IMPLEMENTACIÓN DE TÉCNICAS DE DERIVACIÓN DE CONTRAEJEMPLOS EN EL MODEL CHECKER PRISM

Autor: Matías L. Marenchino
Director: Dr. Pedro R. D'Argenio

Trabajo Especial de Licenciatura en
Cs. de la Computación

18 de marzo de 2011

Facultad de Matemática, Astronomía y Física

Universidad Nacional de Córdoba
Argentina

Tabla de Contenidos

1. Introducción **1**

1.1. La necesidad de métodos formales 1

1.2. Model checking 3

1.3. Model checking probabilista 4

1.4. La necesidad de contraejemplos 5

1.5. Objetivos del trabajo 5

1.6. Estructura del trabajo 5

2. Conceptos básicos de Probabilidad **7**

2.1. Colecciones de conjuntos 7

2.2. Medidas 9

2.3. Espacios probabilistas 9

3. Modelado de sistemas **12**

3.1. Cadenas y Procesos de Decisión de Markov 12

3.2. Conversión de una DTMC en un autómata 14

3.3. Conversión de una DTMC en un digrafo ponderado 16

3.4. Caminos y ejecuciones 17

4. Propiedades a verificar **20**

4.1. Lógica Temporal Lineal LTL 20

4.1.1. Sintaxis 20

4.1.2. Semántica 21

5. Expresiones Regulares **23**

5.1. Propiedades de expresiones regulares 24

5.2. Conversión de autómatas a expresiones regulares 24

5.2.1. Método de Eliminación de Estados 24
5.2.2. Método Algebraico 25
5.2.3. Método de Clausura Transitiva 25
5.2.4. Reducción de Expresiones Regulares 26

6. Contraejemplos **28**
6.1. Contraejemplos Representativos 29
6.2. Rails y torrentes 29
6.3. Contraejemplos 34

7. Implementación **36**
7.1. PRISM 36
7.2. Arquitectura de PRISM 36
7.3. Algoritmo de generación de contraejemplos 37
7.4. Descripción de las SCCs 38
7.5. Expresiones Regulares puras 39
7.6. Expresiones Regulares reducidas 39
7.7. Implementación de los métodos de contraejemplos 39

8. Casos de estudio **42**
8.1. Leader election sincrónico 42
8.2. Crowds 45
8.3. Comparación 48

9. Conclusiones **55**

A. Apéndice A: transformación de un MDP en una DTMC **58**

Índice de figuras

1.1. Costo económico de los errores en los sistemas 2
1.2. El proceso del model checking 4

3.1. Representación gráfica de una cadena de Markov 13
3.2. Representación gráfica de un proceso de decisión de Markov 14
3.3. Representación gráfica de un proceso de decisión de Markov con no-determinismos probabilistas 15
3.4. Representación gráfica de un autómata determinista finito 16
3.5. Representación gráfica de la resolución de un scheduler sobre el proceso de decisión de Markov de la figura 3.2 18

5.1. Subgrafo a procesar - Método de Eliminación de Estados 24
5.2. Eliminación del estado q_k - Método de Eliminación de Estados 25
5.3. Forma final - Método de Eliminación de Estados 25
5.4. Método de Clausura Transitiva 26

6.1. DTMC 30
6.2. Las dos componentes fuertemente conexas en rojo y azul. A la derecha, los cuadrados representan estados de entrada; los rombos las salidas . 31
6.3. DTMC restringidas a componentes fuertemente conexas 31
6.4. DTMC 33
6.5. Un rail a la izquierda y un elemento de su torrente asociado a la derecha 34

7.1. Arquitectura de PRISM 37
7.2. Obtención de contraejemplos en PRISM 41

9.1. Posible visualización de una expresión regular 57

A.1. Ejemplo de MDP a convertir 59

Índice de Tablas

8.1. Resultados de Leader election. 50
8.2. Resultados de Crowds. 51
8.3. Resultados de BRP. 52
8.4. Resultados de Self-Stabilising. 53
8.5. Resultados de Dining cryptographers. 54

CAPÍTULO 1

Introducción

1.1. La necesidad de métodos formales

Hoy en día, los sistemas de hardware y software son ampliamente usados en aplicaciones donde las fallas no pueden ser aceptadas. Tal es el caso de equipos de uso espacial, instrumentos médicos, sistemas de control industrial, software para telecomunicaciones, entre otros. Para tener una noción de los daños que pueden producir estos defectos (llamados computer bugs en inglés), presentamos una lista conteniendo algunos de los más reconocidos en la historia:

- En 1980, NORAD (North American Aerospace Defense Command) detectó que Estados Unidos estaba bajo ataque de misiles; el problema fue causado por una falla en un circuito.
- En la década de los 80's, una máquina de terapia por radiación, llamada Therac-25, fue la responsable de al menos 5 muertes, al administrar cantidades excesivas de rayos X.
- 40 segundos después de su despegue, el Ariane 5 de la ESA (European Space Agency) se destruyó en 1996. El cohete, cuyo valor ascendió a mil millones de dólares, se auto-destruyó debido a un bug en el software de guiado del mismo: la conversión de un número de 64-bits con punto flotante en un entero de 16 bits levantó una excepción que no fue "atrapada".
- A comienzos de 2009, el motor de búsqueda de Google notificó a los usuarios que todos los sitios web en el mundo eran potencialmente maliciosos, incluyendo él mismo.
- El buque norteamericano USS Yorktown quedó durante 3 horas a la deriva en 1997, debido a un clásico bug: "division by zero".

Es de gran importancia detectar las fallas en los sistemas tan pronto como vayan apareciendo. El costo económico de la corrección de fallas de un producto ya entregado, puede ser 500 veces más caro que en una etapa de diseño. El diagrama de 1.1,

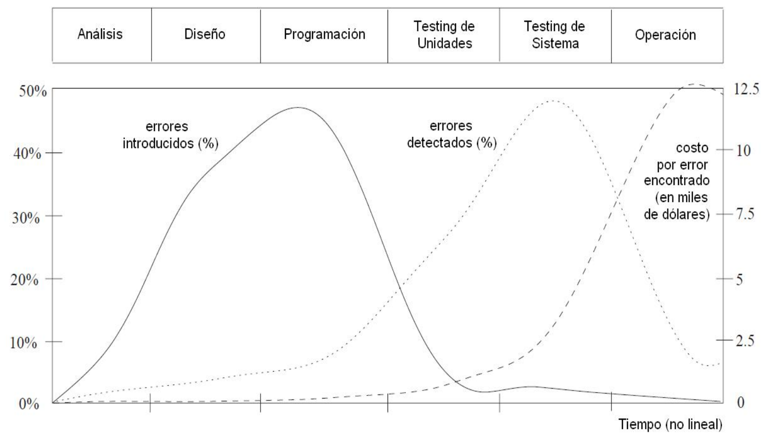

Figura 1.1: Costo económico de los errores en los sistemas

de [BK08], demuestra tal problema (y la manera habitual en que los errores van apareciendo).

Para encontrar las posibles fallas de un sistema, existen técnicas que se pueden englobar en *simulación*, *testing*, *verificación deductiva* y *model checking*. Las dos primeras se realizan antes del lanzamiento del sistema. Mientras la simulación se realiza1 sobre un prototipo abstracto del sistema; el testing se hace sobre el sistema en sí. Básicamente, se emplean para encontrar errores en el diseño o el protocolo y, de no encontrarlas, para ganar confianza en que el sistema satisface ciertas propiedades. Éstas, pueden ser elementales o muy críticas, dependiendo del sistema y en general surgen de los requerimientos que tenga el mismo. Se encuentra un bug cuando el sistema no satisface la propiedad; la “corrección” de un programa está sujeta a tales propiedades (es decir, depende de ellas); por ende no constituye una perfección absoluta.

Estos métodos son los más empleados en el mercado. Presentan la ventaja de que se pueden utilizar desde el comienzo del desarrollo. El problema que acarrean es que no son exhaustivos, es decir que el chequeo de todas las instancias de un sistema resulta imposible, por lo que permiten encontrar errores, pero no garantizan la existencia de los mismos. Pankaj Jalote en [Jal91], describe este problema mediante una pregunta que surge en el desarrollo de software: "¿cuándo dejar de testear?"; es decir, puede que al testear no encontremos errores, pero quizás el debugging no está siendo óptimo.

La verificación deductiva, por su parte, se refiere a una técnica que emplea axiomas y demostradores de teoremas para demostrar la corrección del sistema. En sus comienzos se usó para la comprobación de sistemas críticos y se realizaba a mano, lo cual la hacía completamente costosa. Con el tiempo, se fueron desarrollando herramientas de software para automatizar ésta técnica. De todas maneras, el tiempo empleado es grande, tanto es así, que probar un protocolo o circuito simple puede durar incluso meses.

El último método nombrado, model checking, será descripto en el siguiente apartado por ser una base fundamental para el presente trabajo.

1.2. Model checking

El model checking es un método de verificación formal que, dado un modelo finito de un sistema, verifica automáticamente que cumpla una especificación. Para lograrlo algorítmicamente, debemos emplear un lenguaje matemático adecuado tanto para describir el modelo, como para describir las especificaciones. Gran cantidad de model checkers han sido desarrollados para el chequeo del diseño de modelos de software y hardware donde las especificaciones son dadas como fórmulas de alguna lógica temporal, que permiten describir la ocurrencia de eventos a través del tiempo. Los pioneros en tal actividad fueron Edmund M. Clarke, E. Allen Emerson en [EC80, CE81] y por J. P. Queille y Joseph Sifakis en [QS82]. Por sus contribuciones, Clarke, Emerson, y Sifakis obtuvieron el premio Turing en 2007.

Si bien la restricción de trabajar en modelos finitos puede parecer una desventaja; el model checking es aplicable a gran cantidad de sistemas como ser protocolos de comunicación y controladores de Hardware. Incluso, es posible utilizarlo en sistemas con infinitos estados mediante la abstracción de sus partes.

Entre las ventajas de usar model checking, tenemos que:

- es un método de verificación aplicable a una gran variedad de aplicaciones;
- puede realizarse de manera parcial y;
- una de las más importantes es que provee información de las causas de falla (en caso que las hubiera).

Algunas de sus debilidades son: su aplicación involucra la subjetividad en caso de tener que reducir modelos; verifica un modelo y no el sistema en sí (por ende, pueden aparecer errores humanos en la etapa de producción); sólo se verifican las propiedades que se pretenden evaluar, lo cual no garantiza completitud; requiere de ciertos conocimientos en la materia para poder realizar las abstracciones necesarias para poder expresar el modelo; y para modelos con gran cantidad de estados, es probable que se necesiten de mucha memoria, haciéndolo casi imposible de ejecutar.

A pesar de tales argumentos, nunca es posible garantizar resultados para sistema de tamaño real, por ello el model checking es una herramienta potente para descubrir errores.

El modelo es una descripción abstracta del sistema, frecuentemente generado en forma automática a partir de su descripción en algún lenguaje, que puede generarse en base a su diseño previo a la implementación o bien a partir del código mismo del sistema. Aunque la abstracción reduce el espacio de estados a verificar, la necesidad de considerar todas las ejecuciones posibles lleva a una explosión de estados que dificulta la verificación de grandes sistemas.

En tanto, para las especificaciones (o propiedades), se suelen usar las lógicas temporales LTL (linear time logic) o CTL (computation tree logic).

Una de las grandes bondades del model checking, es que si la propiedad a verificar no es válida; se obtiene un contraejemplo, el cual sirve como una herramienta básica para realizar debugging. Un diagrama que representa el funcionamiento del model checking es presentado en la figura 1.2.

Ahora bien, si en el modelo ocurren eventos que no son certeros, sino que tienen ciertas chances de ocurrir, el model checking pasa a tener una respuesta a una probabilidad de satisfacción de la propiedad. Es cuando el model checking probabilista surge.

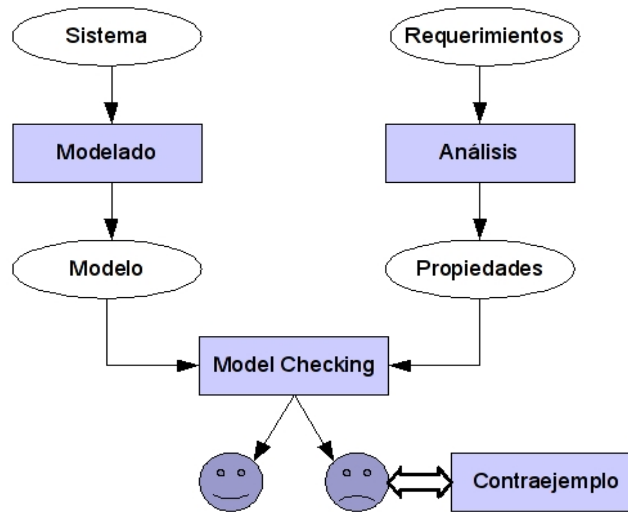

Figura 1.2: El proceso del model checking

1.3. Model checking probabilista

El model checking probabilista ([Var85]) es la técnica de model checking aplicada al análisis de sistemas que exhiben comportamientos probabilísticos o estocásticos. La diferencia crucial con el model checking radica en que los modelos presentan información extra respecto a las probabilidades o tiempos de transición de estados.

Los usos del model checking probabilista está muy ligado al análisis y verificación de sistemas que exhiben incertezas, como ser sistemas embebidos, protocolos de comunicación, sistemas distribuidos, seguridad e incluso sistemas biológicos. Existen varias formas de representar los modelos para tales sistemas, la mayoría de las cuales están basadas en cadenas de Markov. Dependiendo de cómo se modele el paso del tiempo, podremos tener Cadenas de Markov continuas en el tiempo, discretas en el tiempo o procesos de Decisión de Markov. En el presente trabajo centraremos nuestra atención en los dos últimos.

Puesto que estaremos considerando probabilidades dentro del comportamiento de tales sistemas, el conjunto de propiedades asociadas a éstos no es descriptible mediante la lógica usual. Es por ello que se emplean lógicas temporales (que describen la variación del comportamiento en el tiempo). En particular, en el presente trabajo, utilizaremos la lógica LTL.

Muchos algoritmos eficientes de model checking probabilista han sido desarrollados e implementados. La base de ellos está dada por la combinación de técnicas de cálculo numérico con análisis de alcanzabilidad. De esta manera, el computo de probabilidades de que se alcance un conjunto de estados a partir de estados iniciales puede ser calculado, rápidamente, aún cuando los modelos incluyan millones de estados. Remarcamos que nuestro interés no será el de trabajar con modelos sumamente grande ya que los contraejemplos que se generen serán imposibles de “debuggear”.

1.4. La necesidad de contraejemplos

Una de las mayores fortalezas del model checking, es la posibilidad de generar contraejemplos cuando la propiedad a verificar no se cumple. Constituyen una herramienta fundamental para "debuggear" el modelo (es decir, nos da una noción de dónde pueden estar los bugs en el modelo). El contraejemplo describe una ejecución desde los estados iniciales del sistema hasta los estados en que se violan las propiedades. Mediante la simulación, el usuario podrá reproducir el escenario en el cual ocurre el error, y en base a ello, readaptar el sistema.

Si bien los contraejemplos fueron objeto de estudio desde los orígenes del model checking, no ocurrió lo mismo con el model checking probabilista . Alrededor de quince años después de su nacimiento, en el año 2005 con el trabajo [AHL05], se comenzó a centrar la atención en éstos contraejemplos. El problema que surge aquí, es la manera en que se describen los contraejemplos, puesto que ahora no es una ejecución, sino un conjunto de ellas que constituirán nuestros contraejemplos a modelos probabilistas. En [AHL05], los contraejemplos son explícitos (se describe el contraejemplo con la ejecución más probable que viola la propiedad), mientras que en [And06], [ADR09] y [HKD09] son simbólicos, por lo cual consideramos que son más adecuados para poder realizar un análisis de los mismos y por ende serán de mayor uso para debugging. Por tal motivo, centraremos nuestra atención en los últimos.

1.5. Objetivos del trabajo

El objetivo central del presente trabajo es el de añadir una funcionalidad a PRISM [HKNP06] para describir contraejemplos. PRISM es un model checker probabilista, que permite modelar formalmente y analizar sistemas probabilistas. Si bien nosotros trabajaremos sobre Cadenas de Markov de Tiempo Discreto y Procesos de Decisión de Markov como instrumentos para modelar, PRISM también soporta Cadenas de Markov de Tiempo Continuo.

Todos estos modelos son descriptos empleando un lenguaje simple, definido en [Par02]. PRISM se basa en Diagramas de Decisión Binarios y Diagramas de Decisión Binarios Multi-Terminales para realizar el manejo de estructuras de datos y la realización de cálculos. Los mismos están descriptos en [Bed07]. Si bien para el presente trabajo se hizo uso de ellos, no serán aquí definidos.

Gran parte de los algoritmos implementados son propuestos en [And06], [ADR09] y [HKD09]. A su vez, se implementaron algoritmos novedosos para computar contraejemplos que permitirán establecer comparaciones y evaluar los resultados obtenidos.

1.6. Estructura del trabajo

A lo largo de este informe se introducirán los conceptos necesarios para entender el problema, se definirán los contraejemplos y se presentará la implementación de los algoritmos propuestos.

Más precisamente, los temas desarrollados en los próximos capítulos incluirán:

Capítulo 2

En este capítulo se definen las probabilidades y los espacios probabilísticos, esenciales en los capítulos posteriores. Incluso presentaremos algunos teoremas básicos que

serán de vital importancia para el trabajo posterior.

Capítulo 3

El capítulo presenta dos instrumentos básicos para modelar sistemas probabilísticos: las cadenas de Markov, y los procesos de decisión de Markov (que resultan de la generalización de los anteriores). También se agregan conceptos básicos ligados a ellos como ser caminos y conjuntos de aceptación.

Capítulo 4

En este capítulo se explica la lógica LTL; que es el lenguaje que utilizaremos para la definición de propiedades (a evaluar sobre las cadenas y procesos de Markov).

Capítulo 5

En el capítulo se presentan y analizan las expresiones regulares, las cuales nos permitirán generar contraejemplos significativos para poder ser debuggeados. Agregamos también las operaciones sus propiedades.

Capítulo 6

El capítulo introduce los conceptos relacionados con contraejemplos. Se presentan algoritmos para la modificación de los modelos sin alterar los contraejemplos y se añade una noción para debuggear: los carriles.

Capítulo 7

En el capítulo analizaremos la implementación, tanto de PRISM (model checker) como la de los algoritmos aquí presentados.

Capítulo 8

Se presentan en este capítulo casos de estudio para evaluar las técnicas presentadas. Se estudiarán los resultados alcanzados para poder obtener conclusiones de los algoritmos implementados.

Capítulo 9

El capítulo final reúne las conclusiones que se obtuvieron con la realización del presente trabajo final. Además, de los estudios realizados surgen nuevas ideas para trabajos futuros.

CAPÍTULO 2

Conceptos básicos de Probabilidad

La noción de contraejemplos en Model Checking Probabilista, está estrechamente relacionada con las probabilidades de realizar las ejecuciones que involucran estos contraejemplos. Por ello, debemos definir algunos conceptos de probabilidad para así poder establecer medidas de probabilidad sobre el conjunto de tales ejecuciones.

En Probabilidad, en ocasiones es imposible tratar a cualquier subconjunto de un espacio muestral Ω como un posible evento (es decir, un subconjunto al cual se le asigne una probabilidad). Tal es el caso de los números reales $\mathbb{R}$, el cual contiene ciertos subconjuntos patológicos que dificultan las definiciones de tales conceptos. Por citar un ejemplo, los conjuntos de Vitali [Haw79] son no-medibles en el intervalo $[0,1]$ con la medida de Lebesgue. Es por ello, que necesitamos limitar los subconjuntos de nuestro universo a los cuales le asignaremos probabilidades. Para esto, introducimos algunas definiciones básicas.

Como en el presente apartado sólo estamos introduciendo tópicos básicos, no haremos incapié en ninguna prueba; sólo nos acotaremos a enunciarlos.

2.1. Colecciones de conjuntos

Definición 2.1. Sea Ω un conjunto; $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ es una **σ-álgebra** si cumple con las siguientes propiedades:

(1) $\mathcal{F} \neq \emptyset$

(2) $\mathsf{A} \in \mathcal{F} \Rightarrow \mathsf{A}^c \in \mathcal{F}$

(3) $\forall n \geq 1, \mathsf{A}_n \in \mathcal{F} \Rightarrow \bigcup_{n=1}^{\infty} \mathsf{A}_n \in \mathcal{F}$

Además de las σ-álgebras, precisamos del concepto de semi-anillo, el cual será de importancia para definir probabilidades.

Definición 2.2. Sea Ω un conjunto; si $\mathcal{R} \subseteq \mathcal{P}(\Omega)$ satisface:

(1) $\mathcal{F} \neq \emptyset$

(2) $\mathsf{A}, \mathsf{B} \in \mathcal{R} \Rightarrow \mathsf{A} \bigcap \mathsf{B} \in \mathcal{R}$

(3) $\forall N \geq 1, \forall 1 \leq n \leq N, \mathsf{A}_n \in \mathcal{R} \Rightarrow \bigcup_{n=1}^{N} \mathsf{A}_n \in \mathcal{R}$

entonces decimos que $\mathcal{R}$ es un **semi-anillo**.

Es decir, la última condición sobre las σ-álgebras precisa que una unión contable de conjuntos en la σ-álgebra esté también en la σ-álgebra; mientras que la propiedad para los semi-anillos requiere que la unión finita de elementos en el semi-anillos esté dentro del mismo.

Veamos algunos ejemplos:

Ejemplo 2.1. Supongamos $\Omega = \{1,2,3\}$. Entonces

- $\{\emptyset, \Omega\}$
- $\mathcal{P}(\Omega) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \Omega\}$

constituyen σ-álgebras. Más aún, para todo conjunto Ω, es fácil probar que si $\mathcal{F}$ es una σ-álgebra, entonces $\{\emptyset, \Omega\} \subseteq \mathcal{F} \subseteq \mathcal{P}(\Omega)$.

El siguiente es un resultado que se obtiene a partir de la propia definición.

Proposición 2.1. *Si $\mathcal{F}$ es una σ-álgebra de Ω, entonces:*

1. $\Omega \in \mathcal{F}$
2. $\forall n \geq 1, \mathsf{A}_n \in \mathcal{F} \Rightarrow \bigcap_{n=1}^{\infty} \mathsf{A}_n \in \mathcal{F}$
3. *Si A , $B \in \mathcal{F}$, entonces $A \setminus B \in \mathcal{F}$*

A continuación, generaremos σ-álgebras para cualquier colección de conjuntos. Es decir que si tenemos un subconjunto de $\mathcal{P}(\Omega)$, siempre es posible extenderlo para que nos represente una σ-álgebra.

Definición 2.3. Dada una σ-álgebra $\mathcal{F}$ sobre Ω; si $U \subseteq \mathcal{P}(\Omega)$ entonces la **σ-álgebra generada por** U es la menor σ-álgebra que contiene a U y la denotamos por $\sigma(U)$.

Para construir ésta σ-álgebra, consideremos todas las σ-álgebras $\{\Sigma_i\}_{i\in I}$que contienen a U (siempre existe alguna pues $\mathcal{P}(\Omega)$ es una σ-álgebra conteniendo a U). Entonces,

$$\sigma(U) = \bigcap_{i \in I} \Sigma_i$$

puesto que la intersección de una colección de σ-álgebras es una σ-álgebra.

2.2. Medidas

Definición 2.4. Sea $\mathcal{F}$ una σ-álgebra sobre Ω. Una función $\mu : \mathcal{F} \to \bar{\mathbb{R}}$ donde $\bar{\mathbb{R}} = \mathbb{R}\bigcup\{+\infty, -\infty\}$ es una **medida** si satisface:

1. $\mu(\emptyset) = 0$
2. $\forall E \in \mathcal{F}, \mu(E) \geq \emptyset$
3. Si $\{E_i\}_{i\in I}$ es una colección numerable de conjuntos disjuntos, entonces: $\mu(\biguplus_{i\in I} E_i) = \sum_{i\in I} \mu(E_i))$

El par $(\Omega, \mathcal{F})$ se llama **espacio medible** y la terna $(\Omega, \mathcal{F}, \mu)$ es un **espacio de medida**. Asimismo, los elementos de $\mathcal{F}$ son los conjuntos medibles.

Las medidas, satisfacen las siguientes propiedades:

- Si $E, F \in \mathcal{F}$ y $E \subseteq F$ entonces $\mu(E) \leq \mu(F)$ (monotonía).
- Si $E, F \in \mathcal{F}$ con $\mu(E) < \infty$, entonces $\mu(F \setminus E) = \mu(F) - \mu(E)$.
- Si $E_1, E_2, \cdots \in \mathcal{F}$ (no necesariamente disjunta), entonces $\mu(\bigcup_{i=1}^{\infty} E_i) \leq \sum_{i=1}^{\infty} \mu(E_i)$ (sub-aditividad).

Ejemplo 2.2. A continuación enumeramos algunos ejemplos clásicos de medidas:

- Si S es un conjunto arbitrario y $A \in S$ entonces $\mu(A) = |A|$ constituye una medida, donde $|A|$ es el cardinal del conjunto A.
- La medida de Dirac se define para un conjunto $S \in \Omega$ y un elemento $a \in \Omega$ como $\delta_a(S) = \chi_S(a)$; es decir, si $a \in S$ entonces la medida es 1, caso contrario es 0.
- La medida de Lebesgue en $\mathbb{R}$ constituye la manera usual de asignarle longitud, área o volumen a los subconjuntos de los espacios euclídeos. Por ejemplo, $\mu([a,b]) = b - a$ y $\mu([a,b] \times [c,d]) = (b-a)(d-c)$.

2.3. Espacios probabilistas

Ahora estamos en condiciones de presentar el concepto de espacio de probabilidad, el cual será de gran utilidad en lo que resta de este trabajo.

Definición 2.5. El triple $(\Omega, \mathcal{F}, P)$ es un espacio de probabilidad si $(\Omega, \mathcal{F})$ es un espacio medible y P constituye una medida sobre tal espacio, siendo $P(\Omega) = 1$. Llamamos al conjunto Ω el **espacio muestral**, los elementos de $\mathcal{F}$ son los **eventos** y P es la **probabilidad**.

Ejemplo 2.3. Consideremos el experimento de arrojar dos monedas; el resultado puede ser modelado por:

$$\Omega = \{C, S\}$$

$$\mathcal{F} = \mathcal{P}(\Omega)$$

$$P(\emptyset) = 0 \qquad P(\{C\}) = \frac{1}{2} \qquad P(\{C\}) = \frac{1}{2} \qquad P(\Omega) = 1$$

Incluimos la definición de las probabilidades condicionales que serán la base de la idea de modelado de sistemas (por cadenas de Markov).

Definición 2.6. Dado un espacio de probabilidad $(\Omega, \mathcal{F}, P)$ y dos eventos $A, B \in \mathcal{F}$, con $P(B) > 0$, definimos la **probabilidad condicional** de A dado B por:

$$P(A|B) = \frac{P(A \bigcup B)}{P(B)}$$

Más adelante construiremos espacios probabilistas sobre ciertos conjuntos. Pero básicamente lo haremos en subconjuntos, y luego los extenderemos. Es por ello que precisamos del siguiente teorema, que nos permitirá lograrlo.

Teorema 2.2. *Teorema de extensión de Carathéodory. Si Ω es un conjunto, $\mathcal{R}$ un semi-anillo (en Ω), y $\mu : \mathcal{R} \to [0,1]$ una medida de probabilidad en el conjunto $\mathcal{R}$, entonces existe una única medida μ' en $\sigma(\mathcal{R})$ tal que $\mu'(A) = \mu(A)$ para todo $A \in \mathcal{R}$.*

Agregamos otros conceptos necesarios para los capítulos siguientes:

Definición 2.7. Una **variable aleatoria** X es una función $X : \Omega \to Y$ donde (Y, Σ) es un conjunto medible y X es $(\mathcal{F}, \Sigma)$-medible. Diremos que X es una **variable aleatoria discreta** si Y es un conjunto numerable.

Ejemplo 2.4. Los siguientes son algunas variables aleatorias:

- Teniendo en cuenta el ejemplo 2.3, podemos definir la variable X como:
$$X(x) = \begin{cases} 1 & x = C \\ 0 & x = S \end{cases}$$
- Si en lugar de arrojar una moneda, consideramos el experimento de arrojar dos monedas: $\Omega = \{CC, CS, SC, SS\}$ podemos tomar como variable aleatoria $X =$ número de veces que obtenemos "cara", es decir,
$$X(x) = \begin{cases} 2 & x = CC \\ 1 & x \in \{CS, SC\} \\ 0 & x = SS \end{cases}$$

Definición 2.8. Un **proceso estocástico** se define como una familia de variables aleatorias $(X_t)_{t \in I}$ sobre el mismo espacio de probabilidad, donde decimos que I es el conjunto de índices.

Definición 2.9. Una **distribución de probabilidades en** S es una función $p : S \to [0,1]$ tal que $\sum_{s \in S} p(s) = 1$.

La distribución $p(s) = \begin{cases} 1 & s = s_0 \\ 0 & s \neq s_0 \end{cases}$ es denotada por 1_{s_0}.

Definición 2.10. Dados un espacio medible $(\Omega, \mathcal{F})$ y $\sigma \in \Omega^n$, definimos el **cilindro asociado a** σ como

$$Cyl(\sigma) = \{\omega \in \Omega^\omega \,|\, \forall 0 \leq i \leq n, \, \omega_i = \sigma_i\}$$

donde $\Omega^n = \Omega \times \cdots \times \Omega$; $\Omega^\omega = \Omega \times \Omega \times \ldots$ es el producto infinito y $(.)_i$ es la i-ésima proyección, es decir, si $\omega = s_0 s_1 s_2 \ldots$ entonces $\omega_i = s_i$.

Ejemplo 2.5. Para el espacio probabilista del ejemplo 2.3; tenemos que

$$(C,C,S,C,S,C,S,\dots),(C,C,S,S,S,\dots),(C,C,S,C,C,C,\dots) \in Cyl(C,C,S)$$

Notemos, que todos los elementos en $Cyl(C,C,S)$ tienen a (C,C,S) como prefijo.

Con la presentación de estos breves tópicos en probabilidad y teoría de la medida, estamos en condiciones de introducir los modelos que describirán nuestros sistemas. Si bien el capítulo contiene todos los conceptos necesarios para los posteriores, se recomiendan los siguientes libros para profundizar las nociones anteriores: [AG86], [Bre92], [WZ77].

CAPÍTULO 3

Modelado de sistemas

Como dijimos al introducir este trabajo; el Model Checking se realiza sobre ciertos sistemas que deben ser modelados. Más específicamente vamos a hacer incapié en *contraejemplos en Model Checking probabilista*. Es decir, que tales modelos de sistemas deben incluir nociones probabilísticas. Para ello emplearemos cadenas y procesos de Markov; los cuales han de ser descritos en el presente capítulo. Tales modelos constituyen la base sobre la cual trabajaremos en los capítulos posteriores.

3.1. Cadenas y Procesos de Decisión de Markov

Para las siguientes definiciones, consideremos un espacio de probabilidad $(\Omega, \mathcal{F}, P)$.

Definición 3.1. Un proceso estocástico $(X_t)_{t \in I}$ en el cual el conjunto de índices I es discreto, se denomina **Cadena de Markov discreta en el tiempo (DTMC, del inglés Discrete Time Markov Chain)**, si cumple la siguiente propiedad:

$$Pr(X_{n+1} = x \,|\, X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = Pr(X_{n+1} = x \,|\, X_n = x_n)$$

Si el rango de las variables aleatorias $\{X_i\}_{i \in I}$ es el conjunto finito S; podemos considerar la Cadena de Markov como una tupla $\mathcal{M} = (S, \mathbf{P})$ donde:

- S es el conjunto de estados; y
- $\mathbf{P} : S \times S \to [0,1]$ es una matriz estocástica; es decir, $\sum_{s' \in S} \mathbf{P}(s,s') = 1$.

Definición 3.2. Dado un conjunto de proposiciones atómicas AP, definimos **una cadena de Markov etiquetada** como una 4-upla $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ donde:

- $(S, \mathbf{P})$ es una DTMC;
- $s_{init} \in S$ es el estado inicial; y
- $L : S \to \mathcal{P}(AP)$ es la función de etiquetado.

De ahora en adelante, cuando hablemos de cadenas de Markov (o DTMC) estaremos haciendo referencia a cadenas de Markov etiquetadas, según la definición 3.2. La manera como modelamos sistemas probabilistas empleando DTMC, se puede apreciar en el siguiente ejemplo.

Ejemplo 3.1. Podemos modelar el experimento de arrojar una moneda de la siguiente manera:

$$S = \{s_0, s_1, s_2\} \qquad s_{init} = s_0$$
$$AP = \{cara, seca\}$$
$$L(s_0) = \{\neg cara, \neg seca\} \qquad L(s_1) = \{cara\} \qquad L(s_2) = \{seca\}$$

$$P = \begin{bmatrix} 0 & 0,5 & 0,5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

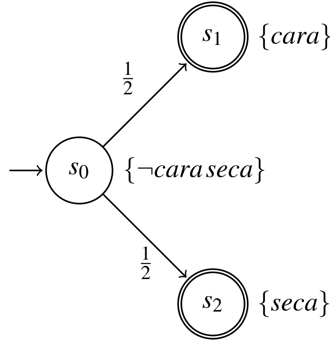

Figura 3.1: Representación gráfica de una cadena de Markov

Tal DTMC puede ser representado gráficamente como se muestra en la figura 3.1. El estado initial s_0 es apuntado por una flecha sin estado inicial. Por su parte, los estados finales (es decir, aquellos estados que tienen loops en sí mismos con probabilidad 1), se grafican con una doble línea (como en s_1 y s_2).

Estamos en condiciones de generalizar el concepto anterior.

Definición 3.3. Un **Proceso de Decisión de Markov (MDP, del inglés, Markov Decision Process** se define como una 4-upla $\mathcal{D} = (S, s_{init}, \tau, L)$

- S es el conjunto de estados;
- $s_{init} \in S$ es el estado inicial;
- $\tau : S \to \mathcal{P}(Distr(S))$ es una función que asocia a cada $s \in S$ un subconjunto (no vacío) finito de $Distr(S)$;
- $L : S \to \mathbf{P}^{AP}$ es la función de etiquetado; con AP un conjunto de proposiciones atómicas.

Ejemplo 3.2. Consideremos un protocolo de exclusión mutua aleatoria para dos procesos P_1 y P_2. El acceso a la región crítica es concedido aleatoriamente, es decir, si ambos procesos entran a su zona crítica se decide aleatoriamente cuál debe esperar, y quién puede continuar. Tal modelo puede ser implementado por un MDP.

- Cada proceso tendrá 3 estados: "no-crítico"(n); "esperando"(e) y "crítico"(c): por ende, nuestro universo es:

$$S = \{(n_1,n_2),(n_1,e_2),(n_1,c_2),(e_1,n_2),(e_1,e_2),(e_1,c_2),(c_1,n_2),(c_1,e_2)\}$$

  Notar que (c_1,c_2) no es un estado válido;

- Consideremos que (n_1,n_2) es el estado inicial;
- $\tau : S \to \mathcal{P}(Distr(S))$ está dada por:

$\tau(e_1,n_2) = \{1_{(e_1,e_2)}, 1_{(c_1,n_2)}\}$ $\quad$ $\tau(c_1,n_2) = \{1_{(c_1,e_2)}, 1_{(n_1,n_2)}\}$ $\quad$ $\tau(e_1,c_2) = \{1_{(e_1,n_2)}\}$

$\tau(n_1,e_2) = \{1_{(e_1,e_2)}, 1_{(n_1,c_2)}\}$ $\quad$ $\tau(n_1,c_2) = \{1_{(e_1,c_2)}, 1_{(n_1,n_2)}\}$ $\quad$ $\tau(c_1,e_2) = \{1_{(n_1,e_2)}\}$

$\tau(n_1,n_2) = \{1_{(e_1,n_2)}, 1_{(n_1,e_2)}\}$ $\quad$ $\tau(e_1,e_2) = \{\frac{1}{2} \times 1_{(c_1,e_2)} + \frac{1}{2} \times 1_{(e_1,c_2)}\}$

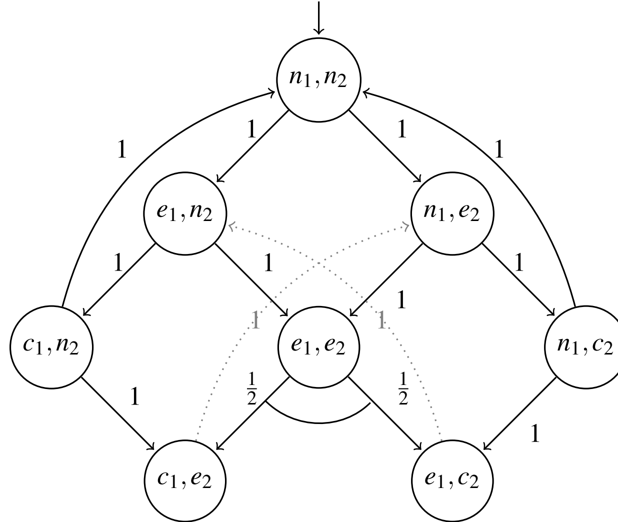

Figura 3.2: Representación gráfica de un proceso de decisión de Markov

El MDP puede representarse gráficamente como se ve en la figura 3.2. Notar la manera en que señalamos las distribuciones de probabilidades, mediante un arco (como en el estado (e_1,e_2)). En los demás estados se aprecia el no-determinismo. Las transiciones partiendo de los estados (c_1,e_2) y (e_1,c_2) se grafican en líneas de puntos grises para facilitar la lectura.

Ejemplo 3.3. En el ejemplo 3.2, todos los no-determinismos tienen probabilidad 1, obviamente no ocurre siempre, como se puede ver en el MDP de la figura 3.3.

3.2. Conversión de una DTMC en un autómata

Definición 3.4. Un autómata determinista finito se define como una 5-upla $(Q,\Sigma,\delta,q_0,Acc)$ donde

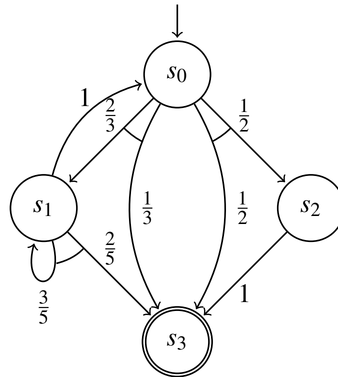

Figura 3.3: Representación gráfica de un proceso de decisión de Markov con no-determinismos probabilistas

- Q es un conjunto finito;
- Σ es un conjunto finito de símbolos (alfabeto);
- $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ es la función de transiciones de estados, con $|\delta(q,a)| \leq 1$ para cada $q \in Q, a \in \Sigma$, es decir, que para cada par estado-símbolo, existe a lo sumo una transición;
- $q_0 \in Q$ es el estado inicial;
- $Acc \subset Q$ es el conjunto de estados de aceptación.

Ejemplo 3.4. Si tomamos la 5-upla con los siguientes valores:

- $Q = \{q_0, q_1\}$;
- $\Sigma = \{a, b\}$;
- q_0 es el estado inicial;
- $Acc = \{q_0\}$;
- δ viene dado por:

$$\delta(s_0, a) = \{s_0\} \qquad \delta(s_1, a) = \emptyset$$
$$\delta(s_0, b) = \{s_1\} \qquad \delta(s_1, b) = \{s_0\}$$

Podremos graficar el autómata entonces como se aprecia en la figura 3.4.

Como se puede ver en los gráficos de los ejemplos 3.1 y 3.4, existe una relación directa entre DTMC y autómatas deterministas. Tanto es así, que la DTMC $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ pueden ser convertida en el autómata $(S', \Sigma, \hat{s}, \delta, \{t\})$ donde:

- $S' = S \bigcup \{\hat{s}\}$;
- $\Sigma \subseteq (0, 1] \times S$;

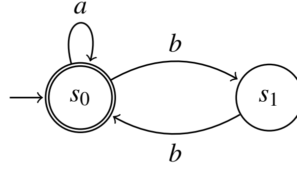

Figura 3.4: Representación gráfica de un autómata determinista finito

- $\delta : S' \times \Sigma \to S'$ está definida como: $\delta(s,(p,s')) = s'$ sii $P(s,s') = p$ y $\delta(\hat{s},(1,s_{init})) = s_{init}$.
- $\hat{s}$ es el estado inicial;
- $t \in S$ es un estado de aceptación de la DTMC (definido en 4.1).

3.3. Conversión de una DTMC en un digrafo ponderado

Para recorrer los caminos en las DTMC y encontrar sus pesos, es preferible hacerlo sobre el grafo que éste genera.

Definición 3.5. Un **digrafo ponderado** es un triple $\mathcal{G} = (V,E,w)$ donde:

- V un conjunto finito;
- $E \subseteq V \times V$ es el conjunto de aristas; y
- $w : E \to \mathbb{R}_{\geq 0}$ representa el peso de cada arista.

Un **camino** σ de u a w en tal grafo es una secuencia $\sigma = v_0 v_1 \dots v_n \in V^+$, con $v_0 = u$, $v_n = w$ y $\forall 0 \leq i < n : (v_i, v_{i+1}) \in E$.

El **peso** de un camino se define por $w(\sigma) = \sum_{i=0}^{n-1} w(v_i, v_{i+1})$.

Haciendo uso de la simple conversión propuesta en [HKD09], podemos transformar las DTMCs en digrafos mediante la siguiente definición.

Definición 3.6. Si $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ es una DTMC, el digrafo ponderado asociado a $\mathcal{M}$ es $\mathcal{G}_{\mathcal{M}} = (V,E,w)$, con $V = S$; $(v,v') \in E \Leftrightarrow \mathbf{P}(v,v') > 0$ y $w(v,v') = -log\mathbf{P}(v,v')$.

Notar que, debido a la propiedad de "la suma de los logaritmos", es posible pasar de un camino en un grafo al camino de la DTMC:

$$w(\sigma) = \sum_{i=0}^{n-1} w(v_i, v_{i+1}) = \sum_{i=0}^{n-1} -log\mathbf{P}(v_i, v_{i+1}) = -\sum_{i=0}^{n-1} log\mathbf{P}(v_i, v_{i+1})$$
$$= -log(\prod_{i=0}^{n-1} \mathbf{P}(v_i, v_{i+1})) = -log(\mathbf{P}(\sigma))$$

La última igualdad es justificada en la sección siguiente.

3.4. Caminos y ejecuciones

Como hemos visto, es posible visualizar una DTMC por medio de un grafo. Es así como naturalmente surge el concepto de camino en un grafo, el cual conlleva a la definición de ejecuciones.

Definición 3.7. Sea $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ una DTMC. Para cada $s \in S$, definimos:

$$Paths(\mathcal{M}, s) = \{s_0 s_1 s_2 \dots \in S^\omega \mid s_0 = s \wedge \forall n \in \mathbb{N}, \mathbf{P}(s_n, s_{n+1}) > 0\}$$
$$Paths_{fin}(\mathcal{M}, s) = \{s_0 s_1 s_2 \dots s_n \in S^* \mid s_0 = s \wedge \forall 0 \leq i < n, \mathbf{P}(s_i, s_{i+1}) > 0\}$$

Tales nociones también se extienden a un MDP a continuación.

Definición 3.8. Sea $\mathcal{D} = (S, s_{init}, \tau, L)$ un MDP; para cada $s \in S$, definimos:

$$Paths(\mathcal{D}, s) = \{s_0 s_1 s_2 \dots \in S^\omega \mid s_0 = s \wedge \forall n \in \mathbb{N}, \exists \pi \in \tau(s_n) : \pi(s_{n+1}) > 0\}$$
$$Paths_{fin}(\mathcal{D}, s) = \{s_0 s_1 s_2 \dots s_n \in S^* \mid s_0 = s \wedge \forall 0 \leq i < n, \exists \pi \in \tau(s_n) : \pi(s_{n+1}) > 0\}$$

Para cada elemento $\sigma \in Paths_{fin}(\mathcal{M}, s)$, $|\sigma|$ denota el tamaño de σ; σ_i es el i-ésimo elemento de σ y $(\sigma \uparrow i)$ es el camino a partir del i-ésimo elemento (si $\sigma = s_0 s_1 \dots$, entonces $(\sigma \uparrow i) = s_i s_{i+1} \dots$).

Para poder asignarle probabilidades a las ejecuciones, debemos introducir la σ-álgebra que determinarán los eventos de nuestro espacio probabilístico.

Definición 3.9. Dado $s \in S$, para cada $\sigma \in Paths(\mathcal{M}, s)$ tenemos el cilindro (ver definición 2.10) asociado a σ:

$$Cyl(\sigma) = \{\omega \in Paths(\mathcal{M}, s) \mid w_0 w_1 \dots w_{|\sigma|} = \sigma_0 \sigma_1 \dots \sigma_{|\sigma - 1|}\}$$

El conjunto de tales cilindros lo denotamos por $\mathsf{Cyl}_{\mathcal{M}}$:

$$\mathsf{Cyl}_{\mathcal{M}} = \{Cyl(\sigma) \mid \sigma \in Paths_{fin}(\mathcal{M}, s)\}$$

Estamos en condiciones de asignarle una medida de probabilidad a los cilindros:

Definición 3.10. Dado un conjunto de cilindros $\mathsf{Cyl}_{\mathcal{M}}$, definimos $\mu : \mathsf{Cyl}_{\mathcal{M}} \to [0,1]$ como:

$$\mu(Cyl(\sigma)) = \prod_{i=0}^{|\sigma|-2} \mathbf{P}(\sigma_i, \sigma_{i+1})$$

Podemos así generar la σ-álgebra de ejecuciones:

Definición 3.11. Sea $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ una DTMC. La σ-álgebra de ejecuciones de $\mathcal{M}$ será $\sigma(\mathsf{Cyl}_{\mathcal{M}})$, la cual denotamos como $\mathcal{F}_{\mathcal{M}}$.

Como $\mathsf{Cyl}_{\mathcal{M}}$ es un semi anillo, por 2.2, la medida de probabilidad μ sobre $\mathsf{Cyl}_{\mathcal{M}}$ (definida en 3.10) se extiende únicamente sobre $\mathcal{F}_{\mathcal{M}}$. Por lo tanto, para cada $s \in S$, $(Paths_{fin}(\mathcal{M}, s), \mathcal{F}_{\mathcal{M}}, \mu)$ constituye un espacio de probabilidad.

Debido a la presencia de no-determinismo para el caso de los MDP, no es posible generar un espacio de probabilidad sobre el conjunto $Paths(\mathcal{D})$ considerando la σ-álgebra de ejecuciones $\mathcal{F}_{\mathcal{D}}$. Pero para poder asignarles probabilidades, a cada evento $\Delta \in \mathcal{F}_{\mathcal{D}}$ le definimos probabilidades máximas y mínimas (las cuales se denotan como $\mu^+(\Delta)$ y $\mu^-(\Delta)$ respectivamente).

En general, como veremos más adelante, estaremos interesados en encontrar contraejemplos para problemas de alcanzabilidad (es decir, para modelos en los que se busca alcanzar ciertos estados), es por ello que presentamos las siguientes definiciones:

Definición 3.12. Diremos que un estado $s \in S$ es absorbente si $\mathcal{P}(s,s) = 1$.

Definición 3.13. Los conjuntos de caminos de alcanzabilidad de un conjunto $A \subseteq S$ se definen como:

$$Reach(\mathcal{D},s,A) = \{\omega \in Paths(\mathcal{D},s) \mid \exists i \geq 0 : w_i \in A\}$$

$$Reach_{fin}(\mathcal{D},s,A) = \{\sigma \in Paths_{fin}(\mathcal{D},s) \mid \sigma_{|\sigma|-1} \in A \wedge \forall i \leq \sigma_{|\sigma|-2} : \sigma_i \notin A\}$$

A continuación, definiremos el concepto de “scheduler”, el cual nos facilitará definir formalmente μ^+ y μ^-.

Definición 3.14. Dado un MDP $\mathcal{D} = (S, s_{init}, \tau, L)$ y $s \in S$, un **scheduler probabilista** η en $\mathcal{D}$ es una función $\eta : Paths_{fin}(\mathcal{D}, s_{init}) \rightarrow Distr(\mathcal{P}(Distr(S)))$ que satisface $\eta(\sigma) \in Distr(\tau(\sigma_{|\sigma|-1}))$, es decir,

$$\eta(\sigma) = p \Rightarrow (\textstyle\sum_{q \in \tau(\sigma_{|\sigma|-1})} p(q) = 1)$$

Básicamente, los schedulers asignan a cada elemento en un Proceso de Decisión de Markov una distribución de probabilidades; eliminando así el no-determinismo. Esto se aprecia en el ejemplo siguiente.

Ejemplo 3.5. Considerando el MDP del ejemplo 3.2, un scheduler podría estar dado por la distribución uniforme en los estados no-deterministas; es decir, que si $\sigma_{|\sigma|-1} \neq (e_1, e_2)$, entonces se le asigna a ambas distribuciones del estado, una probabilidad de $\frac{1}{2}$. Para el estado (e_1, e_2), como no tenemos elecciones, la distribución será la misma. A partir de tal scheduler, obtenemos el grafo de la figura 3.5.

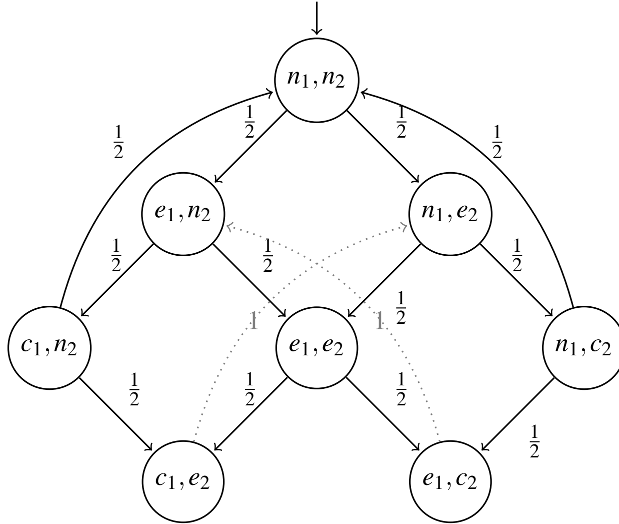

Figura 3.5: Representación gráfica de la resolución de un scheduler sobre el proceso de decisión de Markov de la figura 3.2

Suponiendo que tenemos $\sigma \in Paths_{fin}(\mathcal{D}, s_{init})$, si el sistema se rige por el comportamiento del scheduler η, entonces en el estado $\sigma_{|\sigma|-1}$, la probabilidad de que el estado siguiente sea s (con $s \in S$) está dada por:

$$\sum_{\pi \in \tau(\sigma_{|\sigma|-1})} (\eta(\sigma)(\pi)).\pi(s)$$

Por ende, estamos obteniendo una medida asociada al scheduler:

Definición 3.15. Si $\mathcal{D} = (S, s_{init}, \tau, L)$ es un MDP y η es un scheduler en $\mathcal{D}$, entonces definimos la medida de probabilidad μ_η como la única medida generada en $\mathcal{F}_\mathcal{D}$ tal que si $s_0 s_1 \ldots s_n \in Paths_{fin}(\mathcal{D}, s_{init})$ entonces

$$\mu_\eta(\langle s_0 s_1 \ldots s_n \rangle) = \prod_{i=0}^{n-1} \sum_{\pi \in \tau(s_i)} (\eta(s_0 s_1 \ldots s_i)(\pi)).\pi(s_{i+1})$$

Definición 3.16. Sea $\mathcal{D} = (S, s_{init}, \tau, L)$ un MDP y $\Delta \in \mathcal{B}_\mathcal{D}$. Las probabilidades maximal y minimal de Δ se definen como:

$$\mu^+(\Delta) = \sup_{\eta \in Sched(\mathcal{D})} \mu_\eta(\Delta)$$

$$\mu^-(\Delta) = \text{ínf}_{\eta \in Sched(\mathcal{D})} \mu_\eta(\Delta)$$

En el trabajo de Luca de Alfaro [De 98], se demuestra que para calcular alcanzabilidad (las ejecuciones que llegan a ciertos estados) existen ciertos tipos de schedulers que determinan las probabilidades máximas y mínimas del MDP. A continuación daremos algunos conceptos con el objetivo de presentar estos resultados.

Definición 3.17. Un scheduler η es **determinista** si $\forall \sigma \in Paths_{fin}(\mathcal{D}, s_{init}), \forall \pi \in \tau(\sigma_{|\sigma|-1})$, $\eta(\sigma)(\pi) \in \{0,1\}$. Diremos que el scheduler **no tiene memoria**, si $\forall \sigma_1, \sigma_2 \in Paths_{fin}(\mathcal{D}, s_{init})$, $\sigma_{1|\sigma_1|-1} = \sigma_{2|\sigma_2|-1} \Rightarrow \eta(\sigma_1) = \eta(\sigma_2)$.

Definición 3.18. Si $\mathcal{D} = (S, s_{init}, \tau, L)$ es un MDP y η es un scheduler determinista y sin memoria, entonces podemos construir la **DTMC η-asociado a** $\mathcal{D}$ de la siguiente manera: $\mathcal{D}_\eta = (S, s_{init}, \mathbf{P}_\eta, L)$ donde para cada $s, t \in S$, la matriz de transición de estados está dada por $\mathbf{P}_\eta(s,t) = (\eta(s))(t)$

CAPÍTULO 4

Propiedades a verificar

Como ya lo hemos mencionado; el model checking se basa en la verificación de una propiedad sobre un modelo. En el siguiente apartado, establecemos una definición formal de la lógica con la cual describiremos las especificaciones a ser verificadas; nos estamos refiriendo a la lógica LTL. La lógica temporal lineal (LTL) contiene modalidades referidas al tiempo, que nos permitirán expresar requerimientos sobre ejecuciones en los modelos.

4.1. Lógica Temporal Lineal LTL

A los operadores de la lógica usual, le añadiremos nuevos operadores que dan nociones de evolución en el tiempo. Así, veremos que por $\bigcirc\varphi$ queremos indicar que φ es válida en un estado posterior al actual, $\Box\varphi$ ocurre cuando φ es verdadera en todos los estados y $\Diamond\varphi$ denota que eventualmente (en un estado futuro) la fórmula φ será válida.

A su vez, a partir de estos operadores es posible definir dos nuevos operadores binarios **U** y **R** que se usarán para denotar la validez de un par de estados o alguno de ellos en un lapso de tiempo.

Cuando la semántica sea presentada, todos estos conceptos serán formalmente introducidos.

4.1.1. Sintaxis

Las fórmulas LTL se pueden definir inductivamente de la siguiente manera:

(1) $\top$ es una fórmula.

(2) Toda proposición atómica es una fórmula.

(3) Si φ es un fórmula, entonces $\neg\varphi$ es una fórmula.

(4) Si φ y ϑ son fórmulas, entonces $\varphi \wedge \vartheta$ y $\varphi \vee \vartheta$ son fórmulas LTL.

(5) Si φ es una fórmula, entonces $\bigcirc\varphi$, $\Box\varphi$ y $\Diamond\varphi$ son fórmulas (se pueden encontrar en la literatura como $\mathbf{X}\varphi$: de ne**X**t, $\mathbf{G}\varphi$: de **G**lobally y $\mathbf{F}\varphi$: de **F**inally, respectivamente).

(6) Si φ y ϑ son fórmulas, entonces $\varphi\mathbf{U}\vartheta$ (por **U**ntil) y $\varphi\mathbf{R}\vartheta$ (de **R**elease) son fórmulas.

Nota. Es posible también definir las fórmulas LTL por medio de la gramática:

$$\varphi ::= \top \,|\, AP \,|\, \neg\varphi \,|\, \varphi\wedge\varphi \,|\, \bigcirc\varphi \,|\, \varphi\mathbf{U}\varphi$$

Donde AP es el conjunto de proposiciones atómicas. Los demás operadores son syntax sugar y pueden definirse a partir de los anteriores:

- $\varphi \vee \vartheta = \neg(\neg\varphi \wedge \neg\vartheta)$
- $\varphi \Rightarrow \vartheta = \neg\varphi \vee \vartheta$
- $\varphi \Leftrightarrow \vartheta = (\varphi \Rightarrow \vartheta) \wedge (\vartheta \Rightarrow \varphi)$
- $\Diamond\varphi = \top\mathbf{U}\varphi$
- $\Box\varphi = \neg(\Diamond\neg\varphi)$
- $\varphi\mathbf{R}\vartheta = \neg(\neg\varphi\mathbf{U}\neg\vartheta)$

4.1.2. Semántica

Antes de introducir formalmente la semántica, aclaremos el significado de las fórmulas. Las fórmulas serán válidas sobre ejecuciones $\sigma = s_0 s_1 \ldots$. Diremos que vale $\bigcirc\varphi$ si φ es válida en el estado siguiente; es decir, s_1; $\Box\varphi$ vale si φ se satisface en todos los estados s_i; y $\Diamond\varphi$ significa que hay algún estado en el que vale φ.

Sea $\mathcal{D} = (S, s_{init}, \tau, L)$ un MDP y $\sigma \in Paths(\mathcal{D}, s)$, con $s \in S$ y sea $AP = \{p_1, p_2, \ldots, p_n\}$ el conjunto de fórmulas atómicas. Definimos la noción de validez de φ en σ con el símbolo $\sigma \vDash_{\mathcal{D}} \varphi$:

1. $\sigma \vDash_{\mathcal{D}} \top$
2. $\sigma \vDash_{\mathcal{D}} p_i$ sii $p_i \in L(\sigma_0)$
3. $\sigma \vDash_{\mathcal{D}} \neg\varphi$ sii $\sigma \nvDash_{\mathcal{D}} \varphi$
4. $\sigma \vDash_{\mathcal{D}} (\varphi \Rightarrow \vartheta)$ sii $(\sigma \vDash_{\mathcal{D}} \neg\varphi) \vee (\sigma \vDash_{\mathcal{D}} \vartheta)$
   $\sigma \vDash_{\mathcal{D}} (\varphi \Leftrightarrow \vartheta)$ sii $(\sigma \vDash_{\mathcal{D}} \varphi \Rightarrow \vartheta) \wedge (\sigma \vDash_{\mathcal{D}} \vartheta \Rightarrow \varphi)$
5. $\sigma \vDash_{\mathcal{D}} \varphi \wedge \vartheta$ sii $\sigma \vDash_{\mathcal{D}} \varphi$ y $\sigma \vDash_{\mathcal{D}} \vartheta$
   $\sigma \vDash_{\mathcal{D}} \varphi \vee \vartheta$ sii $\sigma \vDash_{\mathcal{D}} \varphi$ o $\sigma \vDash_{\mathcal{D}} \vartheta$
6. $\sigma \vDash_{\mathcal{D}} \bigcirc\varphi$ sii $\sigma \uparrow 1 \vDash_{\mathcal{D}} \varphi$
   $\sigma \vDash_{\mathcal{D}} \Box\varphi$ sii $\forall i \geq 0, \sigma \uparrow i \vDash_{\mathcal{D}} \varphi$
   $\sigma \vDash_{\mathcal{D}} \Diamond\varphi$ sii $\exists i \geq 0, \sigma \uparrow i \vDash_{\mathcal{D}} \varphi$
7. $\sigma \vDash_{\mathcal{D}} (\varphi\mathbf{U}\vartheta)$ sii $\exists i \geq 0 \,|\, [(\forall j \,|\, 0 \leq j < i, (\sigma \uparrow j \vDash_{\mathcal{D}} \varphi)) \wedge (\sigma \uparrow i \vDash_{\mathcal{D}} \vartheta)]$
   $\sigma \vDash_{\mathcal{D}} (\varphi\mathbf{R}\vartheta)$ sii $(\forall i \geq 0, \sigma \uparrow i \vDash_{\mathcal{D}} \vartheta) \vee$
   $\exists i \geq 0 \,|\, [(\forall j \,|\, 0 \leq j < i, (\sigma \uparrow j \vDash_{\mathcal{D}} \vartheta)) \wedge (\sigma \uparrow i \vDash_{\mathcal{D}} (\vartheta \wedge \varphi))]$

Podemos explicar las últimas fórmulas gráficamente de la siguiente manera:

$\bigcirc\varphi$	La fórmula φ vale en el siguiente estado de la secuencia.	φ $\circ\!\rightarrow\circ\rightarrow\ldots\rightarrow\circ\rightarrow\circ\rightarrow\ldots$
$\Box\varphi$	La fórmula φ vale en todos los estados de la secuencia.	$\varphi \quad \varphi \qquad \varphi \quad \varphi$ $\circ\rightarrow\circ\rightarrow\ldots\rightarrow\circ\rightarrow\circ\rightarrow\ldots$
$\Diamond\varphi$	φ es válida en alguno de los estados de la secuencia.	φ $\circ\rightarrow\circ\rightarrow\ldots\rightarrow\circ\rightarrow\circ\rightarrow\ldots$
$\varphi\mathbf{U}\vartheta$	Vale φ hasta que en algún estados vale ϑ.	$\varphi \quad \varphi \qquad \varphi \quad \vartheta$ $\circ\rightarrow\circ\rightarrow\ldots\rightarrow\circ\rightarrow\circ\rightarrow\ldots$
$\varphi\mathbf{R}\vartheta$	Vale ϑ hasta que en algún estado valen también vale φ. O, si nunca ocurre φ, siempre vale ϑ.	$\vartheta \quad \vartheta \qquad \vartheta \quad \vartheta,\varphi$ $\circ\rightarrow\circ\rightarrow\ldots\rightarrow\circ\rightarrow \quad \circ \quad \rightarrow\ldots$ O bien $\vartheta \quad \vartheta \qquad \vartheta \quad \vartheta$ $\circ\rightarrow\circ\rightarrow\ldots\rightarrow\circ\rightarrow\circ\rightarrow\ldots$

Para los enunciados siguientes, consideramos un MDP $\mathcal{D} = (S, s_{init}, \tau, L)$.

Definición 4.1. El lenguaje $Sat_{\mathcal{D}}(\varphi)$ asociado a una fórmula LTL φ es el conjunto de caminos que satisfacen φ:

$$Sat_{\mathcal{D}}(\varphi) = \{\sigma \in Paths(\mathcal{D}) \mid \sigma \vDash_{\mathcal{D}} \varphi\}$$

Abusaremos de la notación, y diremos que $Sat_{\mathcal{D}}(\vartheta)$, con ϑ una fórmula proposicional, es el conjunto de los estados que satisfacen ϑ:

$$Sat_{\mathcal{D}}(\vartheta) = \{s \in S \mid s \vDash \vartheta\}$$

donde en la última fórmula, $\vDash$, define la semántica usual de la lógica proposicional.

Definición 4.2. Siendo φ una fórmula LTL y un número $p \in [0,1]$, definimos $\vDash_{\leq p}$ y $\vDash_{\geq p}$ por:

$$\mathcal{D} \vDash_{\leq p} \varphi \Leftrightarrow \mu^{+}_{\mathcal{D}}(Sat(\varphi)) \leq p$$

$$\mathcal{D} \vDash_{\geq p} \varphi \Leftrightarrow \mu^{+}_{\mathcal{D}}(Sat(\varphi)) \geq p$$

Análogamente se definen $\vDash_{<p}$ y $\vDash_{>p}$.

Nuevamente, abusaremos de la notación para expresar tal definición, pero para fórmulas proposicionales ϑ.

Habiendo ya introducido los mecanismos para modelar sistemas y propiedades a verificar, podemos realizar model checking sobre los mismos y a partir de ellos, obtener los contraejemplos que son de nuestro mayor interés. En los capítulos siguientes, centraremos nuestra atención en ello.

CAPÍTULO 5

Expresiones Regulares

Las expresiones regulares serán de nuestra utilidad para describir conjuntos de ejecuciones (que surgen de los distintos caminos que se pueden obtener de un grafo). A continuación, procedemos a definirlas.

El conjunto Σ es un conjunto finito de símbolos o caracteres, que constituye un *alfabeto*. Usualmente consideramos alfabetos sobre $a, b, c, \ldots$; es decir $\Sigma = \{a, b, c, \ldots\}$ (con $|\Sigma| = n$). A partir de ellos, podemos construir palabras como una secuencia de elementos de Σ; ab, ε, $baccb$ son algunos ejemplos (ε es la palabra que no tiene ningún elemento). Dadas dos palabras A, B sobre Σ, la concatenación entre ambas se denota por AB, y se obtiene agregando los elementos de B a la derecha de A.

Un lenguaje formal sobre Σ, $\mathcal{L}(\Sigma)$, es un conjunto de palabras sobre Σ. Tenemos entonces, que $\emptyset$ es un lenguaje vacío, $\Sigma^n = \{$palabras en Σ, con longitud n $\}$ donde $n \geq 0$. $\Sigma^+ = \bigcup\limits_{n \geq 1} \Sigma^n$ y $\Sigma^* = \{\varepsilon\} \bigcup \Sigma^+$. Estos últimos conjuntos son llamados la clausura positiva de Σ y la clausura de Kleene, respectivamente.

Definición 5.1. Siendo $\Sigma = \{a, b, c, \ldots\}$ un alfabeto, las siguientes son *expresiones regulares* sobre Σ (definidas recursivamente):

- (Caso Base) ε, a, b, $\ldots$ son expresiones regulares.
- (Recursión) si r y s son expresiones regulares, entonces las siguientes también lo son:
    - la concatenación: rs;
    - la unión: $r + s$;
    - la clausura de Kleene: r^*.
- Un string es una expresión regular si y sólo si se obtiene aplicando las reglas anteriores.

El orden de precedencia es, de mayor a menor: "()", clausura de Kleene, concatenación, unión.

5.1. Propiedades de expresiones regulares

Las expresiones regulares pueden ser reducidas aplicando las siguientes propiedades:

- la unión es conmutativa: $r+s=s+r$;
- la unión es asociativa: $(r+s)+t=r+(s+t)=r+s+t$;
- la concatenación es asociativa: $(rs)t=r(st)=rst$;
- la concatenación es distributiva sobre la unión: $(r+s)t=rt+st$ y $r(s+t)=rs+rt$;
- ε es el elemento neutro para la concatenación:$r\varepsilon=\varepsilon r=r$
- la clausura de Kleene es idempotente:$(r^*)^*=r^*$

5.2. Conversión de autómatas a expresiones regulares

En el artículo de Christoph Neumann [Neu05] se describen 3 métodos para la conversión de autómatas deterministas finitos en expresiones regulares. Aquí, los presentaremos y centraremos la atención en uno de ellos, que usaremos posteriormente.

5.2.1. Método de Eliminación de Estados

Este método se basa en la reducción del número de estados del autómata. En primera instancia, se eliminan los estados que son inalcanzables, es decir, aquellos estados que no son alcanzados por un camino desde el estado inicial.

Luego, se van eliminando los estados de a uno, lo que se hace modificando las transiciones, las cuales se convierten en expresiones regulares. Se buscan patrones en el autómata como el subgrafo de la figura 5.1.

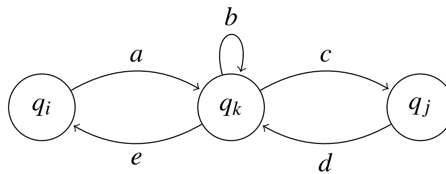

Figura 5.1: Subgrafo a procesar - Método de Eliminación de Estados

A partir de ese entonces, es posible eliminar el estado intermedio (q_k), haciendo uso de expresiones regulares, obteniendo el grafo de la figura 5.2.

El proceso se repite hasta, finalmente, llegar a un autómata conteniendo sólo un estado inicial y uno de aceptación, de la forma de la figura 5.3. Para éste, obtenemos la expresión regular $r_1^* r_2 (r_4 + r_3 r_1^* r_2)^*$.

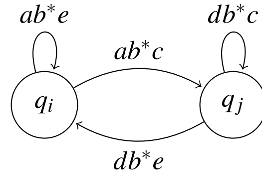

Figura 5.2: Eliminación del estado q_k - Método de Eliminación de Estados

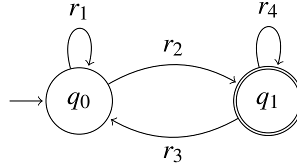

Figura 5.3: Forma final - Método de Eliminación de Estados

5.2.2. Método Algebraico

También conocido como método de Brzozowski, se basa en la resolución de un sistema de ecuaciones lineales.

La idea es la de generar una variable para cada estado del autómata y luego resolver para R_{init} (variable asociada al estado q_{init}).

Si $Q = \{q_{init}, q_1, q_2, \ldots, q_N\}$ y $a_{i,j}$ es la transición entre q_i y q_j, entonces el término será $a_{i,j}R_j$ y si R_i es un estado de aceptación, entonces agregamos un término extra ε. Obtenemos así un sistema de ecuaciones:

$$R_{init} = a_{init,init}R_{init} + a_{init,1}R_1 + ... + a_{init,N}R_N \; R_1 = a_{1,init}R_{init} + a_{1,1}R_1 + ... + a_{1,N}R_N \; \ldots \; R_N = a_{N,init}R_{init} + a_{N,1}R_1 + ... + a_{N,N}R_N + \varepsilon$$

considerando que q_N es el estado final.

El sistema se resuelve por sustitución, pero cuando tenemos variables desconocidas de ambos lados, se usa el teorema de Arden [1].

5.2.3. Método de Clausura Transitiva

El primer método presentado (eliminación de estados) es el que visualmente resulta más sencillo, pero su implementación es complicada e implica la “destrucción” del autómata. Por tal motivo, no consideramos que fuese el apropiado para nuestros objetivos. Por su parte, el método algebraico explota cuando el número de estados es grande. Así, finalmente, el método de la clausura transitiva es el que se usará en el presente trabajo. Su debilidad es que las expresiones regulares que se obtienen son más largas que para los otros métodos.

Para explicarlo, consideremos $Q = \{q_1, q_2, \ldots, q_N\}$. Con $R_{i,j}$ denotamos la expresión regular que representa las transiciones de q_i a q_j. Además, supongamos que $R^k_{i,j}$ son las transiciones de q_i a q_j pasando sólo por los estados $\{q_1, q_2, \ldots, q_k\}$. $R^k_{i,j}$ se puede definir como:

$$R^k_{i,j} = R^{k-1}_{i,k}(R^{k-1}_{k,k})^* R^{k-1}_{k,j} + R^{k-1}_{i,j}$$

[1] La ecuación $X = AX + B$ donde $\varepsilon \notin A$ puede ser resuelta con $X = A^*B$

donde $\emptyset^* = \varepsilon$ y

$$R^0_{i,j} = \begin{cases} R_{i,j} & i \neq j \\ R_{i,j} + \varepsilon & i = j \end{cases}$$

Es decir, $R^k_{i,j}$ se obtiene de $R^{k-1}_{i,j}$, agregándole las transiciones de q_i a q_k, de q_k en sí mismo y de q_k a q_j.

Veamos un ejemplo con un autómata pequeño.

Ejemplo 5.1. Sea el autómata el descripto en la figura 5.4. Tenemos entonces las siguientes expresiones iniciales:

$$\begin{array}{lll} R^0_{0,0} = a + \varepsilon & R^0_{0,1} = b & R^0_{0,2} = c \\ R^0_{1,0} = \emptyset & R^0_{1,1} = \varepsilon & R^0_{1,2} = d \\ R^0_{2,0} = \emptyset & R^0_{2,1} = \emptyset & R^0_{2,2} = \varepsilon \end{array}$$

En la primera iteración obtenemos:

$$\begin{array}{lll} R^1_{0,0} = a + \varepsilon & R^1_{0,1} = b + b & R^1_{0,2} = cd + c \\ R^1_{1,0} = \emptyset & R^1_{1,1} = \varepsilon & R^1_{1,2} = d + d \\ R^1_{2,0} = \emptyset & R^1_{2,1} = \emptyset & R^1_{2,2} = \varepsilon \end{array}$$

Para finalizar, calculemos sólo expresión regular que modela las transiciones del estado inicial q_0 al de aceptación q_2:

$$R^2_{0,2} = (cd + d) + ((a + \varepsilon)(a + \varepsilon)^*(a + \varepsilon) + (a + \varepsilon))$$

Obviamente, tal expresión se puede reducir usando las propiedades de las mismas.

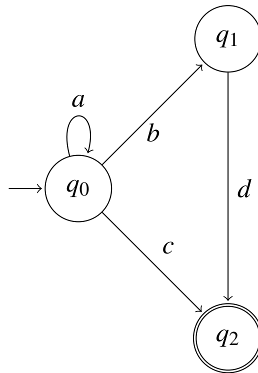

Figura 5.4: Método de Clausura Transitiva

5.2.4. Reducción de Expresiones Regulares

Como ya hemos visto, los operadores sobre expresiones regulares tiene ciertas propiedades. Si utilizamos esas propiedades de una manera inteligente, es posible reducirlas. Por reducirlas nos referimos a obtener una expresión regular equivalente, con alguna característica en particular (no necesariamente que su longitud sea menor).

Las propiedades que usaremos en este trabajo para reducir serán:

- $(r^*)^* \leftrightarrow r^*$
- $(\prod_{1 \leq i \leq k} r_i \; \varepsilon \prod_{k+1 \leq i \leq n} r_i)^* \leftrightarrow (\prod_{1 \leq i \leq n} r_i)$
- $\varepsilon + \varepsilon \leftrightarrow \varepsilon$
- $\sum_{1 \leq i \leq k} r_i + \varepsilon + \sum_{k+1 \leq i \leq n} r_i \leftrightarrow \sum_{1 \leq i \leq n} r_i + \varepsilon$
- $(\sum_i r_i)^* \leftrightarrow (\prod_i r_i^*)^*$
- $r(s+t) \leftrightarrow rs + rt$
- $(r+s)t \leftrightarrow rt + st$

Tales propiedades de reducción fueron pensadas con los objetivos que describimos a continuación.

Las primeras tres reglas permiten obtener la menor cantidad de términos posibles en la expresión resultante. Con ellas, básicamente estamos eliminando las clausuras de Kleene extras o los ε de más.

La cuarta regla sólo pretende hacer más ordenado el resultado, llevando los ε al final de la expresión.

Por último, las 3 reglas finales tienen por propósito "sacar" las uniones de las expresiones regulares. Nos estamos refiriendo a obtener expresiones regulares de la forma $\sum_{1 \leq i \leq n} r_i$ donde para $1 \leq i \leq n$, r_i es una expresión regular que no tiene operadores "unión".

Habiendo ya introducido las expresiones regulares, veremos en los capítulos siguientes el porqué de la aplicación de las reglas enumeradas, y los motivos por los que extraer la unión es útil para nuestros casos de estudio.

CAPÍTULO 6

Contraejemplos

En este capítulo, presentamos los conceptos de contraejemplos (base para este trabajo), y la manera de construirlos para cadenas de Markov discretas en el tiempo, empleando propiedades LTL. En el caso de que el modelo sea descripto por un proceso de decisión de Markov, veremos (en el apéndice A) que el mismo puede reducirse a encontrar contraejemplos en una DTMC.

Definición 6.1. Dado un MDP $\mathcal{D} = (S, s_{init}, \tau, L)$, una propiedad φ en la lógica LTL y un número $p \in [0,1]$; un **contraejemplo** a $\mathcal{D} \vDash_{\leq p} \varphi$ es un conjunto medible $\mathcal{C} \subseteq Sat(\varphi)$ que satisface $\mu^+(\mathcal{C}) > p$. Del mismo modo, un contraejemplos a $\mathcal{D} \vDash_{<p} \varphi$ será deberá satisfacer $\mu^+(\mathcal{C}) \geq p$. Para el caso de $\mathcal{D} \vDash_{\geq p} \varphi$ y $\mathcal{D} \vDash_{>p} \varphi$, usamos la negación de los casos anteriores: $\mathcal{D} \vDash_{\leq 1-p} \neg\varphi$ y $\mathcal{D} \vDash_{<1-p} \neg\varphi$ respectivamente.

Vale la pena aclarar que, si bien uno se ve tentado a pensar que $\mathcal{D} \vDash_{\geq p} \varphi$, es equivalente a $\mu^-(\mathcal{C}) < p$, tal resultado no es válido, puesto que siempre $\emptyset \subseteq Sat(\varphi)$ lo satisface trivialmente.

De ahora en adelante, sólo consideraremos la generación de contraejemplos para $\mathcal{D} \vDash_{<p} \varphi$, puesto que para los otros 3 casos posibles, el desarrollo es completamente similar. Asimismo, podemos suponer que la fórmula LTL será del tipo $\Diamond\varphi$, por el siguiente teorema de [De 98]:

Teorema 6.1. *Si $\mathcal{D} = (S, s_{init}, \tau, L)$ es un MDP y φ una fórmula LTL, entonces existe una fórmula del tipo $\Diamond\vartheta$ con ϑ una fórmula proposicional (i.e. sin operadores temporales) tal que*

$$\mathcal{D} \vDash_{\leq p} \varphi \Leftrightarrow \mathcal{D}' \vDash_{\leq p} \Diamond\vartheta$$

donde $\mathcal{D}'$ es un MDP construido a partir de $\mathcal{D}$.

Nota. No daremos detalles respecto al anterior teorema porque significaría apartarnos demasiado del presente trabajo. Sólo agregaremos que $\mathcal{D}'$ es un MDP el cual es calculado a partir de la composición de $\mathcal{D}$ con un autómata (en realidad, un autómata de Rabin) que se genera a partir de φ.

Resumiendo, de ahora en adelante, para buscar contraejemplos a cualquier fórmula LTL ϑ, resolveremos el problema de alcanzabilidad $\mathcal{D} \vDash_{\leq p} \Diamond\vartheta$ donde $\mathcal{D}$ es un MDP y φ es una fórmula proposicional.

6.1. Contraejemplos Representativos

Como hemos definido los contraejemplos como conjuntos medibles cuya probabilidad satisface una condición respecto con un número, tales conjuntos podrían ser completamente estrafalarios. Por ende, los acotaremos a través de las siguientes definiciones extraídas de [HKD09].

Definición 6.2. Si $\mathcal{D}$ es un MDP y ϑ una fórmula proposicional, un **contraejemplo representativo** a $\mathcal{D} \vDash_{<p} \Diamond\vartheta$ es un conjunto $\mathcal{C} \subseteq Reach_{fin}(\mathcal{D}, s_{init}, Sat(\varphi))$ que satisface $\mu^{+}(Cyl(\mathcal{C})) \geq p$. El conjunto de contraejemplos representativos será denotado por $CR(\mathcal{D}, p, \vartheta)$.

Lo que se pretende hacer con la definición anterior, es obtener un conjunto cuyo conjunto generado sea verdaderamente el contraejemplo visible como ejecuciones del sistema.

En las siguientes definiciones, suponemos que $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ es una DTMC, ϑ una fórmula proposicional y $p \in [0,1]$.

Definición 6.3. Un **contraejemplo mínimo** es un conjunto $\mathcal{C} \in CR(\mathcal{M}, p, \vartheta)$ que satisface $|\mathcal{C}| \leq |\mathcal{C}'|$ para todo $\mathcal{C}' \in CR(\mathcal{M}, p, \vartheta)$.

Definición 6.4. Si $\mathcal{C} \in CR(\mathcal{M}, p, \vartheta)$ es un contraejemplo mínimo y $\forall \mathcal{C}' \in CR(\mathcal{M}, p, \vartheta)$, $\mu(Cyl(\mathcal{C})) \geq \mu(Cyl(\mathcal{C}'))$, entonces tal contraejemplo es el **más indicado**.

Si bien hemos reducido la noción de contraejemplo; aún obtenemos conjuntos que son poco relevantes a la hora de describir contraejemplos "debuggeables". Es por ello que agregamos la noción de testigos, que serán de importancia en lo que viene.

Definición 6.5. Dado un MDP $\mathcal{D}$, una fórmula proposicional ϑ, $p \in [0,1]$ y $\mathcal{C} \in CR(\mathcal{M}, p, \vartheta)$. Los elementos de una partición de $\mathcal{C}$ serán los **testigos**.

La idea para particionar los contraejemplos representativos es la de obtener un número finito de testigos, cuyos elementos provean información similar y que los diferentes testigos presenten diferente información a la hora de debuggear.

6.2. Rails y torrentes

En la presente sección, fijamos en el problema de que los contraejemplos son conjuntos muy generales que aportan información muy amplia, muchas veces imposible de ser empleada para debugging. Entonces, nos fijaremos como objetivo generar testigos de contraejemplos que contengan información diferente. Pretendemos con ello agrupar de alguna manera la información, así los testigos son más acotados y entre un par de ellos aporten datos complementarios. Esto será factible porque dos testigos contendran caminos que difieran fuera de los ciclos del modelo. Para ello, abstraeremos los ciclos, y obtendremos caminos acíclicos. Tales caminos generarán los testigos a los contraejemplos. En esta sección, $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ siempre será una DTMC.

Recordemos el concepto de conexión en grafos. Un subconjunto $K \subseteq S$ es una componente conexa si satisface que para todo $s, t \in S$, hay un camino finito que los conecta. Tal subconjunto es una componente fuertemente conexa (SCC, del inglés strongly connected component) si es maximal (es decir, si K' es componente fuertemente conexa y

$K \bigcap K' \neq \emptyset$ entonces $K' \subseteq K$). SCC^* se utilizará para referirse a las componentes fuertemente conexas no triviales (es decir, aquellas componentes con más de un estado, o con sólo un estado el cual no es absorbente) y si $s \in K$, donde K es un elemento de SCC^*, entonces definimos $SCC^*_s = K$.

A continuación presentamos los conceptos necesarios que posteriormente usaremos con el objetivo de abstraer (es decir, eliminar preservando los valores de probabilidad) los ciclos generados en los caminos de las componentes fuertemente conexas (que no proveen información adicional al momento de debuggear).

Definición 6.6. Para cada $K \subseteq SCC^*$, los conjuntos de entrada y salida a K, se definen respectivamente como:

$$Inp_K = \{k \in K \mid \exists s \in (S-K) \,:\, \mathbf{P}(s,k) > 0\}$$

$$Out_K = \{s \in (S-K) \mid \exists k \in K \,:\, \mathbf{P}(k,s) > 0\}$$

Los elementos del conjunto de entrada de una componente fuertemente conexa están dentro de la misma componente, mientras que los elementos de salida no son elementos de la componente; es decir, si $K \subseteq SCC^*$: $Inp_K \subseteq K$ y $Out_K \bigcap K = \emptyset$. Esto se aprecia en el siguiente ejemplo.

Ejemplo 6.1. Consideremos la DTMC de la figura 6.1.

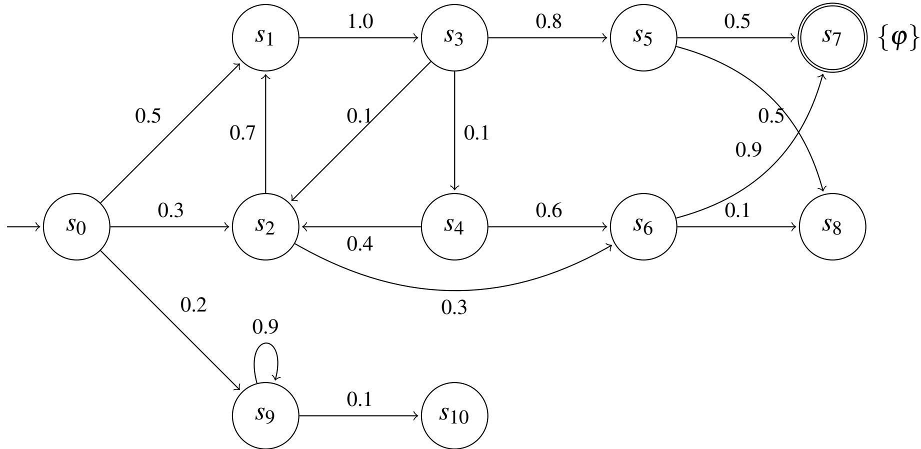

Figura 6.1: DTMC

En la figura 6.2, podemos apreciar las componentes fuertemente conexas, sus entradas y salidas.

Ahora, podremos restringir las cadenas de Markov para cada componentes fuertemente conexa.

Definición 6.7. Para cada $K \in SCC^*$, podemos definir la cadena de Markov restringida a K como: $\mathcal{M}_K = (K \bigcup Out_K, s_K, \mathbf{P}_K, L_K)$ donde s_K es un elemento de Inp_K; $L_K = L|_{K \bigcup Out_K}$; y

$$\mathbf{P}_K(s,t) = \begin{cases} \mathbf{P}(s,t) & s \in K \\ 1 & s \in Out_K, s = t \\ 0 & s \in Out_K, s \neq t \end{cases}$$

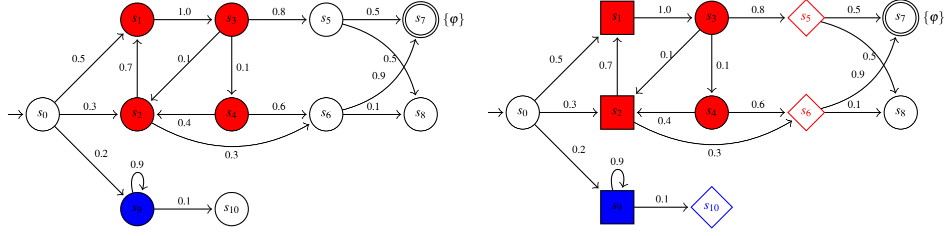

Figura 6.2: Las dos componentes fuertemente conexas en rojo y azul. A la derecha, los cuadrados representan estados de entrada; los rombos las salidas

Ejemplo 6.2. Teniendo en cuenta la DTMC del ejemplo 6.1, para cada una de las componentes fuertemente conexas, obtendríamos las cadenas de Markov restringidas que se aprecian en figura 6.3.

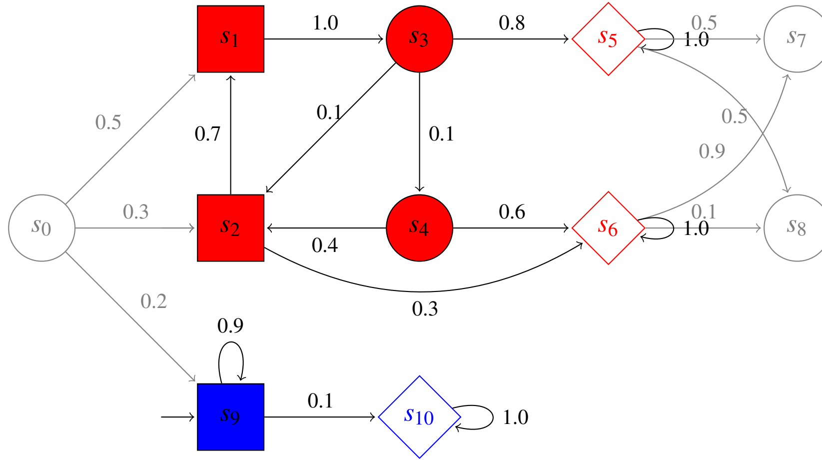

Figura 6.3: DTMC restringidas a componentes fuertemente conexas

Ahora, podemos obtener una cadena de Markov, eliminando los ciclos (SCCs):

Definición 6.8. La cadena de Markov acíclica asociada a $\mathcal{M}$, es $Ac(\mathcal{M}) = (S', s_{init}, \mathbf{P'}, L')$, donde si llamamos $S_{com} = S \setminus \bigcup_{K \in SCC^*} K$ y $S_{inp} = \bigcup_{K \in SCC^*} Inp_K$ entonces

$$S' = (S \setminus S_{com}) \bigcup S_{inp}$$

$$L' = L|_{S'}$$

$$\mathbf{P}_K(s,t) = \begin{cases} \mathbf{P}(s,t) & s,t \in S_{com} \\ \mu(Reach(\mathcal{M}_{SCC_s^*}, s, \{t\})) & s \in S_{inp} \wedge t \in Out_{\mathcal{M}_{SCC_s^*}} \\ 1_s & s \in S_{inp} \wedge Out_{\mathcal{M}_{SCC_s^*}} = \emptyset \\ 0 & \text{caso contrario} \end{cases}$$

Nota. En la definición anterior, para el cálculo de $\mu(Reach(\mathcal{M}_{SCC_s^*}, s, \{t\}))$ se puede emplear la técnica de steady-state analysis, definida a continuación.

Definición 6.9. Diremos que la DTMC $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ es **reducible** si su grafo no es fuertemente conexo. Es decir, si existe un estado $s \in S$ que no sea "alcanzable" por otro estado $t \in S$, o si s no "alcanza" t.

Para tales DTMC reducibles [Geb08], es posible calcular las probabilidades de alcanzar ciertos estados, a partir de lo que se conoce en la literatura como análisis "steady state". Pero antes introducimos otro concepto necesario.

Definición 6.10. Siendo $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ una DTMC y $s \in S$, decimos que s es un **estado de transición** si existe una componente conexa $K \subset S$ y $t \in S$ tal que $s \in K, t \notin K$ y $\mu(Reach(\mathcal{M}, s, \{t\})) > 0$

Teorema 6.2. *Si $\mathcal{M} = (S, s_{init}, \boldsymbol{P}, L)$ es una DTMC reducible y si $s, t \in S$, donde $s \in U$ (con $U \subset S$ el conjunto de estados de transición y t es absorbente, entonces podemos calcular $x_{s,t} \doteq \mu(Reach(\mathcal{M}, s, \{t\}))$ por:*

$$x_{s,t} = P(s,t) + \sum_{u \in U} P(s,u) x_{u,t}$$

Ejemplo 6.3. Para la DTMC de 6.1, para obtener las probabilidades de llegar a los estados de salida, resolvemos los siguientes sistemas de ecuaciones:

- $\mathbf{P}_K(s_1, s_5) = \frac{400}{451}$ y $\mathbf{P}_K(s_2, s_5) = \frac{280}{451}$, resolviendo para x_1 y x_2:

$$x_1 = x_3 \qquad x_2 = 0.7x_1$$
$$x_3 = 0.1x_2 + 0.1x_4 + 0.8 \qquad x_4 = 0.4x_2$$

- $\mathbf{P}_K(s_1, s_6) = \frac{51}{451}$ y $\mathbf{P}_K(s_2, s_5) = \frac{171}{451}$, resolviendo para x_1 y x_2:

$$x_1 = x_3 \qquad x_2 = 0.7x_1 + 0.3$$
$$x_3 = 0.1x_2 + 0.1x_4 \qquad x_4 = 0.4x_2 + 0.6$$

- $\mathbf{P}_K(s_9, s_{10}) = 1.0$, resolviendo para x_9:

$$x_9 = 0.9x_9 + 0.1$$

Al eliminar las SCCs, obtenemos la siguiente cadena de Markov de la figura 6.4.

Definición 6.11. Un **rail** de $\mathcal{M}$ es un camino finito $\sigma \in Paths_{fin}(Ac(\mathcal{M}))$. El conjunto de rails se denota por $Rails(\mathcal{M})$.

Los rails en $\mathcal{M}$, son representaciones de caminos truncados de $\mathcal{M}$, puesto que en $Ac(\mathcal{M})$ hemos eliminado las SCCs (en realidad sólo dejamos las entradas a tales SCCs). Como tenemos en mente obtener contraejemplos para $\mathcal{M}$ (y no para $Ac(\mathcal{M})$); debemos establecer una manera de generar caminos en $\mathcal{M}$ que sean los representativos de los rails. La idea es la de agregar a cada rail, los elementos en las SCCs de una manera ordenada.

Siguiendo ese objetivo, introducimos el término "torrente" que será la extensión de los rails en $\mathcal{M}$. Pero antes veamos cómo crearlo.

Diremos que ω es una **suprasecuencia** de $\sigma \in Rails(\mathcal{M})$ si existe una función creciente $f : [\![0, (|\sigma| - 1)]\!] \to [\![0, (|\omega| - 1)]\!]$ tal que $\forall\, 0 \leq i < |\sigma|$, $\sigma_i = \omega_{f(i)}$.

Podemos refinar la noción de suprasecuencia añadiendo propiedades:

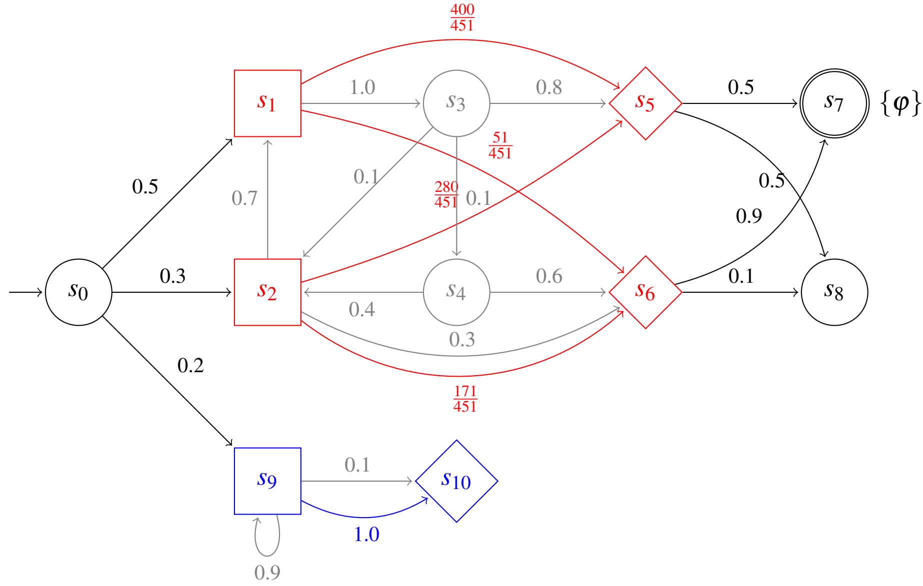

Figura 6.4: DTMC

(1) Si $\sigma_i \in K$, donde K es una SCC; entonces $(\forall 0 \leq j < f(i), \omega_j \notin K)$. Es decir, σ_i es el primer estado de ω en la SCC K.

(2) Si $\sigma_i \in K$, donde K es una SCC; entonces $(\forall f(i) \leq j < f(i+1), \omega_j \in K)$. Lo cual se interpreta como que σ y ω "salen" de la SCC K de manera conjunta.

Definición 6.12. Si $\sigma \in Rails(\mathcal{M})$, entonces el conjunto de suprasecuencias de σ que satisfacen las propiedades anteriores es el torrente asociado a σ. Es decir,

$$Torr(\mathcal{M}, \sigma) \doteq \{\omega \in Paths(\mathcal{M}) | \omega \text{ es subsecuencia de } \sigma \text{ y satisface 1 y 2}\}$$

Ejemplo 6.4. Consideremos un rail σ y un elemento ω del torrente asociado a σ para la DTMC del ejemplo 6.1. Tales, pueden representarse gráficamente de la siguiente manera:

$$\begin{array}{lllllllllllllllll}
\sigma & = & s_0 & \rightarrow & s_2 & \rightarrow & & & & & & & s_6 & \rightarrow & s_7 & & \\
\omega & = & s_0 & \rightarrow & s_2 & \rightarrow & s_1 & \rightarrow & s_3 & \rightarrow & s_4 & \rightarrow & s_6 & \rightarrow & s_7 & \rightarrow & \ldots
\end{array}$$

Esto se representa gráficamente en la figura 6.5

Ahora, recordamos un teorema de [ADR09] que establece que la probabilidad de los rails es igual a la probabilidad de su torrente, y por ende, la información sobre contraejemplos que nos brinda un rail será tan útil como la de su torrente asociado.

Teorema 6.3. *Siendo $\mathcal{M}$ una DTMC, y $\sigma \in Rails(\mathcal{M})$, entonces tenemos:*

$$\mu_{Ac(\mathcal{M})}(Cyl(\sigma)) = \mu_{\mathcal{M}}(Torr(\mathcal{M}, \sigma))$$

En el siguiente teorema, se expresa un concepto que será clave para establecer la relación entre rails y testigos.

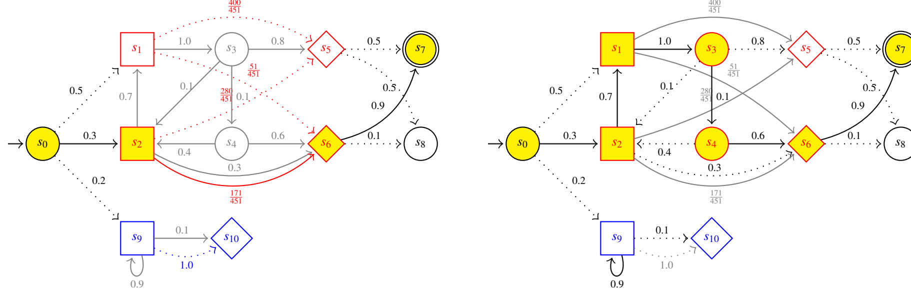

Figura 6.5: Un rail a la izquierda y un elemento de su torrente asociado a la derecha

Definición 6.13. Para cada $\sigma \in Rails(\mathcal{M})$, definimos el conjunto de **generadores de torrentes** como:

$$GenTorr(\mathcal{M}, \sigma) = \{\omega \in Paths_{fin}(\mathcal{M}) \,|\, Cyl(\omega) \text{ es torrente asociado a } \sigma\}$$

De esta definición se desprende que para cada $\sigma \in Rails(\mathcal{M})$ (contraejemplo a $Ac(\mathcal{M}) \vDash_{\leq p} \Diamond\vartheta$), el conjunto $GenTorr(\mathcal{M}, \sigma)$ constituye un testigo (ver definición 6.5); y por ende, la unión $\bigcup_{\omega \in GenTorr(\mathcal{M},\sigma)} Cyl(\omega)$ es un contraejemplo representativo a $\mathcal{M} \vDash_{\leq p} \Diamond\vartheta$.

6.3. Contraejemplos

Todavía tenemos un problema que resolver: si bien hemos visto cómo eliminar las componentes fuertemente conexas, tenemos que asegurarnos no estamos perdiendo información importante. Como los torrentes se construyen a partir de rails, estos últimos no deberían modificarse al eliminar las SCCs (es decir, no deberíamos eliminar rails que alcancen estados que satisfagan ϑ).

Para ello, haremos que los estados que satisfagan ϑ sean absorbentes, y los estados a partir de los cuales no se pueda alcanzar ϑ también lo serán.

Definición 6.14. Si $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ es una DTMC y ϑ una fórmula proposicional, definimos $\mathcal{M}_\vartheta = (S, s_{init}, \mathbf{P}_\vartheta, L)$ como:

$$\mathbf{P}_\vartheta(s,t) = \begin{cases} 1 & s \in Sat(\vartheta) \wedge s = t \\ 1 & s \notin Sat_\Diamond(\vartheta) \wedge s = t \\ \mathbf{P}(s,t) & s \in Sat_\Diamond(\vartheta) \setminus Sat(\vartheta) \\ 0 & \text{caso contrario} \end{cases}$$

donde $Sat_\Diamond(\vartheta) = \{s \in S \,|\, \mu(Reach(\mathcal{M}, s, Sat(\vartheta))) > 0\}$, es decir, el conjunto de los estados de S desde los que se puede alcanzar un estado que satisfaga ϑ.

A continuación, como se refleja en [ADR09], veremos la relación que existe entre los caminos y probabilidades de las DTMC $\mathcal{M}$, $\mathcal{M}_\vartheta$ y $Ac(\mathcal{M})$.

Teorema 6.4. *Sea $\mathcal{M} = (S, s_{init}, \boldsymbol{P}, L)$ y ϑ una fórmula proposicional. Para cada rail de $\mathcal{M}_\vartheta$ $\sigma \in Paths_{fin}(\mathcal{M}_\vartheta)$ son válidas:*

(1) $Reach_{fin}(\mathcal{M}_\vartheta, s_{init}, Sat(\vartheta)) = Reach_{fin}(\mathcal{M}, s_{init}, Sat(\vartheta))$

(2) $\mu_{\mathcal{M}_\vartheta}(Cyl(\sigma)) = \mu_{\mathcal{M}}(Cyl(\sigma))$

(3) $GenTorr(\mathcal{M}_\vartheta, \sigma) = GenTorr(\mathcal{M}, \sigma)$

(4) $\mu_{\mathcal{M}_\vartheta}(Torr(\mathcal{M}_\vartheta, \sigma)) = \mu_{\mathcal{M}}(Torr(\mathcal{M}, \sigma))$

(5) $\mu_{AC(\mathcal{M}_\vartheta)}(Cyl(\sigma)) = \mu_{\mathcal{M}}(Torr(\mathcal{M}, \sigma))$

(6) Para cada $p \in [0,1]$*,* $Ac(\mathcal{M}_\vartheta) \vDash_{\leq p} \Diamond\varphi$ *si y sólo si* $\mathcal{M} \vDash_{\leq p} \Diamond\varphi$

Extenderemos los conceptos de contraejemplos representativos, contraejemplos mínimos y contraejemplos más indicados a una DTMC acíclica $Ac(\mathcal{M})$.

Definición 6.15. Sea p $\in [0,1]$ tal que $\mathcal{M} \vDash_{\leq p} \Diamond\vartheta$. Si $\mathcal{C}$ es un contraejemplo representativo a $Ac(\mathcal{M}) \vDash_{\leq p} \Diamond\vartheta$, entonces definimos

$$TorRepCount(\mathcal{C}) = \{GenTorr(\mathcal{M}, \sigma) \mid \sigma \in \mathcal{C}\}$$

Tal conjunto será denominado contraejemplo torrente de $\mathcal{C}$; y denotaremos por $CR_t(\mathcal{M}, p, \vartheta)$ el conjunto de todos los contraejemplos torrentes a $\mathcal{M} \vDash_{\leq p} \Diamond\vartheta$.

Los contraejemplos torrentes se relacionan con los contraejemplos representativos, por medio del siguiente teorema.

Teorema 6.5. *Sean* $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ *es una DTMC,* ϑ *una fórmula proposicional y* $p \in [0,1]$*, tal que* $\mathcal{M} \nvDash_{\leq p} \Diamond\vartheta$*. Tomemos* $\mathcal{C}$ *un contraejemplo representativo a* $Ac(\mathcal{M}_\vartheta) \vDash_{\leq p} \Diamond\vartheta$*. El conjunto de caminos* $\biguplus_{W \in TorRepCount(\mathcal{C})} W$ *es un contraejemplo representativo de* $\mathcal{M} \vDash_{\leq p} \Diamond\vartheta$*.*

Nota. Como para cada $\sigma \in \mathcal{C}$ tenemos un testigo $GenTorr(\mathcal{M}, \sigma)$ y el número de rails es finito; entonces tendremos un número finito de testigos.

Definición 6.16. Diremos que $\mathcal{C}_t \in CR_t(\mathcal{M}, p, \vartheta)$ es un mínimo contraejemplo torrente si $|\mathcal{C}_t| \leq |\mathcal{C}'_t|$, para toda $\mathcal{C}'_t \in CR_t(\mathcal{M}, p, \vartheta)$.

Definición 6.17. Sean $\mathcal{M} = (S, s_{init}, \mathbf{P}, L)$ es una DTMC, ϑ una fórmula proposicional y $p \in [0,1]$. Diremos que $\mathcal{C}_t \in CR_t(\mathcal{M}, p, \vartheta)$ es el contraejemplo torrente más indicado si es un mínimo contraejemplo torrente y $\mu(\bigcup_{W \in \mathcal{C}_t} Cyl(W)) \geq \mu(\bigcup_{W \in \mathcal{C}'_t} Cyl(W))$ para todos los contraejemplos torrentes mínimos $\mathcal{C}'_t \in CR_t(\mathcal{M}, p, \vartheta)$.

Por el teorema 6.4, es posible obtener el contraejemplo torrente más indicado de una DTMC $\mathcal{M}$ obteniéndolo a partir del de $Ac(\mathcal{M})$.

CAPÍTULO 7

Implementación

En el presente capítulo resumiremos la manera en que la técnica presentada anteriormente fue implementada en la herramienta PRISM (que presentamos a continuación).

7.1. PRISM

PRISM (por sus siglas en inglés **Pr**obabilist**i**c **S**ymbolic **M**odel Checker), es una herramienta (de software libre) de model checking probabilista sobre la que hemos implementado las técnicas aquí descriptas. La herramienta original está documentada en [HKNP06].

PRISM, originalmente fue diseñado para soportar propiedades en lógica PCTL (Probabilistic Computation Tree Logic) que se ejecutan en modelos en DTMC y MDP y en la lógica CSL (Continuous Stochastic Logic) para CTMC, cadenas de Markov de tiempo continuo, (no hemos introducido el tema en el presente trabajo). Las últimas versiones de PRISM incluyen propiedades LTL (de nuestro interés), por el aporte del trabajo de [Bed07].

Básicamente, la totalidad de PRISM está escrita en Java, salvo los algoritmos que requieren mayores recursos, implementados en C, que corren por medio de Java Native Interface. Para la construcción y el almacenamiento de las DTMC, MDP y cadenas de Markov de tiempo continuo se emplean MTBDD (multi-terminal binary decision diagrams) por medio de una aplicación llamada CUDD [Som01] (Colorado University Decision Diagram).

7.2. Arquitectura de PRISM

La herramienta recibe dos entradas, a saber: un modelo y una propiedad a verificar. El modelo se describe en el Lenguaje PRISM, cuya sintaxis y semántica están definidas en el documento [HKNP06]. PRISM parsea la descripción, y crea un modelo del tipo apropiado, es decir, una DTMC, un MDP o una CTMC y determina el conjunto de

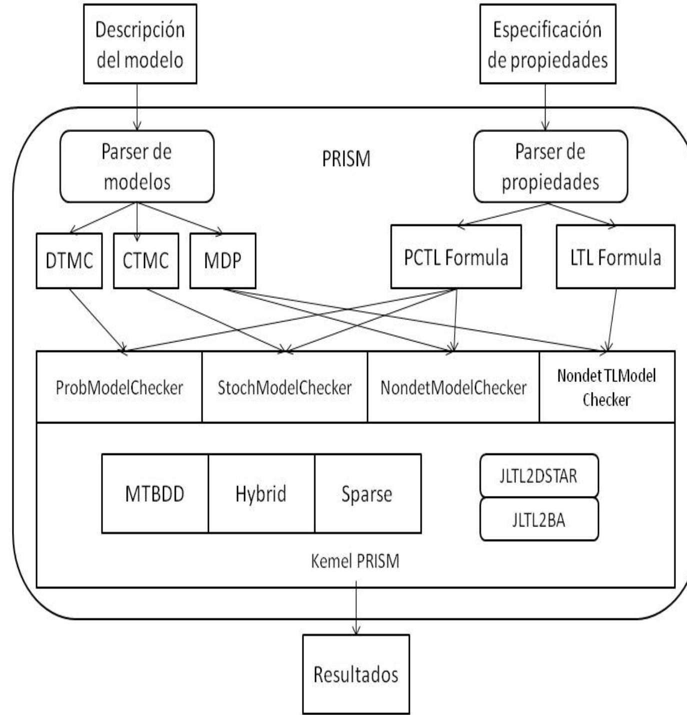

Figura 7.1: Arquitectura de PRISM

estados alcanzables que tienen. Las propiedades consisten de fórmulas PCTL, LTL o CSL, según corresponda. Éstas se parsean y a posteriori se hace el model checking invocando al model checker del modelo en cuestión.

Para los cálculos probabilísticos del model checker, se pueden emplear tres métodos: uno simbólico, basado en los MTBDD (de CUDD); uno explícito, empleando "sparse matrices" (matrices ralas, una forma vectorial de representar las matrices) y uno híbrido, el cual fue la novedad implementada en PRISM.

En la figura 7.1, se añade la arquitectura de PRISM (de las versiones que incluyen lógica LTL de [Bed07]).

7.3. Algoritmo de generación de contraejemplos

Una vez que PRISM determina que una fórmula es inválida, es decir, ya habiendo realizado model checking una vez; se procede al cálculo del contraejemplo. El hecho de haber realizado model checking en el sistema, implica que el problema ya ha sido reducido a uno de alcanzabilidad. Si el modelo de entrada es un MDP, se extrae la DTMC problemática a través del scheduler que maximiza la propiedad. La técnica que se implementa está descripta en el apéndice A.

Como ya se ha calculado $\{s \in S \mid \mu(Reach(\mathcal{M}, s, Sat(\vartheta))) > 0\}$ (durante la fase del model checking), es posible eliminar los estados que no alcanzarán estados satisfaciendo ϑ. Para eso, reducimos nuestra cadena de Markov aplicando la definición de 6.14. Así, reducimos el conjunto de estados a $Sat(\vartheta) \bigcup Sat_{\diamond}(\vartheta)$.

El siguiente paso, consiste en encontrar y eliminar las SCCs. Para ello, existen varios algoritmos que encuentran las componentes fuertemente conexas para representaciones simbólicas de grafos (MTBDD en nuestro caso). El que se usó aquí, es el algoritmo de Xie-Beerel (descripto en [Bed07]).

Una vez identificadas las SCCs, se computan los estados de entrada y salida de 6.6 y se reduce la cadena de Markov a una cadena acíclica según la definición 6.8. Notar que para hacerlo, primero empleamos "steady state analysis" para el cálculo de las probabilidades dentro de cada componente fuertemente conexa (vista como las DTMC de definición 6.7). Los algoritmos numéricos de Jacobi y Gauss-Seidel para la resolución de sistemas de ecuaciones lineales, ya fueron implementados en PRISM para el cálculo probabilístico.

Sólo resta encontrar los caminos desde los estados iniciales a aquellos estados en los que se alcanza la propiedad ϑ. Como se hiciera en [HKD09], se convierte la DTMC en un digrafo ponderado, con la idea de cambiar el problema de encontrar los caminos finitos con mayor probabilidad, en un problema de camino más corto (SPP, del inglés "Shortest Path Problem"). El beneficio que presenta esta aproximación, es que SPP es un problema que ha sido muy estudiado y se han obtenido interesantes algoritmos ([Epp98]). David Eppstein mantiene una lista de algoritmos de SPP que se puede encontrar en la web. En este trabajo, se usó la implementación de [JM99], puesto que en la práctica tiene mejores tiempos de ejecución.

De ahora en adelante, llamaremos al método recientemente descripto como **método de los Rails**.

7.4. Descripción de las SCCs

Si bien los rails pueden ser de gran ayuda para obtener e imaginar los contraejemplos, presentan el problema que no son completos en cuanto a que no describen a la totalidad de los caminos; para eso estarían los torrentes. Ahora, nuestro problema radica en encontrar una manera precisa de caracterizar a los torrentes.

El mayor inconveniente que se nos presenta es que debemos describir las componentes fuertemente conexas (SCCs). Intentar hacerlo mediante caminos, sería poco muy inteligente, por la cantidad que obtendríamos. Por eso, naturalmente se podría pensar en el empleo de expresiones regulares para hacerlo.

De todas maneras, las expresiones regulares pueden llegar a ser incomprensibles para hacer debugging (en general, una fórmula de la forma $(r^* + s)^*(r + s)$ donde r y s son expresiones regulares, es difícil de verla como un progreso). Por ello, suponemos que la reducción de la expresión regular sería de gran utilidad.

Ejemplo 7.1. Para verlo en un caso concreto, nuevamente consideremos el ejemplo 6.1 y el rail σ del ejemplo 6.4:

$$\sigma = s_0 \rightarrow s_2 \rightarrow \qquad\qquad s_6 \rightarrow s_7$$

Entonces, podríamos aplicar la reducción de las expresiones regulares a la SCC que

contiene a S_2, para llegar a una expansión de la siguiente forma:

$$
\begin{array}{l}
((s_1s_3)) \\
+ \\
(s_1s_3(s_2s_1s_3)^*) \\
+ \\
(s_1s_3(s_2s_1s_3)^*(s_2s_1s_3)) \\
+ \\
(s_1s_3(s_2s_1s_3)^*s_4(s_2s_1s_3(s_2s_1s_3)^*s_4)^*(s_2s_1s_3)) \\
+ \\
(s_1s_3(s_2s_1s_3)^*s_4(s_2s_1s_3(s_2s_1s_3)^*s_4)^*(s_2s_1s_3(s_2s_1s_3)^*)) \\
+ \\
(s_1s_3(s_2s_1s_3)^*s_4(s_2s_1s_3(s_2s_1s_3)^*s_4)^*(s_2s_1s_3(s_2s_1s_3)^*(s_2s_1s_3))))
\end{array}
$$

Con lo cual, algunos de los elementos en el torrente asociado a σ son:

$$
\begin{array}{lllllclllllll}
\omega & = & s_0 & \rightarrow & s_2 & \rightarrow & s_1s_3 & \rightarrow & s_6 & \rightarrow & s_7 & \rightarrow & \ldots \\
\omega & = & s_0 & \rightarrow & s_2 & \rightarrow & s_1s_3(s_2s_1s_3)^* & \rightarrow & s_6 & \rightarrow & s_7 & \rightarrow & \ldots \\
\omega & = & s_0 & \rightarrow & s_2 & \rightarrow & s_1s_3(s_2s_1s_3)^*(s_2s_1s_3) & \rightarrow & s_6 & \rightarrow & s_7 & \rightarrow & \ldots \\
\omega & = & s_0 & \rightarrow & s_2 & \rightarrow & s_1s_3(s_2s_1s_3)^*s_4(s_2s_1s_3(s_2s_1s_3)^*s_4)^*(s_2s_1s_3) & \rightarrow & s_6 & \rightarrow & s_7 & \rightarrow & \ldots \\
\omega & = & s_0 & \rightarrow & s_2 & \rightarrow & s_1s_3(s_2s_1s_3)^*s_4(s_2s_1s_3(s_2s_1s_3)^*s_4)^*(s_2s_1s_3(s_2s_1s_3)^*) & \rightarrow & s_6 & \rightarrow & s_7 & \rightarrow & \ldots \\
\omega & = & s_0 & \rightarrow & s_2 & \rightarrow & s_1s_3(s_2s_1s_3)^*s_4(s_2s_1s_3(s_2s_1s_3)^*s_4)^*(s_2s_1s_3(s_2s_1s_3)^*(s_2s_1s_3)) & \rightarrow & s_6 & \rightarrow & s_7 & \rightarrow & \ldots
\end{array}
$$

Este método será llamado **método de los Rails con expresiones regulares reducidas** (y usaremos la expresión "Rails + ERR" para referirnos a él).

7.5. Expresiones Regulares puras

Otra alternativa que se implementa es la de obtener la expresión regular (mediante el método de la Clausura Transitiva 5.2.3). A priori se supone que será completamente ineficaz en cuanto a los contraejemplos que se generan, lo cual se verá en los casos de estudio. El método de contraejemplos será llamado **expresiones regulares** (ER).

7.6. Expresiones Regulares reducidas

Finalmente, consideramos una modificación al método anterior. Una vez que se obtiene la expresión regular, se emplean las técnicas de reducción sobre todo las expresiones regulares que se obtienen con el método anterior, para así conseguir la unión de expresiones regulares que no tendrán operadores de unión. A este método lo llamaremos de **expresiones regulares reducidas** (ERR).

7.7. Implementación de los métodos de contraejemplos

Estamos en condiciones de describir la implementación realizada. Para ello nos basaremos en la arquitectura de la versión de PRISM que permite el model checking de propiedades LTL.

Nos basaremos en la figura 7.1 que refleja la arquitectura de esa versión, para adaptarla a nuestra implementación. Recordemos que los contraejemplos están definidos sólo para cadenas de Markov y procesos de decisión de Markov sobre fórmulas LTL.

Para procesar la fórmula, se crea una clase LTLFormula que almacena la fórmula en cuestión; al mismo tiempo se crea el modelo de DTMC o MDP según corresponda. Las propiedades LTL se convierten a un autómata de Rabin determinista (mediante las herramientas ltl2dstar y LTL2BA) y compuestas con el modelo para crear el producto que será el nuevo modelo para hacer model checking.

Se concluye haciendo el model checking propiamente dicho usando el motor de MTBDDs implementado por PRISM (descripto en [Par02]). A partir de ese entonces se calcula el resultado. En el caso de tener un modelo de MDP, se extraerá la DTMC que maximiza la probabilidad (con el algoritmo del apéndice A); esto se hace al mismo tiempo en que se hace el model checking, ya que significa un ahorro de tiempo de procesamiento.

Como nuestra atención se centra en los contraejemplos, si el resultado que obtenemos es falso, entonces obtenemos el contraejemplo. En la figura 7.2 describimos el proceso para cada método.

Si pretendemos hacer uso de los rails (ya sea mediante el método de rails o mediante los rails con expresiones regulares reducidas), entonces inicialmente se identifican las SCCs con el algoritmo de Xie-Beerel. En ese momento, estamos en condiciones de obtener la cadena de Markov acíclica (6.8); para ello, para cada par de elementos en los conjuntos de entrada y salida asociados a una componente conexa, hacemos uso del método de Gauss-Seidel para resolver los sistemas de ecuaciones lineales.

La cadena de Markov acíclica previamente obtenida está representada por un MTBDD. Llega el momento de buscar los K caminos más cortos. Hacerlo sobre la representación simbólica es sumamente trabajoso; inclusive no se han encontrado trabajos al respecto. Se optó entonces por hacerlo sobre una representación implícita de la cadena de Markov (transformada en un grafo). Por ende, se recorre todo el MTBDD y se va creando tal grafo.

Mediante la adaptación del algoritmo de Eppstein por Jimenez y Marzal, se obtienen los rails; es decir, tenemos nuestro primer método para analizar contraejemplos.

Si en cambio, pretendemos realizar un análisis exhaustivo; entonces deberemos expandir las SCCs. Para ello, una vez encontrados los rails, les añadimos las expresiones regulares generadas por los grafos de las SCCs, reducidas extrayendo las uniones.

Para la obtención de contraejemplos por el método de las expresiones regulares y el de las expresiones regulares reducidas, procedemos de la siguiente manera: de la cadena de Markov que se usó para el model checking, transformada a partir del MDP (en el caso que corresponda), se realiza la conversión en un grafo implícito, nuevamente recorriendo el MTBDD completamente.

A partir del mismo, por medio del método de la clausura transitiva, el grafo resulta en una expresión regular. Es así como el método de las expresiones regulares finaliza.

Para el método de las expresiones regulares reducidas, sólo debemos aplicarle las técnicas de reducción a la expresión regular anterior (de la totalidad del grafo).

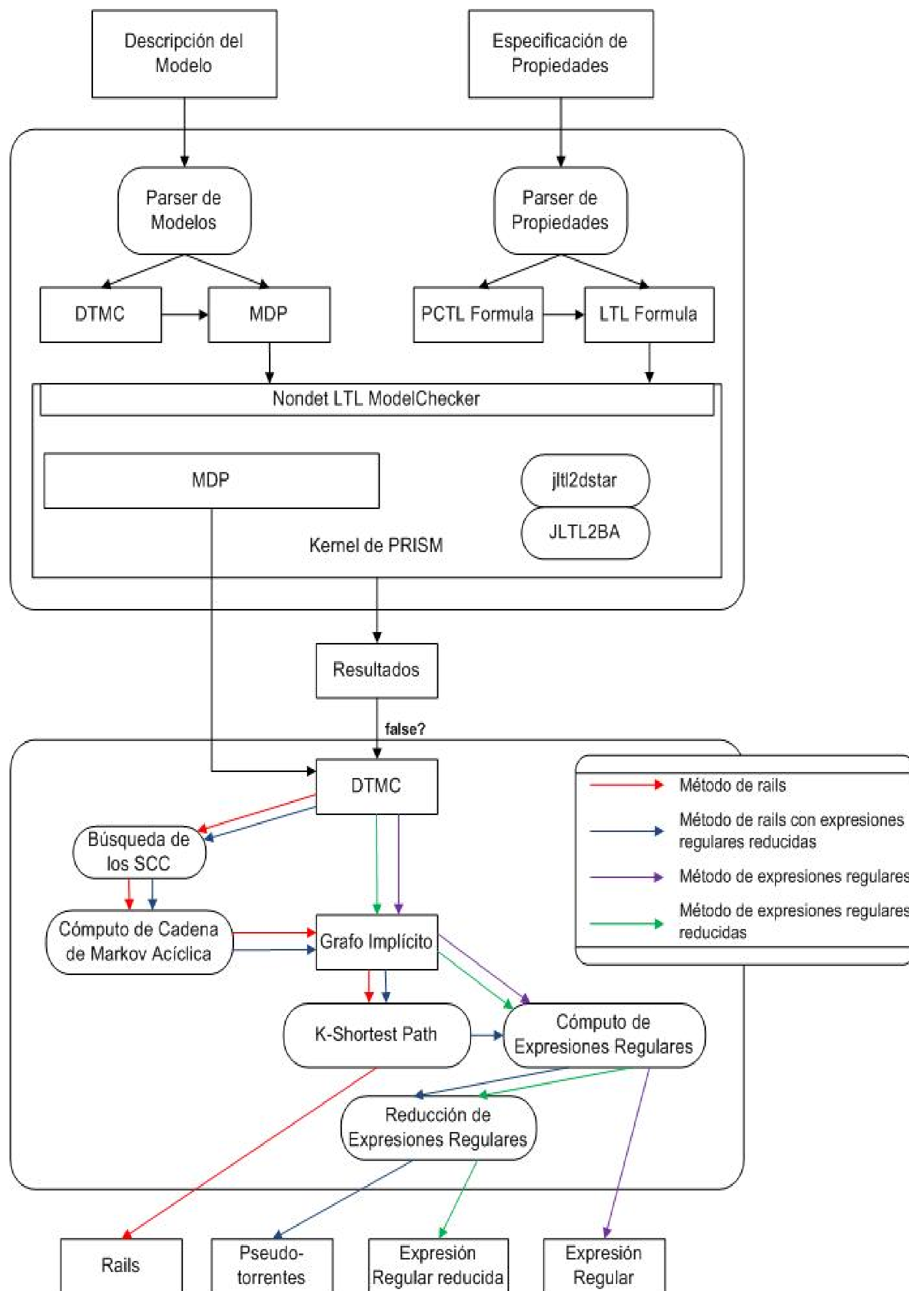

Figura 7.2: Obtención de contraejemplos en PRISM

CAPÍTULO 8

Casos de estudio

Como en [HKD09], inicialmente hemos de usar los protocolos de “leader election” (sincrónico) y “crowds” para evaluar los resultados obtenidos. Esta decisión está basada en que tal publicación es la que más se aproxima a nuestro método de contraejemplos, y nos va a permitir realizar comparaciones. Posteriormente añadiremos otros casos de estudios para sacar nuevas conclusiones.

8.1. Leader election sincrónico

El protocolo de leader election sincrónico (o elección de líder), [IR90], está ideado para resolver la siguiente situación: dado que N procesos están acomodados en un anillo (unidireccional), se debe diseñar un protocolo que permita la elección de un único líder (un único proceso), mediante el envío de mensajes a través del anillo.

Se asume que una constante K es dada, y cada proceso elije aleatoriamente un número en el conjunto $\{1,\ldots,K\}$ (que llamaremos *id*). A partir de ese momento, los procesos pasan el *id* a través del anillo hasta que todos los procesos ven los *ids* de los demás. Si existe algún único *id*, entonces se define como el líder al proceso con máximo *id* único. Caso contrario, empieza una nueva ronda (con una nueva elección de *ids*). Resulta evidente que en algún momento un líder va a ser elegido.

Buscaremos un contraejemplo a la fórmula $\mathcal{P}_{<1.0}(\Diamond leader_elected)$ donde $leader_elected$ se refiere a los estados en los que el líder haya resultado electo.

Para ello, consideremos inicialmente $N = 3$ y $K = 2$, es decir, tenemos 3 procesos que eligen entre los valores 0 y 1 como *id*. Por ende, el líder será elegido siempre que exista un proceso que seleccione el 0 y otro que elija el 1 (en tal caso, si hay dos procesos que eligieron el 0, el máximo *id* único es el 1; caso contrario es el 0).

La implementación del modelo que se usó, es la que presenta PRISM en sus ejemplos. Los contraejemplos dependerán de las variables que tal modelo emplee, pero aquí presentaremos versiones simplificadas. Diremos que I es el estado inicial, R denota la reiniciación de una nueva ronda, F es el estado final, Z es el estado donde todos los *id* son cero y O aquellos donde *id* es uno (notar que si $K = 2$, entonces no se elige

el líder si todos los *ids* son iguales). Las variables $p1$, $p2$ y $p3$ son los *ids* elegidos (aleatoriamente) por cada proceso.

Los siguientes son los resultados obtenidos para cada una de las implementaciones:

- Expresiones regulares: ya para este caso, en el que tenemos pocos estados (26), el resultado es casi imposible de "debuggear". El hecho de tener alternativas ('+') desparramados dentro de los contraejemplos, representa una dificultad para que el usuario analice el progreso. Presentamos el comienzo del contraejemplo (no es su totalidad), para que veamos cuan inmanejable resulta:
  ((((RZ((RZ)+E)*RO)+(RO))((RZ((RZ)+E)*RO)+(RO)+E)*((RZ((RZ)+E)*R(p1=0,p2=1,p3=1))+(R(p1=0,p2=1,p3=1))))+(RZ((RZ)+E)*R(p1=0,p2=1,p3=1))+(R(p1=0,p2=1,p3=1)))F*
  +
  ((((RZ((RZ)+E)*RO)+(RO))((RZ((RZ)+E)*RO)+(RO)+E)*((RZ((RZ)+E)*R(p1=1,p2=0,p3=0))+(R(p1=1,p2=0,p3=0))))+(RZ((RZ)+E)*R(p1=1,p2=0,p3=0))+(R(p1=1,p2=0,p3=0)))F*
  +
  ((((RZ((RZ)+E)*RO)+(RO))((RZ((RZ)+E)*RO)+(RO)+E)*((RZ((RZ)+E)*R(p1=1,p2=1,p3=0))+(R(p1=1,p2=1,p3=0))))+(RZ((RZ)+E)*R(p1=1,p2=1,p3=0))+(R(p1=1,p2=1,p3=0)))F*
  +
  ((RZ((RZ)+E)*R(p1=1,p2=1,p3=0))+(R(p1=1,p2=1,p3=0)))F*
  +
  ((RZ((RZ)+E)*R(p1=1,p2=0,p3=1))+(R(p1=1,p2=0,p3=1)))F*
  +
  ((RZ((RZ)+E)*R(p1=0,p2=1,p3=0))+(R(p1=0,p2=1,p3=0)))F*
  +
  ((RZ((RZ)+E)*R(p1=0,p2=0,p3=1))+(R(p1=0,p2=0,p3=1)))F*

- Expresiones regulares reducidas: consideramos que se ha logrado un progreso respecto al caso anterior; pero todavía tenemos inconvenientes por el número de contraejemplos que se generan:
  ((RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*(RZ(RZ)*R(p1=0,p2=1,p3=1)))F*
  +
  ((RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*(R(p1=0,p2=1,p3=1)))F*
  +
  ((R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*(RZ(RZ)*R(p1=0,p2=1,p3=1)))F*
  +
  ((R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*(R(p1=0,p2=1,p3=1)))F*
  +
  (RZ(RZ)*R(p1=0,p2=1,p3=1))F*
  +
  (R(p1=0,p2=1,p3=1))F*
  +
  ((RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*(RZ(RZ)*R(p1=1,p2=0,p3=0)))F*
  +
  ((RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*(R(p1=1,p2=0,p3=0)))F*
  +
  ((R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*(RZ(RZ)*R(p1=1,p2=0,p3=0)))F*
  +
  ((R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*(R(p1=1,p2=0,p3=0)))F*
  +

(RZ(RZ)*R(p1=1,p2=0,p3=0))F*
+
(R(p1=1,p2=0,p3=0))F*
+
(RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*((RZ(RZ)*R(p1=1,p2=1,p3=0))F*)
+
(RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*((R(p1=1,p2=1,p3=0))F*)
+
(RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*((RZ(RZ)*R(p1=1,p2=0,p3=1))F*)
+
(RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*((R(p1=1,p2=0,p3=1))F*)
+
(RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*((RZ(RZ)*R(p1=0,p2=1,p3=0))F*)
+
(RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*((R(p1=0,p2=1,p3=0))F*)
+
(RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*((RZ(RZ)*RO)F*)
+
(RZ(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*((RO)F*)
+
(R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*((RZ(RZ)*R(p1=1,p2=1,p3=0))F*)
+
(R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*((R(p1=1,p2=1,p3=0))F*)
+
(R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*((RZ(RZ)*R(p1=1,p2=0,p3=1))F*)
+
(R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*((R(p1=1,p2=0,p3=1))F*)
+
(R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*((RZ(RZ)*R(p1=0,p2=1,p3=0))F*)
+
(R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*((R(p1=0,p2=1,p3=0))F*)
+
(R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*((RZ(RZ)*RO)F*)
+
(R(p1=1,p2=1,p3=1))((RZ(RZ)*RO)*(RO)*)*((RO)F*)
+
(RZ(RZ)*R(p1=1,p2=1,p3=0))F*
+
(R(p1=1,p2=1,p3=0))F*
+
(RZ(RZ)*R(p1=1,p2=0,p3=1))F*
+
(R(p1=1,p2=0,p3=1))F*
+
(RZ(RZ)*R(p1=0,p2=1,p3=0))F*
+
(R(p1=0,p2=1,p3=0))F*
+
(RZ(RZ)*RO)F*
+

(RO)F*

- [1] Rails: I -> SCC1 -> R(p1=1,p2=1,p3=0) -> SCC2
- [1] Rails con expresiones regulares reducidas: I -> SCC1 -> R(p1=1,p2=1,p3=0) -> SCC2
  donde:
  - SCC1:
    (Z(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*(RZ(RZ)*RZ)
    +
    (Z(RZ)*RO)((RZ(RZ)*RO)*(RO)*)*RZ
    +
    O((RZ(RZ)*RO)*(RO)*)*(RZ(RZ)*RZ)
    +
    O((RZ(RZ)*RO)*(RO)*)*RZ
    +
    Z(RZ)*RZ
    +
    E
  - SCC2:
    F*

Concluimos entonces que el primer método (empleando sólo expresiones regulares) resulta prácticamente inútil. El beneficio que presenta el segundo método (reduciendo las expresiones regulares previas) es que cada uno de los contraejemplos está generado por transiciones de estados (y loops entre éstos, dados por el '+'). Para éste caso, también podemos ver que el número de contraejemplos es muy grande en relación a los métodos siguientes.

Respecto a los "rails", en este modelo vemos que el contraejemplo es el más significativo. Partiendo del estado inicial (I), se alcanza una componente fuertemente conexa (SCC1) a partir de la cual se inicia una nueva ronda (R) y se alcanza un estado que permite elegir el líder ($p1 = 1, p2 = 1, p3 = 0$); esto nos hace suponer que SCC1 va a ser una serie de rondas donde sucesivamente obtengamos resultados no satisfactorios (ya sea lo que llamamos Z u O), es decir que SCC1 debería tener la forma de $(RZ+RO)^*$. Por su parte, SCC2 tiene que estar dado por el estado final (F) dado que se ha llegado a un estado de aceptación. Finalmente, el método de expansión de las SCCs no tendría sentido hacerlo por las deducciones anteriores. De todas maneras vemos que SCC1 se expande a expresiones como la previa, pero el automatismo lo reduce a múltiples expresiones haciéndolo más complicado (y, básicamente, generando más contraejemplos que no añaden información importante).

8.2. Crowds

Centremos ahora nuestra atención en el protocolo de "crowds". El mismo, descripto en [RR98], presenta una manera de hacer navegación web de manera anónima, es

[1] En éste caso, sólo se presenta un contraejemplo cuando se alcanza el estado (p1=1,p2=1,p3=0). Por la simetría del problema, tendremos otros 5 rails con la misma probabilidad con los estados: (p1=1,p2=0,p3=1), (p1=0,p2=1,p3=1), (p1=0,p2=0,p3=1), (p1=0,p2=1,p3=0) y (p1=1,p2=0,p3=0)

decir, resguardando la identidad de los usuarios. El protocolo funciona de la siguiente manera: un usuario se une a un grupo (o multitud o crowd) notificando su entrada a los demás miembros de tal grupo. Todos los usuarios pueden enviar y recibir mensajes de un servidor ajeno al grupo. Se asume que dentro del grupo hay usuarios corruptos que pretenden establecer la identidad de quién manda mensajes. El envío de datos se realiza de la siguiente manera: un usuario del grupo selecciona a otro usuario (pudiendo ser él mismo) y envía el mensaje (encriptado). Quien lo recibe, o bien distribuye el mensaje al destinatario con probabilidad p_f, o bien lo envía (reencriptándolo) a otro miembro del grupo iniciando otra ronda.

En el caso de que llegue un mensaje a un nodo corrupto, éste usará los datos del remitente para determinar su identidad y lo envía directamente al server. En el caso en que quien envíe el mensaje sea observado al menos 2 veces por un nodo corrupto, diremos que ha sido *positivamente identificado* (o *pos_id*).

El trayecto de un mensaje desde un usuario en particular hacía y desde el servidor será siempre el mismo a partir del momento en que se determina su ruta por primera vez.

La propiedad que verificaremos y para la que buscaremos contraejemplos es precisamente $\mathcal{P}_{<p}(\Diamond pos_id)$. Tomaremos $N = 4, M = 2$ y $R = 2$ donde N es el número de procesos en el grupo, M es el número de nodos corruptos y R la cantidad de rondas (o sesiones) de envío de mensajes. Para lograr una mayor legibilidad de los contraejemplos, daremos nombres a los estados: I será el inicial, N indica que se inicia una nueva instancia del protocolo y S que se inicia una ronda. D ocurre cuando el mensaje ha sido entregado al destinatario y $u = n$ indica que n fue el ultimo usuario por el que pasó el mensaje. Si se señala un estado como *bad*, tal es un nodo corrupto; caso contrario será *good*. Recordemos que siempre que el mensaje llegue a un estado *bad*, éste será entregado a destinatario en el estado inmediatamente posterior. En este modelo, el nodo cero será quien emita el mensaje inicial.

Para las distintas implementaciones se obtienen los resultados:

- Expresiones regulares: no incluiremos los resultados por ser muy pesados, puesto que contiene 209 estados y 39 uniones ('+') y 21 estrellas de Kleene ('*') intercaladas que hacen que sea imposible usar el contraejemplo para debugging.

- Expresiones regulares reducidas: obtenemos 47 uniones de expresiones reducidas, exponemos 4 de ellas para que veamos que las mismas pueden ser útiles a la hora de procesarlas:
  (I((NS(((u=0,good)((u=0,good))*(u=0,good)(u=1,good))((((u=1,good)(u=0,good)
  ((u=0,good))*(u=0,good)(u=1,good)))*(((u=1,good)(u=1,good)))*)*(u=1,good)
  (u=0,good)((u=0,good))*(u=0,bad)DN)S(((u=0,good)((u=0,good))*(u=0,good)
  (u=1,good))((((u=1,good)(u=0,good)((u=0,good))*(u=0,good)(u=1,good)))*
  (((u=1,good)(u=1,good)))*)*(u=1,good)(u=0,good)((u=0,good))*(u=0,bad)DN)N*N)))
  +
  (I(NS(((u=0,good)((u=0,good))*(u=0,good)(u=1,good))((((u=1,good)(u=0,good)
  ((u=0,good))*(u=0,good)(u=1,good)))*(((u=1,good)(u=1,good)))*)*(u=1,good)
  (u=0,good)((u=0,good))*(u=0,bad)DN)S(((u=0,good)((u=0,good))*(u=0,good)
  (u=1,good))((((u=1,good)(u=0,good)((u=0,good))*(u=0,good)(u=1,good)))*
  (((u=1,good)(u=1,good)))*)*(u=1,good)(u=0,good)((u=0,good))*(u=0,bad)DN)N*E))
  +
  (I(NS(((u=0,good)((u=0,good))*(u=0,good)(u=1,good))((((u=1,good)(u=0,good)
  ((u=0,good))*(u=0,good)(u=1,good)))*(((u=1,good)(u=1,good)))*)*(u=1,good)
  (u=0,good)((u=0,good))*(u=0,bad)DN)S(((u=0,good)(u=1,good))((((u=1,good)

(u=0,good)((u=0,good))*(u=0,good)(u=1,good)))*(((u=1,good)(u=1,good)))*)*
(u=1,good)(u=0,good)((u=0,good))*(u=0,bad)DN)N*N))
+
(I(NS(((u=0,good)((u=0,good))*(u=0,good)(u=1,good))((((u=1,good)(u=0,good)
((u=0,good))*(u=0,good)(u=1,good)))*(((u=1,good)(u=1,good)))*)*(u=1,good)
(u=0,good)((u=0,good))*(u=0,bad)DN)S(((u=0,good)(u=1,good))((((u=1,good)
(u=0,good)((u=0,good))*(u=0,good)(u=1,good)))*(((u=1,good)(u=1,good)))*)*
(u=1,good)(u=0,good)((u=0,good))*(u=0,bad)DN)N*E))

- Rails: INS -> SCC1 (u=0,bad)DNS -> SCC2 -> (u=0,bad)DN -> SCC3

- Rails con expresiones regulares reducidas: INS -> SCC1 (u=0,bad)DNS -> SCC2 -> (u=0,bad)DN -> SCC3
  donde:

  - SCC1:
    ((u=0,good)(u=0,good)*(u=0,good)(u=1,good))(((u=1,good)(u=0,good)
    (u=0,good)*(u=0,good)(u=1,good))*(((u=1,good)(u=1,good)))*)*(u=1,good)
    (u=0,good)(u=0,good)*
    +
    ((u=0,good)(u=1,good))(((u=1,good)(u=0,good)(u=0,good)*(u=0,good)
    (u=1,good))*((u=1,good)(u=1,good))*)*(u=1,good)(u=0,good)(u=0,good)*
    +
    (u=0,good)(u=0,good)*
    +
    E
  - SCC2:
    ((u=0,good)((u=0,good))*(u=0,good)(u=1,good))(((u=1,good)(u=0,good)
    (u=0,good)*(u=0,good)(u=1,good))*(((u=1,good)(u=1,good)))*)*(u=1,good)
    (u=0,good)(u=0,good)*
    +
    ((u=0,good)(u=1,good))(((u=1,good)(u=0,good)(u=0,good)*(u=0,good)
    (u=1,good))*((u=1,good)(u=1,good))*)*(u=1,good)(u=0,good)(u=0,good)*
    +
    (u=0,good)(u=0,good)*
    +
    E
  - SCC3:
    N*

Como pasara con el protocolo de leader election, para las expresiones regulares reducidas, el número de contraejemplos es exageradamente grande; y si los comparamos, muchos de ellos son muy parecidos (e incluso algunos sólo difieren en un estado).

Por su parte, el rail nos da una buena noción de cómo serán nuestros contraejemplos: lo que tenemos es la inicialización del protocolo, una componente fuertemente conexa (SCC1) que finaliza cuando el mensaje llega a un nodo corrupto habiendo sido enviado del nodo inicial (u=0,bad); y lo mismo ocurre nuevamente con SCC2. Considerando la definición del algoritmo, podríamos deducir que ambas componentes conexas serán similares en el siguiente aspecto: si el estado posterior a las SCCs es (u=0, bad) entonces el mensaje tiene que haber arrancado del nodo 0, y visitará el nodo 1 yendo y viniendo las veces que se pueda. Por ende, las SCCs deberían verse como

$((u = 0, good)^*(u = 1, good)^*)^*$. En principio, no sería necesario para este protocolo expandir tales SCCs.

De todas maneras, si analizamos la expansión de las mismas, vemos que realmente ocurre lo predicho; aunque las expresiones no se ven tan sencillas. Esto se debe a que cuando se crea el autómata de Rabin para hacer el model checking de la fórmula LTL se introducen estados que reiteran los valores de las variables. Además, no todas las transiciones de estados indican envíos de mensajes; es posible que, debido a la manera en que se ha modelado el protocolo, lo que cambien sean variables locales.

8.3. Comparación

A continuación, plasmamos en tablas los resultados de haber aplicado los algoritmos para modelos ya implementados en PRISM. Todos los experimentos se llevaron a cabo en un Intel Core 2 Duo T5200 @ 1.60GHz con 2 GB de RAM corriendo Linux.

Los dos primeros protocolos son los ya mencionados "Leader election sincrónico" y "Crowds". A ellos, le agregaremos tres protocolos, cuyos modelos y propiedades son parte de los casos de estudio de PRISM. Los mismos son los siguientes (usamos sus nombres en inglés porque así son más conocidos):

- Bounded Retransmission Protocol (BRP) [HSV94], [DJJL01]: Se modela el envío de un archivo dividido en N fragmentos, cada uno de los cuales se pueden enviar hasta MAX veces. Además de los fragmentos, el cliente y el servidor se envían mensajes de estado y error.

- Self-Stabilising [IJ90]: Representa N procesos en un anillo con comunicación bidireccional y N fichas (tokens) que poseen los procesos activos. Cada proceso activo, decide aleatoriamente si entregarle el token al proceso de la derecha o al de la izquierda (y en caso de que tenga más de un token, sólo pasa uno y a los otros los deshace). Así, en un número finito de pasos habrá sólo un proceso activo.

- Dining cryptographers [Cha88]: Modela N (N impar) criptógrafos comiendo en ronda. El mesero les informa que la cuenta ha sido saldada, de manera anónima. O bien uno de ellos la ha pagado, o ha sido su superior. Cada uno de los comensales arroja una moneda e informa sobre el resultado al criptógrafo de su derecha. Ahora cada criptógrafo informa si ambas monedas que vio son iguales o no, salvo si alguno pagó la cuenta, quien dice lo contrario. Un número par de coincidencias indica que el superior pagó, mientras que si fuera impar, entonces un criptógrafo fue quien abonó.

Las tablas se adjuntan a continuación. Las celdas pintadas con rojo indican que tal método no finaliza luego de ser corrido durante 30 minutos. Mientras que el amarillo indica que no se obtuvieron resultados porque se quedó sin memoria.

Para simplificar las tablas, llamamos "Rails" al método de los Rails, "Rails + ERR" al método de los Rails con expresiones regulares reducidas, "ER" al método de las expresiones regulares puras y "ERR" el de las expresiones regulares reducidas.

Cuando enumeramos los representantes de un contraejemplo del método "Rails + ERR", para cada rail, obtendremos n cadenas en cada SCC; por ello los describiremos con signos "+". Por ejemplo, si un contraejemplo tiene 2 rails y el primero tiene 3 expresiones para cada SCC y el segundo 4 y 2 expresiones en cada SCC, diremos que tiene 3+3 | 4+2 representantes (el signo "|" distingue los rails).

En las columnas de "Componentes", "Inputs" y "Outputs", se indica el promedio de todas las SCCs.

Para el protocolo de leader election, la tabla 8.1 nos muestra la mala performance de los algoritmos de reducción de expresiones regulares, y dado que el tamaño de las expresiones generadas por ER es muy grande, sostenemos que el de los Rails, es el método que mejor se adapta.

La siguiente tabla, 8.2, describe el protocolo de crowds. Si bien sólo obtenemos resultados en el primer caso de ERR, vemos la gran diferencia en los representantes contra ambos métodos de rails. Esto se debe a que, cuando se distribuyen las expresiones regulares para eliminar las uniones, se generan muchas expresiones similares.

El protocolo de BRP se muestra en tabla 8.3. Nuevamente, el alto número de representantes de ERR contra los rails, nos permite asegurar que el último método es el más adecuado. La situación se repite en la tabla 8.4 que describe "Self-stabilising".

Por último, la tabla 8.5 relacionada con "Dining Cryptographers", nos muestra que los modelos tienen muchos estados, y $N = 3$ es un caso en el que se repite la situación. Los métodos de rails tienen menos representantes.

Leader election										
Método	N	K	Estados	SCCs	SCCs no triviales	Componentes	Inputs	Outputs	tiempo(seg)	Representantes
Rails	3	2	26	2	1	7	1	6	0.07	1
Rails + ERR									0.11	5+1
ER									0.13	
ERR									0.87	35
Rails	3	4	147	2	1	16	1	120	0.90	1
Rails + ERR										
ER									64.13	
ERR										
Rails	4	2	61	2	1	28	1	80	0.36	1
Rails + ERR										
ER									23.03	
ERR										

Tabla 8.1: Resultados de Leader election.

Crowds											
Método	N	M	R	Estados	SCCs	SCCs no triviales	Componentes	Inputs	Outputs	tiempo(sec)	Representantes
Rails	4	2	2	77	56	2	6	1	4	0.19	1
Rails + ERR										0.29	4+4+1
ER										0.16	
ERR										4.80	48
Rails	4	2	3	183	92	7	6	1	4	0.53	1
Rails + ERR										0.60	4+4+4+1
ER										4.63	
ERR											
Rails	5	3	2	138	111	2	9	1	6	0.36	1
Rails + ERR										18.60	22+22+1
ER										1.20	
ERR											
Rails	5	3	3	396	244	9	9	1	6	1.37	1
Rails + ERR										73.53	22+22+22+1
ER										197.50	
ERR											

Tabla 8.2: Resultados de Crowds.

Bounded Retransmission Protocol							
Método	N	MAX	Estados	SCCs	SCCs no triviales	tiempo(sec)	Representantes
Rails	2	2	89	13	0	0.23	2
Rails + ERR						0.23	2
ER						0.31	
ERR						33.40	192
Rails	5	3	281	22	0	0.85	2
Rails + ERR						0.85	2
ER						1465.91	
ERR							
Rails	12	5	1028	42	0	1.63	2
Rails + ERR						1.63	2
ER							
ERR							

Tabla 8.3: Resultados de BRP.

Self-Stabilising										
Método	N	K	Estados	SCCs	SCCs no triviales	Componentes	Inputs	Outputs	tiempo(sec)	Representantes
Rails	3	3	7	2	1	2	1	1	0.01	1
Rails + ERR									0.05	5+1
ER									0.01	
ERR									0.01	3
Rails	5	3	31	6	3	3	2	1.67	0.03	1
Rails + ERR									0.11	18+1
ER									0.29	
ERR									2.39	75
Rails	8	3	255	19	13	3.85	3.31	1.62	0.31	1
Rails + ERR									0.52	9+10+1
ERR										
ER										
Rails	12	3	4095	57	47	4.34	3.96	1.83	2.67	1
Rails + ERR									4.63	4+4+1
ER										
ERR										

Tabla 8.4: Resultados de Self-Stabilising.

Dining Cryptographers						
Método	N	Estados	SCCs	SCCs no triviales	tiempo(sec)	Representantes
Rails	3	380	119	0	0.36	9
Rails + ERR					0.38	9
ER					4.26	
ERR					5.13	25
Rails	4	2165	496	0	2.33	17
Rails + ERR					2.40	17
ER						
ERR						
Rails	5	11850	2135	0	14.76	33
Rails + ERR					14.77	33
ER						
ERR						

Tabla 8.5: Resultados de Dining cryptographers.

CAPÍTULO 9

Conclusiones

Durante el desarrollo de este proyecto, se implementaron cuatro métodos para obtener contraejemplos en el model checker PRISM y se analizaron los resultados obtenidos. Por ser estos métodos de orden exponencial, y por las dificultades que se presentan al analizar contraejemplos con excesivos estados, es conveniente buscar contraejemplos sobre modelos con el menor número de estados posibles. Es decir, si tenemos por ejemplo un modelo cuyo número de estados, iteraciones, reenvío de mensajes o cualquier parámetro sea configurable, para encontrar los contraejemplos siempre es recomendable hacerlo con esos números alcanzando el menor valor posible. En general, para modelos con más estados, los contraejemplos serán adaptados de tales contraejemplos.

Con respecto a los métodos, consideramos que el de las expresiones regulares puras presenta dificultades para ser visualizado, puesto que analizar las transiciones de estados a partir de las expresiones es sumamente complicado y poco natural.

Por su parte, los tres métodos restantes son de utilidad teniendo en cuenta, como se comentara en capítulos anteriores, que cada uno de ellos posee bondades y debilidades dependiendo del modelo a analizar.

En el presente trabajo, las implementaciones se realizaron a partir del modelo simbólico de DTMC. Por ese motivo se eligió emplear el método de Clausura Transitiva para generar las expresiones regulares. Éstas resultaron extremadamente grandes e inmanejables. Hemos visto que el método de Eliminación de Estados genera expresiones regulares más pequeñas; por eso suponemos que usarlo con el modelo explícito de DTMC sería una buena opción a futuro.

A partir de lo observado en los resultados del presente trabajo, sostenemos que es de gran importancia la generación de una herramienta para la visualización de contraejemplos. A continuación, proponemos los lineamientos para el desarrollo de dicha herramienta: una expresión como la del caso de estudio de Leader election del capítulo 8, $(RZ(RZ)^*RO)((RZ(RZ)^*RO)^*(RO)^*)^*((R(p1=0,p2=1,p3=0))F^*)$, podría representarse como se ve en la figura 9.

Recordemos que tenemos 3 procesos que se envían mensajes para determinar un líder. El paso de los mensajes entre los procesos no sigue un orden en particular; pero para fijar ideas, suponemos que primero se envían de $Proc_0$ a $Proc_1$, luego de $Proc_1$ a

Proc$_2$ y finalmente de *Proc*$_2$ a *Proc*$_0$. Las flechas horizontales indican el envío de esos mensajes. Las clausuras de Kleene presentes en la fórmula pueden ser visualizadas como un bucle o loop.

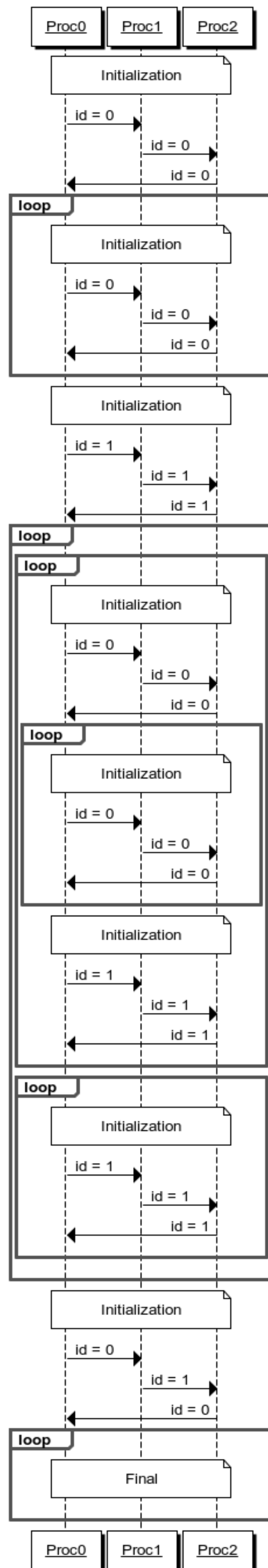

Figura 9.1: Posible visualización de una expresión regular

APÉNDICE A

Apéndice A: transformación de un MDP en una DTMC

Los métodos presentados para la generación de contraejemplos, permite obtenerlos a partir de las cadenas de Markov.

Para el caso en que se modele un sistema por medio de un proceso de decisión de Markov, tendremos que convertirlo al mismo en una DTMC. Tal tarea, es posible por la existencia de un scheduler η que satisface $\mathcal{D} \vDash_{\leq p} \Diamond\vartheta \Leftrightarrow \mathcal{D}_\eta \vDash_{\leq p} \Diamond\vartheta$.

En este apéndice, describiremos la manera en que la conversión se desarrolla. Primero presentaremos un método obtenido de [Bd95], el cual no intenta resolver precisamente este problema, pero será de vital ayuda.

Teorema A.1. *Para un MDP $\mathcal{D} = (S, s_{init}, \tau, L)$, sea $P_{s_i}(\Diamond\varphi)$ la probabilidad maximal de que un estado $s_i \in S$ alcance un estado que satisface $\Diamond\varphi$. Si tomamos $x_i = P_{s_i}(\Diamond\varphi)$, entonces podemos encontrar las probabilidades maximales para cada estado, resolviendo el problema de minimización lineal: minimizar $\sum_{s\in S} x_s$ sujeto a $\sum_{s'\in S} \mu_s(s')x_{s'} \leq x_s$ para cada $s \in S$ y $\mu_s \in \tau(s)$.*

A partir de teorema anterior, estamos tentados en suponer que si $\{p_i\}$ fuera una solución del problema de minimización lineal, entonces tomando η tal que $\eta(s_i) = \mu$, con $\sum_j \mu(s_j)p_j = p_i$, obtenemos un scheduler que por el cual se podría reducir el MDP en una DTMC. Como vemos en el siguiente ejemplo, lo que recién enunciamos no se cumple:

Ejemplo A.1. Sea el $\mathcal{D}$ el MDP de la figura A.1.

Para el mismo, el sistema de inecuaciones será:

$$0.3x_1 + 0.7x_2 \leq x_0 \qquad x_3 \leq x_0$$
$$x_0 \leq x_1 \qquad x_0 \leq x_2$$
$$x_3 \leq x_3$$

Para tal sistema, la solución es $(1.0, 1.0, 1.0, 1.0)$.

Ahora bien, debemos determinar cuál será el no-determinismo a usar. El problema que se nos presenta es que las inecuaciones de ambos no-determinismos son satisfechas por la solución encontrada ($0.3x_1 + 0.7x_2 \leq x_0$ y $x_3 \leq x_0$).

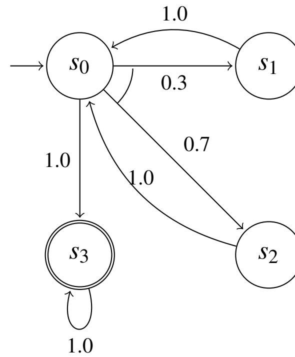

Figura A.1: Ejemplo de MDP a convertir

Es así como necesitamos refinar los resultados obtenidos. Para lograrlo, se propone el siguiente algoritmo.

Básicamente, una vez que se resuelve el problema de minimización lineal, se requiere eliminar los estados no-deterministas que persistan como el que se aprecia en el ejemplo anterior. Para hacerlo, consideramos el conjunto de estados que satisfacen la proposición φ. A continuación, se procede a ir eliminando los no-determinismos "hacia atrás".

Para cada uno de los "estados anteriores" que sean no-deterministas, elegimos un no-determinismo y restringimos las transiciones a esos estados. Calculamos los estados que se alcanzan (bajo esa restricción), y pretendemos llegar a los estados iniciales. Si se logra, entonces hemos encontrado el no-determinismo que buscábamos; caso contrario debemos seguir iterando.

El algoritmo, en pseudo-código, se presenta debajo:

Algorithm 1 get_DTMC

```
states ← "accepting states"
prev_states ← previous_states(states)
while prev_states ≠ states do
    for all nondeterministic_state ∈ prev_states do
        remove_nondeterminism()
    end for
    states ← prev_states
    prev_states ← previous_states(states)
end while
```

Donde la función *remove_nondeterminism*() se podría describir como:

Algorithm 2 *remove_nondeterminism*

for all $nondeterminism \in nondeterministic_state$ **do**
  $done = false$
  while $\neg done$ **do**
    $nondet_trans = trans|_{nondeterminism}$
    $post_states \leftarrow posterior_states(prev_states)$
    if $post_states = prev_states$ **then**
      $trans = nondet_trans$
      $break$
    end if
  end while
end for

Referencias

[ADR09] Miguel E. Andrés, Pedro D'Argenio, and Peter Rossum. Significant diagnostic counterexamples in probabilistic model checking. In *HVC '08: Proceedings of the 4th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, pages 129–148, Berlin, Heidelberg, 2009. Springer-Verlag.

[AG86] Malcolm Adams and Victor Guillemin. *Measure Theory and Probability*. Wadsworth, Monterey, CA, 1986.

[AHL05] Husain Aljazzar, Holger Hermanns, and Stefan Leue. Counterexamples for timed probabilistic reachability. In *FORMATS*, pages 177–195, 2005.

[And06] Miguel E. Andrés. Derivación de contraejemplos para model checking cuantitativo. Master's thesis, FaMAF, 2006.

[Bd95] Andrea Bianco and Luca de Alfaro. Model checking of probabalistic and nondeterministic systems. In *FSTTCS*, pages 499–513, 1995.

[Bed07] Carlos S. Bederián. Model checking cuantitativo de propiedades ltl en prism. Master's thesis, FaMAF, 2007.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[Bre92] Leo Breiman. *Probability*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[Cha88] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[De 98] Luca De Alfaro. *Formal verification of probabilistic systems*. PhD thesis, Stanford, CA, USA, 1998. Adviser-Manna, Zohar.

[DJJL01] Pedro R. D'Argenio, Bertrand Jeannet, Henrik Ejersbo Jensen, and Kim Guldstrand Larsen. Reachability analysis of probabilistic systems by successive refinements. In *PAPM-PROBMIV*, pages 39–56, 2001.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *ICALP*, pages 169–181, 1980.

[Epp98] David Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.

[Geb08] Fayez Gebali. *Analysis of Computer and Communication Networks*. Springer Publishing Company, Incorporated, 2008.

[Haw79] Thomas Hawkins. *Lebesgue's Theory of Integration: Its Origins and Development*. Chelsea, New York, second edition, 1979.

[HKD09] Tingting Han, Joost-Pieter Katoen, and Berteun Damman. Counterexample generation in probabilistic model checking. *IEEE Trans. Software Eng.*, 35(2):241–257, 2009.

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

[HSV94] L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proc. International Workshop on Types for Proofs and Programs (TYPES'93)*, volume 806 of *LNCS*, pages 127–165. Springer, 1994.

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

[IR90] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1), 1990.

[Jal91] Pankaj Jalote. *An integrated approach to software engineering*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[JM99] Víctor M. Jiménez and Andrés Marzal. Computing the k shortest paths: A new algorithm and an experimental comparison. In *WAE '99: Proceedings of the 3rd International Workshop on Algorithm Engineering*, pages 15–29, London, UK, 1999. Springer-Verlag.

[Neu05] Christoph Neumann. Converting deterministic finite automata to regular expressions, 2005.

[Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, pages 337–351, 1982.

[RR98] M. Reiter and A. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC*, 1(1):66–92, 1998.

[Som01] Fabio Somenzi. Efficient manipulation of decision diagrams. *STTT*, 3(2):171–181, 2001.

[Var85] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 327–338, Washington, DC, USA, 1985. IEEE Computer Society.

[WZ77] Richard L. Wheeden and Antoni Zygmund. *Measure and integral : an introduction to real analysis / Richard L. Wheeden and Antoni Zygmund*. M. Dekker, New York :, 1977.