Testing de Sistemas de Caja Negra especificados con Redes de Petri

Tesina de grado presentada
por

Hernán Ponce de León
P-2979/3

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de

Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Av. Pellegrini 250, Rosario, República Argentina

Marzo 2011

Supervisor

Stefan Harr

Laboratoire Spécification et Vérification

Ecole Normale Supérieure de Cachan

61, avenue du Président Wilson, Cachan, France

Agradecimientos

A Emi, Bata y el Oboe, por las idas a Buenos Aires, por las fiestas de disfraces, por los Madryn, los asados.

A Leo, por sorprenderme y acompañarme en ese mes de locuras que me hicieron redescubrir mi pasión.

A Ana Clara, por ser una amiga y escucharme cuando la necesité.

A Javo, Gabo y Lau, porque los años van a seguir pasando y para ellos seguiré siendo "chiro".

A Vir, que después de TODO, sigue cerca.

A Malala, por cinco años de chantadas, sabes que siempre va a haber un espejo que te escuche.

A Grego, Andrés y toda la gente de Reconquista, porque hicieron que un lugar que no sabía que existía, fuera mi segundo hogar.

A Toty, por ser tan distinto, pero a la vez tan similar a mi y compartir muchos de mis sueños.

A Clemen, por ser asi de pesada, pero ASÍ de buena.

A Gonzalito, por ser mi "yo mas joven" y demostrarme que no soy el único bicho raro de la carrera.

A Dante, por animarse a bancarme este último tiempo.

A todos los LCC, profesores y compañeros en estos cinco (?) años de carrera y por hacer que seamos una familia y no un número más dentro de la Universidad.

A Fer, la Negra, Samy, Vanu y Mary, por ser mi segunda familia.

A Javy, por sacarme de mi mundo de fantasias.

A Stefan, mi director, por bancar las mil preguntas e iniciarme en este nuevo camino.

A Normann y Cesar, por tantos picnis en París.

A mis tías, tíos, primos y mi abuela, por todo.

A mi hermana, Magui, ocho años y 2000 kilómetros nos separan, pero sé que sos de las que estás más cerca.

Y como dicen, "los últimos serán los primeros", a las dos personas que les debo todo, que no se los diré en palabras tantas veces como debería, pero que son las personas más importantes en mi vida:

A mi Mamá, que sufrió en silencio el aprender verme volar, pero que lo logró. Seguro jamás se dio cuenta, pero el día que me dijo "con vos ya me curé de espanto", comprendí que había terminado de descubrirme.

A mi Viejo, que me bancó siempre y desde el primer momento en todo lo que me propuse, ahora sí, se me acabó la beca.

Resumen

La validación de la implementación de un sistema a través del testing es un paso importante en la realización de sistemas complejos. Incluso la verificación más cautelosa de una especificación no puede garantizar la ausencia de errores en el proceso de implementación. Dada la especificación S de un sistema según algún modelo formal y una implementación I de caja negra (aquella que sólo puede ser vista en términos de sus datos de entrada y salida sin el conocimiento de cómo trabaja internamente), intentamos derivar a partir de S secuencias de datos de entrada que nos permitan determinar a partir de los datos de salida que generan en I si I se ajusta a S. Dichas secuencias pueden permitir (i) predecir el comportamiento de los datos de salida, y en el caso de un comportamiento erróneo, (ii) encontrar la causa de dicho error.

Varios modelos han sido utilizados para especificar sistemas de caja negra, entre ellos Automatas de Entrada/Salida, Automatas de Entrada/Salida con multi-puertos, Automatas de Entrada/Salida con ordenes parciales, etc. Si bien estos formalismos permiten modelar la concurrencia del sistema, siguen siendo modelos secuenciales y heredan muchas de sus limitaciones. Por lo tanto, debemos abandonar el modelo de máquinas finitas y desarrollar un nuevo modelo para el testing de sistemas de entrada/salida implementados como caja negra.

En este trabajo presentamos una extensión de las Redes de Petri, un modelo que permite la representación de sistemas distribuidos y discretos. Gracias a este nuevo modelo hemos sido capaces de establecer una relación de implementación y diseñar un algoritmo para testearla.

Índice general

Capítulo 1

Introducción

La validación de la implementación de un sistema a través del testing es un paso importante en la realización de sistemas complejos. Incluso la verificación más cautelosa de una especificación no puede garantizar la ausencia de errores en el proceso de implementación. Dada la especificación S de un sistema según algún modelo formal y una implementación I de caja negra (aquella que sólo puede ser vista en términos de sus datos de entrada y salida sin el conocimiento de cómo trabaja internamente), intentamos derivar a partir de S secuencias de datos de entrada que nos permitan determinar a partir de los datos de salida que generan en I si I se ajusta a S. Dichas secuencias pueden permitir (i) predecir el comportamiento de los datos de salida, y en el caso de un comportamiento erróneo, (ii) encontrar la causa de dicho error.

1.1. Testing de modelos secuenciales

Un modelo basado en Automatas de Entrada/Salida (IOA) fue propuesto en [1]. Los Automatas de Entrada/Salida son máquinas de estado finito cuyas transiciones están etiquetadas con tuplas de datos de entrada/salida. Si un IOA se encuentra en un estado s, se puede disparar una transición t si los datos de entrada de t pueden ser ingresados al sistema y este produce los datos de salida especificados para dicha transición. El testing basado en modelos especificados con IOA ha sido el tema central de diversas investigaciones.

1.2. Testing utilizando multi-puertos

La naturaleza de los sistemas distribuidos representa un gran problema a todos los enfoques basados en modelos secuenciales, Los grandes avances en arquitecturas multi-núcleos y las aplicaciones distribuidas sobre redes han

hecho necesario buscar nuevos modelos que soporten las necesidades de estas nuevas demandas. Los IOA comunican la información de entrada/salida (dentro de una misma etiqueta) mediante un único puerto de comunicación. En [10] se propone una extensión de los IOA donde cada transición tiene una etiqueta que permite que los datos se comuniquen mediante varios puertos, recibiendo y enviando información en paralelo. El problema de este modelo es que se necesita o de una entidad que maneje globalmente el testing (comunicandose con cada entidad del sistema), o distribuir la arquitectura del testing colocando un tester en cada entidad.

1.3. Testing utilizando ordenes parciales

El modelo de automatas utilizando multi-puerto no se adapta completamente a los sistemas distribuidos ya que no tiene en cuenta la concurrencia entre eventos de entrada. Los trabajos [2, 3] presentan el uso Automatas de Entrada/Salida con Ordenes Parciales (POIOA) para el testing de sistemas distribuidos, un enfoque similar al de multi-puertos pero en el cual cada transición tiene asociado un orden parcial. Este trabajo no sólo proporcionaba un enfoque más adecuado para sistemas distribuidos, sino que mostraba una mejora en eficiencia.

1.4. Objetivos y Organización del trabajo

Aunque los POIOA muestran grandes ventajas con relación a los demás formalismos, heredan muchas de las limitaciones de los modelos secuenciales, por ejemplo, al transicionar de un estado s a s', es necesario que todo el patrón de eventos de entrada y salida (descriptos según la etiqueta de dicha transición) sea ejecutado antes de poder ejecutar cualquier transición que parta de s'. Por lo tanto, debemos abandonar el modelo de máquinas finitas y desarrollar un nuevo modelo para el testing de sistemas de entrada/salida implementados como caja negra. Para esto elegimos las Redes de Petri, un modelo que permite la representación de sistemas distribuidos y discretos.

El trabajo está organizado de la siguiente manera: El Capítulo 2 introduce los conceptos básicos sobre Redes de Petri. En el Capítulo 3 presentamos el modelo que utilizamos para la especificación de nuestros sistemas. El Capítulo 4 define una relación de implementación entre dos sistemas modelados bajo este formalismo y presenta un algoritmo que verifica dicha relación entre la especificación y la implementación junto con la corrección del mismo. Finalmente, el Capítulo 5 presenta algunas conclusiones del trabajo y los posibles trabajos futuros.

Capítulo 2

Definiciones básicas

2.1. Redes de Petri

Una Red de Petri es una representación matemática de un sistema distribuido discreto. Las redes de Petri fueron definidas en los años 1960 por Carl Adam Petri. Son una generalización de la teoría de autómatas que permite expresar eventos concurrentes.

Una tripleta (P, T, F) es una *Red de Petri* si el grafo dirigido $(P \cup T, F)$ es bipartito. Los elementos de P son denominados *lugares* y los elementos de T *transiciones*. Los arcos de F van de un lugar a una transición o viceversa, pero jamas de lugares a lugares o de transiciones a transiciones. Gráficamente los lugares son representados mediante círculos y las transiciones mediante cuadrados. Nos referimos indiferentemente a lugares o transiciones como nodos. Los arcos de F inducen un orden parcial. Denotamos a su clausura transitiva y refelxiva con $\leq$. Las *pre-condiciones* $({}^\bullet x)$ y *post-condiciones* $(x^\bullet)$ de un nodo x, son respectivamente los elementos de los conjuntos $\{y \in P \cup T \mid F(y, x) = 1\}$ y $\{y \in P \cup T \mid F(x, y) = 1\}$.

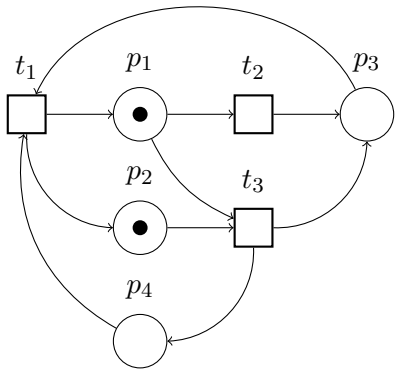

Figura 2.1: Ejemplo de una Red de Petri.

Una *marca* es un mapeo $P \to \mathbb{N}$. Gráficamente, son $M(p)$ puntos en cada $p \in P$. Toda Red de Petri tiene asociada una *marca inicial* M_0. En en ejemplo de la Figura 2.1, la marca inicial es el conjunto $\{p_1 \mapsto 1, p_2 \mapsto 1\}$. Una marca M *habilita* una transición t si $\forall p \in P : F(p,t) \leq M(p)$. Si t está habilitada en M, entonces se puede *disparar*, llevando a una nueva marca M', denotado $M \xrightarrow{t} M'$ y definido por $M'(p) = M(p) - F(p,t) + F(t,p)$ para cada lugar p. Una secuencia de transiciones $\sigma = t_1 t_2 \dots t_n$ es una *secuencia factible* si existen marcas $M_1, M_2 \dots M_n$ tales que:

$$M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{t_3} \dots M_{n-1} \xrightarrow{t_n} M_n$$

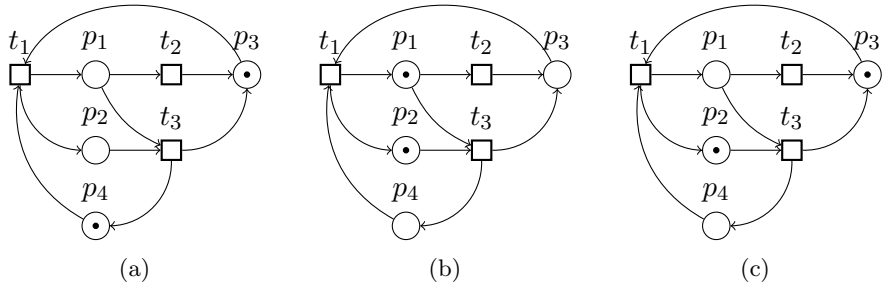

Figura 2.2: Evolución del sistema de la Figura 2.1 al disparar la secuencia $t_3 t_1 t_2$.

M_n es la marca alcanzada desde M_0 por la ocurrencia de σ (también denotado por $M_0 \xrightarrow{\sigma} M_n$). M es una *marca alcanzable* si existe una secuencia factible σ tal que $M_0 \xrightarrow{\sigma} M$. La Figura 2.2 muestra la evolución del sistema de la Figura 2.1 al disparar la secuencia factible $t_3 t_1 t_2$ ($M_0 \xrightarrow{t_3} \{p_3 \mapsto 1, p_4 \mapsto 1\} \xrightarrow{t_1} \{p_1 \mapsto 1, p_2 \mapsto 1\} \xrightarrow{t_2} \{p_2 \mapsto 1, p_3 \mapsto 1\}$).

Una marca M es *n-acotada* si $M(p) \leq n$ para cada lugar p. Si M es 1-acotada, diremos que es segura. Identificamos las marcas seguras con el conjunto de lugares p tal que $M(p) = 1$. Una Red de Petri es n-acotada si toda marca alcanzable es n-acotada y es segura si todas sus marcas lo son.

Seguiremos la notación utilizada en la mayoría de la bibliografía y denotaremos a una la Red de Petri junto con su marca inicial como $\Sigma = (N, M_0)$.

2.2. Redes Factibles

Sea $N = (P, T, F)$ una Red de Petri. Dos transiciones t_1 y t_2 están en *conflicto directo* ($t_1 \#_\mu t_2$) sii $t_1 \neq t_2$ y ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$. La Figura 2.3 muestra dos conflictos directos, $t_2 \#_\omega t_3$ y $t'_2 \#_\omega t'_3$. Dos nodos x_1 y x_2 están en *conflicto* ($x_1 \# x_2$) sii existen dos transiciones t_1 y t_2 tal que: $t_1 \leq x_1 \wedge\ t_2 \leq x_2 \wedge t_1 \#_\mu t_2$, es decir, x_1 y x_2 están en conflicto si existen dos caminos que parten del mismo nodo y divergen llevando a x_1 y x_2 (aunque puedan volver a converger).

En el ejemplo de la Figura 2.3, existe conflicto entre diversos lugares y transiciones, por ejemplo, $p'_3 \# p_4$. Decimos que dos transiciones t_1 y t_2 están en *conflicto real* ($t_1 \#_\omega t_2$) sii están en conflicto directo y existe una marca que habilita ambas transiciones. Un nodo x está en *auto-conflicto* si $x \# x$.

Una *red factible* es una Red de Petri $N = (B, E, F)$ tal que:

- para todo $b \in B, |{}^\bullet b| \leq 1$,
- el grafo $(B \cup E, F)$ es acíclico,
- N es precedido finitamente, es decir, para cada $x \in B \cup E$, el conjunto de elementos $y \in B \cup E$ tal que $y < x$ es finito, y
- para todo $x \in B \cup E, \neg(x \# x)$.

Los elementos de B y E son llamados *condiciones* y *eventos*, respectivamente. $Min(N)$ denota el conjunto de elementos minimales de $B \cup E$ respecto de $<$.

Dados dos nodos $x, y \in B \cup E$, decimos que x e y son *concurrentes* ($x \parallel y$) si no se cumplen ninguna de las siguientes: $x < y$, $y < x$, $x \# y$.

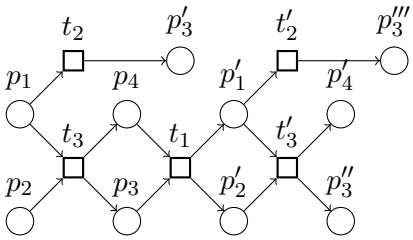

Figura 2.3: Ejemplo de una red factible.

2.3. Procesos de Ramificación

Sean $N_1 = (P_1, T_1, F_1)$ y $N_2 = (P_2, T_2, F_2)$ dos Redes de Petri. Un *homomorfismo* de N_1 a N_2 es un mapeo $h : P_1 \cup T_1 \to P_2 \cup T_2$ tal que:

- $h(P_1) \subseteq P_2$, $h(T_1) \subseteq T_2$, y
- para todo $t \in T_1$, la restricción de h a ${}^\bullet t$ es una biyección entre ${}^\bullet t$ (en N_1) y ${}^\bullet h(t)$ (en N_2), y respectivamente para $t^\bullet$ y $h(t)^\bullet$.

Es decir, un homomorfismo es un mapeo que preserva la naturaleza de los nodos y el entorno de las transiciones.

Un *proceso de ramificación* de una Red de Petri $\Sigma = (N, M_0)$ es un par $\beta = (N', \pi)$ donde $N' = (B, E, F)$ es una red factible, y π es un homomorfismo de N' a N tal que:

1. La restricción de π a $Min(N')$ es una biyección entre $Min(N')$ y M_0,
2. para cada $e_1, e_2 \in E$, si ${}^\bullet e_1 = {}^\bullet e_2$ y $\pi(e_1) = \pi(e_2)$ entonces $e_1 = e_2$.

Dos procesos de ramificación $\beta_1 = (N_1, \pi_1)$ y $\beta_2 = (N_2, \pi_2)$ de una Red de Petri son *isomorfos* si existe un homomorfismo biyectivo h de N_1 a N_2 tal que $\pi_2 \circ h = \pi_1$. Intuitivamente, dos procesos de ramificación isomorfos difieren sólo en los nombres de sus condiciones y eventos.

Sean $\beta' = (N', \pi')$ y $\beta = (N, \pi)$ dos procesos de ramificación de una Red de Petri, β' es un *prefijo* de β si N' es un subconjunto de N que satisface:

- si una condición pertenece a N', entonces sus eventos de entrada en N también pertenecen a N',
- si un evento pertenece a N', entonces sus condiciones de entrada y salida en N también pertenecen a N', y
- π' es la restricción de π a N'.

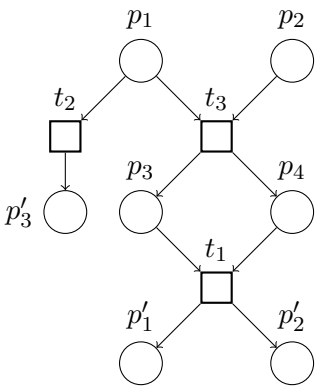

Figura 2.4: Un proceso de ramificación de la red de la Figura 2.1

Está demostrado en [6] que una Red de Petri Σ tiene un único proceso de ramificación maximal (no es prefijo de ningún otro proceso de ramificación) salvo isomorfismos. Llamamos a este proceso de ramificación el *unfolding* de Σ y lo denotamos Unf_Σ. Las Figuras 2.3 y 2.4 son dos procesos de ramificación de la red de la Figura 2.1. El unfolding de esta red es infinito dado que la red tiene ciclos.

El unfolding de una Red de Petri nos muestra como evoluciona el sistema. En [7] está explicada la manera de construir dicho unfolding. Ese proceso es infinito si la red contiene ciclos, pero de no ser así, el proceso termina.

2.4. Configuraciones y Cortes

Una *configuración* C de una red factible es un conjunto de eventos que satisface las siguientes condiciones:

- $e \in C \Rightarrow \forall e' < e : e' \in C$ (C es cerrado bajo $<$), y
- $\forall e, e' \in C : \neg(e \# e')$ (C está libre de conflicto).

Los conjuntos de eventos $\{t_2\}, \{t_3, t_1\}, \{t_3, t_1, t'_2\}, \{t_3, t_1, t'_3\}$ son cuatro configuraciones de la red factible de la Figura 2.3.

Para cada evento $e \in E$, la configuración $[e] = \{e' \mid e' \leq e\}$ es llamada *configuración local* de e. Para un conjunto de eventos $E' \neq \emptyset$, denotamos con $C \oplus E'$ el hecho de que $C \cup E'$ es una configuración tal que $C \cap E' = \emptyset$. El conjunto E' es un *sufijo* de C, y $C \cup E'$ es una *extensión* de C. Tanto $\{t'_2\}$ como $\{t'_3\}$ son sufijos de $\{t_3, t_1\}$. El conjunto de todas las configuraciones de una red factible N es denotado $Conf(N)$.

Un conjunto B' de condiciones de una red de ocurrencias es un *conjunto concurrente* si sus elementos pertenecen a la relación $\|$. Un conjunto concurrente maximal B' respecto de la inclusión de conjuntos es llamado un *corte*.

Las configuraciones finitas y los cortes están fuertemente relacionados. Sea C una configuración finita de un proceso de ramificación $\beta = (N, p)$. El conjunto concurrente $Cut(C)$, definido a continuación, es un corte:

$$Cut(C) = (Min(N) \cup C^\bullet) \backslash {}^\bullet C.$$

Particularmente, dada una configuración finita C, el conjunto de lugares $p(Cut(C))$ es una marca alcanzable a la cual denotaremos con $Mark(C)$.

Una marca M de una Red de Petri es *representada* en un proceso de ramificación β si este contiene una configuración finita C tal que $Mark(C) = M$. Es fácil probar, utilizando los resultados de [5, 6], que toda marca representada en un proceso de ramificación es alcanzable y que toda marca alcanzable es representada en el unfolding.

Para Redes de Petri seguras, tenemos el siguiente resultado:

Proposición 2.4.1 *Sean x_1 y x_2 dos nodos de un proceso de ramificación de una Red de Petri segura. Si $x_1 \| x_2$, entonces $\pi(x_1) \neq \pi(x_2)$*

Dado un corte c de un proceso de ramificación $\beta = (N, \pi)$, definimos $\Uparrow c$ como el par (N', π'), donde N' es la única sub-red de N en la cual el conjunto de nodos es $\{x \mid (\exists y \in c : x \geq y) \wedge \forall y \in c : \neg(x \# y)\}$ y π' es la restricción de π a los nodos de N'. Definimos $\pi(c)$ como el multi-conjunto que contiene una instancia del lugar $\pi(b)$ para cada $b \in c$. Tenemos el siguiente resultado:

Proposición 2.4.2 *Si β es un proceso de ramificación de (N, M_0) y c es un corte de β, entonces $\Uparrow c$ es un proceso de ramificación de $(N, \pi(c))$.*

2.5. Prefijos Completos Finitos

Decimos que un proceso de ramificación β es *completo* si para toda marca alcanzable M existe una configuración C tal que:

- $Mark(C) = M$ (M es representada en β), y
- para cada transición t habilitada en M existe una configuración $C \cup \{e\}$ tal que $e \notin C$ y e está etiquetada con t.

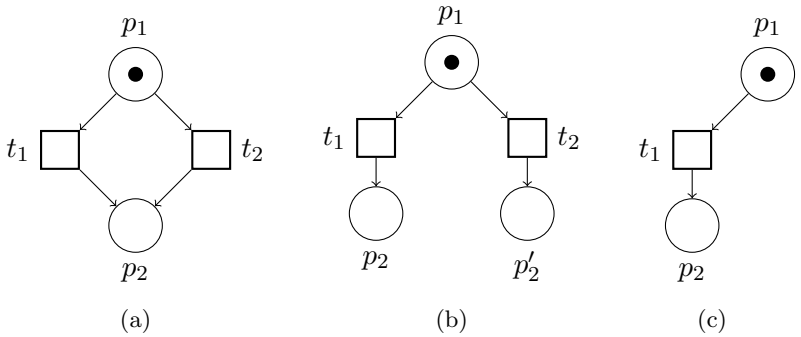

Figura 2.5: Una Red de Petri, su unfolding y un prefijo.

El unfolding de una Red de Petri es siempre completo. Un prefijo completo contiene tanta información como el unfolding, pero tiene la ventaja de ser finito.

Dado que una Red de Petri n-acotada tiene sólo un número finito de marcas alcanzables, su unfolding contiene al menos un prefijo completo.

La Figura 2.5 muestra (a) una Red de Petri, (b) su unfolding y (c) un prefijo que no es completo ya que no representa el hecho de que podemos llegar al estado p_2 disparando t_2.

Sean C_1 y C_2 dos configuraciones finitas tal que $Mark(C_1) = Mark(C_2)$. De la definición se deriva que: $\Uparrow Cut(C_i)$ es isomorfo al unfolding de $\Sigma' = (N, Mark(C_i))$, $i = 1, 2$; por lo tanto $\Uparrow Cut(C_1)$ y $\Uparrow Cut(C_2)$ son isomorfos. Aún más, existe un isomorfismo I de $\Uparrow Cut(C_1)$ a $\Uparrow Cut(C_2)$. Este isomorfismo induce un mapeo de las extensiones finitas de C_1 a las extensiones finitas de C_2.

En [8] fue propuesto un algoritmo para la construcción de un prefijo completo del unfolding de una Red de Petri segura Σ. Denotamos dicho prefijo como $Pref_\Sigma$. El siguiente resultado muestra que toda configuración de $Pref_\Sigma$ es una configuración de Unf_Σ y que podemos construir toda configuración de Unf_Σ a partir de las configuraciones de $Pref_\Sigma$.

Teorema 2.5.1 *Sea C_0 una configuración finita de $Pref_\Sigma$. El conjunto*

$$C_0 \cup I(E_1) \cup \cdots \cup I(E_n)$$

es una configuración de Unf_Σ *sii* $\exists C_1 \dots C_n \in Conf(Pref_\Sigma)$ *tal que* E_i *es un sufijo de* C_i*, y*

$$Mark(C_0 \cup I(E_1) \cup \dots \cup I(E_{i-1})) = Mark(C_i)$$

para todo $i \in \{1 \dots n\}$*.*

Dem) $\Rightarrow$) Dado que $C_0 \cup I(E_1) \cup \dots \cup I(E_{n-1})$ es una configuración de Unf_Σ, por construcción, existe una configuración C_n de $Pref_\Sigma$ tal que

$$Mark(C_0 \cup E_1 \cup \dots \cup E_{n-1}) = Mark(C_n)$$

y dado que $I(E_n)$ es una extensión de $C_0 \cup I(E_1) \cup \dots \cup I(E_{n-1})$, E_n es una extensión de C_n.

$\Leftarrow$) Sea $i \in 1..n$. Dado que C_i es una configuración de $Pref_\Sigma$ y E_i es un sufijo de ella, C_i es una configuración de Unf_Σ y E_i un sufijo de la misma (por ser $Pref_\Sigma$ una sub-red de Unf_Σ). El hecho de que

$$Mark(C_0 \cup I(E_1) \cup \dots \cup I(E_{i-1})) = Mark(C_i)$$

implica que $C_0 \cup I(E_1) \cup \dots \cup I(E_{i-1})$ es una configuración de Unf_Σ. ∎

Capítulo 3

Especificaciones para Sistemas de Caja Negra

Los sistemas de caja negra son aquellos que sólo pueden ser vistos en términos de sus datos de entrada y salida sin el conocimiento de como trabajan internamente.

A la hora de especificar Sistemas de Caja Negra se han utilizado diferentes formalismos [1]. Para los casos donde es necesario modelar la concurrencia del sistema se han propuesto formalismos como Autómatas con Multi-puerto [10] o Autómatas con Ordenes Parciales de Entrada/Salida [2]. El problema de estos formalismos es la dificultad para modelar la concurrencia, aun tienen varias limitaciones o necesitan de suposiciones muy fuertes.

3.1. Redes de Petri para Sistemas de Caja Negra

En esta sección introducimos el formalismo que utilizaremos para modelar el sistema y las suposiciones que hacemos sobre el mismo para garantizar algunas propiedades.

Sean I y O los conjuntos de símbolos de entrada y salida tal que $I \cap O = \emptyset$ y $P_I : (I \cup O) \to I$, $P_O : (I \cup O) \to O$ las proyecciones sobre la entrada y salida.

Definición 3.1.1 Sea $\Sigma = ((P, T, F), M_0)$ una Red de Petri y $\lambda : T \to (I \cup O)$ una función que etiqueta a cada transición con un símbolo de entrada o salida. Llamamos a (Σ, λ) *Red de Petri para Sistemas de Caja Negra (BBSPN)* y a $IT = \{t \in T \mid \lambda(t) \in I\}$ y $OT = \{t \in T \mid \lambda(t) \in O\}$ sus *transiciones de entrada y salida.*

El determinismo es una propiedad imprescindible en los sistemas que deseamos testear para poder estudiar su comportamiento. La Figura 3.1 muestra un sistema en el cual (a) las transiciones t_1 y t_2 se pueden disparar

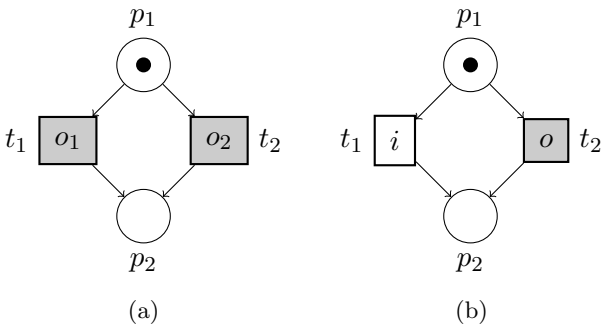

Figura 3.1: BBSPN no determinista.

indiferentemente, (b) el sistema puede esperar a que el usuario ingrese i (y disparar la transición t_1) o disparar la transición t_2. Esta forma de no determinismo no es adecuada para este tipo de sistemas pues el comportamiento de del sistema no depende unívocamente de los datos de entrada.

Definición 3.1.2 Sea (Σ, λ) una BBSPN y $M \in \mathcal{RM}(\Sigma)$, decimos que M es *determinista* sii para todo par de transiciones distintas t_1 y t_2 que no sean ambas transiciones de entrada:

$$^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset \Rightarrow {}^\bullet t_1 \not\subseteq M \vee {}^\bullet t_2 \not\subseteq M.$$

Σ es *determinista* sii todas sus marcas alcanzables lo son.

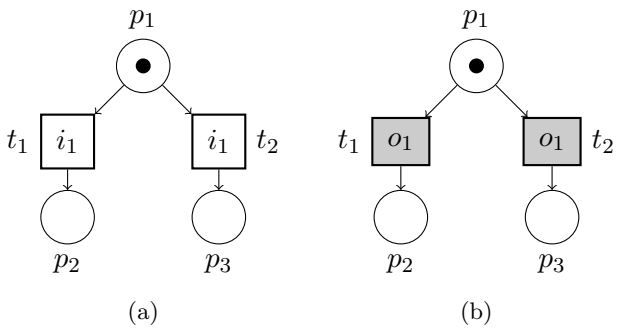

Figura 3.2: BBSPN no deterministicamente etiquetada.

Otro caso problemático, es aquel en el cual al ingresar un mismo símbolo de entrada (o ver un mismo símbolo salida), el sistema puede transicionar a dos marcas diferentes como muestra la Figura 3.2.

Definición 3.1.3 Sea (Σ, λ) una BBSPN, decimos que Σ está *determinísticamente etiquetada* sii para toda marca alcanzable M y todo par de transiciones distintas t_1 y t_2:

$$^\bullet t_1 \subseteq M \wedge {}^\bullet t_2 \subseteq M \Rightarrow \lambda(t_1) \neq \lambda(t_2).$$

La Figura 3.3 muestra un sistema que una vez que alcanza el estado $\{p_2\}$, produce infinitamente el símbolo de salida o. Este es otro tipo de comportamiento que queremos evitar en nuestro sistema.

Definición 3.1.4 Sea (Σ, λ) una BBSPN, decimos que Σ es *finita* sii para todo ciclo en Σ hay al menos una transición $t \in IT$ que pertenece al ciclo.

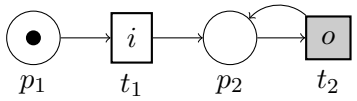

Figura 3.3: BBSPN que produce infinitos símbolos de salida.

Necesitamos una forma de diferenciar un estado del sistema en el cual el mismo está produciendo transiciones de salida de aquellos en los cuales se espera la interacción del usuario.

Definición 3.1.5 Sea M una marca, decimos que M es *estable* sii para toda transición t:

$$^\bullet t \subseteq M \Rightarrow t \in IT.$$

Definición 3.1.6 Definimos las *transiciones de entrada que preceden inmediatamente a t* como:

$$IIP_t = \{t' \in IT \mid t' < t \wedge \forall t'' \in IT : t' \leq t'' < t \Rightarrow t' = t''\}$$

.

Si bien todas estas definiciones son aplicables a las redes factibles (por ser un subconjunto de las Redes de Petri), las redefiniremos en base a sus configuraciones. Sea $\overline{\lambda} : E \to (I \cup O)$ tal que $\overline{\lambda} = \lambda \circ \pi$.

Definición 3.1.7 Sea C una configuración, decimos que C es *determinista* sii para todo par de eventos distintos e_1 y e_2 que no sean ambos eventos de entrada:

$$^\bullet e_1 \cap {}^\bullet e_2 \neq \emptyset \Rightarrow \neg(C \oplus \{e_1\}) \vee \neg(C \oplus \{e_2\}).$$

Σ es *determinista* sii todas sus configuraciones lo son.

Definición 3.1.8 Decimos que una red factible Σ está *determinísticamente etiquetada* sii para toda configuración C y todo par de eventos distintos e_1 y e_2:

$$C \oplus \{e_1\} \wedge C \oplus \{e_2\} \Rightarrow \overline{\lambda}(e_1) \neq \overline{\lambda}(e_2).$$

Definición 3.1.9 Decimos que una red factible Σ es *finita* sii para toda configuración finita C, no existe una extensión E de eventos de salida tal que $Mark(C) = Mark(C \cup E)$.

Definición 3.1.10 Sea C una configuración finita. Decimos que C es *estable* sii para todo evento e:

$$C \oplus \{e\} \Rightarrow e \in IE.$$

3.2. Unfolding de un BBSPN

Dado que el unfolding describe el comportamiento dinámico de una Red de Petri, es natural pensar que si dicha red cumple con las propiedades definidas arriba, su unfolding también deberá cumplirlas.

Teorema 3.2.1 *Si Σ está determinísticamente etiquetada, entonces Unf_Σ también lo está.*

Dem) Sean e_1 y e_2 dos eventos distintos y C una configuración tal que $\{e_1\}$ y $\{e_2\}$ son extensiones de la misma (esto implica que ${}^\bullet e_1 \subseteq Cut(C)$ y ${}^\bullet e_2 \subseteq Cut(C)$ ya que C es cerrado bajo $<$) y $\overline{\lambda}(e_1) = \overline{\lambda}(e_2)$. Entonces:

$$ {}^\bullet\pi(e_1) \subseteq \pi(Cut(C)) \tag{3.1}$$
$$ {}^\bullet\pi(e_2) \subseteq \pi(Cut(C)) \tag{3.2}$$
$$ \pi(Cut(C)) \in \mathcal{RM}(\Sigma) \tag{3.3}$$
$$ \lambda(\pi(e_1)) = \lambda(\pi(e_2)) \tag{3.4}$$

Si $\pi(e_1) \neq \pi(e_2)$, entonces $\pi(e_1)$ y $\pi(e_2)$ son dos transiciones distintas que satisfacen (3.1), (3.2), (3.3) y (3.4), por lo tanto Σ no está determinísticamente etiquetada, lo cual es una contradicción. Si $\pi(e_1) = \pi(e_2)$, entonces ${}^\bullet e_1 \neq {}^\bullet e_2$ (esto se deriva de que $e_1 \neq e_2$ y el punto ii) de la definición de proceso de ramificación) y existe un isomorfismo $\varphi : {}^\bullet e_1 \cup e_1{}^\bullet \rightarrow {}^\bullet e_2 \cup e_2{}^\bullet$ tal que:

$$ b \in {}^\bullet e_1 \Rightarrow \varphi(b) \in {}^\bullet e_2 \tag{3.5}$$
$$ b \in e_1{}^\bullet \Rightarrow \varphi(b) \in e_2{}^\bullet \tag{3.6}$$
$$ \forall b \in {}^\bullet e_1 \cup e_1{}^\bullet : \pi(b) = \pi(\varphi(b)) \tag{3.7}$$

Sea b una condición de ${}^\bullet e_1$ pero no de ${}^\bullet e_2$, (3.5) implica que $\varphi(b) \in {}^\bullet e_2$. Como $b \in {}^\bullet e_1 \subseteq Cut(C)$ y $\varphi(b) \in {}^\bullet e_2 \subseteq Cut(C)$, sabemos que $b \parallel \varphi(b)$ y por la Proposición 2.4.1, concluimos que $\pi(b) \neq \pi(\varphi(b))$, lo cual contradice (3.7). ∎

Teorema 3.2.2 *Si Σ es determinista, entonces Unf_Σ también lo es.*

Dem) Sean e_1 y e_2 dos eventos distintos y no ambos de entrada tales que ${}^\bullet e_1 \cap {}^\bullet e_2 \neq \emptyset$, y C una configuración tal que $\{e_1\}$ es un sufijo de la misma (${}^\bullet e_1 \subseteq Cut(C)$). Entonces:

$$ {}^\bullet\pi(e_1) \cap {}^\bullet\pi(e_2) \neq \emptyset \tag{3.8} $$

$$ \neg(\pi(e_1) \in IT \wedge \pi(e_2) \in IT) \tag{3.9} $$

y (3.1), (3.3) se cumplen.

Queremos probar que $\neg(C \oplus \{e_2\})$, o equivalentemente, ${}^\bullet e_2 \not\subseteq Cut(C)$. Supongamos que ${}^\bullet e_2 \subseteq Cut(C)$, entonces (3.2) se cumple. Si $\pi(e_1) \neq \pi(e_2)$, entonces $\pi(e_1)$ y $\pi(e_2)$ son dos transiciones distintas que satisfacen (3.1), (3.2), (3.3), (3.8) y (3.9), por lo tanto Σ no es determinista, lo cual es una contradicción. Si $\pi(e_1) = \pi(e_2)$, la prueba es análoga a la del Teorema 3.2.1. ∎

Resultados similares se pueden obtener para la propiedad de finitud del sistema y la estabilidad de una marca. Dada esta simetría entre la naturaleza de la Red de Petri y su unfolding, diremos que es el sistema quien es determinista, finito y está etiquetado determinísticamente.

Suposición 1 *De aquí en adelante, trabajaremos con sistemas seguros, deterministas, finitos, determinísticamente etiquetados y en los cuales su marca inicial sea estable.*

Proposición 3.2.3 *Sea (Σ, λ) una BBSPN y e_1, e_2 dos eventos de Unf_Σ. Si $e_1 \#_\omega e_2$, entonces tanto e_1 como e_2 son eventos de entrada.*

Dem) Sea $C = ([e_1] \cup [e_2])/\{e_1, e_2\}$, sabemos que no existe conflicto entre los elementos de C y e_1 o los elementos de C y e_2, y dado que $e_1 \#_\omega e_2$, hemos encontrados dos eventos distintos, tal que ${}^\bullet e_1 \cap {}^\bullet e_2 \neq \emptyset$ y una configuración tal que $\{e_1\}$ y $\{e_2\}$ son sufijos de la misma. Como el sistema es determinista, tanto e_1 como e_2 son eventos de entrada. ∎

Este resultado muestra que en los casos donde exista la posibilidad de tomar una decisión, es el usuario quien la toma y no el sistema.

Teorema 3.2.4 *Sea (Σ, λ) una BBSPN, entonces, para toda configuración no estable C de Unf_Σ, existe una única configuración C' tal que:*

1. *$C' = C \cup E'$ para algún E' tal que $C \cap E' \neq \emptyset$, y*
2. *C' es estable.*

C' es la *única extensión estable de* C y lo denotamos $C' = \mathcal{UPE}(C)$.

Dem) Dado que C no es estable, existe un evento de salida e tal que $\{e\}$ es un sufijo de C. Por la Proposición 3.2.3, cualquier otro evento e' tal que $\{e'\}$ sea un sufijo de $(C \cup \{e\})$ no está en conflicto directo con e y puede ser agregado a $C \cup \{e\}$. Esto demuestra que el orden en el cual agregamos elementos a C no es importante. Podemos continuar agregando elementos a la configuración hasta que sólo podamos agregar eventos de entrada (como Unf_Σ es finito, no podemos agregar eventos de salida infinitamente) y el conjunto de eventos que agregamos de esta manera, por construcción, es único. ∎

El Teorema 3.2.4 nos demuestra que el sistema se comporta de una manera determinista cuando no interactuamos con él, es decir, cuando sólo produce eventos de salida.

Corolario 3.2.5 *Sea (Σ, λ) una BBSPN, para todo marca alcanzable no estable M existe una única marca alcanzable M' tal que:*

1. *$M \xrightarrow{u} M'$ para algún $u \in T^*$, y*
2. *M' es estable*

Llamamos a M' la *única descendiente estable de M* ($M' = \mathcal{UPD}(M)$).

Teorema 3.2.6 *Sean e_1 y e_2 dos eventos de entrada tal que ${}^\bullet e_1 \cap {}^\bullet e_2 = \emptyset$ y C una configuración tal que $\{e_1\}$ y $\{e_2\}$ son sufijos de la misma, entonces $\{e_2\}$ es un sufijo de $\mathcal{UPE}(C \cup \{e_1\})$.*

Dem) Necesitamos probar que $\mathcal{UPE}(C \cup \{e_1\}) \cup \{e_2\}$ es cerrado bajo $<$ y está libre de conflicto.

Dado que $C \cup \{e_1\}$ es cerrado bajo $<$ y está libre de conflicto, por construcción $\mathcal{UPE}(C \cup \{e_1\})$ también lo está. Como $C \oplus \{e_2\}$, sabemos que todo evento e tal que $e < e_2$ pertenece a C, por lo tanto, también pertenece a $\mathcal{UPE}(C \cup \{e_1\})$ y $\mathcal{UPE}(C \cup \{e_1\}) \cup \{e_2\}$ es cerrado bajo $<$.

Tanto $C \cup \{e_1\}$ como $C \cup \{e_2\}$ están libres de conflicto, por lo tanto, sólo debemos probar que e_2 no está en conflicto real con los elementos de $\mathcal{UPE}(C \cup \{e_1\}) \backslash C$. Sabemos que ${}^\bullet e_1 \cap {}^\bullet e_2 = \emptyset$ y que entre ellos no existe conflicto real, y dado que todos los demás elementos de $\mathcal{UPE}(C \cup \{e_1\}) \backslash C$ son elementos de salida, por la Proposición 3.2.3, sabemos que no están en conflicto real con e_2. ∎

Esto significa que si dos eventos de entrada están habilitados, disparar uno de ellos no deshabilita al otro.

3.3. BBSPN y secuencias de eventos

Extendemos λ, $\overline{\lambda}$, P_I y P_O a funciones para secuencias de eventos $T^* \to (I \cup O)^*$, $E^* \to (I \cup O)^*$, $(I \cup O)^* \to I^*$ y $(I \cup O)^* \to O^*$, a las cuales denotamos con los mismos nombres.

Las configuraciones de una red factible respetan el orden entre los eventos de la misma, pero a la vez permite que haya eventos concurrentes. El orden en el cual se disparan dos eventos concurrentes no es importante.

Definición 3.3.1 Sea $u = u_1 \dots u_n \in E^*$ y C una configuración finita de Unf_Σ, decimos que u es un *orden* de C sii $u_i \in C$ para todo i y $u_i < u_j$ implica $i < j$.

Definición 3.3.2 Sea (Σ, λ) una BBSPN y $u \in E^*$. Decimos que Σ *acepta* $\alpha = P_I(\lambda(u)) \in I^*$ y *produce* $\beta = P_O(\lambda(u)) \in O^*$ sii existe una configuración finita y estable C de Unf_Σ tal que u es un orden de C.

Dada esta libertad a la hora de disparar eventos concurrentes, debemos contar con una relación que nos permita relacionar configuraciones que respetan el orden de sus eventos, salvo aquellos que son concurrentes.

Definición 3.3.3 Sea (Σ, λ) una BBSPN y $u, v \in E^*$. Decimos que $u \equiv_{I_\Sigma} v$ sii existe una configuración finita C de Unf_Σ tal que tanto u como v son ordenes de C.

Claramente, si Σ acepta $P_I(\lambda(u))$, produce $P_O(\lambda(u))$ y $u \equiv_{I_\Sigma} v$, entonces Σ acepta $P_I(\lambda(v))$ y produce $P_O(\lambda(v))$.

3.4. Estructura de Eventos de Entrada/Salida

Si bien las Redes de Petri son un formalismo que permite modelar e interpretar los sistemas de caja negra de una manera muy simple, pueden presentar ciertas dificultades a la hora de utilizarlas como la entrada de un algoritmo ya que contienen demasiada información para codificar.

En esta sección definiremos un formalismo que tiene el mismo poder expresivo que las redes factibles, pero que permiten una codificación más simple.

Definición 3.4.1 Una *Estructura de Eventos (PES)* es una tupla $(E, \leq, \#)$ donde:

1. E es un conjunto de eventos,
2. $\leq$ es un orden parcial, y

3. # es una relación simétrica e irreflexiva tal que para tres eventos e_1, e_2 y e_3, se cumple:

$$e_1 \leq e_2 \wedge e_1 \# e_3 \Rightarrow e_2 \# e_3.$$

Dada una estructura de eventos $\psi = (E, \leq, \#)$ y un elemento $e \in E$, el *futuro de e* ($\uparrow e$) es el conjunto $\{y \in E \mid e < y\}$.

Suposición 2 *Futuro compartido: para todo par de eventos e_1, e_2 tales que $\uparrow e_1 \neq \emptyset$ y $\uparrow e_2 \neq \emptyset$, tenemos que $\uparrow e_1 \cap \uparrow e_2 \neq \emptyset$. Esto significa que para todo par de eventos, existe un elemento que es mayor a ambos. Esta suposición será muy importante para los resultados del Capítulo 4.*

Definición 3.4.2 Una *Estructura de Eventos de Entrada/Salida (IOPES)* es una PES junto con una función de etiquetado $\overline{\lambda} : E \to (I \cup O)$.

En [4] está demostrado que es posible obtener una PES a partir de una red factible y que existe un isomorfismo entre ambas.

Definición 3.4.3 Sea $N = (B, E, F)$ una red factible, definimos como $\xi(N) = (E, {F^*}_{|E^2}, \#_{|E^2})$ a la estructura que obtenemos al remover los lugares de N.

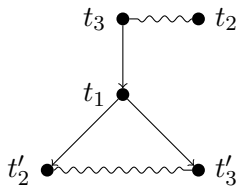

Figura 3.4: Ejemplo de una PES.

La Figura 3.4 muestra la PES $\xi(N)$ siendo N la red de la Figura 2.3.

A partir de los resultados de [4] y [8], podemos aplicar las definiciones del capítulo anterior a las IOPES finitas y utilizarlas para especificar el comportamiento de un BBSPN, aunque esta contenga ciclos.

La definición de un algoritmo para testear una relación entre dos IOPES es el tema del próximo capítulo.

Capítulo 4

Testing para sistemas especificados con BBSPN

En los capítulos anteriores hemos desarrollado toda la teoría necesaria para modelar un Sistema de Caja Negra y su comportamiento. Con estas herramientas ya estamos en condiciones de definir un algoritmo que nos permita detectar fallas en la implementación de un sistema.

4.1. La Implementación de un sistema y sus posibles fallas

A continuación modificaremos el algoritmo propuesto en [3] para detectar las fallas en los sistemas de entrada/salida especificados como BBSPN. Proponemos un algoritmo para testear las siguientes fallas:

- *Pérdida de dependencia de un evento de salida respecto de uno de entrada:* un evento de salida se dispara antes de que todos los eventos de entrada de su pasado hayan sido disparados.
- *Pérdida de dependencia de un evento de entrada respecto de uno de entrada:* podemos disparar un evento de entrada antes de haber disparado todos los eventos de entrada de su pasado.
- *Dependencia adicional de un evento de salida respecto de uno de entrada:* un evento de salida no se dispara en la implementación luego de que todos los eventos de su pasado han sido disparados, pero se dispara luego de que disparamos algún evento de entrada.
- *Dependencia adicional de un evento de entrada respecto de uno de entrada:* la implementación no acepta disparar un conjunto de eventos de entrada en un orden permitido por la especificación.

- *Conflicto adicional:* dos eventos que son concurrentes en la especificación se encuentran en conflicto en la implementación.
- *Pérdida de conflicto:* dos eventos que están en conflicto en la especificación pueden ser disparados en la implementación.

Suposición 3 *Existe una única entrada (llamada reset) que lleva al sistema a su configuración inicial desde cualquier configuración.*

Definición 4.1.1 Decimos que una IOPES $\psi_I = (E_I, \leq_I, \#_I, \overline{\lambda}_I)$ *implementa* una IOPES $\psi_S = (E_S, \leq_S, \#_S, \overline{\lambda}_S)$ sii:

1. los eventos de entrada y salida son los mismos, es decir, $E_I = E_S$ y $\overline{\lambda}_I = \overline{\lambda}_S$,
2. se satisfacen las siguientes condiciones sobre las relaciones entre los eventos. Sean e_1, e_2 dos eventos cualesquiera, i, i' dos eventos de salida y o, o' dos eventos de entrada:
   a) $i <_S e_1$ implica $i <_I e_1$
   b) $o <_S i$ implica $o <_I i$ o $o \parallel_I i$,
   c) $i \parallel_S i'$ implica $i \parallel_I i'$,
   d) $o \parallel_S i$ implica $o \parallel_I i$ o $o <_I i$, y
   e) $e_1 \#_S e_2$ sii $e_1 \#_I e_2$.

Estas condiciones tienen los siguientes significados: la condición $(2a)$ nos asegura que todas las dependencias respecto de un evento de entrada se preservarán. Si no es así, hablamos de una **pérdida de dependencia de un evento de entrada respecto de uno de entrada** o **pérdida de dependencia de un evento de salida respecto de uno de entrada** dependiendo de si e_1 es un evento de entrada o salida. La condición $(2b)$ implica que algunas dependencias de eventos de entrada con respecto a eventos de salida pueden ser removidas.

Las Condiciones $(2c)$ y $(2d)$ nos aseguran que la implementación no introduce **dependencias adicionales respecto a un evento de entrada**. Consideramos que el orden entre eventos de salida que observamos luego de que se dispara un evento de entrada no es importante y por lo tanto la implementación puede introducir una relación de orden entre eventos de salida o incluso invertirlo cuando ambos son inmediatamente precedidos por el mismo evento de entrada, pero en el caso en el cual exista un evento de entrada entre ambos, el orden debe ser preservado. Supongamos $o <_S i <_S o'$. La condición $(2a)$ nos asegura que $i <_I o'$. Si $o' <_I o$, entonces $i <_I o$ lo cual no es posible ya que no se permite que la implementación introduzca dependencias respecto de un evento de entrada.

La condición $(2e)$ implica que la implementación no introducirá o removerá ninguna relación de conflicto. Si no es así, hablamos de **pérdida de conflicto** o **conflicto adicional**.

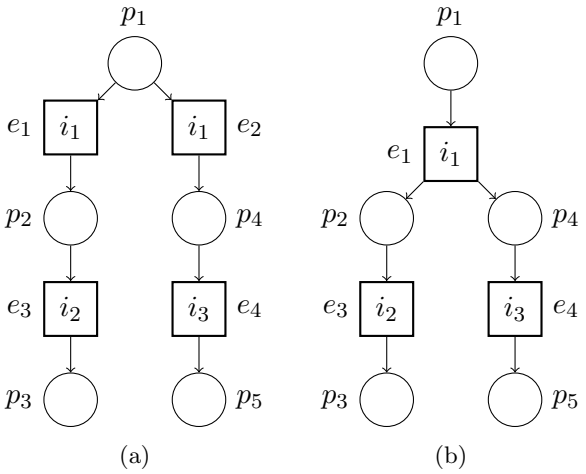

Figura 4.1: Error de especificación.

Aclaración: Si un sistema es especificado según muestra la Figura 4.1 (a), consideramos que el sistema está mal especificado. Si se desea especificar dicho comportamiento, debe ser realizado como muestra la Figura 4.1 (b).

4.2. Un algoritmo para detección de fallas

Ya hemos visto cómo podemos describir el comportamiento de un BBSPN con su unfolding y cómo un prefijo completo finito nos da la misma cantidad de información. También hemos visto que existe un isomorfismo entre las redes factibles y las PES. Con todo esto podemos describir el comportamiento de un BBSPN través de una IOPES.

En esta sección proponemos una manera de testear la relación de implementación entre dos IOPES. De esta forma podemos testear si una implementación y su especificación, dada como un BBSPN, pertenecen a la relación definida anteriormente como muestra la Figura 4.2

Sean $\psi_S = (E, \leq_S, \#_S, \overline{\lambda})$ y $\psi_I = (E, \leq_I, \#_I, \overline{\lambda})$ dos IOPES. Nuestro objetivo es verificar que ψ_I implementa ψ_S. Por lo tanto, debemos testear que la implementación tiene los mismos eventos de entrada y salida que la especificación y que las relaciones de dependencia y conflicto entre eventos es compatible con las definición de 4.1.1.

Dado que queremos testear el comportamiento dinámico de la implementación, debemos utilizar el unfolding del sistema. Esto puede generar un problema si el sistema contiene ciclos, ya que en este caso el unfolding

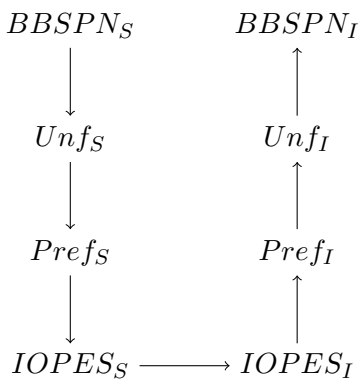

Figura 4.2: Proceso para testear que $BBSPN_I$ implementa a $BBSPN_S$.

es infinito. Es por esto que utilizamos un prefijo complejo finito del mismo para testear la relación de implementación.

Hacemos las siguientes suposiciones sobre el sistema:

Suposición 4 *Podemos detectar cuando la implementación recibe un dato de entrada no identificado. En ese caso, la implementación puede, por ejemplo, devolver algún dato de salida especial considerado como un mensaje de error.*

Suposición 5 *Tiempo de respuesta acotado: cuando todas las precondiciones de un evento de salida han sido disparadas, la implementación disparará el evento de salida en un tiempo acotado. Por lo tanto, si luego de transcurrido ese tiempo el evento de salida no se ha disparado, se puede concluir que no se disparará sin que disparemos otro evento de entrada.*

Definición 4.2.1 Sea E un conjunto de eventos, decimos que $<_{t.o}$ es un *orden de testeo* sobre E sii $e < e' \Rightarrow e <_{t.o} e'$.

Sean $\psi_S = (E, \leq_S, \#_S, \overline{\lambda})$ y $\psi_I = (E, \leq_I, \#_I, \overline{\lambda})$ dos IOPES y $<_{t.o}$ un orden de testo sobre E, utilizamos el Algoritmo 1 para testear que ψ_I implementa ψ_S.

El Algoritmo 1 nos dice en qué forma debemos introducir los datos de entrada y cuáles datos de salida debemos observar (o cuáles no).

Para una relación $e_1 < e_2$, disparamos todos los eventos del pasado de e_2, menos e_1. Si e_2 está habilitado, la relación no fue implementada, sino, disparamos e_1. Si e_2 no se puede disparar después de la cota de tiempo que hemos dispuesto, necesitamos que algún otro evento se dispare.

De manera similar, para cada conflicto especificado en S, intentamos disparar ambos eventos, si el sistema nos lo permite, dicho conflicto no fue implementado. Dado que todos los eventos comparten al menos un evento

futuro y los eventos de entrada son disparados en todos los ordenes posibles en las diferentes iteraciones (respetando $<$), la implementación no puede agregar un conflicto sin que lo detectemos.

Algoritmo 1: Testeando la relación de Implementación

```
1  TE = E;
2  while TE ≠ ∅ do
3      Elegir el menor e ∈ TE respecto de <_{t.o};
4      P_e = IIP_e;
5      while P_e ≠ ∅ do
6          Elegir el menor e' ∈ P_e respecto de <_{t.o};
7          Ingresar reset;
8          Ingresar todos los eventos de entrada de [e]\{e'} respetando
           <_{t.o};
9          if e ∈ IE then
10             Observar S1_{e'e};
11             Ingresar reset;
12         else
13             Observar S2_{e'e};
14             Ingresar e' y observar S3_{e'e};
15         P_e = P_e\{e'};
16     Ingresar todos los eventos de entrada de [e] respetando <_{t.o} y
       observar S4_e;
17     DC_e = {e' ∈ E | e #_μ e'};
18     while DC_e ≠ ∅ do
19         Elegir el menor e' ∈ DC_e respecto de <_{t.o};
20         Ingresar reset;
21         Ingresar todos los eventos de entrada de [e] ∪ [e'] respetando
           <_{t.o} y observar S5_{e'e};
22         DC_e = DC_e\{e'};
23     TE = TE\{e};
```

Proposición 4.2.2 *Sean e_1 y e_2 dos eventos de entrada y e_3 un evento de salida, ψ_I es una implementación de ψ_S sii para toda dependencia directa $e_2 <_S e_1$ y $e_1 <_S e_3$, hay un mensaje de error en $S1_{e_2e_1}$, pero no en $S4_{e_1}$, $\overline{\lambda}(e_3) \notin S2_{e_2e_1}$, pero $\overline{\lambda}(e_3) \in S3_{e_2e_1}$ y para todo conflicto $e_1 \#_S e_2$, hay un mensaje de error en $S5_{e_2e_1}$.*

Dem) Debemos probar que la implementación no introduce ni remueve dependencias respecto de un evento de entrada o conflictos. Por la Proposición 3.2.3, sólo debemos testear los conflictos entre eventos de entrada.

Si la implementación remueve la dependencia $e_2 <_S e_1$, cuando disparamos todos los eventos de entrada de $[e_1]\backslash\{e_2\}$ en la linea 8 del Algoritmo 1, no observamos ningún mensaje de error en $S1_{e_2e_1}$ y no tenemos los valores predictos por la especificación. Si la implementación agrega la dependencia $e_4 <_I e_1$, cuando disparamos $[e_1]$ (que no contiene e_4) en la linea 16, observamos un mensaje de error en $S4_{e_1}$ ya que e_4 está en el pasado de e_1 en ψ_I.

Si ψ_I remueve la dependencia $e_1 <_S e_3$, al disparar los eventos de entrada de $[e_3]\backslash\{e_1\}$ observamos $\overline{\lambda}(e_3)$ en $S2_{e_1e_3}$, lo cual no fue predicto por la especificación. Si la implementación agrega la dependencia $e_1 <_I e_3$, al disparar e_1 en la linea 14 (luego de haber disparado los eventos de entrada de $[e_3]\backslash\{e_1\}$), no observamos $\overline{\lambda}(e_3)$ en $S3_{e_1e_3}$.

Si hay una pérdida de conflicto $e_1 \#_S e_2$ en la implementación, no observamos un mensaje de error en $S5_{e_2e_1}$ al disparar $[e_1] \cup [e_2]$.

Supongamos que la implementación agrega un conflicto. Sean e_1, e_2, x tres eventos tal que $e_1 \leq_I x, e_2 \leq_I x$ y $e_1 \#_I e_2$ (dicho x siempre existe por la Suposición 2). Por 3 en la Definición 3.4.1, $x \#_I e_2$ y nuevamente, por la Definición 3.4.1 y la hipótesis, $x \#_I x$, lo cual es una contradicción ya que ψ_I es una red factible. ∎

El resultado anterior prueba la correctitud del algoritmo, La terminación del mismo viene asegurada por la finitud de la cantidad de relaciones entre eventos en $Pref_\Sigma$.

Podemos diagnosticar las siguiente fallas dependiendo de las salidas observadas:

- *Pérdida de dependencia de un evento de entrada respecto de uno de entrada:* no hay un mensaje de error en $S1_{e'e}$. La dependencia $e' <_S e$ no está implementada.
- *Pérdida de dependencia de un evento de salida respecto de uno de entrada:* observamos $\overline{\lambda}(e)$ en $S2_{e'e}$. En este caso, e es disparado por ψ_I antes de que disparemos e'.
- *Dependencia adicional de un evento de salida respecto de uno de entrada:* no observamos $\overline{\lambda}(e)$ en $S3_{e'e}$. En este caso, e no se dispara en ψ_I hasta que disparamos algún evento e'' que no está especificado en ψ_S.
- *Dependencia adicional de un evento de entrada respecto de uno de entrada:* hay un mensaje de error en $S4_e$. En este caso, e necesita que disparemos un evento e'' en ψ_I mientras que dicho evento no está especificado en ψ_S.

- *Pérdida de conflicto:* no hay un mensaje de error en $S5_{e'e}$. Al disparar un evento que suponemos está en conflicto con e, no obtenemos el error esperado ya que no lo está.

Si bien no se puede hacer un diagnostico como los anteriores para el caso en el cual la implementación agrega un conflicto, dicha falla es testeada por el algoritmo. Todo par de eventos concurrentes tienen un evento en común en su futuro, por lo tanto para testear las dependencias de dicho evento se deben disparar todos los eventos de su pasado y esto producirá un error si la implementación agrega un conflicto. En la sección 4.3 se puede encontrar un ejemplo que explica dicho caso.

La implementación de un sistema debe aceptar cualquier símbolo de entrada compatible con la especificación (y podría aceptar más) y debe producir los símbolos de salida descriptos en la especificación y en un orden compatible con el orden descripto en la misma.

Teorema 4.2.3 *Dadas dos IOPES* $\psi_S = (E, \leq_S, \#_S, \overline{\lambda})$ *y* $\psi_I = (E, \leq_I, \#_I, \overline{\lambda})$*, y una configuración estable* C *de* ψ_S*, si* ψ_I *implementa* ψ_S*, entonces* C *es una configuración de* ψ_I*.*

Dem) Necesitamos probar que C es cerrada bajo $<_I$ y está libre de conflictos respecto de $\#_I$.

Sea $x \in C$ y $y <_I x$. Las condiciones $(2c)$ y $(2d)$ en la Definición 4.1.1 implican que la implementación sólo puede introducir dependencias adicionales de eventos de entrada respecto de eventos de salida o de eventos de salida respecto de eventos de salida si no existen eventos de entrada entre ambos. Por lo tanto $y <_S x$ o $\overline{\lambda}(y) \in O$. Si $y <_S x$, entonces $y \in C$ ya que C es cerrado respecto de $<_S$. Si $\overline{\lambda}(y) \in O$, existe $w \in E$ tal que $w <_I y$ y $\overline{\lambda}(w) \in I$ ya que cada evento de salida debe tener al menos un evento de entrada en su pasado. Por la transitividad de $<_I$, sabemos que $w <_I x$ (elegiremos el mayor evento w con respecto a $<_I$). Las condiciones $(2c)$ y $(2d)$ implican que $w <_S y$ y $w <_S x$. Como C es cerrado respecto de $<_S$, $w \in C$ y como C es persistente, $y \in C$.

Sean $x, y \in C$. Supongamos $x \#_I y$, la condición $(2e)$ implica que $x \#_S y$ lo cual es una contradicción ya que C está libre de conflicto respecto a $\#_S$. ∎

Con este resultado, es fácil ver que una implementación acepta y produce todas las secuencias aceptadas y producidas por su especificación.

4.3. Buscando fallas en la implementación

En esta sección veremos algunos ejemplos de especificaciones y sus posibles implementaciones. Trataremos de ver qué posibles fallas podrían ocurrir y como son detectadas por nuestro algoritmo.

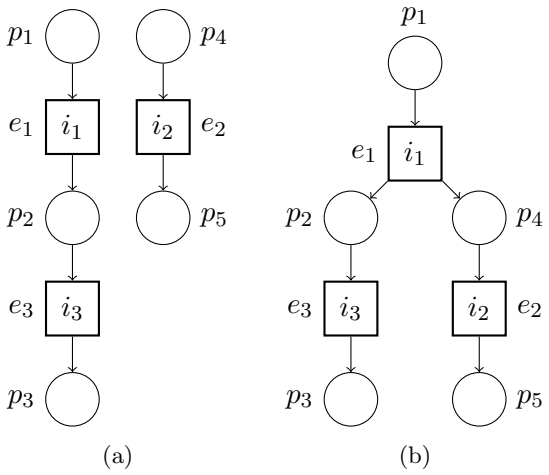

Figura 4.3: Dependencias entre eventos de entrada.

Ejemplo 4.3.1 *Si consideramos a la red de la Figura 4.3 (a) como la especificación del sistema y a (b) su implementación, nos encontramos ante un caso de dependencia adicional del evento de entrada e_2 respecto del de entrada e_1. Al ingresar los datos de entrada de los eventos de $[e_2]\backslash\{e_1\}$, se genera un mensaje de error (observado en $S1_{e_1e_2}$) ya que en la implementación debemos disparar e_1 antes de poder disparar e_2 . Por lo tanto, nuestro algoritmo detecta dicha falla.*

Ejemplo 4.3.2 *Si consideramos a la red de la Figura 4.3 (b) como la especificación del sistema y a la Figura 4.3 (a) como su implementación, la dependencia del evento de entrada e_2 respecto de e_1 no está implementada. Detectamos esta falla con nuestro algoritmo ya que luego de ingresar los datos de entrada de los eventos de $[e_2]\backslash\{e_1\}$, no vemos un mensaje de error en $S1_{e_1e_2}$.*

Ejemplo 4.3.3 *Las Figuras 4.4 (a) y (b) muestran la especificación de un sistema y una implementación del mismo en la cual se agrega la dependencia del evento de salida e_3 respecto del de entrada e_2. El Algoritmo 1 detecta esta falla ya que no observamos la salida o en $S3_{e_2e_3}$ luego de haber disparado todos eventos de entrada de su pasado según la especificación, en este caso, e_1.*

Ejemplo 4.3.4 *Sea la red de la Figura 4.4 (b) la implementación del sistema especificado en la Figura 4.4 (a). En esta implementación hay pérdida de la dependencia del evento de salida e_3 respecto del de entrada e_2. Dicha falla es detectada por nuestro algoritmo ya que observamos la salida o en $S2_{e_2e_3}$ luego de haber disparado e_1, pero no e_2.*

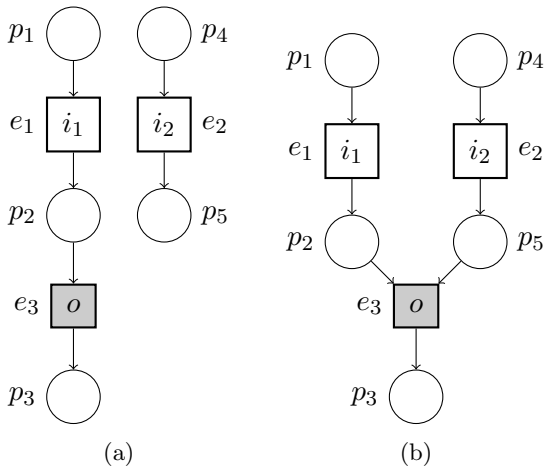

Figura 4.4: Dependencias entre eventos de entrada.

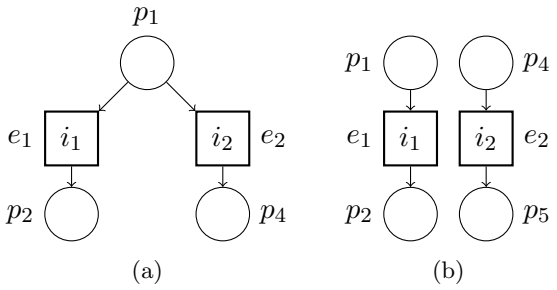

Figura 4.5: Conflicto entre eventos de entrada.

Ejemplo 4.3.5 *La Figura 4.5 muestra (a) una especificación y (b) su implementación, la cual elimina el conflicto entre* e_1 *y* e_2*. Esta falla es detectada por el Algoritmo 1 en la linea 21 ya que no observamos un mensaje de error en* $S5_{e_1e_2}$ *al intentar ingresar* i_1i_2 *(si* $e_1 <_{t.o} e_2$*) o* i_2i_1 *(si* $e_2 <_{t.o} e_1$*).*

Veamos un ejemplo que demuestra la importancia de la Suposición 2.

Ejemplo 4.3.6 *La Figura 4.6 muestra (a) la especificación de un sistema y (b) una posible implementación donde se agrega el conflicto* $e_1 \# e_2$*. Dicho conflicto adicional no es detectado por el Algoritmo 1 ya que la Suposición 2 no se cumple. La Figura 4.6 muestra (c) la especificación de un sistema similar y (d) una posible implementación del mismo con la misma pérdida de conflicto, pero en este caso el Algoritmo 1 detecta dicha falla ya que en la iteración que se testean las dependencias del evento* e_4*, se disparan todos los eventos de su pasado según la especificación y esto genera un error en la implementación ya que no podemos disparar* e_1 *y* e_2*. Las especificaciones que no cuentan con tal evento, pueden ser descompuestas en sub-especificaciones*

que si cumpla esta suposición. Otra alternativa es agregar un evento dummy que sea mayor a todos según $<$ con el propósito de testear el sistema.

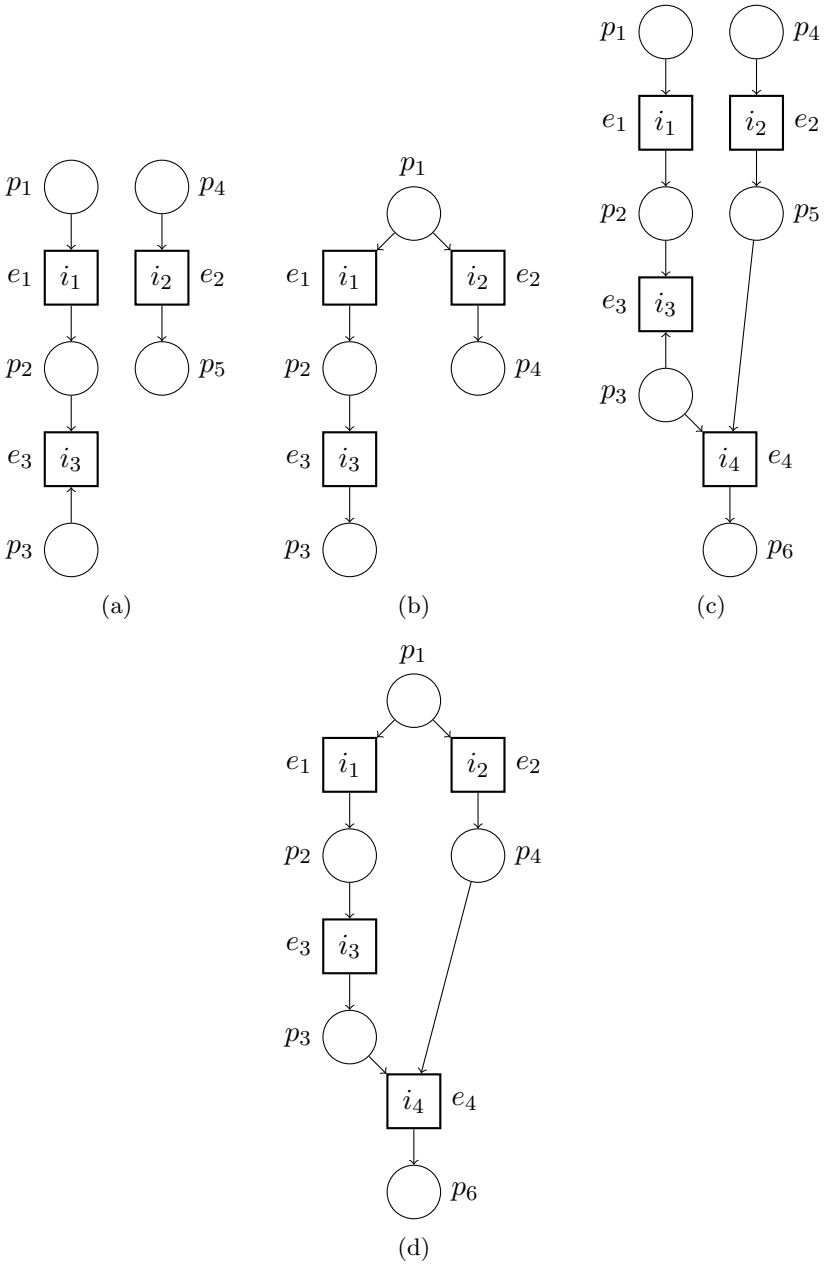

Figura 4.6: Futuro compartido.

Capítulo 5

Conclusiones y trabajos futuros

El estudio de las Redes de Petri y la semántica de su comportamiento dinámico nos llevó en este trabajo a la definición de las Estructuras de Eventos de Entrada/Salida como modelo para el testing de Sistemas implementados como Caja Negra. Es el primer modelo distribuido que se utiliza para este tipo de testing y elimina toda las limitaciones de los modelos secuenciales.

Con este nuevo modelo hemos podido definir una relación de correcta implementación entre una especificación y su correspondiente implementación y proponer un algoritmo que nos permite testear dicha relación. Probamos que el algoritmo es correcto y termina.

Gracias a los resultados de este trabajo y los de [4, 7, 8] hemos podido definir los pasos a seguir para testear una implementación a partir de su especificación dada como un BBSPN. Dada la existencia de diversos softwares que automatizan la creación del unfolding de una Red de Petri y un prefijo completo finito de la misma, es posible la integración de los mismos junto como el algoritmo propuesto para automatizar todo el proceso.

Este trabajo abre la puerta al completo estudio de testing utilizando Redes de Petri. Este estudio puede basarse en los conceptos e ideas ya definidas para las máquinas de estado finito, tales como secuencias de distinción, adaptativas, de separación, etc., e intentar adaptarlos, en la medida que sea posible, al nuevo formalismo. Estas secuencias permiten, en el modelo secuencial, la definición de distintos algoritmos. Resultados similares se esperan para los sistemas modelados bajo este nuevo formalismo.

La definición que hemos dado de completitud para un prefijo no es la más general. En [8] se generaliza esta definición. Dado que nuestro algoritmo depende de esta definición de completitud, se espera que al generalizarla el algoritmo permita testear otro tipo de relaciones.

Una vez definidos estos nuevos métodos y algoritmos, se podrá estudiar

su utilización para otras ramas de estudio tales como diagnosticabilidad o satisfactoriedad.

Existen sistemas donde el no determinismo no genera problemas. Para dichos sistemas, sería interesante ver qué implicaciones tendría no pedir algunas de las propiedades descriptas en el Capítulo 2. La relajación de ciertas suposiciones es otra posible extensión de este trabajo.

Bibliografía

[1] D. Lee, M. Yannakakis: Principles and methods of testing finite state machines – a survey. Proceedings of the IEEE 84(8), 1089–1123 (1996).

[2] S. Haar, C. Jard, GV. Jourdan: Testing Input/Output Partial Order Automata. TestCom/FATES 2007: 171-185.

[3] GV. Bochmann, S. Haar, C. Jard, GV. Jourdan: Testing Systems Specified as Partial Order Input/Output Automata. TestCom/FATES 2008: 169-183.

[4] M. Nielsen, G. Plotkin, G. Winskel: Petri Nets, Event Structures and Domains. Semantics of Concurrent Computation 1979: 266-284.

[5] U. Goltz, W. Reisig: The Non-sequential Behavior of Petri Nets Information and Control 57(2/3): 125-147 (1983).

[6] J. Engelfriet: Branching Processes of Petri Nets. Acta Inf. 28(6): 575-591 (1991).

[7] KL. McMillan: A Technique of State Space Search Based on Unfolding. Formal Methods in System Design 6(1): 45-65 (1995).

[8] J. Esparza, S. Römer, W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Formal Methods in System Design 20(3): 285-310 (2002).

[9] V. Khomenko, M. Koutny, W. Vogler: Canonical prefixes of Petri net unfoldings. Acta Inf. 40(2): 95-118 (2003).

[10] G. Luo, R. Dssouli, GV. Bochmann, P. Ventakaram, A. Ghedamsi: Generating synchronizable test sequences based on finite state machines with distributed ports. Proceeding of the IFIP Sixth International Workshop on Protocol Test Systems, Pau, France, September 1993, pp. 53-68 (1993).