Universidad Nacional de Córdoba
Facultad de Matemática, Astronomía y Física

Trabajo Especial

Model Checking Cuantitativo con enfoque de la Teoría de Autómatas

Matías Lee Maximiliano Combina

Director: Dr. Pedro R. D'Argenio

Resumen

El estudio de la corrección de los sistemas es de suma importancia debido a que las fallas en estos sistemas pueden tener consecuencias catastróficas. Este trabajo contribuye a extender la técnica de Model Checking Cuantitativo desarrollada en [dA97], el cual presenta un algoritmo para calcular la probabilidad máxima de que una propiedad expresada en LTL sea satisfecha por un sistema con componentes probabilísticas y/o no determinísticas. Este algoritmo reduce el problema del cálculo de la probabilidad máxima a un problema de alcanzabilidad. Utilizando un enfoque similar, hemos desarrollado un algoritmo para el cálculo de la probabilidad mínima.

Además, hemos creado una herranienta para el cálculo de dicha probabilidad en un sistema bajo estudio. Esta herramienta se basa en la implementación de nuestros resultados y la integración de estos con otras aplicaciones ya existentes, entre ellas, Rapture [JDL02], un model checker de probabilidad cuantitativa.

Clasificación: D.2.4 Software/Program Verification.

Palabras claves: Model Checking, Model Checking cuantitativo, teoría de autómatas, verificación formal, SSS Reducer, Rapture, ltl2dstar.

Agradecimientos

Agradezco a toda la comunidad de FaMAF que me ha acompañado durante estos años y han dejado muchos lindos recuerdos en mí. Recién estoy terminando y ya extraño los días de estudiante.

También a mi familia, especialmente a mi mamá, siempre se esforzaron para que este día llegara.

Por último, a maxlo, un amigazo y un gran compañero.

Chun

Agradezco, también, a toda la comunidad de FaMAF, a los profesores y compañeros y amigos que me apoyaron incondicionalmente a lo largo de esta importante etapa. Son muchos como para nombrarlos aquí sin olvidarme de alguno, pero todos saben (espero) que este agradecimiento va especialmente dedicado a ellos

También a mi familia y a Mariana, mi novia, quienes me alentaron durante los (casi incontables) momentos de duda y vacilación, y me aguantaron en mis etapas inaguantables

And "last but not least", a Matías "Chun" Lee*, un verdadero amigo y enorme persona, gracias a quien este trabajo fue posible.*

Maximiliano "maxlo" Combina.

Especialmente dedicado, al Dr. Pedro R. "doc" D'Argenio.

Chun & maxlo.

Índice general

Capítulo 1

Introducción

1.1. Los Problemas

La interacción cotidiana con sistemas basados en computadoras es un hecho inevitable y se presenta en cualquier ámbito. No sólo eso, cada día los distintos sistemas se interrelacionan más con el entorno en el que se encuentran y deben reaccionar en función de los estímulos recibidos, de aquí la denominación de *sistemas reactivos*. Es más, luego de realizar la acción predefinida para un estímulo en particular, el sistema tiene que estar listo para seguir interactuando. Este comportamiento, idealmente, nunca termina.

Un sistema reactivo puede presentar distintos niveles de complejidad, por ejemplo, tanto una máquina expendedora de café como un programa para el control aéreo son sistemas reactivos, ya que estos interactúan en función de lo que ocurre en su entorno. Mientras que el entorno de la máquina de café es simplemente un usuario que desea tomar café, el entorno de un programa para el control aéreo es mucho más complejo, pues este incluye los aviones que desean despegar, los aviones que desean aterrizar, el tiempo estimado de despegue y aterrizaje, el total de pistas en el aeropuerto, etc.

Así como la complejidad varía, tambien varía la importancia de un correcto funcionamiento. Si bien es cierto que el usuario de la máquina de café sufrirá un gran disgusto si al pedir un café recibe un té, este mal funcionamiento dudosamente resultará en una tragedia. En cambio, este no es el caso del programa para el control de vuelo. Un mal funcionamiento en este podría resultar en la muerte de cientos de personas. Este tipo de sistemas, con características críticas, deben proveer un servicio correcto y eficiente. La verificación de dichos sistemas es más difícil a medida que la complejidad de los mismos aumenta, debiéndose recurrir a diferentes técnicas para la verificación de su correcto funcionamiento.

La complejidad de los sistemas reactivos no sólo está en función de la cantidad de variables que deben manejar. Además, se suman otros factores: el *no determinismo* y/o la *aleatoriedad* de ciertas acciones y la necesidad de

no finalización de las ejecuciones. Estos factores a veces se introducen para garantizar el logro del resultado deseado, como es el caso de los acuerdos bizantinos donde la componente *no determinista* se mezcla con la *aleatoria*, o en el caso de un sistema operativo, este funciona indefinidamente, satisfaciendo los requerimientos de las aplicaciones y el usuario. En otros casos, se deben al no determinismo generado por el entorno en el cual las componentes del sistema deben interactuar. Estos casos se presentan en situaciones tales como la pérdida de un mensaje en la red o el mal funcionamiento de un sensor. Luego, los factores que inciden sobre la complejidad del sistema, hacen imposible realizar verificaciones basadas en *pre* y *pos* condición.

En consecuencia surge el problema de cómo representar este tipo de propieadades, en la cual entra en juego el tiempo. Supongamos que se define el predicado "se envía un mensaje con destino d en el momento t" como envia(d, t) y el predicado "se recibe un mensaje en d en el momento t" como recibe(d, t). Luego una posible forma de representar el predicado antes mencionado es:

$$\forall d, \forall t(\text{envia}(d, t) \Rightarrow \exists t' > t : \text{recibe}(d, t'))$$

Si bien es posible expresar la propiedad en términos formales, se aprecia un nivel de complejidad considerable para la representación de un propiedad simple, la cual, no será fácilmente manipulable al momento de querer realizar alguna verificación formal automática.

Otro problema que surge es la consideración de probabilidades dentro del comportamiento de los sistemas. Esta consideración implica que el conjunto de propiedades asociadas a estos sistemas se sale de la lógica usual. Por esta razón no podemos utilizar predicados como *"el programa termina"*, en su lugar deberemos utilizar *"el programa termina con probabilidad 1"*. Otras veces el establecer que una propiedad es falsa es muy débil y a la vez establecer su veracidad es imposible. Un ejemplo de esto son los protocolos de retransmición acotada, donde uno no puede asegurar que todo mensaje se recibe pero si puede analizar la validez de lo siguiente *"todo mensaje se recibe con probabilidad 0,99"*

Resumiendo, es de suma importancia la verificación de los sistemas con características críticas, pues un mal funcionamiento de estos pueden llevar a una tragedia. Sin embargo, al momento de querer realizar esto surgen los siguientes inconvenientes:

1. Dificultad para expresar propiedades, en donde el tiempo juega un papel importante, de forma simple y automáticamente manipulable.

2. Imposibilidad de demostrar, de forma tradicional, propiedades relevantes para sistemas de gran complejidad.

3. El no determinismo y las probabilidades de los sistemas hacen imposi-

ble hablar de la total validez o invalidez de las propiedades.

1.2. Las Soluciones

Los problemas mencionados en la sección anterior se empezaron a analizar en las últimas décadas, dando como solución la *lógica temporal* y las técnicas de *model checking*.

Lógica Temporal

La *lógica temporal* fue sugerida por A. Pnueli para la especificación formal de propiedades sobre el comportamiento de un sistema en 1977 [Pnu77]. Estás están especificamente disenãdas para la especificación de propiedades del comportamiento del sistema a través del tiempo. La notación es clara, simple y muy intuitiva, pero siempre en presencia de un fuerte formalismo. Por ejemplo, la fórmula "F *llueve*" (finalmente llueve), representa el predicado *"en algún momento en el futuro lloverá"*. En este trabajo nos enfocaremos en la *lógica temporal lineal (LTL)*, la cual permite establecer propiedades sobre las ejecuciones del sistema bajo estudio.

Model Checking

El *model checking* es un técnica tal que dado un modelo de estados finitos del sistema y una propiedad lógica sistemáticamente verifica, de modo automático, si la propiedad es válida o no en el modelo [Kat99].

Los algoritmos utilizados, para verificar si la propiedad es válida en el modelo, genereralmente se basan en una búsqueda exhaustiva en el espacio de estados posibles del modelo; a esto se debe la necesidad de una cantidad finita de estados. En otras palabras, el algoritmo debe recorrer todos los posibles estados y determinar si la propiedad se cumple para cada posible secuencia de estados. Veamos esto con un ejemplo:

Ejemplo 1. *Supongamos un sistema que manda mensajes indefinidamente, el cual se inicializa listo para madar un mensaje. Cada vez que manda un mensaje, este puede ser mandado con problemas o sin problemas (no determinismo). En el primer caso el mensaje llega. En el segundo caso, no. Sin importar el caso, luego de esto el sistema vuelve a estar listo para mandar otro mensaje. Este sistema se encuentra modelado en la figura 1.2.*

Supongamos que deseamos verificar que, para todo comportamiento posible, es válida la propiedad: "siempre que se manda un mensaje, el mensaje se recibe". El algoritmo de model checking utilizado debería reportar la invalidez de la propiedad. Esto se debe a que un posible comportamiento del modelo es ABC, donde en B se manda un mensaje y en C no se recibe.

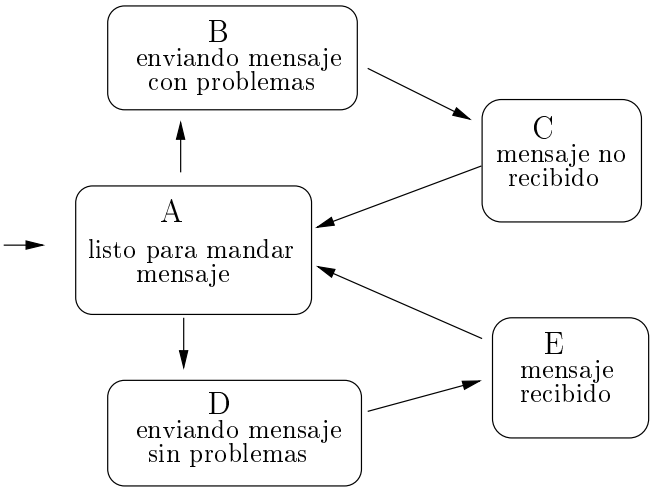


Figura 1.1: Modelo del sistema representado en el ejemplo 1

Si bien el uso de Model Checking permite la verificación de propiedades en sistemas complejos, donde puede entrar en juego el no determinismo, notemos que no posee una forma de verificar propiedades en sistemas compuestos por componententes probabilísticas. Por esta razón, en los años 90, empieza el estudio de las primeras técnicas de *Model Checking Cuantitativo*. En estas técnicas se verifican propiedades tales como *"todo mensaje llega con una probabilidad de 0,99"*. Este tipo de propiedades recibe el nombre de *propiedades cuantitativas* y están presentes en los sistemas donde juegan un papel importante el no determinismo de ciertas acciones, como ya mencionamos, ya sea para lograr el objetivo deseado o por la interacción con componentes aleatorios del entorno del sistema. El Model Checking Cuantitativo, además de lidiar con los mismos problemas que el Model Checking, debe afrontar nuevos. Entre ellos, el de cómo relacionar el no determinismo con las probabilidades. Por ejemplo, si una persona tira una moneda al aire se sabe que la probabilidad de que salga cara es de 0,5. Pero si no estamos seguros si la persona va a arrojar la moneda, ¿cuál es la probabilidad que salga cara?. Y la complejidad de los modelos aumenta. Para ilustrar esto, estudiemos el siguiente ejemplo:

Ejemplo 2. *En una panadería hay 2 hornos que se utilizan para cocinar tortas. Uno funciona correctamente y el otro no. El que anda bien siempre cocina la torta a punto, en cambio, el otro la deja cruda con una probabilidad de* 0,1*. El proceso de cocción de cada torta se inicializa con la elección del horno. Una vez que la torta sale del horno es envasada. En caso de que la torta esté cruda, esto se puede detectar con una probabilidad de* 0,5*; si es el caso, se la vuelve a meter al horno, caso contrario, es envasada. Una vez que la torta está envasada, se repite el proceso de cocción con la siguiente torta.*

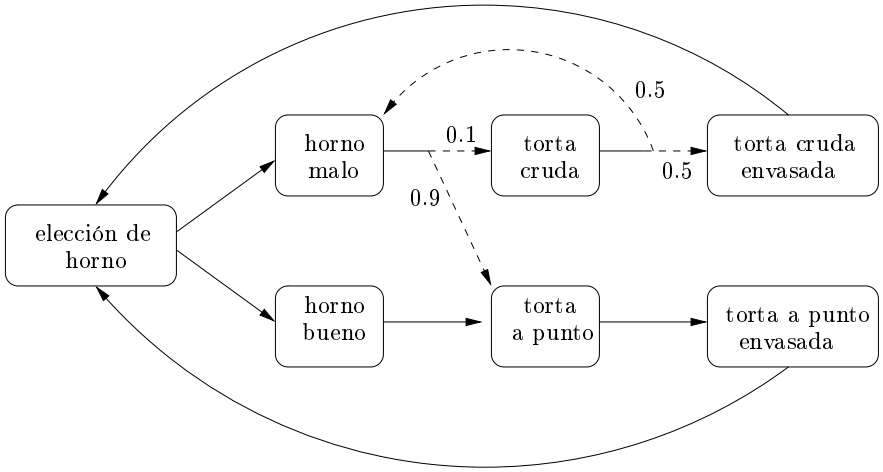


Figura 1.2: Proceso de cocción - Ejemplo 2

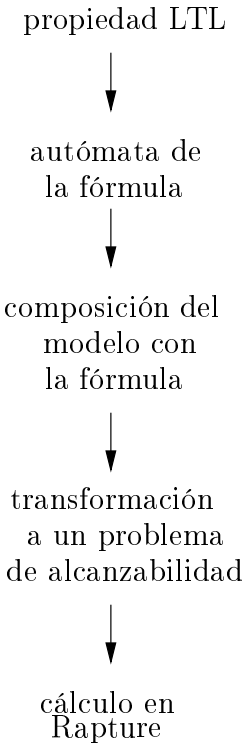
En este ejemplo se podría estar interesado en una propiedad cuantificada como *"al menos el 95 % de las tortas salen cocidas"*. La técnica de model checking cuantitativo permite, de forma automática, la verificación de esta propiedad.

1.3. Objetivos

El objetivo de este trabajo es presentar y extender la técnica de Model Checking Cuantitativo desarrollada en [dA97]. En [dA97] se presenta una técnica para el cálculo de la probabilidad máxima de que una propiedad, expresada en LTL, se satisfaga en un sistema con componentes no deterministas y/o probabilísticas. Basados en esta técnica, presentaremos una nueva técnica para el cálculo de la probabilidad mínima.

La técnica se basa en reducir el problema de verificar la propiedad LTL a un problema de alcanzabilidad cuantitativo. Para ello la propiedad LTL se transforma en un autómata que describe la misma propiedad (en términos de co-lenguajes). Este autómata, a su vez, se sincroniza (se interseca) con el modelo bajo estudio (el cual contiene probabilidades) para obtener un nuevo autómata que sólo contiene la parte del comportamiento original que satisface la propiedad, es decir, sólo contiene las ejecuciones en las cuales la propiedad LTL es válida. En el autómata resultante de la intersección se busca un subconjunto de estados especiales, estos nos garantizan que los conjuntos de *ejecuciones* que satisfacen la propiedad con probabilidad mayor que 0 necesariamente deben lleguar a algún elemento de estos. Entonces el problema se ha reducido a calcular la probabilidad de alcanzar elementos en los subconjuntos especiales, es decir, se ha reducido a un problema de

alcanzabilidad. En la siguiente figura vemos un esquema del proceso:



Esta técnica ha sido implementada, en la cual, el problema de alcanzabilidad es resuelto por Rapture, un model checker conocido, que ha sido desarrollado para resolver este tipo de problemas. Se presentarán los resultados obtenidos en distintos casos de prueba.

1.4. Organización de la Tesis

El trabajo se encuentra organizado de la siguiente forma:

En el capítulo 2 se presentan conceptos básicos sobre *teoría de autómatas finitos para lenguajes infinitos* (definición, lenguaje, determinismo, representación gráfica). Nos enfocamos en autómatas de Buchi y en autómatas de Rabin. Además presentamos un algoritmo para generar un autómata de Rabin determinístico que acepta el mismo lenguaje que un autómata de Buchi no determinístico.

En el capítulo 3 se presentan la sintaxis y la semántica de la *lógica temporal lineal* (LTL). Conceptos básicos (lenguje de la fórmula, operadores auxiliares). En la útima sección del capítulo se presenta un algortimo para generar un automata de Buchi no determinístico que acepte el mismo lenguaje de

una fórmula LTL.

En el capítulo 4 se presentan los *sistemas no determinísticos probabilísticos* (SNP). Los sistemas bajo estudio se modelan a través de un conjunto de SNP puestos en paralelo o *sincronizados* (que resulta en un nuevo y más grande SNP). Definimos el modo de trabajar con probabilidades en ellos. Además presentamos ciertas propiedades útiles sobre los estados de los mismos.

En el capítulo 5 presentamos un método para sincronizar un SNP con una fórmula LTL. También se explican los algoritmos para el cálculo de la probabilidad máxima y mínima de que la propiedad bajo estudio se satisfaga.

En el capítulo 6 se realiza una breve explicación de la implementación realizada, de su arquitectura y de cómo se utiliza la misma.

En el capítulo 7 se presentan los resutados de correr la herramienta desarrollada sobre distintos casos de prueba. La herramienta se probó inicialmente con casos fáciles para testear el correcto funcionamiento y luego con ejemplos más interesantes.

En en capítulo 8 se presentan nuestras conclusiones y las posibles continuaciones a este trabajo.

El trabajo adjunta un CD, con la implementación del algoritmo que se desarrolló y la documentación del mismo. Además se incluyen las aplicaciones necesarias para la realización total del proceso.

Reconocimiento: Matias Lee agradece la beca del plan ConCiencia, otorgada por la Agencia Córdoba Ciencia, para la realización de este trabajo.

Capítulo 2

Autómatas finitos para lenguajes infinitos

En estStreetta sección presentaremos algunos conceptos sobre teoría de autómatas. Un autómata es un modelo matemático que representa un dispositivo de tamaño fijo, el cual utilizaremos para procesar entradas de tamaño infinito. Estas entradas son generalmente denominadas *palabras*.

Principalmente nos concentraremos en autómatas finitos sobre palabras de longitud infinita, también conocidos como ω-*autómatas*. Formalmente, un autómata se define como sigue:

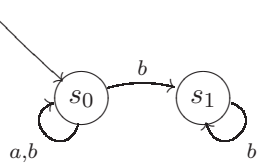
Definición 1. *Un* autómata finito *es una 5-upla* $\langle \Sigma, Q, \delta, q_0, C \rangle$ *tal que:*

1. Σ *es un* alfabeto finito.
2. Q *un conjunto finito de* estados.
3. $\delta : Q \times \Sigma \to 2^Q$ *una* función de transición parcial.
4. $q_0 \in Q$ *un* estado inicial.
5. C *un* criterio de aceptación.

Un autómata de dice *determinístico* si la función δ es tal que para todo $\sigma \in \Sigma$ y para todo $q \in Q$ vale que $|\delta(q, \sigma)| \leq 1$.

Notemos que un autómata puede representarse por medio de un grafo, en el cual, las transiciones poseen una etiqueta. En caso de que el autómata sea determinístico, de cada nodo a lo sumo saldrá una transición por cada elemento de Σ

Ejemplo 3.



Representa al siguiente autómata:

1. $\Sigma = \{a, b\}$
2. $Q = \{s_0, s_1\}$
3. $\delta(s_0, a) = \{s_0\}$
   $\delta(s_0, b) = \{s_0, s_1\}$
   $\delta(s_1, b) = \{s_1\}$
4. $q_0 = s_0$

Claramente el autómata no es determinístico, pues $|\delta(s_0, b)| = 2$.

El criterio de aceptación C aún no lo hemos definido ya que este determina la clase de autómata sobre el que estamos trabajando.

Definición 2. *Sea* $v \in \Sigma^\omega$ *una palabra infinita,* una ejecución de v *sobre un autómata finito* $A = < \Sigma, Q, \delta, q_0, C >$ *es un mapeo* $\rho : \mathrm{N}_0 \to Q$ *tal que:*

1. $\rho(0) = q_0$
2. $\rho(i + 1) = q$ *donde* $q \in \delta(\rho(i), v_i)$.

Una forma simple de representar el mapeo de ρ es mediante palabras de Q^ω, donde el i-esimo elemento se asocia a $\rho(i)$. Usaremos la notación ρ_i para representar $\rho(i)$.

Cabe recalcar que si el autómata no es determinístico diferentes ejecuciones pueden corresponder a una misma palabra, y que una ejecución puede corresponder a varias palabras, sin importar si el autómata es o no es determinístico.

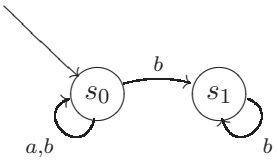
Definición 3. *Una palabra* v *es* aceptada *por el autómata* $A = < \Sigma, Q, \delta, q_0, C >$ *si existe una ejecución* ρ *de* v *satisface el criterio de aceptación* C.

Definición 4. *El* lenguaje del autómata A, $L(A) \subseteq \Sigma^\omega$, *es el conjunto de todas las palabras aceptadas por* A.

Definición 5. *Diremos que un autómata no determinístico* A *es* determinizable *si existe un autómata determinístico* A' *tal que* $L(A) = L(A')$

Definición 6. *Dada una ejecución* ρ *sobre el autómata* A, *inft(*ρ*) es el conjunto de estados que aparecen infinitamente en* ρ. *Más formalmente, inft(*ρ*)=* $\{s : \forall i \geq 0 : \exists j \geq i : s = \rho(j)\}$

Ejemplo 4.



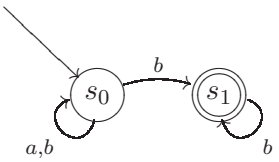
En el siguiente autómata, dada la palabra $v = b^\omega$, una posible ejecución ρ de v está dada por $\rho = s_0 s_1^\omega$. Pero notemos que ésta no es la única ejecución posible, ya que las ejecuciones de v incluyen a todas las ejecuciones de la forma $s_0^+ s_1^\omega$. Para todo ρ' de esta forma, $inft(\rho') = \{s_1\}$. Por último, notemos que otra posible ejecución de v es la de la forma s_0^ω, la cual es a su vez la única ejecución de a^ω.

2.1. Autómatas de Buchi sobre palabras infinitas

Definición 7. *Un autómata de Buchi (AB) sobre palabras infinitas es un autómata $\langle \Sigma, Q, \delta, q_0, B \rangle$ en el que el criterio de aceptación B está definido de la siguiente manera:*

- *Sea $B \subseteq Q$ el* conjunto de estado finales, *la palabra $v \in \Sigma^\omega$ es aceptada si existe una ejecución ρ de v tal que $inft(\rho) \cap B \neq \emptyset$.*

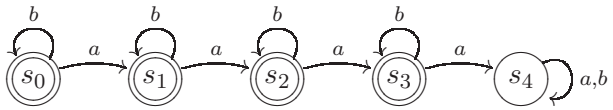
Ejemplo 5. *Tomemos el automáta del ejemplo 1 y definamos el criterio de aceptación como $B = \{s_1\}$.*



Luego tenemos un autómata de Buchi que acepta palabras de la forma $(a+b)^*(b)^\omega$. Es decir, todas las palabras con una cantidad finitas de a. Notemos que la palabra $v = a^\omega$ no es aceptada pues la única ejecución posible de v es $\rho = s_0^\omega$, luego $inft(\omega) \cap B = \emptyset$.

Es importante remarcar que el autómata de este último ejemplo no es determinizable utilizando el criterio de aceptacion de Buchi. El problema se genera al momento de tratar de capturar, usando transiciones determinísticas, el hecho de que se han consumido todas las letras a de la palabra, sin saber previamente cuantas a tiene la misma. En contraposición a esto, en caso de tener una cota N para la cantidad de a, si se podría generar un AB determinístico que acepte este lenguaje. A continuación un AB determinístico con $N = 3$.

Ejemplo 6. *Un AB determinístico que acepta palabras con una cantidad de a's menor o igual a 3.*



2.2. Autómatas de Buchi generalizados

Una variante de los autómatas de Buchi, son *los autómatas de Buchi generalizados.*

Definición 8. *Un* autómata de Buchi generalizado*(ABG) es un autómata* $\langle \Sigma, Q, \delta, q_0, BG \rangle$ *donde el* criterio de aceptación BG *está definido de la siguiente manera:*

- *Sea* $BG = \{B_1, B_2, ..., B_n\}$*, donde* $B_1, B_2, ..., B_n \in 2^Q$*. Luego, la palabra* $v \in \Sigma^\omega$ *es aceptada si existe una ejecución* ρ *de* v *tal que* $inft(\rho) \cap B_i \neq \emptyset$ *para* $i = 1, ..., n$*.*

Si bien el criterio de aceptación es distinto, el conjunto de todos los lenguajes que se pueden generar es el mismo. Notar que un autómata de Buchi es, en particular, un autómata de Buchi generalizado donde $|BG| = 1$. Para obtener un autómata de Buchi a partir de un autómata de Buchi generalizado, el cual acepte el mismo lenguaje, se aplica la siguiente construcción:

Dado un autómata de Buchi generalizado $A = \langle \Sigma, Q, \delta, q_0, \{B_1, B_2, ..., B_n\} \rangle$, definimos el autómata de Buchi $A' = \langle \Sigma, Q \times \{0, ..., n\}, \delta', q_0 \times \{0\}, Q \times \{n\} \rangle$.

La relación de transicción δ' esta construida de forma tal que $(\langle q, x \rangle, a, \langle q', y \rangle) \in \delta'$ si $(q, a, q') \in \delta$ y x e y cumplen las siguientes reglas:

- Si $q' \in B_i$ y $x = i - 1$ entonces $y = i$.
- Si $x = n$ entonces $y = 0$.
- De otra forma, $x = y$.

El esquema de prueba para la construcción es el siguiente: Si $v \in L(A)$, luego existe una ejecución ρ de v tal que $inft(\rho) \cap B_i \neq \emptyset$ para $1 \leq i \leq n$. Esto implica que existen $j_1 < \cdots < j_n$ tales que $\rho_{j_i} \in B_i$. Por como están definidas las transiciones podemos demostrar que existe una ejecucuión ρ' de v en A' tal que ρ'_{j_n} es de la forma (q, n) y ρ'_{j_n+1}, de la forma $(q', 0)$ donde $q, q' \in Q$. Usando este razonamiento y el hecho de que siempre existen $j_{tn+1} < \cdots < j_{(t+1)n}$ tales que $\rho_{j_{tn+i}} \in B_i$ para $t \geq 0$ y $1 \leq i \leq n$, podemos ver que $inft(\rho') \cap (Q \times n) \neq \emptyset$ para alguna ejecución ρ' de v en A'.

Si $v \in L(A')$ luego existe una ejecución ρ' de v tal que $inft(\rho') \cap (Q \times n) \neq \emptyset$. Para que la ejecución pase infinitas veces por algún estado (q, n) tiene que pasar infinitas veces por estados (q, i) con $1 \leq i < n$ y $q \in B_i$. Usando este hecho y como están definidas las transiciones se puede demostrar que existe una ejecución ρ de v en A tal que $inft(\rho) \cap B_i \neq \emptyset$ para $1 \leq i \leq n$.

2.3. Autómatas de Rabin determinísticos

A continuación presentaremos los autómatas de Rabin. Principalmente trabajaremos con autómatas determinísticos.

Definición 9. *Un autómata de Rabin determinístico (AR) es un autómata determístico $\langle \Sigma, Q, \delta, q_0, R \rangle$, donde el criterio de aceptación R está definido de la siguiente manera:*

- *Sean $E_0, E_1..E_n, F_0, F_1..F_n \subseteq Q$, luego $R = \{(E_0, F_0), .., (E_n, F_n)\}$. La palabra $v \in \Sigma^\omega$ es aceptada si la ejecución ρ de v es tal que $inft(\rho) \subseteq E_i$ e $inft(\rho) \cap F_i \neq \emptyset$ para algún $i \in \{0, 1, .., n\}$.*

Luego este criterio informalmente se entiende como que una ejecución será aceptada si ésta queda en algún momento visitando los estados "buenos" (E_i), pero sin dejar de pasar infinitamente por alguno de los estados "necesarios" (F_i).

Notemos que si es cierto que $F_i - E_i \neq \emptyset$, cambiar F_i por $F_i' = F_i - (F_i - E_i)$, no genera un autómata nuevo con un lenguaje distinto al original. Luego podemos suponer que $F_i \subseteq E_i$.

Notemos también que el criterio de aceptación de Rabin puede también expresarse de la siguiente manera:

- Sean $E_0^c, E_1^c..E_n^c, F_0, F_1..F_n \subseteq Q$, luego $R' = \{(E_0^c, F_0), .., (E_n^c, F_n)\}$. La palabra $v \in \Sigma^\omega$ es aceptada si la ejecución ρ de v es tal que $inft(\rho) \cap E_i^c = \emptyset$ e $inft(\rho) \cap N_i \neq \emptyset$ para algún $i \in \{0, 1, .., n\}$

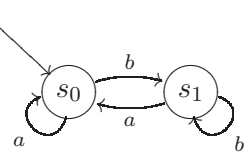
Informalmente el criterio de aceptación se puede entender como que una ejecución será aceptada si en algún momento deja de visitar los estados "malos" (E_i^c) pero siempre pasa por alguno de los estados "necesarios" (F_i).

Ejemplo 7. *Dado el siguiente autómata de Rabin:*

1. $\Sigma = \{a, b\}$
2. $Q = \{s_0, s_1\}$
3. $\delta = \{(s_0, a, s_0), (s_0, b, s_1), (s_1, b, s_1), (s_1, a, s_0)\}$

4. $q_0 = s_0$

5. $R = \{(\{s_1\}, \{s_1\})\}$



Notemos que este autómata, al igual que el ejemplo 3, solo acepta palabras del lenguaje $(a + b)^*b^\omega$*, pero con la variante de que este autómata sí es determinístico.*

2.4. Transformación de AB a AR

Como hemos visto en los ejemplos anteriores existen autómatas de Buchi que no son determinizables. Pero también hemos visto, en el último ejemplo, que un lenguaje no determinizable generado por un AB puede ser generado por AR determinístico.

En esta sección presentaremos una transformación para construir un AR determinístico que acepta el mismo lenguaje que un AB no determinístico. Esta transformación se debe a Safra [Saf89], aunque nosotros presentaremos la variación dada en [Lőd05]. En esencia son la misma tranformación, pero la segunda esta definida de una forma más simple de entender.

La idea detrás de la construcción de Safra es llevar un registro de todos los posibles prefijos de ejecuciones que se generan en el autómata de Buchi, a medida que se consumen los elementos de Σ. Pero esta información es guardada de tal forma que es posible distinguir segmentos de la ejecución que sí o sí pasan por estados de B del AB.

Con esto en mente, los nodos del autómatas de Rabin estarán compuestos por *arboles de Safra*. A continuación una definición formal de ellos.

Definición 10. *Un* árbol de Safra *sobre un conjunto finito no vacio* Q *(*$|Q| = n$*) es un árbol ordenado finito* $\langle N, h, <, et, m\rangle$*, donde:*

1. $N \subseteq \{1, 2, ..., 2n\}$*, un conjunto de nodos identificados con elementos de* $\{1, 2, ..., 2n\}$*.*

2. $h : N \to 2^N$ *una función de descendencia, i.e.* $h(n)$ *son los hijos de* n*.*

3. $< \ \subseteq N \times N$*, una relación de orden que está definida para* $n_1, n_2 \in h(n)$ *para todo* $n \in N$*.* (n_1, n_2) *nos dice que* n_1 *es más joven que* n_2*.*

4. $m : N \to \{\mathsf{True}, \mathsf{False}\}$, *una función que marca o desmarca los nodos. Diremos que el nodo n está marcado si* $m(n) = \mathsf{True}$, *si no diremos que está desmarcado.*

5. $et : N \to 2^Q - \{\emptyset\}$, *una función que etiqueta los nodos con elemetos de* $2^Q - \{\emptyset\}$ *y cumple con las siguientes propiedades:*

   a) $et(n) \supseteq \bigcup_{n' \in h(n)} et(n')$, *i.e. la etiqueta de todo nodo incluye a la unión de las etiquetas de sus hijos.*

   b) $et(n_1) \cap et(n_2) = \emptyset$ *para todo* $n_1, n_2 \in h(n)$ *tal que* $n_1 \neq n_2$, *i.e dos nodos con el mismo padre poseen etiquetas disjuntas.*

Luego, la transformación de Safra es la siguiente:
Dado un AB, $A = \langle \Sigma, Q, q_0, \delta, B \rangle$, definiremos el AR, $A' = \langle \Sigma, Q', q_0', \delta', R \rangle$ de la siguiente forma:

- Q' = El conjunto de todos los arboles de Safra sobre Q.
- q_0' = El arbol de Safra que consiste del nodo 1, etiquetado con $\{q_0\}$.
- Los elementos de la relación de transición $(q', a, p') \in \delta'$ están definidos de forma tal que $q' \in Q'$, $a \in \Sigma$ y p' es el resultado de realizar los siguientes pasos:
  1. $p' := q'$
  2. Desmarcar todos los nodos marcados de p'
  3. Para todo nodo de p' con etiqueta S tal que $S \cap B \neq \emptyset$, crear un hijo con label $S \cap B$ y nombrarlo con un elemento de $\{1, 2, ..., 2n\}$ que no se esté utilizando. En caso de tener hermanos, este será el más joven de ellos,
  4. Remplazar toda etiqueta S de cada nodo del arbol p' por $\bigcup_{q \in S} \delta(q, a)$.
  5. Para todo par de nodos de p' con el mismo padre tal que $q \in Q$ pertenezca a las etiquetas de ambos, eliminar, del hijo más joven y de todos sus descendientes, el elemento q.
  6. Eliminar todos los nodos con etiquetas vacias.
  7. Para todo nodo de p', el cual posea una etiqueta que es igual a la unión de las etiquetas de sus hijos, eliminar todos los descendientes del nodo y marcarlo.
- El criterio de aceptación de $R = \{(E_1, F_1), ..., (E_{2n}, F_{2n})\}$ esta definido para $i \in \{1, ..., 2n\}$ por:
  E_i = Conj. de todos los arboles de Safra que contengan el nodo i.
  F_i = Conj. de todos los arboles de Safra con el nodo i marcado.

Demostremos que la tranformación es correcta. Para realizar esto primero enunciaremos dos lemas auxiliares.

Lema 1. *Un arbol de Safra sobre un conjunto Q con $|Q| = n$ tiene a lo sumo n nodos.*

Lema 2 (Lema de Kőnig). *Todo arbol con infinitos nodos, los cuales contienen una cantidad finita de hijos, contiene un camino infinito.*

Teorema 1. *Dado un autómata de Buchi, $A = \langle Sigma, Q, q_0, \delta, B\rangle$, el autómata de Rabin $A' = \langle \Sigma, Q', q'_0, \delta', R\rangle$ generado mediante la transformación de Safra desde A genera el mismo lenguaje.*

Demostración. $L(A) \subseteq L(A')$: Sea $v \in L(A)$ y sea ρ una ejecución de v sobre el autómata A que cumple con el criterio de aceptación. Veamos como se comporta la ejecución de v en el autómata A'. Supongamos ρ' la ejecución de v sobre A'. Luego ρ' es una sucesión de arboles de Safra, notemos que las raíces de estos nunca van a ser eliminadas, ya que la raíz del arbol ρ'_i contendrá al menos el elemento ρ_i (paso 4). Si la raíz es marcada infinita veces, luego la palabra es aceptada. Si esto no es así, sea j tal que ρ'_j es la última vez que la raíz es marcada. Como ρ es una ejecución de una palabra aceptada, podemos tomar el menor $k \geq j$ tal que $\rho_k \in F$. Luego al momento de crear el nodo ρ'_{k+1} este tendrá un nuevo hijo con al menos el elemento ρ_k (paso 3). Este luego será remplazado por ρ_{k+1} (paso 4). Realizando un razonamiento similar al usado para demostrar que las raíces no son borradas y el hecho que la raíz no es vuelta a marcar, podemos demostrar que este nodo tampoco será eliminado. Si este nodo es marcado infinitamente luego la palabra es aceptada. En caso de que no, podemos repetir el razonamiento y demostrar que este nodo tendrá un hijo que nunca es borrado. Notemos que por el lema 1, la profundidad de los arboles de Safra esta acotada por el cardinal del conjunto sobre el cual se genera, luego esto nos asegura que encontraremos un nodo que es marcado infinitamente y nunca es removido.

$L(A') \subseteq L(A)$: Sea $v \in L(A')$ y sea ρ' una ejecución de v sobre el autómata A'. Luego existe un nodo t en todos los arboles de ρ' tal que a partir de un punto, éste siempre aparece y es marcado una cantidad infinita de veces. Sea x la posición tal que el nodo (arbol) t aparece en ρ'_y para todo $y \geq x$. Ahora definamos $i_1, i_2, ... \geq x$ las posiciones donde t es marcado y $Q_1, Q_2, ...$ las etiquetas del nodo en esas posiciónes. Además definamos $i_0 = 0$ y $Q_0 = \{q_0\}$. Por la definición de δ', para todo $j \in N$ y $q \in Q_{j+1}$ existe un $p \in Q_j$ tal que $p \rightarrow_\delta q$ consumiendo la subpalabra $v_{[i_j, i_{j+1})}$ y pasa por al menos por un estado de F. Ahora definamos un arbol con los nodos de la forma (q, j) donde $q \in Q_j$ y $j \in N$. Los padres del nodo $(q, j+1)$ es cualquier nodo (p, j) tal que $p \rightarrow_\delta q$ consumiendo la subpalabra $v_{[i_j, i_{j+1})}$. Esto nos define de forma correcta un arbol con infinitos nodos. Como todo

nodo tiene a lo sumo n hijos, existe un camino infinito en el arbol por el lema de Kőnig. Luego, la construcción de este arbol demuestra que existe una ejeccución ρ de v que pasa infinitamento por los estados finales. □

Ejemplo 8. *Aplicación de la transformación de Safra al autómata del ejemplo 5. Representaremos los nodos con conjuntos y su nombre estará indicado en el superindice. El simbolo ! representará que el nodo está marcado.*

- $q'_0 = \{q_0\}^1$
- *Luego de realizar la transformación δ' queda definida como:*

$\delta'(q'_0, a) = q'_0 = \quad \{q_0\}^1$

$\delta'(q'_0, b) = q'_1 = \quad \{q_0, q_1\}^1$

$\delta'(q'_1, a) = q'_0 = \quad \{q_0\}^1$

$\delta'(q'_1, b) = q'_2 = \quad \{s_0, s_1\}^1 \downarrow \{s_1\}^2$

$\delta'(q'_2, a) = q'_0 = \quad \{q_0\}^1$

$\delta'(q'_2, b) = q'_3 = \quad \{s_0, s_1\}^1 \downarrow \{s_1\}^2!$

$\delta'(q'_3, a) = q'_0 = \quad \{q_0\}^1$

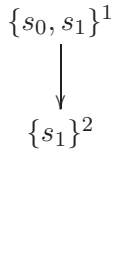
$\delta'(q'_3, b) = q'_3 = \quad \{s_0, s_1\}^1 \downarrow \{s_1\}^2!$

- *El único nodo marcado es el 2 en q'_3 y éste nodo no aparace en q'_0 y q'_1. Luego el criterio de aceptación es $R = \{(\{q'_2, q'_3\}, \{q'_3\})\}$.*

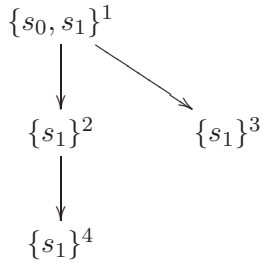
Realicemos a continuación en detalle los pasos para definir $\delta'(q'_2, a)$ y $\delta'(q'_2, b)$ por ser éstos los más interesantes. (nota: cuando haya más de un paso representado en un solo gráfico, el paso que produce los cambios es el primero.)

Empecemos con $\delta'(q_2', a)$*:*

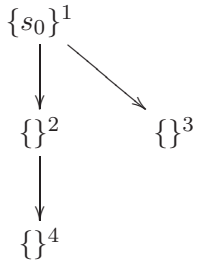
Paso 1 y 2:



Paso 3:



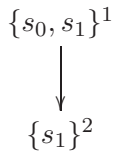
Pasos 4 y 5:



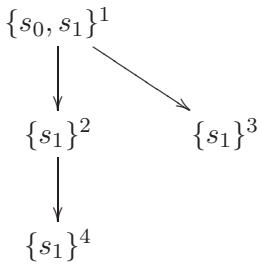
Pasos 6 y 7:

Ahora veamos en detalle $\delta'(q_2', b)$*:*

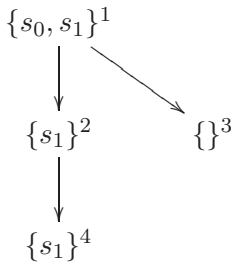
Pasos 1 y 2:



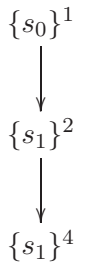
Pasos 3 y 4:



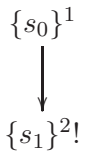
Paso 5:



Paso 6:



Paso 7:



Capítulo 3

LTL y su relación con los autómatas de Buchi

En esta sección presentaremos conceptos sobre lógica temporal lineal (LTL). LTL se utiliza para especificar propiedades sobre sistemas reactivos, en la cual el tiempo se interpreta en una forma discreta y se evaluan proposiciones lógicas sobre los estados del sistema.
La lógica temporal, a diferencia de la lógica proposicional, utiliza operadores *modales*.

Los elementos básicos de las fórmulas LTL son las *proposiciones atómicas*, es decir declaraciones que no pueden ser subdivididas. Por ejemplo: *x es menor que 100*, *2 igual 3*, *llueve*. Al conjunto de proposiciones atómicas con el que trabajamos lo declararemos AP.

Notemos que las proposiciones se hacen *verdaderas* y *falsas* dependiendo del estado en el que se encuentre el sistema. Por ejemplo, *"la maquina está sirviendo cafe"* es una proposición atómica que será *verdadera* cuando el sistema este sirviendo cafe y será *falsa* cuando no lo este haciendo. Notemos también que las proposiciones atómicas serán la base con la que expresaremos las propiedades a verificar sobre el sistema.

Entre los operadores modales, que utiliza LTL, podemos encontrar por ejemplo: G , F . Estos se utilizan para expresar propiedades a través del tiempo. El operador G se utiliza para especificar propiedades que se cumplen siempre a lo largo del tiempo, mientras que el operador F se utiliza para especificar propiedades que en el futuro valdrán. Por ejemplo *"F llueve"* es cierto pues sabemos que en algún momento en el futuro lloverá y *"G llueve"* es falso pues no siempre llueve. Estos operadores también pueden utilizarse en forma combinada para expresar propiedades más ricas, por ejemplo *"G (F llueve)"* expresa la propiedad de que no importa en que momento en el futuro nos encontremos siempre sabremos que existe un momento más adelante en el que lloverá.

En este capítulo definiremos la sintaxis de las fórmulas LTL y su semántica, definiremos operadores auxiliares para simplificar la notación y por último presentaremos un algoritmo para generar un AB determinístico a partir de una fórmula LTL ψ. Este autómata tendrá como alfabeto asociado $\Sigma = 2^{AP}$ y la palabra $\rho \in \Sigma^{\omega}$ representará que el momento i-esimo en el sistema son validadas los proposiciones de ρ_i. El lenguaje de este autómata será el conjunto de palabras que satisfacen la propiedad ψ.

3.1. Sintaxis de LTL

Definimos el conjunto de fórmulas de LTL de la siguiente manera:

Definición 11. *Dado un conjunto de proposiciones atómicas AP, las fórmulas LTL satisfacen las siguientes reglas:*

1. *p es una fórmula para todo $p \in AP$.*
2. *Si ϕ es una fórmula, luego $\neg\phi$ y $\mathsf{X}\,\phi$ son fórmulas.*
3. *Si ϕ y ψ son fórmulas, luego $\phi \vee \psi$ y $\phi\ \mathsf{U}\ \psi$ son fórmulas.*

Cualquier otra cosa no es fórmula.

LTL extiende la lógica proposicional con los operadores temporales X (llamado *neXt*) y U (llamado *Until*). El significado intuitivo de estos es el siguiente:

- $\mathsf{X}\,\phi$ es cierto en este momento, si en el siguiente momento ϕ lo es.
- $\phi\ \mathsf{U}\ \psi$ es cierto en este momento, si ϕ es cierto desde este momento hasta el momento previo en el que vale ψ

3.2. Semántica de LTL

Definición 12. *Dado un conjunto de proposiciones atómicas AP, un* camino *ρ será una secuencia infinita de elementos de 2^{AP}, es decir $\rho = \rho_0\rho_1\rho_2...$ donde $\rho_i \in 2^{AP} \quad i = 0, 1, 2, ...$*

Intuitivamente, los *caminos* representan el conjunto de proposiciones atómicas que son verdaderas en el sistema, en cada momento, a lo largo del tiempo. ρ_i denota *el elemento i-esimo* del camino y el primer elemento del camino es ρ_0. (ρ, i) es el sufijo i-esimo de ρ, es decir $(\rho_0\rho_1 \cdots, i) = \rho_i\rho_{i+1}\rho_{i+2} \cdots$

Ahora definiremos la semántica de las fórmulas LTL como una relación de satisfacción entre un *camino* ρ y una fórmula ϕ. Usaremos el operador

$\models$ para expresar esta relación. Luego $\rho \models \phi$ será equivalente a decir que la fórmula ϕ es válida en el camino ρ. Más formalmente:

Definición 13. *Sea $p \in AP$, ρ un camino y ϕ, ψ fórmulas LTL. La relación de satisfacción $\models$ esta definida por:*

1. $(\rho, i) \models p$ *sii* $p \in \rho_i$
2. $(\rho, i) \models \neg\phi$ *sii no se satisface* $(\rho, i) \models \phi$
3. $(\rho, i) \models \phi \vee \psi$ *sii se satisface* $(\rho, i) \models \phi$ *ó* $(\rho, i) \models \psi$
4. $(\rho, i) \models \mathsf{X}\,\phi$ *sii se satisface* $(\rho, i+1) \models \phi$
5. $(\rho, i) \models \phi\ \mathsf{U}\ \psi$ *sii* $\exists j \geq i : (\rho, j) \models \psi \wedge \forall k : i \leq k < j : (\rho, k) \models \phi$

Definiremos como el lenguaje de la fórmula ϕ *al conjunto:*

$$L(\phi) = \{\rho \in (2^{AP})^\omega \mid (\rho, 0) \models \phi\}.$$

Ejemplo 9. *Dado $\phi = \mathsf{X}\,p$ con $p \in AP$, el $L(\phi) = \{\rho \in (2^{AP})^\omega \mid p \in \rho_1\}$.*

3.3. Operadores temporales auxiliares

Para facilitar la especificación de propiedades relevantes, introduciremos nuevos operadores modales. Estos operadores están definidos en terminos de los operadores ya presentados, por esta razón no agregarán expresividad al lenguaje de las fórmulas LTL. Los nuevos operadores son F (*Finally*), G (*Globally*) y R (*Release*) y están definidos de la siguiente manera:

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

Con $\mathsf{F}\ \phi$ estamos expresando que en algún momento en el futuro la proposición ϕ será valida, i.e. *Finalmente valdrá* ϕ. Con $\mathsf{G}\ \phi$ estamos expresando que en ningún momento en el futuro vale $\neg\phi$, lo que es equivalente a decir que en todo momento vale ϕ, i.e. *Globalmente vale* ϕ. Por último, el operador R, el cual es el operador lógico dual del U. Este requiere que la proposición ψ sea válida hasta que el mismo momento el en que ocurra ϕ, pero sin exigir que en algún momento sea válida ϕ.

3.4. Transformación de una fórmula LTL a un autómata de Buchi

En esta sección presentaremos el algoritmo para generar un autómata de Buchi que acepta el mismo legunaje de una fórmula LTL. Este fue presentado por Gerth, Peled, Vardi y Wolper.

El algoritmo se divide en 4 partes:

1. La fórmula LTL es llevada a su *forma normal negativa*
2. Se genera un grafo auxiliar G a partir de la fórmula LTL.
3. A partir de G se genera un autómata de Buchi generalizado BG.
4. Se genera a partir de BG el autómata de Buchi.

En la *forma normal negativa* la fórmula LTL solo utiliza operadores básicos y la negación esta aplicada solo a las proposiciones atómicas. Para realizar esta normalización, remplazamos los operadores auxliares (G , F , R) por su definición, por ejemplo: $\mathsf{F}\ \psi$ es remplazado por $\mathsf{True}\ \mathsf{U}\ \psi$. Para lograr negar sólo las proposiciones atómicas utilizamos las siguientes igualdades:

- $\neg(\phi\ \mathsf{U}\ \psi) = (\neg\phi)\ \mathsf{R}\ (\neg\psi)$.
- $\neg(\phi\ \mathsf{R}\ \psi) = (\neg\phi)\ \mathsf{U}\ (\neg\psi)$.
- $\neg(\mathsf{X}\ \psi) = \mathsf{X}\ (\neg\psi)$.

Para la generación del grafo auxiliar utilizaremos la función *crearGrafo()*. A continuación explicaremos en detalle el algoritmo que utiliza:

La estructura básica que usa el algoritmo se llama *nodo*. Los nodos del grafo serán representados por el conjunto de nodos. Un nodo n contiene la siguiente información:

nodo = [*IN*: id_Nodo;
Entrantes : conjunto id_Nodo;
Procesadas, Siguientes: conjunto FórmulaLTL;]

IN es el identificador del nodo.

Entrantes es un conjunto con los nodos predecesores. Cada nodo n' del conjunto representa una arista desde n' a n.

Procesadas, Siguientes: Cada una de estos conjuntos esta formado por subfórmulas de la fórmula de la cual estamos construyendo el grafo. La construcción de éstos se realiza de tal forma que los posibles prefijos de secuencias de estados que alcancen el nodo, satisfacen todas las fórmulas de *Procesadas*

y el siguiente nodo alcanzable es de tal forma que satisface todas las fórmulas de *Siguientes*.

A continuación utilizaremos un *pseudo-codigo* para explicar el algoritmo. En él algoritmo utilizaremos la variable global *nodos*, en la cual guardaremos el conjunto de nodos generados. También utilizaremos la función *gen_ id()* la cual genera un identificador nuevo para cada nodo. La función principal, como ya fue mencionado, será *crearGrafo()* la cual toma como parametro una fórmula LTL. El núcleo de la función principal será la función recursiva:

expand(ent: lista identificadorDelNodo,proc, nuevas, sig : lista Fórmula)

la cual tiene como parametros los datos necesarios para crear un nuevo nodo, i.e. los nodos *Entrantes* y los conjuntos *Procesadas, Siguientes* del nodo a crear. Además toma un tercer conjunto, *Nuevas*, este es el conjunto de las fórmulas que aún no han sido procesadas. Esta función será la encargada de crear los nodos siguientes a los nodos listados en la variables *ent*.

La función *crearGrafo()* inicializa el conjunto de *nodos* a vacio. Luego utiliza un nodo especial *init*, este será el nodo inicial y se agrega al conjunto de nodos al finalizar la función *expand()*.

```
funcion crearGrafo(φ : formulaLTL){
        nodos := ∅
        expand({init}, ∅, { φ },∅);
        nodos := nodes ∪ init
}
```

El llamado a *expand()*, empezará a generar los nodos siguientes a *init*. El psudo-codigo de *expand()* es el siguiente:

```
funcion expand(ent, proc, nuevas, sig) {
        if (nuevas = ∅) {
                if( ∃n ∈ Nodos : n.Procesadas = proc & n.Siguientes = sig) {
                        n.entrantes := n.entrantes ∪ ent;
                        return;
                } else {
                        new node nn;
                        nn.IN := gen_ID();
                        nn.Procesadas := proc;
                        nn.Siguiente := sig;
                        nodos := nodos ∪ {nn};
                        expand({nn.IN}, ∅, nn.Siguintes, ∅);
                }
```

```
        } else {
                expand_form(ent, proc, nuevas, sig);
        }
}
```

Si el conjunto *nuevas* es vacio es el momento de generar un nuevo nodo, el algoritmo verifica si el este ya existe, si es el caso, actualiza los nodos entrantes, caso contrario crea un nuevo nodo y continua generando los nodos siguientes a éste, por esta razón realiza un nuevo *expand()*. Notemos que para realizar el nuevo *expand()* toma como conjunto de fórmulas *nuevas* el conjunto de fórmulas *siguientes*, lo cual refleja el hecho de que los nodos siguientes al último creado deben satisfacer las fórmulas que este requiere que sus sucesores cumplan.

Otra cosa interesante para remarcar es que el algoritmo utiliza los conjuntos *Procesadas* y *Siguientes* para verificar la existencia del nodo y no su identificador *IN*. Esto se debe a que lo que realmente identifica a un nodo son estos conjuntos, la variable *IN* se utiliza simplemente para representar de forma simple el conjunto de predecesores. En caso de que el nodo exista no se realiza un *expand* porque este ya fue realizado al momento de crear el nodo.

```
funcion expand_form(ent, proc, nuevas, sig) {
        Sea η ∈ nuevas;
        nuevas := nuevas − {η};
        if (η ∈ proc) {
                expand(ent, proc, nuevas, sig);
        }else{
                if(η ∈ AP){
                        expand_prop(η, ent, proc, nuevas, sig);
                }else if(η = μ ∨ ψ) {
                        expand_Or(η, μ, ψ, ent, proc, nuevas, sig);
                }else if(η = μ ∧ ψ) {
                        expand_And(η, μ, ψ, ent, proc, nuevas, sig);
                }else if(η = X μ){
                        expand_Next(η, μ, ent, proc, nuevas, sig);
                }else if(η = μ U ψ){
                        expand_Until(η, μ, ψ, ent, proc, nuevas, sig);
                }else if(η = μ R ψ){
                        expand_Release(η, μ, ψ, ent, proc, nuevas, sig);
                }
        }
}
```

En la función *expand_form* si la fórmula η está en *proc*, significa que ésta ya fue procesada, por esta razón se vuelve a empezar la expansión, pero sin la fórmula η en *nuevas*. Esto puede ocurrir ya que una misma proposición atómica puede ocurrir varias veces en una misma fórmula. En caso de que no sea ese el caso, realizamos diferentes tipos de expansiones en función de la forma de la fórmula. Las expansiones son las siguientes:

```
funcion expand_prop(η, ent, proc, nuevas, sig){
        if((η = False) ∨ ((¬η) ∈ proc)) {
                return;
        }else{
                proc := proc ∪ {η};
                expand(ent, proc, nuevas, sig));
        }
}
```

La función *expand_ prop()* verifica si la proposición η no genera incoherencias con las fórmulas procesadas, si es asi, ésta se descarta. En caso de que no sea así, la fórmula η es agregada al conjunto de las fórmulas procesadas y continuamos la expansión. Notemos que con incluir η al conjunto de fórmulas procesadas el algoritmo refleja el hecho de que el nodo nuevo que se va a crear satisface esta proposición atómica.

```
/* η = μ ∨ ψ */
funcion expand_Or(η, μ, ψ, ent, proc, nuevas, sig){
        proc' := proc ∪ {η} ;

        nuevas' := nuevas ∪ {μ};
        expand(ent, proc', nuevas', sig);

        nuevas' := nuevas ∪ {ψ};
        expand(ent, proc', nuevas', sig);
}
```

Si recordamos la definición formal de la semántica de la disyunción, ésta dice que debe existir un camino que satisfaga una de las dos fórmulas. Esto esta reflejado con los dos llamados a la función *expand()*, notemos que ambas llamadas a *expand()* tienen el mismo conjunto de nodos entrantes y que el conjunto *nuevas* es cambiados para reflejar que en los nuevos nodos a crear deben satisfacer cada uno, una de las fórmulas que componen la disyunción.

```
/* η = μ ∧ ψ */
```

```
funcion expand_And(η, μ, ψ, ent, proc, nuevas, sig){
        proc' := proc ∪ {η};
        nuevas' := nuevas ∪ {μ, ψ};
        expand(ent, proc', nuevas', sig);
}
```

El caso de *expand_ And()* es mucho más sencillo, ya que si el nuevo nodo debe satisfacer la conjunción de dos fórmulas es equivalente a pedir que las dos fórmulas sean validos. Luego el algoritmo solo agrega este par de fórmulas al conjunto de fórmulas *nuevas* y vuelve a empezar las expansión.

```
/* η = X μ */
funcion expand_Next(η, μ, ent, proc, nuevas, sig) {
        proc' := proc ∪ {η};
        sig' := sig ∪{μ};
        expand(ent, proc', nuevas, sig');
}
```

Notemos que en caso de tener un X μ, el algoritmo tiene que asegurarse de que los nodos siguientes al que estamos creando satisfagan μ, notenemos que esto se logra con incluir μ en el conjunto *sig'* ya que este será el conjunto de fórmulas que deberán satisfacer los nodos siguientes. Esto se refleja con claridad en el llamado a la función *expand()* que se realiza dentro de la función *expand()*. (Los problemas de usar la recursividad :D)

```
/* η = μ U ψ */
funcion expand_Until(η, μ, ψ, ent, proc, nuevas, sig){
        proc' := proc ∪ {η};
        nuevas':= nuevas ∪ {μ};
        sig' := sig ∪ {η};
        expand(ent, proc', nuevas', sig');

        proc' := proc ∪ {η};
        nuevas':= nuevas ∪ {ψ};
        sig' := sig;
        expand(ent, proc', nuevas', sig');
}
```

El código de esta función se debe a que μ U $\psi \equiv \psi \vee (\mu \wedge$ X $(\mu$ U $\psi))$, luego si se debe crear un estado que satisfaga μ U ψ es equivalente a crear dos estados donde uno satisface ψ y el otro satisface $(\mu \wedge$X $(\mu$ U $\psi))$. En este último tenemos en cuenta lo realizado en la función *expand_ Next()*.

```
/* η = μ R ψ */
funcion expand_Release(η, μ, ψ, ent, proc, nuevas, sig){
        proc' := proc ∪ {η};
        nuevas':= nuevas ∪ {μ, ψ};
        sig' := sig ∪ {η};
        expand(ent, proc', nuevas', sig');

        proc' := proc ∪ {η};
        nuevas':= nuevas ∪ {ψ};
        sig' := sig;
        expand(ent, proc', nuevas', sig');
}
```

El código de *expand_ Release()* es similar al de *expand_ Until()*, esto se debe a que $\mu \; \mathsf{R} \; \psi \equiv \psi \wedge (\mu \vee (\mathsf{X} \; (\mu \; \mathsf{R} \; \psi)))$, distribuyendo el $\wedge$ con el $\vee$ obtenemos $(\psi \wedge \mu) \vee (\psi \wedge (\mathsf{X} \; (\mu \; \mathsf{R} \; \psi)))$.

Luego de que se termina de construir el grafo, el autómata de Buchi generalizado se obtiene de la siguiente forma:

- El alfabeto es $\Sigma = 2^{AP}$. Donde dados $\sigma \in \Sigma$ y $p \in AP$, $p \equiv \mathsf{True}$ si y solo si $p \in \sigma$.
- El conjunto de nodos Q esta compuesto por los elementos del conjunto *nodos* generado por el algoritmo.
- $(n, \sigma, m) \in \delta$ si y solo si $n \in m.Entrantes$ y σ satisface la conjunción de las proposiciones negadas y no negadas de $m.Procesadas$.
- El estado inicial es $q_0 = init$. No hay nodos que lleguen a éste.
- Los conjuntos de estados de aceptación BG son generados de la siguiente forma: por cada subfórmulas de la forma $\mu \; \mathsf{U} \; \psi$ generamos un conjunto B_i que está compuesto por todos los nodos n tal que satisfacen $\psi \in n.Procesadas$ ó $(\mu \; \mathsf{U} \; \psi) \notin n.Procesadas$.

Por último lo único que falta es transformar el autómata de Buchi generalizado en un autómata de Buchi, para esto utilizamos la transformación presentada en la sección de autómatas de Buchi generalizados.

Capítulo 4

Sistemas no determinísticos probabilísticos

Los sistemas reactivos pueden ser modelados por un *sistema no determinístico probabilístico* (SNP). Los SNP son similarares a los *procesos de decisión de Markov* con una cantidad finita de estados. Para cada estado del preceso de decisión de Markov se puede elegir, de forma no determinística, una acción de un conjunto de acciones y el siguiente estado es elegido determinísticamente, mediante una distribución de probabilidades asociada a la acción elegida.

En este capítulo definiremos formalmente un SNP y las probabilidades máximas y mínimas de un conjunto de ejecuciones sobre estos. Demostraremos también propiedades de algunos subconjuntos especiales del SNP, éstas serán útiles para el cálculo de las probabilidades ya mencionadas.

Un SNP se define formalmente de la siguiente forma:

Definición 14 (SNP). *Un* SNP *es una 6-upla* $\langle AP, S, Accs, k, p, s_{in} \rangle$ *tal que:*

1. AP *es un* conjunto de proposiciones atómicas.
2. S *es un* conjunto finito de estados. *Todo estado* $s \in S$ *le asigna un valor de verdad* $s[x]$ *a todo proposición* $x \in AP$.
3. $Accs$ *es* conjunto de acciones *que puede realizar el sistema.*
4. $k : S \to 2^{Accs} - \{\emptyset\}$ *una función que le asigna a cada estado el conjunto de acciones que puede realizar.*
5. $p : S \times Accs \times S \to [0, 1]$, *es una función tal que para todo* $s \in S$ *y para todo* $a \in k(s)$, $p(s, a, \cdot)$ *es una distribución de probabilidades. Dado* $t \in S$, *el valor* $p(s, a, t)$ *da la probabilidad de pasar al estado* t

dado que se realiza la acción a mientras el sistema estaba en el estado s.

6. *s_{in} es el estado inicial.*

Dado que p es una función de probabilidad debe satisfacer que para todo $s \in S$ y $a \in k(s)$ se cumple $\sum_{t \in S} p(s, a, t) = 1$.

Dado un estado $s \in S$, el sucesor de s es elegido en dos pasos: primero, una acción $a \in k(s)$ es elegida de forma no determinística; segundo, el estado sucesor $t \in S$ es elegido de acuerdo a la probabilidad $p(s, a, t)$. Una vez que el sistema se encuentra en el estado t, este procedimiento se repite para realizar una nueva transición al estado t'. Este proceso se repite una cantidad infinita de veces y genera una *ejecución* del sistema.

Definición 15 (Ejecución). *Una* ejecución *de un SNP* Π *es una secuencia infinita de* $\omega : s_0 s_1 s_2 \cdots$ *tal que para todo* $s_i \in S$*, existe* $a_i \in k(s_i)$ *tal que* $p(s_i, a_i, s_{i+1}) > 0$ *para todo* $i \geq 0$.

Definición 16. *Dado* $s \in S, a \in Accs$*, definimos los* estados siguientes *de* s *con respecto a la acción* a *como* $Sigs(s, a) = \{s' \mid p(s, a, s') > 0\}$

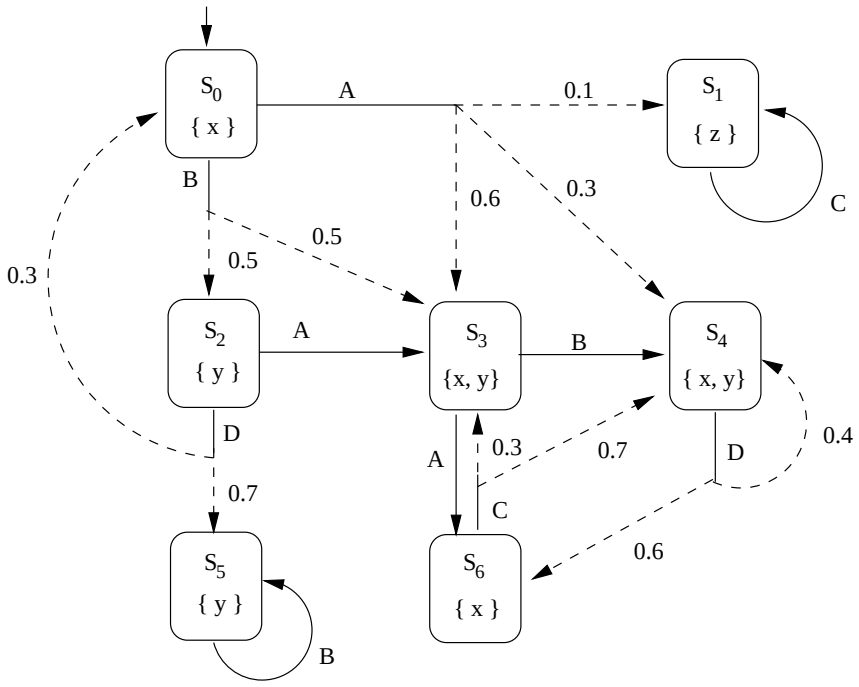


Figura 4.1: Sistema No determinístico Probabilístico

Ejemplo 10. *En la figura 4.1 podemos ver un SNP representado mediante un gráfico. El conjunto de proposiciones átomicas es* $AP = \{x, y, z\}$*, el conjunto de estados es* $S = \{s_0, \cdots, s_6\}$ *y el conjunto de acciones es* $Accs = \{A, B, C, D\}$*. Además se puede ver que* $k(s_0) = \{A, B\}$ *y* $k(s_5) = \{B\}$*. También se puede ver que* $p(s_0, a, s_1) = 0{,}1$*,* $p(s_0, a, s_3) = 0{,}6$*,* $p(s_0, a, s_4) = 0{,}3$*.*

Entre las ejecuciones del SNP podemos encontrar: $(s_0 s_2)^\omega$*,* $s_0(s_4 s_6 s_3)^\omega$*,* $s_0(s_3 s_6)^\omega$*,* $s_0 s_1^\omega$*.*

4.1. Probabilidades en un SNP

El espacio muestral sobre el cual trabajaremos serán las ejecuciones de un SNP. Para hacer esto debemos primero definir una estructura sobre estas que esté compuesta por elementos medibles. Esta estructura es un álgebra, llamada *σ-algebra de Borel* de conjuntos de secuencias, y sus elementos son conjuntos a los cuales son posibles asignar una probabilidad.[JKK66]

Notemos que a cada estado $s \in S$ podemos asociar un conjunto:

$$\Omega_s = \{s_0 s_1 s_2 \cdots \mid s = s_0 \wedge \forall n \in N : \exists a \in k(s_n) : p(s_n, a, s_{n+1}) > 0\}$$

Luego, dada una secuencia finita de estados $\sigma = \sigma_0 \cdots \sigma_n$ se define como cilindro básico inducido por σ al conjunto:

$$\sigma^\uparrow = \{\rho \in \Omega_s \mid \rho_0 = s_0 \wedge \cdots \wedge \rho_n = s_n\}$$

Denotamos con B_s a la σ-algebra generado por todos los cilindros básicos, i.e. $B_s \subseteq 2^{\Omega_s}$ es el menor conjunto conteniendo a los cilindros básicos que es cerrado por complemento y unión numerables. Luego, los elementos sobre los cuales mediremos una probabilidad pertenecerán al conjunto B_s.

Debido a la presencia del no determinismo, al momento de elegir las acciones, en los SNP no es posible definir una única función de probabilidad sobre los elementos de la σ-algebra de Borel B_s. A pesar de esto, para cada conjunto de secuencia $\Delta \in B_s$, podemos definir la *probabilidad máxima* $\mu^+(\Delta)$ y la *probabilidad mínima* $\mu^-(\Delta)$. Intuitivamente, $\mu^+(\Delta)$ representa la probabilidad de que el sistema siga una secuencia en Δ, remplazando el no determinismo por una elección arbitraria de las acciones tal que maximice este valor. $\mu^-(\Delta)$ es análogo con respecto a la mínimización. Para formalizar esta idea se utiliza el concepto de *estrategia*, la cual determina la probabilidad con la cual se eligen las acciones en un estado en particular.

Definición 17 (Estrategia). *Una* estrategia η *es un conjunto de funciones de probabilidades condicionales* $Q_\eta(a \mid s_0 s_1 \cdots s_n)$ *donde* $n \in N$*,* $s_0, s_1, \cdots, s_n \in S$ *y* $a \in k(s_n)$*, tal que* $\Sigma_{a \in k(s_n)} Q_\eta(a \mid s_0 s_1 \cdots s_n) = 1$*.*

Cuando un sistema se comporta de acuerdo a una estrategia η y desde el estado $s = s_0 \in S$ se alcanza el estado $s_n \in S$, siguiendo la secuencia $s_0 \cdots s_n$, se elige la próxima acción $a \in k(s_n)$ con una probabilidad $Q_\eta(a \mid s_0 \cdots s_n)$. Luego, la probabilidad de que el siguiente estado de la secuencia $s_0 \cdots s_n$ sea $t \in S$ es igual a:

$$\Pr_s^\eta(t \mid s_0 \cdots s_n) = \textstyle\sum_{a \in k(s_n)} Q_\eta(a \mid s_0 \cdots s_n) p(s_n, a, t).$$

Ahora podemos asociar a cada secuencia finita $s_0 \cdots s_n$ iniciada en $s = s_0$ de Ω_s la probabilidad $\Pr_s^\eta(s_0 \cdots s_n) = \prod_{i=0}^{n-1} \Pr_s^\eta(s_{i+1} \mid s_0 \cdots s_i)$. Esta probabilidad para secuencias finitas genera una única función de probabilidad $\mu_{s,\eta}$ en B_s tal que $\mu_{s,\eta}(\rho^\uparrow) = \Pr_s^\eta(\rho)$ para todo cilindro básico $\rho^\uparrow$.

Notemos que sería mucho más simple que la función de probabilidad con la cual se elige la siguiente acción no dependiera de toda la secuencia de estados recorrida y dependiera solamente del estado actual en la que se encuentra. Si bien esto es cierto, también es cierto que la definición que utilizamos contempla este caso, y además nos permite una mayor *expresividad* al momento de definir la estrategia, pues podemos elegir acciones en función de lo que ya pasó, por ejemplo, supongamos que quisieramos modelar un sistema que sólo pasa una vez por el estado s_j, luego, la probabilidad condicional de elegir una acción que me permita pasar por s_j dada una secuencia que esta conpuesta por el estado s_j será 0.

Ejemplo 11. *Tomemos SNP de la figura 4.1 y supongamos que la estrategia η esta compuesta por funciones de probabilidad condicional tales que:*

$$\Pr(A \mid s_0) = 0{,}5 \quad \Pr(B \mid s_0) = 0{,}5 \quad \Pr(D \mid s_0 s_2) = 0 \quad \Pr(A \mid s_0 s_2) = 1$$

luego la probabilidad de $s_0 s_1$ es:
$Q_\eta(A \mid s_0) * p(s_0, A, s_1) = 0{,}5 * 0{,}1 = 0{,}05$
la probabilidad de $s_0 s_2 s_5$ es:
$Pr_{s_0}^\eta(s_0 s_2 s_5) = Pr_{s_0}^\eta(s_2 \mid s_0) Pr_{s_0}^\eta(s_5 \mid s_0 s_2) = (0,5 * 0,5) * (1 * 0{,}3) = 0{,}55$
y la probabilidad de $s_0 s_3$ es:
$Pr_{s_0}^\eta(s_0 s_3) = Q_\eta(A \mid s_0) * p(s_0, A, s_3) + Q_\eta(B \mid s_0) * p(s_0, B, s_3) = 0{,}55$

Utilizando la definición de $\mu_{s,\eta}$ podemos definir la probabilidad máxima y mínima como sigue:

Definición 18. *La probabilidad máxima $\mu_s^+(\Delta)$ y la probabilidad mínima $\mu_s^-(\Delta)$ de un conjunto de secuencias $\Delta \in B_s$ están definidas por:*

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

Luego, $\mu_s^-(\Delta)$ y $\mu_s^+(\Delta)$ representan la probabilidad de que el sistema siga una evolución en Δ cuando las decisiones no deterministicas son tan *desfavorable* o *favorables* como sea posible.

Ejemplo 12. *Notemos en la figura 4.1, que la probabilidad de la ejecución* $s_0 s_1^\omega$ *es igual a la probabilidad del cilindro básico inducido por* $s_0 s_1^\uparrow$*. Esto se debe a que una vez que la ejecución llega a* s_1*, con probabilidad 1 elegirá siempre* s_1 *como el siguiente estado a transicionar. Calculemos ahora la probabilidad máxima y mínima de* $s_0 s_1^\uparrow$*:*
$\mu^\eta_{s_0}(s_0 s_1^\uparrow) = Pr^\eta_{s_0}(s_0 s_1) = Q_\eta(A \mid s_0) * p(s_0, A, s_1) = Q_\eta(A \mid s_0) * 0{,}1$
Luego la probabilidad máxima y mínima serán dadas, respectivamente, por la estrategias que maximicen y minimicen el valor de $Q_\eta(A \mid s_0)$*. Luego el valor máximo será dado por la estrategia que asigne a* $Q_\eta(A \mid s_0)$ *el valor 1 y el mínimo por la que le asigne el valor 0. Entonces tenemos que* $\mu^+_{s_0}(s_0 s_1^\omega) = 0{,}1$ *y que* $\mu^\eta_{s_0}(s_0 s_1^\omega) = 0$*.*

Los siguientes lemas fueron presentados en [dAB]. El primero establece que μ^+ y μ^- no son aditivos y el segundo relaciona la probabilidad máxima y la probabilidad mínima.

Lema 3. *Si* $\Delta_1, \Delta_2 \in B_s$*, con* $\Delta_1 \cap \Delta_2 = \emptyset$*, entonces:*

$$\mu^+_s(\Delta_1 \cup \Delta_2) \leq \mu^+_s(\Delta_1) + \mu^+_s(\Delta_2) \qquad \mu^-_s(\Delta_1 \cup \Delta_2) \geq \mu^-_s(\Delta_1) + \mu^-_s(\Delta_2)$$

Lema 4. *Para todo* $\Delta \in B_s$ *se cumple que* $\mu^-_s(\Delta) = 1 - \mu^+_s(\Omega_s - \Delta)$

4.2. Conjuntos Estables

Intuitivamente, un subconjunto de estados de un SNP es *estable* si hay una estrategia tal que para todo comportamiento que entra al subconjunto, este permanecerá siempre en él [dA97, dAB]. Más formalmente:

Definición 19. *Dado un SNP* $\Pi = \langle AP, S, Accs, k, p, s_{in}\rangle$ *y un conjunto* $E \subseteq S$*, E es un* conjunto estable *(CE) si:*

$$\forall s \in E : \exists a \in k(a) : Sigs(s, a) \subseteq E$$

Diremos que un CE B es un conjunto estable maximal *en* $C \subseteq S$ *si* $B \subseteq C$ *y no existe otro CE* B' *tal que* $B' \subseteq C$ *y* $B \subset B'$*.*

Dado un conjunto estable E, definamos la relación $\rho_E \subseteq E \times E$ como:

$$\rho_E = \{(s, t) \mid \exists a \in k(a) : t \in Sigs(s, a) \wedge Sigs(s, a) \subseteq E\}$$

luego si $(s, t) \in \rho_E$ hay una acción $a \in k(s)$ que salta desde s a t con probabilidad mayor que cero y que cae fuera de E con probabilidad nula. Si el grafo (B, ρ_B) es fuertemente conexo, diremos que B es un *conjunto estable fuertemente conexo*(CEFC). CEFC maximal se define de forma análoga a CE maximal.

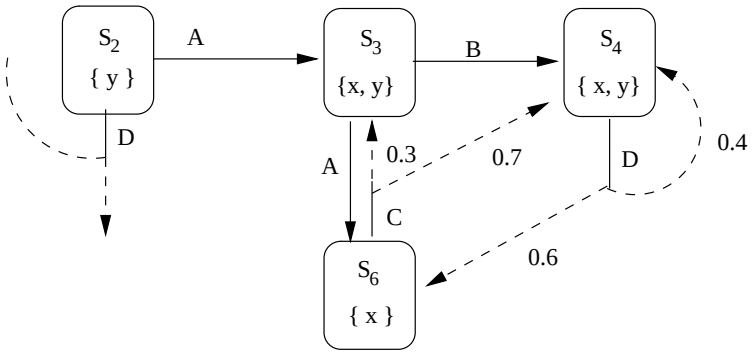


Figura 4.2: Conjunto Estable

Ejemplo 13. *En la figura 4.2 vemos un conjunto estable del SNP presentado en la figura 4.1. Notemos que no es fuertemente conexo pues no hay forma de alcanzar el estado s_2 desde los restantes.*

A continución unos lemas que resumen las propiedades relevantes de los CE y los CEFC [dA97].

El primer lema establece que una vez que se alcance un CE E, es posible elegir las acciones de tal forma que todo ejeccuión que entra a E se quede transicionando entre estados de E para siempre. Esto es posible ya que por como está definido un CE, para todo estado del CE existe una acción que no me permite transicionar a un estado fuera de él.

Lema 5. *Dado un CE E, existe una estrategia tal que para todo comportamiento que alcanza un elemeto de E, se quedará siempre en E con probabilidad igual a 1.*

El siguiente lema es similar al anterior, pero con respecto a un CEFC J. Por como están definidos los CEFC, para todo estado es posible *alcanzar* cualquier otro estado de J con probabilidad mayor que 0. Este hecho nos permite poder definir una estraregia tal que toda ejecución que alcance a J, se quede siempre transicionando por todos los estados de J.

Lema 6. *Dado un CEFC J, existe una estrategia tal que para todo comportamiento que alcanza un elemeto de J, se quedará siempre en J y visitará todos los estados de J con probabilidad igual a 1.*

El próximo lema establece que toda ejecución válida finalmente llega a un CEFC y se queda transicionando en ese conjunto para siempre. Este hecho es importante ya que si bien hay infinitas ejecuciones validas, la cantidad de CEFC es limitada.

Lema 7. *Para toda estado $s \in S$ y toda estrategia η:*

$$\mu_s^\eta(\{\omega \in \Omega_s \mid \mathit{inft}(\omega) \text{ es un } CEFC\}) = 1.$$

El último lema es un resultado del anterior. Este estable que si una ejecución visita infinitamente elemtos de un conjuntos de estados C, entonces esos estados tienen que estar incluidos en la union de los CEFC maximales del conjunto C.

Lema 8. *Dado cualquier conjunto C de estados de un SNP, sean $D_1, \cdots, D_n$ los CEFC maximales en C y sea $D = \bigcup_{i=1}^{n} D_i$. Entonces, para todo $s \in S$ y toda estrategia η, $\mu_s^\eta(\{\omega \mid \mathit{inft}(\omega) \subseteq C \wedge \mathit{inft}(\omega) \not\subseteq D\}) = 0$.*

Estos lemas son importantes pues se utilizan para el cálculo de la probabilidad máxima de que una propiedad expresada mediante fórmulas LTL se cumpla en un SNP. Los siguientes serán importantes para el cálculo de la probabilidad mínima.

Primero definiremos un conjunto del cual es imposible salir para toda estrategia. Estos nos garatizará que para toda estrategia, toda ejecución que alcance el conjunto siempre permanecerá en él.

Definición 20 (conjunto fuertemente estable). *Dado un SNP $\Pi = \langle AP, S, Accs, k, p, s_{in}\rangle$ y un conjunto $M \subseteq S$, M es un* conjunto fuertemente estable *(CFE) si:*

$$\forall s \in M \wedge \forall a \in k(s) : \mathit{Sigs}(s, a) \subseteq M$$

Diremos que un CFE M es un conjunto fuertemente estable maximal *en $C \subseteq S$ si $M \subseteq C$ y no existe otro CFE M' tal que $M' \subseteq C$ y $M \subset M'$.*

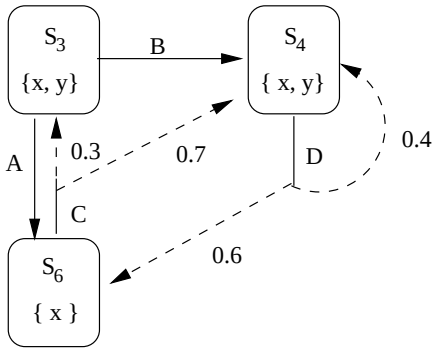


Figura 4.3: Conjunto Fuertemente Estable

Dado 2 conjuntos de estados, definiremos una relación entre ellos cuando desde todo estado de uno de los conjuntos se puede alcanzar un elemento del otro con probabilidad mayor que cero para toda estrategia.

Definición 21 (conjunto obligado). *Dado un SNP* $\Pi = \langle AP, S, Accs, k, p, s_{in}\rangle$ *y un conjunto* $C \subseteq S$*, diremos que* $A \subseteq S$ *es un* conjunto obligado *(CO) de* C *si para todo estado* $s \in C$ *y estrategia* η *se cumple :*

$$\exists \rho \in S^* : \rho_0 = s \wedge \rho_{|\rho|} \in A \wedge \Pr_s^\eta(\rho) > 0$$

A continuación un lema que relaciona un CFE C con A, un CO de C. Este nos garantiza que las ejecuciones dentro de C que nunca visitan los estados de A tienen probabilidad 0, a este hecho se debe el nombre *conjunto obligado*. Utilizaremos la notación $est(\rho)$ para denotar los estados que aparacen en la secuencia de estados o en la ejecución ρ.

Lema 9. *Dado un conjunto fuertemente estable* C*, sea* A *un CO de* C*. Entonces para toda estrategia* η *y todo estado* $s \in C$ *se cumple:*

$$\mu_s^\eta(\{\omega \in \Omega_s \mid est(\omega) \cap A = \emptyset\}) = 0$$

lo cual es equivalente a :

$$\mu_s^\eta(\{\omega \in \Omega_s \mid est(\omega) \cap A \neq \emptyset\}) = 1$$

Demostración. Fijemos η y sea L un conjunto de secuencias de estados finita tal que todas tienen una probabilidad mayor que cero, empiezan en elementos de C y terminan en un estado de A. Además no hay 2 secuencias que empiecen en el mismo elemento. Formalmente:

$$\begin{aligned} L \subset C^* : \quad & (\forall l \in L : Pr_{l_0}^\eta(l) > 0 \wedge l_0 \in C \wedge l_{|l|-1} \in A) \wedge \\ & (\nexists l, l' \in L : l_0 = l'_0) \end{aligned}$$

Por definición de conjunto obligado sabemos que $|L| = |C|$, es decir, tenemos un camino para cada estado de C. Definamos $l_{max} = \max\{|l| : l \in L\}$.

Definamos ahora para $s \in C$ el conjunto ρ^s formado por secuencia de estados finitas que empiezan en s, tienen longitud l_{max}, una probabilidad mayor que 0 y pasan por algún estado de A. Formalmente:

$$\rho^s = \{\rho \in C^* \mid \rho_0 = s \wedge |\rho| = l_{max} \wedge \Pr_s^\eta(\rho) \wedge (\exists 0 \leq i < |\rho| : \rho_i \in A)\}.$$

Notemos que ρ^s no puede ser vacío ya que por lo menos incluye un elemento que tiene como prefijo un elemento de L que inicia en s.

Sea $m = \min\{\Pr_{\rho_0}^\eta(\rho) \mid \rho \in \bigcup_{s \in C} \rho^s\}$. Luego, la probabilidad mínima de que una secuencia de estados válida de longitud l_{max}, que empieza en un estado C, pase por A es m. Entonces la probabilidad máxima de que una secuencia válida de longitud l_{max}, que empieza en un estado de C, no pase por un estado de A es de $(1 - m)$. Es decir que dado $T(s) = \{\rho \in C^* \mid \rho_0 = s \wedge |\rho| = l_{max} \wedge est(\rho) \cap A = \emptyset\}$, tenemos que:

$Pr_s^\eta(T(s)) \leq (1-m)$

Entonces tenemos que $\mu_s^\eta(\{\rho^\uparrow \mid \rho \in T(s)\}) \leq (1-m)$ para todo estado y estrategia.

Notemos que para todo $k > 0$ tenemos que:
$\{\omega \in \Omega_s \mid est(\omega) \cap A = \emptyset\}) \subseteq$
$\{\rho^0 q^1 \rho^1 \cdots q^k \rho^k \uparrow \mid \rho^0 q^1 \in T(s) \wedge (q^i \rho^i) \in T(q^i)\}$

Demostremos entonces que para todo η, dado $s = \rho_0^0$, vale:
$\mu_s^\eta(\{\rho^0 q^1 \rho^1 \cdots q^k \rho^k \uparrow \mid \rho^0 q^1 \in T(s) \wedge (q^i \rho^i) \in T(q^i)\}) \leq (1-m)^{(k+1)}$
para todo $k > 0$. Utilizaremos indución en k para realizar la demostración. Además utilizaremos la notación $\eta \mid \rho$ para denotar a la estrategia que resulta de de η luego de haber recorrido la secuencia ρ. Es decir, $Q_{\eta|\rho}(A \mid \rho') = Q_\eta(A \mid \rho\rho')$

Caso base:
$\mu_s^\eta(\{\rho^0 q^1 \rho^1 \uparrow \mid (\rho^0 q^1) \in T(s) \wedge (q^1 \rho^1) \in T(q^1)\}) =$
$\sum_{(\rho^0 q^1) \in T(s)} \Pr_s^\eta(\rho^0 q^1) \cdot \mu_{q^1}^{\eta|\rho^0 q^1}(\{\rho^\uparrow \mid \rho \in T(q^1)\}) \leq$
$\sum_{(\rho^0 q^1) \in T(s)} \Pr_s^\eta(\rho^0 q^1) \cdot (1-m) \leq$
$\Pr_s^\eta(T(s)) \cdot (1-m) \leq (1-m)^2$

Caso Inductivo:
$\mu_s^\eta(\{\rho^0 q^1 \rho^1 \cdots q^k \rho^{(k+1)} \uparrow \mid \rho^0 q^1 \in T(s) \wedge (q^i \rho^i) \in T(q^i)\}) =$
$\sum_{(\rho^0 q^1) \in T(s)} \Pr_s^\eta(\rho^0 q^1) \cdot \mu_{q^1}^{\eta|\rho^0 q^1}(\{q^1 \rho^1 q^2 \rho^2 \cdots q^k \rho^{k+1} \uparrow \mid (q^i \rho^i) \in T(q^i)\}) =$
$\sum_{(\rho^0 q^1) \in T(s)} \Pr_s^\eta(\rho^0 q^1) \cdot \Pr_s^{\eta|\rho^0 q_1}(q^1 \rho_0^1) \cdot \mu_{\rho_0^1}^{\eta|\rho^0 q_1 \rho_0^1}(\{\rho^1 q^2 \rho^2 \cdots q^k \rho^{k+1} \uparrow \mid (\rho^1 q^2) \in$
$T(\rho_0^1) \wedge (q^i \rho^i) \in T(q^i)\}) \leq$
$\sum_{(\rho^0 q^1) \in T(s)} \Pr_s^\eta(\rho^0 q^1) \cdot \mu_{\rho_0^1}^{\eta|\rho^0 q_1 \rho_0^1}(\{\rho^1 q^2 \rho^2 \cdots q^k \rho^{k+1} \uparrow \mid (\rho^1 q^2) \in T(\rho_0^1) \wedge (q^i \rho^i) \in$
$T(q^i)\}) \leq$
$\sum_{(\rho^0 q^1) \in T(s)} \Pr_s^\eta(\rho^0 q^1) \cdot (1-m)^k \leq$
$(1-m) \cdot (1-m)^k = (1-m)^{k+1}$

Esto implica que $\mu_s^\eta(\{\omega \in \Omega_s \mid est(\omega) \cap A = \emptyset\}) \leq (1-m)^k$ para todo $k \geq 0$. Entonces $\mu_s^\eta(\{\omega \in \Omega_s \mid est(\omega) \cap A = \emptyset\}) = 0$. □

Ejemplo 14. *Tomemos la figura 4.3 y definimos $O = \{s_3\}$. Luego O es conjunto obligado del conjunto fuertemente estable, ya que $\Pr_{s_4}^\eta(s_4 s_6 s_3) > 0$ y $\Pr_{s_6}^\eta(s_6 s_3) > 0$ para todo η. Entonces, por el último lema, ejecuciones como s_4^ω tienen probabilidad cero. Este caso es fácil de analizar pues $\Pr_{s_4}^\eta(s_4^k) = (0{,}4)^k$ para todo $k > 0$, luego la probabilidad de transicionar siempre sobre s_4 es 0. Ahora analicemos las secuencias de la forma $(s_4 + s_6)^\omega$. Notemos que la cantidad de secuencias de esta forma es infinita, a pesar de tener un*

conjunto infinito, el lema nos asegura que la probabilidad de este conjunto también es 0.

Como resultado del lema anterior surge el siguiente, el cual nos garantiza que toda ejecución válida de un SNP que alcanza un conjunto fuertemente estable debe pasar infinitamente por los estados de sus conjuntos obligados.

Lema 10. *Dado un conjunto fuertemente estable C, sea A un CO de C. Entonces para toda estrategia η y todo estado $s \in S$ se cumple:*

$$\mu_s^\eta(\{\omega \in \Omega_s \mid (\exists q \in est(\omega) : q \in C) \wedge (inft(\omega) \cap A = \emptyset)\}) = 0$$

Demostración. Notemos que:
$\{\omega \in \Omega_s \mid (\exists q \in est(\omega) : q \in C) \wedge (inft(\omega) \cap A = \emptyset)\} =$
$\{(\rho q \omega) \in \Omega_s \mid \rho \in S^* \wedge q \in C \wedge est(q\omega) \cap A = \emptyset\}$

Definamos a R como:
$R = \{\rho q \mid (\rho q \omega) \in \Omega_s \wedge \rho \in S^* \wedge q \in C \wedge est(q\omega) \cap A = \emptyset\}$

Notemos que si $(\rho q \omega) \in \Omega_s$, entonces $(q\omega) \in \Omega_q$. Fijemos η, s y calculemos $\mu_s^\eta(\{\omega \in \Omega_s \mid (\exists q \in est(\omega) : q \in C) \wedge (inft(\omega) \cap A = \emptyset)\})$.

$\mu_s^\eta\{\omega \in \Omega_s \mid (\exists q \in est(\omega) : q \in C) \wedge (inft(\omega) \cap A = \emptyset)\} =$
$\mu_s^\eta\{(\rho q \omega) \in \Omega_s \mid \rho \in S^* \wedge q \in C \wedge est(q\omega) \cap A = \emptyset\} =$
$\sum_{\rho q \in R} \Pr_s^\eta(\rho q) \cdot \mu_q^{\eta|\rho q}\{(q\omega) \in \Omega_q \mid est(q\omega) \cap A = \emptyset\} =$
$\sum_{\rho q \in R} \Pr_s^\eta(\rho q) \cdot 0 = 0$

□

Capítulo 5

Model Checking

En este capítulo presentaremos algoritmos para cacular la probabilidad máxima y mínima de que una propiedad expresada mediante una fórmula LTL se satisfaga en un SNP.

Para realizar esto se debe generar primero el autómata de Buchi correspondiente a la fórmula a verificar y a continuación transformar este en un autómata de Rabin determinista. Luego se sincroniza el SNP con el autómata de Rabin de la fórmula LTL. Sobre el resultado de esta sincronización se buscan dos tipos de subconjuntos de estados con particularidades especiales. Un tipo de subconjunto será utilizado para el cálculo del máximo y el otro, para el cálculo del mínimo. Estos subconjuntos son de tal forma que nos garantizan que los conjuntos de ejecuciones que satisfacen la fórmula con probabilidad mayor que 0 sí o sí deben alcanzar algún elemento de estos. Entonces el problema de calcular máximo y mínimo se reduce a resolver el problema de calcular la probabilidad máxima y mínima de alcanzar algún estado en estos subconjuntos.

En la primera sección explicaremos cómo sincronizar un SNP con el autómata de Rabin derivado de una fórmula LTL. En la siguiente sección presentaremos la forma de calcular la probabilidad máxima y mínima de que la propiedad se satisfaga.

5.1. Sincronización entre un SNP y una fórmula LTL

Dado un SNP $\Pi = \langle AP, S, Accs, k, p, s_{in} \rangle$ y una fórmula LTL ψ, primero debemos generar el AR que acepta el mismo lenguaje que ψ. Una vez que se obtiene el AR y el SNP estos se sincronizan generando una nueva estructura. Esta nueva estructura se define de la siguiente manera:

Definición 22. *Dado un PSN* $\Pi = \langle AP, S, Accs, k, p, s_{in} \rangle$ *y el AR generado a partir de la fórmula LTL* ψ, $A_\psi = \langle \Sigma, Q, q_0, \delta, R \rangle$, *un* SNP de Rabin *es*

una 7-upla $\Pi' = \langle AP, S', Accs, k', p', s'_{in}, R' \rangle$ *que se genera de la siguiente forma:*

1. $S' = S \times Q$, *donde* $(t,q)[p] = t[p]$ *para todo* $(t,q) \in S'$ *y* $p \in P$.
2. *Para todo* $(t,q) \in S', k'(t,q) = k(t)$.
3. *Para cada* $t \in S$ *y* $a \in k(t)$, *la probabilidad* $p'((t,q),a,(t',q'))$ *de transicionar del estado* (t,q) *al estado* (t',q') *con la acción* a *es igual a* $p(t,a,t')$ *si* $\delta(q,l(t')) = q'$, *y es igual a cero caso contrario.*
4. $s'_{in} = (s_{in}, \delta(q_{in}, l(s_{in})))$.
5. $R' = \{(E'_0, F'_0), \cdots, (E'_{|R|-1}, F'_{|R|-1})\}$, *donde* $E'_i = S \times E_i$ *y* $F'_i = S \times F_i$ *para* $0 \leq i < |R|$.

El resultado de la sincronización es un nuevo SNP con una nueva variable la cual está codificada en los estados de S'. Èsta corresponde a la segunda componente del par (i.e. q en $(t,q) \in S'$) y se encarga de llevar un "registro" de como se comportaría una ejecución en el autómata de la fórmula, en función de las proposiciones que son válidas en los estados por los que pasa una ejecución del SNP. Luego, si una ejecución ρ del SNP de Rabin satisface $inft(\rho) \subseteq E'_i$ y $inft(\rho) \cap F'_i \neq \emptyset$ para algun i, ρ es una ejecución que satisface la propiedad ψ.

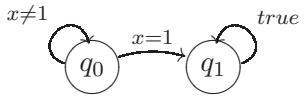


Figura 5.1: Autómata de rabin que representa $\mathsf{F}\ (x = 1)$

Ejemplo 15. *En la figura 5.1 tenemos la fórmula* $\psi = \mathsf{F}\ (x = 1)$ *representada por un AR donde* $R = \{(\{q_0, q_1\}, \{q_1\})\}$. *En la figura 5.2 un SNP y en la figura 5.3 un SNP de Rabin, resultado de la intersección entre ambos.*

5.2. Cálculo de probabilidades máximas y mínimas

Para calcular las probabilidades máxima y mínima de que una propiedad ψ expresada en LTL se satisfaga en un SNP Π debemos primero generar, en función de éstos, el SNP de Rabin:

$$\Pi_\psi = \langle AP, S, Accs, k, p, s_{in}, \{(E'_0, F'_0), \cdots, (E'_n, F_{n'})\} \rangle.$$

Cálculo de la probabilidad máxima:

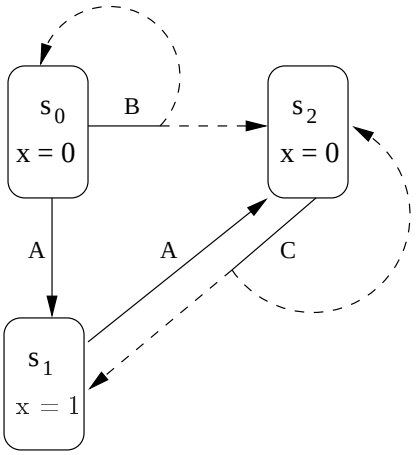


Figura 5.2: Sistema No determinístico Probabilístico

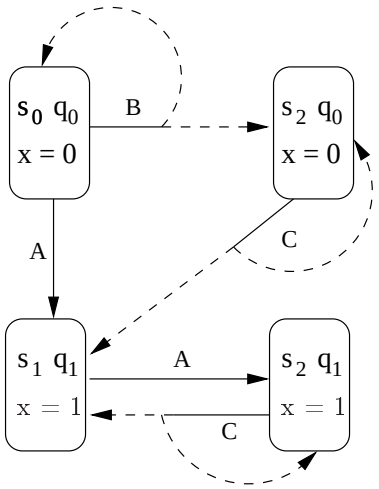


Figura 5.3: Intersección entre un SNP y el autómata de un fórmula LTL

Notemos que por el lema 7 todas las ejecuciones válidas de un SNP terminan recorriendo estados en un CEFC. Notemos también que las ejecuciones ω sobre Π_ψ que satisfacen la propiedad ψ, deben ser tal que $inft(\omega) \subseteq E'_i$ y $inft(\omega) \cap F'_i \neq \emptyset$. Entonces el conjunto de secuencias que satisfacen ψ deben alcanzar algún CEFC incluido en algún E'_i que no posea intersección vacia con el F'_i. Recordemos que, por como esta definido un CEFC, sabemos que existe una estrategia tal que es posible visitar todos los estados del CEFC con probabilidad 1. Usando este hecho, el problema de calcular la probabilidad máxima de que la propiedad se cumpla, se reduce a calcular la probabilidad máxima de alcanzar algún elemento de algún CEFC incluido en algún E'_i que no posea intersección vacia con el conjunto F'_i. Es decir que hemos reducido nuestro problema a calcular un problema de alcanzabilidad. Remarquemos una vez más el detalle de que no sabemos como es la estrategia a partir de que se alcanza algún elemento del CEFC pero si sabemos de su existencia y que con ésta, a partir de ese momento, la probabilidad de que la propiedad se cumpla es 1 y por lo tanto no existe otra estrategia tal que genere un valor de probabilidad mayor para el conjunto de secuencias que satisfacen ψ.

Entonces para calcular la probabilidad máxima primero debemos buscar $B_0^{(i)}, \cdots, B_{n_i}^{(i)}$, los CEFC contenidos en E'_i que no tinen intersección vacia con F'_i. Luego para $0 \leq i \leq n$ y $0 \leq j \leq n_i$:

$$B_j^{(i)} \subseteq E'_i \qquad\qquad B_j^{(i)} \cap F'_i \neq \emptyset$$

Definamos $T = \bigcup_{i=0}^{n} \bigcup_{j=0}^{n_i} B_j^{(i)}$. Ahora solo debemos calcular la probabilidad máxima de alcanzar T.

Este proceso esta fundamentado por el siguiente teorema [dA97]:

Teorema 2. $\mu^+_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi\}) = \sup_\eta \mu^\eta_{s_{in}}\{\omega \in \Omega_{s_{in}} \mid \exists k : \omega_k \in T\}$

Cálculo de la probabilidad mínima:

La idea para calcular el mínimo es similar a la utilizada para calcular la probabilidad máxima. Es decir, debemos encontrar subconjuntos del SNP, en los cuales la propiedad se cumpla con probabilidad 1. Pero notemos, que a diferencia del caso del máximo, no alcanza con pedir la existencia de una estrategia tal que le asigne a las ejecuciones dentro del subconjunto, una probabilidad 1 de que se satisfaga ψ. Esto es debido a que de esta forma se permite que existan otras estrategias tales que las ejecuciones dentro del subconjunto que satisfacen ψ tengan una probabilidad menor a 1, por lo cual el mínimo no sería bien calculado. En función de este hecho debemos buscar subconjuntos tales que para toda estrategia y estado, la probabilidad

de que ψ se satisfaga sea 1. Luego si la probabilidad de las ejecuciones de SNP que satisfacen ψ tienen una probabilidad mínima mayor que cero, estas deben alcanzar alguno de estos subconjuntos. Entonces, una vez encontrados estos subconjuntos, calcular la probabilidad mínima de que la probabilidad se satisfaga se reduce a mínimizar la probabilidad de alcanzar los mismos.

Estos subconjuntos "especiales" a los cuales nos estamos refiriendo, son los CFE incluidos en los E'_i que poseen como CO F'_i. Entonces, para calcular la probabilidad mínima debemos buscar los subconjuntos $B_i \subseteq E'_i$, los cuales satisfacen que son el mayor subconjunto no vacio de E'_i tal que:

1. B_i es CFE.
2. F'_i es CO de B_i.

Notemos que una vez más vale para $0 \leq i \leq n$:

$$B_i \subseteq E'_i \qquad\qquad B_i \cap F'_i \neq \emptyset$$

La inclusión es clara y la intersección se debe a que si F'_i es CO de B_i, debe existir un camino desde los elementos de B_i a al menos un elemento de F_i, pero las transiciones de los estados de B_i son, por ser B_i CFE, de tal forma que nunca saltan a un estado fuera de B_i, por lo tanto $B_i \cap F'_i \neq \emptyset$.

Definamos $T = \bigcup_{i=0}^{n} B_i$. Ahora calcular la probabilidad mínima se reduce a calcular la probabilidad mínima de alcanzar T. A continuación la demostración de este hecho:

Teorema 3. $\mu^{-}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi\}) = \inf_{\eta} \mu^{\eta}_{s_{in}}\{\omega \in \Omega_{s_{in}} \mid \exists k : \omega_k \in T\}$

Demostración. Primero notemos que todo secuencia ω alcanza T si o solo si $\mathit{inft}(\omega) \subseteq T$. Ahora fijemos una estrategia η, luego:
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi\}) =$
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge \exists k : \omega_k \in T\}) +$
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge \nexists k : \omega_k \in T\}) =$
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge \mathit{inft}(\omega) \subseteq T\}) +$
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge \mathit{inft}(\omega) \not\subseteq T\})$

Entonces por como definimos T y el lema 9, estamos seguros que toda secuencia que alcance T satisface ψ, por lo tanto:
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge \mathit{inft}(\omega) \subseteq T\}) = \mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \exists k : \omega_k \in T\})$

Analicemos ahora $\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge \mathit{inft}(\omega) \not\subseteq T\})$. Si $\mathit{inft}(\omega) \not\subseteq T$ pueden pasar 2 cosas: la primera es que $\mathit{inft}(\omega) \not\subseteq C$ un CFE incluido en algún E_i, i.e. ω nunca llega a un CFE, o que $\mathit{inft}(\omega) \subseteq C$ un CFE incluido en algún E_i con $C \cap T = \emptyset$, i.e. llegamos a un CFE que el cual no tiene como CO algún F_i. Pedimos $C \cap T = \emptyset$ para dejar claro que C no esta compuesto

por elemetos de T, pues no alcanza con pedir "F_i no es CO de C para todo i", pues de esta forma se permitirian casos como $C = C' \cup B_i$ para cualquier i. Es decir, incluriamos CFE formados por los conjuntos "especiales" que estamos buscando más otros estados del SNP de Rabin, y los caminos que caen en los conjuntos "especiales" ya han sido contemplados.

Usamos la siguiente notación para estos predicados:
$p_1(\omega) = \mathit{inft}(\omega) \not\subseteq C$ un CFE incluido en algún E_i.
$p_2(\omega) = \mathit{inft}(\omega) \subseteq C$ un CFE incluido en algún E_i con $C \cap T = \emptyset$

Luego, como toda secuencia que satisface ψ y no alcance T satisface solamente uno de los predicados p_i, tenemos que :
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge \mathit{inft}(\omega) \not\subseteq T\}) =$
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge p_1(\omega)\}) +$
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge p_2(\omega)\})$

Hasta ahora tenemos lo siguiente:
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi\}) =$
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \exists k : \omega_k \in T\}) +$
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge p_1(\omega)\}) +$
$\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge p_2(\omega)\})$

No podemos asegurar que los 2 últimos terminos de la suma sean cero, pues eso depede de la estrategia η y no tenemos ninguna hipotesis sobre ésta. Pero supongamos momentaneamente que para todo η se pueda definir η' tal que:
$\mu^{\eta'}_{s_{in}}(\{\omega \in \Omega a_{s_{in}} \mid \exists k : \omega_k \in T\}) = \mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \exists k : \omega_k \in T\})$
$\mu^{\eta'}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge p_1(\omega)\}) = 0$
$\mu^{\eta'}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge p_2(\omega)\}) = 0$

Entonces se cumpliría para todo η:

$$\mu^{\eta'}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi\}) \leq \mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi\})$$

Luego esta desigualdad para todo η nos indica que el ínfimo, con respecto a η, de $\mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi\})$ se encuentra en las estrategias de tipo primadas. Entonces:

$$\begin{aligned} \mu^{-}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi\}) &= \inf_{\eta} \mu^{\eta}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi\}) \\ &= \inf_{\eta'} \mu^{\eta'}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \omega \models \psi\}) \\ &= \inf_{\eta'} \mu^{\eta'}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \exists k : \omega_k \in T\}) \\ &= \mu^{-}_{s_{in}}(\{\omega \in \Omega_{s_{in}} \mid \exists k : \omega_k \in T\}) \end{aligned}$$

Demostremos ahora que efectivamente para todo η podemos definir η' con las características deseadas, con lo cual el teorema estaría demostrado.

Demostremos primero la existencia de una estrategia η_1 tal que para $K = \{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge p_1(\omega)\}$, $\mu^{\eta_1}_{s_{in}}(K) = 0$.

Notemos que, para todo $\omega \in K$, por valer $p_1(\omega)$ existe un estado $s \in \mathit{inft}(\omega)$ para el cual existe una acción $a \in k(s)$, tal que existe $s' \notin \mathit{inft}(\omega)$ para el cual se cumple $p(s,a,s') > 0$. Denotemos a estos estados, para una ejecucion ω, de la siguiente forma:

$M(\omega) = \{s \in \mathit{inft}(\omega) \mid \exists a \in k(s) : \exists s' \notin \mathit{inft}(\omega) : p(s,a,s') > 0\}$

Usando este hecho podemos definir inductivamente conjuntos de prefijos de los elementos de K, de la siguiente forma:

$$\begin{aligned} P_0 = \quad & \{\rho \in S^* \mid \exists \omega' \in S^\omega : \rho\omega' \in K \wedge \mathit{inft}(\omega') = \mathit{est}(\omega') \\ & \wedge \mathit{inft}(\omega') \subseteq \mathit{est}(\rho) \wedge inft(\omega') \not\subseteq \mathit{est}(\rho) - \{\rho_{|\rho|-1}\}\} \\ P_{n+1} = \quad & \{(\rho\rho^1 \cdots \rho^{n+1} s s') \in S^* \mid s, s' \in S \wedge \rho, \rho^1, \cdots, \rho_{n+1} \in S^* \\ & (\rho\rho^1 \cdots \rho^n) \in P_n \ \wedge [\exists \omega' \in S^\omega : \rho\rho^1 \cdots \rho^{n+1} s s' \omega' \in K \\ & \wedge s \in M(\omega') \wedge \nexists s'' \in \mathit{est}(\rho^{n+1}) : s'' \in M(\omega')]\} \end{aligned}$$

Luego tenemos que $\lim_{i\to\infty} P_i = K$.

Sea $m = \min\{p(s,a,s') > 0 \mid s, s' \in S \wedge a \in k(s)\}$, es decir la mínima probabilidad, distinta de 0, de transicionar desde un estado cualquiera a otro.

Definamos ahora la estrategia η_1 de forma tal que si $\rho \in S^*$ y $a \in k(\rho_{|\rho|-1})$ son tales que $\mathit{Sigs}(\rho_{|\rho|-1}, a) - \mathit{est}(\rho) \neq \emptyset$, se elige a con probabilidad uno. Es decir que la estrategia elegirá la acción que nos posibilite visitar un estado no visitado. Con esta estrategia vamos a demostrar que $\Pr^{\eta_1}_{s_{in}}(P_i) \leq (1-m)^i$ para $i > 0$.

El caso base es trivialmente(:P) cierto, pues $\Pr^{\eta_1}_{s_{in}}(P_0) \leq (1-m)^0 = 1$.

En el caso inductivo tenemos que:

$$\begin{aligned} \Pr^{\eta_1}_{s_{in}}(P_{n+1}) \quad & = \textstyle\sum_{(\rho\rho^1\cdots\rho^{n+1}ss')\in P_{n+1}} \Pr^{\eta_1}_{s_{in}}(\rho\rho^1 \cdots \rho^{n+1} s s') \\ & = \textstyle\sum_{(\rho\rho^1\cdots\rho^{n+1}ss')\in P_{n+1}} \Pr^{\eta_1}_{s_{in}}(\rho\rho^1 \cdots \rho^{n+1} s) \\ & * \Pr^{\eta_1}_{s_{in}}(s' \mid \rho\rho^1 \cdots \rho^{n+1} s) \end{aligned}$$

Por como fueron construidos los prefijos, sabemos que existe una accion en el estado s con la cual es posible visitar un estado aún no visitado con una probabilidad de al menos m. Ese hecho y como fue definida la estrategia nos aseguran que:

$\Pr^{\eta_1}_{s_{in}}(s' \mid \rho\rho^1 \cdots \rho^{n+1} s) < (1-m)$

Luego tenemos que:

$$\begin{aligned} \Pr_{s^{\eta_1}_{in}}(P_{n+1}) \quad & \leq \textstyle\sum_{(\rho\rho^1\cdots\rho^{n+1}ss')\in P_{n+1}} \Pr^{\eta_1}_{s_{in}}(\rho\rho^1 \cdots \rho^{n+1} s) * (1-m) \\ & \leq \textstyle\sum_{(\rho\rho^1\cdots\rho^{n+1}ss')\in P_{n+1}} \Pr^{\eta_1}_{s_{in}}(\rho\rho^1 \cdots \rho^{n}) * (1-m) \\ & \leq \textstyle\sum_{(\rho\rho^1\cdots\rho^{n+1}ss')\in P_{n+1}} (1-m)^n * (1-m) = (1-m)^{n+1} \end{aligned}$$

Como $\mu^{\eta_1}_{s_{in}}(K) = \lim_{i\to\infty} \Pr^{\eta_1}_{s_{in}}(P_i) \leq \lim_{i\to\infty}(1-m)^i = 0$, tenemos finalmente que $\mu^{\eta_1}_{s_{in}}(K) = 0$.

Demostremos ahora la existencia de la estrategia η_2 tal que si $K = \{\omega \in \Omega_{s_{in}} \mid \omega \models \psi \wedge p_2(\omega)\}$, $\mu^{\eta_2}_{s_{in}}(\{K\}) = 0$.

Sea $I = \bigcup_{\omega \in K} \mathit{inft}(\omega)$. Además definamos P, un conjunto de prefijos particular del conjunto K, de la siguiente forma:

$$P = \{\rho^0\rho^1 \in S^* \mid \quad (\exists \omega' \in S^\omega : \rho^0\rho^1\omega' \in K \wedge \\ \mathit{est}(\rho^0) \cap \mathit{inft}(\omega') = \emptyset \wedge \rho^1_0 \in \mathit{inft}(\omega')) \wedge \\ \rho^1_{|\rho^1|-1} \in F \wedge \forall i : 0 < i < |\rho^1| - 1 : \rho^1_i \notin F \ \}$$

Definamos ahora para todo $s \in I$, $P_s = \{\rho^0\rho^1 \in P \mid \rho^1_0 = s\}$. Luego $P_s \cap P_{s'} = \emptyset$ si $s \neq s'$, pues el primer estado de I que alcanza una secuencia es único. Notemos que $K \subseteq \bigcup_{s \in I}\{(\rho^0\rho^1)^\uparrow \mid (\rho^0\rho^1) \in P_s\}$, pues el operador $\uparrow$ no impone ninguna restriccion sobre el subfijo de la ejecución. Luego tenemos que para toda estrategia vale:
$\mu^\eta_{s_{in}}(K) \leq \mu^\eta_{s_{in}}(\bigcup_{s \in I}\{(\rho^0\rho^1)^\uparrow \mid (\rho^0\rho^1) \in P_s\})$
$\mu^\eta_{s_{in}}(K) \leq \sum_{s \in I} \mu^\eta_{s_{in}}(\{(\rho^0\rho^1)^\uparrow \mid (\rho^0\rho^1) \in P_s\})$

Recordemos que por hipotesis, ningún subconjunto de I tiene como conjunto obligado a T, entonces para cada $s \in I$ existe una estragia η_s tal que la $\Pr^{\eta_s}(\rho) = 0$ si $\rho \in S^*$ y $\mathit{est}(\rho) \cap F \neq \emptyset$. Luego, como $\mathit{est}(\rho^1) \cap F = \{\rho^1_{|\rho^1|-1}\}$, si definimos a η_2 de tal forma que se comporte como η_s para todo cilindro que alcance $s \in I$, tenemos que:
$\mu^{\eta_2}_{s_{in}}(K) \leq \sum_{s \in I} \mu^{\eta_2}_{s_{in}}(\{(\rho^0\rho^1)^\uparrow \mid (\rho^0\rho^1) \in P_s\})$
$\mu^{\eta_2}_{s_{in}}(K) \leq \sum_{s \in I} \mu^{\eta_s}_{s_{in}}(\{(\rho^0\rho^1)^\uparrow \mid (\rho^0\rho^1) \in P_s\})$
$\mu^{\eta_2}_{s_{in}}(K) \leq 0$

Luego para toda estrategia η podemos definir una estrategia η' que se comporte como η para las transiciones que alcanzan T, que se comporte como η_1 para las transiciones que no alcanzan T y satisfacen p_1 y que se comporte como η_2 para las que no alcanzan T y satisfacen p_2, pues estos conjuntos de ejecuciones son disjuntos.

□

5.3. Algoritmo para calcular B_i

En esta sección se explicará cómo calcular los conjuntos fuertemente estables en E_i que no poseen intersección vacía con F_i, es decir, los B_i para el cálculo de la probabilidad mínima.

Primero debemos buscar, para cada i, el conjunto fuertemente estable maximal en E_i, al que llamaremos C_i. Estos se calculan de la siguiente manera: dado $A \subseteq S$, donde S son los estados de SNP, definimos a $F : 2^S \to 2^S$ como:

$F(A) = \{s \in A \mid \forall a \in k(s) : \mathit{Sigs}(s, a) \subseteq A\}$, luego
$C_i = F^\infty(E_i)$

Notemos que, si bien la recursión es infinita, en a lo sumo $|E_i|$ pasos obtendremos C_i, pues en cada llamado, como mínimo, se descarta un elemento de E_i; o no se descarta nada. Si no se descarta nada, no tiene sen-

tido seguir haciendo la recursión. En otras palabras se continúa hasta que $F^n(E_i) = F^{n+1}(E_i)$

C_i es un conjunto fuertemente estable pero este puede no tener como CO a F_i, esto se debe a que este puede incluir subconjuntos fuertemente estables los cuales, pueden o no, poseer intersección vacía con F_i. La siguiente figura refleja la situación planteada:

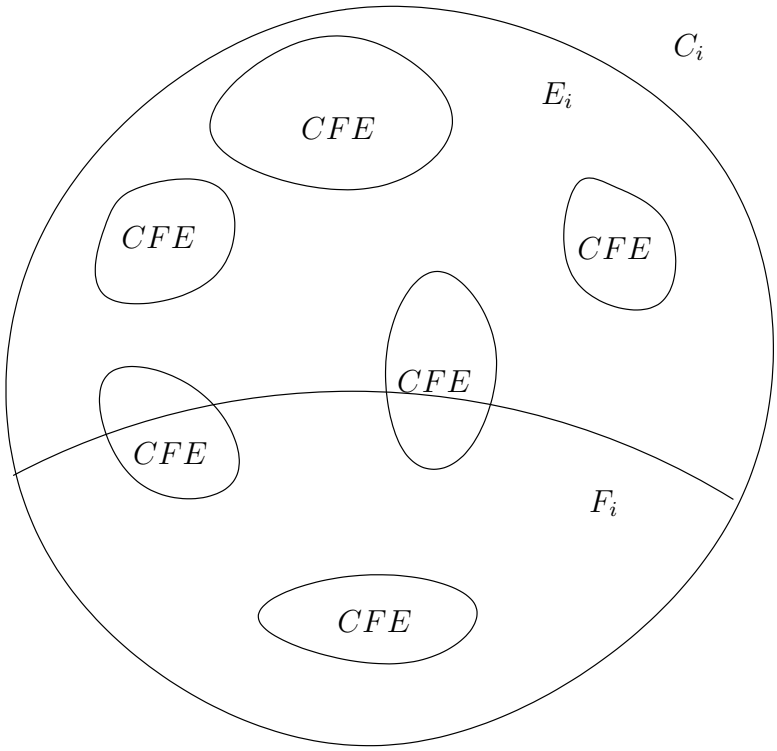


Figura 5.4: CFE maximal en E_i

Observemos que para todo estado que pertenece a un CFE en C_i que posee intersección vacía con F_i, por definición de CFE, es imposible alcanzar un estado de F_i. Por esta razón, se deben calcular y descartar estos subconjuntos. Llamemos D_i a la unión de estos y utilicemos a la función F para calcularlos. Entonces $D_i = F^\infty(C_i - F_i)$.

Ahora sólo resta calcular el CFE maximal en $C_i - D_i$, pues esto da como resultado la unión de CFE en C_i que no poseen intersección vacía con F_i. Es decir que $B_i = F^\infty(C_i - D_i)$.

Capítulo 6

Implementación

Se implementó una herramienta llamada *SSS reducer* que utiliza los resultados del capítulo 5 para reducir el problema de encontrar la probabilidad mínima de una propiedad expresada en LTL a un problema de alcanzabilidad. En las siguientes secciones se describen en detalle los aspectos más relevantes de la herramienta, como son, por ejemplo, su arquitectura y modo de uso. La herramienta se distribuye bajo la licencia GPLv2, como se indica en el archivo COPYING incluido en el código fuente.

SSS reducer

Como se indicó en la introducción, SSS reducer es una herramienta para reducir el problema de calcular la probabilidad mínima de una propiedad (expresada en LTL) sobre un sistema a un problema de alcanzabilidad. La herramienta está basada en los resultados del catpítulo 5 para hacer la reducción, y el output generado se puede utilizar con la herramienta *Rapture* [JDL02], que se encarga de resolver el problema de alcanzabilidad en un SNP. Esta herramienta puede descargarse del sitio web de Bertrand Jeannet, ubicado en http://pop-art.inrialpes.fr/people/bjeannet/.

Consideraciones generales y modo de uso

El programa está implementado en C++, utilizando la implementación del compilador de GNU (g++). SSS reducer fue compilado y ejecutado con éxito utilizando g++ version 3.4.5 y 3.4.6 en arquitecturas x86 y x86_64.

SSS reducer lee su entrada de *stdin* y escribe el resultado en *stdout*. Por lo tanto, una invocación típica del programa será:

```
$ ./sss_reducer < descripcion_del_sistema.pts > output_para_rapture.out
```

Luego de la ejecución, el archivo *output_para_rapture.out* puede ser utilizado con el programa Rapture.

La sintaxis del archivo de entrada es muy similar a la que soporta Rapture, aunque existen algunas diferencias. En el apédince A se encuentra la gramática soportada. Es recomendable, también, leer el archivo README provisto con el código fuente, pues en él se describen en detalle esas diferencias. En el apéndice B se encontrarán instrucciones de compilación, descripción de la orgranización del código y documentación.

También es recomendable analizar en detalle los ejemplos provistos con el código fuente, que se encuentran en el directorio *examples*.

6.1. Arquitectura de SSS reducer

6.1.1. Programas externos necesarios

SSS reducer utiliza programas externos para su correcto funcionamiento. Particularmente, usa el programa *ltl2dstar* y este, a su vez, el programa *Spin*. Además, genera output para *Rapture* por lo que, en cierto modo, también necesita dicho programa. SSS reducer ha sido probado con versiones particulares de dichos programas, y no se sabe sin funcionará con otras versiones. Se puede consultar el archivo README, incluido en el código fuente del programa, para obtener más información sobre los programas auxiliares. Además, se usan otras herramientas (descriptas con detalle en el archivo README), entre ellas se encuentran, por ejemplo, *flex* y *bison*. Para la compilación de la herramienta, se utilizaron las *GNU autotools*. Las versiones de cada programa son las siguientes:

- autoconf version 2.59
- automake version 1.9.6
- m4 version 1.4.4
- libtool version 1.5.22
- flex version 2.5.33
- bison version 2.1
- ltl2dstar version 0.4.2
- Spin version 4.2.6

Es posible, pero no ha sido probado, que se pueda compilar exitosamente con otras versiones de las herramientas mencionadas.

Finalmente, se puede generar documentación del código utilizando el programa *doxygen*. Basta con tipear:

```
$ make docs
```

Esto creará la documentación en el directorio *doc*. Para más detalles, leer el archivo README.

En las siguientes tres secciones explicaremos brevemente qué hacen los programas *Spin*, *ltl2dstar* y *Rapture*

6.1.2. Spin

Este programa, que cuenta con mas de 15 años de desarrollo, es un Model Checker utilizado para la verificación formal the sistemas distribuidos. El sistema se describe en un lenguaje propio (PROMELA, PROcess MEta LAnguage), y las propiedades a verificar en LTL.

Además, este programa puede ser utilizado como un simulador, que ejecuta un camino posible de ejecución, presentando una traza de dicha ejecución al usuario.

SSS Reducer no utiliza directamente a Spin, sino que ltl2dstar lo hace.

6.1.3. ltl2star

ltl2dstar (LTL to deterministic Streett and Rabin automata) es una herramienta para traducir una fórmula escrita en LTL a un autómata determinístico, específicamente, a un autómata de Rabin o a un autómata de Streett.

Esta herramienta implementa la transformación de Safra explicada en la sección 2.4, utilizando técnicas heurísticas y ciertas optimizaciones para para generar autómatas pequeños en la práctica.

6.1.4. Rapture

Rapture es una herramienta que verifica propiedades de alcanzabilidad cuantificadas sobre una cadena de Markov (o SNP). Acepta la descripción del sistema como un conjunto de procesos que luego compone en parelelo. La sintaxis que acepta SSS Reducer es muy similar a la de Rapture.

6.1.5. Organización del código fuente

Una vez descomprimido el archivo *sss_reducer.tar.bz2* se crea el directorio *sss_reducer*. Dentro de este directorio, la estructura es la siguiente:

- directorio *src*: contiene el código fuente de SSS reducer
- directorio *doc*: contiene documentación del código fuente generada con *doxygen*
- directorio *examples*: contiene archivos de ejemplo para utilizar SSS reducer.

6.1.6. Módulos que componen a SSS reducer

El programa se divide en 4 módulos, que se detallan a continuación:

1. Parser: Este módulo se encarga de parsear el archivo de entrada. La información parseada se guarda en una estructura de datos intermedia, que no es el SNP final. Se utilzaron las herramientas *flex* y *bison* para implementar el parser. Hay 2 archivos que implementan el parser: ellos son `aut_flex.ll` y `aut_bison.yy`. El archivo `aut_flex.ll` contiene la descripción de los tokens que componen la gramática soportada. El archivo `aut_bison.yy` contiene la descripción de la gramática soportada por el programa, y las acciones que se ejecutan cuando se parsea la gramática.

2. Capa intermedia o *abstracción del SNP*: Esta capa contiene la información obtenida por el Parser. No es un SNP, sino que contiene una representación (en una estructura de datos conveniente) de los procesos descriptos en el archivo de entrada. Además, provee determinadas funcionalidades, como por ejemplo, la capacidad de componerse en paralelo con otra abstracción para generar una nueva abstracción que representa la composición de las dos originales. En otras palabras, esta capa contiene una *descripción* simbólica (o abstracta) del SNP real. Utilizando esta información se genera el SNP final, que resulta de sincronizar los procesos descriptos en el archivo de entrada con el autómata asociado a la fórmula que se quiere verificar. Esta capa intermedia es llamada la *abstracción* del SNP, está representada en una clase de C++ (*automataAbs*), y está implementada en los archivos `automataAbs.h` y `automataAbs.cpp`, utilizando además los archivos `auxiliar.h`, `expression.h` y `expression.cpp`.

3. Analizador de AR: Como se mencionó anteriormente, el programa SSS Reducer necesita programas externos para funcionar. Uno de ellos es *ltl2dstar*. Este programa, que a su vez necesita al programa *Spin*, es el que genera un automáta de Rabin. Dicho autómata se guarda en un archivo de texto (por defecto, */tmp/out_ltl2dtsar*), que luego es interpretado por este módulo. Las tarea del *analizador de AR* es la de parsear el output generado por *ltl2dstar* y guardarlo en una estructura de datos adecuada, de la cual sea fácil recuperar infomación para el momento de la sincronización entre el autómata de Rabin y el autómata descripto en el archivo de entrada. La implementación de este módulo se encuentra en los archivos `dr_analyzer/dr_analyzer.h` y `dr_analyzer/dr_analyzer.cpp`, y también está implementado en una clase de C++ (*drAnalyzer*).

4. SNP: esta clase (*pns*) implementa un Sistema No determinístico Probabilístico. Además, provee la funcionalidad adecuada para detectar los *conjuntos fuertemente estables* y para generar el ouput adecuado para el programa *rapture*. El SNP está implementado en los archivos `pns/pns.h` y `pns/pns.cpp`.

Además de los módulos descriptos, el programa tiene un *controlador* que es el encargado de coordinar las acciones necesarias para generar el output. El controlador está implementado en el archivo `main.cpp`, y sus tareas son las siguientes:

1. invocar el parser, que lee de *stdin*. Aquí se usa el módulo *Parser*.
2. componer en paralelo todos los procesos descriptos en el archivo de entrada. Aquí se usa el módulo de *abstracción*.
3. invocar al programa *ltl2dstar*, el cual genera el autómata determinístico de Rabin asociado a la fórmula obtenida en el primer paso.
4. utilizar el módulo *analizador de AR* para parsear la información generada por *ltl2dstar*, y guardarla en memoria.
5. sincronizar el autómata de la *abstracción* con el autómata del paso 4, generando así el SNP final.
6. detectar los *conjuntos fuertemente estables* en el SNP resultante.
7. reducir los *conuntos fuertemente estables* a un único estado, el cual será el objetivo a alcanzar.
8. generar el output para el programa *rapture*, escribiendo en *stdout*

La figura 6.1 muestra cómo se relacionan los módulos que componen a SSS reducer. En **negrita** se destacan los módulos descriptos arriba, y las líneas punteadas indican una clase de C++. En el apéndice A se describe la gramática que define la sintaxis del texto de entrada.

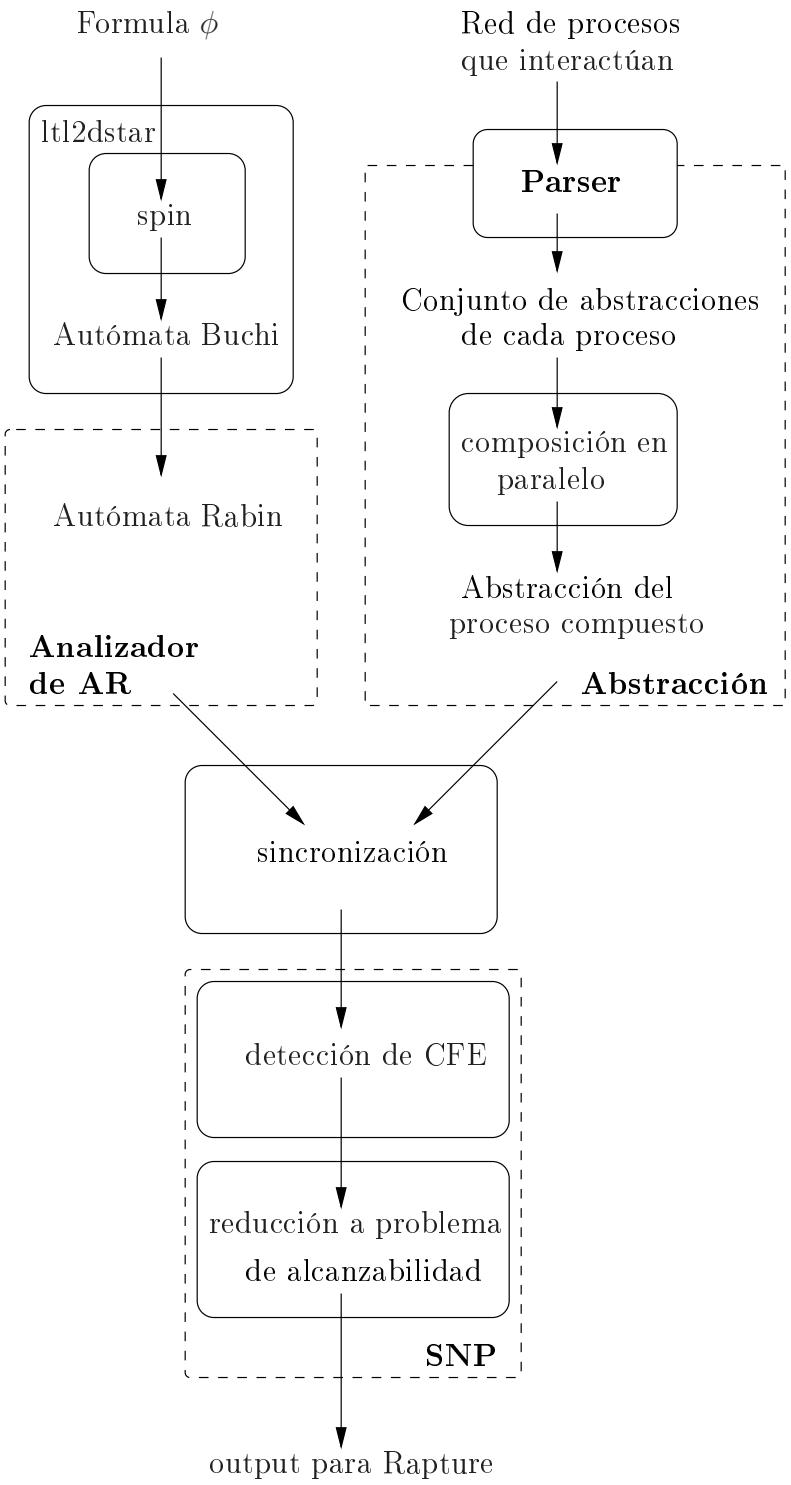


Figura 6.1: Estructura de la herramienta

Capítulo 7

Casos de Estudios

Se realizaron varios casos de estudio para verificar tanto el correcto funcionamiento de SSS Reducer como la factibilidad de utilizarlo en casos realistas. En este capítulo de describen en detalle el resultado de los mismos.

7.1. Caso 1

En esta sección se muestra un ejemplo de la reducción efectuada por SSS reducer sobre un SNP. Este autómata es un ejemplo artificial que tiene como propósito mostrar la transformación que sufre el SNP en las etapas *detección de CFE* y *reducción a problema de alcanzabilidad*. Estas etapas se encuentran al final del procesamiento total de SSS reducer, como puede observarse en la figura 6.1.

El SNP original es el que se muestra en la figura 7.1, y cuenta con el criterio de aceptación $R = \{(E_0, F_0)\}$, donde:

$$\begin{aligned} E_0 &= \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}\} \\ F_0 &= \{s_7, s_8\} \end{aligned}$$

Este SNP es el que resulta de la etapa *sincronización* de la figura 6.1. Se provee junto al código fuente una implementación de dicho SNP; se encuentra en el archivo `src/pns/main.cpp`. Se puede compilar dicho archivo y luego linkear el resultado a la librería `libpns.a` para obtener el ejemplo funcional. Para la compilación puede usarse el siguiente comando:

```
$ cd sss_reducer/src/pns
$ make
$ g++ -I.. main.cpp libpns.a -o test_pns
```

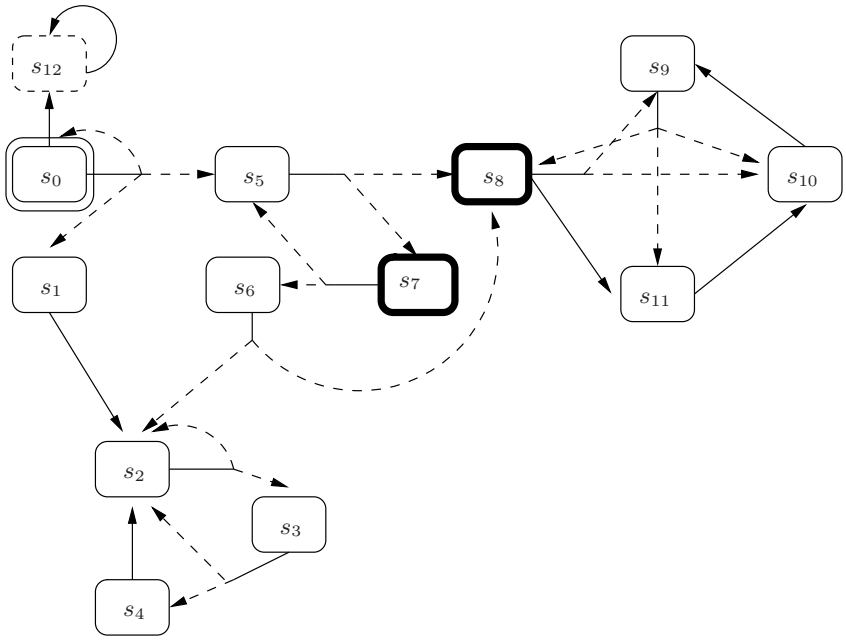


Figura 7.1: SNP a reducir

Luego de efectuar la *detección de CFE*, se obtiene $CFE = (E_0 \cup F_0) - \{s_0\}$, pues desde el estado s_0 es posible transicionar al estado s_{12}, cayendo fuera del conjunto E_0. La figura 7.2 muestra el SNP luego de esta etapa.

Finalmente, SSS reducer aplica la etapa *reducción a problema de alcanzabilidad*, obteniendo el conjunto de estados $\{s_8, s_9, s_{10}, s_{11}\}$, como se muestra en la figura 7.3. Notar que todos los estados que tienen probabilidad mayor que 0 de llegar a s_2 son descartados, dado que una vez alcanzado s_2 es imposible volver a pasar por un estado de F_0.

Este conjunto de estados es, finalmente, "comprimido" en un sólo estado, y se genera el ouput para rapture, como lo muestra la figura 7.4.

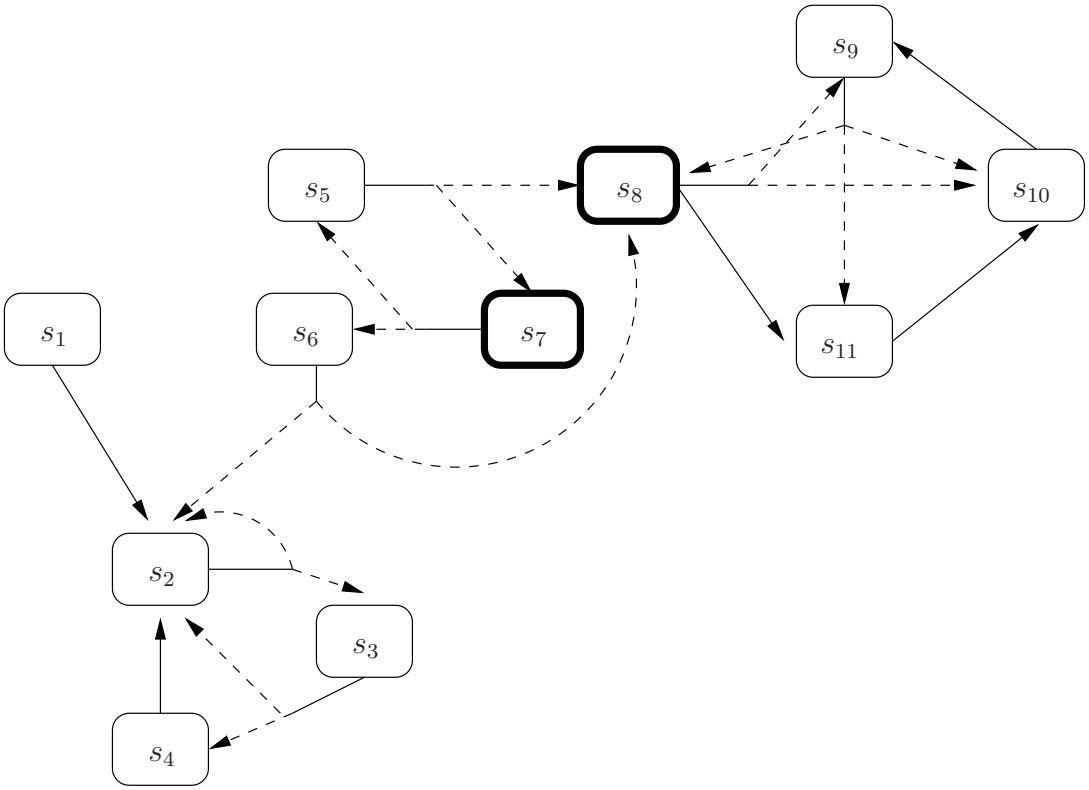


Figura 7.2: SNP luego de aplicar la *generación de los CFE*

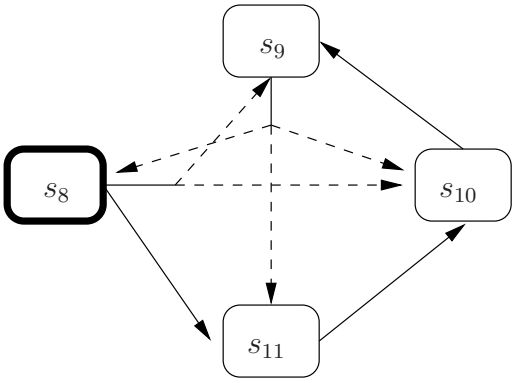


Figura 7.3: CFE con conjunto obligado F_0

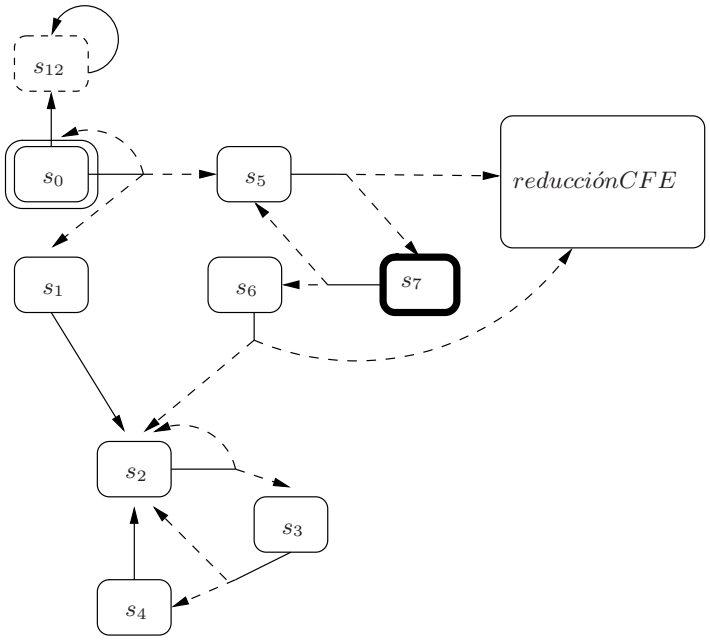


Figura 7.4: Autómata resultante, luego de aplicar la reducción

7.2. Caso 2

En esta sección también se utilizó un SNP artificial, pero esta vez con otro objetivo: el de verificar si la herramienta genera outputs adecuados El SNP utilizado es el de la figura 7.5, y se probaron las siguientes propriedades:

1. $\mathsf{F}\,\mathsf{G}\,(x = 0)$: esta propiedad expresa que desde algún momento en adelante, la variable x valdrá siempre 0.
2. $\mathsf{F}\,(x = 2)$: esta propiedad expresa que en algún momento, x tomará el valor 2.
3. $(x = 0)\ \mathsf{U}\ (x = 2)$: esta propiedad dice que x vale 0 durante varios estados seguidos (pudiendo ser 0) y en algun punto dado, x toma el valor 2.

Notar que todas estas propiedas no se cumplen nunca en el SNP de la figura 7.5, por lo tanto la probabilidad mínima (y en este caso, también la máxima) es 0. Se comprobó que, efectivamente, SSS reducer produce el output esperado, i.e., para cada una de las fórmulas mencionadas, SSS reducer produce un output tal que rapture reporta que la probabilidad mímina es 0. Además se probó a SSS reducer con la negación de cada una de las fórmulas, generando también el output esperado, i.e., la probabilidad máxima (y en

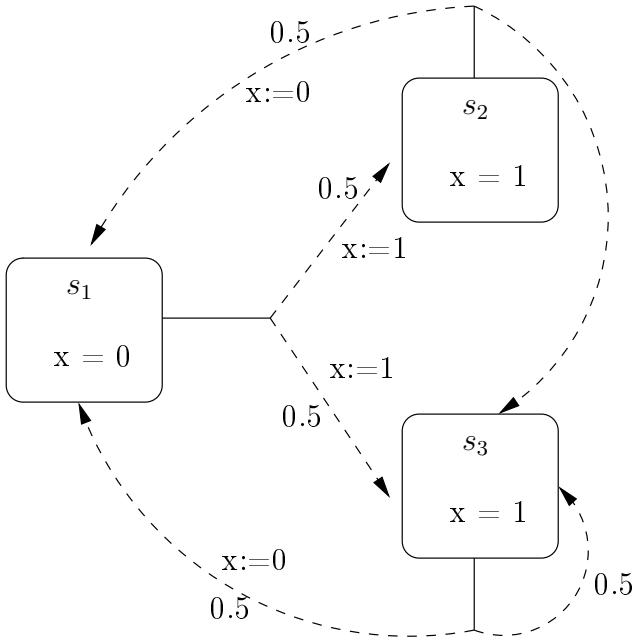


Figura 7.5: SNP para el Caso 2

este caso, también la mínima) es 1. Los ejemplos mencionados anteriormente pueden encontrarse en los archivos `examples/misc/ex0[3-5]/pts`.

Un segundo caso de estudio es el que se muestra en la figura 7.6. La propiedad bajo estudio en este caso es $\mathsf{G}\ ((x = 0) \Rightarrow (\mathsf{F}\ x = 1))$, con probabilidad mínima (y máxima) de que se cumpla igual a $\frac{1}{3}$. Se verificó que SSS Reducer genera el output adecuado. Este ejemplo puede encontrarse en el archivo `examples/misc/ex02.pts` del código fuente.

7.3. Caso 3

Se modeló el protocolo "binary exponential backoff" utilizado en las redes con CSMA/CD para resolver los conflictos generados por colisión. Este protocolo se utiliza actualmente en las redes Ethernet para lograr el acceso al canal, y funciona de la siguiente manera: cuando un host envía un paquete y hay colisión con otro paquete de otro host, el host espera una determinada cantidad de slots de tiempo (elegida al azar) para volver a tratar de acceder al canal. Luego de i colisiones, el host elige la cantidad de slots a esperar entre 0 y $2^i - 1$, es decir, luego de cada colisión se duplica la cantidad de slots para elegir la espera. Además, se establece una cota K, a partir de la cual dejará de duplicarse la cantidad de slots. Este proceso no continúa indefinidamente, sino que luego de R reintentos el host determina que el

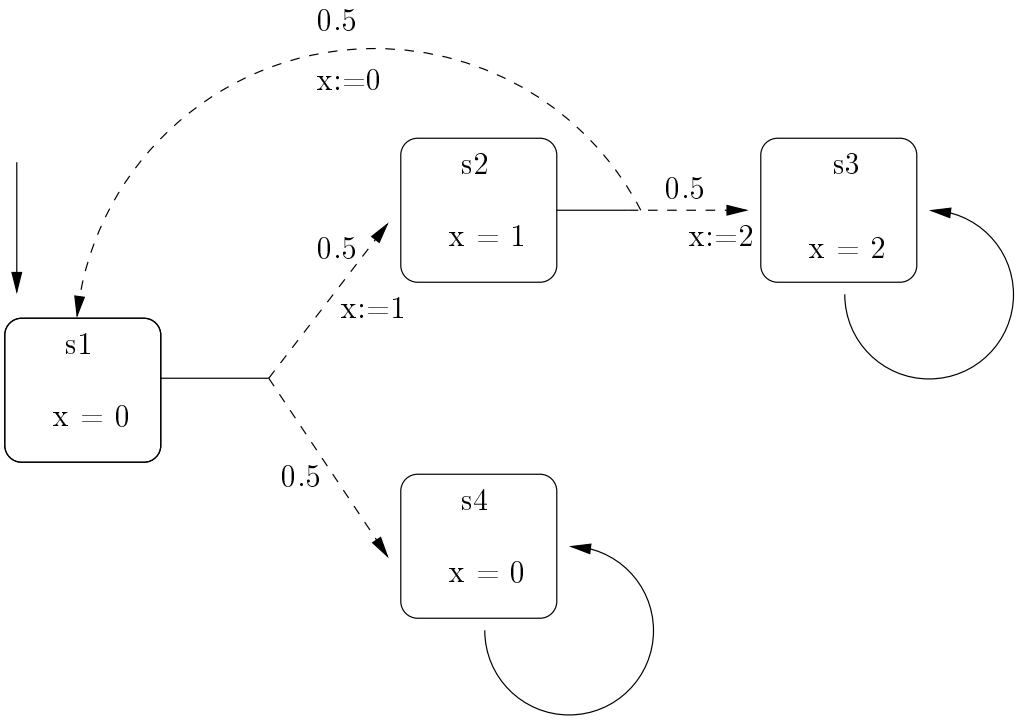


Figura 7.6: Otro SNP para el Caso 2

canal está definitivamente roto y que no se puede acceder a él, abortando la transmisión. Notar que es posible que cuando hay varios hosts tratando de acceder al canal suceda que los hosts determinen, equivocadamente, que el canal está roto.

El problema modelado es con 3 hosts tratando de acceder al canal, que están sincronizados a través de un reloj externo. La propiedad estudiada es:

$$\mathsf{G}\ \ (\ (\neg(p1 \wedge \mathsf{F}\ p2)) \bigwedge (\neg(q1 \wedge \mathsf{F}\ q2)) \bigwedge (\neg(r1 \wedge \mathsf{F}\ r2)))$$

Donde:

- $p1$ = Host 1 o Host 2 abortan, pues creen que el canal no está funcional.
- $p2$ = Host 0 obtiene el acceso al canal.

y el resto de las proposiciones está definido de modo similar.

Intuitivamente, esta propiedad expresa que "no suceda que algún host aborte creyendo que el canal no está funcional, existiendo la posibilidad de que otro host pueda acceder en el futuro". Idealmente, esta probabilidad

debería ser alta. En el cuadro 7.1 se tabulan los resultados obtenidos. Notar que para caso $R = 5$ y $K = 8$ Rapture no terminó de ejecutar.

# H	R	K	pinf	# S	# RS	% red	T	TR
3	2	2	0.8008	4001	2676	33.12	5	6
3	3	2	0.9271	20067	13195	34.25	15	125
3	3	4	0.9572	75091	39218	47.77	66	1648 (27')
3	4	2	0.9752	51745	34217	33.87	43	879 (14')
3	4	4	0.9931	292365	143017	51.08	308	112492 (1.3 ds)
3	4	8	0.9944	863213	325182	62.33	1080	256854 (3 ds)
3	5	2	0.9919	114047	75537	33.77	105	5882 (1.63 hs)
3	5	4	0.9973	765143	363627	52.48	881	207738 (2.4 ds)
3	5	8	?	3447063	1162231	66.28	1813 (30')	?

Cuadro 7.1: Resultados obtenidos

Donde el significado de las columnas es el siguiente:

- # H: la cantidad de hosts que intervienen.
- R: la cantidad de reintentos antes de abortar.
- K: la cota a partir de la cual no se duplicará el rango para elegir el tiempo de espera.
- pinf: la probabilidad mínima buscada.
- # S: la cantidad de estados del modelo, antes de que SSS reducer aplique la *reducción CFE*.
- # RS: la cantidad de estados del modelo, luego de que SSS reducer aplique la *reducción CFE*.
- % red: el porcetanje de reducción de estados que sufrió el modelo original.
- T: el tiempo (en segundos) que tardó SSS reducer.
- RT: el tiempo (en segundos) que tardó Rapture.

Capítulo 8

Conclusiones y trabajos futuros

8.1. Conclusión

Puntualmente en este trabajo se hicieron 2 aportes:

- Se desarrolló un algoritmo para reducir el cálculo de la probabilidad mínima de una propiedad expresada en LTL a un problema de alcanzabilidad.
- Se realizó una herramienta que implementa dicho algoritmo, probándola con varios casos realistas.

Las pruebas realizadas con la herramienta demostraron que la implementación puede ser efectivamente utilizada sobre situaciones reales. En todos los casos, reducir el problema de calcular la probabilidad mínima a un problema de alcanzabilidad fue satisfactorio. Invariablemente el tiempo de CPU y la memoria utilizada se mantuvieron dentro de límites razonables. A pesar de esto, y aunque en este caso no fue necesario, existe la posibilidad de realizar una implementación que optimice la utilización de dichos recursos.

Sin embargo, una vez que SSS reducer generó el output, la herramienta Rapture utilizó un excesivo tiempo de CPU y memoria para completar su tarea. Probablemente esto se debe al modo actualmente utilizado para comunicar ambas herramientas, dado que dicho modo no permite a Rapture efectuar ciertas reducciones en el espacio de estados. Una posible solución a este problema sería la integración del algoritmo implementado en SSS reducer en el programa Rapture, pero para garantizar esto se debería realizar un estudio más profundo de los algoritmos utilizados por este último y de su implementación.

8.2. Trabajos futuros

- Extender la herramienta SSS reducer para que soporte el cálculo de la probabilidad máxima, como se describe en [dA97].

- Integración de SSS reducer con Rapture.
- Extender la herramienta para que soporte otras lógicas temporales (por ejemplo, CTL).
- Extender SSS Reducer para integrarlo con otras herramientas existentes que resuelven el problema de alcanzabilidad, como por ejemplo Prism [KGP04].
- Integrar SSS Reducer con la búsqueda de contraejemplos [And06] en caso de que la propiedad LTL no satisfaga la probabilidad deseada.

Apéndice A

Gramática del archivo de entrada

La gramática soportada por SSS reducer se describe a continuación:

<*input*> ::= <rapture_section> <ltl_section>

rapture_section

<*rapture_section*> ::= <channels> <process>$^+$
<*channels*> ::= 'channel' <names_channel>
<*names_channel*> ::= <name> ',' <names_channel>
| <name> ';'
<*process*> ::= 'process' <name> '{'<sync_decl> <var_decl> <init> <location>$^+$'}'
<*sync_decl*> ::= ϵ
|'sync' <name> (','<name>)* ';'
<*var_decl*> ::= ϵ
|'var' <local_vars>$^+$
<*local_vars*> ::= <name> ':' <var_type> ';'
<*var_type*> ::= 'bool'
| 'uint' '(' <integer> ')'
| 'sint' '(' <integer> ')'
<*init*> ::= 'init' '#' <name> <initial_assign>
<*initial_assign*> ::= ';'
| (<name> ':=' <expression> ';')$^+$
<*location*> ::= 'loc' <name> ':' <transition>*
<*transition*> ::= 'when' <expression> <sync_label> 'goto' <destination> ';'
<*sync_label*> ::= ϵ
| 'sync' <sync_label_name>
<*destination*> ::= <unique branch>
| '{' <branch list> '}'

<*unique branch*> ::= <name> <action>
<*branch list*> ::= | (<branch> ';')*
<*branch*> ::= <name> <probability> <action>
<*action*> ::= ϵ
|'assign' '{' <assign list> '}'
<*assign list*> ::= <assign> ';' <assign list>
| <assign> ';'
| <assign>
<*assign*> ::= <name> ':=' <expression>
<*expression*> ::= <constant>
| <variable_ref>
| <unary_ expr>
| <binary_ expr>
| '(' <expression> ')'
<*constant*> ::= 'true' | 'false' | <integer>
<*variable_ref*> ::= <name>
<*unary_ expr*> ::= '$' <expression> (Notar que $ es el menos unario)
| 'not' <expression>
<*binary_ expr*> ::= <bool_expr>
| <expression> + <expression>
| <expression> - <expression>
| <expression> * <expression>
<*bool_ expr*> ::= <expression> '=>' <expression>
| <expression> 'or' <expression>
| <expression> 'xor' <expression>
| <expression> 'and' <expression>
| <expression> '<=>' <expression>
| <expression> '<>' <expression>
| <expression> '=' <expression>
| <expression> '>=' <expression>
| <expression> '>' <expression>
| <expression> '<=' <expression>
| <expression> '<' <expression>
<*name*> ::= <identifier>
<*sync_ label_ name*> ::= <identifier>

ltl_section

<*ltl_ section*> ::= 'prop' '{' <list_prop>* '}' <ltl_formula> ';'
<*list_ prop*> ::= <name> ':=' <bool_expr> ';'
| <name> ':=' 'true' ';'
| <name> ':=' 'false' ';'
| <name> ':=' <variable_ref> ';'

<*ltl_formula*> ::= '(' <ltl_formula>')'
| <state_formula>
| <seq_formula>
| <ltl_formula> 'and' <ltl_formula>
| <ltl_formula> 'or' <ltl_formula>
| 'not' <ltl_formula>
| <ltl_formula> '=>' <ltl_formula>
| <ltl_formula> '<=>' <ltl_formula>
| <ltl_formula> 'xor' <ltl_formula>

<*seq_formula*> ::= 'X' <ltl_formula>
| <ltl_formula> 'U' <ltl_formula>
| <ltl_formula> 'R' <ltl_formula>
| 'F' <ltl_formula>
| 'G' <ltl_formula>

<*state_formula*> ::= <name>

terminales

<*identifier*> ::= string que no comienza con un número
<*integer*> ::= número entero no negativo
<*probability*> ::= número de punto flotante no negativo

Apéndice B

Estructura del código fuente

La estructura de directorios del código es la siguiente:

```
.
|-- autom4te.cache
|-- doc
|   |-- html
|   `-- latex
|-- examples
|   |-- BinaryExpponentialBackOff
|   `-- misc
|-- src
|   |-- dr_analyzer
|   `-- pns
|-- textos
`-- utils
```

El contenido de cada directorio es el siguiente:

- `autom4te.cache`: este directorio es generado automáticamente por las *autotools*.
- `doc`: en este directorio se generará la documentación cuando se ejecute el comando `'make docs'` desde el directorio raíz del código fuente. La documentación es generada con *doxygen*. Por defecto, la documetación será creada en formatos HTML y LaTeX. Para navegar por la documentación HTML, basta con utilizar un navegador web y abrir el archivo `doc/html/index.html`. En el caso de la documentación en formato LaTeX, se deberá ejecutar el comando `'make'` dentro del directorio `doc/latex/`. Luego, basta con abrir el archivo `refman.dvi`. Es posible modificar el archivo `Doxyfile` (ubicado en el directorio raíz del código fuente) para personalizar las opciones de generación de documentación.

- `examples`: contiene algunos ejemplos para probar SSS Reducer. Particularmente útiles son los ejemplos de `examples/misc`. Además, están los archivos utilizados para generar los resultados del capítulo 7.
- `src`: Aquí se encuentra todo el código fuente de SSS Reducer. En `src/pns` está la implementacion del SNP, y en `src/dr_analyzer` se encuentra la implementación del Analizador de AR.
- `textos`: aquí está el código que generó este documento. Basta con ejecutar el comando `'latex tesis.tex'` dentro de este directorio.
- `utils`: contiene scripts con diversas funcionalidades generales.

Ciertos archivos merecen especial atención. Ellos son:

- `Makefile.am`: este archivo está presente en el directorio raíz del código fuente. Utilizando *automake* se generará automáticamente cierto archivo (*Makefile.in*) necesario para la compilación de SSS Reducer. También hay un archivo `Makefile.am` dentro de los directorios `src`, `src/pns` y `src/dr_analyzer`. Este archivo controla ciertas opciones de compilación, como las opciones que se le pasarán a g++.
- `configure.in`: ubicado en el directorio raíz del código, este archivo es utilizado por *autoconf* para generar el script llamado *configure*.

Compilación de SSS Reducer

SSS Reducer se distribuye con el script *configure* incluído. Es decir, no es necesario que el sistema en donde se desee utilizar SSS Reducer tenga instaladas las *autotools*. Para compilar SSS Reducer, se deben ejecutar los siguientes comandos:

```
$ ./configure
$ make
```

Esto creará el archivo `src/sss_reducer`. Opcionalmente, `'make install'` instalará SSS Reducer en el sistema. Se puede consultar el archivo INSTALL para ver opciones de configuración.

Para limpiar los archivos creados durante la compilación, se debe ejecutar el comando `'make clean'`. Existe también la posibilidad de hacer una limpieza más profunda (por ejemplo, borra la documentacion autogenerada), mediante el comando `'make distclean'`. En caso de hacer una limpieza profunda, se deberá contar con las *autotools* de GNU para volver a generar el script *configure*.

Bibliografía

[And06] Miguel Andrés. Derivación de contraejemplos para model checking cuantitativo. Master's thesis, Fa.M.A.F. - U.N.C., 2006.

[dA97] Luca de Alfaro. Temporal logics for the specification of performance and reliability, 1997.

[dAB] Luca de Alfaro and Andrea Bianco. Model checking of probabilistic and nondeterministics systems.

[JDL02] 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.

[JKK66] J.L Snell J.G. Kemeny and A.W. Knapp. Denumerable markov chains, 1966.

[Kat99] Joost-Pieter Katoen. *Concepts, algorithms, and tools for model checking.* Herstellung, 1999.

[KGP04] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In *International Journal on Software Tools for Technology Transfer*, volume 6, pages 128–142, Providence, RI, USA, 2004.

[Lőd05] Christof Lőding. Methods for the transformation of the ω-automaton: Complexity and connection to the second order logic. Master's thesis, Institute of Computer Science and Aplied Mathematics Christian-Albrechts-Univesity of Kiel, 2005.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th IEEE Symp. Foundation of Computer Science (FOCS '77)*, pages 46–57, Providence, RI, USA, Oct.-Nov. 1977.

[Saf89] Shmuel Safra. *Complexity of Automata on Infinite Objects.* PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1989.