Córdoba, 20 de Marzo de 2014

CLOUSEAU: Verificación de Propiedades de Seguridad en Protocolos Distribuidos con Probabilidades

Autor: Pedro Eduardo Waquim
Directores: Pedro D'Argenio
Silvia Pelozo

UNIVERSIDAD NACIONAL DE CÓRDOBA
Facultad de Matemática, Astronomía y Física
Licenciatura en Ciencias de la Computación

Resumen

Presentamos una herramienta para analizar propiedades de seguridad en protocolos distribuidos. La herramienta está construida sobre los schedulers llamados fuertemente distribuidos, donde el secreto también es considerado. El secreto es presentado como una clase de equivalencia sobre acciones que las componentes no tienen acceso al él; sin embargo estas acciones pueden ser distinguidas por los que tienen autorización apropiada.

También presentamos un algoritmo para resolver análisis de alcanzabilidad sobre estos tipos de modelo. El algoritmo codifica apropiadamente el modelo no determinista interpretando las decisiones de los schedulers como parámetros. El problema está en reducirlo a un problema de optimización polinomial.

Clasificación: C.2.0 General (Security and Protection), C.2.4 Sistemas Distribuidos, D.2.4 Software/Program Verification (Model Checking).

Palabras Clave: model checking, sistemas distribuidos, propiedades de alcanzabilidad, modelos probabilistas, modelos no deterministas.

Agradecimientos

Quiero dedicarle todo mi agradecimiento a las personas que estuvieron conmigo a lo largo de este trayecto. Principalmente agradecer a mi familia que me sostuvo con mucho esfuerzo para que pudiera realizar mis metas. También a mis profesores que sin dudarlo me brindaron más de lo que pudiera pedir. A mis compañeros de clase que siempre estuvieron dispuestos a dar una mano. Y por último a Dios, que estuvo conmigo en cada hora de estudio, parcial u examen, obrando de infinitas maneras para formarme como persona. Una vez más les agradezco de corazón a todos lo que tocaron mi vida y me animaron en esta lucha. Espero ardientemente que este trabajo dé fruto y sirva para muchas personas.

Índice general

Capítulo 1

Motivación

Una mirada a nuestro alrededor sería suficiente para saber que la digitalización y la automatización son la clave no solo del futuro sino también del presente. Las computadoras han ido ocupando un espacio cada vez más importante en nuestra vida cotidiana y sobre todo en nuestro trabajo. Esto se pone de manifiesto cuando una empresa experimenta una caída del sistema. Todo el trabajo o al menos una gran parte de él, se detiene por completo. Pero como el ser humano está en constante comunicación, una computadora aislada no reporta un gran beneficio. Así es como, por la necesidad de estar comunicados, fueron surgiendo los *sistemas distribuidos*.

Existen innumerables aplicaciones para estos sistemas, incluso muchos de ellos son gratuitos, pero sin duda podemos notar una gran desconfianza por parte de los usuarios a la hora de utilizar estos productos. Los usuarios necesitan garantías sobre la seguridad de estas aplicaciones así como la protección de su información privada.

En numerosos ámbitos de la vida cotidiana como cuentas bancarias, antecedentes policiales, historias clínicas, etc; nuestra información personal se encuentra publicada para un grupo acotadas de personas: gerentes del banco, personal policial, médico de cabecera. La gran mayoría de estas instituciones utilizan sistemas distribuidos. Nuestra preocupación consiste de que la información no se filtre, es decir, que algún individuo ajeno a la institución pueda acceder a dicha información publicada en sus bases de datos. Esto puede dar lugar a injusticias tales como no otorgar empleo a una persona solo por tener antecedentes policiales, o discriminar a una persona por padecer algún tipo de enfermedad.

Aunque el testing está destinado a eliminar todas estas fallas o sucesos imprevistos, tenemos la experiencia de casos que han pasado controles de calidad exitosamente y luego han resultado en fallas severas. Por esto nos vemos en la necesidad de tener herramientas de verificación formal, en donde el resultado de verificar una propiedad sea una garantía de que no existen fallas en el modelo. Esta técnica se denomina *model checking*.

1.1. Objetivos

Esta tesis apunta a la creación de una herramienta capaz de estudiar mediante el *model checking*, el comportamiento de aplicaciones sobre sistemas distribuidos y verificar propiedades de seguridad sobre ellos, encontrando patrones no tenidos en cuenta a la hora de su creación. Nuestro enfoque consiste en modelar sistemas distribuidos en un lenguaje sencillo que Clouseau (nombre que le dimos a nuestra herramienta) requiere para su análisis y Clouseau reduce el problema de seguridad en un problema de optimización polinomial en breves lapsos de tiempo. El poder de trabajar con modelos no deterministas es lo que hace a Clouseau una herramienta atractiva y versátil para operar sobre estos sistemas. Clouseau utilizó como backend diversas herramientas capaces de atacar este problema de optimización lineal, incluyendo SMT solvers como z3 [2] y otras herramientas que permiten soluciones analíticas como redlog [3].

1.2. Organización de este Trabajo

El resto de la tesis está estructurada de la siguiente manera:

- En el Capítulo 2 se presentan los fundamentos teóricos para modelar un sistema distribuido.
- En el Capítulo 3 introduciremos herramientas necesarias para la construcción del árbol de ejecuciones, así como también nociones de equivalencia y schedulers distribuidos.
- En el Capítulo 4 introduciremos un algoritmo que nos permitirá reducir el problema de seguridad en un problema de optimización polinomial.
- En el Capítulo 5 presentamos la sintaxis del lenguaje en que se expresará el modelo, junto con las consideraciones y reglas del lenguaje a tener en cuenta.
- En el Capítulo 6 detallaremos cualidades de la herramienta. También explicaremos sus procedimientos y su composición.
- En el Capítulo 7 analizaremos un ejemplo clásico de la criptografía mediante la herramienta Clouseau y corroboraremos los resultados obtenidos.

Capítulo 2

Análisis de Seguridad en Sistemas Distribuidos con Probabilidad

"El que busca la verdad
corre el riesgo de encontrarla".

Manuel Vicent, 1936.

Todos sabemos la importancia de la seguridad en nuestros días, pero para poder decir que un sistema es seguro se requiere un análisis riguroso que lo garantice. Clouseau ejecuta dicho análisis sobre sistemas distribuidos. Nuestro enfoque es que una vez analizado el Sistema, no exista ninguna secuencia posible que beneficie indebidamente a alguna de las partes.

Antes que nada, no se puede analizar algo que no se conoce, por lo tanto veremos qué es un sistema distribuido, como está compuesto y que técnicas usaremos para analizarlo.

2.1. Prototipos

Definimos un *sistema distribuido* como aquel en el que los componentes hardware o software, localizados en computadores unidos mediante red, comunican y coordinan sus acciones sólo mediante paso de mensaje [1]. Llamaremos componentes a cada uno de estos computadores unidos mediante red.

Para realizar cualquier análisis de seguridad sobre estos sistemas, deberemos primero crear un modelo que nos permita reproducir el funcionamiento de los mismos. Si queremos modelar un sistema distribuido, debemos modelar cada una de las componentes del sistema. El modelo completo es construido componiendo los submodelos, considerando posibles interacciones entre las componentes y todos los posibles interleaving. Se entiende por *interleaving* a la abstracción de la concurrencia mediante el in-

tercalado de las acciones de las componentes. Cada componente es descrita como una instancia de un prototipo. El comportamiento del sistema es modelado como la ejecución concurrente de estas instancias.

Definición 2.1.1. *Un prototipo P es una 4-upla* (name,Param,init,T)*, donde* ***name*** *es el nombre del prototipo,* ***Param*** *es el conjunto de parámetros del prototipo,* ***init*** *es una distribución inicial y* $T = T^R \uplus T^a$ *es la unión disjunta de los conjuntos de transiciones reactivas y transiciones activas.*

Un prototipo es la representación abstracta de una componente en un sistema distribuido. Si hacemos la comparación con un Lenguaje Orientado a Objetos (LOO), podría decirse que el prototipo cumple la función de una clase y las componentes son análogas a los objetos instanciados por dicha clase. Al igual que los objetos pueden ejecutar métodos, las instancias tienen la capacidad de ejecutar acciones (transiciones). También los prototipos contienen un conjunto de parámetros, tales como nombres de canales y constantes que constituyen su interfaz. La distribución inicial define la probabilidad de cada estado inicial posible de la componente.

2.1.1. Parámetros

En una comunicación es crucial tener conocimiento de cómo y por dónde un dispositivo envía y recibe información. A este conocimiento se lo llama *interfaz*. Esto lo modelamos con parámetros, entre ellos distinguimos: los *nombres de canales* y las *constantes*. Los *nombres de canales* (ChanNames) cumplen la función de establecer el flujo de información del prototipo hacia el exterior y viceversa. Existen dos tipos de ChanNames:

- *OutNames*: Nombres de canales de salida denotados con (!).
- *InNames*: Nombres de canales de entrada denotados con (?).

Muchas veces necesitamos que múltiples instancias de un prototipo posean distintos valores propios, por ejemplo, una clave de encriptado distinta para cada instancia. Para solucionar esto, utilizaremos *constantes*. Las *constantes* ($Const$) son parámetros cuyo valor se define en el momento de instanciar un prototipo y no se altera en el transcurso de la ejecución. Los *valores* (Val) en nuestra herramienta serán tomados de elementos pertenecientes al conjunto de los enteros (Int).

2.1.2. Distribución

Todo sistema parte de un punto inicial, transcurre su ejecución y termina. Si durante el transcurso de dicha ejecución no se efectuaron cambios en el sistema, entonces se dice que la ejecución es ociosa. Al contrario si se efectuaron cambios y es normal

que así suceda, decimos que el sistema contiene múltiples estados. Un *estado* es el conjunto de valores que poseen las variables de una componente en un momento dado. Entendemos por variables a aquellas cuyos valores cambian durante el transcurso de la ejecución. Un *estado* es distinto de otro para una componente si posee distintos valores en c/u para una misma variable.

Ahora con la noción de estados podremos introducir el concepto de una distribución. Antes notar que trabajamos con sistemas probabilísticos y que podemos llegar a un punto en el sistema en que un estado sea elegido aleatoriamente con una probabilidad determinada. Por eso, una *distribución* es la elección probabilística del estado próximo para una componente.

Definición 2.1.2. *Sea Var un conjunto de variables. Sea Exp_{Var} un conjunto de expresiones sobre las variables en Var. Con $v' = exp$ denotamos que el nuevo valor de la variable v (indicado como v') estará dado por el resultado de evaluar exp con los valores actuales de las variables en Var. Cuando escribimos $\{v'_1 = exp_1, v'_2 = exp_2, ...\}_{v_i \in Var}$ representamos la asignación simultánea de los nuevos valores de las variables v_i de acuerdo a las expresiones exp_i evaluadas en los valores actuales de las variables en Var. Sea $P_i \in (0, 1]$ para todo i, $1 \leq i \leq n$, tal que $\sum_{i=1}^{n} P_i = 1$, luego:*

$$\sum_{i=1}^{n} P_i : \{v'_1 = exp_1^i, v'_2 = exp_2^i, ...\}_{v_j \in Var}$$

denota una distribución sobre la asignación de nuevos valores a las variables, donde cada variable v_i tomará un nuevo valor de acuerdo a exp_j^i con probabilidad P_i.

En nuestro modelo las distribuciones aparecen en dos ocasiones: como *distribución inicial*, o como consecuencia de una *transición*. La distribución inicial selecciona de forma probabilista el estado inicial. En cambio, las otras seleccionan el estado próximo luego de una transición.

2.1.3. Expresiones

Dijimos que las variables almacenan valores. Es muy común que un valor dependa de una función ($y = x + 2$). Las expresiones nos permitirán (valga la redundancia) expresar dichos valores de una forma adecuada. En numerosas oportunidades, los usuarios cometen errores al formar una expresión o incluso ingresar algo que no es una expresión propia. Al evaluarlas, el compilador es incapaz de procesarlas e introduce un fallo. Para evitar esto introduciremos un nuevo símbolo $\perp$ llamado *bottom*, que representará una expresión mal formada.

Las expresiones Exp_V están definidas sobre símbolos $v \in V$, $k \in Exp_V \setminus \{\perp\}$:

$$\begin{aligned} BoolExp_V &::= true \mid Exp_V = Exp_V \mid \neg BoolExp_V \mid BoolExp_V \wedge BoolExp_V \\ Exp_V &::= v \mid (Exp_V, Exp_V) \mid \pi_1(Exp_V) \mid \pi_2(Exp_V) \\ &\quad \mid enc(Exp_V, k) \mid dec(Exp_V, k') \end{aligned}$$

Como se sabe ***false*** es sinónimo de $\neg$***true*** y los operadores lógicos $\vee, \neq$ pueden ser derivados. Aquí podemos ver que el conjunto V es un parámetro de las expresiones; en el caso de la distribución inicial, V tomara el valor de $\{Const \cup Int\}$. En definitiva:

- v es un valor del conjunto V.
- (Exp_V, Exp_V) es una par de expresiones de V.
- $\pi_1\,(Exp_V)$ es la proyección de la 1er expresión de la tupla Exp_V.
- $\pi_2\,(Exp_V)$ es la proyección de la 2da expresión de la tupla Exp_V.
- $enc(Exp_V, k)$ representa el encriptado de la expresión Exp_V con la clave k.
- $dec(Exp_V, k')$ representa el desencriptado de la encriptación Exp_V.

2.1.4. Transiciones

Una transición representa un cambio de estado en una componente. Tal cambio de estado se puede producir por iniciativa de la componente, ya sea enviando un mensaje a sus pares o simplemente por una acción interna de dicha componente, o puede producirse como respuesta al entorno aceptando un mensaje de entrada a través de uno de sus canales. En el primer caso decimos que la transición es *activa*, y en el segundo caso decimos que es *reactiva*.

De una u otra manera, al realizarse la transición se produce un cambio de estado que es probabilista. Esto es, el nuevo estado se seleccionará aleatoriamente de acuerdo a una distribución de probabilidad dada e indicada como se definió en la sección 2.1.2

Definición 2.1.3. *Una transición se define como una 4-upla (L,guard,Dist,Stop) donde:*

- ***L*** *es un conjunto de etiquetas* $L = (ChanNames \times Exp_V) \cup \{\tau\}$ *cuyos elementos se denotan de la siguiente forma:*
  - $c!exp \in OutNames \times Exp_V$ *(Transición de salida)*
  - $c?msg \in InNames \times Exp_V$ *(Transición de entrada)*
  - τ *(Acción silenciosa o interna)*

  con $\{OutNames, InNames\} \in ChanNames$
- ***guard*** *es una expresión booleana en* $BoolExp_V$.
- ***Dist*** *es una distribución de la forma:*
  $\sum_i p_i : \{v'_1 = exp^i_1, v'_2 = exp^i_2, \dots\}_{v_j \in Var}$ *con* $exp^i_j \in Exp_V$ *y* $(\sum_i p_i) = 1$.
- ***Stop*** *es un* bandera *que puede tomar valor* ***true*** *o* ***false***.

En el caso de las transiciones reactivas, tenemos que V (recordar que V es el conjunto que tomarán como parámetro las expresiones) tomará el valor de $\{Const \cup Int \cup Var \cup \{msg\}\}$, siendo ***msg*** una palabra reservada que será vista como una variable de Var que tomará un valor de Val. Val es el conjunto de valores concretos que toma una expresión Exp_V. El valor que tome la variable ***msg***, será el valor recibido por el canal.

En el caso de las transiciones activas, tenemos que V tomará el valor de $\{Const \cup Int \cup Var\}$. Si la transición es de salida, el valor de la expresión que aparece en la etiqueta junto con el nombre del canal será mandado por dicho canal. Si la transición es del tipo interna, solo se realizarán los cambios de estados acorde a la distribución.

Se dice que una transición está ***habilitada*** si su guarda se satisface en la valuación actual del estado actual. Solo las transiciones habilitadas pueden ser ejecutadas por la componente. Las componentes pueden arbitrariamente ejecutar cualquier transición *activa* habilitada, pero para ejecutar una transición *reactiva* deben suceder dos cosas: que la transición se encuentre habilitada y que otra componente haya mandado algún valor por el mismo canal.

Nuestro modelo requiere también la propiedad *input enabled*, lo cual significa que las componentes reaccionen ante todos los mensajes recibidos por cada uno de sus canales de entrada. Para asegurar esto, vamos a suponer que para cada canal de entrada de las componentes tenemos una transición con una guarda de la siguiente forma:

$$\neg \bigvee \text{guard}_{t \in T^R | Chan}$$

y que con probabilidad 1 no cambia el valor de ninguna variable. Esto nos garantiza que en todo momento habrá una transición de entrada habilitada para cada canal. Por otra parte, si nuestra transición contiene el valor *true* en el campo de ***Stop***, significa que luego de ejecutarse, nuestra componente quedará con todas las transiciones activas deshabilitadas hasta el final de la ejecución del modelo.

2.2. Instancias

Definición 2.2.1. *Una instancia de un prototipo define una componente concreta. Dado un prototipo P, una instancia de P se define como una substitución sintáctica:*

$$P[const_1, \ldots, const_{\#Const}, ichan_1?, \ldots, ichan?_{\#InNames}, ochan_1!, \ldots, ochan!_{\#OutNames} / c_1, c_2, \ldots, c_{\#Const}, in_1, in_2, \ldots, in_{\#InNames}, out_1, out_2, \ldots, out_{\#OutNames}]$$

con constantes $c_1, c_2, \ldots, c_{\#Const} \in Val$ *y canales* $in_1, in_2, \ldots, in_{\#InNames} \in Chan^I, out_1, out_2, \ldots, out_{\#OutNames} \in Chan^O$ $(Chan^I \cap Chan^O = \emptyset)$.

Se puede observar que si tenemos varias instancias de un mismo prototipo, significa que tendremos varias componentes que se comportarán de la misma forma, pero cada una podrá tener canales y constantes distintas entre sí.

La diferencia entre ChanNames y Chan es que ChanNames es el conjunto de nombres con el cual los canales están declarados dentro del prototipo y Channels es el conjunto de canales reales que utilizarán las instancias del prototipo.

2.2.1. Interpretación de Expresiones

Las expresiones representan valores. En el caso de las asignaciones de variables, estas expresiones se evalúan y se guardan en las variables. Diremos que una valuación es el mapeo de una expresión a un valor. A continuación notarán un valor $E(v, k, r)$ un tanto extraño. Este valor es la encriptación del valor v con la clave k y la sal r. La sal nunca se repite o se repite luego de un periodo extenso, esto sirve para obtener distintos valores de encriptación, aún si encriptamos el mismo valor con la misma clave repetidas veces. De esta manera, un valor encriptado será sólo una sucesión de dígitos incomprensibles para cualquier posible atacante.

Mas formalmente, dada una valuación $\xi : V \to Val$, las expresiones se interpretan de acuerdo al mapeo $[\![\cdot]\!]_\xi : Exp_V \to Val$ definido como:

$$
\begin{array}{ll}
[\![e]\!]_\xi = e & \text{if } e \in Const \\
[\![e]\!]_\xi = \xi\,(e) & \text{if } e \in Dom\,(\xi) \\
[\![(e_1, e_2)]\!]_\xi = \left([\![e_1]\!]_\xi, [\![e_2]\!]_\xi\right) & \text{if } [\![e_1]\!]_\xi \neq \bot \text{ and } [\![e_2]\!]_\xi \neq \bot \\
[\![\pi_1\,(e_1, e_2)]\!]_\xi = [\![e_1]\!]_\xi & \\
[\![\pi_2\,(e_1, e_2)]\!]_\xi = [\![e_2]\!]_\xi & \\
[\![enc\,(e_1, e_2)]\!]_\xi = E\left([\![e_1]\!]_\xi, [\![e_2]\!]_\xi, r\right) & \text{if } [\![e_2]\!]_\xi \neq \bot \text{ y } r \text{ es un valor fresco global} \\
[\![dec\,(e_1, e_2)]\!]_\xi = v & \text{if } [\![e_1]\!]_\xi = E\left(v, [\![e_2]\!]_\xi, r\right) \text{ and } [\![e_2]\!]_\xi \neq \bot \\
[\![e]\!]_\xi = \bot & \text{en cualquier otro caso.}
\end{array}
$$

y las expresiones booleanas se interpretan de acuerdo al mapeo (el overloading es inocuo) $[\![\cdot]\!]_\xi : BoolExp_V \to \{true, false\}$ definido como:

$$
\begin{array}{ll}
[\![b]\!]_\xi = b & \text{if } b \in \{true, false\} \\
[\![\neg b]\!]_\xi = \neg [\![b]\!]_\xi & \\
[\![b_1 \wedge b_2]\!]_\xi = [\![b_1]\!]_\xi \wedge [\![b_2]\!]_\xi & \\
[\![e_1 = e_2]\!]_\xi = true & \text{if } [\![e_1]\!]_\xi = [\![e_2]\!]_\xi \\
[\![e_1 = e_2]\!]_\xi = false & \text{if } [\![e_1]\!]_\xi \neq [\![e_2]\!]_\xi
\end{array}
$$

2.3. Propiedades

Una propiedad representa la premisa de seguridad que se desea analizar sobre el modelo. Clouseau verifica que se haga justicia sobre una propiedad específica, ya que sería bastante caro e ineficiente analizar todas las premisas posibles. Un ejemplo sería que todas las componentes posean la misma información acerca de sus pares. Otra propiedad posible sería que una componente sea incapaz de averiguar un valor determinado. Tenemos dos formas de modelar nuestra propiedad a probar:

- Usando *anonimidad*: Cuando la propiedad a probar tiene la característica de que una componente acierte más de lo debido sobre una elección binaria, se puede formular de una manera especial.

$$P(secret = x | guess = x) \leq P(secret = x) \quad \forall\, secreto\ x \tag{2.1}$$

- *Alcanzabilidad probabilista*: La probabilidad de llegar a un estado con ciertas características sea menor o igual a la probabilidad esperada.

$$P(y = 2) \leq 0{,}5 \tag{2.2}$$

Cuando tratamos con propiedades del tipo *alcanzabilidad probabilista*, Clouseau solo nos pedirá una guarda booleana tal que pueda calcular la probabilidad de llegar a un estado que la satisface, y ver si esa probabilidad es mayor, menor o igual a la esperada. En el caso de la ecuación anterior, la guarda a ingresar sería: $y = 2$. Cuando nuestra propiedad sea del tipo *anonimidad*, la trataremos de forma diferente. En el capítulo 5 entraremos más en detalle sobre estos aspectos.

2.4. Modelo Formal

En la sección 2.1 dijimos que una componente es una instancia de una 4-upla (name, Param, init, T). Esta definición es válida para entender como está compuesta, pero nos resulta algo engorroso a la hora de explicar su funcionamiento. Para esto utilizaremos una variante de autómatas probabilistas. Cada instancia de prototipo induce un *modelo interactivo probabilista* que es una tupla $(S, L, init, \rightarrow)$ donde cada componente se define como sigue:

- S es un conjunto de estados, donde cada $s \in S$ es una valuación $s : Var \rightarrow Val$.

- $L = (Chan \times Val) \cup \{\tau\}$ es un conjunto de etiquetas donde $Chan = Chan_I \cup Chan_O$. En particular, distinguimos con $L^I = Chan_I \times Val$ y $L^O = \{Chan_O \times Val\} \cup \{\tau\}$ a los conjuntos de etiquetas que representan las entradas y salidas respectivamente. Denotamos con $c.v$ a los elementos $(c,\ v) \in L$.

- $init$ es la distribución inicial, $init\,(s) = \sum_i p_i\, \chi_{\left\{v_j \mapsto [\![exp_j^i]\!]_\emptyset\right\}_{v_j \in Var}}(s)$

- $\rightarrow \subseteq S \times L \times Dist\,(S)$ es la relación de transición, (s, l, μ) de $\rightarrow$ se denota $s \xrightarrow{l} \mu$, que se define con las siguientes reglas:

$$\frac{[c?m]guard \to \sum_i p_i : \{v_1' = exp_1^i, v_2' = exp_2^i, \dots\}_{v_j \in Var} \in T \qquad [\![guard]\!]_{s \cup \{m \mapsto v\}} \qquad \forall s' : \mu(s') = \sum_i p_i \chi_{\left\{v_j \mapsto [\![exp_j^i]\!]_{s \cup \{m \mapsto v\}}\right\}_{v_j \in Var}} \qquad v \in Val}{s \xrightarrow{c.v} \mu} \tag{2.3}$$

$$\frac{[c!o]guard \to \sum_i p_i : \{v_1' = exp_1^i, v_2' = exp_2^i, \dots\}_{v_j \in Var} \in T \qquad \neg s(\textbf{\textit{stop}}) \qquad [\![guard]\!]_s \qquad [\![o]\!]_s = v \qquad \forall s' : \mu(s') = \sum_i p_i \chi_{\left\{v_j \mapsto [\![exp_j^i]\!]_s\right\}_{v_j \in Var}}}{s \xrightarrow{c.v} \mu} \tag{2.4}$$

$$\frac{[tau]guard \to \sum_i p_i : \{v_1' = exp_1^i, v_2' = exp_2^i, \dots\}_{v_j \in Var} \in T \qquad \neg s(\textbf{\textit{stop}}) \qquad [\![guard]\!]_s \qquad \forall s' : \mu(s') = \sum_i p_i \chi_{\left\{v_j \mapsto [\![exp_j^i]\!]_s\right\}_{v_j \in Var}}}{s \xrightarrow{\tau} \mu} \tag{2.5}$$

Para cualquier estado s, definiremos un conjunto de transiciones habilitadas en s como los pares $(l, \mu) \in L^O \times Dist(S)$ tal que $s \xrightarrow{l} \mu$.

2.4.1. Modelo Compuesto

Al principio del capítulo habíamos dicho que para modelar un sistema distribuido era necesario modelar cada una de las componentes y luego componer los submodelos. Esto se debe a que la ejecución del sistema distribuido es parte de la interacción de las ejecuciones de las componentes del sistema. La forma en que modelamos la concurrencia es a través del interleaving que és la habitual en estos tipos de modelos. Ahora es tiempo de realizar la composición de submodelos de la siguiente forma:

Definición 2.4.1. *Dos componentes $P_1 = (S_1, L_1, init_1, \to_1)$ y $P_2 = (S_2, L_2, init_2, \to_2)$ se pueden componer si $L_1^O \cap L_2^O = \emptyset$, resultando en $P_1 \parallel P_2 = (S_1 \times S_2, L, init_1 \times init_2, \to)$, donde:*

- $L = L^I \cup L^O$, *con* $L^I = L_1^I \cup L_2^I$ *y* $L^O = L_1^O \cup L_2^O \setminus L^I$, *y*
- $\to$, *queda definido por las siguientes reglas:*

$$\frac{s_1 \xrightarrow{c.m}_1 \mu_1 \qquad s_2 \xrightarrow{c.m}_2 \mu_2}{(s1, s2) \xrightarrow{c.m} \mu_1 \times \mu_2} \tag{2.6}$$

$$\frac{s_1 \xrightarrow{l}_1 \mu_1 \qquad l \notin L_2 \vee l = \tau}{(s1, s2) \xrightarrow{l} \mu_1 \times \chi_{s_2}} \qquad \frac{s_2 \xrightarrow{l}_2 \mu_2 \qquad l \notin L_1 \vee l = \tau}{(s1, s2) \xrightarrow{l} \chi_{s_1} \times \mu_2} \tag{2.7}$$

donde $\times$ denota un producto de distribuciones y χ_s denota la función característica de s.

Notar que, para permitir más composición, todas las transiciones reactivas son preservadas en el sistema compuesto.

2.5. Conclusión

En este capítulo hemos sentado las bases teóricas para modelar sistemas distribuidos. Esto quiere decir que podremos describir estos sistemas y construir su árbol de ejecución para el análisis debido. Al construir el árbol de ejecución, entran en juego varios parámetros nuevos, como por ejemplo, la toma de decisiones sobre no determinismos, las distintas secuencias posibles de ejecución, etc. Por esto es necesario introducir conceptos, que nos ayudarán al manipuleo y análisis. En el capítulo siguiente entraremos mas en detalle sobre estos aspectos.

Capítulo 3

Resolución de No Determinismos contemplando Secretos

"Las cosas no se dicen, se hacen, porque al hacerlas se dicen solas"

Woody Allen, 1935

La hora de la ejecución ha llegado y e aquí donde se presentan los mayores problemas. Es muy común que de chicos hallamos jugado al fútbol entre amigos, aun incluso siendo buenos amigos, es muy difícil tomar una decisión al momento de una falta. Ambos bandos quieren siempre la ventaja, y la única solución posible es establecer un juez. En los sistemas distribuidos pasa lo mismo, todas las componentes quieren ejecutarse al mismo tiempo pero la herramienta solo permite una por vez. Nuestro juez designado para el trabajo se llamara *scheduler*. Esta entidad no solo se encargará de elegir una componente habilitada sino que también tiene el trabajo de designarle la probabilidad con la que la elige y revisar si esa elección es equivalente con alguna otra elección en el sistema, así como también otras tareas que serán detalladas más adelante.

3.1. Ejecución

Es necesario definir un par de conceptos previos que nos ayudarán a entender y trabajar sobre la marcha. Estos conceptos nacen precisamente de poner el sistema en funcionamiento.

3.1.1. Caminos

En la sección 2.4.1 hablamos de modelar la concurrencia mediante el interleaving. Podemos deducir entonces que existen numerosas combinaciones y secuencias de eje-

cución. Llamaremos *camino* a cada una de las secuencias posibles en el modelo.

Definición 3.1.1. *Un camino es una secuencia* $s_0 l_0 s_1 \ldots$ *de estados y etiquetas tal que* $init\,(s_0) > 0$ *y, para todo* i, $s_i \xrightarrow{l_i} \mu_i$ *y* $\mu_i\,(s_{i+1}) > 0$.

El conjunto de caminos de un sistema dado P es denotado por $Paths\,(P)$ (o simplemente $Paths$ donde P es claro desde el contexto). Un camino finito σ de P es un prefijo válido $s_0 l_0 s_1 \ldots s_{n-1} l_{n-1} s_n$ de un camino de P, con longitud $len\,(\sigma) = n$ y último estado $last\,(\sigma) = s_n$. El conjunto de caminos finitos de longitud n es denotado por $Paths^n\,(P)$ y $\cup_{n\in\mathbb{N}} Paths^n\,(P)$ por $Paths_{fin}\,(P)$.

3.1.2. Trazas

Muchas veces es necesario visualizar un camino solo para conocer la permutación de transiciones que se fueron sucediendo. Es en este punto cuando la información de los estados del camino resulta una molestia. Para obtener un mejor seguimiento de la ejecución del modelo, introduciremos la noción de una traza.

Definición 3.1.2. *La traza de un camino es la secuencia de etiquetas en él. La función tr extrae la traza de un camino dado y está definido como sigue:*

$$tr\,(s) = \epsilon \quad y \quad tr\,(sl\sigma) = l\,tr\,(\sigma)$$

De esta forma podremos llevar el hilo de la ejecución de una manera sencilla, lo cual intensifica fuertemente la velocidad del análisis desechando gran cantidad de datos no relevantes para el seguimiento del modelo en el manipuleo de los datos.

3.1.3. Transiciones Habilitadas en un Camino

El conjunto de transiciones habilitadas en un camino σ si las hay, son las habilitadas en el último estado:

$$enab^T\,(\sigma) = \left\{(l,\mu) \in L^O \times Dist\,(S) : last\,(\sigma) \xrightarrow{l} \mu\right\}$$

Al principio del capítulo dijimos que el *scheduler* tiene la función de elegir una componente de entre el conjunto de componentes habilitadas. Las componentes habilitadas en un camino finito σ son las que tienen transiciones activas habilitadas:

$$enab^C\,(\sigma) = \left\{P_i : \exists\,(l,\mu) \in enab\,(\sigma) : l \in L_i^O\right\}$$

Esto quiere decir que las componentes habilitadas son aquellas que no deben esperar para realizar una acción sino que esta puede ejecutarse en el acto.

3.2. Resoluciones del No Determinismo

El no determinismo es un problema que surge de la toma de decisiones sobre un conjunto de caminos posibles de manera arbitraria. Es muy distinto a una elección probabilista, las elecciones probabilistas respetan siempre el porcentaje de acierto sobre un camino, en cambio el no determinismo lo hace de manera indistinta sin satisfacer ningún patrón. Sabemos que las maquinas no piensan por si solas y que son incapaces de realizar una elección arbitraria, por lo tanto, es necesaria la presencia de un scheduler que asigna una distribución probabilista al conjunto no determinista.

Formalmente un scheduler para P es una función $\eta : Paths_{fin}(P) \mapsto Dist(L)$ tal que si $\eta(\sigma)(l) > 0$, entonces $last(\sigma) \xrightarrow{l} \mu$ para alguna distribución μ.

3.2.1. Medida Probabilista inducida por un Scheduler

Cuando todas las elecciones no deterministas en un *modelo interactivo probabilista* son resueltas por un scheduler, el sistema resultante es una (posiblemente infinita) cadena de Markov. Por lo tanto es posible definir una medida probabilista sobre el conjunto infinito de caminos del modelo.

Primero definimos una σ-álgebra sobre el conjunto infinito de caminos y luego la medida probabilista sobre la σ-álgebra inducida por un scheduler dado. El *cilindro* inducido por el camino finito σ, es el conjunto de caminos infinitos $\sigma^{\uparrow} = \{\sigma' \in Paths(\mathcal{P}) \mid \sigma'$ es infinito y σ es un prefijo de $\sigma'\}$. Definimos $\mathcal{F}$ como la σ-álgebra sobre el conjunto infinito de caminos de $\mathcal{P}$ generados por el conjunto de cilindros, siendo $\mathcal{P}$ un modelo compuesto.

Definición 3.2.1. *Sea η un scheduler para $\mathcal{P}$. La medida probabilista inducida por η sobre $\mathcal{F}$ es la única medida probabilista Pr_η tal que, para cualquier estado $s \in S$, cualquier acción $l \in L$ y cualquier distribución $\mu \in Dist(S)$:*

$$Pr_\eta\left(s^{\uparrow}\right) = 1 \qquad si\ s = \hat{s}$$

$$Pr_\eta\left(s^{\uparrow}\right) = 0 \qquad si\ s \neq \hat{s}$$

$$Pr_\eta\left(\sigma l s^{\uparrow}\right) = Pr_\eta\left(\sigma^{\uparrow}\right) \cdot \sum_{(l,\mu) \in enab(\sigma)} \eta(\sigma)(l) \cdot \mu(s)$$

Dado un scheduler η para $\mathcal{P}$, la probabilidad de alcanzar un estado objetivo dado por el conjunto $Goal \subseteq S$, puede ser computado por $Pr_\eta(Goal) = Pr_\eta(\cup\{\sigma^{\uparrow} \mid last(\sigma) \in Goal\})$. Denotamos a Goal, como el conjunto de estados cuya evaluación satisface la guarda de la propiedad. Sea $\bar{G}$ un conjunto de caminos $\sigma \in Paths_{fin}(\mathcal{P})$ tal que last(σ)$\in Goal$ y para cada prefijo propio $\hat{\sigma}$ de σ, last($\hat{\sigma}$) $\notin Goal$. Notar que para todo $\sigma' \in Paths_{fin}(\mathcal{P})$ que alcanza un estado en $Goal$, existe un único $\sigma \in \bar{G}$ tal que $\sigma' \in \sigma^{\uparrow}$. Entonces, tenemos que:

$$Pr_\eta\left(Goal\right) = Pr_\eta\left(\uplus\left\{\sigma^\uparrow \mid \sigma \in \bar{G}\right\}\right) = \sum_{\sigma\in\bar{G}} Pr_\eta\left(\sigma^\uparrow\right). \tag{3.1}$$

El problema del model checking en sistemas probabilistas no deterministas se enfoca en encontrar los peores escenarios posibles. Por lo tanto, tiene como objetivo encontrar la máxima o la mínima probabilidad de alcanzar un estado en el conjunto *Goal* partiendo sobre clases particulares de schedulers. Esto es, si $\mathcal{K}$ es una clase de schedulers (es decir, el conjunto de todos los schedulers que satisfacen una condición dada), entonces estamos interesados en encontrar el $sup_{\eta\in\mathcal{K}} Pr_\eta\left(Goal\right)$ o el $inf_{\eta\in\mathcal{K}} Pr_\eta\left(Goal\right)$.

No todas las resoluciones de no determinismos son apropiadas en una configuración distribuida. Existen *"schedulers todopoderosos"* que permiten a una componente adivinar el resultado de una elección probabilista de una segunda componente incluso cuando no existe comunicación entre ellas (incluso indirectamente). Por lo tanto restringiremos la clase a *schedulers distribuidos*. Esto nos lleva a una solución más realista de no determinismos, pero desafortunadamente hace que el problema del model checking sea indecidible en general.

3.2.2. Proyecciones

Los Schedulers Distribuidos consideran la noción de conocimiento local por cada componente que es obtenida observando parcialmente el comportamiento del sistema global. Entonces, una componente puede ver solamente la ejecución global a través de sus estados locales y de las acciones que realiza. Por lo tanto, dos ejecuciones globales diferentes pueden parecer similares para una misma componente. Implementaremos esto con una *función de proyección*.

Definición 3.2.2. *Dado un sistema compuesto $P = P_1 \parallel \cdots \parallel P_n$, la proyección $\sigma|_{P_i}$ de un camino σ de P sobre el participante P_i es definido recursivamente como:*

$$s|_{P_i} = s_i$$
$$\sigma l s|_{P_i} = \begin{cases} \sigma|_{P_i} & if\, l \notin L_i \\ \sigma|_{P_i} l s_i & otherwise \end{cases}$$

donde s_i representa la i-th componente de la tupla s.

Es decir, la proyección de un camino σ de un sistema compuesto sobre un participante P_i, es exactamente lo que el participante pudo observar desde su lugar.

3.2.3. Equivalencia $\sim$

La equivalencia es una parte esencial del sistema, esta surge de trabajar con secretos. Si una componente posee dos acciones distintas que las demás componentes no pueden distinguir, entonces se dice que esas acciones son equivalentes.

Ejemplo 3.2.1. *Supongamos que existen dos caminos distintos σ y σ', P_a y P'_a proyecciones de los caminos σ y σ' respectivamente sobre el participante a. Entonces si $P_a \sim P'_a$, quiere decir que para el participante a, σ y σ' son* ***equivalentes****.*

Mas específicamente definimos a $\sim$ como la menor relación que satisface lo siguiente:

$$\begin{array}{ll} \tau \sim \tau & \\ c.v \sim c.v' & \iff v \sim v' \\ E(v1,k1,r1) \sim E(v2,k2,r2) & \text{para todo } v1,v2,k1,k2,r1,r2 \\ (a,b) \sim (c,d) & \iff a \sim c \text{ y } b \sim d \end{array}$$

Extendemos la noción de equivalencia $\sim$ sobre trazas en la manera obvia: dos trazas son equivalentes si son de igual longitud y las etiquetas en las mismas posiciones son equivalentes. Dos caminos son equivalentes si sus trazas lo son. Dado un camino del sistema σ, $[\sigma]_{P_i}$ denota la clase de equivalencia asociada a las proyecciones del camino σ sobre el participante P_i.

3.2.4. Scheduler de Interleaving

En un estado (compuesto) con transiciones habilitadas, el no determinismo se resuelve en dos fases. Primero, la componente es elegida entre todas las componentes habilitadas. Esta elección debe ser probabilista y es realizada por el llamado *scheduler de interleaving*, el cual es una función

$$inter : Paths \to Dist\,(Participantes)$$

tal que:

$$inter\,(\sigma)\,(enab\,(\sigma)) = 1$$

esto es, solo asigna probabilidades mayores que cero a participantes habilitados, y además

$$\begin{array}{c} \forall i \in I \subseteq \{1,...,n\} :\ P_i \in \bigcap_{i \in I} enab\,(\sigma_i) \wedge tr\left(\sigma|_{P_i}\right) \sim tr\left(\sigma'|_{P_i}\right) \\ \Rightarrow\ \forall j \in I :\ \dfrac{inter\,(\sigma)\,\left(P_j\right)}{\sum_{i \in I} inter\,(\sigma)\,(P_i)} = \dfrac{inter\,(\sigma')\,\left(P_j\right)}{\sum_{i \in I} inter\,(\sigma')\,(P_i)} \end{array} \tag{3.2}$$

provisto que $\sum_{i \in I} inter\,(\sigma)\,(P_i) \neq 0$ y $\sum_{i \in I} inter\,(\sigma')\,(P_i) \neq 0$.

3.2.5. Scheduler Local

Luego de que el *scheduler de interleaving* haya escogido una componente, la componente elegida decide que transición realizar entre todas las transiciones activas habilitadas. Esta decisión local es resuelta por el *scheduler local* tomando en cuenta solo el

conocimiento local. Por eso, para cada participante P_i definimos un *scheduler local* como una función:

$$local_i : Paths_i \to Dist\left(L_i^O \times Dist\left(S_i\right)\right)$$

tal que:

$$local_i\left(enab\left(\sigma\right)\right) = 1$$

esto es, solo asigna probabilidades mayores que cero a las transiciones habilitadas, y además

$$\begin{aligned} &\forall \sigma, \sigma' : \sigma \sim \sigma' \wedge enab\left(\sigma\right) \sim enab\left(\sigma'\right) \\ &\quad \Rightarrow \ \forall \left(l, \mu\right) \in enab\left(\sigma\right) : local_i\left(\sigma\right)\left(\left[l, \mu\right]\right) = local_i\left(\sigma'\right)\left(\left[l, \mu\right]\right) \end{aligned} \tag{3.3}$$

respecto a las equivalencias definidas, con

$$\left(l, \mu\right) \sim \left(l', \mu'\right) \ \Leftrightarrow \ l \sim l' \wedge \left(l = \tau \ \Rightarrow \ \mu = \mu'\right)$$

y

$$enab\left(\sigma\right) \sim enab\left(\sigma'\right) \ \Leftrightarrow \ enab\left(\sigma\right) / \sim \ = enab\left(\sigma'\right) / \sim$$

3.2.6. Scheduler Distribuido bajo $\sim$

Un *scheduler distribuido* η consiste de un scheduler de interleaving y de un scheduler local para cada participante tal que $\forall \sigma \in Paths_{fin}\left(\mathcal{C}\right)$ con $enab\left(\sigma\right) \neq \emptyset$ y $\forall \left(l, \mu\right) \in L_{\mathcal{C}}$, estos schedulers fueron introducidos por primera vez en [10]:

$$\eta\left(\sigma\right)\left(\left(l, \ \mu\right)\right) = \sum_{i=1}^{n} inter\left(\sigma\right)\left(P_i\right) \cdot local_i\left(\sigma|_{P_i}\right)\left(l, \mu\right)$$

Si a lo sumo una componente puede realizar la acción $\left(l, \mu\right)$, la ultima ecuación se reduce a:

$$\eta\left(\sigma\right)\left(\left(l, \ \mu\right)\right) = inter\left(\sigma\right)\left(P_i\right) \cdot local_i\left(\sigma|_{P_i}\right)\left(l, \mu\right)$$

siempre que $\left(l, \mu\right) \in L_{P_i}^O$. Con las restricciones de las ecuaciones (3.2) y (3.3), este es un *scheduler distribuido bajo secreto* (o bajo $\sim$), porque respeta no solo la naturaleza distribuida del sistema, sino también la equivalencia definida a partir del encriptado.

3.3. Conclusión

En este capítulo hemos introducido la noción de equivalencia así como también el *scheduler distribuido bajo* $\sim$, que es el encargado de asignar probabilidades y construir restricciones durante el unfolding del modelo, que veremos en el capítulo siguiente, lo cual lo convierte en la entidad que agrupa los caminos en clases de equivalencias. También se han detallado herramientas útiles que contribuirán al análisis durante su ejecución.

Capítulo 4

Resolución de un Problema a un Sistema de Ecuaciones

"¿Debo rechazar mi cena porque no entiendo completamente el proceso de digestión?"

Wilfred Trotter, 1872-1939

Ya definimos nuestro modelo, ya definimos las herramientas necesarias para poner el sistema en marcha. Podemos incluso construir el árbol de ejecución del modelo, pero no servirá de nada a menos que podamos interpretarlo sobre números. Al trabajar con sistemas distribuidos probabilistas, todo camino y decisiones deben ser reflejados mediante un número probabilista, que junto con las restricciones necesarias nos permitirán construir el sistema de ecuaciones que probará nuestras propiedades.

4.1. Un ejemplo

Para poder comprender la generación del sistema de ecuaciones, vamos a utilizar un ejemplo simple en el cual un cliente debe seleccionar un servidor de entre los disponibles de manera anónima. El modelo, que se lo da en la Figura 1, está constituido por 4 componentes; un cliente, dos servidores y un adversario. Podemos notar que solo fueron necesarios tres prototipos para describirlo (Cliente, Servidor y Adversario) y que en el caso del servidor se crearon dos instancias. El cliente cuyas transiciones le permiten elegir entre alguno de los dos servidores, efectúa su elección, mientras que los servidores esperan con ansias y contestan al cliente con un acknowledgement. Mientras tanto el adversario solo puede escuchar los acknowledgement y adivinar cual de los servidores fue elegido.

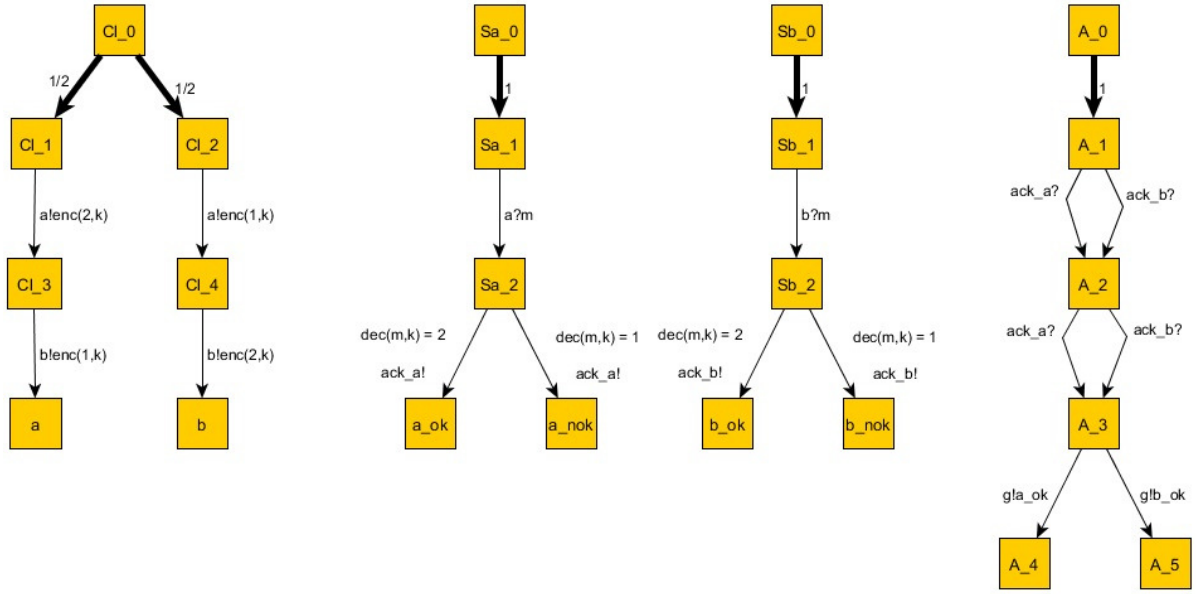

Fig. 1. Modelo de un sistema distribuido cuyas componentes son C_l, S_a, S_b y $\mathcal{A}$ respectivamente.

Nuestra preocupación consiste en que exista algún camino σ donde el adversario pueda adivinar con una probabilidad mayor a $\frac{1}{2}$, por lo tanto tendremos que crear el árbol de ejecución del modelo, recolectar los caminos donde el adversario acierta su adivinanza y ver que la suma de las probabilidades de los caminos sea menor o igual a $\frac{1}{2}$.

4.2. El Unfolding

A continuación, simularemos todas los posibles caminos y crearemos un árbol de ejecución del modelo. A cada paso del unfolding, que acompaña una acción de alguna de las componentes habilitadas, se le asignará determinadas variables que reflejarán la probabilidad de elegir esa acción en particular. La multiplicación de las variables asignadas en cada acción, de un mismo camino, reflejara la probabilidad total del camino, es decir, la probabilidad de que el camino se ejecute completamente.

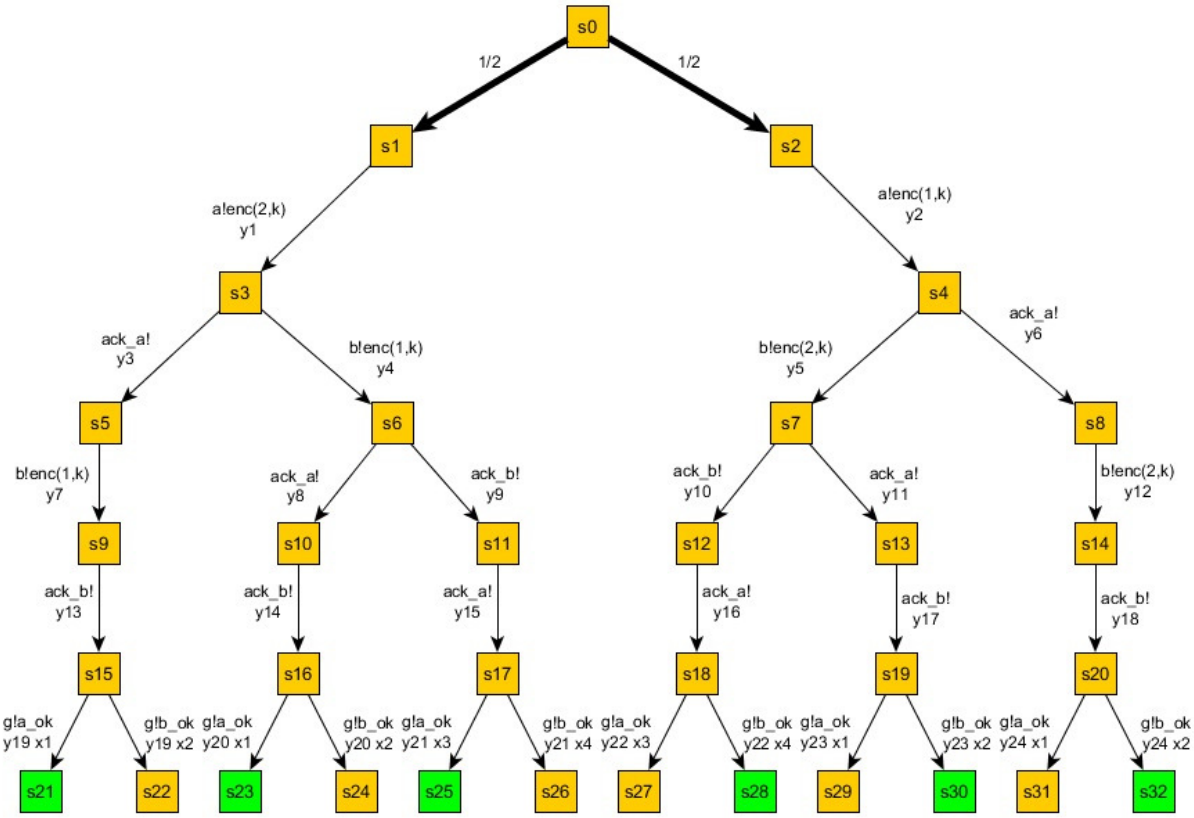

Fig. 2. Scheduler paramétrico η. Los estados son las tuplas obvias. Los estados en verde representan la adivinación correcta.

La probabilidad del camino $\sigma = s_0\ \mu\ s_1\ a!_{enc(2,k)}\ s_3\ b!_{enc(1,k)}\ s_6\ ack_a!\ s_{10}\ ack_b!\ s_{16}$ puede ser calculado usando la definición 3.2.1 para el scheduler simbólico: $Pr\left(\sigma^{\uparrow}\right) = \frac{1}{2} \cdot y_1 \cdot y_4 \cdot y_8 \cdot y_{14}$. Para construir un scheduler distribuido bajo secreto, tenemos que considerar el scheduler de interleaving y los schedulers locales. Notar que en nuestro ejemplo, solo el scheduler local de $\mathcal{A}$ es relevante. En el scheduler paramétrico de la ***Fig. 2.***, las variables y_i's corresponden a las elecciones probabilistas del scheduler de interleaving, mientras que las variables x_i's corresponden a las elecciones probabilistas del scheduler local de $\mathcal{A}$ (todos los otros schedulers locales solo tienen elecciones triviales y por lo tanto los omitiremos). La multiplicación de las variables $y_i's$ e $x_j's$ en el último paso corresponden a la composición del scheduler de interleaving con los schedulers locales en orden a definir el scheduler distribuido η así como esta en la sección 3.2.6. Notar que, contrariamente al hecho de que las variables $y_i's$ son todas diferentes, x_1, x_2, x_3 y x_4 se repiten en varias ramas. Esto tiene que ver con el hecho de que algún camino local de $\mathcal{A}$ es el mismo para diferentes caminos del sistema compuesto. Por ejemplo, la elección del scheduler local $\mathcal{A}$ en los estados s_{15}, s_{16}, s_{19} y s_{20} son determinadas por el mismo camino local $a_0\ \mu'\ a_1\ ack_a!\ a_2\ ack_b!\ a_3$, con $\mu'\ (a_1) = 1$. (Notar que, por ejemplo $\left(s_0\ \mu\ s_1\ a!_{enc(2,k)}\ s_3\ ack_a!\ s_5\ b!_{(enc1,k)}\ s_9\ ack_b!\ s_{15}\right)|_{\mathcal{A}} = a_0\ \mu'\ a_1\ ack_a!\ a_2\ ack_b!\ a_3$.)

4.3. El Problema de Optimización

Siguiendo la ***Def. 3.2.1*** y la ecuación (3.1), la probabilidad paramétrica de alcanzar un estado de adivinación correcta (que están coloreados en la Fig. 2), es dada por el polinomio

$$\frac{1}{2}\cdot y_1\cdot y_3\cdot y_7\cdot y_{13}\cdot y_{19}\cdot x_1+\frac{1}{2}\cdot y_1\cdot y_4\cdot y_8\cdot y_{14}\cdot y_{20}\cdot x_1+\frac{1}{2}\cdot y_1\cdot y_4\cdot y_9\cdot y_{15}\cdot y_{21}\cdot x_3+$$
$$\frac{1}{2}\cdot y_2\cdot y_5\cdot y_{10}\cdot y_{16}\cdot y_{22}\cdot x_4+\frac{1}{2}\cdot y_2\cdot y_5\cdot y_{11}\cdot y_{17}\cdot y_{23}\cdot x_2+\frac{1}{2}\cdot y_2\cdot y_6\cdot y_{12}\cdot y_{18}\cdot y_{24}\cdot x_2.$$

Maximizando (resp. minimizando) el polinomio anterior bajo las restricciones obvias (cada variable toma un valor entre $[0,1]$, y esto define una distribución de probabilidad propia, por ejemplo $y_3+y_4=1$), cede a la probabilidad máxima (resp. mínima) bajo schedulers distribuidos.

De cualquier forma, esto no es suficiente para caracterizar un scheduler distribuido bajo secreto. Notar que, para $\sigma_a = s_0\ \mu\ s_1\ a!_{enc(2,k)}\ s_3\ b!_{enc(1,k)}\ s_6$ y $\sigma_b = s_0\ \mu\ s_2\ a!_{enc(1,k)}\ s_4\ b!_{enc(2,k)}\ s_7$, y las componentes $S_a\ yS_b$, se cumplen las condiciones dadas en las ecuaciones (3.2) del scheduler de interleaving. Por lo tanto, debería mantenerse que $\frac{inter(\sigma_a)(S_a)}{inter(\sigma_a)(S_a)+inter(\sigma_a)(S_b)} = \frac{inter(\sigma_b)(S_a)}{inter(\sigma_b)(S_a)+inter(\sigma_b)(S_b)}$. Ya que $inter\,(\sigma_a)\,(S_a)=y_8$, $inter\,(\sigma_a)\,(S_b)=y_9$, $inter\,(\sigma_b)\,(S_a)=y_{11}$, $inter\,(\sigma_b)\,(S_b)=y_{10}$, esto significa que la restricción $\frac{y_8}{y_8+y_9}=\frac{y_{11}}{y_{10}+y_{11}}$ necesita ser considerada en el problema de optimización. Similarmente, las restricciones $\frac{y_9}{y_8+y_9}=\frac{y_{10}}{y_{10}+y_{11}}$, $\frac{y_3}{y_3+y_4}=\frac{y_6}{y_5+y_6}$, y $\frac{y_4}{y_3+y_4}=\frac{y_5}{y_5+y_6}$ también son necesarias.

A continuación, daremos la construcción formal del problema de optimización.

$$\begin{aligned}
K=\{&y^i_\sigma \mid && \sigma\in Paths_{fin}(\mathcal{C})\wedge 1\le i\le \#\mathcal{C}\wedge enab\left(\sigma|_{\mathcal{P}_i}\right)\neq\emptyset\}\cup\\
\{&x^a_{\sigma|_{\mathcal{P}_i}} \mid && \sigma\in Paths_{fin}(\mathcal{C})\wedge 1\le i\le \#\mathcal{C}\wedge a\in enab\left(\sigma|_{\mathcal{P}_i}\right)\}\cup\\
\{&z^{j,I}_f \mid && I\subseteq\{1,....,\#\mathcal{C}\}\wedge j\in I\wedge f: I\mapsto Paths_{fin}\wedge\\
& && \exists\sigma\in Paths_{fin}(\mathcal{C}):\forall i\in I: enab\left(\sigma|_{\mathcal{P}_i}\right)\neq\emptyset\wedge f(i)=[\sigma|_{\mathcal{P}_i}]_{\sim i}\}
\end{aligned}$$

Las variables y^i_σ y $x_{\sigma|_{\mathcal{P}_i}}$ están asociadas al interleaving y a los schedulers locales respectivamente, y se espera que $inter\,(\sigma)\,(i)=y^i_\sigma$ y $local_{\mathcal{P}_i}(q)=x^a_{\sigma|_{\mathcal{P}_i}}$. Las variables $z^{j,I}_f$ estan asociadas a las restricciones del scheduler de interleaving de la sección 3.2.4. Se espera que $z^{j,I}_f=\frac{inter(\sigma)(\mathcal{P}_j)}{\sum_{i\in I} inter(\sigma)(\mathcal{P}_i)}$. Esto asegura la igualdad deseada.

Ejemplo 4.3.1. *Volvamos a tomar como ejemplo los caminos* $\sigma_a = s_0\ \mu\ s_1\ a!_{enc(2,k)}\ s_3\ b!_{enc(1,k)}\ s_6$ *y* $\sigma_b = s_0\ \mu\ s_2\ a!_{enc(1,k)}\ s_4\ b!_{enc(2,k)}\ s_7$, *y las componentes* $S_a\ yS_b$.

Entonces con las nuevas variables $z^{j,I}_f$, *tendremos restricciones de la siguiente forma:*

$$
\begin{aligned}
inter\left(\sigma_a\right)\left(S_a\right) &= \left(\ inter\left(\sigma_a\right)\left(S_a\right) + inter\left(\sigma_a\right)\left(S_b\right)\ \right)\ z_f^{S_a,\{S_a,S_b\}} \\
inter\left(\sigma_b\right)\left(S_a\right) &= \left(\ inter\left(\sigma_b\right)\left(S_a\right) + inter\left(\sigma_b\right)\left(S_b\right)\ \right)\ z_f^{S_a,\{S_a,S_b\}} \\
inter\left(\sigma_a\right)\left(S_b\right) &= \left(\ inter\left(\sigma_a\right)\left(S_a\right) + inter\left(\sigma_a\right)\left(S_b\right)\ \right)\ z_f^{S_b,\{S_a,S_b\}} \\
inter\left(\sigma_b\right)\left(S_b\right) &= \left(\ inter\left(\sigma_b\right)\left(S_a\right) + inter\left(\sigma_b\right)\left(S_b\right)\ \right)\ z_f^{S_b,\{S_a,S_b\}}
\end{aligned}
$$

4.4. Construcción del Polinomio Probabilista

Como pudimos notar, el enfoque del unfolding es encontrar aquellos caminos que alcanzan estados objetivos en *Goal*. En realidad nos interesa la probabilidad de ejecutar dichos caminos y más aún la probabilidad de ejecutar alguno de estos caminos. Esto se realiza construyendo un polinomio probabilista, tal como lo hicimos al principio de la sección anterior. En esta sección definiremos el polinomio de manera formal.

Sea G el conjunto de caminos σ con $last\left(\sigma\right) \in Goal$ y sea $Paths_G\left(\mathcal{C}\right) = Paths\left(\mathcal{C}\right) \cap \bar{G}$. La función $\mathbf{P}$ que asigna términos polinómicos con variables en K a cada $path \in Path_G\left(\mathcal{C}\right)$, es definida como

$$
\begin{aligned}
&\mathbf{P}\left(\hat{s}_{\mathcal{C}}\right) = 1 \\
&\mathbf{P}\left(\sigma\alpha s\right) = \begin{cases} \mathbf{P}\left(\sigma\right) \cdot y_\sigma^i \cdot x^\alpha_{\sigma|_{\mathcal{P}_i}} \cdot \mu\left(s\right) & \text{if } enab\left(\sigma\right) \neq \emptyset \wedge \left(\alpha,\ \mu\right) \in L^O_{\mathcal{P}_i} \times Dist\left(S_i\right) \wedge last\left(\sigma\right) \xrightarrow{\alpha} \mu \\ \mathbf{P}\left(\sigma\right) \cdot \mu\left(s\right) & \text{if } \left(\alpha,\ \mu\right) \in L^I_{\mathcal{P}_i} \times Dist\left(S_i\right) \end{cases} \\
&\mathbf{P}\left(\sigma\right) = 0
\end{aligned}
$$

Entonces, siguiendo la ecuación (3.1), el objetivo polinomial del problema de optimización es

$$\sum_{\sigma \in Paths_G(\mathcal{C})} \mathbf{P}\left(\sigma\right)$$

y está sujeto a las siguientes restricciones:

$$
\begin{aligned}
&0 \leq v \leq 1 && if\ v \in V \\
&\sum_{a \in A} x^a_{\sigma|_{\mathcal{P}_i}} = 1 && if\ \sigma \in Paths\left(\mathcal{C}\right), 1 \leq i \leq \#\mathcal{C}, A = enab\left(\sigma|_{\mathcal{P}_i}\right) \\
&\sum_{i \in I} y^i_\sigma = 1 && if\ \sigma \in Paths\left(\mathcal{C}\right), I = \left\{i \mid 1 \leq i \leq \#\mathcal{C},\ enab\left(\sigma|_{\mathcal{P}_i}\right) \neq \emptyset\right\} \\
&z_f^{j,I}\left(\sum_{i \in I} y^i_\sigma\right) = y^j_\sigma && if \begin{cases} \sigma \in Paths\left(\mathcal{C}\right), I \subseteq \left\{i \mid 1 \leq i \leq \#\mathcal{C},\ enab\left(\sigma|_{\mathcal{P}_i}\right) \neq \emptyset\right\} \\ j \in I,\ \forall i \in I:\ enab\left(\sigma|_{\mathcal{P}_i}\right) \neq \emptyset \wedge f\left(i\right) = \left[\sigma|_{\mathcal{P}_i}\right]_{\sim i} \end{cases}
\end{aligned}
$$

De esta forma, el resultado obtenido es un polinomio probabilista cuyo probabilidad consiste en que los caminos alcancen estados con ciertas características. No olvidar

que para la construcción del polinomio fueron necesarias declarar ciertas restricciones lineales y no lineales, que acompañaran al polinomio en el resultado del análisis. Pero todavía nos podemos hacer una pregunta: Es este el resultado óptimo?. No es difícil notar que se pueden realizar optimizaciones sobre estos resultados. En la sección siguiente hablaremos sobre este asunto.

4.5. Optimizaciones

Notar que al ir construyendo el árbol, siempre trabajamos con información parcial del sistema, lo cual hace que perdamos la capacidad de saber si el resultado obtenido puede simplificarse. Por ello desarrollamos un conjunto de heurísticas que permiten reducir el tamaño del problema a analizar, eliminando ecuaciones y variables que resultan redundantes. Las heurísticas se agrupan en varias fases. A continuación iremos mencionando estas estrategias en el orden en que se aplican. Recordar que el resultado final del unfolding es un conjunto de restricciones y un polinomio probabilista.

Los siguientes items representan las reducciones correspondientes al polinomio y al conjunto de restricciones. Las expresiones señaladas en rojo serán eliminadas, mientras que las expresiones en verde serán añadidas.

- Detecta factores comunes en el polinomio.

  Ejemplo 4.5.1. *Supongamos el siguiente polinomio:*

  $0,5 * y1 * y2 * y3 * x1 + 0,5 * y4 * y5 * y6 * x2$

  entonces:

  $0,5 * (y1 * y2 * y3 * x1 + y4 * y5 * y6 * x2)$

- Elimina restricciones cuyas variables no aparecen en el polinomio.

  Ejemplo 4.5.2. *Supongamos el siguiente polinomio:*

  $y1 * y2 * y3 * x1 + y4 * y5 * y6 * x2$

  y la siguiente restricción:

  $y7 + y8 = 1$

  entonces la restricción anterior será eliminada.

- Identifica variables equivalentes desde restricciones lineales.

  Ejemplo 4.5.3. *Tenemos el siguiente caso:*

  $x1 + x2 = 1$

  $x1 + x3 = 1$

  entonces:

  $x2 = x3$

- Identifica variables equivalentes desde restricciones equivalentes.

  Ejemplo 4.5.4. *Tenemos el siguiente caso:*

  $y1 = (y1 + y2) * z1$

  $y1 = (y1 + y2) * z2$

  entonces:

  $z1 = z2$

- Reemplaza variables equivalentes.

  Ejemplo 4.5.5. *Supongamos el siguiente polinomio:*

  $y1 * y2 * y3 * x1 + y4 * y5 * y6 * x2$

  y la siguiente restricción:

  $x1 = x2$

  entonces el polinomio queda de la siguiente forma:

  $y1 * y2 * y3 * x1 + y4 * y5 * y6 * x1$

- Elimina factores 1 del polinomio probabilista.

  Ejemplo 4.5.6. *Supongamos el siguiente polinomio:*

  $1 * 1 * 1 * y1 * y2 * x1 + 1 * 1 * y3 * y4 * x2$

  entonces:

  $y1 * y2 * x1 + y3 * y4 * x2$

- Saca factor común de expresiones iguales a 1 en el polinomio probabilista.

  Ejemplo 4.5.7. *Supongamos dos términos del polinomio:*

  $y1 * y2 * y3 + y1 * y2 * y4$

  y tenemos la restricción:

  $y3 + y4 = 1$

  entonces:

  $y1 * y2 * (y3 + y4) = y1 * y2 * 1 = y1 * y2$

Si al finalizar estas optimizaciones se logró reducir alguna expresión o eliminar alguna restricción, entonces el ciclo de optimizaciones se repite. Si al contrario, no se produjeron cambios, entonces el módulo termina.

4.6. Conclusión

En este capítulo hemos presentado un algoritmo para realizar análisis de alcanzabilidad en sistemas probabilistas distribuidos que deben administrar pasaje de mensajes de manera confidencial. La solución se basa en reducir este problema a un problema de optimización no lineal. Tal problema está conformado por un polinomio objetivo, que representa la probabilidad de alcanzar los estados deseados, y un sistema de desigualdades no lineales, cuyo espacio de soluciones captura exactamente al conjunto de todos los schedulers distribuidos bajo secreto.

Además, hemos presentado un conjunto de heurísticas que permiten reducir el problema en cantidad de variables y desigualdades dentro del sistema.

Capítulo 5

Un Lenguaje para el Modelado de Protocolos Criptográficos

"En el proceso de la Escritura, la imaginación y la memoria se confunden".

Adelaida G. Morales, 1945

Clouseau recibe un modelo y prueba alguna propiedad de seguridad sobre el mismo. Esto quiere decir que el modelo debe ser especificado de alguna forma que Clouseau pueda comprender. Para ello, se creo un lenguaje apto para trabajar con sistemas distribuidos. En este capítulo daremos la sintaxis correspondiente a los elementos del modelo.

Como vimos en el capítulo anterior, un modelo está compuesto por *componentes* que interactúan entre si a través de *transiciones* que ejecutan a lo largo del tiempo. Por eso, empezaremos dando la sintaxis de las componentes.

5.1. Prototipos

Una componente es una instancia de un elemento llamado prototipo. Los prototipos poseen la siguiente sintaxis:

proctipe ProcName (Param)

initial-distribution

transitions

end

Vale la pena aclarar que el nombre del prototipo (ProcName) debe ser *único*, no pueden existir dos prototipos con el mismo nombre. Además, por cada prototipo solo puede existir una distribución inicial y debe ir al inicio del prototipo.

5.2. Parámetros

Recordemos que los parámetros de un prototipo son nombres de canales y constantes, mientras que los parámetros de una instancia son canales y valores. Existe la peculiaridad de que la notación de los nombres de canales (ChanNames) es idéntica a la de los canales (Channels), y es así:

- Canales de Salida: Se define con un nombre seguido de un signo de exclamación (!).
- Canales de Entrada: Se define con un nombre seguido de un signo de interrogación (?).
- Constantes: Puede ser una palabra o un número entero.
- Valores: Los Valores se definen como números enteros.

Ejemplo 5.2.1. *Parámetros de Prototipos: (A?,B!,mama) siendo A? un nombre de canal de entrada, B! un nombre de canal de salida y mama* $\in$ *Const.*

Ejemplo 5.2.2. *Parámetros de Instancias: (A?,B!,2) siendo A? un canal de entrada, B! canal de salida y* $2 \in Val$.

El número de parámetros de la instancia y el prototipo deben ser iguales así como el tipo de parámetro en cada campo también debe coincidir. Tampoco está permitido declarar en los parámetros dos canales del mismo tipo con el mismo nombre, aunque si pueden coexistir dos canales con el mismo nombre si son de distinto tipo, es decir, dos canales A pero uno es de entrada y el otro de salida.(A?,A!)

5.3. Distribuciones

Una distribución tiene la siguiente sintaxis:

$$\sum_i P_i : v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

donde $exp_i \in Exp_V$, $\sum_i P_i = 1$, siendo $0 \leq P_i \leq 1$ la probabilidad del conjunto de asignación de variables y V el parámetro de las expresiones, que dependerá de si la distribución es inicial, reactiva o activa. P_i puede escribirse como un número de punto flotante o como una división de números naturales.

Ejemplo 5.3.1. *Ambos casos son iguales:*

$$0{,}5 : var1 = 1 + 0{,}5 : var1 = 0;$$
$$\textit{ó}$$
$$1/2 : var1 = 1 + 1/2 : var1 = 0;$$

Syntax Sugar 5.3.1. *Si $P_i = 1$, entonces puede omitirse:*

$$v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

Notar que todas las distribuciones deben terminar en punto y coma (;). Si la distribución es de tipo *inicial*, entonces deberá tener el prefijo ***init*** $\rightarrow$ y $V = \{Const \cup Int\}$. Además, toda palabra no declarada como constante en los parámetros, ni declarada como variable en la distribución inicial, será interpretada como una variable ($palabra \in Var$), con valor cero ($palabra = 0$).

Ejemplo 5.3.2. *En el caso de que un prototipo prescinda de la distribución inicial, se deberá escribir de la siguiente forma:*

$$\textbf{\textit{init}} \rightarrow;$$

De esta manera, toda variable que ocurra en el prototipo, será inicializada con valor cero y probabilidad igual a 1*:*

$$\textbf{\textit{init}} \rightarrow \ 1 : v'_1 = 0, v'_2 = 0, ..., v'_n = 0;$$

5.4. Transiciones

A continuación daremos la sintaxis de los tres tipos de transiciones existentes, pero antes dejare bien en claro que la palabra reservada ***stop*** solo puede ser utilizada en transiciones de *salida e interna*, esto se debe a la naturaleza de su acción y a como se ejecutan internamente las transiciones.

Antes de comenzar, quisiera decir que la transición que define la propiedad *input enabled* se crea automáticamente para cada componente del modelo, tomando las guardas de las transiciones reactivas de la componente.

5.4.1. Transiciones de Entrada

La sintaxis de las transiciones de entrada es la siguiente:

$$[c?msg]\ guard^{in} \rightarrow \sum_i P_i : v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

donde $c \in InNames$, $\textbf{\textit{msg}} \in Var$ es una palabra reservada, $guard^{in} \in BoolExp_{Var \cup Const \cup Int \cup \{msg\}}$, $exp_i \in Exp_{Const \cup Int \cup Var \cup \{msg\}}$, $\sum_i P_i = 1$ y $0 \leq P_i \leq 1$.

Syntax Sugar 5.4.1. *Si* $\forall \xi \ [\![guard^{in}]\!]_{\xi} = \textbf{\textit{true}}$*, entonces se puede omitir siempre y cuando se deje un espacio en blanco entre el cierre del paréntesis y la flecha:*

$$[c?msg] \rightarrow \sum_i P_i : v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

5.4.2. Transiciones de Salida

La sintaxis de las transiciones de salida es la siguiente:

$$[c!exp]\ guard \rightarrow \sum_i P_i : v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

ó

$$[c!exp]\ guard \rightarrow \sum_i P_i :v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n; stop;$$

donde $c \in OutNames$, exp,$exp_i \in Exp_{Var \cup Const \cup Int}$, $guard \in BoolExp_{Var \cup Const \cup Int}$, $\sum_i P_i = 1$ y $0 \leq P_i \leq 1$. Notar que la palabra ***stop*** se escribe después de la distribución y debe llevar punto y coma (;). Esto es opcional.

Syntax Sugar 5.4.2. *Si* $\forall \xi \ [\![guard]\!]_{\xi} = \textbf{\textit{true}}$*, entonces se puede omitir siempre y cuando se deje un espacio en blanco entre el cierre del paréntesis y la flecha:*

$$[c!exp] \rightarrow \sum_i P_i : v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

5.4.3. Transiciones Internas

La sintaxis de las transiciones internas es la siguiente:

$$[tau]\ guard \rightarrow \sum_i P_i : v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

ó

$$[tau]\ guard \rightarrow \sum_i P_i :v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n; stop;$$

donde $guard \in BoolExp_{Var \cup Const \cup Int}$, $exp_i \in Exp_{Var \cup Const \cup Int}$, $\sum_i P_i = 1$ y $0 \leq P_i \leq 1$. Notar que la palabra ***stop*** se escribe después de la distribución y debe llevar punto y coma (;). Esto es opcional.

Syntax Sugar 5.4.3. *Si* $\forall \xi \ [\![guard]\!]_{\xi} = \textbf{\textit{true}}$*, entonces se puede omitir siempre y cuando se deje un espacio en blanco entre el cierre del paréntesis y la flecha:*

$$[tau] \rightarrow \sum_i P_i : v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

Syntax Sugar 5.4.4. *También se puede omitir la palabra tau si así lo desea, o incluso [tau]:*

$$[tau] \rightarrow \sum_i P_i : v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

$$ó$$

$$[] \rightarrow \sum_i P_i : v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

$$ó$$

$$\rightarrow \sum_i P_i : v'_1 = exp_1, v'_2 = exp_2, ..., v'_n = exp_n;$$

5.5. Instancias

Una instancia se genera de la siguiente forma:

instance Instance-Name = Proctype-Name (Param)

donde Instance-Name es el nombre de la instancia, Proctype-Name es el nombre del prototipo a instanciar y Param son los parámetros que le pasaremos al prototipo. Al igual que los nombres de los prototipos deben ser únicos, también los nombres de las instancias deben ser únicos, pero vale la pena aclarar que el nombre de la instancia puede ser igual al nombre del prototipo.

5.6. Expresiones y Operadores

A continuación daremos la sintaxis de los operadores que podremos utilizar en las expresiones booleanas:

- (!) es el operador Not ($\neg$).
- (&&) es el operador And ($\wedge$)
- (||) es el operador Or ($\vee$).
- (==) es el operador de igualdad (==).
- (! =) es el operador de desigualdad ($\neq$).

Como dijimos en el capítulo 2, los operadores Or y de desigualdad son derivables de los otros pero nuestro lenguaje nos permite usarlos. También nos permite usar la expresión ***false*** si es necesario. Las expresiones $\pi_1(Exp_V)$, $\pi_2(Exp_V) \in Exp_V$, tendrán las sintaxis pi1(Exp_V) y pi2(Exp_V) respectivamente. Las demás permanecerán iguales.

5.7. Propiedades

En el capítulo 2, vimos que existen dos formas de expresar la propiedad deseada. Una de ellas se llama *anonimidad* y se define de la siguiente manera:

$$P_\eta(secret = x | guess = x) \leq P_\eta(secret = x) \quad \forall\ secreto\ x\ y\ \forall\ scheduler\ \eta \tag{5.1}$$

usualmente, la probabilidad $P_\eta(secret = x)$ es conocida de antemano e igual para todo scheduler, con lo que la ecuación anterior puede reescribirse de la siguiente manera:

$$\begin{aligned} & P_\eta\left(secret = x \mid guess = x\right) && \leq && z \\ \equiv & \frac{P_\eta\left(secret = x \wedge guess = x\right)}{P_\eta\left(guess = x\right)} && \leq && z \\ \equiv & P_\eta\left(secret = x \wedge guess = x\right) && \leq && z * P_\eta\left(guess = x\right) \end{aligned} \tag{5.2}$$

(con z siendo la variable que caracteriza a $P_\eta(secret = x)$). Para calcular $P_\eta(secret = x \wedge guess = x)$ y $P_\eta\left(guess = x\right)$, debemos encontrar en el polinomio que representa la suma de las probabilidades de los caminos que alcanzan un estado donde el cliente elige el servidor A y el adversario adivina correctamente; y el polinomio que representa la suma de las probabilidades de los caminos que alcanzan un estado donde el adversario cree que el cliente eligió el servidor A, respectivamente.

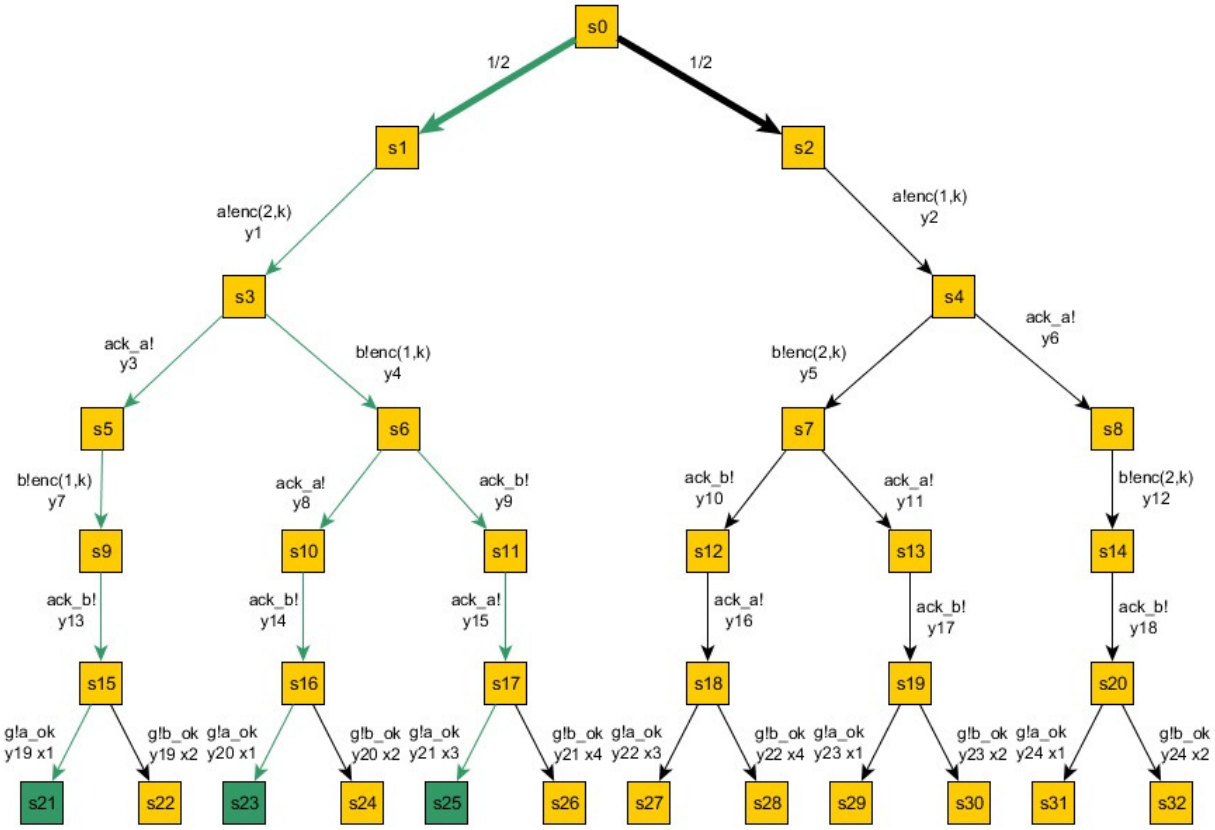

Fig. 3. Scheduler paramétrico η. Los caminos coloreados, son aquellos que alcanzan estados donde el secreto es 1 y el adversario adivina correctamente.

$$\begin{aligned} Poly\,(cl.secret = 1 \wedge adv.guess = 1) = \frac{1}{2} * y_1 * y_3 * y_7 * y_{13} * y_{19} * x_1 + \\ \frac{1}{2} * y_1 * y_4 * y_8 * y_{14} * y_{20} * x_1 + \\ \frac{1}{2} * y_1 * y_4 * y_9 * y_{15} * y_{21} * x_3. \end{aligned}$$

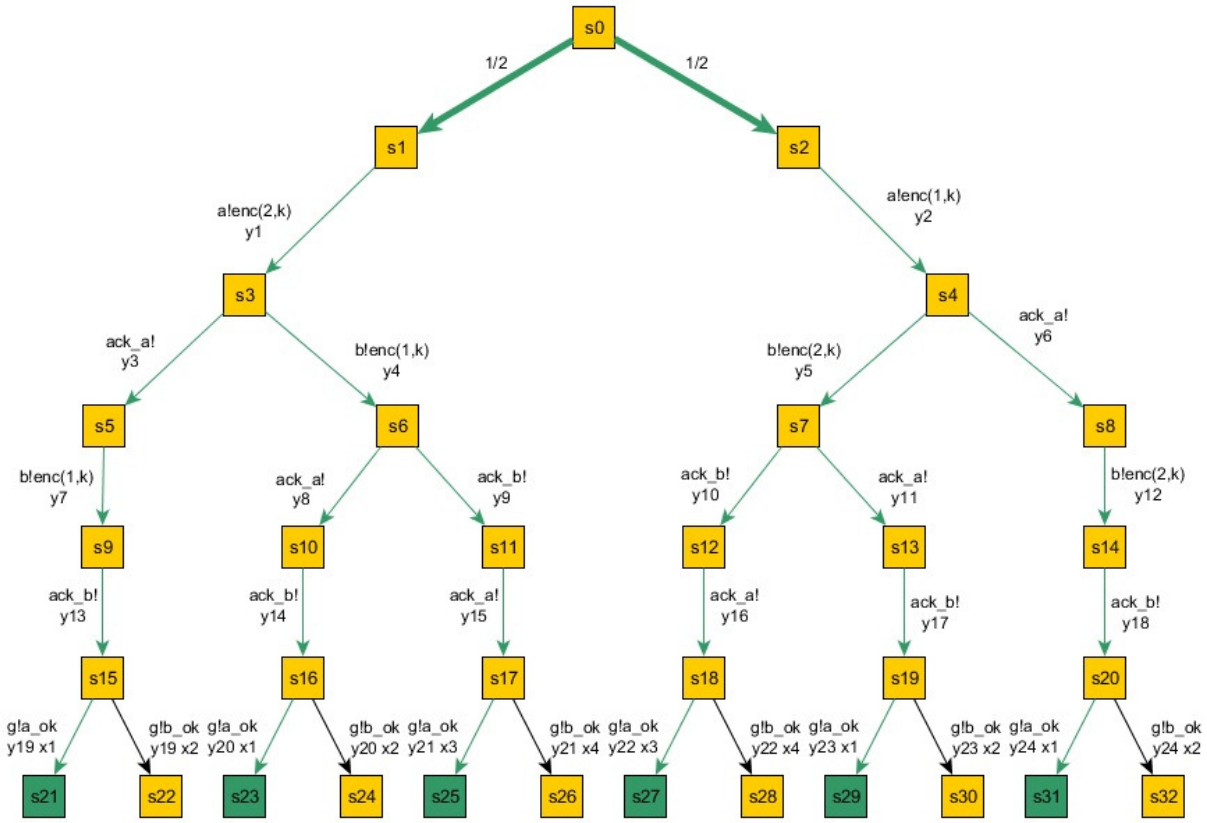

Fig. 4. Scheduler paramétrico η. Los caminos coloreados, son aquellos que alcanzan estados donde el adversario piensa que el secreto es 1.

$$\begin{aligned} Poly\,(adv.guess = 1) = {} & \frac{1}{2} * y_1 * y_3 * y_7 * y_{13} * y_{19} * x_1 + \frac{1}{2} * y_1 * y_4 * y_8 * y_{14} * y_{20} * x_1 + \\ & \frac{1}{2} * y_1 * y_4 * y_9 * y_{15} * y_{21} * x_3 + \frac{1}{2} * y_2 * y_5 * y_{10} * y_{16} * y_{22} * x_3 + \\ & \frac{1}{2} * y_2 * y_5 * y_{11} * y_{17} * y_{23} * x_1 + \frac{1}{2} * y_2 * y_6 * y_{12} * y_{18} * y_{24} * x_1. \end{aligned}$$

siendo $z = \frac{1}{2}$ en particular. Entonces tenemos que la expresión (5.2) queda de la siguiente forma:

$$Poly\,(cl.secret = 1 \wedge adv.guess = 1) \leq z * Poly\,(adv.guess = 1) \tag{5.3}$$

lo mismo para el caso del servidor 2:

$$Poly\,(cl.secret = 2 \wedge adv.guess = 2) \leq (1 - z) * Poly\,(adv.guess = 2) \tag{5.4}$$

Como la propiedad cumple la expresión (5.1), tenemos que las expresiones (5.3) y (5.4) deben satisfacerse para toda evaluación. El objetivo de Clouseau es entonces, tratar

de encontrar una valuación que ***NO*** satisfaga alguna de las expresiones (5.3) y (5.4). Mas formalmente:

$$\exists\ una\ valuación \mid \begin{array}{l} Poly\,(cl.secret = 1 \wedge adv.guess = 1) \geq z * Poly\,(adv.guess = 1) \vee \\ Poly\,(cl.secret = 2 \wedge adv.guess = 2) \geq (1 - z) * Poly\,(adv.guess = 2) \end{array} \tag{5.5}$$

Pero no hay que desesperar. Clouseau posee una sintaxis muy sencilla para expresar las propiedades de *anonimidad*:

$$\textbf{\textit{anonimidad}}\,(g1, g2, p1 \wedge g1, p2 \wedge g2)\ \ con\ g1, g2, p1 \wedge g1, p2 \wedge g2 \in BoolExp_{Int \cup Var}$$

Ejemplo 5.7.1. *Para comprender mejor la notación, presentamos un ejemplo posible:*

$$\begin{array}{l} \textbf{\textit{anonimidad}}(adv.guess = 1, \\ \qquad adv.guess = 2, \\ \qquad cl.secret = 1\ \&\&\ adv.guess = 1, \\ \qquad cl.secret = 2\ \&\&\ adv.guess = 2) \end{array}$$

siendo secret $\in$ Var de cl y guess $\in$ Var de adv.

De esta forma le otorgamos a Clouseau la información necesaria para construir la expresión (5.5). Hasta ahora hemos vista un solo tipo de propiedad, mientras que en el capítulo 2 mencionamos dos tipos posibles. La otra forma de expresar una propiedad se llama *alcanzabilidad probabilista* y su notación es la siguiente:

$$\textbf{\textit{max-reach}}(bool)\ con\ bool\ \in\ BoolExp_{Int \cup Var}$$

Hay que dejar en claro que para poder distinguir las distintas variables de cada componente, estas deberán ir acompañada de un prefijo que indique la componente a la cual esa variable pertenece, seguida de un punto y el nombre de la variable.

Ejemplo 5.7.2. $\textbf{\textit{max-reach}}(NameComp.NameVar == 1)$

Notar que para el caso *alcanzabilidad probabilista* ingresamos una sola expresión booleana, mientras que para el caso *anonimidad* ingresamos cuatro expresiones. Esto quiere decir que Clouseau deberá chequear cada una de las expresiones para cada camino y construir cuatro probabilidades diferentes para armar la propiedad.

5.8. Un Ejemplo a Analizar

Para comprender mejor lo aprendido, expondremos un modelo en el cual un cliente debe realizar una elección entre dos servidores. En este ejemplo, hemos tratado de exponer las distintas consideraciones que hemos nombrado a lo largo del capítulo. Notar que solo se ha modelado el cliente y que los supuestos servidores escucharan por los canales C y D respectivamente.

instance CLIENT = CLIENT (C!,D!,1,2)

proctype CLIENT (A!,B!,keyA,keyB)

init $\rightarrow$ 0.5: packA = 2, packB = 1 + 0.5: packA = 1, packB = 2;

[A ! enc(packA, keyA)] step == 0 $\rightarrow$ step = 1;

[B ! enc(packB, keyB)] step == 1 $\rightarrow$; stop;

end

max-reach(CLIENT.packA == 2)

- Primero notar que la instancia fue declarada antes que el prototipo, esto es para dejar en claro que no importa el orden en que se declaren las instancias y los prototipos. Al contrario, la declaración de la propiedad sí importa, y debe ir siempre al final.
- Como dijimos en las secciones anteriores, el nombre de la instancia puede ser igual al nombre del prototipo, pero no pueden existir dos prototipos con el mismo nombre o dos instancias con el mismo nombre.
- A, B, keyA y keyB son los parámetros del prototipo donde A y B $\in$ *OutNames* son nombres de canales de salida, y keyA junto con keyB son constantes.
- C, D, 1 y 2 son los parámetros de la instancia donde C y D $\in Chan^O$ son canales de salida, y 1 y 2 $\in$ *Val* son valores.
- Notemos que la palabra ***step*** no está declarada en ningún lado, el lenguaje la tratará como una variable en *Var* inicializada con valor 0.
- Gracias al item anterior, es claro que la primera transición está habilitada al comienzo de la ejecución pero la segunda transición no.
- Al ejecutar la primera transición, la instancia manda la expresion enc(packA, 1) por el canal C, mientras que la variable step adquiere el valor 1, por lo que dicha transición no está más habilitada, siendo así que la segunda transición queda habilitada.
- Recordar que al ejecutar una transición con la palabra stop, todas las transiciones activas quedan deshabilitadas. Notar entonces que si la segunda transición no tuviera la palabra stop, podría ejecutarse indefinidamente.

5.9. Conclusión

En este capítulo, hemos descrito brevemente la sintaxis del lenguaje de entrada para la herramienta Clouseau. Se describió tanto el lenguaje que permite modelar el comportamiento del sistema de manera operacional, como las primitivas que permiten describir las propiedades de alcanzabilidad y de anonimidad.

Capítulo 6

Detalles de la Herramienta

"No tengo ningún talento especial, yo solo soy apasionadamente curioso".

Albert Einstein, 1879 - 1955

Clouseau se compone de 4 módulos:

- El *parser del lenguaje* visto en el capítulo 5: Se encarga de exigir que se cumplan la sintaxis y las reglas del lenguaje.
- El *constructor de restricciones y del polinomio probabilista resultante*: Se encarga de construir el árbol de caminos asignando las variables probabilistas correspondientes a cada paso de ejecución, creando las restricciones necesarias y adquiriendo los caminos que cumplen con la propiedad exigida.
- El *optimizador matemático* visto en el capítulo 4: Se encarga de reducir las expresiones matemáticas tales como las restricciones y el polinomio resultante. También se encarga de eliminar las restricciones cuyas variables no aparecen en el polinomio.
- Por último el *exportador*: Se encarga de transcribir los resultados a un formato estandar, para ser procesados por distintos resolvedores.

En este capítulo hablaremos mas bien del segundo módulo, el constructor de restricciones y del polinomio resultante. Este módulo recoge los datos del parser, crea el modelo formal y lo pone en ejecución. A continuación listaremos los objetos que manipula el módulo:

- ***Goal***: Se encarga de manipular los caminos que cumplen con la propiedad deseada.

- ***Path***: Es la abstracción de un camino. Este objeto ha sido dotado con un atributo *probability*, que guardará la probabilidad de ejecutar dicho camino.
- ***Scheduler***: Es la abstracción del scheduler distribuido bajo $\sim$.

Como vimos en el capítulo 2, las componentes y las transiciones pueden ser vistas como 4-uplas, por lo que fueron modeladas como *arreglos*. Habíamos dicho también que un estado es un mapeo $s : Var \rightarrow Val$, es decir, un conjunto de asignaciones de variables. Esto ha sido modelado como un *diccionario* donde las claves son las variables, y los valores del diccionario son los valores de las variables en Val. También tendremos una lista *pending_paths*, que es un arreglo donde se irán guardando todos los caminos creados por el modelo.

6.1. Clouseau en Acción

A continuación observaremos como Clouseau se ejecuta y qué cosas hay que tener presentes a la hora de ejecutarlo. Para explicarlo mejor, hemos dividido la ejecución de Clouseau en fases.

6.1.1. Fase 1

Se revisan las distribuciones iniciales de cada componente y se crean todos los caminos iniciales posibles.

Ejemplo 6.1.1. *Si una componente P_1 tiene una distribución inicial compuesta de dos terminos, i_0 e i_1 con probabilidad 0,5 c/u, y tenemos otra componente P_2 cuya distribución inicial elige a i_2 con probabilidad 1; entonces los caminos iniciales son: (i_0, i_2) e $(i_1, i_2) \in init_1 \times init_2$ con probabilidad 0,5 c/u. Ver definición 2.4.1.*

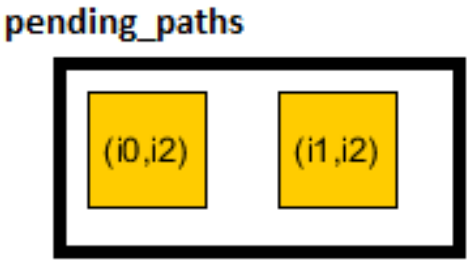

Fig. 5

Estos caminos encabezan la lista *pending_paths*. Los caminos iniciales, en definitiva, están compuestos solamente por un estado, que resulta de la composición de las distribuciones iniciales de cada componente en el modelo.

6.1.2. Fase 2

El objeto *Goal* verifica el primer camino, si $last\,(\sigma)$ satisface la expresión booleana que la caracteriza, lo añade a su lista y pasa a verificar el segundo camino. En cambio, si no satisface la guarda de la propiedad, pasa a la fase 3.

6.1.3. Fase 3

En esta fase se calcula cuales son las componentes habilitadas, es decir, que componentes poseen transiciones activas habilitadas (al menos 1). Si las componentes habilitadas para un camino son mayores o iguales a 2, entonces el objeto *Scheduler* asigna una variable probabilista a c/u y se crea una restricción donde dice que la suma de dichas variables debe ser igual a 1. Al contrario, si existe una única componente habilitada se le asigna probabilidad 1.

Ejemplo 6.1.2. *Si para el camino $s_0 l_1 s_1$ tengo 3 componentes habilitadas $c_1, c_2 y c_3$, y la probabilidad de elegir c_1 es y_1, la probabilidad de elegir c_2 es y_2 y la probabilidad de elegir c_3 es y_3; entonces tenemos la restricción $y_1 + y_2 + y_3 = 1$.*

Además, si el número de componentes habilitadas es mayor o igual que 2, como en el ejemplo anterior, se crean unas restricciones no lineales que serán útiles a la hora de establecer equivalencias (ver sección 4.3). Estas equivalencias, surgen de las restricciones (3.2) del Scheduler de Interleaving en la sección 3.2.4.

Ejemplo 6.1.3. *Restricciones no-lineales correspondientes al ejemplo anterior:*

$y_1 = (y_1 + y_2) * z_1$
$y_2 = (y_1 + y_2) * z_2$
$y_1 = (y_1 + y_3) * z_3$
$y_3 = (y_1 + y_3) * z_4$
$y_2 = (y_2 + y_3) * z_5$
$y_3 = (y_2 + y_3) * z_6$

Las variables y_i son asignadas por el scheduler de interleaving. Las variables z_i son asignadas por el scheduler distribuido bajo $\sim$.

6.1.4. Fase 4

Por cada transición de cada componente habilitada se crea un nuevo camino utilizando como prefijo el camino anterior. La probabilidad del nuevo camino será la probabilidad del antiguo multiplicada por la probabilidad de ejecutar la componente, multiplicada por la probabilidad de ejecutar la transición. Si esta última es menor que uno, es decir, las transiciones habilitadas para la componente *Comp* son mayores o iguales a dos, entonces el *scheduler local* asigna una variable probabilista a cada transición y crea un restricción tal que la suma de dichas variables sea igual a uno (ej: $x_1 + x_2 = 1$). Por

último, estos nuevos caminos son encolados al final de la lista *pending_paths* y retomamos la fase 2, con el próximo camino de la lista.

Ejemplo 6.1.4. *Tomamos como muestra el primer camino del Ejemplo 6.1.1. Supongamos que tiene dos componentes habilitadas solamente por una transición c/u. La variable* ***y1*** *corresponde a la probabilidad de elegir la primer componente habilitada y la variable* ***y2*** *corresponde a la probabilidad de elegir la segunda.*

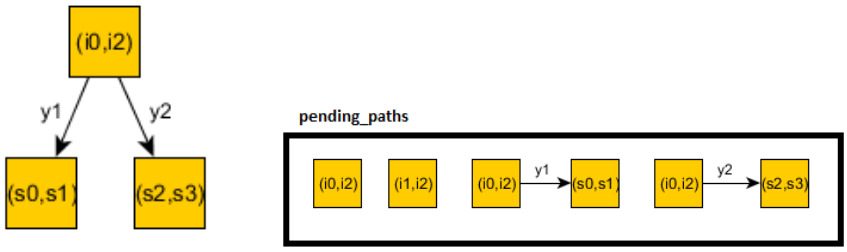

Fig. 6

Después de construir todos los caminos posibles, chequeamos que alguno de ellos cumpla con la expresión booleana que caracteriza a la propiedad. De no ser así, Clouseau devuelve una excepción diciendo que la probabilidad de que un camino alcance un estado donde la guarda de la propiedad se cumpla, es cero. En caso contrario, avanzamos a la fase 5.

6.1.5. Fase 5

El objeto *Goal* crea el polinomio probabilista sumando la probabilidad de cada camino que cumple con la propiedad (Ver sección 4.4). En el caso de propiedades de *anonimidad*, *Goal* creará cuatro polinomios distintos. Una vez creados los polinomios, son transferidos a un archivo de texto junto con el conjunto de restricciones creadas durante el unfolding del modelo, para ser procesados por el *optimizador matemático* (Ver sección 4.5).

6.2. Exportador

Una vez reducido el polinomio a su mínima expresión y eliminadas las restricciones innecesarias, el *exportador* toma el control para traducir el archivo de manera que pueda ser pasado a un smt solver o para ser analizado por otros resolvedores donde se decidirá si el modelo contiene alguna falla imprevista, y en el caso de poseerla indicar cual es.

El *módulo exportador* es muy útil ya que provee versatilidad a la herramienta, permitiendo salidas en numerosos formatos estándares. A continuación explicaremos algunos de los siguientes formatos:

- ***smt***: El problema de las teorías del modulo de satisfacibilidad (SMT), es un problema de decisión de fórmulas lógicas con respecto a las combinaciones de las teorías de fondo expresadas en lógica clásica de primer orden con igualdad. SMT puede ser considerado como una forma del problema de satisfacción de restricciones y por lo tanto un cierto enfoque formalizado de programación de restricciones. Un SMT solver aplica estas teorías para resolver distintos problemas de alta complejidad expresados en lenguaje SMT. En particular trabajamos con z3, un SMT solver fabricado por Microsoft [2].

- ***redlog***: Redlog es una parte integral del sistema de reducción interactiva del álgebra computacional. Este suplemento reduce la amplia colección de métodos de gran alcance de computación simbólica mediante el suministro de más de 100 funciones en fórmulas de primer orden. Redlog ha estado a disposición del público desde el año 1995 y está siendo constantemente mejorado. El nombre Redlog significa Reductor de Sistemas Lógicos [3].

- ***qepcad***: QEPCAD es una implementación de eliminación de cuantificadores por descomposición algebraica cilíndrica parcial debido originalmente a Hoon Hong, y posteriormente se añade a otros muchos. Es un programa de línea de comandos interactiva escrito en C, y se basa en la biblioteca SACLIB de funciones de álgebra computacional [4].

- ***nlopt***: NLopt es una biblioteca de optimización no lineal escrito en C por Steven G. Johnson y con licencia LGPL. Además, cuenta con algunos solucionadores escritos por otros autores y conectados al paquete, algunos de ellos fueron traducidos de Fortran por f2c. Varios de los mejores solucionadores NLopt se han conectado a nuestro software y por lo tanto pueden ser utilizados para resolver problemas codificados en OpenOpt (como las funciones de Python) o modelos FuncDesigner con diferenciación automática. Tal vez, algunos más se añadirán en el futuro [5].

- ***ipopt***: IPOPT (Optimizador de punto interior) es una biblioteca de software para la optimización no lineal a gran escala de sistemas continuos. Este es uno de los solucionadores NLP (no lineales) disponibles en OpenOpt [6].

- ***texto plano***: Simplemente exporta el polinomio probabilístico con las restricciones en un archivo de texto plano para ser leído por el usuario en caso de necesidad.

Vale aclarar que si decidimos expresar la propiedad en forma de *anonimidad*. Solo la podremos exportar en formato ***smt***. Todos los demás formatos admiten exclusivamente la forma *alcanzabilidad probabilista*.

Un detalle a tener en cuenta, es que cuando ejecutamos el módulo *exportador*, deberemos pasar como parámetro la probabilidad esperada de que la propiedad se cumpla mediante el comando -p o –prob. Si la propiedad es de *anonimidad*, entonces la probabilidad pasado por parámetro será z.

6.3. Conclusión

En este capítulo hemos descrito la secuencia de ejecución de Clouseau, como este manipula los conocimientos vistos en los capítulos anteriores y como realiza el unfolding del modelo. También describimos como implementamos las distintas herramientas, así como también las entidades que participan e interactúan en el mismo. Por ultimo, solo faltaría validar Clouseau sobre un modelo concreto, lo cual realizaremos en el capítulo siguiente.

Capítulo 7

Casos de Estudio

"No debemos tener miedo de confrontarnos: hasta los planetas chocan, y del caos nacen las estrellas"

Charles Chaplin, 1889-1977

7.1. El Problema

En este capítulo veremos una aplicación de la herramienta para solucionar uno de los problemas más clásicos. Se trata de la *cena de los criptógrafos* introducido por primera vez en [12]:

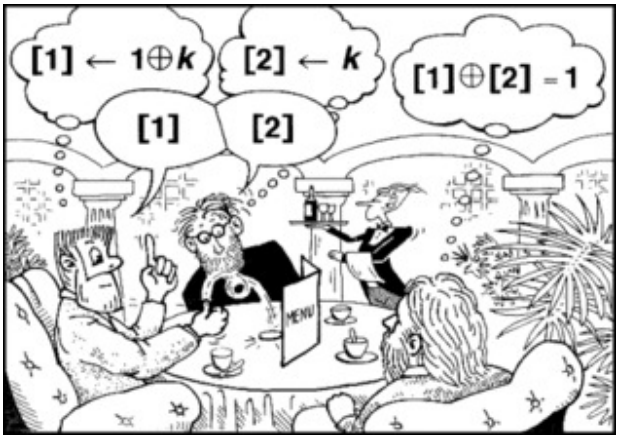

Fig. 7: `http://plaintext.crypto.lo.gy/en/article/354/ali-baba-waldo-and-the-dining-cryptographers`

Tres criptógrafos están sentados en una mesa de restaurante, el problema se presenta a la hora de pagar la cuenta. Cualquiera de los tres criptógrafos puede pagar de

forma anónima, pero también la cena puede ser pagada por la agencia de seguridad nacional (NSA). Los criptógrafos quieren saber si pagó la NSA o pagó alguno de ellos.

Para averiguarlo, el criptógrafo *A* tira una moneda en forma secreta con el criptógrafo *B* y una moneda en forma secreta con el criptógrafo *C*. El criptógrafo *B* por su lado también tirará una moneda en forma secreta con el criptógrafo *C*. Cada criptógrafo tendrá entonces dos valores provenientes de las dos monedas tiradas con ambos criptógrafos.

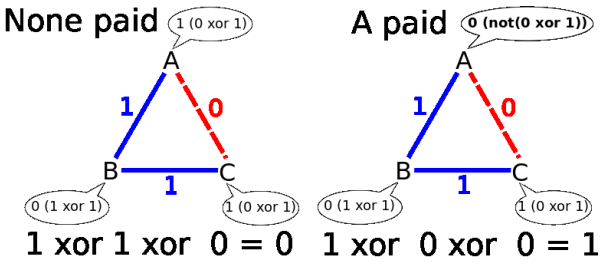

Fig. 8: `http://commons.wikimedia.org/wiki/File:Dining_Cryptographers.svg`

Luego cada criptógrafo hará un ***XOR*** con ambos valores. Si uno de los criptografos fue el responsable del pago, entonces dicho critógrafo deberá ***NEGAR*** el valor obtenido. Mas tarde cada criptógrafo publica su valor, realizando un ***XOR*** de los tres valores y obteniendo como resultado 0 si la NSA fue la autora del pago, u obteniendo 1 si algún criptógrafo pagó la cuenta.

7.2. El Modelo

Para esto, hemos escrito el modelo del problema para que Clouseau pueda analizarlo. Este modelo es ligeramente diferente al problema debido a su complejidad y a lo que queremos probar. Nos interesa particularmente el caso donde alguno de los criptógrafos pagó la cuenta. Queremos probar si cualquier persona que escucho la conversación pública de los criptógrafos, puede adivinar quien pagó con probabilidad mayor a 1/3. Debido a esto, quitamos a la NSA y agregamos una componente ATTACK con capacidad de escuchar lo que los criptografos publican.

proctype MASTER (c1! ,c2! ,c3! ,k1 ,k2 ,k3)

init → 1/3: m1=1 + 1/3: m2=1 + 1/3: m3=1;

[c1 ! enc(m1,k1)] s==0 → s=1;

[c2 ! enc(m2,k2)] s==1 → s=2;

[c3 ! enc(m3,k3)] s==2 → s=3;

end

proctype CRYPT (c? ,n! ,n? ,a! ,a1? ,a2? ,kc ,kn)

init $\rightarrow$ 1/2: coin = 1 + 1/2: coin = 2;

[n ? msg] s1 == 0 $\rightarrow$ ncoin = dec(msg,kc), s1 = 1;

[c ? msg] s == 0 $\rightarrow$ me = dec(msg,kc), s = 1;

[n ! enc(coin, kn)] s == 1 $\rightarrow$ s == 2;

[a ! me] (s == 2) && (s1 == 1) && (ncoin == coin) $\rightarrow$ s = 3;

[a ! 1] (s == 2) && (s1 == 1) && (me == 0) && (ncoin != coin) $\rightarrow$ s = 3;

[a ! 0] (s == 2) && (s1 == 1) && (me == 1) && (ncoin != coin) $\rightarrow$ s = 3;

end

proctype ATTACK (a1?, a2?, a3?)

init $\rightarrow$;

[a1 ? msg] s1 == 0 $\rightarrow$ s1 == 1;

[a2 ? msg] s2 == 0 $\rightarrow$ s2 == 1;

[a3 ? msg] s3 == 0 $\rightarrow$ s3 == 1;

(s1 != 0) && (s2 != 0) && (s3 != 0) && (g == 0) $\rightarrow$ g == 1;

(s1 != 0) && (s2 != 0) && (s3 != 0) && (g == 0) $\rightarrow$ g == 2;

(s1 != 0) && (s2 != 0) && (s3 != 0) && (g == 0) $\rightarrow$ g == 3;

end

instance m = MASTER (c1! ,c2! ,c3! ,1 ,2 ,3)

instance crypt1 = CRYPT (c1? ,cn2! ,cn1? ,a1! ,a2? ,a3? ,1 ,2)

instance crypt2 = CRYPT (c2? ,cn3! ,cn2? ,a2! ,a1? ,a3? ,2 ,3)

instance crypt3 = CRYPT (c3? ,cn1! ,cn3? ,a3! ,a1? ,a2? ,3 ,1)

instance att = ATTACK (a1? , a2? ,a3?)

max-reach(((att.g == 1) && (m.m1 == 1)) || ((att.g == 2) && (m.m2 == 1)) ||
((att.g == 3) && (m.m3 == 1)))

7.3. Mediciones

Como se puede esperar, hemos dejado que Clouseau analice nuestro modelo exactamente como lo hemos descrito arriba. Recordemos que Clouseau nos responde con un polinomio probabilista que representa la probabilidad de la propiedad y sus restricciones, para ser analizados por un resolvedor. En este caso particular, Clouseau retornó:

- $Poly = \frac{1}{3}$

- *Restricciones* $= \emptyset$

siendo que el módulo constructor del polinomio demoró 14 *seg*, el módulo optimizador demoró 11 *seg* y el módulo exportador demoró 2 *seg*. Esto nos dice que no existe la posibilidad de que algún atacante acierta en mayor medida que eligiendo al azar, solo con los conocimientos públicos de los criptógrafos.

Pero quizás si le otorgamos al atacante la capacidad de que escuche por los canales secretos de los criptógrafos, podría tener ventaja. Para esto modificamos el prototipo del atacante de la siguiente manera:

proctype ATTACK (a1?, a2?, a3?, cn1?, cn2?, cn3?)

init $\rightarrow$;

[a1 ? msg] s1 == 0 $\rightarrow$ s1 == 1;

[a2 ? msg] s2 == 0 $\rightarrow$ s2 == 1;

[a3 ? msg] s3 == 0 $\rightarrow$ s3 == 1;

(s1 != 0) && (s2 != 0) && (s3 != 0) && (g == 0) $\rightarrow$ g == 1;

(s1 != 0) && (s2 != 0) && (s3 != 0) && (g == 0) $\rightarrow$ g == 2;

(s1 != 0) && (s2 != 0) && (s3 != 0) && (g == 0) $\rightarrow$ g == 3;

end

y su instancia:

instance att = ATTACK (a1? , a2? , a3?, cn1?, cn2?, cn3?)

de esta forma tenemos que la cantidad de variables que aparecen en el polinomio y en las restricciones son 3674 y la cantidad de restricciones son 1302. El módulo constructor del polinomio demoró 14 *seg*, el módulo optimizador demoró 1 *minuto y* 47 *seg*. Esta respuesta ha sido entregada a z3 para su análisis, pero este devuelve un fallo de memoria en breve tiempo.

7.4. Conclusión

Finalmente hemos comprobado Clouseau mediante el ejemplo clásico de *la cena de los criptógrafos*. Si bien Clouseau funcionó bien con una versión básica del problema, no pudimos obtener respuesta al incrementar la dificultad del modelo. Esto se debe a que z3 no pudo computar el problema de optimización polinomial por la escasez de memoria. Esto se podría solucionar comprobando el modelo en un computador con mayor memoria.

Capítulo 8

Conclusión

Hemos presentado en este trabajo una herramienta que trabaja sobre sistemas distribuidos probabilistas no deterministas para analizar (automáticamente) propiedades expresadas como alcanzabilidad. También se presentó un algoritmo encargado de reducir el problema de seguridad en un problema de optimización polinomial, este algoritmo fue introducido por primera vez en [11]. Además presentamos un lenguaje práctico para trabajar sobre sistemas de estas características que posee la propiedad de ser reutilizado en otras herramientas que también necesiten trabajar sobre estos sistemas.

Al comprobar Clouseau mediante el caso de estudio de la cena de los criptógrafos, pudimos notar que para los modelos pequeños o de pequeña dificultad Clouseau nos devuelve el resultado correcto. Esto también se dio en modelos como el problema del cliente que elige dos servidores, que no fueron introducidos como casos de estudio pero que sí han sido modelados para ayudar a testear la herramienta. Sin embargo, al incrementar la dificultad, Clouseau retorna un problema de optimización polinomial con muchas variables y restricciones, tanto lineales como no lineales, lo cual demanda mucha memoria de trabajo para que z3 pueda computarlo. Debido a esto, nos vimos limitados por la escasez de memoria, lo cual obstaculiza el procesamiento de grandes volúmenes de información, así como también por la velocidad del lenguaje interpretado *Ruby* en que se implemento la herramienta.

Una solución cercana sería ejecutar Clouseau en computadores con alta cantidad de memoria, pero como podremos intuir no es la mejor solución. Las opciones a considerar en un futuro serían:

- Incrementar las estrategias para obtener un problema de optimización polinomial más reducido.
- Transcribir Clouseau a un lenguaje de más bajo nivel como C, lo cual daría más velocidad de procesamiento, necesarios para modelos de alta complejidad. Mientras más complejo es el modelo, más tiempo tarda Clouseau en procesarlo. Esto no soluciona el problema de complejidad de la salida.
- Hacer que z3 utilice la memoria del disco duro como memoria RAM. Para esto

habría que crear un módulo que le permita a z3 hacer un swap más eficiente que el swap propio de la maquina escribiendo en el disco duro información no necesaria en la brevedad.

Estas son algunas de las posibles opciones para mejorar la herramienta. No obstante, no existen en el mercado herramientas de esta índole que trabajen con probabilidades y no determinismo, lo cual hace a Clouseau una potencial herramienta para la verificación de protocolos y sistemas distribuidos que hasta el momento se analizan manualmente.

Bibliografía

[1] George Coulouris, Jean Dollimore, y Tim Kindberg. *Sistemas Distribuidos: Conceptos y Diseños.* Addison Wesley.

[2] Leonardo de Moura y Nikolaj Bjorner. *z3: An Efficient SMT Solver.* Microsoft Research.

[3] Andreas Dolzmann y Thomas Sturm. `http://www.redlog.eu`

[4] Hoon Hong y Christopher W. Brown. `http://www.usna.edu/CS/ qepcad/B/QEPCAD.html`

[5] Steven G. Johnson. `http://openopt.org/nlopt`

[6] Andreas Wächter. `http://openopt.org/IPOPT`

[7] Silvia S. Pelozo y Pedro R. D'Argenio. *Security analysis in probabilistic distributed protocols via bounded reachability.* CONICET - FaMAF - Universidad Nacional de Córdoba.

[8] Giro, S.: On the automatic verification of distributed probabilistic automata with partial information. Ph.D thesis, Universidad Nacional de Córdoba (2010).

[9] Giro, S., D'Argenio, P.R.: Quantitative model checking revisited: Neither decidable nor approximable. In: FORMATS. LNCS, vol.4763, pp. 179-194. Springer (2007).

[10] Giro, S., D'Argenio, P.R.: *On the expressive power of schedulers in distributed probabilistic system.* Electron. Notes Theor. Comput. Sci. 253(3), 45-71 (2009).

[11] Calin, G., Crouzen, P., D'Argenio, P.R., Hahn, E., Zhang, L.: *Time-bounded reachability in distributed input/output interactive probabilistic chains.* In: Model Checking Software, LNCS, vol. 6349, pp. 193-211. Springer (2010), extended version: AVACS Technical Report No. 64, June 2010.

[12] David Chaum: *The dinning cryptographers problem: Unconditional sender an recipient untraceability.* (1988).