene exactamente m decisiones no deterministas en cada estado, cada una correspondiente al siguiente módulo a seleccionar. Continuando con el razonamiento de ubicar cerca las variables que están relacionadas entre sí, queremos ubicar cada variable no determinista cerca del módulo que representa. Por lo tanto, utilizaremos una codificación *estructurada*: una variable de MTBDD para cada decisión no determinista. Esta codificación puede no parecer óptima ya que introduce una variable MTBDD por cada módulo, pero el hecho de mantener cada variable cerca de las variables de su módulo reduce considerablemente el tamaño del MTBDD final.

Para un sistema con m módulos, cuyo espacio de estados para cada módulo i es codificado en n_i variables para las filas $(x_1^i, \ldots, x_{n_i}^i)$ y n_i variables para las columnas $(y_1^i, \ldots, y_{n_i}^i)$. Nuestro orden queda estructurado de la siguiente manera:

$$(z^1, x_1^1, y_1^1, \ldots, x_{n_1}^1, y_{n_1}^1), \ldots, (z^m, x_1^m, y_1^m, \ldots, x_{n_m}^m, y_{n_m}^m)$$

Para el caso del no determinismo interno a cada módulo no hay una estructura particular. Pero como suponemos que esas variables están relacionadas con el módulo al que pertenecen, las codificamos de manera no estructurada junto a las otras variables de ese módulo.

4.1.3. Construcción del MTBDD

Ahora explicaremos brevemente el proceso con el que se obtiene un MTBDD representando la matriz de transiciones de un SNP, el cuál está expresado en el lenguaje de PRISM. Veremos que la traducción es bastante directa debido a que el lenguaje fue pensado con este propósito.

Ilustramos la idea informalmente a través de un ejemplo, pero antes explicamos los conceptos básicos necesarios para interpretar el código PRISM de la figura 4.1. Como vimos anteriormente cuando hablamos de la representación de los estados, este está conformado por dos módulos M_1 y M_2. La declaración de un módulo consiste en:

module *nombre* ... **endmodule**

Los elementos fundamentales de un módulo son las *variables locales* y los *comandos*. Las variables definen el espacio de estados del modelo. En PRISM se aceptan varias maneras de declarar sus rangos, tipos y valores iniciales, en nuestro caso utilizamos la siguiente:

$$nombre : [rango] \textbf{ init } valor$$

En nuestro código PRISM, declaramos las variables v_1 y v_2, una en cada módulo, con los rangos $\{0, 1\}$ y $\{0, 1, 2\}$ respectivamente, y ambas inicializadas en 0. Las otras declaraciones que encontramos son los *comandos*, los cuales establecen las transiciones del modelo. Su declaración consiste en:

$$[]\ guarda -> prob_1 : modificación_1 + \cdots + prob_n : modificación_n$$

La *guarda* es un predicado sobre todas las variables del modelo (incluidas las de otros módulos). Cada *actualización* describe una transición que puede hacer el módulo si la *guarda* es verdadera. Una transición es especificada asignando nuevos valores a las variables del módulo, posiblemente como funciones de otras variables. Además a cada *actualización* se le asigna una probabilidad la cual será asignada a la transición correspondiente. Notar como en las *actualizaciones* las variables están primadas.

En nuestro código, el espacio de estados es $\{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)\}$. Estos son codificados utilizando tres variables MTBDD para las filas (x_1, x_2, x_3) y tres para las columnas (y_1, y_2, y_3).

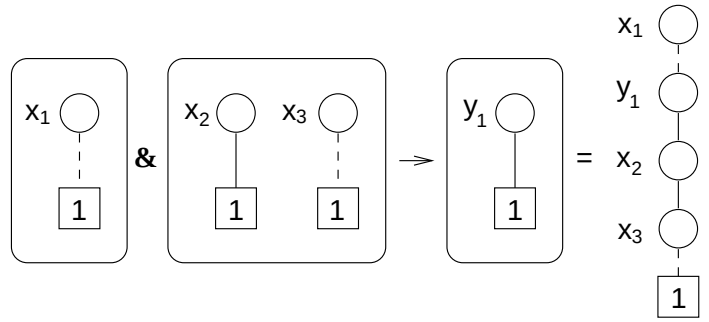
Una vez codificados los estados, lo primero que se hace es construir un MTBDD para cada comando. Ya que cada comando modifica las variables *locales* de su módulo basado en el estado de algún subconjunto de todas las variables, el MTBDD de un comando, está definido sobre las variables de las filas de todo el sistema y las variables de columna de ese módulo.

Ejemplo 4.2. *En la figura 4.3 vemos la traducción de dos comandos del modelo en el lenguaje PRISM de la figura 4.1 en la página 29. Notar como las variables no primadas y las primadas son traducidas a las variables de fila y columna respectivamente.*

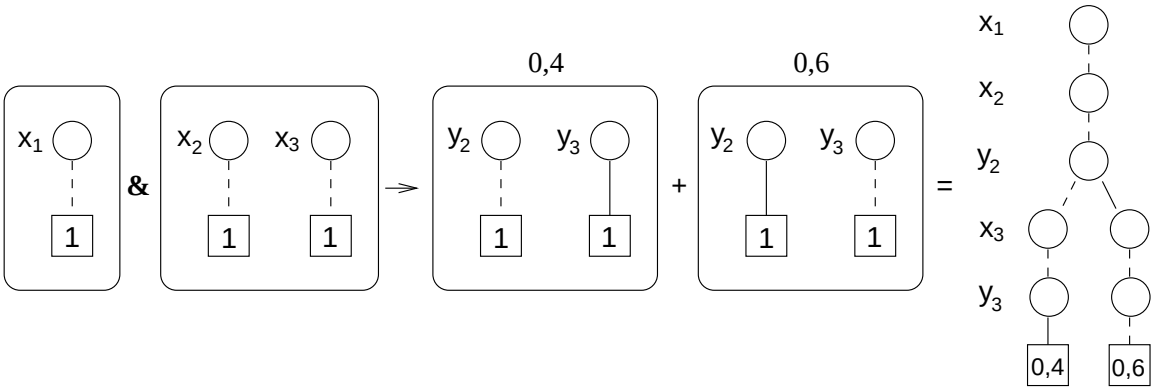
El MTBDD de un módulo se construye sumando los de todos sus comandos. Y el MTBDD representando todo el SNP es entonces obtenido combinando los MTBDD de todos sus módulos.

Finalmente, se aplica un algoritmo de alcanzabilidad, el cuál, puede efectuarse mediante una búsqueda en profundidad en el espacio de estados comenzando en el estado inicial. Notar que en el *model checking* no probabilista, determinar los estados alcanzables puede ser suficiente para obtener la solución. En nuestro caso, aún tenemos que realizar los cálculos de probabilidad sobre el modelo *alcanzable*. Por esta razón el cálculo de alcanzabilidad es considerado parte de la fase de construcción del modelo.

Otra observación interesante es que eliminar los estados no alcanzables del modelo causa un pequeño incremento en el tamaño del MTBDD. Más allá de que se eliminan estados y transiciones esto es causado porque se reduce la regularidad del modelo. Este



(a) $(v_1 = 0)$ & $(v_2 = 2) \rightarrow (v_1' = 1)$



(b) $(v_1 = 0)$ & $(v_2 = 0) \rightarrow 0{,}4 : (v_2' = 1) + 0{,}6 : (v_2' = 2)$

Figura 4.3: Traducción de comandos PRISM a MTBDD

problema, como veremos más adelante también ocurre al aplicar la *reducción a estados esenciales* objeto de este trabajo. Sin embargo, no tiene sentido dejarlos ya que la menor cantidad de estados disminuye considerablemente el trabajo de la etapa de cálculo numérico posterior.

4.2. Model checking probabilista sobre MTBDD

Luego de explicar la manera en que obtenemos la representación de un SNP en un MTBDD, pasamos al problema de efectuar el *model checking* sobre estas estructuras. En la sección 2.3 explicamos los fundamentos del model checking de propiedades PCTL sobre SNP y se planteó una manera algorítmica de aproximar la solución. A continuación explicamos la técnica desarrollada por [Par02] para aplicar esta solución al model checking simbólico. Nos concentraremos en las etapas previas al cálculo de $\mathcal{P}_{\bowtie p}[\phi_1 \,\mathcal{U}\, \phi_2]$, es decir justo luego de los precómputos y antes del cálculo numérico, ya que es donde nuestro método es aplicado.

El primer requerimiento de la implementación de los algoritmos de model checking es que tanto las entradas como las salidas estén codificadas en MTBDD. Como vimos con anterioridad, la entrada de los algoritmos consiste en un modelo y una fórmula a verificar

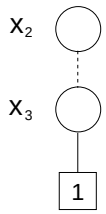
sobre este, y la salida es el conjunto de estados del modelo que satisface esa fórmula. La manera de representar el modelo con un MTBDD fue el objeto de las secciones anteriores de este capítulo. Por otro lado, podemos representar un conjunto de estados S' mediante un BDD de una manera muy simple, mediante su función característica, es decir, la función $f_{S'}$ tal que $f_{S'}(s) = 1$ si $s \in S'$ y 0 en caso contrario. Finalmente, nos queda por ver la manera en se le pasan las fórmulas al algoritmo.

Recordemos que el algoritmo de model checking de PCTL procede recorriendo el árbol de parseo de la fórmula y recursivamente va evaluando el conjunto de estados que satisface cada sub-fórmula. Por lo tanto, el algoritmo está compuesto por varios algoritmos. Uno para cada operador lógico. Como el algoritmo es recursivo, podemos asumir que las sub-fórmulas que componen la fórmula a verificar ya han sido evaluadas. Por ejemplo al computar $Sat(\neg\phi)$, podemos asumir que el resultado de $Sat(\phi)$ es conocido. Entonces, sólo necesitamos generar los MTBDD para representar las hojas de nuestro árbol, es decir, sólo necesitamos generar los conjuntos Sat para las proposiciones atómicas y $true$.

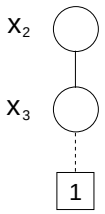
El cálculo de $Sat(\phi)$ para los operadores $(true, a, \wedge, \neg)$ es simple: $\wedge$ y $\neg$ se obtienen directamente con la operación sobre BDD `apply`; $true$ es el BDD con sólo el 1, es decir `const`(1); y finalmente para una proposición atómica a definimos la función `encode`(a) que devuelve el BDD representando los estados que satisfacen a. Esto es muy fácil de obtener ya que las proposiciones atómicas son predicados sobre las variables de PRISM que a su vez tienen una correspondencia directa con las variables del MTBDD que representa el modelo.

v_2	x_2	x_3
0	0	0
1	0	1
2	1	0

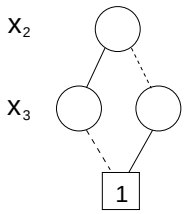
(a) codificación de v_2.



(b) `encode`($v_2 = 1$)



(c) `encode`($v_2 = 2$)



(d) `encode`($v_2 > 0$)

Figura 4.4: Codificación de proposiciones atómicas en BDD

Ejemplo 4.3. *Consideremos nuevamente el código de PRSIM de la figura 4.1 en la página 29, y supongamos que queremos verificar la propiedad* $(v_2 = 1 \vee v_2 = 2)$. *Para obtener el BDD que representa* $Sat(v_2 = 1 \vee v_2 = 2)$ *el algoritmo calcula* $\mathbf{apply}(\vee, Sat(v_2 = 1), Sat(v_2 = 2))$. *Por lo que primero debe calcular* $Sat(v_2 = 1) = \mathbf{encode}(v_2 = 1)$ *y* $Sat(v_2 = 2) = \mathbf{encode}(v_2 = 2)$ *cuyos BDD podemos en ver en las figuras 4.4(b) y 4.4(c), respectivamente. Finalmente, en la figura 4.4(d) vemos el resultado final, el cual es equivalente a* $\mathbf{encode}(v_2 > 0)$.

Algo importante a destacar es que el haber eliminado los estados no alcanzables del modelo, no es suficiente para evitar que reaparezcan en los BDD generados por los algoritmos de *model checking*. Por lo que en algunas situaciones es necesario volver a eliminarlos

para que no sean considerados en las recursiones siguientes. Tomemos el caso de la operación $Sat(\neg\phi)$. Supongamos que el espacio de estados es $\{(0,0),(0,1),(1,0),(1,1)\}$ pero los estados alcanzables son $\{(0,0),(0,1),(1,0)\}$. Entonces, si $Sat(\phi) = \{(0,0),(0,1)\}$, al calcular $Sat(\neg\phi)$ obtenemos $\{(1,0),(1,1)\}$. Para evitar esto, en realidad se calcula como $reach \wedge \neg Sat(\phi)$, donde $reach$ es el BDD representando el conjunto de estados alcanzables.

Finalmente, nos queda por considerar el operador probabilista $\mathcal{P}_{\bowtie p}$. El *model checking* de este operador es la parte más significativa de todo el algoritmo. A grandes rasgos, el procedimiento consiste en obtener un vector de probabilidades p_s^{max} o p_s^{min} de cada estado y luego comparar las probabilidades con la cota $\bowtie p$ para determinar que estados satisfacen la fórmula.

Como vimos en la sección 2.3, antes de considerar la necesidad del cálculo numérico para obtener la solución, efectuamos una serie de precómputos. En estos se determina mediante un análisis del grafo del modelo, los estados para los que la probabilidad en cuestión es exactamente 0 o 1. Además del ahorro en el costoso cálculo posterior, las probabilidades obtenidas por estos algoritmos no sufren de errores de redondeo presentes en el cálculo numérico.

Los algoritmos utilizados son cuatro, dos previos al cálculo de la probabilidad máxima y otros dos para el caso de la probabilidad mínima. Nos concentraremos en el funcionamiento de `prob0E` utilizado para calcular S^{no} en el caso de la *probabilidad mínima*, ya que es suficiente para explicar los detalles de la implementación.

Algoritmo 4.2.1 prob0E(phi_1, phi_2, T)

```
1: sol := phi_2
2: fin := false
3: while (fin = false) do
4:    tmp := thereExists(y, T ∧ replaceVars(sol, x, y))
5:    sol' := sol ∨ (phi_1 ∧ forAll(z, tmp))
6:    if (sol' = sol) then
7:       fin := true
8:    end if
9: end while
10: return not(sol)
```

`prob0E`, detallado en el algoritmo 4.2.1, obtiene todos los estados con probabilidad 0 para alguna *estrategia*, por lo que es utilizado al durante el cálculo de $p_s^{min}(\phi_1 \,\mathcal{U}\, \phi_2)$. Al igual que para el cálculo de los otros operadores, se asume que los conjuntos de estados $Sat(\phi_1)$ y $Sat(\phi_2)$ ya fueron generados. Estos son representados por los BDD sobre las variables $\vec{x}$, phi_1 y phi_2. El algoritmo sólo considera las transiciones entre estados y no la probabilidad de hacer la transición. Por lo que es más eficiente utilizar un BDD T que representa únicamente la relación de transiciones del modelo y no un MTBDD M representando la matriz con las probabilidades de las transiciones. Este se puede calcular como $T =$ `threshold`$(M, >, 0)$.

Como todos los algoritmos de precómputo, `prob0E` efectúa un cálculo de punto fijo para obtener los resultados. En este caso, el algoritmo obtiene el conjunto con los estados tales que, para todas las *estrategias*, existe una *ejecución* a partir de los mismos, que tiene una probabilidad mayor a cero de alcanzar algún estado en phi_2 sin haber dejado estados en phi_1. Luego, devuelve la negación de ese conjunto, es decir, los estados con probabilidad 0 para alguna *estrategia*.

Entonces, al comenzar el conjunto *sol* que contendrá el resultado del algoritmo, es inicializado a phi_2. Luego, en cada iteración, todos los estados que tienen una transición a estados en *sol* para todas sus acciones, se agregan a *sol*. Si durante una iteración no se agrega ningún nuevo estado a *sol*, el algoritmo finaliza y devuelve $\neg sol$. El algoritmo termina debido a que la cantidad de estados es finita.

Detallamos ahora la interpretación de las líneas 4 y 5 del algoritmo: Recordemos que $\vec{x}$, $\vec{y}$ y $\vec{z}$ son los vectores de variables de fila, columna y no determinismo, respectivamente. Entonces, comenzando por la línea 4, $\texttt{replaceVars}(sol, \vec{x}, \vec{y})$ devuelve el conjunto que representa los mismos estados que *sol*, pero sobre las variables de columna. De esta manera $(T \wedge \texttt{replaceVars}(sol, \vec{x}, \vec{y}))$ retorna la matriz de transiciones T, pero sólo con los estados que tienen transiciones a algún estado en *sol*. Finalmente $tmp := \texttt{thereExists}(\vec{y}, T \wedge \texttt{replaceVars}(sol, \vec{x}, \vec{y}))$ abstrae de T todas las variables en $\vec{y}$ dejando en *tmp* un vector en $\vec{x}$ y $\vec{z}$ que representa todos los estados con transiciones a estados en *sol* y además mantiene la información de con qué acción se efectúa la transición. En la línea 5 con $\texttt{forAll}(\vec{z}, tmp)$ se abstraen de *tmp* las variables en $\vec{z}$ dejando un vector únicamente en $\vec{x}$ tal que representa los estados que tienen transiciones a estados en *sol* para todas sus acciones. Al hacer $(phi_1 \wedge \texttt{forAll}(\vec{z}, tmp))$ reducimos el vector en $\vec{x}$ eliminando todos los estados en los que no vale phi_1. Finalmente, con $sol' := sol \vee (phi_1 \wedge \texttt{forAll}(\vec{z}, tmp))$ dejamos en sol', los estados *sol* más el conjunto recién calculado.

Al finalizar los algoritmos de precómputo, verificamos si la suma de los conjuntos de estados obtenidos por ambos algoritmos es equivalente al conjunto total de estados, es decir si $S^{yes} \cup S^{no} = S$. De ser así, la solución del algoritmo es S^{yes}, en caso contrario, continuamos con el *método de reducción a estados esenciales* para minimizar la cantidad de estados, cuya probabilidad debe ser calculada mediante métodos numéricos. La implementación de este método se explica con detalle en el siguiente capítulo.

Aunque no profundicemos en la implementación de los cálculos numéricos, destacamos la diferencia fundamental entre el cálculo MTBDD *puro* y el *híbrido*. Como mencionamos en la sección 2.3, el cálculo numérico involucrado en la obtención de las probabilidades de una propiedad *Until*, consiste fundamentalmente una serie de multiplicaciones matriz-vector donde la matriz permanece constante, pero el vector es modificado en cada iteración. En el caso MTBDD puro, ambas estructuras de datos están representadas con MTBDD y durante esta serie de multiplicaciones ocurre que la pérdida de regularidad del vector causa un aumento en su tamaño que en muchos casos, hace que realizar el cálculo sea imposible. Esta limitación fomentó la creación del método *híbrido* donde la matriz sigue siendo almacenada utilizando MTBDD y, por lo tanto aprovechando el ahorro de memoria que esto produce, pero el vector es almacenado de manera explícita

en un arreglo. El tamaño inicial del vector es mucho mayor a su contraparte MTBDD, pero, su tamaño es constante logrando un gran aumento en la eficiencia de los cálculos. Para más detalles, consultar [Par02].

Capítulo 5

Reducción a estados esenciales

En este capítulo, detallamos los algoritmos utilizados en la implementación simbólica del *método de reducción a estados esenciales* introducido en la sección 2.4. Como estuvimos viendo en el capítulo anterior, este método forma parte del *model checking* de las probabilidades $p_s^{min}(\phi_1 \mathcal{U}\, \phi_2)$ y $p_s^{max}(\phi_1 \mathcal{U}\, \phi_2)$. Específicamente, se ocupa de realizar modificaciones sobre el modelo, previo al cálculo numérico de las probabilidades.

Para efectuar la reducción necesitamos la matriz de transición T del SNP, y los conjuntos de estados S^{yes}, S^{no} y $S^{?}$ obtenidos como se explica en las secciones 2.3 y 4.2. Recordemos que los algoritmos de precómputo obtienen todos los estados con las probabilidades exactas 0 o 1 por lo que S^{yes}, S^{no} se ven incrementados y $S^{?}$ reducido; pero no efectúan cambios sobre la matriz de transiciones. Por el contrario, nuestro método sí elimina estados y transiciones de la matriz de transiciones. Los algoritmos de precómputo son fundamentales para el correcto funcionamiento de nuestro método de reducción, más adelante cuando detallemos la solución, explicaremos la razón de esto.

El algoritmo de reducción de estados se divide en dos partes: Calcular los *estados esenciales* y modificar el modelo apropiadamente. Para calcular los *estados esenciales*, se separan los estados de $S^{?}$ en las clases de equivalencia inducidas por $\sim$ (definición 2.4.4), cada una correspondiente a cada uno de los *estados esenciales*. Esto se efectúa mediante un algoritmo de punto fijo sobre los estados que aplica una variante del método de *union-find*. La diferencia radica en que en nuestro método las clases de equivalencia contienen un elemento especial que debe ser diferenciado: el *estado esencial*. Notar que los estados en S^{yes} y S^{no} son siempre *estados esenciales*. Luego se realizan las modificaciones al modelo, que consisten en eliminar los estados no esenciales de la matriz de transiciones junto con sus transiciones salientes y redirigir las transiciones que iban hacia esos estados hacia el *estado esencial* correspondiente. En algunas situaciones varias transiciones deben ser reemplazadas por una, en este caso, la probabilidad de la nueva transición es la suma de las probabilidades de las anteriores.

5.1. Algoritmos

A continuación explicamos los fundamentos que nos conducirán al algoritmo que obtiene nuestra reducción y a su implementación. Primero vemos cómo calcular los estados esenciales y luego cómo modificar el SNP.

Supongamos un SNP $\Pi = (PA, S, A, V, \kappa, p, s_{in})$. Durante el resto del capítulo asumimos que $S^?$, S^{yes} y S^{no} son los conjunto de estados obtenidos por los algoritmos de precómputo durante el *model checking* de $(\phi_1 \,\mathcal{U}\, \phi_2)$. Sea $\mathcal{E}$ la partición de S asociada a $\sim$ (ver definición 2.4.2), y sea $E = (S/\mathcal{E})$ el conjunto de estados esenciales. Para cualquier $s \in S$, denotemos con $[s]$ la clase de equivalencia de s asociada a la partición $\mathcal{E}$.

5.1.1. Cálculo de los estados esenciales

Para determinar los *estados esenciales* de Π y los estados dominados correspondientes a cada uno, necesitamos ir aproximando la solución de una manera iterativa que obtenga la partición $\mathcal{E}$ de S y mientras tanto, mantenga la información de cuál es el *estado esencial de cada clase*.

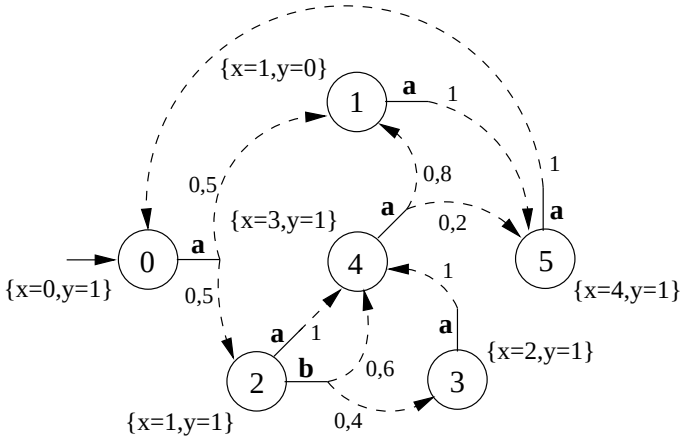


Figura 5.1: SNP Π

El algoritmo *union find* opera de la siguiente manera: Al inicio, todos lo elementos pertenecen a su propia clase. Luego, dados 2 elementos s y t verifica si se cumple que $s \sim t$ (*find*) y de ser así, une los conjuntos $[s]$ y $[t]$ (*union*) y procede con otros 2 elementos hasta que todos hayan sido comparados. Una de las implementaciónes más eficientes es llamada *union find with path compression*, en esta se utiliza un arreglo de enteros indexado por una codificación numérica de los elementos y en donde cada valor es -1 o la codificación de un elemento de su clase de equivalencia, lo que significa que el elemento del índice apunta al elemento de su valor. En esa implementación, todos los elementos de una clase apuntan a un único elemento, de esta manera se evita realizar muchos saltos durante la operación *find*. En la figura 5.2, podemos ver un ejemplo de vector en el que los elementos 2,3 y 4 pertenecen a la misma clase de equivalencia.

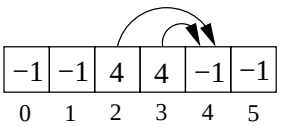


Figura 5.2: *Union Find with path compression*

Podríamos utilizar esta estructura para implementar nuestro método, teniendo la precaución de mantener el *estado esencial* de cada clase en otra estructura de datos; o podríamos forzar a que el elemento apuntado por todos lo elementos de una clase, sea el estado esencial. Pero mantener un vector de posiblemente millones de estados va en contra de la idea del ahorro de memoria del *model checking simbólico*, además un vector de estados en estados no tiene sentido sobre MTBDD, por lo que optamos utilizar una matriz booleana EC indexada por estados. Intuitivamente, la matriz es tal que si $s, e \in S$ y $EC(s, e) = 1$, entonces s es de la clase de e, y e es un *estado esencial*. De esta manera mantenemos toda la información necesaria en una sola estructura de datos. Esta matriz se almacena eficientemente en un BDD y cómo veremos más adelante, una vez generada simplifica mucho la implementación de la segunda parte del algoritmo encargada de modificar la matriz de transiciones del SNP.

Matemáticamente, la función representada por la matriz EC puede pensarse como la función $ec : S \to E$ tal que para todo $s \in S$, $ec(s) = e$ indica que e es el *estado esencial* de la clase $[s]$. Veamos como obtener ec mediante un cálculo de punto fijo.

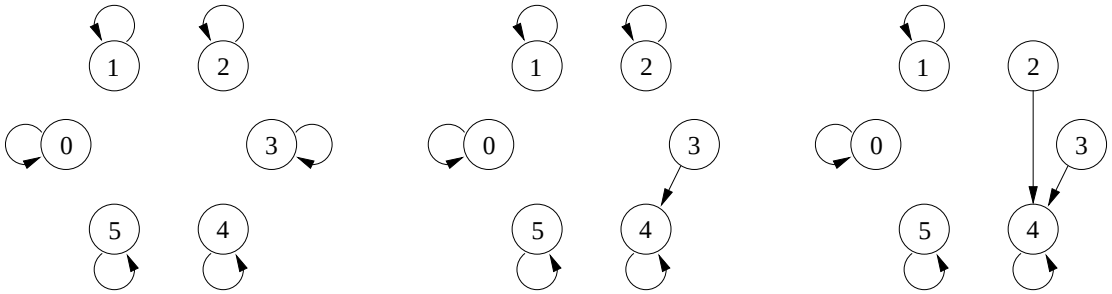
Inicialmente, para todo $s \in S$ tenemos que $ec^0(s) = s$. Sabemos que los estados $t \in (S^{yes} \cup S^{no})$ son *estados esenciales* por lo que permanecen en "sus" clases i.e. $ec^n(t) = t$ para $n \geq 0$. Entonces, para todo $s \in S^?$ calculamos:

$$ec^{n+1}(s) = \begin{cases} s' & si \quad \exists s' \cdot succ(s) \subseteq (ec^n)^{-1}(s') \\ s & c.c. \end{cases}$$

Finalizando cuando $ec^{n+1}(s) = ec^n(s)$ para todo s.

Entonces, el algoritmo que implemente este cálculo en cada iteración compara cada estado $s \in S$ con los pertenecientes a $succ(s)$ y en caso de que todos sus sucesores sean de una misma clase, hacemos a s de esa clase. Notar que una vez que hacemos a $ec(s)$ apuntar a otro estado distinto de s, s es eliminado de la imagen de ec. Para garantizar que el algoritmo termina tenemos que asegurarnos que no ocurra que un estado cambie de clase cíclicamente: Si un estado cambia de clase y no repite la misma clase nunca, entonces en algún momento deja de cambiar ya que la cantidad de clases posibles es finita. Entonces, para que el algoritmo no termine, debe haber al menos un estado r que cambia de clase de equivalencia continuamente y eventualmente repita una clase c a la que perteneció antes. Supongamos que esto ocurre, entonces, sea $(c, c_1, \ldots, c_n, c)$ con $c, c_i \in S^?$ un fragmento de la sucesión de clases a las que perteneció r, desde la iteración k a la iteración $k + n + 1$. Escribimos $[s]^n$ para denotar $(ec^n)^{-1}(s)$. Tenemos que:
$succ(r) \subseteq [c]^k$.
$succ(r) \subseteq [c_1]^{k+1}$ lo cual implica que $succ(c) \subseteq [c_1]^k$.

$succ(r) \subseteq [c_2]^{k+2}$ lo cual implica que $succ(c_1) \subseteq [c_2]^k$.
Y así sucesivamente. Por lo tanto, los estados $(c, c_1, \ldots, c_n, c)$ forman un ciclo sin transiciones salientes, pero los estados que se encontraban en $S^?$ y que no alcanzaban estados en S^{yes} fueron eliminados por los algoritmos de precómputo. Entonces, no existen estados que cambien de clase cíclicamente por lo que el algoritmo termina.



(a) Situación inicial: cada estado pertenece a su propia clase

(b) El estado 3 pasa a pertenecer a la clase del estado 4

(c) Situación final: los estados 2, 3 y 4 son de la clase del estado 4

Figura 5.3: Clases de equivalencia y estados esenciales del SNP

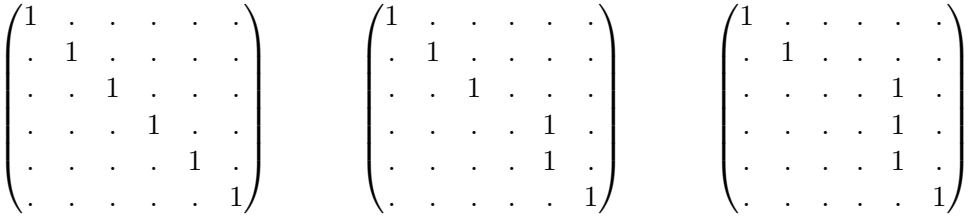
Ejemplo 5.1. *Tomemos el SNP de la figura 5.1. Supongamos que queremos verificar la propiedad* $\mathcal{P}_{>0,2}[(y = 1)\ \mathcal{U}\ (x = 4)]$. *Los algoritmos de precómputo nos devuelven los siguientes conjuntos:* $S^{yes} = \{5\}$, $S^{no} = \{1\}$ *y* $S^? = \{0, 2, 3, 4\}$ *por lo que podemos aplicar el método de reducción. En la figura 5.3 podemos ver una representación del proceso de separación de los estados en clases de equivalencia. Los estados conectados con flechas pertenecen a la misma clase, y el estado esencial de la clase es el único estado al que apuntan las flechas. En la figura 5.3(a) vemos el estado inicial en donde cada estado pertenece a su clase. En la siguiente iteración, figura 5.3(b), vemos que el estado 3 pasa a ser de la clase del estado 4. Finalmente vemos en la figura 5.3(c) el resultado del algoritmo.*
Entonces, los ***estados esenciales*** *son* $0, 1, 4$ *y* 5 *ya que* $ec(0) = 0$, $ec(1) = 1$, $ec(2) = 4$, $ec(3) = 4$, $ec(4) = 4$ *y* $ec(5) = 5$*; o visto de otra manera:* $ec^{-1}(0) = \{0\}$, $ec^{-1}(1) = \{1\}$, $ec^{-1}(2) = \emptyset$, $ec^{-1}(3) = \emptyset$, $ec^{-1}(4) = \{2, 3, 4\}$ *y* $ec^{-1}(5) = \{5\}$.
Cómo las probabilidades de los estados 1 *y* 5 *son conocidas, durante el cálculo numérico sólo se necesita obtener las de los estados* 0 *y* 4.

Tomando como base lo expuesto más arriba procedemos a caracterizar la matriz con la que implementaremos nuestro método.

Definición 5.1.1. *Sea* T *la matriz de transición del SNP* Π*, definimos la* ***matriz de clases de equivalencia*** *de* T *como una matriz booleana* $|S| \times |S|$ *indexada por estados tal que:*

- $(\forall s'\ \exists s \cdot (s, s') = 1) \ \Rightarrow\ (s', s') = 1$
- $\forall s, s'\ \big((s, s') = 1 \ \ \Rightarrow\ \ (\forall t \neq s' \cdot (s, t) = 0)\big)$

Lo primero asegura que cada *estado esencial* pertenece a su propia clase, notar que un estado es esencial cuando hay al menos un elemento en su columna. La segunda parte establece que cada estado pertenece a una única clase, esto es, que haya un sólo 1 por fila.



(a) Estado inicial: todos los estados pertenecen a su propia clase

(b) El estado 3 pasa a la clase del 4

(c) Estado final: El estado 2 pasa a la clase del 4

Figura 5.4: Progreso de la matriz de clases de equivalencia

Ejemplo 5.2. *En la figura 5.4 vemos 3 situaciones de la* ***matriz de clases de equivalencia*** *replicando las 3 etapas de las clases de equivalencia mostradas en la figura 5.3. (por claridad los* 0 *son remplazados por puntos).*

5.1.2. Modificación del SNP

Una vez obtenido el conjunto de clases de equivalencia de S generamos el nuevo SNP *esencial* $\Pi_{\mathcal{E}}$ (ver definición 2.4.5). Nuestro interés está centrado en su matriz de transiciones, la cual es una representación de la función p (ver definición 4.1.1). Entonces, sea ec la función de clases obtenida como se describe más arriba, generamos la función $p_{\mathcal{E}}$ de la siguiente manera:

Sea E el conjunto de *estados esenciales* de Π. Para todo $s, t \in E$ tenemos:

$$p_{\mathcal{E}}(s, a, t) = \sum_{t' \in ec^{-1}(t)} p(s, a, t') \tag{5.1}$$

Con esta información podemos obtener la matriz de transiciones de $|E| \times |E|$ de nuestro SNP $\Pi_{\mathcal{E}}$, es decir la nueva matriz de transiciones donde los estados no esenciales fueron eliminados y las transiciones apropiadamente redirigidas. En lugar de generar nuestra nueva matriz $T_{\mathcal{E}}$, lo que haremos es modificar la matriz de transiciones T original. Primero eliminamos las transiciones salientes de los estados no esenciales, es decir ponemos en cero todos los valores de las filas de estos estados. Llamemos T' a esta matriz, luego redireccionamos las transiciones que parten de un estado esencial a uno no esencial hacia el estado esencial correspondiente. Para esto simplemente multiplicamos la matriz T' por la matriz de clases de equivalencia EC. i.e $T_{\mathcal{E}} = T' \cdot EC$.

5.2. Reducción a estados esenciales con MTBDD

Pasamos ahora al algoritmo simbólico propiamente dicho. Al igual que en la sección anterior, separamos el algoritmo en dos partes: Búsqueda de estados esenciales y creación de la matriz de transiciones esencial. Estas dos etapas se detallan respectivamente en los algoritmos 5.2.1 y 5.2.2. La *matriz de clases EC* se representa en un BDD utilizando la misma codificación de estados de la matriz transiciones, presentada en la sección 4.1.1. Para efectuar el cálculo, el algoritmo necesita el BDD que representa la matriz de transiciones del SNP donde las probabilidades han sido abstraídas. Este BDD se obtiene de la siguiente manera: sea T la matriz de transiciones, entonces $T_{01} = \texttt{threshold}(T, >, 0)$ es la matriz representando las transiciones.

Algoritmo 5.2.1 findEssential($S^?, T_{01}$)

1: $C = \texttt{Identity}(\vec{x}, \vec{y})$
2: $T^?_{01} = \texttt{ThereExists}(\vec{z}, T_{01} \wedge S^?)$
3: *fin* := **false**
4: **while** (*fin* = **false**) **do**
5: $A := \texttt{BoolMatrixMultiply}(T^?_{01}, C)$
6: *bien* := $\texttt{ClassCheck}(A)$
7: $C' := (bien \wedge A) \wedge (\texttt{Not}(bien) \wedge C)$
8: **if** $(C' = C)$ **then**
9: *fin* := **true**
10: **end if**
11: $C := C'$
12: **end while**
13: **return** C

El algoritmo `findEssential` comienza asignando a C (la matriz de clases) la identidad $|S| \times |S|$. En la línea 2, $(T_{01} \wedge S^?)$ resulta en la matriz T_{01} pero sin las filas de los estados fuera de $S^?$, es decir, se eliminan sus transiciones salientes. Sobre ese resultado se abstrae el no determinismo, unificando todas las transiciones salientes de cada estado en una sola fila, dejando en $T^?_{01}$ una matriz con una sola fila por estado. En la línea 5 se efectúa la multiplicación booleana entre $T^?_{01}$ y C dando como resultado la matriz A tal que para todo $s, s' \in S^?$, $A(s, s') = \bigvee_{s'' \in S} \left(T^?_{01}(s, s'') \wedge C(s'', s')\right)$. Esta matriz es un paso intermedio y se utiliza para averiguar para cada estado, si todas sus transiciones llevan a estados de la misma clase. Esto es indicado cuando en la fila correspondiente a un estado hay exactamente un solo 1. En la línea 6, a partir de A se crea un conjunto de estados *bien* que contiene las filas de A que posean un único 1. Finalmente, en la línea 7 utilizamos *bien* como una mascara para generar nuestra C' con las filas *bien* de A y el resto, tal cual como estaban en C. El algoritmo termina cuando se haya alcanzado el punto fijo de C, es decir cuando $C' = C$.

Haciendo uso de la información en C, el algoritmo `makeEssential` genera la matriz de transiciones esencial $T_{\mathcal{E}}$ a partir matriz de transiciones original T.

Algoritmo 5.2.2 makeEssential(T, C)

1: $esenciales :=$ `ThereExists`$(\vec{y}, C)$
2: $A := (T \wedge$ `SwapVars`$(\vec{x}, \vec{y}, esenciales))$
3: $T_{\mathcal{E}} :=$ `MatrixMultiply`(A, C)
4: **return** $T_{\mathcal{E}}$

En la línea 1, con `ThereExists`$(\vec{y}, C)$ se abstraen de C todas las variables de las filas, dejando en *esenciales* un vector sobre $\vec{x}$ únicamente con los *estados esenciales*. En la fila 2, con `SwapVars`$(\vec{x}, \vec{y}, esenciales)$ nuestro *esenciales* pasa a estar en términos de las variables $\vec{y}$. Por lo que luego, al aplicar la disyunción con T, deja en A la matriz T sin las filas correspondientes a estados no esenciales, i.e. sin transiciones salientes. Finalmente sólo queda aplicar la sumatoria de la ecuación 5.1. En la línea 3 se efectúa la multiplicación de la matriz A por la C, dejando como resultado en $T_{\mathcal{E}}$, la matriz esencial completa.

Como ocurre al eliminar los estados no alcanzables, quitar los estados no esenciales disminuye la regularidad del MTBDD que representa el SNP. En general, esto produce un incremento en la cantidad de nodos, aumentando tanto el uso de memoria, como el costo de las operaciones. Afortunadamente, la cantidad de iteraciones necesarias durante el cálculo numérico se ve también reducida. Esto, sumado a la demora que introduce nuestro algoritmo de reducción, son las cuestiones que definen la mejora introducida en el model checker.

Se espera que nuestro método tenga un buen desempeño al aplicarse durante la verificación de modelos con mucho costo numérico. Además, ya que nuestro método ataca a cadenas de estados, cuya transición de uno a otro se efectúa con probabilidad 1, se espera que se produzcan grandes reducciones en los casos: que se modele tiempo (transiciones que representen *ticks*) o, en donde el no determinismo produzca islas de estados muy interconectados ente sí, pero con pocas transiciones salientes.

Notar que al haber eliminado estados durante la reducción, se pudieron haber eliminado estados con probabilidad mayor a cero, es más, si el estado inicial no es esencial, también fue eliminado. Afortunadamente, conocemos las probabilidades de los estados eliminados: la misma probabilidad que su estado esencial dominador. Por lo que, utilizando la misma matriz generada en el proceso iterativo, podemos reconstruir el vector de solución completo. A nivel implementación, cuando se efectúa una verificación utilizando el método *híbrido*, recuperar el vector de soluciones completo implica un gran costo. Esto es, al menos con el método que utilizamos para efectuar esta recuperación. Por esto se decidió recuperar sólo la probabilidad del *estado esencial* correspondiente al estado inicial del modelo, lo cual es relativamente instantáneo. La misma idea puede aplicarse al caso puro, pero la mejora sería ínfima.

Capítulo 6

Casos de estudio

A continuación analizamos el desempeño de la implementación de la herramienta. A partir de los resultados intentamos identificar con mayor precisión los casos donde su uso es beneficioso (tanto en uso de memoria como en costo computacional), contrastando la información obtenida del *model checking* con y sin la reducción de estados. Los cálculos fueron hechos sobe un procesador AMD Opteron(tm) Dual-Core 2010 Mhz. (4020.67 BogoMIPS) con 32 GB de RAM. La CPU está dedicada 100 % al proceso debido a que la PC cuenta con 4 procesadores.

6.1. Binary exponential backoff

El algoritmo *binary exponential backoff* es utilizado para resolver la contienda entre varios *hosts* comunicándose a través de un canal único. Ya que no puede haber dos *hosts* enviando datos por el canal al mismo tiempo, el algoritmo define los tiempos de espera y retransmisión, los cuales, están relacionados me manera exponencial binaria. Este método es utilizado en redes LAN como IEEE 802.3 (Ethernet) y redes HFC.

Podemos describir al algoritmo de la siguiente manera: Cuando un *host* intenta enviar un mensaje y ocurre un conflicto en el canal, el *host* selecciona de manera aleatoria un valor i de un intervalo dado, y antes de reintentar la comunicación, espera un tiempo $i \cdot \tau$. Donde τ es una constante fija relacionada al sistema. Si durante el reintento ocurre otro conflicto, se repite la operación, pero el valor se toma de un intervalo mayor.

En nuestro modelo, el protocolo elige $i \in [0, 2^j]$ de manera uniforme, donde j es el número de intentos o $2^j = K$ si la cantidad de reintentos es mayor que $log_2(K)$. Esto se repite hasta que se logra transmitir el mensaje o se alcanza el límite N de reintentos, momento en el cual la comunicación falla.

A continuación, vemos los resultados del *model checking* en donde una cantidad H de *hosts* intenta enviar un mensaje simultáneamente con un límite de reintentos N y una cota de intervalos $log_2(K)$.

Propiedad a verificar: $p^{min}[\mathbf{true}\,\mathcal{U}\,\mathbf{giveUp}]$
(probabilidad mínima de fallar)

6.1.1. MTBDD

Espacio										
H	N	K	MTBDD				MTBDD con reducción			
			Estados	Mat. it.	Mx. vec.	Mem.	Est. Ese.	Mat. it.	Mx. vec.	Mem.
			cant.	nodos	nodos	MB	cant.	nodos	nodos	MB
3	5	8	779094	98881	207052	5,84	734363	112238	165934	5,31
		16	3504254	222941	582364	15,36	3402860	252606	514458	14,63
		32	9490126	525011	1035783	29,77	9328024	572644	1000787	30,01
4	3	4	532326	69880	68640	2,64	508272	94608	66405	3,07
		8	3020342	163059	217151	7,25	2941604	210121	219419	8,19
	4	4	1662254	121577	245625	7	1535984	173254	213570	7,38
		8	17544478	359368	1307567	31,79	16899576	493526	1263759	33,52
		16	76029534	940253	3709862	88,69	74405816	1160150	3615288	91,08
	5	4	5210606	201944	642927	16,11	4786212	284327	585072	16,58
		8	69933358	613857	5336262	113,49	67000420	836948	5022660	111,76
		16	515212046	1732875	23905018	489	503155140	2216827	22294772	467,52
5	3	2	425680	42617	23083	1,25	414945	60921	22330	1,59
		4	13866186	203773	337973	10,33	13530977	315778	335160	12,42
		8	115442926	480998	2000007	47,32	113502147	694797	1866828	48,86
	4	2	1468734	79064	71683	2,88	1400162	130713	66253	3,76
		4	50983200	366540	1534580	36,26	48529826	611695	1709632	44,28
		8	1013071140	1186488	16335333	334,2	988159390	1832246	17860183	375,6
6	3	2	5192918	80241	45949	2,41	5150162	120602	46035	3,18
		4	357387886	514328	1724448	42,7	352902234	895853	1273523	41,38
		8	4318481382	1166891	16361053	334,32	4272806046	1876926	14893740	319,88
	4	2	21296478	155987	168976	6,2	20865758	290632	162910	8,65
		4	1540484738	962251	10307169	214,95	1496184324	1833166	12788066	278,88

Tiempo											
H	N	K	MTBDD			MTBDD con reducción					
			Cálculo		TOTAL	Find		Make	Cálculo		TOTAL
			iters.	seg.	seg.	iters	seg.	seg.	iters.	seg.	seg.
3	5	8	193	128,13	152,65	47	13,88	1,49	62	48,86	90,23
		16	234	463,42	527,3	87	43,2	2,77	68	169,4	283,24
		32	237	1015,55	1180,82	87	69,66	3,57	71	325,53	538,14
4	3	4	88	23,97	37,71	29	6,85	1,21	42	18,33	41,98
		8	92	89,09	126,12	29	15,89	2,22	46	67,71	126,11
	4	4	128	134,49	168,79	29	15,83	3,13	57	89,55	145,83
		8	156	1052,27	1184,79	49	63,72	7,13	65	683,83	886,15
		16	160	2742,67	3007,37	49	124,18	9,7	69	1853,65	2266,83
	5	4	163	454,25	522,92	29	29,16	4,98	72	331,92	435,11
		8	215	5247,79	5571,81	49	109,76	13,86	83	3001,88	3486,35
		16	258	39374,44	40418,26	89	390,32	30	91	15596,62	17247,52

Tiempo											
H	N	K	MTBDD			MTBDD con reducción					
			Cálculo		TOTAL	Find		Make	Cálculo		TOTAL
			iters.	seg.	seg.	iters	seg.	seg.	iters.	seg.	seg.
5	3	2	82	7,95	18,74	21	3,11	1,1	43	8,44	24
		4	102	174,35	236,23	31	21,69	6,16	53	155,42	242,14
		8	107	1063,78	1218,91	31	52,21	8,43	58	1000,37	1218,77
	4	2	112	34,89	54,56	21	9,29	2,25	57	37,08	69,38
		4	147	1322,42	1520,21	31	62,02	11,65	72	1167,39	1418,21
		8	177	18023,66	18826,54	51	255,03	32,57	81	11616,61	12709,68
6	3	2	94	23,41	46,65	23	5,56	2,44	52	27,31	57,68
		4	116	1264,45	1516,88	33	77,58	16,08	64	902,72	1200,1
		8	122	9720,29	10371,34	33	172,3	31,48	70	10087,12	10843,94
	4	2	123	113,65	162,97	23	20,47	5,56	69	126,73	204,46
		4	161	10236,22	11088,77	33	213,51	42,78	86	10613,92	11677,36

Desde el punto de vista del modelo, todos los parámetros contribuyen a un incremento en el costo de los cálculos numéricos. Cada incremento de H introduce un módulo que intenta enviar un mensaje, el incremento de K introduce más transiciones probabilistas y finalmente, para N mayores, se incrementan los reintentos de espera.

Observamos que, el incremento de N, en general, permite una mayor reducción de estados. En cambio, los estados que introduce el aumento de K no son tan susceptibles a la reducción. Finalmente vemos que a medida que aumenta H la reducción de estados, disminuye.

Con respecto a nuestro método, notamos que efectúa un mayor número de iteraciones cuando K es mayor. Este aumento de iteraciones se traduce en una disminución en el cálculo numérico posterior.

Podemos concluir que, en general, a mayor K nuestro método elimina estados costosos para el cálculo numérico, y a mayor N, elimina más estados, pero menos costosos. Por esto es que, al aumentar N, la regularidad de la matriz de iteración disminuye más (por lo tanto aumenta el uso de memoria).

Por otro lado, como es de esperarse, la disminución de estados mejora el peor caso del vector de soluciones. Lamentablemente, como en los casos más grandes se pierde mucha regularidad, la mejora del tamaño del vector no alcanza para que se obtenga la solución más rápido.

En general, nuestra reducción, no es apropiada para la verificación de esta propiedad en el caso MTBDD, pero se pueden obtener mejoras con K suficientemente grande.

6.1.2. Híbrido

Espacio								
H	N	K	Híbrido			Híbrido con reducción		
			Estados	Iter matrix	TOTAL	Est. Ese.	Iter matrix	TOTAL
			cant	nodos	MB	cant.	nodos	MB
3	5	8	779094	186516	33,4	734363	302708	34,6
		16	3504254	431836	110	3402860	711267	119,1
		32	9490126	1104545	291,2	9328024	1481800	301,7
4	3	4	532326	129043	25,5	508272	246591	27,7
		8	3020342	311859	94	2941604	522746	100,4
	4	4	1662254	235538	56,1	1535984	532048	65,8
		8	17544478	735459	474,6	16899576	1633014	495,7
		16	76029534	2139988	1945,6	74405816	3576209	2048
	5	4	5210606	429396	153,7	4786212	1008333	168,5
		8	69933358	1414545	1843,2	67000420	3321839	1843,2
		16	515212046	4139586	13004,8	503155140	8907381	12902,4
5	3	2	425680	80467	20,8	414945	150810	21,6
		4	13866186	419867	367,1	13530977	1049205	388,1
		8	115442926	1015903	2867,2	113502147	2239122	2969,6
	4	2	1468734	163944	49,4	1400162	392431	57,5
		4	50983200	826518	1331,2	48529826	2553133	1331,2
		8	1013071140	2872804	25292,8	988159390	8895312	24883,2
6	3	2	5192918	165014	142,5	5150162	330206	147,4
		4	357387886	1138791	8908,8	352902234	3380092	8908,8
	4	2	21296478	356794	551,1	20865758	1000059	568,3

Tiempo											
H	N	K	Híbrido			Híbrido con reducción					
			Cálculo		TOTAL	Find		Make	Cálculo		TOTAL
			iters	seg.	seg.	iters	seg.	seg.	iters	seg.	seg.
3	5	8	193	27,05	58,53	47	13,88	1,49	62	33,24	82,89
		16	234	135,72	196,66	87	43,2	2,77	68	155,3	261,02
		32	237	606,91	739,97	87	69,66	3,57	71	613,76	817,76
4	3	4	88	8,56	21,97	29	6,85	1,21	42	12,51	34,62
		8	92	50,39	79,69	29	15,89	2,22	46	58,08	101,38
	4	4	128	35,22	66,45	29	15,83	3,13	57	64,24	112,88
		8	156	477,55	585,74	49	63,72	7,13	65	605,14	776,91
		16	160	3014,4	3280,86	49	124,18	9,7	69	2822,61	3195,54
	5	4	163	147,4	215,23	29	29,16	4,98	72	306,52	425,51
		8	215	2911,34	3320	49	109,76	13,86	83	2921,49	3381,58
		16	258	24593,96	25613,39	89	390,32	30	91	22407,56	24278,57

Tiempo											
H	N	K	Híbrido			Híbrido con reducción					
			Cálculo		TOTAL	Find		Make	Cálculo		TOTAL
			iters	seg.	seg.	iters	seg.	seg.	iters	seg.	seg.
5	3	2	82	6,41	16,59	21	3,11	1,1	43	3,77	17,53
		4	102	241,41	294,89	31	21,69	6,16	53	242,24	322,11
		8	107	2123,22	2261,78	31	52,21	8,43	58	1458,41	1636,6
	4	2	112	24,09	43,2	21	9,29	2,25	57	26,53	56,42
		4	147	1338,27	1494,79	31	62,02	11,65	72	1550,54	1764,73
		8	177	33539,29	34400,26	51	255,03	32,57	81	22286,84	24245,51
6	3	2	94	88,66	112,06	23	5,56	2,44	52	46,63	81,6
		4	116	8323,23	8530,62	33	77,58	16,08	64	5038,69	5328,08
	4	2	123	422,09	474,9	23	20,47	5,56	69	278,39	357,2

En el caso Híbrido, el aumento del tamaño de la matriz de iteración luego de la reducción de estados, es mucho mayor. Esto es normal ya que en el caso híbrido, la matriz se preprocesa y se le agrega información que optimiza su rendimiento para ser multiplicada por un vector explícito. Afortunadamente, la eliminación de estados, compensa el uso de memoria ya que el vector de iteración es menor.

Como con el caso MTBDD, si el aumento de los estados es producto de un K mayor, la mejora que introduce nuestro método en el cálculo numérico, es también mayor. (o, la demora que introduce, es menor).

Claramente, la mejora en el uso de memoria no es significativa debido a la pequeña reducción de estados. Con respecto al costo computacional, es difícil predecir el comportamiento del método para casos mayores. Por lo que en general, nuestra reducción tampoco es apropiada para utilizar en este caso.

6.2. Zeroconf

Zeroconf es un protocolo diseñado para que la inicialización de una red local se realice de manera automática. Ya que cada vez es más común contar con dispositivos domésticos capaces de intercomunicarse a través de una red IP, es deseable que el proceso de configuración sea automatizado. Esta propiedad requiere que cada dispositivo sea capaz de configurar su dirección IP única y, que la red pueda mantener esta asignación de direcciones IP, inclusive cuando un dispositivo se elimina de la misma.

Al conectarse a la red, un dispositivo selecciona una dirección IP entre 65024. Luego, enviá mensajes al resto de los dispositivos de la red preguntando si la dirección se encuentra actualmente en uso. Si no recibe respuesta, el dispositivo comienza a utilizar esa dirección, y comunica esto enviando otra serie de mensajes.

Los mensajes enviados inicialmente por el dispositivo (que llamaremos *host*) se denominan *probes*. Se envían 4 de estos mensajes separados por 2 segundos. Si la dirección está siendo usada por otro dispositivo, este lo indica enviando una respuesta, por lo que, el *host* original reinicia el proceso con otra dirección IP. Si luego de enviar los 4

probes no recibe una respuesta, comienza a utilizar la dirección y enviá 2 mensajes más denominados *gratuitous*.

En el caso de que un *host* reciba un *probe* con la misma dirección que se encuentra utilizando, debe responder al paquete y continuar utilizando la dirección. En cambio, si recibe un *gratuitous* conteniendo su dirección, debe defenderla o entregarla. El *host* solo puede defender la dirección si no recibió un mensaje conflictivo en los últimos 10 segundos; de lo contrario, debe entregar la dirección. Un *host* que defiende una dirección, responde enviando un *gratuitous*, indicando que la dirección está en uso. De lo contrario, deja de utilizar la dirección y reinicia el protocolo.

Este protocolo puede operar en el contexto de cualquier IEEE 802 o similar, con velocidades de transferencia de al menos 1Mb/s y que soporte ARP, como Ethernet.

A continuación verificamos una propiedad sobre un modelo de *zeroconf* en el que se asume un canal único de comunicación sin routers, en donde los mensajes llegan en el mismo orden en el que fueron enviados. También, asumimos que un *host* nunca reintenta con la misma dirección y, además, los *hosts* siempre defienden su IP.

N representa la cantidad de *hosts* presentes en la red con su dirección IP resuelta, K indica el número de *probes* a enviar y *loss* la probabilidad de perdida de un mensaje.

Propiedad a verificar: $p^{min}\,[\mathbf{true}\,\mathcal{U}\,(\mathbf{use}\,\&\,\mathbf{ipInUse})]$
(La probabilidad mínima de tomar una dirección IP que se encuentra en uso)

6.2.1. MTBDD

Espacio										
K	loss	N	MTBDD				MTBDD con reducción			
			Estados	It. mat.	Max vec.	Mem.	Est. ese.	It. mat.	Max vec.	Mem.
			cant.	nodos	nodos	KB	cant.	nodos	nodos	KB
2	10^{-4}	10	89586	20187	9958	588	74035	20663	9770	594
		10^3	89586	20187	26445	910	74035	20663	25736	906
		10^4	89586	20187	41814	1210	74035	20663	43190	1247
	0,1	10	89586	20187	140392	3136	74035	20663	101895	2393
		10^3	89586	20187	204042	4379	74035	20663	116824	2685
		10^4	89586	20187	285099	5962	74035	20663	128534	2914
4	10^{-4}	10	307768	30915	8788	775	248735	32072	7844	779
		10^3	307768	30915	8778	775	248735	32072	7844	779
		10^4	307768	30915	8768	775	248735	32072	7844	779
	0,1	10	307768	30915	305515	6570	248735	32072	280605	6106
		10^3	307768	30915	591793	12162	248735	32072	347133	7406
		10^4	307768	30915	958465	19323	248735	32072	450606	9427
5	10^{-4}	10	496291	40827	9218	977	399181	42395	8245	989
		10^3	496291	40827	9323	979	399181	42395	8245	989
		10^4	496291	40827	9223	977	399181	42395	8245	989
	0,1	10	496291	40827	425884	9115	399181	42395	394034	8524
		10^3	496291	40827	818101	16775	399181	42395	506767	10725
		10^4	496291	40827	1378659	27724	399181	42395	708989	14675

Tiempo											
K	loss	N	MTBDD			MTBDD con reducción					
			Cálculo		TOTAL	Find		Make	Cálculo		TOTAL
			iter.	seg.	seg.	iter.	seg.	seg.	iter.	seg.	seg.
2	10^{-4}	10	72	4,87	22,19	109	2,96	0,33	72	5,57	26,41
		10^3	145	16,94	34,26	109	2,96	0,33	89	15,72	37,05
		10^4	368	66,52	84,02	109	2,96	0,33	152	49,93	71,01
	0,1	10	206	186,54	204,17	109	2,96	0,33	92	127,71	148,98
		10^3	282	490,56	509,23	109	2,96	0,33	113	206,45	228,83
		10^4	507	1490,73	1508,64	109	2,96	0,33	176	371,28	394,47
4	10^{-4}	10	64	4,31	44,87	117	5,46	0,54	59	3,98	50,75
		10^3	64	4,35	45,51	117	5,46	0,54	59	3,95	51,69
		10^4	64	4,26	44,97	117	5,46	0,54	59	3,96	51,22
	0,1	10	228	593,01	635,66	117	5,46	0,54	120	477,09	526,65
		10^3	307	2065,89	2114,2	117	5,46	0,54	153	897,6	952,06
		10^4	532	8757,87	8808,75	117	5,46	0,54	249	1893,08	1950,62
5	10^{-4}	10	64	4,08	67,13	121	6,51	0,8	59	4,38	75,32
		10^3	64	4,24	68,23	121	6,51	0,8	59	4,5	78,08
		10^4	64	4,06	67,73	121	6,51	0,8	59	4,4	75,66
	0,1	10	241	891,38	956,93	121	6,51	0,8	133	775,45	849,63
		10^3	316	3182,18	3250,5	121	6,51	0,8	173	1590,35	1668,7
		10^4	543	18074,05	18143,29	121	6,51	0,8	286	4466,94	4545,29

En nuestro modelo, N define la probabilidad de tomar una dirección IP libre y *loss* define la probabilidad de que nuestro mensaje no llegue, por lo que el único parámetro que afecta el tamaño del modelo es K, es decir, la cantidad de *probes*.

Consistentemente, mientras mayor es K, hay una mayor reducción de estados, para $K = 2$ es del 17 % y para $K = 5$ del 20 %.

Notar como, en el caso donde $K = 2$, para la probabilidad de perdida de 10^{-4}, el incremento de N produce que las iteraciones del cálculo numérico, sean mayores. Esto indica que la mayor probabilidad de perder los *probes* al ser pocos, hace más difícil la convergencia del método numérico.

En el resto de los casos, con $K \geq 4$, cuando la probabilidad de perder un mensaje es pequeña. La convergencia es muy rápida. Por lo que nuestro método no logra reducir notablemente el cómputo numérico, produciendo una ligera demora y un pequeño aumento del tamaño.

Para los casos donde el error es significativo, incluido el que mencionamos al principio, vemos que nuestro método mejora notablemente la velocidad, reduciendo mucho las iteraciones, y también el espacio ocupado ya que el vector de soluciones no tiene tiempo de perder mucha regularidad.

Para este caso, nuestro método es muy efectivo ya que logra reducir mucho el costo computacional cuando este es grande y no introduce demoras importantes cuando el costo del cálculo numérico es chico.

6.2.2. Híbrido

Espacio								
K	loss	N	Híbrido			Híbrido con reducción		
			Estados	It. mat.	TOTAL	Est. ese.	It. mat.	TOTAL
			cant.	nodos	memoria	cant.	nodos	memoria
2	10^{-4}	10	89586	57328	6,2 MB	74035	62146	5,7 MB
		10^3	89586	57328	6,2 MB	74035	62146	5,7 MB
		10^4	89586	57328	6,2 MB	74035	62146	5,7 MB
	0,1	10	89586	57328	6,2 MB	74035	62146	5,7 MB
		10^3	89586	57328	6,2 MB	74035	62146	5,7 MB
		10^4	89586	57328	6,2 MB	74035	62146	5,7 MB
4	10^{-4}	10	307768	89565	16,7 MB	248735	96189	14,4 MB
		10^3	307768	89565	16,7 MB	248735	96189	14,4 MB
		10^4	307768	89565	16,7 MB	248735	96189	14,4 MB
	0,1	10	307768	89565	16,7 MB	248735	96189	14,4 MB
		10^3	307768	89565	16,7 MB	248735	96189	14,4 MB
		10^4	307768	89565	16,7 MB	248735	96189	14,4 MB
5	10^{-4}	10	496291	120436	24,9 MB	399181	132663	21,2 MB
		10^3	496291	120436	24,9 MB	399181	132663	21,2 MB
		10^4	496291	120436	24,9 MB	399181	132663	21,2 MB
	0,1	10	496291	120436	24,9 MB	399181	132663	21,2 MB
		10^3	496291	120436	24,9 MB	399181	132663	21,2 MB
		10^4	496291	120436	24,9 MB	399181	132663	21,2 MB
7	10^{-4}	10	1248568	204677	47,7 MB	1008973	233538	42,6 MB
		10^3	1248568	204677	47,7 MB	1008973	233538	42,6 MB
		10^4	1248568	204677	47,7 MB	1008973	233538	42,6 MB
	0,1	10	1248568	204677	47,7 MB	1008973	233538	42,6 MB
		10^3	1248568	204677	47,7 MB	1008973	233538	42,6 MB
		10^4	1248568	204677	47,7 MB	1008973	233538	42,6 MB
8	10^{-4}	10	1870338	267752	66,6 MB	1516175	354475	59,4 MB
		10^3	1870338	267752	66,6 MB	1516175	354475	59,4 MB
		10^4	1870338	267752	66,6 MB	1516175	354475	59,4 MB
	0,1	10	1870338	267752	66,6 MB	1516175	354475	59,4 MB
		10^3	1870338	267752	66,6 MB	1516175	354475	59,4 MB
		10^4	1870338	267752	66,6 MB	1516175	354475	59,4 MB

Tiempo											
K	loss	N	Híbrido			Híbrido con reducción					
			Cálculo		TOTAL	Find		Make	Cálculo		TOTAL
			iter.	seg.	seg.	iter.	seg.	seg.	iter.	seg.	seg.
2	10^{-4}	10	208	2,68	19,75	109	2,96	0,33	93	1,01	21,37
		10^3	284	3,58	20,62	109	2,96	0,33	119	1,27	21,71
		10^4	580	7,15	24,3	109	2,96	0,33	200	1,95	22,56
	0,1	10	206	2,64	19,66	109	2,96	0,33	92	1,02	21,54
		10^3	282	3,56	20,55	109	2,96	0,33	113	1,2	21,56
		10^4	507	6,28	23,41	109	2,96	0,33	176	1,77	22,31
4	10^{-4}	10	227	12,59	53,69	117	5,46	0,54	123	5,41	51,54
		10^3	304	16,82	57,98	117	5,46	0,54	163	6,96	54,21
		10^4	598	32,39	73,09	117	5,46	0,54	292	12	59
	0,1	10	228	12,73	54,11	117	5,46	0,54	120	5,28	51,9
		10^3	307	17,03	58,32	117	5,46	0,54	153	6,91	54,68
		10^4	532	28,78	69,84	117	5,46	0,54	249	10,3	56,71
5	10^{-4}	10	236	21,4	85,59	121	6,51	0,8	138	10,59	82,17
		10^3	316	29,25	94,68	121	6,51	0,8	185	13,96	86,45
		10^4	606	52,58	116,71	121	6,51	0,8	337	24,61	96,94
	0,1	10	237	21,49	85,88	121	6,51	0,8	133	10,33	82,08
		10^3	316	28,34	92,19	121	6,51	0,8	173	13,1	85,45
		10^4	543	54,22	124,84	121	6,51	0,8	286	21,29	93,71
7	10^{-4}	10	255	52	204,53	129	11,44	1,53	166	32,44	197,25
		10^3	340	67,92	221,1	129	11,44	1,53	229	43,51	209,87
		10^4	624	121,54	273,26	129	11,44	1,53	430	77,27	245,42
	0,1	10	254	51,76	203,94	129	11,44	1,53	161	31,59	199,15
		10^3	334	67,47	221,59	129	11,44	1,53	212	40,66	209,21
		10^4	567	111,3	264,42	129	11,44	1,53	358	65,08	231,18
8	10^{-4}	10	264	79,46	304,9	133	14,91	1,81	180	52,1	296,01
		10^3	352	103,17	343,53	133	14,91	1,81	251	68,69	329,08
		10^4	633	179,98	405,49	133	14,91	1,81	472	118,21	362,75
	0,1	10	263	79,92	304,62	133	14,91	1,81	174	51,19	296,61
		10^3	345	101,34	328,63	133	14,91	1,81	230	63,91	309,87
		10^4	578	166,2	396,3	133	14,91	1,81	392	99,92	342,08

En el caso híbrido la velocidad de convergencia del cálculo numérico es mucho más homogénea y no se ve notablemente afectada por la probabilidad de perder los *probes*.

La reducción de estados esenciales reduce notablemente las iteraciones y el costo del cálculo numérico. De cualquier forma, notamos que esta mejora, disminuye ligeramente a medida que aumenta K.

Con respecto al espacio, a partir de $K = 7$, la reducción de estados comienza a disminuir y la irregularidad de la matriz de iteración comienza a ser mayor, incrementando su tamaño.

Más allá que el tiempo total del model checking sea menor con K mayores, vemos que a partir de $K = 7$ nuestro en método, en términos generales comienza a dejar de introducir mejoras, tanto en costo computacional, como en espacio de almacenamiento.

6.3. Bounded retransmission protocol

Bounded retransmission protocol (BRP) es un protocolo diseñado para enviar un archivo sobre un canal donde se puede perder información. Está basado en el conocido protocolo *alternating bit* y fue diseñado por Philips para la comunicación entre controles remotos y equipos de audio/video/TV. Básicamente, BRP se utiliza para enviar un archivo en trozos, en donde se permite una cantidad acotada de retransmisiones de cada uno. Por lo tanto, la entrega del archivo no está garantizada y el protocolo puede abortar la transferencia.

En nuestro modelo se toman en cuenta las posibles fallas tanto en el canal de transmisión, como en el canal donde se reciben las confirmaciones del envío. La probabilidad de un envío correcto es del 99 %. N indica la cantidad de *chunks* (paquetes) que requiere el archivo para ser transmitido por completo y MAX, define el límite de retransmisiones posibles.

Propiedad a verificar: p^{max} [**true** $\mathcal{U}$ **error** & **notOk** & **i** > 8]
(La máxima probabilidad de que el transmisor reporte una transmisión fallida luego de que más de 8 trozos fueran enviados exitosamente)

6.3.1. MTBDD

Espacio									
N	MAX	MTBDD				MTBDD con reducción			
		Est.	It. mat.	Max. vec.	Mem.	Est. ese.	It. mat.	Max. vec.	Mem.
		cant.	nodos	nodos	KB	cant.	nodos	nodos	KB
64	2	2693	1763	14408	315,84	1498	1578	9454	215,47
	5	5192	1811	28246	587,05	2488	1622	19149	405,68
	6	6025	1819	31629	653,28	2818	1631	21739	456,45
	7	6858	1806	34623	711,5	3148	1624	24097	502,36
	8	7691	1859	-	-	3478	1659	26448	548,96
128	5	10376	1879	55351	1117,77	4920	1686	38689	788,57
	6	12041	1887	63284	1272,87	5570	1695	44619	904,57
	7	13706	1874	70318	1410	6220	1688	50375	1016,86
	8	15371	1927	-	-	6870	1723	-	-
256	6	24073	1955	126462	2508,14	11074	1759	90382	1799,63
	7	27402	1942	142201	2815,29	12364	1752	103251	2050,84
	8	30731	1995	-	-	13654	1787	-	-
512	6	48137	2023	276962	5448,93	22082	1823	181532	3581,15
	7	54794	2010	284469	5595,29	24652	1816	207911	4096,23

Tiempo										
N	MAX	MTBDD			MTBDD con reducción					
		Cálculo		TOTAL	Find		Make	Cálculo		TOTAL
		iter.	seg.	seg.	iter.	seg.	seg.	iter.	seg.	seg.
64	2	409	15,32	19,55	4	0,03	0,02	143	3,37	7,54
	5	419	32,9	39,02	4	0,03	0,02	147	6,77	12,81
	6	647	45,73	52,61	4	0,03	0,02	169	7,22	13,84
	7	633	40,52	47,47	4	0,03	0,02	214	9,06	16,02
	8	-	-	-	4	0,03	0,03	368	14,53	23,45
128	5	811	93,97	105	4	0,03	0,03	280	19,74	30,52
	6	1028	120,84	132,82	4	0,03	0,02	390	26,96	39,02
	7	1031	126,14	138,81	4	0,03	0,02	359	27,98	41
	8	-	-	-	4	0,03	0,02	-	-	-
256	6	2008	419,63	442,74	4	0,04	0,02	600	78,49	102,5
	7	1843	387,86	412,68	4	0,03	0,02	623	86,69	111,97
	8	-	-	-	4	0,03	0,02	-	-	-
512	6	3674	1304,11	1350,08	4	0,03	0,02	1194	276,86	324,46
	7	5081	1773,51	1822,93	4	0,04	0,02	1442	362,37	414,8

En BRB, Tanto N como MAX contribuyen al tamaño del modelo, esto es porque N introduce ciclos completos, es decir paquetes en donde cada uno se reintenta enviar MAX veces. Particularmente, en este caso esto introduce grupos de estados que son rápidamente eliminadas por nuestro método, en solo 4 iteraciones. Cada incremento en el modelo introduce más de estos grupos, pero no aumenta la longitud de los existentes, es por eso que las iteraciones se mantienen en 4.

Con respecto al caso MTBDD, nuestro método reduce más del 50 % de los estados, y como las iteraciones de este son pocas, al no introducir mucha irregularidad, el tamaño de la matriz de iteración se ve reducido un 10 %. Notamos una ligera mejora, con respecto a los estados eliminados, a medida que MAX es mayor, pero no cuando N es mayor. Como el tamaño de la matriz de iteraciones es muy pequeño, la mejora que introduce nuestro método en el uso de memoria es definida por la mejora en el tamaño del vector de iteraciones, aproximadamente 25 %-35 %.

Cuando observamos las iteraciones del cálculo numérico vemos la gran mejora que introduce nuestro método. Este, reduce entre un 50 % y un 70 % el tiempo total del cálculo. La mejora es ligeramente menor en los casos más grandes y se puede observar que la mejora general va disminuyendo con el aumento de N y MAX.

Finalmente vemos que en varios casos con $N = 8$ el método no converge en el límite de 10000 iteraciones. En el caso $N = 8, MAX = 64$ vemos como gracias a nuestro método, la verificación si converge.

6.3.2. Híbrido

Espacio							
N	MAX	Híbrido			Híbrido con reducción		
		Estados	Mat. iter.	TOTAL	Est. ese.	Mat. iter.	TOTAL
		cant.	nodos	KB	cant.	nodos	KB
64	2	2693	2653	227,1	1498	2070	143.9
	5	5192	2716	328,7	2488	2125	180,9
	6	6025	2733	362,4	2818	2130	192,6
	7	6858	2723	394,8	3148	2131	204,2
	8	7691	2786	430,6	3478	2165	217,2
128	5	10376	2820	535	4920	2230	265,9
	6	12041	2837	601,6	5570	2235	288,8
	7	13706	2827	667	6220	2236	311,6
	8	15371	2890	735,8	6870	2270	335,9
256	6	24073	2941	1126,4	11074	2340	476,2
	7	27402	2931	1228,8	12364	2341	521.5
	8	30731	2994	1331,2	13654	2375	568,3
512	6	48137	3045	2048	22082	2445	846,2
	7	54794	3035	2252,8	24652	2446	936,4
	8	61451	3098	2560	27222	2480	1024,1
1024	6	96265	3149	3891,2	44098	2550	1536
	7	109578	3139	4403,2	49228	2551	1740,8
	8	122891	3202	4915,2	54358	2585	1945,6
2048	6	192521	3253	7680	88130	2655	3072

Tiempo										
N	MAX	Híbrido			Híbrido con reducción					
		Cálculo		TOTAL	Find		Make	Cálculo		TOTAL
		iter.	seg.	seg.	iter.	seg.	seg.	iter.	seg.	seg.
64	2	409	0,1	4,19	4	0,03	0,02	143	0	4,04
	5	419	0,2	6,05	4	0,03	0,02	147	0,02	5,9
	6	421	0,25	6,49	4	0,03	0,02	148	0,02	6,24
	7	425	0,29	6,87	4	0,03	0,02	150	0,03	6,7
	8	427	0,33	7,79	4	0,03	0,03	151	0,03	7,55
128	5	813	0,81	11,28	4	0,03	0,03	280	0,05	10,44
	6	815	0,95	12,35	4	0,03	0,02	281	0,06	11,49
	7	817	1,08	13,35	4	0,03	0,02	283	0,06	12,28
	8	821	1,23	15,11	4	0,03	0,02	284	0,08	14,02
256	6	1599	3,79	26,23	4	0,04	0,02	545	0,21	22,51
	7	1601	4,35	28,32	4	0,03	0,02	547	0,23	25,67
	8	1605	4,98	33,77	4	0,03	0,02	548	0,25	29,28

Tiempo										
N	MAX	Híbrido			Híbrido con reducción					
		Cálculo		TOTAL	Find		Make	Cálculo		TOTAL
		iter.	seg.	seg.	iter.	seg.	seg.	iter.	seg.	seg.
512	6	3163	17,23	62,23	4	0,03	0,02	1072	0,76	45,78
	7	3167	20,88	69,31	4	0,04	0,02	1073	0,86	49,03
	8	3169	24,58	80,27	4	0,04	0,02	1074	0,94	56,05
1024	6	6287	84,31	176,93	4	0,04	0,02	2122	3,52	96,71
	7	6289	103,67	203,52	4	0,03	0,03	2123	4,24	103,86
	8	6293	121,73	233,93	4	0,03	0,03	2124	4,91	118,76
2048	6	-	-	-	4	0,04	0,03	4218	16,66	213,66

En el caso Híbrido, al igual que el MTBDD, la eliminación de los estados no esenciales, al ser tan grande, logra disminuir el tamaño de la matriz de iteraciones. En este caso (debido a la complejidad de ésta en el caso híbrido), esta mejora ronda el 20 %, disminuyendo paulatinamente en los casos más grandes. En los casos donde N y MAX son más grandes nuestro método disminuye el uso de memoria, esto es, porque se eliminan más estados.

Cuando observamos las iteraciones del cálculo numérico vemos que el caso híbrido converge mucho más rápido que el MTBDD, sobre todo a partir de $MAX = 6$. Es decir, para un mismo MAX las iteraciones son mucho más homogéneas. De cualquier manera, nuestro método sigue brindando una mejora aproximada del 60 % para todos los casos. La mejora que introduce nuestro método en el cálculo numérico es consistentemente del 90 %, pero esta mejora solo comienza a apreciarse en los casos mayores ($N \geq 512$), donde el costo agregado por nuestro método deja de ser significativo.

Para esta verificación, el método es muy efectivo gracias al ahorro de memoria y tiempo.

Capítulo 7

Conclusión

Logramos diseñar e implementar un algoritmo que efectúe la *reducción de estados esenciales* sobre el *model checker* PRISM. Además, se consiguió demostrar la factibilidad de realizar una implementación utilizando únicamente estructuras de datos simbólicas.

Tal como suponíamos, el mayor impacto de la reducción puede verse en casos donde el cómputo numérico demora muchas iteraciones en converger y no donde los modelos sean muy grandes y las iteraciones pocas. Esto se debe a que, mientras más largos sean los caminos de estados en $S^?$ dentro del modelo a evaluar, el cálculo precisa más iteraciones. Estas cadenas de estados son, justamente, las que nuestro método ataca. Al aplicar el algoritmo ocurren dos cosas: Primero, la cantidad de estados en el modelo y las cadenas de este tipo se reducen; y segundo, se pierde regularidad en el modelo y, por lo tanto, su cantidad de nodos es, en general, mayor. Pero como vimos, cuando se elimina una gran cantidad de estados, la matriz de iteraciones puede achicarse. Esto afecta de manera similar al model checking con MTBDD puro y al Híbrido.

En el caso MTBDD, como el cómputo numérico se efectúa modificando un vector representado en un BDD, al disminuir las iteraciones del mismo, se disminuye el tamaño máximo que alcanza este BDD. por esta razón, digamos, la peor iteración mejora. Por otro lado, al perder regularidad en la matriz de iteración, en promedio, cada iteración es más lenta. Es el balance de esto lo que define la mejora que introduce nuestro método. En conclusión, el método mejora mucho el *model checking* sobre MTBDD cuando, en proporción al modelo, hay muchos estados en $S^?$, y el cálculo de sus probabilidades requiere muchas iteraciones. Afortunadamente, en los peores casos el método no introduce grandes demoras.

En el caso híbrido, el uso de memoria es más sensible a la disminución de estados, ya que esto afecta directamente el tamaño del vector de iteraciones. El cálculo numérico, sólo es afectado por la irregularidad de la matriz de iteraciones y no por la perdida de regularidad del vector. Por lo que, la disminución en el costo del cálculo numérico que introduce nuestro método, es más lineal a la reducción de iteraciones del cálculo. Vemos también que el tamaño de la matriz de iteraciones se ve más afectado por la eliminación de estados, pero esto, es por la información extra que se le aporta durante el preproceso, lo cual, no se traduce en una perdida mayor de regularidad. Finalmente, debido a la introducción de demoras, los beneficios de nuestro método se ven en los casos

más costosos, donde el impacto es menos significativo.

Pudimos observar que el costo del cálculo de la matriz esencial, no tiene un gran impacto en la demora total del *model checking*, por lo que su desempeño es aceptable.

Para el caso Híbrido, la implementación no es óptima debido a que, el ahorro de memoria proporcionado por usar estructuras simbólicas es innecesario. Esto es porque, durante el cálculo numérico, se utiliza mucha más memoria que la que fuera necesaria para la reducción de estados, si la misma fuera implementada con vectores explícitos. Además, debido a la complejidad de las estructuras utilizadas en el caso Híbrido, se prevé que la implementación del método puede ser optimizada significativamente.

Bibliografía

[Ake78] S. Akers. *Binary decision diagrams.* IEEE Transactions on Computers, C-27(6):509–516, 1978.

[ASB+95] A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli. *It usually works: The temporal logic of stochastic systems.* In P. Wolper, editor, Proc. 7th International Conference on Computer Aided Verification (CAV'95), volume 939 of LNCS, pp. 155–165. Springer, 1995.

[Bai98] C. Baier. *On algorithmic verification methods for probabilistic systems.* Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim, 1998.

[BdA95] A. Bianco and L. de Alfaro. *Model checking of probabilistic and nondeterministic systems.* In P. Thiagarajan, editor, Proc. 15th Conference on Foundations of Software Technology and Theoretical Computer Science, volume 1026 of LNCS, pp. 499–513. Springer, 1995.

[BFG+93] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E.Macii, A. Pardo, and F. Somenzi. *Algebraic decision diagrams and their applications.* In Proc. International Conference on Computer-Aided Design (ICCAD'93), pp. 188–191, 1993. Also available in Formal Methods in System Design, 10(2/3):171–206, 1997.

[Bry86] R. Bryant. *Graph-based algorithms for Boolean function manipulation.* IEEE Transactions on Computers, C-35(8):677–691, 1986.

[BT91] D. Bertsekas and J. Tsitsiklis. *An analysis of stochastic shortest path problems.* Mathematics of Operations Research, 16(3):580–595, 1991.

[BW96] B. Bollig and I. Wegner. *Improving the variable ordering of OBDDs is NPcomplete.* IEEE Transactions on Computers, 45(9):993–1006, 1996.

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

[CFM+93] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. *Multi-terminal binary decision diagrams: An efficient data structure for matrix representation.* In Proc.International Workshop on Logic Synthesis (IWLS'93), pp. 1–15, 1993. Also available in Formal Methods in System Design, 10(2/3):149–169, 1997.

[CMZ+93] E. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang. *Spectral transforms for large Boolean functions with applications to technology mapping.* In Proc. 30th Design Automation Conference (DAC'93), pp. 54–60. ACM Press, 1993. Also available in Formal Methods in System Design, 10(2/3):137–148, 1997.

[CY90] C. Courcoubetis and M. Yannakakis. *Markov decision processes and regular events.* In M. Paterson, editor, Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90), volume 443 of LNCS, pp. 336–349. Springer, 1990.

[dA97] L. de Alfaro. *Formal Verification of Probabilistic Systems. PhD thesis*, Stanford University, 1997.

[DER89] I. S. Duff, A. M. Erisman, J. K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, 1989

[DJ+02] Pedro R. D'Argenio, Bertrand Jeannet and Henrik E. Jensen and Kim G. Larsen, *Reduction and Refinement Strategies for Probabilistic Analysis.* H. Hermanns and R. Segala, eds. LNCS 2399 pp. 57-76, 2002.

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

[HJ94] H. Hansson and B. Jonsson. *A logic for reasoning about time and probability.* Formal Aspects of Computing, 6(5):512–535, 1994.

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

[KSK66] J. Kemeny, J. Snell, and A. Knapp. *Denumerable Markov Chains.* D. Van Nostrand Company, 1966.

[Lee59] C. Lee. *Representation of switching circuits by binary-decision programs.* Bell System Technical Journal, 38:985–999, 1959.

[Par02] David Anthony Parker, *Implementation of symbolic model checking for probabilistic systems.* University of Birmingham, Agoust 2002.

[Pnu77] A. Pnueli. *The temporal logic of programs.* In Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS'77), pp. 46–57. IEEE Computer Society Press, 1977.

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

[THY93] S. Tani, K. Hamaguchi, and S. Yajima. *The complexity of the optimal variable ordering problems of shared binary decision diagrams.* In K. Ng, P. Raghavan, N. Balasubramanian, and F. Chin, editors, Proc. 4th International Symposium on Algorithms and Computation (ISAAC'93), volume 762 of LNCS, pp. 389–398. Springer, 1993.

[Var85] M. Vardi. *Automatic verification of probabilistic concurrent finite state programs.* In Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS'85), pp. 327–338. IEEE Computer Society Press, 1985.