Trabajo Especial

Reducción de orden parcial en model checking probabilista simbólico

Luis María Ferrer Fioriti

Director:

Dr. Pedro R. D'Argenio

Co-Director:

Dr. Sergio Giro

Universidad Nacional de Córdoba

Facultad de Matemática Astronomia y Física

Todo dura siempre un poco más
de lo que debería.

Rayuela
Julio Cortazar

Resumen

El problema fundamental de los model checkers es la explosión exponencial del espacio de estados que se produce al agregar nuevas componentes o variables. El problema se exacerba en los model checkers probabilistas dado que no sólo requiere una búsqueda exhaustiva del espacio de estado, sino cálculos numéricos cuya cantidad de variables y (des)igualdades depende directamente de la cantidad de estados y transiciones.

Existen varias técnicas que intentan mitigar el crecimiento del espacio de estados. El model checking simbólico es una de las más exitosas. Esta técnica se basa en el uso de BDD para representar los estados y transiciones. Otra técnica exitosa es la reducción de orden parcial, la cual considera solo algunas ejecuciones representativas a la hora de verificar una propiedad, eliminando estados "redundantes".

En este trabajo presentamos la implementación de la tecnica de orden parcial en un model checker probabilista simbólico. La noción de orden parcial elegida para implementar en este trabajo es la más moderna. Ésta permite una mayor reducción ya que no tiene en cuenta ejecuciones probabilistas irreales consideradas en técnicas anteriores. La implementación se realizó sobre PRISM, que es un model checker probabilista moderno y potente, y cuya distribución es de carácter libre.

Palabras clave: model cheking simbólico,sistemas distribuidos, sistemas probabilistas, reducción de orden parcial

Clasificación:
D. Software
D.2 SOFTWARE ENGINEERING
D.2.4 Software/Program Verification
Tema:
Model checking

Índice general

Capítulo 1

Introducción

Con la introducción de los sistemas computacionales hace más de cincuenta años, cada vez son más las actividades que utilizan en mayor o menor medida sistemas digitales. En un principio las computadoras fueron utilizadas a nivel militar para calcular trayectorias y a nivel científico para realizar cálculos numéricos complejos. En la actualidad podemos encontrar microcontroladores en artefactos como heladeras, hornos microondas, automóviles, teléfonos, etc. Los sistemas informáticos también permitieron el desarrollo de actividades que requieren de gran precisión como ser la medicina nuclear, sistemas de transporte de gran velocidad, telecomunicaciones, etc.

Dentro de los sistemas existentes los sistemas distribuidos, son aquellos que poseen más de una componente que se ejecutan en paralelo, Este tipo de sistemas introducen un comportamiento no determinista. El resultado del sistema global puede depender del orden en que las componentes fueron ejecutadas. Un ejemplo de sistemas distribuidos son las redes ATM utilizada en los bancos. Otro tipo de sistemas que son interesantes de analizar son aquellos que poseen comportamientos aleatorios. Estos comportamientos pueden ser introducido por el ambiente, por ejemplo en un canal compartido se puede perder un paquete con una probabilidad dada. El comportamiento aleatorio también puede ser originado por parte de alguna componente del sistema mediante una elección probabilista. Muchas veces los algoritmos aleatorios pueden ser mucho más eficientes que los algoritmos tradicionales, como ser el caso del test de primalidad de Miller-Rabin utilizado en RSA para comprobar la primalidad de un número. Este algoritmo es mucho más eficiente que los test convencionales. Incluso hay ciertos problemas que no pueden ser solucionados si no es mediante el uso de aleatoriedad, como ser el problema del Leader Election Protocol.

Debido a la complejidad que poseen los sistemas computacionales, es posible que tengan fallas. Los defectos pueden ser introducidos tanto en la fase de diseño como en la fase de desarrollo. Estos errores pueden ser molestos como ser que

un teléfono celular se apague luego de marcar un número inexistente o que el calendario de un reloj considere al año 2100 como bisiesto, etc. Sin embargo en algunos sistemas ciertos errores no deben ocurrir ya que pueden ocasionar grandes pérdidas económicas, incluso pérdidas de vidas humanas. Un error de cálculo en el sistema bancario puede ocasionar pérdidas multimillonarias, un error en la implementación de un algoritmo de cifrado puede revelar secretos de estado, la aplicación de una mayor dosis de radiación en un tratamiento oncológico puede ocasionar la muerte de pacientes, una falla en el sistema de semáforos de trenes de alta velocidad puede causar la muerte de cientos de pasajeros, etc.

A la hora de detectar fallas en los sistemas existen una gran cantidad de técnicas. Estas pueden ser clasificadas en dos categorías. Por un lado esta el *testing*, estas técnicas consisten en ejecutar el sistema que se desea verificar o parte de este para encontrar comportamientos que no son deseados. El problema que tiene el testing es que no explora todas las posibles ejecuciones del sistema, por lo que solo es posible encontrar errores no la ausencia de estos. Por otro lado se encuentran las técnicas de *verificación formal*, que a diferencia del testing, comprueban de manera exhaustiva todas los posibles comportamientos del sistema. La ventaja que tienen estas técnicas es que a diferencia del testing permiten realizar comprobaciones durante la fase de diseño. Generalmente los errores introducidos en esta fase son difíciles de detectar y muy costosos de eliminar una vez que se esta en la fase de desarrollo ya que es necesario rediseñar y reimplementar el sistema ya sea parcialmente o totalmente.

Dentro de las técnicas de verificación formal se encuentra el *model checking*. Esta técnica consiste en verificar de manera automática mediante el empleo de un algoritmo si cierto modelo de un sistema satisface o no una propiedad dada. El modelo suele ser una descripción abstracta del sistema real, que es generada durante la fase de diseño. También es posible trabajar con descripciones obtenidas a partir del código fuente del sistema.

Uno de los principales problemas del model checking es la gran cantidad de estados que se deben representar debido a que es necesario considerar todos los posibles estados que se pueden alcanzar en una ejecución. Además la complejidad de la comprobación también esta ligada a la cantidad de estados alcanzados. Se han desarrollado varias técnicas para enfrentar el problema de la explosión de estados, como ser la utilización de estructuras de datos que permiten eliminar redundancias en la representación en memoria de los modelos, obteniendo una gran reducción en el uso de memoria. Este tipo de técnicas se la denomina *model checking simbólico* [10, 24, 28, 25]. Otra de las técnicas utilizadas es la denominada *reducción de orden parcial* [26, 14, 6, 4], que consiste en generar un nuevo modelo a partir del original, pero que posea una menor cantidad de estados. Esto es logrado mediante la elección de algunas

ejecuciones que son representativas del resto eliminando estados y transiciones que son redundantes para determinar la validez de la propiedad dada. Observar que el model checking simbólico elimina redundancia en la representación pero mantiene todos los estados y transiciones del modelo dado mientras que la reducción de orden parcial elimina estados y transiciones redundantes.

Cuando se realiza model checking sobre sistemas probabilistas se suelen obtener cotas seguras a las propiedades que se quieren analizar mediante el análisis de los peores escenarios de ejecución, esto suele ocasionar que se consideren ciertas ejecuciones que pueden ser imposibles en el sistema que se quiere analizar, ya que involucra que ciertas componentes tomen decisiones en base al estado interno de otras componentes que aparentemente no están relacionadas. En el marco del trabajo de doctorado de Sergio Giro[19], se plantearon nuevas condiciones que permiten realizar reducción de orden parcial en sistemas distribuidos probabilísticos teniendo en cuenta que los schedulers respetan el carácter distribuido y parcialmente oculto del sistema. Estas nuevas condiciones permiten obtener mejores reducciones e incluso puede permitir la obtención de cotas más ajustadas que realizando reducción de orden parcial convencional.

El objetivo de este trabajo es implementar por primera vez la técnica de POR descripta en [17, 18], además esta implementación esta hecha sobre el model checking probabilista simbólico PRISM [25].

Capítulo 2

Modelos

A la hora de verificar el comportamiento de sistemas es necesario contar con una representación de estos para poder aplicar algoritmos que comprueben o refuten las propiedades que queremos verificar. En este trabajo vamos a utilizar modelos abstractos de los sistemas. La forma en que describiremos dichos modelos será descripta en este capítulo. El uso de modelos abstractos es importante a la hora de realizar verificaciones ya que por la naturaleza del modelado permite que la representación de los sistemas sean más pequeñas y debido a que las técnicas de model checking son exhaustivas en el espacio de estados la viabilidad del model checking es considerablemente mayor. Otro aspecto importante al trabajar con modelos abstractos es que se pueden realizar tareas de verificación antes de tener una implementación del sistema, por lo que algunos errores introducidos en la fase de diseño pueden detectarse y corregirse de manera temprana, permitiendo un ahorro tanto en el tiempo de desarrollo y, como consecuencia en los costos de este.

En este capítulo veremos cómo son los objetos matemáticos que usaremos para modelar los sistemas que estamos interesados en verificar.

2.1. Procesos de decisión de Markov

Los procesos de decisión de Markov han sido ampliamente utilizados para modelar sistemas con comportamientos no deterministas y probabilistas, en áreas como inteligencia artificial, economía y sistemas biológicos. Debido a que los sistemas que queremos modelar están compuestos de varias componentes que se ejecutan en paralelo, es posible que dado un estado del sistema exista más de una acción para ejecutar, por lo que es importante contar con no determinismo en los modelos.

Definición 1. *Un MDP es una tupla* $M = (S, Trans, enabled)$ *donde* S *es un*

conjunto finito de estados, $Trans$ es un conjunto finito de transiciones y $enabled$ es una función $S \to \mathcal{P}(Trans)$. Para cada $\alpha \in Trans$, $\alpha : S \times S \to [0,1]$ es tal que $\sum_{s' \in S} \alpha(s,s') = 1$ para todo $s \in S$ con $\alpha \in enabled(s)$.

Diremos que una acción α está habilitada en un estado s, si $\alpha \in enabled(s)$. Una acción α se llamará determinista si $\forall s, s' \in S : \alpha(s,s') \in \{0,1\}$.

2.2. Una alternativa a los MDPs

Si bien los procesos de decisión de Markov son la herramienta matemática utilizada a la hora de demostrar la corrección de algoritmos o la veracidad de teoremas, en esta sección presentaremos una nueva forma de modelar sistemas probabilistas no-deterministas que es más adecuada para modelar sistemas distribuidos.

De hecho, estos nuevos modelos permiten describir el comportamiento de cada una de las componentes del sistema distribuido. Estas componentes se ejecutan independientemente y pueden compartir información a través de variables o sincronizarse por medio de etiquetas siguiendo el estilo de CSP.

2.2.1. Estados

Cuando presentamos los MDPs los estados eran muy abstractos ya que lo único que podíamos saber es si dos estados eran diferentes o no. Si bien esto puede ser útil a la hora de tener modelos matemáticos pequeños y fáciles de manejar, especialmente a la hora de demostrar teoremas, tener estados muy abstractos a la hora de modelar puede resultar poco útil, ya que los estados representan una configuración del sistema en un momento particular, y en base a dichas configuraciones el sistema cambia de estado de una forma dada. En este sentido, consideraremos que un estado está dado por el conjunto de variables que maneja el sistema conjuntamente con los respectivos valores que estas toman en dicho estado.

Definición 2. *Sea V un conjunto finito de variables, para cada $v \in V$, D_v es un conjunto finito llamado dominio de v. Un estado en V es una función $s : V \to \biguplus_{v \in V} D_v$ tal que $s(v) \in D_v$ para todo $v \in V$. S_V es el conjunto de todos los estados en V.*

Notar que el uso de estas variables pueden modelar tanto "variables" de un lenguaje de programación como también, archivos, flags, ó simplemente estados abstractos.

Observar que, usualmente, cada una de las distintas componentes de un sistema distribuido tiene acceso limitado a una porción del estado, más precisamente, a un subconjunto de variables. Esta separación de variables que cada componente puede ver o modificar será de gran importancia a la hora de realizar reducción de orden parcial. Para poder separar e identificar la parte del estado que una componente puede manipular, nos serán muy útiles las operaciones de proyección y composición de estado.

Definición 3. *Sea s un estando en V y sea $V' \subseteq V$, $s|_{V'}$ la proyección de s en V' es la función $s|_{V'} : V' \to \biguplus_{v \in V'} D_v$ tal que $\forall v \in V' : s|_{V'}(v) = s(v)$.*

Definición 4. *Sea V, V', $V \cap V' = \emptyset$, $s \in S_V$, $s' \in S_{V'}$. El estado $s \oplus s' \in S_{V \cup V'}$ esta definido por $s \oplus s'(v) = s(v)$ si $v \in V$ y $s \oplus s'(v) = s'(v)$ si $v \in V'$.*

2.2.2. Átomos

Los sistemas que vamos a modelar están compuestos por subsistemas que pueden ser analizados de manera aislada, de forma que el sistema compuesto que modelamos sea más sencillo de comprender.

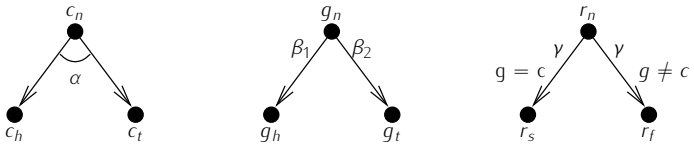
Dichos subsistemas los llamaremos átomos, un átomo va a modelar una componente aislada del sistema, esto puede ser un programa que que corre en una máquina particular, un hilo de ejecución, una componente de hardware, un ente abstracto, etc. Cada átomo va a tener su propia visión del estado global, es decir solo va a interactuar mediante la modificación o la lectura de ciertas variables del sistema global. Por otra parte para formar el sistema global los diferentes átomos van a interactuar entre si mediante el uso de variables compartidas y la sincronización.

Definición 5. *Sea V un conjunto finito de variables, L un conjunto finito de etiquetas. Un átomo es la tupla $A = (V_A, L_A, Act_A, enabled_A)$, donde $V_A \subseteq V$ es el conjunto de variables de A, $L_A \subseteq L$ es el conjunto de etiquetas de A, $Act_A \subseteq L_A \times T_A$ el conjunto de acciones de A, donde $T_A = S_{V_A} \times S_{V_A} \to [0, 1]$ y si $c \in T_A$ luego:*

$$\forall s \in S_V : \sum_{s' \in S_V} c(s, s') = 1 \tag{2.1}$$

y $enabled_A : S_{V_A} \to \mathcal{P}(Act_A)$ es la función de habilitación de A.

Los átomos son simples sistemas de transición de estados donde V son las variables que componen el sistema global, mientras que V_A son las variables a las que tiene acceso el átomo A. Al igual que para el caso de las variables L es el conjunto de etiquetas del sistema global, mientras que L_A son las etiquetas con las que sincronizan las acciones del átomo A. En cada estado hay un conjunto



(a) Átomo A (b) Átomo B (c) Átomo C

Figura 2.1: Algunos átomos de ejemplos

de acciones habilitadas que son las únicas que pueden ejecutarse cuando el átomo se encuentra en dicho estado. Las acciones son funciones que definen como se va a modificar el estado. Una acción es una tupla $\alpha = (a, c)$, que consiste en una etiqueta a que indica con que otras acciones del sistema global sincroniza y una transición probabilista c. La función $enabled$ indica cuales son las acciones que pueden ser ejecutadas en un estado dado. Diremos que la acción α esta habilitada en un estado s si $\alpha \in enabled(s)$.

Ejemplo 1. *Modelaremos un sistema simple en el cual un agente lanza una moneda y otro intenta adivinar el resultado del lanzamiento. Modelaremos este sistema con tres átomos, dos de ellos modelarán respectivamente los agentes antedichos y el tercero representará al juez que determinará si el segundo agente acertó o no al resultado. Denotaremos con A al átomo que describe el lanzamiento de la moneda. Este contiene una sola variable coin que puede tomar cualquier valor en* $\{null, head, tail\}$*. El valor null indica que no se ha arrojado aún la moneda, head indica que la moneda ha sido lanzada y su resultado fue cara. De forma similar tail indica que el lanzamiento resultó en ceca. Este átomo va a contar con una única acción probabilista cuya función de transición vaya del estado donde coin vale null a head y a tail con probabilidad* $0,5$ *Además esta acción solamente va a estar habilitada cuando coin vale null.*

El segundo átomo, que denominaremos B, modela al agente que intenta adivinar el resultado del lanzamiento de la moneda Este átomo tiene una sola variable guess que puede tomar los mismos valores que la variable coin. En este caso null indica que B todavía no realizó la predicción, head indica que el agente supone que va a salir cara, y tail indica que supone que saldrá ceca. La manera en que el agente va a decidir que lado de la moneda va salir no está determinado por nada excepto por la propia elección del agente. Este comportamiento lo vamos a modelar mediante una elección no determinista, es decir el átomo va a contar con dos acciones que van a estar habilitadas en el mismo estado (cuando guess vale null), donde una de ellas cambia con probabilidad 1 *de null a head, mientras que la segunda cambia con probabilidad* 1 *el valor*

de la variable coin a tail.

El tercer átomo , que denominaremos C, determina si el agente acertó o no el resultado del lanzamiento de la moneda. Este átomo contiene tres variables. Las dos primeras son coin y guess y son las descritas anteriormente. La tercer variables es result que puede tomar como valores a null que indica que todavía no existe un resultado, a success que indica que el agente acertó el resultado de la tirada, o a fail que indica que el agente no acertó el resultado de la tirada. Este átomo tiene dos acciones: una está habilitada cuando result es null y el valor de las variables coin y guess es igual pero distinto de null y la otra está habilitada cuando result es null y las variables coin y guess son distintas pero ninguna de las dos tiene a null como valor. En el primer caso, la transición asigna success a la variable result con probabilidad 1. *En el segundo caso, la transición asigna fail a la variable result con probabilidad* 1.

Notar que no hemos dado las etiquetas de las funciones, pero podemos pensar que son diferentes entre sí ya que no se espera sincronización entre las componentes.

En la figura 2.1 podemos observar una representación gráfica de los tres átomos.

Más adelante analizaremos como es el sistema compuesto que generan estos tres átomos.

Notar que un átomo puede verse como un MDP de la siguiente manera. Dado un átomo $A = (V_A, L_A, Act_A, enabled_A)$ el MDP que define A está definida por $M = (S, Trans, enabled)$, donde $Trans = \{c : \exists a \in L.(a, c) \in Act_A\}$ y $c \in enabled(s)$ si $\exists a \in L_A.\ (a, c) \in enabled_A(s)$.

2.2.3. Variables leídas y escritas

Dada una acción podemos diferenciar dos tipos de variables. En primer lugar están las variables que son modificadas por una acción, este tipo de variables la llamaremos *variables escritas*. En el Ejemplo 1 *coin*, *guess* y *result* son las únicas variables escritas por las acciones de los módulos A, B, y C, respectivamente. En segundo lugar están las variables que afectan la condición de habilitación de una acción, es decir, aquellas variables que se utilizan para determinar si una acción está habilitada o no en un estado. Más precisamente, son las variables que si sólo su valor fuera cambiado por una entidad externa en un estado dado, entonces la condición de habilitación de la acción cambia (i.e. pasa de deshabilitada a habilitada o viceversa). A este segundo tipo de variables las denominaremos variables leídas por la acción. En nuestro ejemplo, la única acción de A lee la variable *coin*, ambas acciones de B leen la variable

guess, y las dos acciones de C leen todas las variables (i.e., *result*, *coin* y *guess*).

Definiremos dos relaciones $Read$ y $Write$ para identificar las variables leídas y escritas por una acción. Formalmente, dado un átomo A, una acción $\alpha = (a, c)$, las relaciones $Read_A, Write_A \subseteq Act_A \times V_A$, se definen de la siguiente forma:

$$\begin{aligned} Write_A(\alpha) =& \{v \in V_A | \exists s, s' \in S_{V_A} : \alpha \in enabled_A(s) \wedge \\ & c(s, s') > 0 \wedge s(v) \neq s'(v) \in \} \end{aligned} \tag{2.2}$$

$$\begin{aligned} Read_A(\alpha) =& \{v \in V_A | \exists s \in S_{V_A \setminus \{v\}}, d, d' \in D_v : \\ & (\alpha \in enabled_A(s \oplus v \mapsto d) \wedge \alpha \notin enabled(s \oplus v \mapsto d')) \vee \\ & c(s \oplus v \mapsto d, *) \neq c(s \oplus v \mapsto d', *)\} \end{aligned} \tag{2.3}$$

La relación Var contiene a todas las variables que son sensibles a la acción, i.e.:

$$Var_A(\alpha) = Read_A(\alpha) \cup Write_A(\alpha) \tag{2.4}$$

2.2.4. Sistemas compuestos

Anteriormente vimos como modelar partes aisladas de un sistema mediante átomos, a continuación veremos como construir sistemas concurrentes complejos mediante la composición de diferentes átomos. Los sistemas compuestos van a estar formados por átomos y su estructura va a ser la de un nuevo átomo.

Sea V un conjunto finito de variables, L un conjunto finito de etiquetas de sincronización y $\mathcal{A}$ un conjunto finito de átomos donde para cada $A \in \mathcal{A}$ se tiene que $A = (V_A, S_{V_A}, L_A, Act_A, enabled_A)$ donde $V_A \subseteq V$, $L_A \subseteq L$. El sistema compuesto formado por los átomos en $\mathcal{A}$ es el átomo $\mathcal{S} = (V, S_V, L, Act, enabled)$, a continuación veremos cuales son las acciones presentes en Act y cuando estas van a estar habilitadas.

Sea $\mathcal{A}_a \subseteq \mathcal{A}$ el conjunto de átomos tal que $A \in \mathcal{A}_a \Leftrightarrow a \in L_A$, es decir todos los átomos que intervienen en la sincronización de a. Cómo $\mathcal{A}$ es finito, luego $\mathcal{A}_a$ también lo es, por lo que podemos suponer que $\mathcal{A}_a = \{A_1, \ldots, A_m\}$.

Para todo $A, B \in \mathcal{A}_a$ se debe cumplir:

$$\bigcup_{\substack{label(\alpha)=a \\ \alpha \in Act_A}} Write(\alpha) \cap \bigcup_{\substack{label(\beta)=a \\ \beta \in Act_B}} Write(\beta) = \emptyset \tag{2.5}$$

Es decir que las acciones que sincronizan deben modificar partes disjuntas del estado. Esto se debe a que si dos acciones en dos átomos distintos modificaran la misma variable con dos valores distintos, generarían una contradicción.

Luego si $c_1, \ldots, c_m$ funciones de transición de estados tales que $(a, c_i) \in Act_{A_i}$, estas generarán una acción (a, c) que va a pertenecer a Act y que esta definida según 2.2.4.

Con el fin de simplificar la definición de la transición c, introduciremos las siguientes notaciones:

- $V_i = V_{A_i}$
- $W_i = Write_{A_i}((a, c_i))$
- $W_i^c = V_i \setminus W_i$
- $W_a = \bigcup_{i=1}^{m} W_i$
- $W_a^c = V \setminus W_a$

Habiendo introducido esta notación, c queda definido de la siguiente manera:

$$c(s, s') = \begin{cases} \prod_{i=1}^{m} c_i(s|_{V_i}, s'|_{W_i} \oplus s|_{W_i^c}) & \text{si} \quad s|_{W_a^c} = s'|_{W_a^c} \\ 0 & c.c. \end{cases}$$

Conceptualmente c tiene que actuar como si todas las acciones (a, c_i) actuaran simultáneamente, luego c no puede modificar partes del estado que no sean modificadas por las otras acciones. Por otro lado al ser las ejecuciones de las acciones (a, c_i) independientes unas de otras la chance de ir de s a s' a través de c tiene que ser la misma probabilidad de que cada uno de las otras acciones modifique de la misma manera la porción del estado que escriben.

A continuación veremos que la función c que hemos definido es realmente una función de transición de estados, es decir que satisface la condición 2.1 que vimos al introducir el concepto de átomos:

$$
\begin{aligned}
\sum_{s' \in S_V} c(s, s') &= \sum_{\substack{s' \in S_V \\ s|_{W_a^c} = s'|_{W_a^c}}} \prod_{i=1}^{m} c_i(s|_{V_i}, s'|_{W_i} \oplus s|_{W_i^c}) \\
&= \sum_{s'_1 \in S_{W_1}} \cdots \sum_{s'_m \in S_{W_m}} \prod_{i=1}^{m} c_i(s|_{V_i}, (s|_{W_a^c} \oplus \bigoplus_{i=0}^{m} s'_i)|_{W_i} \oplus s|_{W_i^c}) \\
&= \sum_{s'_1 \in S_{W_1}} \cdots \sum_{s'_m \in S_{W_m}} \prod_{i=1}^{m} c_i(s|_{V_i}, s'_i \oplus s|_{W_i^c}) \\
&= \prod_{i=1}^{m} \sum_{s'_i \in S_{W_i}} c_i(s|_{V_i}, s'_i \oplus s|_{W_i^c}) \\
&= \prod_{i=1}^{m} \sum_{s'_i \in S_{V_i}} c_i(s|_{V_i}, s'_i) \\
&= \prod_{i=1}^{m} 1 \\
&= 1
\end{aligned}
$$

Finalmente queda por definir relación $enabled_{\mathcal{A}}$. Para ello, notar que para que una acción en el sistema compuesto esté habilitada en un estado dado necesariamente cada una de las acciones que la componen debe estar habilitada en sus respectivos átomos. Es decir, si $\alpha_1 \in Act_{A_1}, \ldots, \alpha_m \in Act_{A_m}$ son tales que generan a α en el sistema compuesto, luego:

$$\alpha \in enabled_{\mathcal{A}}(s) \Leftrightarrow \forall i : \alpha_i \in enabled_{A_i}(s|_{V_{A_i}}) \tag{2.6}$$

En caso de que en un estado no exista ninguna acción que puede ejecutarse se habilitara una acción acción ζ que con probabilidad 1 permanece en el mismo estado.

Ejemplo 2. *Retomando el Ejemplo 1 en el que un agente intenta adivinar cara o ceca en el lanzamiento de una moneda, mostramos aquí el resultado de la composición entre los tres átomos dados en la Figura 2.1. Debido a que las etiquetas de las acciones las habíamos supuesto disjuntas, no existe ninguna sincronización entre átomos. Es decir que el nuevo sistema es el resultado de juntar todas las acciones de los átomos que la conforman, con la salvedad que dichas acciones son extendidas para tener un dominio en el estado global. Una representación gráfica del sistema puede observarse en la figura 2.2a.*

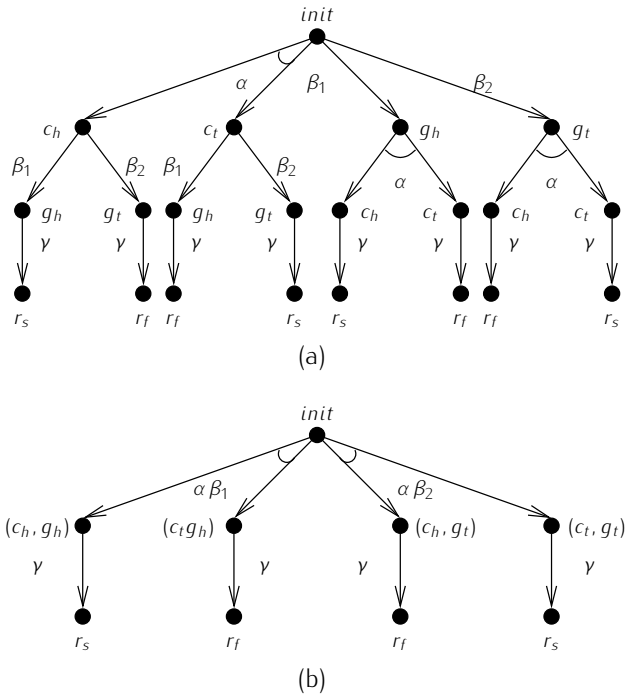


Figura 2.2: Composición de los átomos A, B y C

Es interesante analizar el caso en que las acciones del átomo que arroja la moneda y el que intenta adivinar el resultado tienen la misma etiqueta i.e. $label(\alpha) = label(\beta_1) = label(\beta_2)$*. Una representación gráfica de este nuevo sistema esta dada en la figura 2.2b. La diferencia de comportamiento de ambos sistemas compuestos será analizada en el capítulo 3 cuando veamos como se resuelven las elecciones no deterministas en la ejecución de un sistema.*

Dado que los sistemas compuestos son también átomos, las definiciones de variables escritas y leídas por las acciones es igualmente válida en este nuevo contexto. Notar, a su vez, que las variables leídas (o escritas) por una acción α en el sistema compuesto tienen una relación concreta con las variables leídas (o escritas) por las acciones en los átomos originales que generan la acción α. Más precisamente:

$$Read_{\mathcal{A}}(\alpha) \subseteq \bigcup_{A \in \mathcal{A}_{label(\alpha)}} Read_A(\alpha_A)$$

$$Write_{\mathcal{A}}(\alpha) \subseteq \bigcup_{A \in \mathcal{A}_{label(\alpha)}} Write_A(\alpha_A)$$

donde los α_A son las acciones que definen a α.

2.3. Condiciones adicionales

Si analizamos los sistema compuestos, mirando el estado global podemos inferir cuales son las acciones del sistema global que están habilitadas en dicho estado. Ahora bien si tomamos un átomo de manera aislado y miramos únicamente su estado local, es probable que no sepamos cuales de sus acciones están habilitadas para intervenir en una sincronización en el sistema global. Esto se debe a que una acción sincrónica está habilitada en un estado global, si esta está habilitada en cada uno de los átomos que la conforman.

Cuando veamos el comportamiento de los *schedulers* en el siguiente capítulo vamos a necesitar que los átomos puedan inferir cuáles de sus acciones están habilitadas en el estado global, mirando solo su estado local. Para lograr dicha propiedad vamos a introducir una nueva condición a los átomos que conforman el sistema compuesto. Para todo átomo $A \in \mathcal{A}$ y todo estado $s \in S_V$ se debe satisfacer:

$$(a, c_A) \in enabled_A(s|_{V_A}) \Leftrightarrow \forall B \in \mathcal{A}_a \exists (a, c_B) \in Act_B : enabled_A(s|_{V_B}) \quad (2.7)$$

Es decir que la habilitación de una etiqueta es independiente del átomo. Esto implica que dada una etiqueta de sincronización todas las acciones locales deben leer las mismas variables para definir su habilitación. Dado un conjunto de átomos que satisfacen todas las condiciones para formar un sistema compuesto a excepción de 2.7. Se puede obtener un nuevo conjunto de átomos que generan un sistema equivalente (i.e. que genere el mismo MDP). Para ello vamos a definir los átomos duales. Sea A un átomo el átomo dual $\tilde{A}$ esta definido por:

- $L_{\tilde{A}} = L_A$
- $V_{\tilde{A}} = V_A \cup \bigcup_{a \in L_A} Read_{\mathcal{A}}(a)$
- Si $(a, c) \in Act_A$ luego $(a, \tilde{c}) \in Act_{\tilde{A}}$ donde:

$$\tilde{c}(s, t) = \begin{cases} c(s|_{V_A}, t|_{V_A}) & s|_{V_{\tilde{A}} \setminus Write_A((a,c))} = t|_{V_{\tilde{A}} \setminus Write_A((a,c))} \\ 0 & c.c. \end{cases}$$

- Para todo $s \in S_V$, $(a, c_{\tilde{A}}) \in Act_{\tilde{A}}$ se cumple:

$$(a, c_{\tilde{A}}) \in enabled_{\tilde{A}}(s|_{V_{\tilde{A}}}) \Leftrightarrow (a, c_A) \in enabled_A(s|_{V_A}) \wedge \exists (a, c_{\mathcal{A}}) \in Act_{\mathcal{A}} : (a, c) \in enabled_{\mathcal{A}}(s)$$

Notar que si las acciones $\alpha_i, \ldots, \alpha_m$ generan la acción α en $\mathcal{A}$ y $\tilde{\alpha}_i, \ldots, \tilde{\alpha}_m$ generan la acción $\tilde{\alpha}$ en $\tilde{\mathcal{A}}$ luego $\alpha = \tilde{\alpha}$

Capítulo 3

Schedulers

En el capítulo anterior vimos como construir sistemas probabilistas no deterministas. En este capítulo veremos que el comportamiento de un sistema (modelado como un conjunto de átomos) está definido por las ejecuciones que este realiza, i.e., por las secuencias de acciones que se ejecutan y estados que se atraviesan. A lo largo de la ejecución deben realizarse dos tipos de elecciones que determinan precisamente la ejecución que se está realizando. Por un lado están las elecciones probabilistas. Estas surgen al ejecutar acciones que incluyen una decisión probabilista. En este caso la chance de alcanzar un estado dado está determinada por una distribución de estados que es conocida a la hora de modelar el sistema. Por ejemplo, esto se da en casos donde el sistema basa su elección mediante la elección de un número aleatorio o en casos donde el entorno ha sido estudiado y se conoce con detalle su comportamiento aleatorio como puede ser en el modelado de un canal de comunicación con pérdida de paquetes donde la probabilidad de perder un paquete es conocida. Por otro lado están las elecciones no deterministas. Estas se dan en puntos de indeterminación donde el sistema encuentra múltiples opciones (más precisamente, acciones) que elegir. La forma en la que se rigen tales elecciones no son conocidas en general y los criterios de elección pueden variar para cada ejecución e, inclusive, a lo largo de ésta. De esta manera, cada forma de resolución del no-determinismo está dada por una función que, por cada ejecución parcial, devuelve la próxima acción a ejecutar y, en todo caso, también la probabilidad con que ésta debe realizarse. Tal función se denomina *scheduler*. En este capítulo presentamos dos clases de schedulers diferentes. En primer lugar presentamos los *schedulers de información total*. Estos schedulers realizan sus elecciones usando todo lo transcurrido a lo largo de la ejecución del sistema hasta el momento de resolver el no-determinismo. En segundo lugar veremos a los *schedulers fuertemente distribuidos*, los cuales realizan sus elecciones mirando solo una porción de la ejecución global.

3.1. Distintos tipos de no-determinismo

Dentro de las elecciones no deterministas podemos diferenciar dos tipos de no-determinismo. En primer lugar está el *no-determinismo de interleaving* ligado a la ejecución concurrente de los átomos: en un momento dado del sistema puede haber más de un átomo habilitado para ejecutarse. Es decir este tipo de no-determinismo es el que modela el *interleaving* de los átomos en la ejecución del sistema. Este comportamiento está influido fuertemente por el contexto de ejecución del programa. Si trabajamos con sistemas de computadoras algunos factores que pueden intervenir son, por ejemplo, el planificador del sistema operativo, o si los átomos son ejecutados en diferentes computadoras, la red utilizada. Todos estos factores son habitualmente desconocidos a la hora de realizar el modelado, incluso pueden variar con el tiempo.

El segundo tipo de no-determinismo que podemos diferenciar es el *no-determinismo propio de cada componente*. Éste está dado cuando dentro de un mismo átomo existen más de una acción habilitada para ejecutar. Cuando se realiza el modelado del sistema este tipo de no-determinismo puede surgir debido a que el átomo esta pensado como un conjunto de procesos, o como un proceso con varios hilos de ejecución, que igual que el no-determinismo de interleaving su comportamiento esta ligado al ambiente. Otra posibilidad surge cuando se abstrae de las posibles formas de implementar un sistema. Un ejemplo concreto de ello puede darse en un átomo que atiende pedidos de dos colas, pero la manera en que éste atiende los pedidos no está especificada, dando lugar así al no-determinismo. Ahora bien a la hora de implementar el sistema, el programador puede elegir atender siempre los pedidos de una cola en particular; elegir una cola al azar; lanzar una moneda ponderada de acuerdo a la cantidad de pedidos que hay en cada una de las colas; ó simplemente determinar la cola según el valor de un registro o el ruido de la red. Cada una de las implementaciones va a utilizar un scheduler diferente.

3.2. Caminos y proyecciones

La noción de ejecuciones en nuestros sistemas van estar dadas por caminos. Los caminos son sucesiones de estados, donde se indican cuales fueron las acciones que permitieron el cambio de estado y cual fue el átomo que ejecuto la acción.

Definición 6. *Dado un sistema compuesto $\mathcal{A}$, un camino de $\mathcal{A}$ es una sucesión finita de la forma $s_0.\alpha_0.A_0.s_1, \ldots, s_{n-1}.\alpha_{n-1}.A_{n-1}.s_n$ tal que:*

- $s_i \in S_V$

- $\alpha_i \in Act$
- $label(\alpha_i) \in L_{A_i}$

El conjunto de todos los caminos de $\mathcal{A}$ lo denotaremos por $Path_{\mathcal{A}}$.

Notar que los caminos pueden modelar ejecuciones que no son válidas en el sistema. Esto ocurre si existe algún α_i tal que $\alpha_i(s_i, s_{i+1}) = 0$. Así como definimos a las ejecuciones del sistema compuesto podemos definir cuales son las ejecuciones observadas por los átomos.

Definición 7. *Dado un sistema compuesto $\mathcal{A}$ y un átomo $A \in \mathcal{A}$ un camino de A es una sucesión finita de la forma $s_0.\alpha_0.A_0.s_1, \ldots, s_{n-1}.\alpha_{n-1}.A_{n-1}.s_n$ tal que:*

- $s_i \in S_{V_A}$
- $\alpha_i \in Act_A$ *si* $label(\alpha_i) \in L_A$
- $\alpha_i = (a, id_A)$ *si* $a = label(\alpha_i) \notin L_A$

donde id_A esta dada por:

$$id_A(s,t) = \begin{cases} 1 & si\ s = t \\ 0 & c.c. \end{cases}$$

Denominamos por $Path_A$ al conjunto de todos los caminos de A y $Path_A^{en}$, a todos los caminos σ de A tales que $|enabled_A(last(\sigma)) \neq \emptyset|$. Por último $Path_{A,a}^{en}$ es el conjunto de todos los caminos σ de A que estén habilitados tales que exista $\alpha \in enabled(last(\sigma))$ tal que $label(\alpha) = a$.

Dado una ejecución en el sistema compuesto podemos obtener la ejecución observada por un átomo mediante el uso de proyecciones:

Definición 8. *Dado un sistema compuesto $\mathcal{A}$ y un átomo $A \in \mathcal{A}$ la proyección de caminos de $\mathcal{A}$ en A es la función $[]_A : Path_{\mathcal{A}} \to Path_A$ definida por:*

- $[s]_A = s|_A$.
- $[\sigma.\alpha.B.s]_A = [\sigma]_A$ *si* $Write_{\mathcal{A}}(\alpha) \cap V_A = \emptyset$ *y* $label(\alpha) \notin L_A$.
- $[\sigma.\alpha.B.s]_A = [\sigma]_A.(label(\alpha), id_A).B.s|_{V_A}$ *si* $Write_{\mathcal{A}}(\alpha) \cap V_A \neq \emptyset$ *y* $label(\alpha) \notin L_A$.
- $[\sigma.\alpha.B.s]_A = [\sigma]_A.\alpha_A.B.s|_{V_A}$ *si* $Write_{\mathcal{A}}(\alpha) \cap V_A \neq \emptyset$, $label(\alpha) \in L_A$ *y* α_A *es la acción de A que interviene en la sincronización de α.*

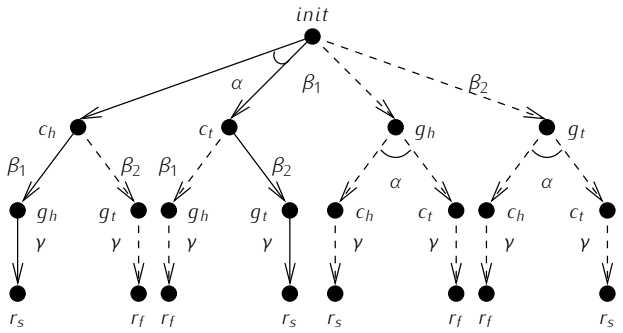


Figura 3.1: Ejemplo de scheduler del sistema de la figura 2.2a.

3.3. Schedulers de información total

Un scheduler de información total es un scheduler que no tiene ninguna restricción para elegir la próxima acción a ejecutar. Particularmente resuelve el no-determinismo de interleaving y el no-determinismo de las componentes observando la ejecución global del sistema.

Definición 9. *Dado un sistema compuesto $\mathcal{A}$ un scheduler de información total es una función $\eta : Path^{en}_{\mathcal{A}} \times Act_{\mathcal{A}} \times \mathcal{A} \longmapsto [0,1]$ tal que:*

- $\eta(\sigma, \alpha, A) > 0 \Rightarrow \alpha \in enabled_{\mathcal{A}}(last(\sigma)) \wedge label(\alpha) \in L_A$.
- $\sum_{\alpha,A} \eta(\sigma, \alpha, A) = 1$.

En el capítulo anterior vimos como ejemplo el problema de la moneda (Ej.1) un scheduler de información total es el siguiente:

$$\eta(s_{n,n,n}, \alpha, A) = 1 \qquad \eta(s_{n,n,n}.\alpha.A.s_{t,n,n}, \beta_1, B) = 1$$
$$\eta(s_{n,n,n}.\alpha.A.s_{h,n,n}, \beta_2, B) = 1 \qquad \eta(s_{n,n,n}.\alpha.A.s_{*,n,n}.\beta_*.B.s_{*,*,n}, \gamma, C) = 1$$

Notar que hemos obviado algunos caminos que no pueden ser generados por el scheduler como ser $s_{n,n,n}.\beta_1.B.s_{n,t,n}$ ya que en el estado inicial la única acción que se puede ejecutar es α. En la figura 3.1 puede observarse una representación gráfica del scheduler. Las líneas sólidas indican las elecciones realizadas por el scheduler. Observar que en todos los caminos generados por el scheduler la variable res que pertenece al átomo C nunca toma como valor $fail$. Esto indica que que bajo este scheduler el átomo B siempre adivina el lanzamiento efectuado por A. Los schedulers como el anterior donde la probabilidad de elegir una acción es 0 ó 1 son denominados *schedulers deterministas*, los demás son llamados *schedulers randomizados*.

3.4. Schedulers fuertemente distribuidos

Los schedulers fuertemente distribuidos son un subconjunto propio de los schedulers de información total. Estos schedulers resuelven el no-determinismo de las componentes observando solo la ejecución observada por la componente. El no-determinismo de interleaving se resuelve observando la ejecución global, aunque con algunas restricciones para impedir revelar información.

Los schedulers fuertemente distribuidos son modelados mediante el uso de tres clases diferentes de subschedulers: los schedulers de interleaving, los schedulers generativos y schedulers reactivos. Los schedulers de interleaving seleccionan el próximo átomo que va a ejecutarse. Los schedulers generativos eligen cual de las acciones habilitadas en el átomo activo va a ejecutarse provisto que el scheduler de interleaving designó esto átomo para ser ejecutado. Esta decisión es tomada considerando sólo la ejecución observada por dicho átomo. Los schedulers reactivos resuelven el no-determinismo interno que se produce en un átomo con acciones de igual nombre. Esto es, dado que una acción con una etiqueta a fue producida por otro átomo (i.e. el átomo elegido por el scheduler de interleaving) el scheduler reactivo resuelve con cual de las acciones etiquetada con a va a sincronizar. Al igual que los schedulers generativos, los schedulers reactivos deciden teniendo en cuenta los caminos locales de los átomos. Observar que los schedulers de interleaving son los encargados de resolver el no-determinismo de interleaving. Mientras que los schedulers generativos y reactivos son los encargados de resolver el no-determinismo de las componentes.

Definición 10. *Dado un sistema compuesto $\mathcal{A}$ un scheduler de interleaving va a ser una función $\mathcal{I} : Path^{en}_{\mathcal{A}} \times \mathcal{A} \mapsto [0,1]$ tal que:*

- $\mathcal{I}(\sigma, A) > 0 \Rightarrow \exists \alpha \in Act_{\mathcal{A}} : \alpha \in enabled_{\mathcal{A}}(last(\sigma)) \wedge label(\alpha) \in L_A$
- $\forall \sigma, \sigma' \in Path^{en}_{\mathcal{A}} : [\sigma]_A = [\sigma']_A \wedge [\sigma]_B = [\sigma']_B \Rightarrow$

$$\frac{\mathcal{I}(\sigma, A)}{\mathcal{I}(\sigma, A) + \mathcal{I}(\sigma, B)} = \frac{\mathcal{I}(\sigma, B)}{\mathcal{I}(\sigma, A) + \mathcal{I}(\sigma, B)}$$

- $\sum_{A \in \mathcal{A}} \mathcal{I}(\sigma, A) = 1$

Definición 11. *Dado un sistema compuesto $\mathcal{A}$ y un átomo $A \in \mathcal{A}$ un scheduler generativo de A es una función $\Theta_A : Path^{en}_A \times Act_A \mapsto [0,1]$ tal que:*

- $\Theta_A(\sigma, \alpha) > 0 \Rightarrow \alpha \in enabled_A(last(\sigma))$

- $\sum_{\alpha \in Act_A} \Theta_A(\sigma, \alpha) = 1$

Definición 12. *Dado un sistema compuesto $\mathcal{A}$ y un átomo $A \in \mathcal{A}$ y una etiqueta de sincronización $a \in L_A$ un scheduler reactivo de A y a es una función $\Upsilon_A^a : Path_A^{en} \times Act_A \mapsto [0,1]$ tal que:*

- $\Upsilon_A^a(\sigma, \alpha) > 0 \Rightarrow \alpha \in enabled_A(last(\sigma)) \wedge label(\alpha) = a$
- $\sum_{\alpha \in Act_A \wedge label(\alpha) = a} \Upsilon_A^a(\sigma, \alpha) = 1$

Un scheduler fuertemente ditribuido es la composición de los schedulers definidos anteriormente:

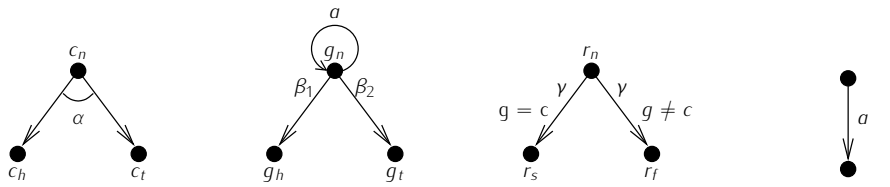
Definición 13. *Sea $\mathcal{A}$ un sistema compuesto. Sea $\mathcal{I}$ un scheduler de interleaving para $\mathcal{A}$ y, para todo átomo $A \in \mathcal{A}$ y etiqueta $a \in A$, sean Θ_A y Υ_A^a El scheduler fuertemente distribuido $\eta : Path_\alpha^{en} \times Act_{\mathcal{A}} \times \mathcal{A} \mapsto [0,1]$ generado por estos sub-schedulers esta dado por:*

$$\eta(\sigma, \alpha, A) = \mathcal{I}(\sigma, A) \cdot \Theta_A([\sigma]_A, \alpha_A) \cdot \prod_{\substack{B \in \mathcal{A} \setminus \{A\} \\ label(\alpha) \in L_B}} \Upsilon_A^{label(\alpha)}([\sigma]_B, \alpha_B) \tag{3.1}$$

donde α_A y las α_C son las acciones que generan la acción compuesta α.

Observar que el scheduler de información total dado como ejemplo al final de la sección anterior no es un scheduler fuertemente distribuido. El scheduler generativo de B no puede percibir los cambios en el estado de la moneda.

La condición del scheduler de interleaving es necesaria para evitar que el scheduler revele información del estado global a los átomos. En la figura 3.2 se puede ver un sistema compuesto, que es una modificación del ejemplo 1. En este nuevo sistema se añade un nuevo átomo determinista que ejecuta una única acción. Se modifica el átomo B para que sincronice con dicha acción en el estado inicial. Notar que todos los átomos excepto B son deterministas. Sólo es necesario definir el scheduler generativo de B y el scheduler de interliving. El scheduler generativo de B elige cara si con anterioridad se ejecuto la acción con label a, caso contrario elige ceca. Observar que es un scheduler válido ya que B sincroniza con a (i.e. $a \in L_B$). El scheduler de interleaving en el estado inicial escoge al átomo A. A continuación escoge al átomo D seguido de B y C si el resultado del lanzamiento de la moneda es cara. Si el resultado del lanzamiento es ceca escoge al átomo B seguido de C. Notar que este scheduler siempre con probabilidad 1 el resultado de la moneda. Observar que los átomos



(a) Átomo A (b) Átomo B (c) Átomo C (d) Átomo D

Figura 3.2: Ejemplo de sistema donde es necesario restringir a los schedulers de interleaving.

B y D al no acceder a la variable *coin* no modifican sus estados locales luego del lanzamiento de la moneda. Por consiguiente el scheduler de interleaving no satisface la condición 3.1.

3.5. Caminos, schedulers y probabilidades

Un scheduler puede generar más de un camino posible. Esto depende de las elecciones probabilistas que realiza el sistema, así como también las elecciones probabilista que realiza el mismo scheduler. Dado un scheduler η y un camino finito σ se puede obtener la probabilidad de el scheduler genere dicho camino de la siguiente manera, si suponemos que el estado inicial es s_{init}:

$$Pr^{\eta}(s) = \begin{cases} 1 & \text{si } s = s_{init} \\ 0 & \text{c.c.} \end{cases}$$

$$Pr^{\eta}(\sigma.\alpha.A.s) = Pr^{\eta}(\sigma)\eta(\sigma, A)\alpha(last(\sigma), s)$$

Los sistemas al tener siempre una acción habilitada en todos los estados generan ejecuciones infinitas. La noción de probabilidad de un camino puede extenderse a los caminos infinitos utilizando el teorema de extensión de Carathéodory [3]. Es decir si tenemos un conjunto H de caminos infinitos podemos obtener $Pr^{\eta}(H)$.

Capítulo 4

Propiedades

En los capítulos anteriores vimos como modelar sistemas distribuidos mediante la composición de diferentes átomos. También vimos como resolver las elecciones no deterministas mediante el uso de *schedulers*. En este capítulo veremos distintas formas de expresar propiedades del comportamiento de los sistemas compuestos. Más precisamente veremos como expresar propiedades referidas a la historia de ejecución de un sistema.

Existen varias lógicas modales que permiten expresar distintos tipos de propiedades. En este trabajo nos centraremos en las propiedades LTL.

4.1. Lógica temporal lineal

La lógica temporal lineal ó simplemente LTL, es una lógica modal desarrollada a finales de los setentas por Pnueli[27], esta lógica sirve para describir ciertas propiedades en sistemas reactivos. La lógica temporal LTL considera a la ejecución del sistema como una sucesión de estados, donde no se tiene en cuenta tantos las acciones como los átomos que intervienen en la ejecución como cuando vimos los schedulers. LTL extiende a la lógica proposicional con operadores modales. Estos operadores modales permiten describir cómo se suceden los cambios en los estados a lo largo de una ejecución.

4.2. Sintaxis

Dado V un conjunto de variables, podemos definir las fórmulas LTL de la siguiente manera:

- $True$, $False$ son fórmulas.

- $(v = d)$ es una fórmula para toda $v \in V$, $d \in D_v$.
- $\neg\phi$ es una fórmula para cualquier fórmula ϕ.
- $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$ son fórmulas para cualesquiera fórmulas ϕ, ψ.
- $\mathsf{G}\,\phi$, $\mathsf{F}\,\psi$, $\mathsf{X}\,\phi$ son fórmulas para cualesquier fórmula ϕ.
- $\phi\,\mathsf{U}\,\psi$ es una fórmula para cualesquiera fórmulas ϕ, ψ.

Informalmente $\mathsf{G}\phi$ se va satisfacer si ϕ se satisface durante toda la ejecución, $\mathsf{F}\phi$ se va satisfacer si ϕ se satisface en algún lugar de la ejecución, $\mathsf{X}\phi$ se va satisfacer si ϕ se satisface luego del estado inicial, por último $\phi\,\mathsf{U}\,\psi$ se va satisfacer si existe un tramo inicial donde ϕ siempre se satisface e inmediatamente después ψ se satisface.

4.3. Semántica

Definición 14. *Sea $\sigma = s_0 s_1 s_2 \ldots$ una sucesión infinita de estados, los siguientes operadores sobre σ:*

(i) $\sigma[i] = s_i$

(ii) $\sigma[i\ldots] = s_i s_{i+1} \ldots$

A continuación vamos a dar la semántica de algunas fórmulas LTL, el resto de las fórmulas se definen en términos de estas.

- $\sigma \vDash True$ siempre es válida.
- $\sigma \vDash (v = d)$ si y solo si $\sigma[0](v) = d$.
- $\sigma \vDash \neg\phi$ si y solo si $\sigma \nvDash \phi$.
- $\sigma \vDash \phi \wedge \psi$ si y solo si $\sigma \vDash \phi$ y $\sigma \vDash \psi$.
- $\sigma \vDash \mathsf{X}\,\phi$ si y solo si $\sigma[1\ldots] \vDash \phi$
- $\sigma \vDash \phi\,\mathsf{U}\,\psi$ si y solo si $\exists\, i \geq 0 :\ \sigma[i\ldots] \vDash \psi$ y $\forall\, j :\ 0 \leq j < i :\ \sigma[j\ldots] \vDash \phi$

Los otros operadores se definen a partir de los anteriores como sigue:

- $False = \neg True$
- $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$

- $\phi \rightarrow \psi = \neg(\phi \wedge \neg\psi)$
- $\mathsf{F}\,\phi = True\,\mathsf{U}\,\phi$
- $\mathsf{G}\,\phi = \neg(True\,\mathsf{U}\,\neg\phi)$

Cuando introduzcamos la noción de reducción de orden parcial veremos que las fórmulas que queremos comprobar no deben tener el operador X para poder realizar la reducción, por lo que en lo que queda de este informe no hablaremos del mismo.

Las fórmulas LTL son propiedades sobre sucesiones de estados, mientras que el comportamiento de un sistema está dado por ejecuciones. Las ejecuciones de los sistemas pueden verse como sucesiones de estados, pero con información adicional como ser las acciones que se ejecutaron para cambiar de estado. Dado una ejecución podemos obtener su sucesión de estados de la siguiente forma:

Definición 15. *Dado un sistema compuesto $\mathcal{A}$ luego $states : Path_{\mathcal{A}} \longmapsto S_V^{\omega}$ esta dada por $states(s_0.\alpha_0.A_0.s_1, \ldots) = s_0.s_1 \ldots$.*

Dada una fórmula LTL ϕ y una ejecución π diremos que π satisface ϕ, que denotaremos por $\pi \vDash \phi$, si la sucesión de estados que define la ejecución π satisface ϕ, i.e. si se cumple $states(\pi) \vDash \phi$. Dado un sistema compuesto, un scheduler y un estado inicial pueden existir más de una ejecución posible. Estas ejecuciones están dadas por las diferentes elecciones que realiza el scheduler así como también las elecciones probabilistas que realizan las acciones. Luego no necesariamente todas las ejecuciones que definen un sistema junto con un scheduler van a satisfacer una fórmula LTL dada. Solo lo harán una porción de las ejecuciones. En 3.5 vimos como dada una ejecución podíamos obtener la probabilidad de ocurrencia de dicha ejecución en un scheduler dado. Con esa noción podemos obtener la probabilidad de que una fórmula LTL ϕ sea válida en un sistema compuesto $\mathcal{A}$, un scheduler η y un estado inicial s_0 como sigue:

$$Pr^{\eta}(\phi) = Pr^{\eta}(\{\pi \in Path_{\mathcal{A}}^{inf} : states(\pi) \vDash \phi\}) \tag{4.1}$$

Recordemos el problema de la moneda visto en el Ejemplo 1. Donde un átomo lanzaba una moneda y otro átomo intentaba adivinar. En el capítulo anterior se dio como ejemplo un scheduler de información total donde siempre se podía adivinar el resultado del lanzamiento de la moneda, de forma análoga había un scheduler que nunca adivinaba el lanzamiento y por último se vio un scheduler cuya chance de adivinar el lanzamiento era de $\frac{1}{2}$. Luego si queremos verificar la fórmula $\mathsf{F}(res = success)$ si utilizamos el primer scheduler la probabilidad

de que la formula sea válida es 1, con el segundo la probabilidad es 0 y con el último dicha probabilidad es de 0,5. Debido a que el entorno de ejecución de un sistema es por lo general desconocido cuando se modela dicho sistema, calcular la probabilidad de validez de una fórmula en un scheduler dado no es útil. Dicho scheduler puede no ser el que modele el verdadero entorno. Además la probabilidad de validez de la fórmula a verificar en el scheduler dado puede ser distante de la probabilidad real. Es decir de la probabilidad del scheduler que modela correctamente el entorno de ejecución. En su lugar es requerido conocer cuales son las probabilidades máximas y mínimas de que una fórmula sea válida. Estas probabilidades garantizan el rango de validez de la fórmula en el sistema que se modela, sin importar cual es el verdadero ambiente de ejecución.

Capítulo 5

Reducción de orden parcial

Uno de de los principales problemas del model checking es la explosión de estados. Si pensamos que los sistemas son similares a los presentados en los capítulos anteriores, i.e. un conjunto de subsistemas que se ejecutan en paralelo. La cantidad de estados necesarios para representar al sistema crece exponencialmente cuando añadimos más componentes y más variables al sistema. Precisamente, el objetivo de la reducción de orden parcial es atacar la explosión que se produce en el espacio de estado. Esta técnica aprovecha la estructura del sistema compuesto eliminando estados y transiciones redundantes introducidos por el interleaving de las componentes. y manteniendo estados y transiciones "representativos". Por representativos queremos decir que dado un camino en el modelo original que es relevante para la propiedad bajo estudio, entonces el modelo reducido tiene un camino (el mismo u otro) que es equivalente desde el punto de vista de la propiedad. Para asegurar que los estados y las transiciones representativas permanecen en el modelo reducido, esté debe satisfacer ciertas condiciones. Primero analizaremos las condiciones necesarias en los sistemas no probabilistas (que son un caso particular de los sistemas probabilistas), ya que los argumentos utilizados a la hora de realizar las reducciones son similares. Luego veremos las condiciones adicionales que son necesarias cuando se tienen elecciones probabilistas.

No es el objetivo de este trabajo demostrar que las condiciones son completas, i.e. que realmente generan un sistema equivalente. Dichas demostraciones se las pueden encontrar en [18, 4, 15]. Solo se darán ejemplos que justifiquen dichas reglas.

5.1. Independencia e Invisibilidad

Uno de los conceptos fundamentales a la hora de realizar reducción de orden parcial el la "independencia" entre las acciones. Dos acciones son independientes si la ejecución de una acción no interfiere con la ejecución de la otra.

Definición 16. *Sean α y β dos acciones diferentes de un sistema $\mathcal{A}$, diremos que son independientes si:*

$$Write_{\mathcal{A}}(\alpha) \cap Var_{\mathcal{A}}(\beta) = \emptyset$$
$$Write_{\mathcal{A}}(\beta) \cap Var_{\mathcal{A}}(\alpha) = \emptyset$$
$$\mathcal{A}_{label(\alpha)} \cap \mathcal{A}_{label(\beta)} = \emptyset$$

Dos acciones son dependientes si no son independientes. Vale la pena notar que la relación de independencia es antirreflexiva y simétrica. Denotaremos con $dep \subseteq Act_{\mathcal{A}} \times Act_{\mathcal{A}}$ a la relación de dependencia.

Dos acciones independientes cumplen con dos propiedades importantes a la hora de realizar la reducción: *no interferencia* y *conmutatividad*. La propiedad de no interferencia garantiza que la ejecución de una acción no habilita ni deshabilita a otra acción independiente. Esto se debe a que las variables que determinan la habilitación de una acción (que están incluidas en la relación $Read$) y las variables escritas por la otra acción son disjuntas. La propiedad de conmutatividad asegura que el estado resultante después de la ejecución de dos acciones independientes es el mismo sin importar el orden en que se ejecutaron. Esto se debe a dos factores, el primero es que las variables que escriben ambas acciones son disjuntas, luego no puede haber una superposición de valores y en segundo lugar las variables que determinan el estado resultante y las variables modificadas difieren, por lo que al ejecutarse la primera acción no se alteran las variables que usa la segunda acción para cambiar de estado.

En la figura 5.1 se puede observar un ejemplo de dos acciones independientes. Si los valores de los estados s_1 y s_2 no son relevantes o de alguna manera son "equivalentes", da igual cual de las dos acciones se eligen en s, pues el estado final en ambos casos es t. Notar que no siempre se puede eliminar una de las transiciones pues en los estados $s1$ o s_2 pueden ser sensibles a la fórmula que queremos verificar. Otra posibilidad es que que al eliminar alguno de los estados del sistema se elimine también la posibilidad de ejecutar algunas acciones que pueden ser importantes en la verificación.

La noción de independencia presentada aquí es más fuerte que las nociones presentadas en otros trabajos como ser [15, 5]. esto se debe al uso de los

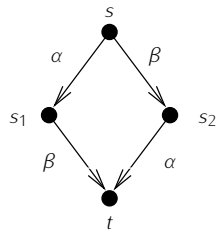


Figura 5.1: Ejemplo de dos acciones independientes

schedulers fuertemente distribuidos. Sin embargo las heurísticas presentadas en dichos trabajos son similares a nuestra noción de independencia.

El otro concepto importantes a la hora de realizar reducciones son las denominadas acciones invisibles:

Definición 17. *Dado un conjunto de variables V y una acción α diremos que α es invisible con respecto a V si $Write(\alpha) \cap V = \emptyset$.*

Es decir que una acción es invisible con respecto a un conjunto de variables si al ejecutarse no altera esa porción del estado.

5.2. Stuttering

Dado un conjunto de variables $V' \subseteq V$ y dos sucesiones de estados infinitos en V $\sigma = s_0.s_1.s_2\ldots$ y $\rho = r_0.r_1.r_2\ldots$ diremos que son *equivalentes bajo stuttering*, lo cual denotaremos por $\sigma \sim_{st,V'} \rho$, si existen dos sucesiones de enteros positivos $0 = i_0 < i_1 < i_2\ldots$ y $0 = j_0 < j_1 < j_2\ldots$ tal que para todo $k \geq 0$ se cumpla:

$$s_{i_k}|_{V'} = s_{i_k+1}|_{V'} = \ldots s_{i_{k+1}-1}|_{V'} = r_{j_k}|_{V'} = r_{j_k+1}|_{V'} = \ldots r_{j_{k+1}-1}|_{V'}$$

Las secuencias finitas de estados idénticos bajo las variables que están en V' los llamaremos bloques. Intuitivamente dos sucesiones de estados son equivalentes bajo stuttering si pueden ser particionados en una cantidad infinita de bloques tal que en el k-ésimo bloque de ambas sucesiones las variables que están en V' tienen el mismo valor en los estados del mismo bloque. Notar que los bloques pueden tener un tamaño diferente.

Precisamente, el sistema obtenido luego de realizar reducción de orden parcial preserva todos los caminos del sistema original módulo equivalencia bajo stuttering [26]. El siguiente resultado garantiza entonces que la reducción de

orden parcial preserva la validez de propiedades LTL sin el operador next (i.e. $LTL_{\setminus X}$). En el caso no probabilista.

Diremos que una fórmula LTL φ es *invariante bajo stuttering* si para cada par de sucesiones infinitas σ y σ' tal que $\sigma \sim_{st,V_\varphi} \sigma'$:

$$\varphi \vDash \sigma \text{ si y solo si } \varphi \vDash \sigma'$$

El siguiente resultado se debe a Leslie Lamport [23]

Teorema 1. *Toda fórmula $LTL_{\setminus X}$ es invariante bajo stuttering. Recíprocamente toda fórmula LTL que es invariable bajo stuttering puede expresarse con una fórmula $LTL_{\setminus X}$.*

5.3. Componentes terminales

La noción de componentes terminales es similar al concepto de componentes fuertemente conexas de los grafos pero aplicad a MDPs. Una componente terminal [16] es un subconjunto de estados y acciones del sistema compuesto dado de manera que estas acciones permiten ir de un estado a otro de este subconjunto con probabilidad , independientemente de la longitud del camino que se realice.

Dado un sistema $\mathcal{S} = (V, L, Act, enabled)$, definiremos formalmente a una componente terminal como una tupla (T, A) donde $T \subseteq S_V$, $A : T \to \mathcal{P}(Act)$, tal que:

- $A(s) \subseteq enabled(s)$.
- $\forall (a, c) \in A(s) \sum_{s' \in T} c(s, s') = 1$.
- $\forall s, s' \in T : \exists \sigma = s_0 \alpha_0 s_1 \ldots s_n : s_0 = s \wedge s_n = s' \wedge s_i \in T \wedge \alpha_i \in A(s_i)$.

5.4. Condiciones para la reducción

El sistema reducido se obtiene expandiendo sólo un subconjunto de las acciones habilitadas en cada uno de los estados. Este conjunto se denominada "*ample*" y con $ample(s)$ denotaremos el conjunto de acciones habilitadas en el estado s del modelo reducido. Por lo tanto $ample(s) \subseteq enabled(s)$ para todo estado s. Además requerimos que los conjuntos ample satisfagan un conjunto de condiciones que asegurarán que el sistema reducido preserve todas las propiedades de interés del sistema original. Dichas condiciones varían según

si el sistema es probabilista o no, y que tipo de schedulers se utilizan. Dado un sistema compuesto $\mathcal{S}$ generado por el conjunto de átomos $\mathcal{A}$ y φ una fórmula $LTL_{\setminus X}$, las condiciones que debe cumplir la función *ample* para generar el sistema reducido $\widehat{\mathcal{S}}$ son:

A1 Para todo estado $s \in S_V$, $\emptyset \neq ample(s) \subseteq enabled(s)$.

A2 Si $ample(s) \neq enabled(s)$ entonces todas las acciones $\alpha \in ample(s)$ son invisibles.

A3 Para todo camino $\sigma = s\alpha_1.s_1.\alpha_2 \ldots \alpha_n.s_n.\gamma$ en S, donde $s \in \widehat{\mathcal{S}}$ y α es dependiente de alguna acción de $ample(s)$ existe $1 \leq i \leq n$ tal que $\alpha_i \in ample(s)$.

A4 Si (T, A) es una componente terminal en $\widehat{\mathcal{S}}$ y $\alpha \in \bigcap_{t \in T} enabled_{\mathcal{S}}(t)$, entonces $\alpha \in \bigcup_{t \in T} ample(t)$

A5 Si $\sigma = s\alpha_1.s_1.\alpha_2 \ldots \alpha_n.s_n.\gamma$ es un camino en S, donde $\alpha_1, \alpha_2, \ldots \alpha_n, \gamma \notin ample(s)$ y γ es una acción probabilista entonces $|ample(s)| = 1$

Teorema 2. *Sea φ una fórmula $LTL_{\setminus X}$, $\mathcal{S}$ un sistema no-probabilista y sea $\widehat{\mathcal{S}}$ un sistema reducido que satisface las condiciones **A1**-**A4**. Luego $\mathcal{S} \vDash \varphi$ sii $\widehat{\mathcal{S}} \vDash \varphi$.*

Teorema 3. *Sea φ una fórmula $LTL_{\setminus X}$, $\mathcal{S}$ un sistema probabilista y sea $\widehat{\mathcal{S}}$ un sistema reducido que satisface las condiciones **A1**-**A5**. Luego $\sup_{\eta \in Sched(\mathcal{S})} Pr^{\eta}(\varphi) = \sup_{\eta \in Sched(\widehat{\mathcal{S}})} Pr^{\eta}(\varphi)$.*

Teorema 4. *Sea φ una fórmula $LTL_{\setminus X}$, $\mathcal{S}$ un sistema probabilista y sea $\widehat{\mathcal{S}}$ un sistema reducido que satisface las condiciones **A1**-**A4**. Luego $\sup_{\eta \in SDSched(\mathcal{S})} Pr^{\eta}(\varphi) = \sup_{\eta \in SDSched(\widehat{\mathcal{S}})} Pr^{\eta}(\varphi)$.*

Notar que en el caso no probabilista la condición **A4** es equivalente a pedir que si existe un ciclo en el sistema reducido y hay una acción que esta habilitada en el sistema original en todos los estados del ciclo entonces debe estar presente en alguno de los ample set del ciclo. De hecho esta última es la condición equivalente es la presentada en [26].

La condición **A1** garantiza que el sistema reducido está incluido en el sistema original. Además la condición de que el *ample* no sea vacío garantiza que siempre sea posible avanzar en el sistema reducido. Esto evita algunos casos patológicos como tener un sistema vacío.

Cuando el ample no esta expandido totalmente indica que el sistema reducido esta ignorando un conjunto de trazas. Si el ample contiene alguna acción

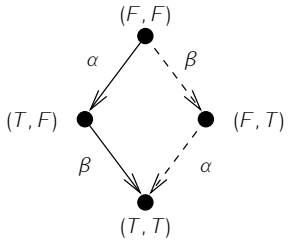


Figura 5.2: La importancia de las acciones invisibles.

visible puede afectar la validez de la fórmula que queremos verificar.Para ello se impone la condición A2.

En la figura 5.2 se puede ver un sistema compuesto cuyo estado esta formado por las variables booleanes p y q y cuyas acciones son α y β. Dichas acciones cambian el valor de $false$ a $true$ de las variables p y q respectivamente. Las acciones α y β son independientes entre si, ya que acceden y modifican a variables distintas. Un sistema reducido posible (si obviamos la condición A2) es el que primero ejecuta la acción α y luego la acción β. Sea ϕ la fórmula LTL $\mathbf{F}(p = true \wedge q = false)$. Debido a que p aparece sintácticamente en la fórmula ϕ, y α modifica a la variable p, α es una acción visible. Luego el sistema reducido no satisface la condición A2, ya que el ample inicial posee una acción visible y no es expandido totalmente. En el sistema reducido, la fórmula ϕ es siempre válida pues el estado $s_{t,f}$ siempre es alcanzado. Sea η el scheduler que consiste en ejecutar β en el estado inicial y α en el estado $s_{f,t}$. Bajo dicho scheduler ϕ es inválida. Es decir, el sistema reducido no posee un scheduler equivalente a η.

Recordemos que dos acciones son dependientes si una de las acciones escribe parte del estado que la otra acción accede. La ejecución de la acción que modifica alguna de las variables compartidas puede inhabilitar, habilitar o cambiar el comportamiento de la acción dependiente. Por ello, es importante no obviar los posibles interleavings que puedan ocurrir entre acciones dependientes. La acción A3 es la encargada de asegurar que dichos interleavings sean observados en el sistema reducido.

En la figura 5.3 se observa un sistema compuesto que posee dos variables: p que es una variable booleana y x que es una variable que toma valores en $\{0, 1, 2\}$. El sistema a su vez está integrado por tres acciones: α que cambia el valor de p de $false$ a $true$, β que cambia el valor de x de 0 a 1 y γ que cambia el valor de p de $false$ a $true$ y el valor de x de 1 a 2. Se puede que α es independiente de β, mientras que γ depende de α y β. Sea $\phi = \mathbf{GF}(x = 1)$ la fórmula LTL que se quiere verificar en el sistema. Notar que α es una acción invisible ya que

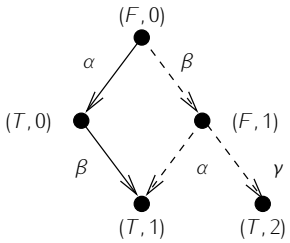


Figura 5.3: La importancia de **A3**.

no modifica a x, la única variable que aparece en ϕ. Luego un sistema reducido que cumple con todas las condiciones excepto **A3**, es el sistema cuyo ample inicial es α y los demás amples son expandidos totalmente. En éste subsistema ϕ siempre es válido ya que el estado $s_{t,1}$ siempre es alcanzado y una vez que llega a dicho estado nunca se vuelve a cambiar el valor de x. En el sistema original el scheduler que consiste en primero ejecutar β y luego γ, no satisface la propiedad ϕ. El camino dado por el scheduler anterior viola la condición **A3**. Pues, $\{\alpha\}$ es el ample inicial, α depende de γ y hay un camino en el sistema original que permite ejecutar a γ sin antes ejecutar α.

La noción de componentes terminales es similar al concepto de ciclos en un grafo, cuando construimos un ample set cada vez que dejamos fuera una acción al ser esta independiente con la que están incluidas sabemos que va a estar habilitada en el sistema original en los estados alcanzados utilizando acciones del ample, por lo que su inclusión puede ser "demorada" ahora si tenemos una componente terminal dicha demora es infinita ya que la acción nunca es incluida en ninguno de los amples de la componente terminal. Debido a esto podemos ignorar caminos relevantes si por ejemplo la acción no incluida habilita a una acción visible.

La condición **A5** (incluida originalmente en [15, 4]) es necesaria si se tienen en cuenta todos los schedulers de información total ya que un scheduler luego de una elección probabilista puede ejecutar diferentes acciones teniendo en cuenta el resultado de la elección probabilista. Recordemos el problema de la moneda que vimos en el ejemplo 1 donde había un scheduler de información total que primero arrojaba la moneda y luego decidía cual de las dos acciones del otro átomo ejecutaba de manera tal que adivina la tirada. Si bien las acciones de ambos átomos son independientes e invisibles si el sistema reducido primero ejecuta el átomo que intenta adivinar y luego arroja la moneda, este sistema no siempre adivina la tirada, por lo que no puede ser equivalente al anterior.

Capítulo 6

Heurísticas

En este capítulo analizaremos la complejidad de comprobar las condiciones A1-A5 y veremos que algunos de ellas son muy costosas. Para enfrentar este problema introduciremos algunas heurísticas que nos permitirán obtener buenos resultados con costos significativamente menores. Nuestras heurísticas asumimos que la representación del sistema es simbólica y hace uso de BDDs.

En este capítulo compararemos las heurísticas utilizadas en otros trabajos donde se realiza reducción de orden parcial tanto en sistemas probabilistas como no probabilistas y utilizando tanto técnicas de representación simbólicas como explícitas. Más precisamente, compararemos nuestras técnicas contra la tradicional sobre representación explícita [14, 26], la no-probabilista sobre representación simbólica [2] y la implementada en el model checker probabilista LiQuor [12] que usa representación explícita de MDPs.

6.1. Construcción del sistema reducido

En primer lugar notemos que las condiciones A1 y A2 son muy simples de chequear y se pueden verificar localmente a medida que se genera el sistema. De hecho, al generar el sistema reducido, es deseable definir la función ample sólo sobre los estados alcanzados por el sistema reducido. Sin embargo, notemos que las restricciones A3 y A5 necesitan conocer también partes del sistema original que no forman parte del sistema reducido. Esto no es problema real en nuestro caso dado que la construcción del sistema completo en model checking simbólico (i.e. usando BDDs y MTBDDs) es de muy bajo costo. La búsqueda de componentes terminales en modelos representados con MTBDDs es también una tarea muy costosa. Daremos también heurísticas para evitar tener que hacer una búsqueda específica de componentes terminales y mejorar así los costos a la hora de verificar la condición A4.

Para construir el ample de manera gradual podemos pensar al sistema como un grafo dirigido donde los nodos son estados y las aristas indican que hay una acción que va de un estado a otro con probabilidad no nula. Luego para construir el sistema reducido podemos utilizar algoritmos de recorrido de grafos. Los algoritmos más utilizados son DFS y BFS. Los algoritmos de model checking suelen utilizar ambos, pero la técnica de DFS es la más usada en los model checkers que utilizan estados explícitos, mientras que BFS es la más utilizada en los model checkers simbólicos. Una de las diferencias es que para implementar DFS de manera simbólica es necesario implementar un stack que no suele ser muy eficiente usando BDDs. Por otro lado las técnicas simbólicas suelen analizar mucho estados a la vez haciendo uso de sus similitudes para ahorrar espacio. Además se puede implementar BFS utilizando solamente conjuntos, que son sencillos de construir si se usan BDDs. En conclusión debido a que nuestra implementación es usando estados simbólicos, nuestro algoritmo utiliza BFS.

6.2. Construcción del *ample*

Notar que las acciones que pertenecen a un mismo átomo son dependientes. Esto implica que siempre que una acción esta en el ample, todas las demás acciones de este átomo que están habilitadas deben estar también en el ample caso contrario se violaría la condición A3. Esto nos permite pensar a la función ample como una función que mapea los estados alcanzables en el modelo reducido en conjuntos de átomos (i.e. $ample : S_V \mapsto \mathcal{P}(\mathcal{A})$). En el resto de este capítulo denotaremos con $ample(s)$ tanto a los átomos que componen el ample como a las acciones. Solo las dififerenciaremos cuando sea ambiguo a cual de los dos conceptos nos estamos refiriendo.

Existen diversas estrategias para construir conjuntos ample, las que podemos separar en dos categorías. La primer categoría corresponde a las técnicas *greedy*, donde se elige un ample inicial y mientras se viole alguna condición se eligen nuevas acciones. Notar que esta técnica converge pues eventualmente el ample se expande totalmente y en tal caso $ample(s) = enabled(s)$. Este tipo de estrategias son las utilizadas en los trabajos de Peled y Alur. Si bien esta técnica tiene una complejidad lineal en la cantidad de átomos, el éxito de las reducciones está determinado por el orden en que se eligen los átomos. Por ejemplo si el átomo que se elige primero tiene la mayoría de sus acciones visibles el sistema reducido va a ser muy similar en tamaño al original. La otra categoría corresponde a las técnicas que intentan generar el ample más pequeños posible, es decir comprueban todos los amples posibles y escogen el que menos acciones posee. Este tipo de técnica es utilizada en LiQuor. Dado que LiQuor implementa la variable de A5 presentada en [15] (los ample sets tienen una sola acción ó

tienen todas las habilitadas) la complejidad sigue siendo lineal respecto de la cantidad de átomos. La técnica que nosotros implementamos se enmarca en esta segunda categoría. Sin embargo, dado que nosotros implementamos la restricción A5, la cantidad de candidatos a explorar para encontrar el ample es exponencial respecto del número de átomos. Para paliar este problema brindaremos la opción de acotar el tamaño del conjunto ample que no se expande completamente. De esta manera una cota de 1 resultaría en un comportamiento similar a LiQuor.

6.3. Alcanzabilidad local y A3

La verificación de la condición A3 requiere el conocimiento del sistema original y, en el peor de los casos, la totalidad de este. El siguiente lema aparece en [14]:

Lema 1. *Comprobar A3 es equivalente a hacer alcanzabilidad en el sistema original.*

Dado que realizar alcanzabilidad es lineal en la cantidad de estados y para cada estado hay que verificar A3, la complejidad de construir el ample es al menos cuadrática en la cantidad de estados del sistema original. Dada la naturaleza exponencial del tamaño del espacio de estados, y teniendo en cuenta que la reducción de orden parcial es un procesamiento previo al proceso de verificación, los costos de verificación de la condición A3 son muy altos. Por consiguiente es necesario utilizar heurísticas que permitan garantizar que esta condición se cumpla. En general la idea de estas heurísticas es enfocarse en un átomo y suponer que los demás átomos pueden cambiar su estado de manera arbitraria y comprobar que en esos estados resultantes no se habiliten acciones dependientes.

En lo que resta de esta sección analizaremos primero heurísticas conocidas y finalmente introduciremos nuestra propuesta.

6.3.1. Heurística de Alur et al

[2] introduce reducción de orden parcial para model checkers simbólicos no probabilistas. El modelo utilizado allí supone que no existe sincronización de acciones entre átomos y que cada átomo puede modificar una porción del estado disjunto de la que puede modificar cualquier otro átomo. Sin embargo los átomos pueden leer cualquier parte del estado global. De esta manera, la habilitación de una acción depende del estado global de manera similar a nuestros modelos.

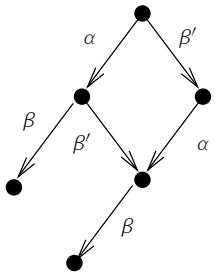


Figura 6.1: Caso espurio de la heurística de Alur et al.

Dado un átomo A se define una función $localEnabled_A : S_V \mapsto Act_A$ que satisface:

$$\alpha \in localEnabled_A(s) \Leftrightarrow \exists s' \in S_{V \setminus V_A} : \alpha \in enabled_{\mathcal{A}}(s|_{V_A} \oplus s') \qquad (6.1)$$

Podemos decir que una acción α está habilitada en un estado local $s|_{V_A}$ del átomo A si existe un estado global $\widehat{s}$ que extiende a $s|_{V_A}$ de manera que α está habilitado en $\widehat{s}$. Por extensión, diremos que α está habilitada localmente en el estado global s si α esta habilitado localmente en el estado local $s|_{V_A}$. Esto es lo que refleja la definición de $localEnabled$ en la ecuación 6.1. Claramente $enabled(s) \subseteq localEnabled_A(s)$. Más aún, si estando en el estado s se ejecuta una acción que no es del átomo A y que lleva al estado s', tenemos que:

$$enabled_A(s') \subseteq localEnabled_A(s) \qquad (6.2)$$

dado que las variables de A no fueron modificadas, es decir $s|_{V_A} = s'|_{V_A}$. Esta noción se puede extender a ejecuciones.

La heurística propuesta por Alur para comprobar **A3** en un estado s es verificar:

$$\forall A \in ample(s), B \notin ample(s) : dep(localEnabled_A(s)) \cap Act_B = \emptyset \qquad (6.3)$$

Esta heurística es válida por la propiedad 6.2. Notar que esta heurística no tiene en cuenta la semántica del sistema, sólo realiza un análisis sintáctico. Esta heurística es una sobre-aproximación de **A3** y rechaza amples que satisfacen esta condición como veremos en el siguiente ejemplo.

Consideremos el sistema de la figura 6.1 donde α es una acción del átomo A y β y β' son acciones del átomo B. Notar que β se habilita luego de ejecutar α y por consiguiente α y β son dependientes. Notar que $\{A\}$ (o equivalentemente $\{\alpha\}$) es un ample válido: **A3** es satisfecha porque antes de β se ejecuta una acción del ample. Sin embargo, la heurística rechaza a $\{A\}$ pues

$\alpha \in enabled_A(s) \subseteq localEnabled_A(s)$ pero $\beta \in B \notin \{A\}$ es dependiente de α. Una situación semejante a esta, ocurre sobre nuestro ejemplo de la moneda (Ej. 1) donde la heurística no permite ninguna reducción.

6.3.2. Heurística de Peled [14]

Esta heurística sigue las ideas de la anterior pero es un poco más débil, acercándose más a las condiciones impuestas por **A3**. Para la comprobación se define una relación $pre \subseteq Act_{\mathcal{A}} \times Act_{\mathcal{A}}$ tal que $\beta \in pre(\alpha)$ si β modifica alguna de las variables que α lee para determinar su habilitación. Luego la heurística consiste en comprobar lo siguiente:

$$\forall \alpha \in ample(s), B \notin ample(s) : dep(\alpha) \cap Act_B = \emptyset \tag{6.4}$$

$$\forall A \in ample(s), B \notin ample(s) : pre(localEnabled_A(s) \setminus ample(s)) \cap Act_B = \emptyset \tag{6.5}$$

6.4 debilita la condición 6.3 considerando solamente as acciones del conjunto ample. Sin embargo 6.4 debilita demasiado 6.3 dado que no considera la posibilidad de futuras habilitaciones de acciones dependientes de acciones del ample por parte de eventos fuera del ample. Precisamente, la condición 6.5 considera este caso.

6.3.3. Heurística en LiQuor

Podemos decir que la heurística anterior comienza a considerar aspectos semánticos pero sólo lo hace a nivel de un paso de ejecución (codificado en la relación *pre*). A cambio, LiQuor considera las posibles ejecuciones locales de cada átomo para lograr mayor precisión en la verificación de **A3**. Sin embargo esta nueva heurística hace uso de **A5** como hipótesis. Como consecuencia la heurística asume que el conjunto ample tiene a lo sumo una acción (sino estaría completamente expandido y no es necesario comprobar **A3**).

Dado un átomo A se define la relación $\rightarrow_A \subseteq S_{V_A} \times S_{V_A}$. De manera que $s \rightarrow_A t$ si existen $\alpha \in Act_A, s', t' \in S_{V \setminus V_A}$ tal que $\alpha(s \oplus s', t \oplus t') > 0$. Es decir se define cuando se puede realizar una transición de manera local. Definimos también la relación $\rightarrow_A^* \subseteq S_{V_A} \times S_{V_A}$ que satisface $s \rightarrow_A^* t$ si $s = t$ o existen $s_0, \ldots, s_n \in S_{V_A}$ tal que $s = s_0 \rightarrow_A s_1 \rightarrow_A \ldots \rightarrow_A s_n = t$. Notar que si el sistema compuesto hay un camino de s a t luego $s|_{S_{V_A}} \rightarrow_A^* t|_{S_{V_A}}$ para cualquier átomo $A \in \mathcal{A}$

Seguidamente definiremos alcanzabilidad de la manera usual.

Definición 18. *Sea $\mathcal{A}$ un sistema compuesto luego $reach \subseteq S_V \times S_V$ es una relación tal que $t \in reach(s)$ si existe un camino $\sigma \in Path_{\mathcal{A}}$, tal que $first(\sigma) = s$ y $last(\sigma) = t$.*

De manera similar, definiremos la noción de alcanzabilidad local.

Definición 19. *Sea $\mathcal{A}$ un sistema compuesto y $A \in \mathcal{A}$ luego $localReach_A \subseteq S_{V_A} \times S_{V_A}$ es la relación tal que $t \in localReach_A(s)$ si $s \rightarrow_A^* t$.*

Es fácil ver que $reach(s)|_{V_A} \subseteq localReach_A(s|_{V_A})$ Si A es el único átomo que pertenece al $ample(s)$, la heurística comprueba que:

$$\forall B \in \mathcal{A} \setminus \{A\} : localEnabled_B(localReach_B(s|_{V_B})) \cap dep(ample(s)) = \emptyset \tag{6.6}$$

$$\forall B \in \mathcal{A} \setminus \{A\} : Write(localEnabled_B(localReach_B(s|_{V_B}))) \cap Read(Act_A \setminus ample(s)) = \emptyset \tag{6.7}$$

La primera condición garantiza que no se van a ejecutar acciones dependientes de átomos que no pertenecen al ample, mientras que la segunda garantiza que no se van a habilitar acciones que pertenecen al átomo del ample, pero que no están habilitadas.

6.3.4. Nuestra heurística

Nuestra heurística es similar a la anterior, pero tiene las siguientes propiedades:

- Permite amples de más de un átomo.
- Define la idea de alcanzabilidad local teniendo en cuenta la interrelación de las ejecuciones de los átomos fuera del ample.
- Es más eficiente que realizar alcanzabilidad

Lo que hace la heurística es analizar los potenciales estados alcanzados por un átomo si no se ejecutan acciones que pertenezcan a los átomos del ample. Para ello vamos a redefinir la noción de alcanzabilidad local vista en 6.3.3

Sea $A \in \mathcal{A}$ un átomo y $\mathcal{B} \subseteq \mathcal{A}$ un conjunto de átomos (que podemos pensar como el conjunto ample). Definimos al conjunto $W_A^{\mathcal{B}}$:

$$W_A^{\mathcal{B}} = \bigcup_{\substack{B \neq A \\ B \in \mathcal{A} \setminus \mathcal{B}}} \bigcup_{\substack{\alpha \in Act_B \\ label(\alpha) \notin L_{\mathcal{B}}}} Write_B(\alpha)$$

$W_A^{\mathcal{B}}$ contiene todas las variables que pueden ser modificadas por los átomos que no son A ni pertenecen a $\mathcal{B}$. Notar que las modificaciones pueden provenir de acciones que sincronizan con A pero no con algún átomo de B. Definimos $\rightarrow_{A,\mathcal{B}} \subseteq S_V \times S_V$ como la transición que ejecuta acciones de A suponiendo un comportamiento arbitrario por parte de los átomos que no están en B (ni son A). $s \rightarrow_{A,\mathcal{B}} t$ si $s|_{V \setminus W_A^{\mathcal{B}}} = t|_{V \setminus W_A^{\mathcal{B}}}$ o si existen $s', t' \in S_{W_A^{\mathcal{B}}}$ y $\alpha \in Act_A$ tal que:

$$\alpha(s|_{V_A \setminus W_A^{\mathcal{B}}} \oplus s'|_{W_A^{\mathcal{B}} \cap V_A}, t|_{V_A \setminus W_A^{\mathcal{B}}} \oplus t'|_{W_A^{\mathcal{B}} \cap V_A}) > 0$$

Definiremos como es usual $\rightarrow^*_{A,\mathcal{B}} \subseteq S_V \times S_V$ como la clausura reflexo transitiva de $\rightarrow_{A,\mathcal{B}}$. Nuestra noción refinada de alcanzabilidad local $localReach_{A,\mathcal{B}} \subseteq S_V \times S_V$ se define por:

$$t \in localReach_{A,\mathcal{B}}(s) \Leftrightarrow s \rightarrow^*_{A,\mathcal{B}} t$$

Notar que si $s = s_0.\alpha_0.s_1. \ldots s_{n-1}.\alpha_{n-1}.s_n = t$ es un camino válido en $\mathcal{A}$ tal que $label(\alpha_i) \notin L_{ample(s)}$ luego $t \in LocalReach_{A,ample(s)}$ para todo $A \in \mathcal{A}$. Pues, si $label(\alpha_i) \in L_A$ entonces $s_i \rightarrow_{A,ample(s)} s_{i+1}$ y si $label(\alpha_i) \notin L_A$, $s_i|_{V \setminus W_A^{\mathcal{B}}} = s_{i+1}|_{V \setminus W_A^{\mathcal{B}}}$ y por consiguiente $s_i \rightarrow_{A,ample(s)} s_{i+1}$.

Nuestra heurística consiste entonces en verificar la siguiente propiedad:

$$enabled_{\mathcal{A}}\left[\bigcap_{A \in \mathcal{A}} localReach_{A,ample(s)}(s) \cap reach(s_{init})\right] \cap dep(ample(s)) = ample(s) \tag{6.8}$$

Veamos ahora que nuestra heurística es válida. Supongamos que nuestro sistema reducido no satisface A3, por lo que existe un camino de la forma $s_0.\alpha_0.s_1 \ldots s_{n-1}.\alpha_{n-1}.s_n\gamma.s_{n+1}$ donde $\alpha_i \notin ample(s_0), dep(ample(s_0))$ y $\gamma \in dep(ample(s_0))$. Luego $s_n \in localReach_{A,ample(s_0)}(s_0)$ para todo $A \in \mathcal{A}$, además $s_n \in reach(s_{init})$ por lo tanto $\gamma \in enabled_{\mathcal{A}}(\bigcap_A \in \mathcal{A} localReach_{A,ample(s_0)}(s_0) \cap reach(s_0)) \cap dep(ample(s_0))$ contradiciendo 6.8.

6.4. Detección de componentes terminales

A4 requiere encontrar todas las componentes terminales del sistema reducido y verificar, para cada uno de ellos, que las acciones que están continuamente habilitadas puedan efectivamente ejecutarse en el sistema reducido. Esto implica que uno debería construir primero un sistema reducido tentativo y expandirlo en caso de que A4 falle. Obviamente, un proceso como este es altamente ineficiente.

En el resto de la sección nos abocaremos a definir una heurística eficiente para asegurar la satisfacción de A4.

El siguiente lema, que es la base de nuestra heurística, es una adaptación de un resultado de Peled en el contexto no probabilista [26].

Lema 2. *Sea $\mathcal{A}$ un sistema compuesto y sea $\tilde{\mathcal{A}}$ un sistema reducido de $\mathcal{A}$. Luego si para toda componente terminal (T, A) de $\mathcal{A}$ existe un estado $s \in T$ tal que $ample(s) = enabled_{\mathcal{A}}(s)$, entonces $\tilde{\mathcal{A}}$ satisface **A4**.*

A continuación analizaremos dos heurísticas para detectar componentes terminales una basada en DFS y la otra en BFS.

6.4.1. DFS

El siguiente lema provee una condición necesaria para identificar la presencia de una componente terminal.

Lema 3. *Dado un sistema $\mathcal{A}$ y una componente terminal (T, A) en $\mathcal{A}$. entonces existe un ciclo incluido en los estados de T.*

Usando este lema podemos usar la siguiente variante más fuerte de **A4**:

$$\begin{aligned} &\text{si } C \text{ es un ciclo en } \widehat{S} \text{ y } \alpha \in \cap_{t \in C} enabled_S(t) \text{ luego} \\ &\text{existe un estado } s \in C \text{ tal que } ample(s) = enabled_{\widehat{S}}(s). \end{aligned} \tag{6.9}$$

Esta condición es la definida en [15] y la que está implementada en LiQuor. La detección de ciclos por medio de DFS es muy sencilla: basta con comprobar que el próximo estado a visitar se encuentra en la pila de búsqueda (la cual alberga el camino recorrido). Para verificar la condición 6.9 se asocia un flag a cada estado indicando si el ample en ese estado se expandió completamente. Al detectar la presencia se un ciclo, si ningún flag en los estados del ciclo está activado, se expande completamente el ample en el último estado visitado.

6.4.2. BFS

En [2] se presenta una estrategia basada en BFS para verificar 6.9. Los autores muestran que las estrategias DFS y BFS para verificar 6.9 no son comparables. Es decir, hay ejemplos en los cuales uno obtiene mejores reducciones que la otra y viceversa. Sin embargo, hay resultados empíricos que sitúan BFS como una mejor estrategia. En primer lugar BFS se implementa de manera natural en model checking simbólico mientras que DFS es altamente ineficiente. En segundo lugar, BFS tiene la ventaja que no es sensible al orden del recorrido del grafo mientras que DFS puede obtener reducciones muy distintas para dos recorridos diferentes. En nuestro caso se suma una ventaja muy importante a

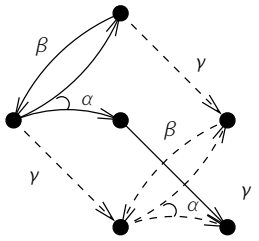


Figura 6.2: La condición 6.9 no permitiría la reducción del sistema mientras que que nuestra heurística eliminaría las transiciones indicadas con las líneas de punto.

favor de BFS: no necesitamos depender del lema 3 y podemos enfocarnos en la búsqueda efectiva de componentes terminales. De esta manera reduciríamos significativamente la cantidad de expresiones espurias (ver figura 6.4.2).

La heurística parte de la siguiente observación:

> Una condición necesaria para que (T, A) sea una componente terminal requiere que para todo $s \in T$ y $\alpha \in A(s) \subseteq ample(s)$, $\alpha(s, t)$ implica que $t \in T$.

El punto clave de esta heurística radica, entonces, en verificar si una nueva acción candidata a incluir en un ample visita estados ya visitados con probabilidad 1. De ser así, es posible estar ante la presencia de una componente terminal y el ample debería expandirse completamente.

Más precisamente, el algoritmo considera al espacio de estados dividido en tres conjuntos: el conjunto S_1 de estados visitados cuyo ample definido, el conjunto S_2 de los estados visitados cuyos ample no fueron definidos (i.e. los que se analizaran en la próxima iteración) y el resto. Notar que $S_1 \cup S_2 \subseteq \widehat{S}$, es decir forman pare del modelo reducido.

Dado que los amples asociados a los estados S_1 ya han sido fijados podemos suponer que cualquier componente terminal incluido en S_1 satisface con la condición **A4**.

Consideremos ahora una componente terminal potencial en $S_1 \cup S_2$ tal que contenga un estado $s_2 \in S_2$. Al analizar el conjunto candidato a ample de s pueden ocurrir dos casos: (*i*) que no haya ninguna acción en tal conjunto que llega a $S_1 \cup S_2$ con probabilidad 1, en cuyo caso no se formaría una componente terminal, o (*ii*) que alguna de las acciones llegue a $S_1 \cup S_2$ con probabilidad 1, en cuyo caso se asume la presencia de una componente terminal y se rechaza tal conjunto como ample.

6.5. Heurísticas para comprobar A5

Para comprobar A5 vamos a presentar dos heurísticas, la primera de ellas es pedir que $|ample(s)| = 1$ si $ample(s) \neq enabled_{\mathcal{A}}(s)$. Esta condición es la definida en [15] y la implementada en LiQuor. La segunda heurística consiste en hacer uso de alcanzabilidad local en el sentido de 6.8. La condición que hay que pedir es que si una acción probabilista que no es del ample está habilitada en alguno de los posibles estados alcanzados localmente entonces el ample es el total o cuenta con una solo acción.

Capítulo 7

Diagramas de decisión binarios

Los diagramas de decisión binarios o simplemente BDD, son un tipo abstracto de datos utilizado para representar de manera compacta funciones booleanas. Los BDDs fueron introducidos por Akers [1], pero su uso fue popularizado por el trabajo de Bryant [9]. La utilización de BDDs en model checking fue introducida a principios de los 90's en los trabajos de Burch et al [10] y de McMillan [24]. Existen varios model checkers simbólicos que utilizan BDDs como ser NuSMV [13], PRISM [20], RAPTURE [22], Rabbit [8].

Este capítulo está basado en [21].

7.1. Funciones booleanas

Sea $V = \{x_1, \ldots, x_n\}$ un conjunto de variables booleanas. Una función booleana f de n argumentos es una función que va de $\mathbb{B}^n$ en $\mathbb{B}$. Escribiremos $f(x_1, \ldots, x_n)$ o $f(V)$ para indicar que la función f depende de las variables en V. Por otro lado denotaremos por $f(b_1, \ldots, b_n)$ a la evaluación de f donde la variable x_i es evaluada en $b_i \in \mathbb{B}$. Algunas funciones básicas son:

- $0 \doteq 0$
- $1 \doteq 1$
- $\overline{x} \doteq 1 - x$
- $x.y \doteq \begin{cases} 1 & \text{si } x, y = 1 \\ 0 & \text{c.c.} \end{cases}$
- $x + y \doteq \begin{cases} 0 & \text{si } x, y = 0 \\ 1 & \text{c.c.} \end{cases}$

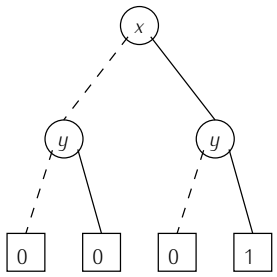


Figura 7.1: Árbol de decisión binario para la función $\overline{(x + y)}$

- $x \oplus y \stackrel{.}{=} \begin{cases} 1 & \text{si } x \neq y \\ 0 & \text{c.c.} \end{cases}$

Notar que cualquier función booleana puede ser expresada en términos de operadores definidos previamente.

7.2. Árboles de decisión binarios

Los árboles de decisión binarios son árboles binarios cuyos nodos no terminales están etiquetados con variables booleanas, mientras que los nodos terminales están etiquetados con 0 ó 1. Cada nodo no terminal posee dos vértices: *then* y *else*. La forma de representar una función booleana es la siguiente: si se tiene una valuación de las variables se recorre el árbol empezando por la raíz y utilizando el vértice *then* si la valuación de la variable del nodo actual es 1, caso contrario se sigue el vértice *else*. El nodo terminal alcanzado es la evaluación de la función que representa el árbol. En la figura 7.2 se puede observar una representación de la función $\overline{(x + y)}$. Las líneas punteadas denotan los vértices *else* y las lineas firmes los vértices *then*.

7.3. Diagramas de decisión binarios

Los diagramas de decisión binarios son estructuras de datos similares a los árboles de decisión binarios, pero permiten eliminar la redundancia que poseen estos. Formalmente un diagrama de decisión binario o BDD es un grafo dirigido acíclico con una única raíz, tal que los nodos no terminales están etiquetados con variables y los nodos terminales con 0 ó 1. Cada nodo no terminal posee dos vértices *then* y *else*. La forma de evaluar una función es exactamente la misma

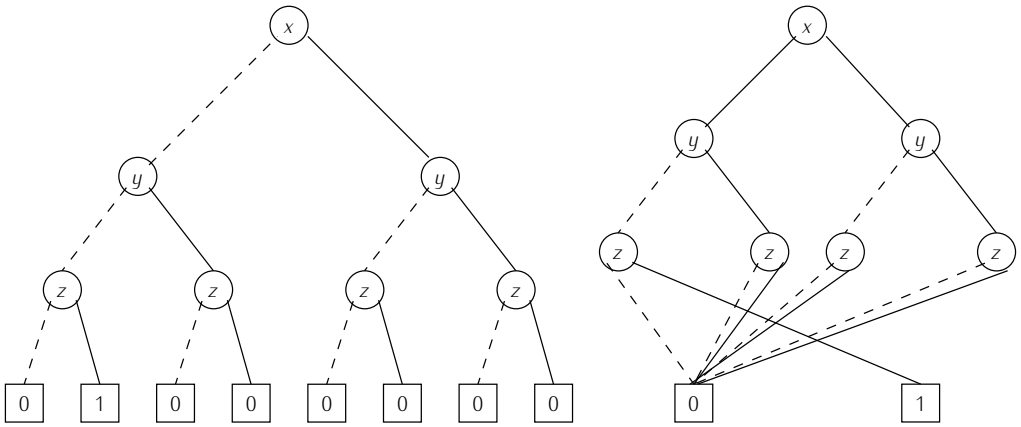


Figura 7.2: Reducción de la función $\overline{(x+y)}.z$

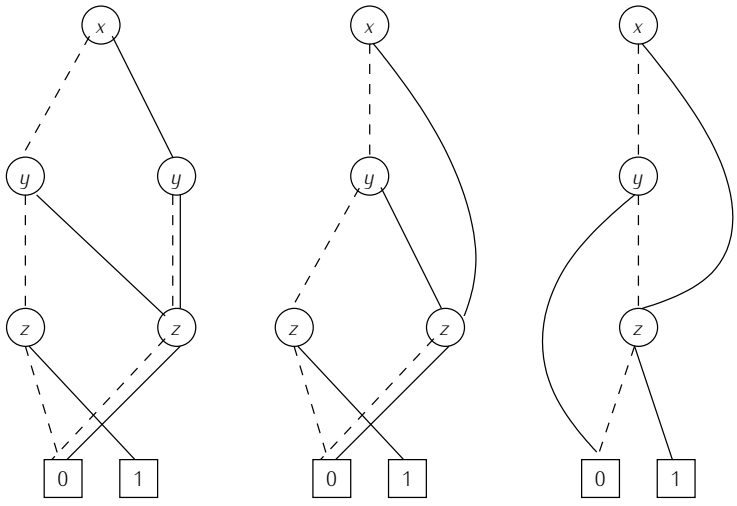
que con los árboles de decisión binarios: se recorre el grafo según la evaluación que tienen las variables. Dado un BDD existen varias formas de generar un nuevo BDD equivalente, i.e. que represente la misma función, pero que elimina parte de la redundancia que posee el BDD original. Las diferentes formas de reducir un BDD son:

C1. Eliminar terminales duplicados: Si un BDD tiene más de un nodo etiquetado con 0, se deja uno y se redirigen todos los vértices que apuntan a un nodo etiquetado con 0 a este nodo. Un procedimiento similar se realiza con los nodos cuya etiqueta es 1.

C2. Eliminar valuaciones redundantes: Si ambos vértices de un mismo n apuntan hacia el mismo nodo m, se elimina el nodo n y todos los vértices que estaban dirigidos a n son redirigidos a m.

C3. Eliminar no terminales duplicados: Si dos nodos distintos n y m apuntan a dos subBDDs distintos, pero que poseen a misma estructura (i.e. representan la misma función), se elimina uno de los subBDDs, redirigiendo los nodos al otro subBDD.

En la figura 7.3 se puede observar como se puede reducir un BDD que representa la función $\overline{(x+y)}.z$. Diremos que un BDD es reducido si no se puede aplicar ninguna de las reducciones **C1**-**C3**.



(c) (d) (e)

Figura 7.2: Reducción de la función $\overline{(x+y)}.z$ *(cont.)*

7.4. BDDs Ordenados

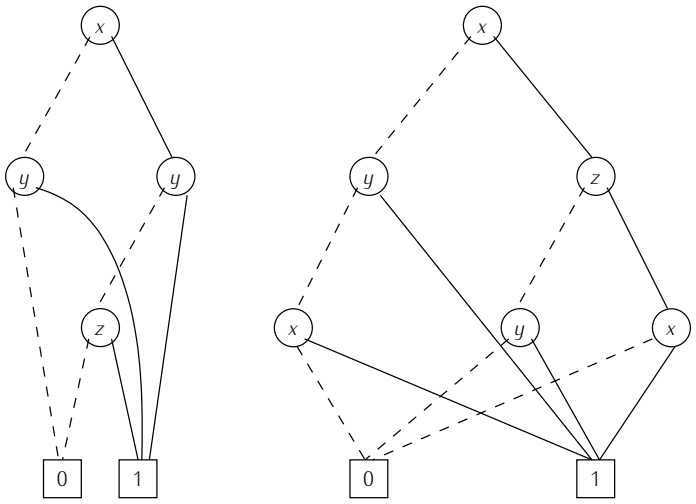
Notar que que en la definición anterior de BDDs se permite que dado un camino en el grafo se repitan las etiquetas en dos nodos diferentes. Esto provoca que existan más de un grafo (en particular existen infinitos) que represente una misma función. En la figura 7.4 se observan dos BDDs diferentes que representan a la función $x.z + y$.

Dado $<$ un orden lineal estricto sobre V, un diagrama de decisión binario reducido y ordenado o simplemente ROBDD es un BDD reducido tal que todo camino entre nodos no terminales diferentes n y m donde x es la etiqueta de n e y es la etiqueta de m se da que $x < y$. Los ROBDD cumplen con una propiedad muy útil:

Teorema 5. *Dada f, una función booleana, y $<$, un orden lineal total sobre V, el ROBDD que representa a f y utiliza el orden $<$ es único.*

Este resultado nos permite determinar satisfactibilidad en tiempo constante: solo hace falta ver que la raíz del ROBDD no tenga como etiqueta a 0. Por otro lado al ser única la representación, se puede fácilmente determinar si dos ROBDD representan la misma función: los grafos tienen que ser iguales.

El tamaño que posee la representación de la función, i.e. la cantidad de nodos que posee el ROBDD, esta sujeto al orden establecido. Dicho tamaño en



(a) (b)

Figura 7.3: Dos BDDs que representan $x.z + y$

el peor de los casos puede ser exponencial. Sin embargo la búsqueda del orden óptimo es un problema NP-completo, por lo cual el ordenamiento se basa en heurísticas.

7.5. Operaciones entre BDDs

Existen varias operaciones que se pueden realizar entre BDD algunas de ellas son:

- $ITE(M_1, M_2, M_3)$ es el BDD que representa a la función $f_{M_1} f_{M_2} + \overline{f_{M_1}} f_{M_3}$.
- $NOT(M)$ es el BDD que representa a la función $\neg f_M$.
- $AND(M_1, M_2)$ es el BDD que representa a la función $f_{M_1} \wedge f_{M_2}$.
- $OR(M_1, M_2)$ es el BDD que representa a la función $f_{M_1} \vee f_{M_2}$.
- $FORALL(V, M)$ donde $V = \{v_1, \ldots, v_n\}$ es un conjunto de variables, representa a la función $\forall v_1, \ldots, v_n : f_M$.
- $EXISTS(V, M)$ donde $V = \{v_1, \ldots, v_n\}$ es un conjunto de variables, representa a la función $\exists v_1, \ldots, v_n : f_M$.

- $PERMUTE(V_1, V_2, M)$ donde $|V_1| = |V_2|$, es el resultado de permutar simultáneamente todas las ocurrencias de $v_{i,1}$ por $v_{2,i}$ en la función f_M.
- $EQUALVARS(V_1, V_2)$ donde $|V_1| = |V_2|$, representa a la función $\bigwedge(v_{1,i} = v_{2,i})$.

7.6. MTBDD

Vimos que los BDD representan funciones de la forma $f : \mathbb{B}^n :\to \mathbb{B}$, sin embargo también es útil poder representar funciones que no son predicados sino funciones de la forma $f : \mathbb{B}^n :\to D$ con D un conjunto arbitrario por ejemplo $\mathbb{R}$. Los MTBDD (*Multi-Terminal Binary Decision Diagrams*) cumplen con las mismas condiciones que los BDDs, excepto que los nodos terminales están etiquetados con elementos de D en vez de $\mathbb{B}$. En PRISM los MTBDDs se utilizan principalmente para representar la función de transición de estados. Al igual que los BDD se pueden realizar ciertas operaciones entre MTBDDs como sumar, multiplicar, etc

7.7. Expresividad de los BDD

A continuación veremos algunas utilidades de los BDDs y los MTBDDs, que son usadas en *PRISM* y en la implementación de *POR*.

7.7.1. Representación de variables no booleanas

Si bien las funciones solo pueden tener como argumentos variables booleanas, es posible representar variables de cualquier tipo de dominio, siempre que sea finito. Supongamos que queremos tener una variable v cuyo dominio es $D_v = \{d_1, \ldots, d_n\}$, luego podemos utilizar $\ulcorner log_2(n) \urcorner$ variables booleanas $\{v_1, \ldots v_{\ulcorner log_2(n) \urcorner}\}$ donde la evaluación de las variables v para a representar al elemento d_i es $(b_1, \ldots, b_{\ulcorner log_2(n) \urcorner})$, donde b_j es 1 si en la representación binaria de i el bit j es 1 y 0 en caso contrario. De esta forma se puede expresar la proposición $v = d_i$ como:

$$\bigwedge_j v'_j$$

Donde v'_j es igual a v_j si la representación binaria de i el j-ésimo bit es 1 y v'_j es igual a $\neg v_j$ en caso contrario. Debido a que los estados son funciones que van

de variables a su evaluación, podemos representar a un estado como un conjunto de variables. Es decir codificando a cada variable como vimos anteriormente. Sean $v_1, \ldots v_n$ las variables que conforman los estados, las cuales poseen un representación en los BDDs. Con el vector $\overrightarrow{s}$ es el utilizado para representar a todas las variables anteriores. El predicado $\overrightarrow{s} = s$ donde s es un estado dado se expresa de la siguiente manera:

$$\bigwedge_{i=1}^{n}(v = s(v))$$

7.7.2. Conjuntos

Supongamos que poseemos un universo finito $\mathcal{U} = \{a_1, \ldots, a_n\}$. Si queremos representar un subconjunto A de dicho universo, una manera posible es utilizar n variables $\{v_1, \ldots v_n\}$. El conjunto A puede representarse con una valuación en las variables de tal forma que $v_i = 1$ si $a_i \in A$ y $v_i = 0$ si $a_i \notin A$. De esta manera podemos representar a la fórmula $\overline{v} = A$ como:

$$\bigwedge_{a_i \in A} v_i \wedge \bigwedge_{a_i \notin A} \neg v_i$$

A su vez podemos representar a la función característica de la siguiente forma:

$$\chi_A(v_1, \ldots, v_n) = \bigwedge_{a_i \notin A} \neg v_i$$

7.7.3. Conjunto de estados

Si $\overrightarrow{s}$ es el conjunto de variables que son utilizadas para representar los posibles estados. Se puede representar al conjunto de estados $\{s_1, \ldots, s_n\}$ como:

$$\bigvee_{i=1}^{n}(\overrightarrow{s} = s_i)$$

donde el predicado $\overrightarrow{s} = s_i$ es el que vimos en 7.7.1.

Capítulo 8

PRISM

PRISM [25], es un model checker simbólico de sistemas probabilísticos de código abierto, desarrollado inicialmente en la Universidad de Birmingham y actualmente en la Universidad de Oxford. Esta herramienta permite comprobar propiedades PCTL (*Probabilistic Computation Tree Logic*), para cadenas de Markov de tiempo discreto, así como también para procesos de decisión de Markov, propiedades CSL (Continuos Stochastic Logic) para cadenas de Markov de tiempo continuo.

Bederián[7], extendió PRISM para poder verificar propiedades LTL en procesos de decisión de Markov que es el contexto de interés en nuestro trabajo.

Cabe aclarar que la manipulación de MTBDD, en PRISM se realiza con la biblioteca Cuud[28].

8.1. Arquitectura de PRISM

PRISM toma como entrada un modelo del sistema dado en un lenguaje de módulos específico de la herramienta y una propiedad especificada en alguna de la lógicas. Ambas especificaciones son parseadas separadamente. Es sistema es compilad en el modelo subyacente apropiado (DTMC, CTMC, o MDP) y las propiedades se guardan en una estructura de datos ad-hoc. La verificación procede haciendo primero pre-procesamientos específicos y luego realizando los cálculos numéricos apropiados. Este cálculo numérico puede realizarse con tres motores de cálculo distintos: basado en MTBDD, basado en matrices ralas, o con un motor híbrido donde la matriz se almacena como MTBDD y el vector de estados se almacena explícitamente. La figura 8.1 describe la arquitectura de PRISM.

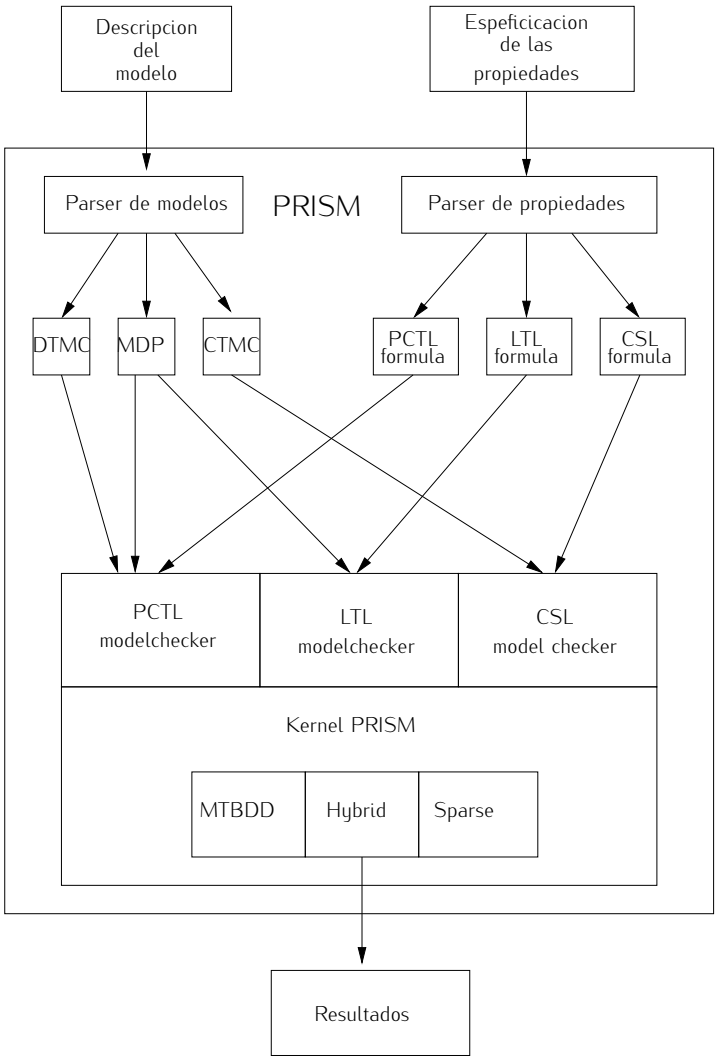


Figura 8.1: Arquitectura de PRISM

8.2. Lenguaje de PRISM

PRISM permite especificar diferentes tipos de modelos como ser MDP, CTMC y DTMC. Solo veremos los modelos MDP, que son el tipo de modelos que estamos interesados en analizar.

Un modelo en PRISM está dado por un conjunto de módulos $M_1, \ldots, M_n$ donde cada módulo esta compuesto por un par (V_i, C_i), donde V_i es el conjunto de variables del módulo y C_i es un conjunto de comandos. Las variables del sistema están dadas por $Var = V_{global} \cup \bigcup V_i$, donde V_{global} son las variables globales que posee el sistema. Las variables pueden ser de tipo booleano y entero aco-

```
global v: [0..2] init 0

module M1
b1: bool init false
v1: [0..2] init 0
[] p = q -> (v' = 1);
[a] v1 < 2 -> 0.5 : (v1'=v1+1) + 0.5: (b1' = !b1);
endmodule

module M2
b2: bool init false
v2: [0..2] init 0
[] v < 2 -> 0.5 : (v'=v+1) & (v2'=v1) + 0.5 (b2'=true);
[a] true -> (b2' = !b2);
endmodule
```

Listing 8.1: Ejemplo de modelo PRISM

tado. Los comandos pueden verse como estructuras $(a, g, (\lambda_1, u_1), \ldots, (\lambda_m, u_m))$ donde:

- a es una etiqueta de sincronización. Existe una etiqueta especial que denotaremos con ε, que indica que el comando es interno al módulo y por lo tanto no admite sincronización.
- g es una guarda que indica en que estados está habilitado el comando.
- cada u_i es una actualización de variables que define el nuevo estado alcanzado. Tiene la forma $(v_1 = expr_1) \wedge \ldots \wedge (v_m = expr_m)$ donde las variables v_j son locales al módulo o, si la etiqueta es ε, puede ser también variables globales. Cada $expr_j$ es una expresión booleana o aritmética, dependiendo del tipo de v_j.
- Cada λ_i indica cual es la probabilidad de actualizar el estado de acuerdo a u_i. Por consiguiente se requiere que $\sum_i lambda_i = 1$.

La figura 8.1 muestra un simple modelo en notación PRISM. Allí se observa un sistema compuesto de dos módulos $M1$ y $M2$, donde cada un posee dos variables, uno de tipo booleano y otro de tipo entero. Además existe una variable global v de tipo entero. Además, cada módulo posee una acción local y una acción que se sincroniza a través de la etiqueta a.

8.3. Modelos de PRISM

Damos a continuación la semántica del lenguaje de PRISM en nuestros modelos. Sea m un módulo, $V_{m,local}$ las variables locales de m, $V_{m,global}$ las

variables globales que aparecen sintácticamente en m, $\mathcal{C}_m$ los comandos de m y Lab_m las etiquetas de los comandos que aparecen en $\mathcal{C}_m$. El átomo asociado a m, A_m, está dado por la 4-upla $(V, LV, Act, enabled)$ donde:

$$\begin{aligned} V &= V_{m,local} \cup V_{m,global} \\ L &= Lab_m \cup \{\varepsilon_m\} \end{aligned}$$

Por cada comando $(a, g, (\lambda_1, u_1), \ldots, (\lambda_m, u_m)) \in \mathcal{C}$ definimos la acción (a', c) donde a' es igual a a si $a \neq \varepsilon$ y es igual a ε_m en caso contrario. Por otro lado c esta definida por:

$$c(s, s') = \begin{cases} \lambda_i & \text{si } s' = u_i(s) \\ 0 & cc \end{cases} \tag{8.1}$$

La función $enabled$ esta dada por:

$$(a', c) \in enabled(s) \Leftrightarrow g(s)$$

Notar que la condición 2.5 se satisface ya que las acciones que pueden sincronizar solo pueden modificar variables locales al módulo.

8.4. Variables, BDDs y heurísticas

PRISM utiliza varios BDDs para representar al sistema, aquí analizaremos aquellos que pertenecen al motor híbrido y que además son utilizados por la reducción de orden parcial. También analizaremos como se codifican los estados y el orden con que son almacenadas en los BDDs.

En los distintos BDDs necesitaremos almacenar estados, etiquetas y transiciones. Para ello se necesitarán definir distintos tipos grupos de variables. El primer grupo de variables corresponde a los estados. Para codificar los estados se utiliza un enfoque similar al que presentamos en la sección 7.7.1. Necesitamos además un segundo grupo de variables para representar estados que corresponderían a la poscondición de las transiciones. Las etiquetas de las acciones se implementan utilizando una variable por cada una de ellas. Además las etiquetas pueden dar lugar a no-determinismo interno (i.e. la misma etiqueta para dos acciones distintas habilitadas) por lo que se necesitarán variables extras para implementar este no-determinismo.

Como vimos en la sección 7.4 el orden de las variables tiene un alto impacto en el tamaño de los BDDs, por lo que PRISM posee ciertas heurísticas para el ordenado de las variables. La forma en que se ordenan las variables en el motor híbrido es la siguiente. Primero se colocan las variables correspondientes a las etiquetas, luego las variables correspondientes a las elecciones no deterministas y finalmente las variables que codifican los estados. Como los estados se codifican dos veces, dichas variables están intercaladas, esto permite una gran reducción en los BDDs debido a la correlación entre las variables.

Las principales funciones codificadas en PRISM son las siguientes:

$init(s)$ esta función indica si s es el estado inicial.

$trans(s, a, c, s')$ esta función representa las funciones de transiciones del MDP, esto es la probabilidad de ir del estado s, al estado s' usando la acción a de las cuales se utiliza la elección no determinista c.

$trans01(s, a, c, s')$ esta función es similar a la anterior, indica si existe una transición en el MDP de s a s' usando la acción a y la elección probabilista c.

$reach(s)$ indica si s es un estado alcanzable desde el estado inicial.

Capítulo 9

Implementación

En este capítulos revisamos la arquitectura de PRISM incorporando reducción de orden parcial y explicamos detalles de como fueron implementados los nuevos heurísticas presentadas en el capítulo 6.

9.1. Nueva arquitectura

El principal cambio que se realizó fue la introducción de una nueva clase `POR_NondetModel` que hereda de la clase `NondetModel`. Esta nueva clase se encarga de generar un modelo reducido cada vez que se realiza una verificación de una fórmula $\mathrm{LTL}_{\setminus X}$.

En la arquitectura original de PRISM los modelos no tenían acceso a las propiedades que se querían verificar. Ya que para realizar la reducción se requiere modificar el modelo en base a la propiedad a verificar, se añadió el método `setExpression` que sirve para pasar una propiedad al modelo.

En la figura 9.1 se puede apreciar la nueva arquitectura de PRISM.

9.2. Nuevas variables

Para poder implementar POR fue necesario introducir nuevas variables en los BDDs. Por ejemplo, la relación de dependencia es una relación entre acciones. Entonces, para poder representarla como BDDs es necesario un segundo juego de variables para representar etiquetas. Por motivos similares se añadieron nuevas variables para representar un tercer conjunto de estados. Dado que los conjuntos amples se representa como conjunto de módulos, es necesario poder identificar módulos como BDDs, por lo que introducimos una nueva variable por cada módulo. La forma de representar los conjuntos de módulos es la misma que presentamos en 7.7.2.

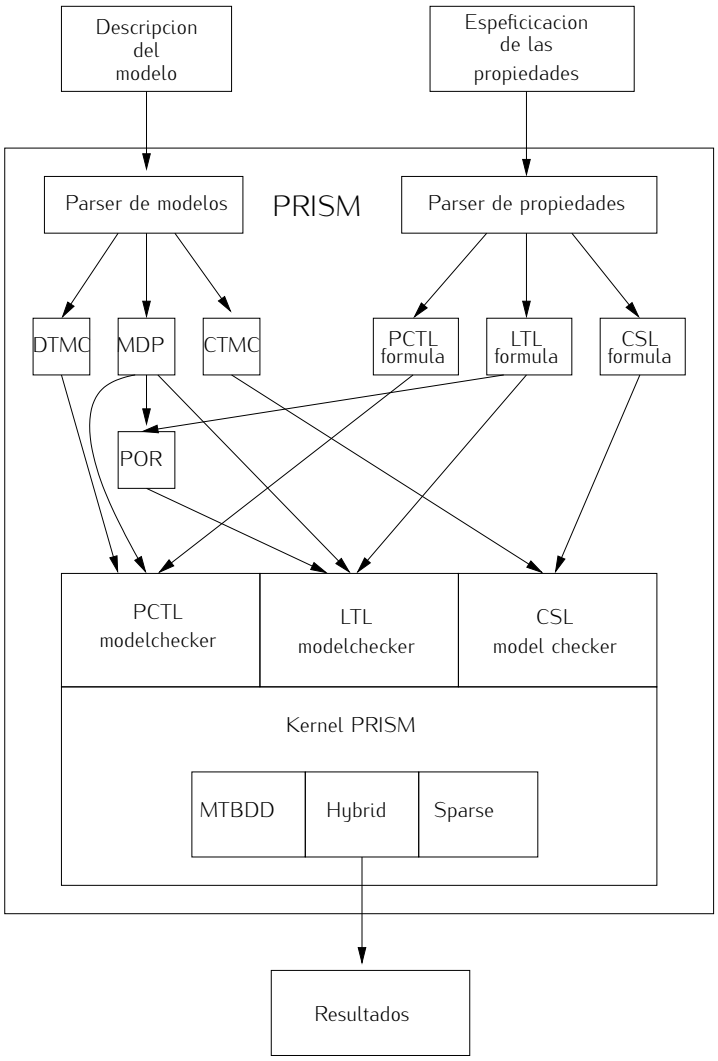


Figura 9.1: Nueva arquitectura de PRISM

9.3. Nueva matriz de transiciones

Para la realización de POR no es importante conocer cual es la probabilidad de ir de un estado a otro, solo es importante saber si la probabilidad es nula o no.

Cuando PRISM construye el BDD de las transiciones realizo optimizaciones de manera de minimizar la cantidad de variables asociadas al no-determinismo. Tal optimización resulta que no haya una correlación entre los valores de las variables de no-determinismo y las acciones de los módulos. Por consiguiente es imposible identificar la idea de "acción" en los BDDs. En este sentido, so-

breaproximaremos la idea de acción con la de etiqueta. Esta sobreaproximación no tiene impacto negativo en nuestra técnica dado que las variables de no-determinismo sólo tienen sentido para diferenciar acciones con la misma etiqueta y las acciones con las mismas etiquetas son de todas maneras dependientes entre sí.

Definimos entonces una nueva función de transición de estados $detTrans01(s, a, s')$ que indica que existe alguna acción cuya etiqueta es a y que va del estado s a s' con probabilidad no nula. Esta nueva matriz de transición puede ser implementada usando BDDs de la siguiente forma:

$$detTrans01(s, a, s') \doteq \exists c : trans01(s, a, c, s')$$

9.4. Dependencia e invisibilidad

Tanto la relación de dependencia como la de invisibilidad se obtienen haciendo un análisis sintáctico sobre los modelos PRISM teniendo en cuenta la propiedad bajo análisis. Para ello sólo es necesario considerar las ocurrencias de las variable. En el caso de la dependencia, también es necesario tener en cuenta cuando las variables son de lectura y escritura. Esto es fácil de obtener dado que en el texto, las variables de escritura aparecen primadas y las de lectura no. Por ejemplo si consideramos la acción:

```
[a] x != y -> 0.5: (x'=i-j) + 0.5 (x'= z)
```

la única acción escrita es `x` mientras que las variables leídas son `x`, `y` y `z`.

El BDD que implementa la relación de dependencia es $dependent(a, b)$, mientras que la relación de invisibilidad es implementada por $invisible(a)$.

9.5. La relación Involved

La relación *involved* indica cuales son los módulos que sincronizan con una determinada etiqueta de sincronización, es decir que $involved(a, m)$ indica que existe una acción en el módulo m cuya etiqueta es a. Este BDD al igual que las relaciones *dependent* e *invisible* se obtienen de manera sintáctica, observando las etiquetas que usan los diferentes módulos.

Se implementaron dos variaciones de esta relación: $localInvolved(a, M)$ que indica que la etiqueta a solo sincroniza con módulos que están en el conjunto M. Este BDD se puede definir de la siguiente manera:

$$localInvolved(a, M) \doteq \bigvee_{\overline{M}} (M = \overline{M}) \wedge (\forall m : involved(a, m) \implies m \in \overline{M})$$

La otra relación verifica si algún módulo de M manipula la etiqueta a:

$$someLocalInvolved(a, M) \doteq \exists m \in M : involved(a, m)$$

9.6. Relaciones de habilitación

El cálculo del sistema reducido depende directamente de las distintas relaciones de habilitación que vimos en el capítulo 6. $globalEnabled(s, a)$ indica si existe una acción con etiqueta a que esta habilitada en el estado s:

$$globalEnabled(s, a) \doteq \exists s' : detTrans01(s, a, s')$$

$localEnabled(s, a, M)$ indica si alguna acción con etiqueta a esta habilitada en el estado s y además que dicha etiqueta solo puede sincronizarse con módulos que están en M. De forma similar vamos a definir la relación $someLocalEnabled(s, a, M)$ donde se permite que a pueda sincronizar con módulos que no están en M. Las definiciones de estas funciones son:

$$localEnabled(s, a, M) \doteq globalEnabled(s, a) \wedge localInvolved(a, M)$$
$$someLocalEnabled(s, a, M) \doteq globalEnabled(s, a) \wedge someLocalInvolved(a, M)$$

Finalmente, la relación $moduleEnabled(s, M)$ que indica que alguno de los módulos de M tiene alguna acción habilitada y todas las acciones habilitadas por los módulos son locales. La forma que se implemento esta función es:

$$moduleEnabled(s, M) \doteq \exists a : localEnabled(s, a, M) \wedge \big(\forall a : someLocalEnabled(s, a, M) \implies localEnabled(s, a, M)\big)$$

9.7. Alcanzabilidad local

En esta sección reportaremos como implementamos alcanzabilidad local sobre BDDs.

Recordemos que $W_A^{\mathcal{B}}$ definida en la sección 6.8, contiene todas las variables que son escritas por acciones que no sincronicen con A ni con ninguna acción de los átomos que pertenecían a $\mathcal{A}$. $W_A^{\mathcal{B}}$ puede adaptarse a módulos (en lugar de átomos) y puede obtenerse por simple análisis sintáctico de los modelos PRISM. de manera semejante a como obtuvimos la relación *dependent* (ver sección 9.4).

La función de transición de estados locales, $localTrans01(s, a, t, m, M)$ que indica que existe un camino de s a t donde se ejecuta sólo una vez una acción etiquetada con a, a no es etiqueta de ningún módulo de M, y además pueden ocurrir acciones que no sean ni de m ni de ningún módulo en M, $localTrans01$ se define como sigue:

$$localTrans01(s, a, t, m, M) \doteq \exists s', t' : detTrans01(s|_{V \setminus W_m^M} \oplus s'|_{W_m^M}, a, t|_{V \setminus W_m^M} \oplus t'|_{W_m^M})$$

La relación $localReach(s, t, m, M)$ se obtiene como la convergencia de la siguiente iteración (notar que termina debido a la finitud del espacio de estados):

$$\begin{aligned} localReach_0(s, t, m, M) &\doteq (s|V \setminus W_m^M = s|V \setminus W_m^M) \\ localReach_{i+1}(s, t, m, M) &\doteq localReach_i(s, t, m, M) \vee \\ &\quad \exists s' a, localReach_i(s, s', m, M) \wedge localTrans01(s', a, t, m, M) \end{aligned}$$

Notar que la relación $localReach_i(s, t, m, M)$ indica que existe puede existir un camino de s a t donde se realizan a lo sumo i acciones de m y no se realiza ninguna acción de algún módulo en M.

9.8. Conjuntos persistentes

Los conjuntos persistentes pueden verse como conjuntos candidatos a ser ample. De hecho son como conjuntos ample que no necesariamente satisfacen la propiedad **A4**. Precisamente por esto pueden calcularse antes de construir el sistema reducido. La relación *persistenSet* se define de la siguiente manera:

$$\begin{aligned} persistentSet(s, M) \doteq\ & moduleEnabled(s, M) \wedge \\ & \big(\forall s', a, b : localReach(s, s', M) \wedge localEnabled(s, a, M) \wedge \\ & \neg localEnabled(s, b, M) \wedge globalEnabled(s', b) \\ & \quad \Longrightarrow invisible(a) \wedge \neg dependent(a, b) \wedge \\ & \neg someLocalInvolved(b, M)\big) \end{aligned}$$

9.9. Ample set

Finalmente, reportamos la construcción del sistema reducido. Este está definido por la relación $ampleSet(s, M)$, que indica que las acciones habilitadas en el estado s son las acciones habilitadas del sistema original y que pertenecen a los módulos en el conjunto M. El espacio de estados alcanzables estará definido por la relación $reach$ que se construirá conjuntamente con la relación $ampleSet$. Estas dos relaciones se obtienen como un punto fijo y cuyas iteraciones se definen debajo por medio de las operaciones $ampleSet_i$ y $reach_i$. El subíndice i indica la i-ésima iteración en el algoritmo BFS. Adicionalmente, se definen las relaciones $newReach_i(s)$ y $reachNewState_i(s, M)$. La relación $newReach_i$ representa a todos los estados alcanzables en el sistema reducido que en la i-ésima iteración del algoritmo BFS no tienen definido aún el conjunto ample. Esta relación se define de la siguiente manera:

$$\begin{aligned} newReach_0(s) &\doteq init(s) \\ newReach_{i+1}(s) \doteq \exists \overline{s}, a, M : \big(& ampleSet_i(\overline{s}, M) \wedge localEnabled(\overline{s}, a, M) \wedge \\ & detTrans01(\overline{s}, a, s) \wedge \neg reach_i(s) \big) \end{aligned}$$

Utilizando la relación $newReach_i$ se construye la relación de estados alcanzables como sigue:

$$\begin{aligned} reach_0(s) &\doteq init(s) \\ reach_{i+1}(s) &\doteq reach_i(s) \vee newReach_{i+1}(s) \end{aligned}$$

La relación $reachNewState_i$ es la encargada de verificar la condición **A4**. $reachNewState_i(s, M)$ indica que todas las acciones que pertenecen a alguno de los módulos de M posee una elección probabilista que va del estado s a un estado no alcanzado por el algoritmo BFS en la iteración i-ésima. La relación $reachNewState_i$ esta definido por:

$$\begin{aligned} reachNewState_i(s, M) \doteq \forall a, c : \big[& localEnabled(s, a, M) \wedge \big(\exists s' : trans01(s, a, c, s')\big) \implies \\ & \exists s' : trans01(s, a, c, s') \wedge \neg reach_i(s') \big] \end{aligned}$$

Notar que dado una etiqueta a pueden existir más de una acción habilitada en s que posea dicha etiqueta. Recordar que las acciones habilitadas en el estado s están identificadas por su etiqueta de sincronización y una valuación particular de las variables de no-determinismo (que están representadas por c).

Para construir el ample set se establece un orden en el conjunto de módulos $M_0 < M_1 \ldots < M_{2^n-1}$ de manera tal que $|M_i| \leq M_j$ si $i < j$. Notar que M_0 es el conjunto vacío y M_{2^n-1} es el conjunto conformado por todos los átomos. $ampleSet_i(s, M_j)$ indica que M_j es el menor de los j tal que conforma un conjunto ample válido. Es decir, si s es un estado alcanzado por una iteración anterior, M_j es el ample definido con anterioridad y si s pertenece a $newReach_i$ M_j es el menor de los conjuntos persistentes de s tal que todas las acciones alcanzan un estado no visitado. Para calcular el mínimo de los M_j se utiliza un nuevo conjunto de relaciones $ampleSet_{i,j}$. Si $k \leq j$ y $s \in newReach_i$ entonces $ampleSet_{i,j}(s, M_k)$ es válido si M_k es efectivamente el mínimo conjunto de módulos que es un ample set válido en la iteración i-ésima. Las relaciones $ampleSet_i$ y $ampleSet_{i,j}$ se definen como:

$$
\begin{aligned}
ampleSet_{0,0}(s, M), &\doteq 0 \\
ampleSet_{i+1,0}(s, M) &\doteq ampleSet_i(s, M) \\
ampleSet_{i,j+1}(s, M) &\doteq ampleSet_{i,j}(s, M) \vee \\
&\quad ((M = M_{j+1}) \wedge \neg \exists \overline{M} : \; ampleSet_{i,j}(s, \overline{M}) \wedge \\
&\quad newReach_i(s) \wedge persistentSet(s, M) \wedge reachNewState_i(s.M)) \\
ampleSet_{i,2^n-1}(s, M) &\doteq ampleSet_{i,2^n-2}(s, M) \vee ((M = M_{2^n-1}) \wedge \exists \overline{M} : \; \neg ampleSet_{i,2^n-2}(s, \overline{M}))
\end{aligned}
$$

$$
ampleSet_i(s, M) \doteq ampleSet_{i,2^n-1}(s, M)
$$

Notar que el conjunto vacío de módulos (i.e. M_0) no puede ser nunca un ample válido ya que viola la condición **A1**. Adicionalmente el conjunto de todos los módulos (i.e. M_{2^n-1}) es siempre un ample válido ya que es el conjunto de todas las acciones habilitadas. Observar que tomar el menor conjunto de módulos que es un ample válido, no garantiza que el ample que representa sea minimal.

Capítulo 10

Casos de estudio

En este capítulo se presentan diferentes casos de estudios que fueron analizados para comprobar el poder de la reducción de orden parcial presentada en el capítulo 5, para los sistemas que utilizan solo schedulers fuertemente distribuidos. Los datos a medir fueron el tiempo empleado, la cantidad de memoria y la tasa de reducción. En forma comparativa realizamos el chequeo en los sistemas sin realizar ninguna reducción, así como también reduciendo de acuerdo a A1-A4 y A1-A5.

Todos los casos de estudios fueron ejecutados en un Opteron 8212 (dual core) con 128 GB de RAM utilizando Ubuntu 8.04.3. En los experimentos realizados se midió la reducción obtenida mediante la aplicación del algoritmos de reducción de orden parcial, Para ello se tomó el cociente entre la cantidad de estados alcanzados por el sistema reducido y la cantidad de estados alcanzables por el sistema original. También se midió la performance tanto en tiempo como es el uso de la memoria. Los tiempos totales de ejecución fueron medidos mediante el comando `time`. Además se midió el tiempo empleado en la reducción. La memoria utilizada fue medida mediante la cantidad máxima de memoria física RAM utilizada por el proceso. Dicho dato fue obtenido de la variable `VmHWM` que se encuentra en el archivo `/proc/pid/status`. Adicionalmente se obtuvo la memoria utilizada por los MTBDDs y los vectores de iteración utilizados durante los cálculos numéricos.

10.1. Criptógrafos comensales

El problema de los criptógrafos comensales fue introducida por David Chaum en 1988 [11]. El problema consiste en tres amigos criptógrafos empleados de la NSA[1] que van a comer a un restaurante. Al final de la velada es sabido que

[1] *National Security Agency*, es la agencia de seguridad de EE.UU.

```
mdp

// cantidad de criptografos
const int N = 3;

// identidad de los criptografos
const int p1 = 1;
const int p2 = 2;
const int p3 = 3;

// indica cual criptografo paga, 0 indica que es la NSA
global pay : [0..N];

module crypt1
    coin1 : [0..2]; // valor de la moneda
    s1 : [0..1]; // indica si finaliza o no
    agree1 : [0..1]; // lo que dice el criptografo

    // se arroja la moneda
    [] coin1=0 -> 0.5 : (coin1'=1) + 0.5 : (coin1'=2);

    // las monedas coinciden y el criptografo no paga
    [] s1=0 & coin1>0 & coin2>0 & coin1=coin2    & (pay!=p1) -> (s1'=1) &
        (agree1'=1);

    // las monedas difieren y el criptografo no paga
    [] s1=0 & coin1>0 & coin2>0 & !(coin1=coin2) & (pay!=p1) -> (s1'=1);

    // las monedas coinciden y el criptogafo paga
    [] s1=0 & coin1>0 & coin2>0 & coin1=coin2    & (pay=p1)  -> (s1'=1);

    // las monedas difieren y el criptografo paga
    [] s1=0 & coin1>0 & coin2>0 & !(coin1=coin2) & (pay=p1)  -> (s1'=1) &
        (agree1'=1);

    [done] s1=1 -> true;
endmodule

// los demas criptografos construidos por renombramiento
module crypt2 = crypt1 [ coin1=coin2, s1=s2, agree1=agree2, p1=p2,
    coin2=coin3 ] endmodule
module crypt3 = crypt1 [ coin1=coin3, s1=s3, agree1=agree3, p1=p3,
    coin2=coin1 ] endmodule
```

Listing 10.1: Criptógrafos comensales en PRISM

alguno de los comensales pagó por la cena, ó que el pago fue efectuado por la NSA. Los comensales deben saber si la comida fue pagada por la NSA o no, de tal manera que si la NSA no pagó no se sepa cual de los criptógrafos lo hizo. La solución propuesta por Chaum consiste en que cada uno de los criptógrafos arroje una moneda justa y que se la muestre al comensal que se encuentra a su derecha. Luego cada uno de los criptógrafos dice en voz alta si su moneda y la de su vecino de la izquierda coinciden o no, excepto que el criptógrafo haya efectivamente pagado la cuenta, en cuyo caso dice lo contrario. La forma

Modelo	Sin reducir	Reducido	
n	total	POR	total
7	4s	1s	4s
8	14s	4s	11s
9	1m1s	5s	23s
10	4m56s	8s	1m20s
11	31m37s	14s	5m59s

Cuadro 10.1: Comparación del tiempo empleado en los criptógrafos comensales

de determinar si la NSA pagó la cuenta es contar el número de coincidencias y ver que son impares. En caso de ser par alguno de los criptógrafos pagó la comida.

Para ver que este protocolo realmente no revela información basta analizar alguno de los criptógrafos que no haya pagado la cuenta y que desea inferir cual de los otros dos fue. Notar que cuando paga la NSA no hay nada que descubrir y en el caso del criptógrafo que pago, él sabe quién fue. Veamos los dos casos posibles, el primero es cuando el criptógrafo ve que ambas monedas son iguales. Como no pagó la NSA, el numero de diferencias es impar, luego uno de sus vecinos dijo que las monedas coincidían y el otro dijo que eran diferentes. Si la moneda que el criptógrafo no ve es igual a las que observó, el que dijo que eran diferentes es el que pagó. Mientras que, si dicha moneda es diferente, el que pagó fue el otro vecino. Debido a que la moneda es justa, ambas posibilidades tienen la misma chance de ocurrir, por lo que el criptógrafo no puede inferir cual de sus vecinos pagó. El segundo caso es cuando las monedas que observa el criptógrafo son diferentes, luego los otros dos criptógrafos dicen lo mismo, pero de forma similar al caso anterior dependiendo de la moneda que esta oculta, es el criptógrafo que pagó.

Este protocolo puede ser extendido a más de tres criptógrafos, el resultado de anonadad se sigue cumpliendo, lo único que para saber si la NSA pagó hay que ver que la cantidad de coincidencias tenga la misma paridad que la cantidad de criptógrafos. En la figura 10.1 se puede observar la implementación de este protocolo en PRISM. La propiedad a analizar son:

```
Pmin=? [ F s1=1&s2=1&s3=1 & outcome = 0]
Pmax=? [ F s1=1&s2=1&s3=1 & outcome = 0]
```

Si se analiza el modelo se puede ver que las únicas acciones que son visibles son las que realizan el lanzamiento de las monedas. Además dichas acciones son independientes. Por lo tanto, toda la reducciones posibles pueden ser obtenidas tomando amples unitarios. En nuestros experimentos pudimos comprobar lo anterior. La reducción obtenida utilizando amples arbitrarios fue la misma que utilizando sólo amples unitarios. En las tablas 10.1, 10.1 y 10.1 se pueden observar los resultados experimentales obtenidos. Primero notar que el

Modelo	Sin reducir		Reducido	
n	vectores	RAM	vectores	RAM
7	11MB	593MB	4MB	613MB
8	46MB	632MB	21MB	646MB
9	199MB	797MB	66MB	705MB
10	977MB	1.55GB	264MB	948MB
11	4.75GB	5.38GB	1.12GB	1.83GB

Cuadro 10.2: Comparación del uso de la memoria en los criptógrafos comensales

Modelo	Reducciones			
n	% estados	% vectores	% RAM	% tiempo
7	40.18	38.21	103.32	100.00
8	35.10	46.00	102.22	78.57
9	30.72	33.51	88.53	37.70
10	26.91	27.02	59.56	27.03
11	23.58	23.61	33.94	18.92

Cuadro 10.3: Reducciones hechas en los criptógrafos comensales

tiempo empleado en la reducción es cada vez menor a medida que el tamaño de los modelos crece. Para los casos más grandes, cuando *n* vale 10 y 11, el tiempo empleado en la reducción es inferior al 10 % del tiempo total. En segundo lugar notar la similitud que existe entre la tasa de reducción de estados y la proporción entre los tamaños de los vectores utilizados por el motor híbrido. Sin embargo, notar que esto no sucede con la máxima cantidad de memoria utilizada por el proceso. Para los modelos pequeños el uso de memoria RAM es similar, mientras que en modelos más grandes se nota una reducción apreciable. La no correlación entre la proporción de estados y la proporción de la memoria RAM se debe a la memoria adicional que se requiere para construir los BDDs de POR.

10.2. Binary Exponential Backoff

El algoritmo de *binary exponential backoff* es utilizado en el protocolo CSMA/CD (Carrier Sense Multiple Access with Collision Detection) está estandarizado en la norma IEEE 802.3, también conocida como *Ethernet*. Este algoritmo es usado para resolver las colisiones que pueden ocurrir en un canal compartido donde varios *hosts* intentan enviar información de manera simultánea. El tiempo esta dividido en intervalos. Cuando un host quiere transmitir un mensaje escucha la línea hasta que esta se desocupa y luego envía su mensaje. Si el host detecta que hubo una colisión con otro mensaje espera cero o un intervalo de tiempo con probabilidad $\frac{1}{2}$ antes de transmitir nuevamente. Si nuevamente ocurre una colisión espero 0,1,2, ó 3 intervalos de tiempo con probabilidad $\frac{1}{4}$ antes de volver a intentar transmitir. El protocolo continua trabajando de esta

Modelo	Sin reducir	A1-A5		A1-A4	
$h/N/2^K$	total	POR	total	POR	total
4 / 3 / 4	1m19s	14s	1m9s	23s	1m16s
5 / 3 / 4	14m35s	42s	6m56s	1m32s	7m0s
6 / 3 / 4	5h10m	2m11s	1h9m	6m19s	49m13s
4 / 3 / 8	3m55s	20s	3m20s	26s	2m25s
5 / 3 / 8	1h31m	55s	27m12s	2m9s	22m6s
6 / 3 / 8	-	3m9s	9h38m	8m46s	5h57m

Cuadro 10.4: Comparación del tiempo empleado en *Binary Exponential Backoff*

Modelo	Sin reducir		A1-A5		A1-A4	
$h/N/2^K$	vectores	RAM	vectores	RAM	vectores	RAM
4 / 3 / 4	25MB	666MB	14MB	714MB	10MB	754MB
5 / 3 / 4	367MB	1.22GB	99MB	1008MB	73MB	1.29GB
6 / 3 / 4	8.71GB	10.09GB	976MB	2.29GB	593MB	3.31GB
4 / 3 / 8	93MB	865MB	51MB	807MB	36MB	870MB
5 / 3 / 8	2.85GB	4.07GB	509MB	1.63GB	330MB	1.92GB
6 / 3 / 8	-	-	8.64GB	10.40GB	5.24GB	9.25GB

Cuadro 10.5: Comparación del uso de la memoria de *Binary Exponential Backoff*

manera. Es decir en la colisión i-ésima espera entre 0 y $2^i - 1$ intervalos con probabilidad $\frac{1}{2^i}$ para cada intervalo. Esto ocurre hasta la colisión numero K, a partir de la cual en las sucesivas colisiones se espera entre 0 y $2^K - 1$ intervalos de tiempo con probabilidad $\frac{1}{2^K}$. Si ocurren más de N colisiones con $K \leq N$ el host aborta la transmisión. En la figura 10.2 se puede ver como modelamos este protocolo en PRISM. Las propiedades que analizamos son la probabilidad de que alguno de los host consiga enviar su mensaje satisfactoriamente y la probabilidad de que alguno de los host aborte la transmisión. Estas propiedades pueden ser expresadas utilizando LTL de la siguiente manera:

```
F sc
F gu
```

La reducción observada cuando se permitían ample sets de tamaño acotado era siempre la misma que cuando se permitían amples unitarios. Debido a esto sólo mostraremos los resultados para cuando los amples son unitarios y para cuando los amples son arbitrariamente grandes. Si bien el ample set unitario no es equivalente a comprobar **A1-A5**, en este caso es equivalente, ya que no existe no determinismo interno en los módulos. En este caso de estudio se puede apreciar la utilidad de considerar amples arbitrarios y no solo amples unitario. En las tablas 10.2, 10.2 y 10.2 se encuentran los resultados experimentales realizados. Al igual que con los igual que con los criptógrafos comensales el tiempo utilizado en la reducción de orden parcial se vuelve despreciable cuando los modelos se hacen más grandes. Sin embargo la reducción obtenida en los

Modelo	A1-A5				A1-A4			
$h/N/2^K$	% estados	% vectores	% RAM	% tiempo	% estados	% vectores	% RAM	% tiempo
4 / 3 / 4	36.07	55.29	107.17	87.34	23.79	39.36	113.18	96.20
5 / 3 / 4	19.85	27.01	80.85	47.54	12.19	20.01	105.60	48.00
6 / 3 / 4	10.35	10.95	22.65	22.48	6.09	6.66	32.82	15.86
4 / 3 / 8	30.24	55.21	93.30	85.11	20.01	38.70	100.61	61.70
5 / 3 / 8	16.09	17.47	40.14	29.78	10.04	11.33	47.03	24.20
6 / 3 / 8	8.18	-	-	-	4.93	-	-	-

Cuadro 10.6: Reducciones hechas en *Binary Exponential Backoff*

estados no es similar a la reducción obtenida en los vectores. Aunque en los casos 6/3/4 y 5/3/8 las reducciones son muy similares. Observar que el tiempo utilizado en la reducción con **A1-A4** es menor que el tiempo empleado con **A1-A4**. Cabe aclarar que **A1-A4** requiere analizar una cantidad exponencial de amples posibles. En cuanto a la memoria utilizada, la reducción con **A1-A5** necesita levemente menos memoria que las reducciones con **A1-A4**. Sin embargo en el caso 6/3/8, **A1-A4** utiliza menos memoria. No se obtuvieron resultados para el modelo 6/3/8 cuando no se realiza POR. Esto es debido a que el proceso necesitaba más de 32GB de memoria (la cota máxima que se le asignó a cada proceso),

```
const int H = 3; // nunero hosts
const int N ;    // N = cantidad maxima de intentos antes de abortar
const int K ;    // sqrt(K) = cantidad maxima de slots para esperar

global cr : [0..H+1];   // chan_req
global ls : bool ;    // line_seized_flag
global gu : bool ;    // gave_up_flag

module clock
    pc : [0..5] ;
    // 0 start
    // 1 begin_slot
    // 2 mid_slot
    // 3 reset
    // 4 end_slot

    [] (pc=0) -> (pc'=1) ;
    [tick] (pc=1) -> (pc'=2) ;
    [tack] (pc=2) -> (pc'=3) ;
    [] (pc=3) -> (cr'=0) & (pc'=4) ;
    [tock] (pc=4) -> (pc'=0) ;
endmodule

module Host1
    p1  : [0..7] ;
    // 0 wait_tick
    // 1 init_cycle
    // 2 check_collision
    // 3 process_collision
    // 4 wait_tack
    // 5 wait_tock
    // 6 line_seized
    // 7 gave_up
    na1 : [0..N];     // nr_attempts
    ev1 : [0..K];     // exp_val
    ax1 : [0..K];     // aux_exp
    wt1 : [0..K];     // slots_to_wait

    // Begining of the slot: try to seize the line or just wait.
    //   Esperar por el slot
    [tick] (p1=0) & (wt1>0) -> (p1'=4) & (wt1'=wt1-1) ;
    //   Intentar tomar la linea
    [] (p1=0) & (wt1=0) & (cr<H+1) -> (p1'=1) & (cr'=cr+1) ;
    [tick] (p1=1) -> (p1'=2) ;
    //   Linea tomada
    [] (p1=2) & (cr=1) -> (p1'=6) & (ls'=true) ;
    //   Se alcanzo el maximo numero de coliciones
    [] (p1=2) & (cr>1) & (na1>=N) -> (p1'=7) & (gu'=true) ;
    //   No se pudo tomar la linea
    [] (p1=2) & (cr>1) & (na1<N) -> (p1'=3) & (na1'=na1+1) & (ax1'=1) &
        (wt1'=0) ;
    // Se elige un numero al azar de intervalos
    [] (p1=3) & (ax1>=ev1) & (ev1>=K) -> (p1'=4) ;
    [] (p1=3) & (ax1>=ev1) & (2*ev1<=K) -> (p1'=4) & (ev1'=2*ev1) ;
    [] (p1=3) & (ax1<ev1) & (2*ax1<=K) & (wt1+ax1<=K) ->  0.5 :
        (p1'=3)&(ax1'=2*ax1) + 0.5 : (p1'=3)&(wt1'=wt1+ax1)&(ax1'=2*ax1) ;
    [tack] (p1=4) -> (p1'=5) ;
    [tock] (p1=5) -> (p1'=0) ;
    // Linea tomada
    [] (p1=6) -> (p1'=6) ;
    // Transmision abortada
    [] (p1=7) -> (p1'=7) ;
endmodule

module Host2 = Host1 [ p1=p2, na1=na2, ev1=ev2, ax1=ax2, wt1=wt2 ] endmodule
module Host3 = Host1 [ p1=p3, na1=na3, ev1=ev3, ax1=ax3, wt1=wt3 ] endmodule
```

Listing 10.2: Binary Exponential Backoff en PRISM

Capítulo 11

Conclusiones

En el marco del presente trabajo hemos realizado una revisión de las diferentes técnicas de reducción de orden parcial en sistemas no-probabilistas y probabilistas. Aunque nos centramos en sistemas probabilistas que pueden ser modelados como MDPs, especialmente en aquellos cuyos schedulers son fuertemente distribuidos.

Analizamos las diferentes heurísticas existentes. En el caso de la heurística de **A4** se realizó un refinamiento de la heurística de BFS para el caso no-probabilista. Si bien las heurísticas basadas en BFS detectan erróneamente ciclos, nuestra heurística permite eliminar casos donde incluso las técnicas de DFS observan falsas componentes terminales. La heurística de **A3** presentada en este trabajo es similar a la heurística implementada en LiQuor. La principal diferencia es que nuestro algoritmo considera el comportamiento del sistema en su conjunto y no el comportamiento individual de los átomos.

La técnica de reducción de orden parcial se implementó edentro del model checker simbólico PRISM, Nuestra implementación es compatible con la comprobación de fórmulas LTL desarrollada en [7], así como también con la paralelización de los cálculos numéricos (trabajo que se está realizando actualmente). En general nuestra técnica no presenta conflicto con ninguna mejora que no modifique los MTBDDs ni que utilice la estructura sintáctica de los modelos.

Se realizaron pruebas en diferentes sistemas distribuidos. Los casos de estudios presentes en este trabajo muestran la utilidad de la reducción de orden parcial. También se observa que eliminar la condición **A5** puede permitir mejores reducciones. Sin embargo POR no es una técnica que logrará reducciones importantes en todo tipo de modelos, sino para aquellos sistemas distribuidos cuyos nodos posean una gran independencia. Para ciertos modelos la reducción es casi nula, o el taza de reducción no es suficiente para contrarrestar el tiempo y el uso de recursos que utiliza POR. Para los modelos que presentan

una independencia considerable se han observado, sin embargo, que la técnica obtiene sistemas reducidos pequeños. La reducción en dichos sistemas permitió una redución considerable en el tiempo de model checking.

Los cambios introducidos introducidos en PRISM para permitir POR pueden ser añadido en futuras versiones oficiales del model checker.

Bibliografía

[1] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978. 7

[2] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Formal Methods in System Design*, 18(2):97–116, 2001. 6, 6.3.1, 6.4.2

[3] Robert B. Ash and Catherine A. Doléans-Dade. *Probability and Measure theory*. Academic Press, 2 edition, 1999. 3.5

[4] C. Baier, M. Größer, and F. Ciesinski. Partial order reduction for probabilistic systems. In *QEST '04*, pages 230–239, Washington, DC, USA, 2004. IEEE CS. 1, 5, 5.4

[5] C. Baier, M. Größer, and F. Ciesinski. Quantitative analysis of distributed randomized protocols. In *Proc. of FMICS'05*, pages 2–7. ACM, 2005. 5.1

[6] C. Baier and J. P. Katoen. *Principles of Model Checking*. MIT Press, New York, May 2008. 1

[7] Carlos S. Bederián. Model checking cuantitativo de propiedades ltl en prism. Master's thesis, Facultad de Matemática Astronomia y Física, 2007. 8, 11

[8] Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A tool for bdd-based verification of real-time systems. In *In: Computer-Aided Verification (CAV 2003), Volume 2725 of Lecture Notes in Computer Science, Springer-Verlag*, pages 122–125. Springer-Verlag, 2003. 7

[9] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986. 7

[10] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, 1990. 1, 7

[11] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988. 10.1

[12] Frank Ciesinski, Christel Baier, Marcus Größer, and Joachim Klein. Reduction techniques for model checking markov decision processes. In *QEST*, pages 45–54. IEEE Computer Society, 2008. 6

[13] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model checker. *STTT*, 2(4):410–425, 2000. 7

[14] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000. (document), 1, 6, 6.3, 6.3.2

[15] P. R. D'Argenio and P. Niebert. Partial order reduction on concurrent probabilistic programs. In *QEST '04*, pages 240–249, Washington, DC, USA, 2004. IEEE CS. 5, 5.1, 5.4, 6.2, 6.4.1, 6.5

[16] Luca de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997. 5.3

[17] S. Giro and P.R. D'Argenio. Partial order reduction for probabilistic systems assuming distributed schedulers. Technical report, FaMAF, UNC, 2009. Disponible en `http://cs.famaf.unc.edu.ar/~sgiro/TR-A-INF-09-2.pdf`. 1

[18] S. Giro, P.R. D'Argenio, and L.M. Ferrer Fioriti. Partial order reduction for probabilistic systems: a revision for distributed schedulers. In M. Bravetti and G. Zavattaro, editors, *Proceedings CONCUR 2009,* Bologna, Italy, volume 5710 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2009. 1, 5

[19] Sergio Giro. *On the Automatic Verification Distributed Probabilistic Automata with Partial Information*. PhD thesis, Facultad de Matemática Astronomia y Física, Universidad Nacional de Córdoba, 2010. 1

[20] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Proc. of TACAS'06*, LNCS 3920, pages 441–444. Springer, 2006. 7

[21] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, New York, NY, USA, 2 edition, 2004. 7

[22] B. Jeannet, P.R. D'Argenio, and K.G. Larsen. RAPTURE: A tool for verifying Markov Decision Processes. In I. Cerna, editor, *Tools Day'02,* Brno, Czech Republic, Technical Report. Faculty of Informatics, Masaryk University Brno, 2002. 7

[23] Leslie Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983. 5.2

[24] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993. 1, 7

[25] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002. 1, 8

[26] Don Peled. Computer aided verification, 5th international conference, cav '93, elounda, greece, june 28 - july 1, 1993, proceedings. In Costas Courcoubetis, editor, *CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993. 1, 5.2, 5.4, 6, 6.4

[27] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Comp. Soc. Press, October-November 1977. 4.1

[28] F. Somenzi. Cudd: Cu decision diagram package release, 1998. 1, 8