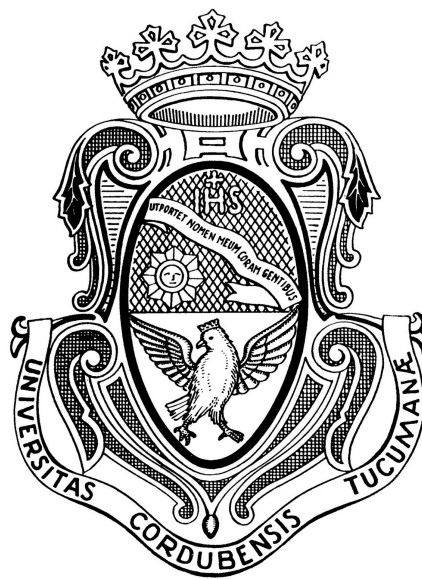


TESIS DE GRADO
INGENIERÍA EN COMPUTACIÓN

JOAQUÍN RODRÍGUEZ FELICI
LEANDRO JOSÉ ASSÓN



EDITOR, SIMULADOR
Y ANALIZADOR DE REDES DE PETRI

Director de tesis: Ing. Orlando Micolini
Laboratorio de Arquitectura de Computadoras
Universidad Nacional de Córdoba

Agosto 2017

All we have to decide is what to
do with the *time* that is given to us.

— J.R.R. Tolkien

RESUMEN

El contenido de este documento abarca las motivaciones, los objetivos, los detalles de implementación y demás datos pertinentes del desarrollo de una tesis de grado para la carrera de *Ingeniería en Computación*. El mismo se estructura en tres grandes partes: introducción del proyecto, desarrollo de las funcionalidades y conclusiones obtenidas.

El tema desarrollado es principalmente el análisis y la simulación de *redes de Petri*, pretendiendo documentar algunos algoritmos ya existentes así como integrar e implementar una herramienta modular y formal para el desarrollo de sistemas embebidos.

AGRADECIMIENTOS

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetur.

ÍNDICE

I	INTRODUCCIÓN	1
1	INTRODUCCIÓN	3
1.1	Motivaciones	3
1.2	Descripción general	3
1.3	Objetivos	4
1.4	Requerimientos	4
1.5	Modelo de desarrollo	6
1.6	Análisis de riesgos	9
1.6.1	Identificación de los riesgos	11
1.6.2	Análisis de los riesgos	11
1.6.3	Selección de los riesgos	12
1.6.4	Elaboración de planes de riesgos	13
2	MARCO TEÓRICO	15
2.1	Redes de Petri	15
2.1.1	Estructura de una red de Petri ordinaria	16
2.1.2	Matriz de incidencia	17
2.2	Dinámica de una red de Petri	18
2.2.1	Sensibilizado de una transición	18
2.2.2	Disparo de una transición	19
2.2.3	Función de transferencia y ecuación de estado	20
2.2.4	Extensión de la ecuación de estado	21
2.3	Propiedades de las redes de Petri	22
2.3.1	Propiedades de limitación	22
2.3.2	Propiedades de vivacidad	22
2.3.3	Alcanzabilidad de una red de Petri	23
2.3.4	Cobertura de una red de Petri	24
2.3.5	Sifones y trampas	25
2.3.6	Invariantes de plazas y transiciones	27
2.3.7	Propiedades de concurrencia	29
2.4	Monitores	30
2.5	Redes SPN y GSPN	33
II	DESARROLLO	37
3	INTRODUCCIÓN	39
4	IT. 1: IMPORTACIÓN Y EXPORTACIÓN DE ARCHIVOS	43
4.1	Introducción	43
4.2	Objetivos	43
4.3	Desarrollo	43
4.4	Testing	48
4.5	Conclusiones	48
5	IT. 2: SIMULACIÓN DE REDES DE PETRI	49
5.1	Introducción	49
5.2	Objetivos	49
5.3	Desarrollo	50
5.4	Testing	57

5.5	Conclusiones	59
6	IT. 3: CLASIFICACIÓN DE REDES DE PETRI	61
6.1	Introducción	61
6.2	Objetivos	61
6.3	Desarrollo	61
6.3.1	Máquina de estados	63
6.3.2	Grafo marcado	64
6.3.3	Libre elección	65
6.3.4	Libre elección extendida	67
6.3.5	Red simple	68
6.3.6	Red simple extendida	69
6.3.7	Safeness	69
6.3.8	Boundedness e interbloqueo	70
6.3.9	Obtención de matrices	71
6.4	Testing	73
6.5	Conclusiones	74
7	IT. 4: CÁLCULO DE SIFONES, TRAMPAS E INVARIANTES	75
7.1	Introducción	75
7.2	Objetivos	75
7.3	Desarrollo	75
7.3.1	Invariantes de plazas	75
7.3.2	Invariantes de Transición	78
7.3.3	Sifones y trampas	79
7.4	Testing	81
7.5	Conclusiones	81
8	IT. 5: ANÁLISIS DE REDES ESTOCÁSTICAS	83
8.1	Introducción	83
8.2	Objetivos	83
8.3	Desarrollo	83
8.3.1	Estados tangibles	84
8.3.2	Distribución de los estados tangibles	84
8.3.3	Densidad de probabilidad de tokens	87
8.3.4	Promedio de tokens por plaza	88
8.3.5	Productividad de las transiciones	88
8.3.6	Tiempo de permanencia para estados tangibles	89
8.4	Testing	90
8.5	Conclusiones	90
9	IT. 6: ALGORITMOS DE ALCANZABILIDAD Y COBERTURA	93
9.1	Introducción	93
9.2	Objetivos	93
9.3	Desarrollo	93
9.4	Testing	94
9.5	Conclusiones	94
10	IT. 7: DISPARO DE TRANSICIONES ESTOCÁSTICAS	97
10.1	Introducción	97
10.2	Objetivos	97
10.3	Desarrollo	98
10.3.1	Exportación de archivos	98
10.3.2	Disparo de transiciones estocásticas	98

10.3.3	Funciones de distribución de probabilidad . . .	101
10.4	Conclusiones	102
11	IT. 8: SIMULACIÓN DE REDES GSPN	103
11.1	Introducción	103
11.2	Objetivos	103
11.3	Desarrollo	104
11.4	Testing	106
11.5	Conclusiones	107
III	CONCLUSIONES	109
12	CONCLUSIONES	111
13	TRABAJOS FUTUROS	113
	BIBLIOGRAFÍA	115

INDICE DE FIGURAS

Fig. 1	Diagrama de componentes simplificado	4
Fig. 2	Modelo iterativo e incremental	7
Fig. 3	Red de Petri simple	16
Fig. 4	Transiciones sensibilizadas en red de Petri simple	19
Fig. 5	Disparo de una red de Petri simple	20
Fig. 6	Red de Petri de tres estados	24
Fig. 7	Grafo de alcanzabilidad	24
Fig. 8	Red de Petri no acotada	25
Fig. 9	Grafo de cobertura	25
Fig. 10	Red de Petri con trampa y sifón	26
Fig. 11	Red de Petri con invariantes de plazas	27
Fig. 12	Bifurcación	29
Fig. 13	Sincronización	29
Fig. 14	Red de Petri con propiedades <i>cobegin/coend</i> . .	30
Fig. 15	Red de Petri con exclusión mutua	30
Fig. 16	Diagrama de un monitor	33
Fig. 17	Red de Petri estocástica	34
Fig. 18	Red de Petri estocástica generalizada	35
Fig. 19	Diagrama de clases típico de un <i>command pattern</i>	40
Fig. 20	Diagrama de clases PNEditor	42
Fig. 21	Diagrama de secuencia de la obtención de datos	46
Fig. 22	Diagrama de secuencia de la clase <i>SimulateAction</i>	50
Fig. 23	Diagrama de la función <i>fireGraphically()</i>	55
Fig. 24	Sim: red a simular	57
Fig. 25	Sim: botones	58
Fig. 26	Sim: ingreso de parámetros	58
Fig. 27	Sim: resultado de simulación	59
Fig. 28	Máquina de estados: ejemplos	63
Fig. 29	Diagrama de secuencia de la función <i>stateMachine()</i>	64
Fig. 30	Grafos marcados: ejemplos	65
Fig. 31	Diagrama de secuencia de la función <i>markedGraph()</i>	66
Fig. 32	Redes de libre elección: ejemplos	66
Fig. 33	Diagrama de la función <i>freeChoiceNet()</i>	67
Fig. 34	Redes simples: ejemplos	68
Fig. 35	Diagrama de secuencia de la función <i>simpleNet()</i>	69
Fig. 36	Diagrama de secuencia de la función <i>isSafe()</i> . .	70
Fig. 37	Diagrama de secuencia de la función <i>analyzeTreeRecursively()</i>	71
Fig. 38	Diagrama de secuencia de la función <i>forwardIncidenceMatrix()</i>	72
Fig. 39	Clasificación de una red	73
Fig. 40	Propiedades de una red	73
Fig. 41	Matriz <i>post</i>	74

Fig. 42	Matriz de incidencia	74
Fig. 43	Red de Petri productor y consumidor	76
Fig. 44	Red de Petri con trampa y sifón	80
Fig. 45	P – invariantes	81
Fig. 46	T – invariantes	81
Fig. 47	Sifones y trampas de una red de Petri simple .	81
Fig. 48	Red de Petri de dos estados	84
Fig. 49	Red de Petri estocástica simple	85
Fig. 50	Grafo de alcanzabilidad de una red de Petri es- tocástica	85
Fig. 51	Diagrama de secuencia de la función <i>averageTo- kens()</i>	88
Fig. 52	Análisis GSPN: ejemplos	90
Fig. 53	Grafo de alcanzabilidad	94
Fig. 54	Grafo de cobertura	94
Fig. 55	Diagrama de secuencia de la función <i>internal- FireTransition()</i>	99
Fig. 56	Distribución normal	102
Fig. 57	Distribución uniforme	102
Fig. 58	Diagrama de secuencia de la función <i>internal- FireTransition()</i> para redes estocásticas	105
Fig. 59	Sim: red de Petri estocástica	106
Fig. 60	Sim: configuración de una transición	107
Fig. 61	Sim: historia de una única plaza	107
Fig. 62	Sim: historia de múltiples plazas	108

INDICE DE TABLAS

Tab. 1	Identificación de riesgos de proyecto	10
Tab. 2	Identificación de riesgos técnicos	10
Tab. 3	Identificación de riesgos de negocio	11
Tab. 4	Análisis de los riesgos	12
Tab. 5	Riesgos a tratar	13

Parte I

INTRODUCCIÓN

En esta sección se tratarán temas como el propósito, los objetivos y la motivación para la realización de este proyecto. De igual manera se especificará el método de trabajo que se utilizó y se explicarán los conceptos básicos relacionados con las redes de Petri; así como sus usos, maneras de implementación y propiedades matemáticas.

INTRODUCCIÓN

1.1 MOTIVACIONES

Las motivaciones para el desarrollo de este proyecto pueden separarse en dos grandes grupos. Por un lado aquellas relacionadas al producto en sí, entre las cuales puede destacarse la necesidad de realizar una herramienta modular y formal para el desarrollo de sistemas embebidos, puesto que en la actualidad no existe un simulador y analizador de *redes de Petri* que posea todas las características que consideramos necesarias para la realización de este tipo de trabajos.

Por otro lado, existen sin duda ciertas motivaciones de naturaleza académica. Entre ellas podemos mencionar la integración de los conocimientos adquiridos a lo largo de nuestros estudios, la contribución a la comunidad *open source* e incluso la puesta en práctica de procedimientos estrictos de investigación, de desarrollo de *software* y de documentación que nos serán sin duda de gran valor durante nuestro futuro ejercicio como profesionales.

1.2 DESCRIPCIÓN GENERAL

El proyecto consiste a grandes rasgos en el diseño e implementación de una herramienta para la edición, la simulación y el análisis de las *redes de Petri*. Existen en la actualidad varias herramientas con funcionalidades similares, aunque ninguna posee todas las deseadas. Por este motivo se buscará, además del diseño, integrar ciertos componentes ya implementados. Estos componentes serán principalmente:

1. Un editor de *redes de Petri*, al cual deberán incorporarse las funcionalidades que se plantearán en los requerimientos más adelante en esta sección. La obtención de este editor será desarrollada en el capítulo 3.
2. Un conjunto de algoritmos que se utilizarán para analizar la *red de Petri*. Entre ellos podemos mencionar uno para la obtención del *grafo de cobertura* de una red, uno para el cálculo de las ecuaciones de *invariantes*, etc.
3. Un monitor de *redes de Petri temporales*, el cual fue diseñado por estudiantes del *Laboratorio de Arquitectura de Computadoras* como proyecto integrador y el cual se utilizará como motor de ejecución del simulador.

Por lo tanto, el producto final será la integración de estos tres grandes componentes, así como la incorporación de un conjunto de funcionalidades deseadas que aún no se encuentran presentes en los

misimos. Esto se ilustra de manera simplificada en la figura 1. Los componentes serán desglosados y explicados en profundidad en el capítulo 3.

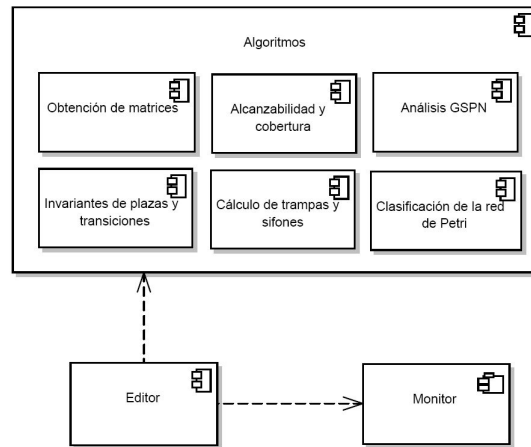


Fig. 1: Diagrama de componentes simplificado

1.3 OBJETIVOS

Como se mencionó en la motivación, el objetivo final es el desarrollo de una herramienta modular y formal para el desarrollo de sistemas embebidos. Para alcanzar esta meta se deben plantear varios objetivos intermedios, algunos de los cuales se mencionaron en la *descripción general* de este capítulo. Nuestros objetivos puntuales serán: implementar un editor de *redes de Petri*, integrar al editor un monitor que funcionará como motor de ejecución de las redes, adaptar (o desarrollar de ser necesario) los algoritmos deseados para el análisis y diseñar una interfaz de simulación que haga uso del monitor anteriormente mencionado.

1.4 REQUERIMIENTOS

En la ingeniería, los requerimientos se utilizan como datos de entrada en la etapa de diseño del producto. Establecen qué debe hacer el sistema, aunque no especifican la manera en que debe hacerlo. La fase de captura y registro de requisitos puede estar precedida por una fase de análisis conceptual del proyecto. Para el caso de éste, la obtención de dichos requerimientos se llevó a cabo en tres fases ejecutadas de manera secuencial:

1. *Entrevista*: Como punto de partida, se efectuó una entrevista con el director de tesis *Orlando Micolini*, en la cual se plantearon las necesidades de crear una herramienta con las funcionalidades mencionadas brevemente en la descripción general de este capítulo. En la misma se expuso la problemática, las posibles soluciones y se documentó una lista tentativa de requerimientos.

2. *Análisis de documentos*: Como segundo paso, se analizó con detalle una base de datos¹ de editores de *redes de Petri* con el objetivo de comprender y documentar las funcionalidades ya existentes en las herramientas presentes, así como buscar componentes que podrían resultar útiles para el sistema deseado (como algún algoritmo ya implementado en una herramienta *open-source*). De esta manera, se clasificaron estas herramientas según los siguientes criterios:
 - Lenguaje en que está escrita.
 - *Open-source* o no.
 - Tipo de licencia.
 - Código disponible.
 - Interfaz de usuario.
 - Posesión de otras funcionalidades deseadas.
3. *Entrevista cerrada*: Se repitió la fase uno, pero esta vez se tuvo en cuenta la información concreta obtenida en la fase 2. Se fusionaron los requerimientos tentativos con aquellos que no se habían considerado pero que resaltaron al analizar la base de datos, y se eliminaron otros cuya probabilidad de implementación disminuyó significativamente luego del análisis en la fase anterior.

Este procedimiento produjo como resultado una lista de requerimientos cuyo cumplimiento repercutirá directamente en la calidad del sistema final. Debe tenerse en cuenta que algunos conceptos técnicos que aún no se trataron se aclararán y/o referenciarán en el siguiente capítulo. Por lo tanto, los **requerimientos** son:

1. El sistema debe permitir crear, abrir y editar una *Red de Petri* en un espacio gráfico denominado *worksheet*.
2. El sistema debe brindar la clasificación de la red presente en el editor. Esto incluye mostrar si la misma es: una maquina de estado, un grafo marcado, acotada, segura, o si presenta interbloqueo.
3. El sistema debe generar y mostrar las matrices asociadas a la red, así como el marcado actual de la misma y las transiciones sensibilizadas.
4. El sistema debe ser capaz de generar las ecuaciones asociadas a las invariantes de plazas y transiciones.
5. El sistema debe calcular y mostrar los sifones y trampas existentes en la red.
6. El sistema debe brindar un análisis para una *general stochastic Petri net*. Esto incluye mostrar: el conjunto de estados tangibles, el promedio de tokens por plazas, la densidad de probabilidad de los tokens y la productividad de las transiciones temporales.

¹ *Petri Nets Tool Database*, [Online]. Disponible: <https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html>

7. El sistema debe ser capaz de generar y mostrar el grafo de alcanzabilidad ó cobertura de la red, dependiendo de si la misma es acotada o no.
8. El sistema debe permitir la exportación e importación de la red al/del formato *.pnml*, tanto para el dialecto del monitor como para el de los algoritmos.
9. El sistema debe incorporar las siguientes propiedades para las transiciones: nombre de la guarda asociada, si la misma es temporal (con su valor de *rate* en caso de serlo), automática e informada.
10. El sistema debe proveer de una interfaz de simulación para redes de Petri no temporales, implementando acciones como la de iniciar y detener la simulación, así como configurar algunos parámetros entre los cuales se encuentran la cantidad de transiciones que se desea disparar y el tiempo que debe transcurrir entre el disparo de las mismas.
11. El sistema debe implementar la misma simulación para redes estocásticas, caso para el cual se deberá poder determinar el *rate* de las transiciones, así como la selección de la función de distribución asociada a las mismas.
12. El sistema debe graficar el comportamiento de las redes estocásticas, por ejemplo mostrando la cantidad de tokens por plaza en cada instante de tiempo.

1.5 MODELO DE DESARROLLO

Para la elaboración del presente proyecto se optó por utilizar un modelo iterativo e incremental. A grandes rasgos, este tipo de modelo de desarrollo no es más que un conjunto de tareas agrupadas en pequeñas etapas repetitivas, las cuales inician con un análisis y finalizan con la instauración y aprobación del sistema.

Se planifica un proyecto en distintos bloques temporales denominados iteraciones. Dentro de una iteración se repite un determinado proceso de trabajo sobre uno o varios objetivos, obteniéndose al final de la misma un resultado con más funcionalidades implementadas que el de la iteración anterior. La ventaja principal de este modelo es que no se debe esperar a que el sistema esté completo para que el mismo sea utilizable y operacional.

El primer incremento cumple con los requerimientos elementales y críticos de forma que, desde un principio, el sistema cuente con las funcionalidades necesarias para su uso. Para lograr esto, al realizar el análisis de una iteración, se especifican los requerimientos y los objetivos que se esperan conseguir al finalizar la misma. Los objetivos se establecen en función de los requerimientos, de los riesgos y de la

evaluación de los resultados de las iteraciones precedentes. A continuación, se realiza el desarrollo e implementación, para luego ponerlo a prueba y evaluarlo al final de la iteración.

Se busca que en cada iteración los componentes logren evolucionar el producto dependiendo de aquellos completados en las iteraciones anteriores. En la *figura 2* se representa el funcionamiento del modelo.



Fig. 2: Modelo iterativo e incremental

Una vez explicado esto, se procede a definir las iteraciones. Cada iteración abarca uno o más requerimientos, los cuales se desglosaron en múltiples tareas:

- **Iteración 1: Exportación e importación de formatos compatibles con el monitor y con los algoritmos.**
 1. Realizar la exportación de la red en formato *.pnml* cuyo dialecto sea compatible con el monitor.
 2. Realizar la exportación de la red en formato *.xml* cuyo dialecto sea compatible con los algoritmos.
 3. Modificar el guardado y lectura del formato *.pflow* para que el mismo incorpore las nuevas características y propiedades añadidas en la red, como la guarda, el *rate*, etc.
 4. Permitir importar un archivo *.xml* con el dialecto utilizado por la herramienta PIPE.
- **Iteración 2: Simulación de una red de Petri no temporal en el monitor y en el editor.**
 1. Desarrollar una ventana que le permita al usuario ingresar la cantidad de transiciones a disparar y el tiempo entre el disparo de cada una de ellas.
 2. Crear los hilos para aquellas transiciones que no sean automáticas, de manera que los mismos intenten disparar una única transición cada una cantidad aleatoria de tiempo.
 3. Implementar un *observer* y asociarlo a las transiciones de manera que el mismo reciba un evento cada vez que una transición sea disparada. Los eventos serán almacenados en una lista ordenada que representará la evolución de la red durante el tiempo de simulación.

4. Realizar la simulación propiamente dicha, proveyendo al monitor con la red y el resto de información necesaria.
5. Generar en el editor la representación de la simulación una vez que se hayan generado en el *observer* la cantidad de eventos especificada por el usuario.

■ **Iteración 3: Clasificación de la red y cálculo de las matrices asociadas.**

1. Clasificar la red presente en el editor, esto incluye indicar si: la red es una maquina de estado, un grafo marcado, de libre elección, acotada, segura, o si la misma presenta interbloqueo.
2. Calcular y mostrar matrices asociadas a la red, entre las cuales se encuentran:
 - Matriz I^+
 - Matriz I^-
 - Matriz de inhibición
 - Matriz de incidencia
 - Transiciones sensibilizadas
 - Marcado actual

■ **Iteración 4: Cálculo de sifones, trampas e invariantes**

1. Cálculo del conjunto de plazas que cumple con las condiciones para ser un sifón.
2. Cálculo del conjunto de plazas que cumple con las condiciones para ser una trampa.
3. Cálculo del conjunto de plazas que forman un P-invariante.
4. Cálculo del conjunto de transiciones que forman un T-invariante.
5. Obtención de las ecuaciones asociadas a los P-invariantes.

■ **Iteración 5: Análisis de una *general stochastic Petri net*.**

1. Obtener y mostrar en pantalla la siguiente información sobre la red presente en el editor:
 - Conjunto de estados tangibles.
 - Promedio de tokens por plazas.
 - Densidad de probabilidad de los tokens.
 - Productividad de las transiciones temporales.
 - Tipo de permanencia en los estados tangibles.
2. Exportar en un archivo *.html* la información obtenida.

■ **Iteración 6: Cálculo de grafo de alcanzabilidad y cobertura.**

1. Calcular y representar gráficamente el grafo de alcanzabilidad de la red en caso de que ésta sea acotada. En caso de que la misma no lo fuera, realizar el mismo procedimiento para el cálculo del grafo de cobertura.
2. Hacer diferencia entre los estados tangibles y efímeros para el caso de *redes de Petri* temporales.

- **Iteración 7: Disparo de transiciones de *general stochastic Petri net*.**
 1. Modificar el editor para que las transiciones temporales posean las propiedades relacionadas con la distribución de probabilidad asociada a las mismas.
 2. Realizar modificaciones en el monitor para que el mismo realice disparos de una *general stochastic Petri net*.
 3. Permitir la selección de una función de densidad de probabilidad para cada transición.
- **Iteración 8: Simulación de una *general stochastic Petri net*.**
 1. Modificar el editor para que el mismo pueda crear los hilos necesarios que intenten disparar transiciones de una *red de Petri* estocástica.
 2. Generar la simulación en el monitor, informando los eventos ocurridos al *observer*. En este caso particular, el evento también incluirá el tiempo de disparo de la transición en cuestión.
 3. Generar la simulación en base a la lista de eventos almacenada en el *observer*.
 4. Para cada plaza, generar un gráfico de dos dimensiones que represente el comportamiento de la misma durante la simulación. Este comportamiento hace referencia a la cantidad de *tokens* que contuvo para cada instante de tiempo.

1.6 ANÁLISIS DE RIESGOS

Un riesgo es un evento o condición incierta que, en caso de ocurrir, tendrá consecuencias sobre al menos uno de los requerimientos del proyecto. Se pueden identificar varios tipos de riesgos en un sistema como el que se intenta desarrollar, entre los cuales podemos destacar:

- Riesgos de proyecto
- Riesgos técnicos
- Riesgos de negocio

Los primeros amenazan la planificación de proyecto, principalmente los aspectos relacionados al tiempo, tanto para cada iteración como la duración global del mismo. Los riesgos técnicos repercuten directamente en la calidad, mientras que aquellos de negocio amenazan la viabilidad del proyecto. Debido a las posibles consecuencias que los riesgos pueden ocasionar, es generalmente recomendable realizar un *plan de gestión riesgos* luego de definir los requerimientos y antes de comenzar con el desarrollo.

La *gestión de riesgos* es un enfoque estructurado para manejar la incertidumbre ante amenazas a través de una secuencia de actividades que tienen como objetivo reducir los efectos negativos de los riesgos,

Riesgo	Descripción
Problemas de coordinación entre los miembros del equipo	Los problemas internos dentro del equipo causan complicaciones en el desarrollo del proyecto, pudiendo provocar la desintegración del grupo de trabajo.
Subestimación de tamaño	Que el producto a realizar sea más grande y complejo de lo que se creyó en un principio.
Subestimación de los tiempos	Los tiempos necesarios para la finalización del proyecto pueden ser mayores de los estimados al inicio del mismo.
Aumento de la complejidad del proyecto	El producto se va haciendo cada vez más complejo, implicando tiempo y esfuerzos que no se tenían planeados.
Metodologías inadecuadas en el desarrollo del proyecto	La metodología de desarrollo seleccionada es contraproducente o no es la adecuada para este tipo de proyecto.
Planificaciones demasiado optimistas	Se planifica el proyecto en base a criterios de situaciones óptimas o ideales en las que no se presentan problemáticas.
No encontrar un editor con las características deseadas	Problemas para encontrar un <i>software</i> de código abierto que contenga una interfaz gráfica adecuada y una estructura proyecto organizada.

Tabla 1: Identificación de riesgos de proyecto

Riesgo	Descripción
Pérdida del trabajo realizado	Se pierde el trabajo ya realizado, ya sea por extravío del dispositivo que lo contenga o por problemas con la nube.
Enfoque incorrecto de las necesidades cubiertas	El producto desarrollado abarca funcionalidades que no satisfacen las necesidades.
Priorizar la programación	Se centra el esfuerzo del equipo en la programación, dejando de lado otras actividades fundamentales dentro del proyecto.
Incompatibilidad de comunicación entre el editor y algún algoritmo determinado	Debido a que los algoritmos son reutilizados de otro <i>software</i> , existe la posibilidad que alguno de ellos sea incompatible, teniendo que ser desarrollado en su totalidad.
No encontrar algún algoritmo determinado	Existe la posibilidad de que algún algoritmo deseado para incluir en el proyecto no este disponible dentro de los componentes de <i>software</i> que se tiene acceso, debiendo desarrollarlo en su totalidad.

Tabla 2: Identificación de riesgos técnicos

Riesgo	Descripción
Falta de aceptación en el mercado	El mercado abarcado considera que no necesita, o que no sirve el producto realizado.

Tabla 3: Identificación de riesgos de negocio

evadirlos de ser posible, o aceptar las consecuencias en el peor de los casos. Las actividades principales que forman parte de este plan son:

1. Identificación de los riesgos.
2. Análisis de los riesgos.
3. Selección de los riesgos.
4. Elaboración de los planes de mitigación y respuesta a riesgos.

1.6.1 IDENTIFICACIÓN DE LOS RIESGOS

La identificación de los riesgos se realizará mediante las dos estrategias más populares existentes: *brainstorming*² y *checklist*³, en el orden mencionado, siendo esto representado en las tablas 1, 2 y 3, de acuerdo al tipo de riesgo que se trate. En cada una de ellas se especificarán el nombre del riesgo y una descripción general del mismo.

1.6.2 ANÁLISIS DE LOS RIESGOS

Ya que no se pueden tratar todos los riesgos existentes, en esta etapa se realizará la cuantificación de los mismos para compararlos y elegir aquellos que se consideren más relevantes para su tratamiento. La posibilidad de ocurrencia se medirá en una escala de 0 a 1, donde 0 significa que es prácticamente imposible que ocurra y 1 que es muy posible. Por otro lado, el impacto se medirá en una escala de 1 a 4, cuyos niveles representan:

1. Insignificante.
2. Tolerable.
3. Importante.
4. Muy grave.

La exposición es el producto entre la probabilidad y el impacto del riesgo, representando con este valor la gravedad del mismo. El análisis de dichos riesgos puede apreciarse en la tabla 4.

² *Brainstorming* es una herramienta de trabajo grupal que facilita el surgimiento de nuevas ideas sobre un tema o un problema determinado. Permite obtener una lista de riesgos que luego serán analizados más detalladamente.

³ *Checklist* consiste en un análisis detallado de los riesgos listados, descartando aquellos irrelevantes.

Riesgos	Probabilidad	Impacto	Exposición
Problemas de coordinación entre los miembros del equipo	0.2	2	0.4
Subestimación de tamaño	0.6	3	1.8
Subestimación de los tiempos	0.6	3	1.8
Aumento de complejidad del proyecto	0.5	3	1.5
Planificaciones demasiado optimistas	0.5	2	1.0
Metodologías inadecuadas en el desarrollo del proyecto	0.4	2	0.8
No encontrar un editor con las características deseadas	0.7	4	2.8
Pérdida del trabajo realizado	0.3	3	0.9
Enfoque incorrecto de las necesidades cubiertas	0.4	2	0.8
Priorizar la programación	0.7	3	2.1
Incompatibilidad de comunicación entre el editor y algún algoritmo determinado	0.5	4	2
No encontrar algún algoritmo determinado	0.8	4	3.2
Falta de aceptación en el mercado	0.4	1	0.4

Tabla 4: Análisis de los riesgos

1.6.3 SELECCIÓN DE LOS RIESGOS

LA LEY DE PARETO Es una ley también conocida como la regla del 80/20 y establece que, de forma general y para un amplio número de fenómenos, aproximadamente el 80% de las consecuencias proviene del 20% de las causas con mayor impacto. Esta regla no tiene un fundamento teórico, sino empírico. Su validez proviene del hecho de que la aproximación del 80/20 resulta ser correcta de forma empírica en una gran variedad de fenómenos tanto naturales como humanos. El origen de este principio se encuentra en la observación experimental que realizó el economista y sociólogo Vilfredo Pareto (1848 – 1923) respecto a la distribución de la propiedad en Italia. Así, concluyó que, de hecho, el 80% de la propiedad del país correspondía tan sólo al 20% de la población. Tras estudiar la distribución de la propiedad en otros países, descubrió que seguía el mismo patrón.

Luego del análisis de los riesgos obtenidos y de una entrevista con el director de tesis se decidió aplicar el principio anteriormente mencionado para la selección de los riesgos a tratar. De esta manera, se seleccionará el 20% de los riesgos identificados (aquellos que presenten el mayor nivel de exposición), y trataremos sólo éstos, que en

Riesgos	Tipo	Probabilidad	Impacto	Exposición
No encontrar algún algoritmo determinado.	Técnico	0.8	4	3.2
No encontrar un editor con las características deseadas.	Técnico	0.7	4	2.8
Priorizar la programación.	Técnico	0.7	3	2.1

Tabla 5: Riesgos a tratar

relación a este principio asegurarían el 80% de la exposición. Debido a que se encontraron 13 riesgos, y el 20% de este valor es igual a 2.6, se tomaran 3 riesgos para analizar y elaborar el plan de riesgos. Los riesgos que fueron seleccionados se observan en la tabla 5, ordenados de acuerdo a la exposición de cada uno.

1.6.4 ELABORACIÓN DE PLANES DE RIESGOS

Para la elaboración de los planes de riesgos se realizarán dos tipos de planes. Por un lado el plan de mitigación, el cual tiene por objetivo reducir la probabilidad de ocurrencia del riesgo o el impacto que el mismo puede provocar. Por otra lado se elaborará el plan de contingencia, el cual esta comprendido por las acciones que se deberán realizar solamente si el riesgo se presenta. El plan de riesgos será aplicado para aquellos que se definieron en la etapa de selección de riesgos y se muestran en la tabla 5.

PLAN DE MITIGACIÓN

Tanto el riesgo de no encontrar algún algoritmo determinado como el de no encontrar un editor con las características deseadas son riesgos que existen cuando se intenta hacer uso de código *open source*. Por este motivo, para evitar esta potencial problemática, se propone efectuar a priori una investigación profunda de las herramientas que cuenten con las características deseadas.

Por otro lado, para evitar el riesgo de priorizar la programación, se elaboró un plan, en donde cada vez que una iteración del sistema o alguna funcionalidad relevante se finalice, se desarrolle la documentación asociada y los diagramas necesarios, para que de esta forma no se pasen por alto detalles que podrían olvidarse si se realizara este proceso al finalizar.

PLAN DE CONTINGENCIA

En caso de presentarse la eventual problemática generada por el riesgo de no encontrar un editor, el sistema deberá desarrollarse en su totalidad, lo cual implicaría ya sea un retraso en la finalización del proyecto o la entrega de un sistema que no cumple completamente con los requerimientos planteados. Lo mismo sucede si alguno de los algoritmos no es encontrado.

Por último, para el caso del riesgo de priorizar la programación, se deberá detener el desarrollo del sistema y realizar la documentación y diagramas asociados desde su inicio hasta el estado actual del sistema, ocasionando posibles pérdidas de detalles del desarrollo que no se documentaron en su momento.

MARCO TEÓRICO

2.1 REDES DE PETRI

Una red de Petri es un modelo gráfico, formal y abstracto para la representación de sistemas distribuidos y el análisis del flujo de información. Este modelo facilita la comprensión sobre la estructura y el comportamiento dinámico y estático del sistema modelado. Las redes de Petri son de utilidad principalmente en el diseño de sistemas de *hardware* y *software* para especificación, simulación y diseño de diversos problemas de ingeniería, especialmente útiles para representar procesos concurrentes, así como procesos donde puedan existir restricciones en cuanto a la simultaneidad, la precedencia o la frecuencia de eventos concurrentes.

Las redes de Petri están fuertemente asociadas a la teoría de grafos, ya que las mismas pueden representarse como un grafo dirigido bipartito compuesto por cuatro elementos:

- **Plazas:** Representan los estados del sistema. Las plazas son variables de estado que pueden tomar valores enteros.
- **Tokens:** Los *tokens* figuran como puntos negros dentro de las plazas. Éstos representan el valor específico de una condición o estado y generalmente se traducen a la presencia o ausencia de algún recurso del sistema.
- **Transiciones:** Las transiciones representan el conjunto de sucesos cuya ocurrencia produce la modificación de los estados (y en consecuencia del estado global) del sistema.
- **Arcos:** Los arcos indican las interconexiones entre las plazas y las transiciones, estableciendo el flujo de *tokens* que sigue el sentido de la flecha.

Una vez definidos sus componentes se puede decir que una red de Petri es un grafo dirigido con dos tipos de nodos: plazas y transiciones. Estos nodos están vinculados por arcos, los cuales sólo pueden conectar una plaza con una transición o viceversa. Por otro lado, una red de Petri puede ser descripta mediante dos componentes:

1. Una estructura de red
2. Un marcado inicial

La estructura de red hace referencia a la red en sí, mientras que el marcado inicial sólo representa el estado inicial del sistema, es decir sin que ninguna transición haya sido disparada. Un ejemplo simple

de una red de Petri marcada se muestra en la figura 3. Éste será utilizado en las secciones siguientes para ilustrar operaciones y/o propiedades de las redes de Petri.

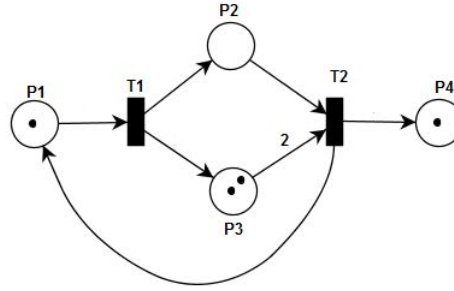


Fig. 3: Red de Petri simple

2.1.1 ESTRUCTURA DE UNA RED DE PETRI ORDINARIA

La estructura de una red de Petri puede definirse como una tupla de 5 elementos (5-tupla) de la siguiente manera:

$$N = \{P, T, I^-, I^+, M_0\} \quad (1)$$

Donde:

- $P = \{P_1, P_2, \dots, P_n\}$ es un conjunto finito y no vacío que contiene todas las plazas de la red.
- $T = \{T_1, T_2, \dots, T_m\}$ es un conjunto finito y no vacío que contiene todas las transiciones.
- I^- y I^+ son las matrices *pre* y *post* respectivamente, cuya composición se abordará en la próxima sección.
- M_0 es el marcado inicial de la red. Definido como un vector con un elemento para cada plaza, donde $M_0[i]$ contendrá la cantidad de *tokens* en la plaza i para el estado inicial.

Siguiendo con el ejemplo propuesto en la sección anterior (figura 3), se puede representar la red como $N = \{P, T, I^-, I^+, M_0\}$ donde:

- $P = \{P_1, P_2, P_3, P_4\}$
- $T = \{T_1, T_2\}$
- $M_0 = [1, 0, 2, 1]$

A continuación se explicará la manera de obtener las matrices I^+ e I^- .

2.1.2 MATRIZ DE INCIDENCIA

Las matrices I^+ e I^- son las funciones de incidencia de entrada y salida de las plazas. Para el caso de la matriz I^+ , denominada *post*, se tiene que cada elemento $post(P_i, T_j)$ contiene el peso asociado al arco que va desde T_j hasta P_i . Este peso indica la cantidad de *tokens* que se generan en la plaza P_i cuando la transición T_j es disparada.

Por otro lado, en la matriz I^- , denominada *pre*, cada elemento $pre(P_i, T_j)$ contiene el peso asociado al arco que va desde P_i hasta T_j e indica la cantidad de *tokens* que se retiran de la plaza P_i cuando se dispara la transición T_j .

Siguiendo con el ejemplo de la figura 3, las matrices I^+ e I^- asociadas son:

$$I^- = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 2 \\ 0 & 0 \end{pmatrix} \quad (2)$$

Fijarse bien (DUDAS)

$$I^+ = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \quad (3)$$

Las filas de las matrices representan las plazas mientras que las columnas representan las transiciones, lo cual quiere decir que las matrices tendrán tantas filas como plazas tenga la red de Petri, y tantas columnas como transiciones.

De esta forma se puede observar como el elemento $I^- [0][0]$ indica la relación de salida entre P_1 y T_1 . Más precisamente indica que cuando T_1 se dispara, solo un *token* es retirado de P_1 (ya que el peso del arco entre P_1 y T_1 es 1). De igual manera, el elemento $I^+ [0][0] = 0$ indica que cuando la misma transición se dispara, no se genera ningún token en P_1 (ya que no existe ningún arco que parta de T_1 hacia P_1).

A partir de estas definiciones, se puede obtener la matriz de incidencia de la red. La misma está definida a continuación:

$$I = I^+ - I^- \quad (4)$$

Cabe aclarar que una red de Petri puede reconstruirse completamente a partir de sus matrices I^+ e I^- , pero no así si se tiene sólo la matriz de incidencia. Esto quiere decir que puede haber varias redes de Petri distintas con la misma matriz de incidencia, pero solamente una para las matrices I^+ e I^- . Sin embargo, cuando una red de Petri no tiene autobucles¹, su matriz de incidencia determina com-

¹ Un *autobucle* se presenta cuando se tienen dos arcos (con sentidos contrarios) entre una misma plaza y transición

pletamente su estructura.

EJEMPLO La matriz de incidencia asociada a la red de Petri de la figura 3 será entonces:

$$I = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 2 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} -1 & 1 \\ 1 & -1 \\ 1 & -2 \\ 0 & 1 \end{pmatrix} \quad (5)$$

2.2 DINÁMICA DE UNA RED DE PETRI

Una vez definida la estructura de las redes de Petri es necesario introducir el comportamiento dinámico de las mismas. A grandes rasgos, este comportamiento está dado por la sensibilización y el disparo de sus transiciones.

2.2.1 SENSIBILIZADO DE UNA TRANSICIÓN

Se dice que una transición está sensibilizada cuando el marcado de todas las plazas entrantes a la transición es mayor o igual al peso de los arcos que las unen con dicha transición.

Antes de expresar la condición de sensibilizado de manera general, es necesario definir los siguientes conjuntos y funciones:

- $\bullet T_j$ es el conjunto compuesto por las plazas entrantes a T_j .
- $T_j \bullet$ es el conjunto compuesto por las plazas salientes de T_j .
- $m_k(P_i)$ es el marcado de la plaza P_i antes de disparar la transición T_j .
- $m_{k+1}(P_i)$ es el marcado de la plaza P_i después de disparar la transición T_j .
- w_{ij} es el peso del arco $P_i \rightarrow T_j$.
- w_{ji} es el peso del arco $T_j \rightarrow P_i$.

Esta definición de sensibilizado es solo válida cuando los arcos que conectan las plazas con T_j son arcos comunes.

Entonces, el sensibilizado de una transición T_j está dado por:

$$T_j \text{ está sensibilizada sii } \forall P_i \in \bullet T_j \Rightarrow m_k(P_i) \geq w_{ij} \quad (6)$$

EJEMPLO En la figura 4 se resaltan las transiciones sensibilizadas del ejemplo anterior.

- T_1 está sensibilizada (ya que el marcado de P_1 es igual al peso del arco $P_1 \rightarrow T_1$).
- T_2 no está sensibilizada (ya que el marcado de P_2 es menor al peso del arco $P_2 \rightarrow T_2$).

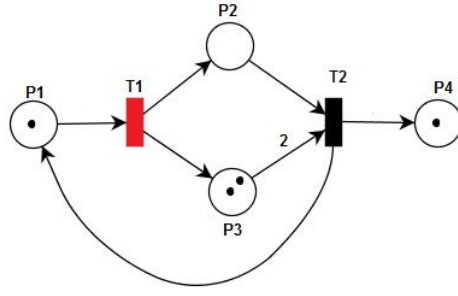


Fig. 4: Transiciones sensibilizadas en red de Petri simple

2.2.2 DISPARO DE UNA TRANSICIÓN

Si una transición está sensibilizada, la misma puede dispararse. El disparo de una transición resulta en un nuevo marcado de la red. Más precisamente, al ejecutarse una transición T_j con un marcado m_k , los marcados de las plazas pertenecientes a la red se alteran cumpliendo con las siguientes declaraciones:

$$\sigma(m_k, T_j) = \begin{cases} \forall P_i \in \bullet T_j \Rightarrow m_{k+1}(P_i) = m_k(P_i) - w_{ij} \\ \forall P_i \in T_j \bullet \Rightarrow m_{k+1}(P_i) = m_k(P_i) + w_{ji} \\ \forall P_i \notin \bullet T_j \cup T_j \bullet \Rightarrow m_{k+1}(P_i) = m_k(P_i) \end{cases} \quad (7)$$

Es decir:

- Para todas las plazas entrantes a T_j , el nuevo marcado de cada plaza se habrá **decrementado** tantos *tokens* como peso tenga el arco $P_i \rightarrow T_j$.
- Para todas las plazas salientes de T_j , el nuevo marcado de cada plaza se habrá **incrementado** tantos *tokens* como peso tenga el arco $T_j \rightarrow P_i$.
- Para el resto de las plazas, el nuevo marcado será exactamente igual al que tenían antes del disparo de T_j .

EJEMPLO Continuando con el ejemplo anterior, se puede observar en la figura 5 el nuevo marcado de la red luego del disparo de la transición T_1 :

- La única plaza entrante a T_1 (P_1), ha **decrementado** su marcado en 1 *token* (ya que el peso del arco $P_1 \rightarrow T_1$ es 1).
- Las dos plazas salientes de T_1 (P_2 y P_3) han **incrementado** su marcado (*tokens*) de acuerdo a los pesos de los arcos correspondientes.
- La única plaza que no es entrante ni saliente de T_1 (P_4) ha mantenido su marcado original.

Cabe destacar que como consecuencia del disparo de T_1 , se ha producido la sensibilización de T_2 .

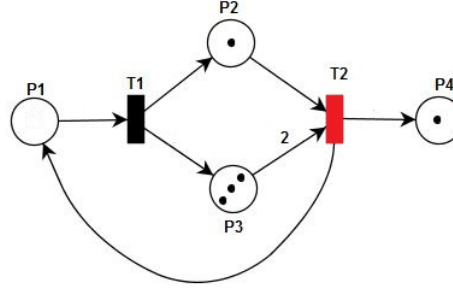


Fig. 5: Disparo de una red de Petri simple

2.2.3 FUNCIÓN DE TRANSFERENCIA Y ECUACIÓN DE ESTADO

Una vez explicada la dinámica del disparo de una transición y la forma de obtener la matriz de incidencia, se detallará una expresión matemática necesaria para obtener el nuevo marcado luego del disparo de una transición. La misma se denomina **función de transferencia** y está definida como el producto de la matriz de incidencia I con un vector $\vec{\delta}$ cuyos componentes son todos ceros, exceptuando el componente asociado a la transición que se quiere disparar, cuyo valor será uno. Entonces, se tendrá:

$$I \cdot \vec{\delta} \quad (8)$$

donde, para el disparo de una transición T_j , se tiene:

- $\delta[j] = 1$
- $\delta[i] = 0 \forall i / i \neq j$

Por otro lado, es necesario introducir la **ecuación de estado** de las redes de Petri. Con esta ecuación es posible obtener el siguiente estado del sistema luego del disparo de una transición. Esta es una manera más simple que la metodología gráfica para analizar la evolución de los sistemas. La ecuación de estado en un tiempo i , para calcular el nuevo marcado de la red en un tiempo $i + 1$ se define como:

$$M_{i+1} = M_i + I \cdot \vec{\delta} \quad (9)$$

donde M_{i+1} es el marcado luego del disparo de la transición, M_i es el marcado antes del disparo y el segundo término de la ecuación es la función de transferencia.

EJEMPLO Siguiendo con el ejemplo hasta ahora analizado se verá como calcular el marcado de la red luego del disparo de la transición T_1 haciendo uso de la ecuación de estado.

En la figura 4 se observan las transiciones sensibilizadas de la red para el marcado inicial M_0 . En este caso, sólo T_1 puede dispararse. La ecuación de estado requiere tres elementos:

1. El marcado antes del disparo. Éste es:

$$M_i = M_o = \begin{pmatrix} 1 \\ 0 \\ 2 \\ 1 \end{pmatrix} \quad (10)$$

2. La matriz de incidencia de la red. La misma fue calculada en la sección 2.1.2 y es la siguiente:

$$I = \begin{pmatrix} -1 & 1 \\ 1 & -1 \\ 1 & -2 \\ 0 & 1 \end{pmatrix} \quad (11)$$

3. El vector de disparo $\vec{\delta}$, que tendrá tantos elementos como transiciones haya en la red, cuyos valores serán cero para todas las transiciones excepto para aquella que se desee disparar:

$$\vec{\delta} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad (12)$$

con lo cual el nuevo marcado está definido por:

$$M_{i+1} = \begin{pmatrix} 1 \\ 0 \\ 2 \\ 1 \end{pmatrix} + \begin{pmatrix} -1 & 1 \\ 1 & -1 \\ 1 & -2 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad (13)$$

$$M_{i+1} = \begin{pmatrix} 0 \\ 1 \\ 3 \\ 1 \end{pmatrix} \quad (14)$$

resultado que coincide con lo obtenido en la figura 5. La ecuación de estado representa matemáticamente el comportamiento dinámico del sistema, permitiendo calcular el nuevo estado del mismo luego de la ocurrencia de un evento a través de una simple ecuación.

2.2.4 EXTENSIÓN DE LA ECUACIÓN DE ESTADO

Como se mencionó en la sección anterior, la ecuación 9 permite calcular el siguiente estado luego del disparo de una transición. Sin embargo, puede que se desee obtener el marcado final luego de una secuencia de disparos. Suponiendo que se parte del estado inicial M_0 , esto puede representarse como:

$$M_i = M_o + I \cdot \sum_{j=1}^i U_j \quad (15)$$

donde la sumatoria representa un vector asociado a la secuencia de transiciones que se desea disparar y se denomina vector \vec{S} . Para ejemplificar, el cálculo de un marcado M_i a partir del marcado inicial y luego del disparo de las transiciones $\{T_1, T_2, T_3, T_1\}$ está dado por:

$$M_i = M_o + I \cdot \vec{S} \quad (16)$$

donde $\vec{S} = \{2, 1, 1\}$ e I es la matriz de incidencia asociada a la red.

2.3 PROPIEDADES DE LAS REDES DE PETRI

2.3.1 PROPIEDADES DE LIMITACIÓN

Dada una red de Petri definida por $PN = \{P, T, I^+, I^-, M_o\}$, se dice que una plaza P está **k-limitada** si existe un número entero k que, para todo marcado posible de la red, se verifica que la cantidad de tokens de la plaza siempre es igual o menor a k . Es decir:

$$\exists k \in \mathbb{N} \quad / \quad \forall M \in \text{marcados}(PN) \Rightarrow M(P) \leq k \quad (17)$$

Por otro lado, se dice que la red está **k-limitada** si todas las plazas que contiene son **k-limitadas**.

A partir de la definición de limitación surgen varios conceptos, entre los cuales se encuentran los siguientes:

- Una red de Petri es **segura** si todas sus plazas son **1-limitadas**. Esto significa que nunca puede darse un disparo si la plaza de llegada ya contiene un *token*.
- Una red de Petri es **cíclica** si siempre existe la posibilidad de alcanzar el marcado inicial desde cualquier otro marcado alcanzable. Es decir, $\forall M \in \text{marcados}(PN)$, M_o es dinámicamente alcanzable desde M .
- Una red de Petri es **repetitiva** si existe una secuencia de disparos σ que contiene todas las transiciones de la red y existe un marcado M que para el cual $M \xrightarrow{\sigma} M$. Es decir, existe una secuencia de disparos que contiene todas las transiciones y que lleva la red del marcado actual al mismo marcado.
- Una red de Petri es **conservativa** si se cumple que $\forall M \in \text{marcados}(PN)$, el número total de *tokens* en el marcado M es igual al número de tokens en el marcado M_o . En otras palabras, la red siempre contiene la misma cantidad de marcas.

2.3.2 PROPIEDADES DE VIVACIDAD

La **vivacidad** de una transición indica que, en todo instante de la evolución de la red, su disparo es posible. Este concepto es particularmente relevante ya que determina si la ejecución de la red puede o no detenerse en un estado determinado. A partir de esto se puede

definir la vivacidad de una red de Petri. Esta propiedad indica que una red $PN = \{P, T, I^+, I^-, M_0\}$ es viva para un marcado si todas sus transiciones lo son.

Por otro lado, la **cuasi-vivacidad** de una transición expresa la posibilidad de dispararla al menos una vez a partir de un marcado inicial M_0 . De la misma manera que para el caso de la vivacidad, una red de Petri es cuasi-viva si todas sus transiciones lo son.

Gracias a esta última definición, se puede definir la vivacidad en función de la **cuasi-vivacidad** de la siguiente manera: una transición es viva si la misma es cuasi-viva en la red para todo marcado alcanzable desde M_0 .

DEADLOCK La vivacidad está directamente asociada con la ausencia de *deadlock* o interbloqueo. En términos generales, el **deadlock es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente que compiten por recursos del sistema o bien se comunican entre ellos**. En el caso de una red de Petri, esto suele ocurrir cuando dos o más transiciones esperan mutuamente por el disparo de la otra, produciendo el bloqueo permanente de esa porción de la red. **Una red de Petri viva garantiza la ausencia de interbloqueo sin importar la secuencia de disparos.**

2.3.3 ALCANZABILIDAD DE UNA RED DE PETRI

La **alcanzabilidad** de una red de Petri es fundamental para el análisis de las propiedades dinámicas de un sistema. A grandes rasgos, permite determinar si el sistema modelado puede alcanzar un determinado estado.

Un marcado M_i es alcanzable desde M_0 si existe una secuencia finita de disparos σ tal que $M_0 \xrightarrow{\sigma} M_i$.

Los marcados alcanzables por la red pueden ser representados como nodos de un grafo o árbol, donde los arcos indican los disparos necesarios para alcanzar dicho marcado. El algoritmo para determinar el árbol de alcanzabilidad de una red de Petri será explicado con detalle en el desarrollo del proyecto.

Entonces, el grafo de alcanzabilidad A se define como el menor conjunto que cumpla con las expresiones 18 y 19.

$$M_0 \in A \quad (18)$$

Esta condición simplemente aclara que el marcado inicial de la red siempre forma parte del grafo de alcanzabilidad, ya que el mismo no requiere ninguna secuencia de disparos para ser alcanzable.

$$\forall M \text{ sii } M \xrightarrow{\sigma} M_i \Rightarrow M_i \in A \quad (19)$$

Esto significa que para cualquier marcado M , si a partir del mismo puede alcanzarse otro marcado M_i , entonces M_i forma parte del grafo.

EJEMPLO Suponiendo una red simple como la de la figura 6, compuesta por dos plazas y una única transición T_1 , se puede afirmar que sólo hay tres estados posibles en la red:

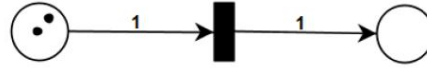


Fig. 6: Red de Petri de tres estados

- El marcado inicial $[2, 0]$
- El marcado obtenido al disparar T_1 $[1, 1]$
- El marcado obtenido al disparar nuevamente T_1 $[0, 2]$

Luego del segundo disparo de T_1 no existen disparos posibles. Por lo tanto, la red de la figura 6 produce el grafo de alcanzabilidad mostrado en la figura 7.

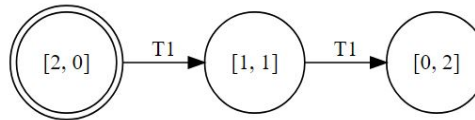


Fig. 7: Grafo de alcanzabilidad

2.3.4 COBERTURA DE UNA RED DE PETRI

El algoritmo para obtener el árbol de alcanzabilidad de una red de Petri no converge si la red no es acotada, ya que habrá plazas cuya cantidad de *tokens* seguirá aumentando indefinidamente, con lo que también lo harán los nodos del grafo de alcanzabilidad. Para estos casos existe otro tipo de análisis denominado **grafo de cobertura**.

El grafo de cobertura consiste en graficar todas las marcas posibles de la red, pero colapsando en un único marcado genérico aquellas plazas cuya cantidad de *tokens* crecerá infinitamente a medida que la red evolucione. De esta manera se representa con el símbolo ω el marcado de la plaza cuyo valor es un **entero arbitrariamente alto** que no afectará la evolución de la red.

EJEMPLO Si se analiza brevemente la red ilustrada en la figura 8, la cual está compuesta de tres plazas y de cuatro transiciones, se puede afirmar que la red es **no acotada**. Esto puede observarse por ejemplo en la plaza P_2 . Cuando T_3 o T_4 son disparadas, el marcado de P_2 aumenta. Por otro lado, cuando se dispara T_1 o T_2 , su marcado

permanece intacto. Esto implica que el marcado de la plaza P_2 es incrementado a medida que la red evoluciona, pero el mismo nunca decrementa.

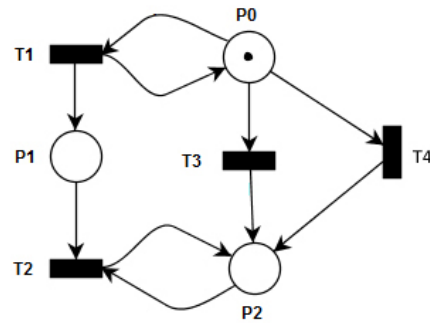


Fig. 8: Red de Petri no acotada

Entonces, el grafo de cobertura asociado a la red no acotada de la figura 8 puede apreciarse en la figura 9, donde el estado con doble círculo representa el marcado inicial y los arcos representan el disparo de las transiciones.

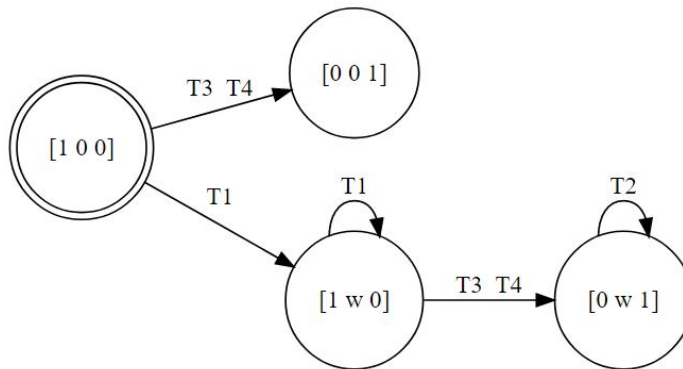


Fig. 9: Grafo de cobertura

Cabe aclarar que, cuando la red de Petri es acotada, el árbol de cobertura y el de alcanzabilidad son exactamente iguales.

2.3.5 SIFONES Y TRAMPAS

Los conceptos de sifón y trampa están directamente relacionados con las propiedades de **interbloqueo** y **vivacidad** de una red de Petri.

Un **sifón** se define como un subconjunto no vacío de plazas S para el cual se cumple que el subconjunto de transiciones entrantes a S está contenido dentro del subconjunto de transiciones salientes de S . En otras palabras, un grupo de plazas es un sifón si, una vez que un *token* sale del grupo de dichas plazas, el mismo nunca puede volver a entrar. En la figura 10 se puede observar una red que contiene una trampa y un sifón.

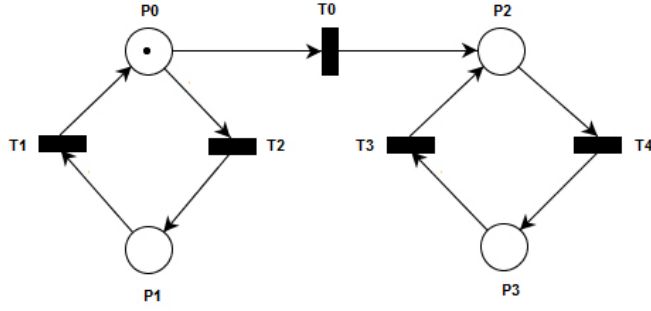


Fig. 10: Red de Petri con trampa y sifón

Tomando el subconjunto de plazas $S = \{P_0, P_1\}$ se deben obtener los siguientes subconjuntos de transiciones:

- El subconjunto $\bullet S = \{T_1, T_2\}$ será aquel compuesto por las transiciones entrantes a las plazas que componen S .
- El subconjunto $S \bullet = \{T_0, T_1, T_2\}$ será aquél compuesto por las transiciones salientes de las plazas que componen S .

Por propiedad de los sifones, para que S pueda considerarse como tal debe cumplirse que:

$$\bullet S \subseteq S \bullet \quad (20)$$

En este caso, se comprueba que $\{T_1, T_2\} \subseteq \{T_0, T_1, T_2\}$, quedando demostrado que $\{P_0, P_1\}$ es en efecto un sifón y que, si la transición T_0 se dispara, el *token* removido de P_0 nunca volverá a ingresar al subconjunto.

Por otro lado, una **trampa** se define como un subconjunto de plazas G para el cual se cumple que el subconjunto de transiciones salientes de G está contenido dentro del subconjunto de transiciones entrantes a G . Esto quiere decir que un conjunto de plazas constituyen una trampa si una vez que un *token* entra dicho grupo éste nunca vuelve a salir.

Siguiendo con el ejemplo de la figura 10, se analizará el subconjunto de plazas $G = \{P_3, P_4\}$. Los subconjuntos de transiciones serán:

- El subconjunto $\bullet G = \{T_0, T_3, T_4\}$ será aquél compuesto por las transiciones entrantes a las plazas que componen G .
- El subconjunto $G \bullet = \{T_3, T_4\}$ será aquél compuesto por las transiciones salientes de las plazas que componen G .

Por propiedad de las trampas, para que G pueda considerarse como tal, debe cumplirse que:

$$G \bullet \subseteq \bullet G \quad (21)$$

Propiedad que es simplemente comprobable ya que $\{T_3, T_4\} \subseteq \{T_0, T_3, T_4\}$, demostrando que $\{P_3, P_4\}$ es una trampa.

2.3.6 INVARIANTES DE PLAZAS Y TRANSICIONES

Las invariantes de una red son propiedades independientes tanto del marcado inicial como de la secuencia de disparos, y pueden asociarse a ciertos subconjuntos de plazas o de transiciones; con lo cual surgen dos conceptos:

P-INVARIANTES

Una **invariante de plazas** o **p-invariante** es un conjunto de plazas cuya suma de tokens no se modifica con una secuencia de disparos arbitraria. Esto se puede observar en el ejemplo de la figura 11:

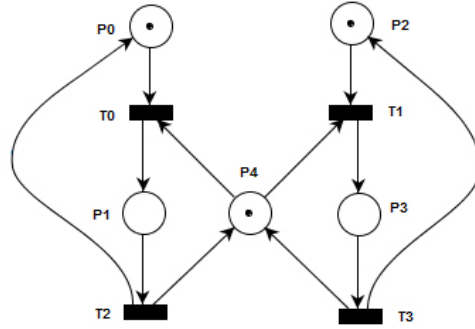


Fig. 11: Red de Petri con invariantes de plazas

Tras el análisis de la red (ya sea a través de las ecuaciones matemáticas que se mostrarán a continuación o de algún algoritmo explicado en el capítulo 7), se obtienen las siguientes **invariantes de plazas**:

- $m(P_0) + m(P_1) = 1$
- $m(P_2) + m(P_3) = 1$
- $m(P_1) + m(P_3) + m(P_4) = 1$

El primer ítem expresa que la sumatoria de *tokens* en las plazas P_0 y P_1 siempre será igual a uno, afirmación completamente observable al mirar la red de la figura 11. Estas declaraciones implican la siguiente consecuencia:

$$I \cdot x = 0 \quad (22)$$

donde I es la matriz de incidencia y x es un vector característico de un subconjunto Q de las plazas que forman parte de la invariante (un uno en una posición indica que esa plaza es parte de la invariante y un cero indica lo contrario). A partir de esto surge la siguiente fórmula:

$$\sum_{P \in \bullet t \cap Q} W(p, t) = \sum_{P \in t \bullet \cap Q} W(t, p) \quad (23)$$

la cual puede ser expresada en función de un vector t de la siguiente manera:

$$\sum_{P \in \bullet t \cap Q} t(p) = \sum_{P \in t \bullet \cap Q} t(p) \quad (24)$$

Esto quiere decir que:

$$\sum_{p \in (t \bullet \cup \bullet t) \cap Q} t(p) = 0 \quad y \quad \sum_{p \in Q} t(p) = 0 \quad (25)$$

Si reemplazamos Q por los vectores característicos I_q , estas dos igualdades pueden escribirse como:

$$\sum_{p \in Q} t(p) I_q(p) = 0 \quad y \quad \sum_{p \in P} t(p) I(p) = 0 \quad (26)$$

Lo cual es simplemente la definición del producto escalar entre dos vectores:

$$t \cdot I_q = 0 \quad (27)$$

Como los disparos son arbitrarios, podemos establecer la siguiente relación:

$$t_j I_q; \forall t_j \in T \iff I^T I_q = 0 \quad (28)$$

donde I^T es la matriz de incidencia transpuesta.

T-INVARIANTES

Una **invariante de transición** ó **t-invariante** es el conjunto de transiciones que deben dispararse para que la red de Petri retorne a su estado inicial.

Como se mencionó en el apartado anterior, para el cálculo de las p-invariantes se hace uso de la ecuación $I \cdot x = 0$, siendo I la matriz de incidencia y x un vector característico constituido por las plazas que forman parte de la invariante.

En este caso, para el cálculo de los vectores que constituyen las t-invariantes, la ecuación asociada será similar a las p-invariantes, a diferencia que se hace uso de I^T en vez de I :

$$I^T \cdot x = 0 \quad (29)$$

Aquí, a diferencia de las p-invariantes, el vector x está constituido por el conjunto de transiciones que deben dispararse para que la red retorne al estado inicial. Un uno en una posición indica que esa transición es parte de la invariante y un cero indica lo contrario.

Tomando la figura 11 como ejemplo y tras realizar el análisis explicado en el capítulo 7 se obtienen las siguientes invariantes de transiciones:

$$t - \text{invariantes} = \begin{pmatrix} T0 & T1 & T2 & T3 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad (30)$$

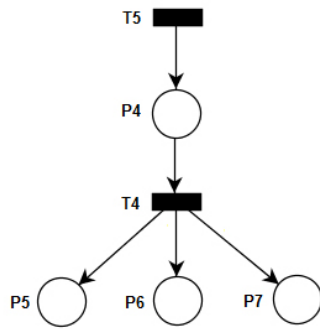


Fig. 12: Bifurcación

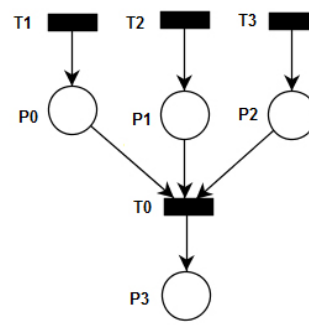


Fig. 13: Sincronización

Ambos vectores cumplen la condición planteada ($I^T \cdot x = 0$) y si se observa la imagen en cuestión, se puede apreciar que la red retorna a su estado inicial si las transiciones especificadas en los vectores se disparan.

2.3.7 PROPIEDADES DE CONCURRENCIA

Es posible representar con redes de Petri varias **primitivas de control de concurrencia entre procesos**. Entre ellas se pueden encontrar la bifurcación, la sincronización, el *cobegin/coend* y la exclusión mutua.

BIFURCACIÓN También conocido como *fork*, hace referencia a la creación de una copia del proceso que se ejecuta actualmente, el cual comienza su ejecución como un proceso hijo del proceso originario denominado proceso padre. En un entorno *multithread*, un *fork* significa que un hilo de ejecución se bifurca en dos procesos resultantes idénticos salvo por el hecho de que sus números de procesos *PID* serán distintos. Este caso se representa en la figura 12.

SINCRONIZACIÓN También llamado *join*, nos indica el punto de sincronización entre varios procesos. Sirve cuando uno o más procesos deben esperar la finalización de otro/s para finalizar. Esta situación puede observarse en la figura 13 donde la transición inferior no podrá ser disparada hasta que todos los procesos anteriores hayan terminado su ejecución.

COBEGIN/COEND Es a grandes rasgos la combinación de las dos primitivas mencionadas anteriormente. Sirve para indicar la ejecución concurrente de varios procesos y consiste de un *fork* indicando la ejecución en paralelo de los mismos, seguido de un *join* para sincronizar los resultados obtenidos. Este comportamiento se observa en la figura 14.

EXCLUSIÓN MUTUA La exclusión mutua, a veces mencionada como *mutex*, nos permite que dos o más procesos concurrentes se comuniquen usando recursos (generalmente variables) que no pueden ser accedidos simultáneamente. Esto se realiza cuando un solo proceso excluye temporalmente a todos los demás de hacer uso de un recurso

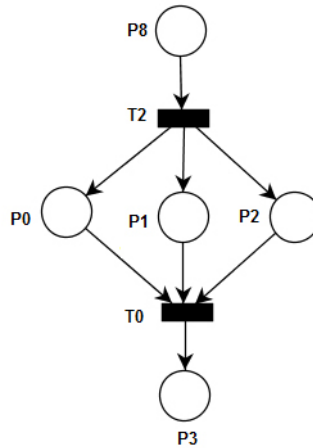


Fig. 14: Red de Petri con propiedades *cobegin/coend*

compartido con el fin de garantizar la integridad del sistema.

La exclusión mutua es fundamental, ya que su ausencia podría provocar que el acceso simultáneo y no controlado a una variable por parte de dos o más procesos afecte el resultado de una manera indeseada. Para evitar esto se deben sincronizar los procesos evitando que las funciones de acceso se intercalen. Esto se logra haciendo que las regiones de los procesos que comparten variables, conocidas como **regiones críticas**, se ejecuten como si fueran una única instrucción.

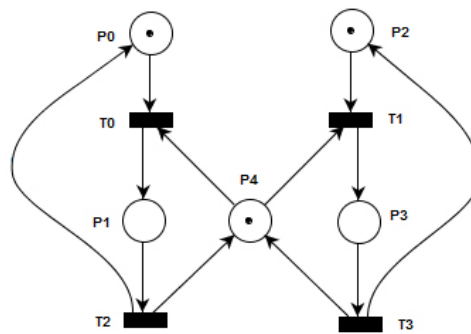


Fig. 15: Red de Petri con exclusión mutua

Esta situación es observable en la figura 15. Las regiones críticas de los procesos están representadas por las plazas P_1 y P_3 . Se utiliza la plaza P_4 como *mutex* para la sincronización entre los procesos, impidiendo que las regiones críticas se ejecuten simultáneamente.

2.4 MONITORES

Existen varias técnicas para el control de concurrencia de procesos que tienen como objetivo garantizar la integridad de un sistema. La primitiva más conocida es sin duda el semáforo. Un **semáforo** es una

variable especial que constituye un método clásico para restringir o permitir el acceso a recursos compartidos (como puede serlo una variable).

El modo básico de funcionamiento de los semáforos es el siguiente: cuando un proceso desea acceder a una variable compartida, realiza una operación *wait* sobre la variable semáforo que controla el acceso a dicha variable. Si la misma no está siendo utilizada (o la cantidad de procesos que la están utilizando es menor al máximo permitido), el semáforo otorga el permiso al proceso y cambia de estado. Cuando un nuevo proceso realice la operación *wait* sobre el mismo semáforo, el semáforo dormirá al hilo hasta que el proceso que está haciendo uso del recurso termine. Cuando un proceso cuyo acceso a la variable compartida ha sido autorizado termina de ejecutar su sección crítica, realiza una operación *signal* sobre la variable semáforo para indicar que ha terminado de utilizar el recurso.

Los semáforos son primitivas con las cuales es difícil expresar una solución a grandes problemas de concurrencia, ya que tienen algunas debilidades:

- La omisión de una de estas primitivas puede corromper la operación de un sistema concurrente.
- El control de concurrencia es responsabilidad del programador.
- Las primitivas de control se encuentran esparcidas por todo el código, lo que hace muy difícil la corrección de errores y el mantenimiento del mismo.

Debido a estas razones existe otro mecanismo de *software* para el control de concurrencia denominado monitor. El **monitor** contiene toda la información necesaria para realizar la asignación de un determinado recurso o grupo de recursos compartidos reutilizables en serie y se utiliza para manejar todas las funciones de concurrencia, comunicación entre procesos y localización física de recursos en una región crítica.

Un monitor contiene tres componentes esenciales: las variables que representan el estado del recurso, los procedimientos que implementan operaciones sobre el recurso y un código de inicialización. Desde el punto de vista lógico, un monitor consta de dos partes:

1. El algoritmo para la manipulación del recurso.
2. El mecanismo para la asignación del orden en el cual los procesos asociados pueden compartir el recurso.

La principal tarea del monitor, será garantizar la exclusión mutua del acceso al monitor mismo. El control de esta exclusión mutua está basada en una cola de entrada. Si un proceso activo está ejecutando un procedimiento del monitor y otro proceso intenta ejecutar

el mismo procedimiento u otro que también esté encapsulado dentro del monitor, el código de acceso al mismo bloquea la segunda llamada, enviándolo a la cola de entrada. Cuando un proceso activo abandona el monitor, éste toma el proceso que se encuentra al frente de la cola y lo desbloquea.

Sin embargo, el procedimiento anterior sólo controla la exclusión mutua. Puede haber casos donde un proceso activo tenga acceso al monitor (ha obtenido la exclusión mutua al mismo), pero no puede seguir su ejecución debido a alguna razón, tal como un *buffer* lleno que no puede ser escrito. En estos casos, es necesario bloquear ese proceso y permitir que otro ingrese al monitor. Para realizar esto surgen nuevos **componentes que deben formar parte del monitor**:

- **Variables de condición**
 - Las mismas son declaradas en el monitor.
 - Deben ser privadas.
 - Tienen una cola *FIFO* asociada.
- **Operaciones sobre las variables de condición.**
 - *Delay*
 - *Resume*
 - *Empty*

La operación *delay* se realiza sobre una variable de condición. Si se supone la existencia de una variable *c*, al realizar *delay(c)*, el proceso que la ejecutó libera el *mutex* del monitor, se bloquea y se envía al final de la cola asociada a la condición *c*. A diferencia de la operación *wait* que se utiliza en los semáforos, *delay* bloquea al proceso incondicionalmente.

La operación *resume*, cuando se realiza sobre una variable *c*, libera al primer proceso que ejecutó *delay(c)*. Si la cola está vacía, *resume* es una operación nula. Por otra parte, la función *empty* simplemente devuelve un valor *boolean true* si una cola se encuentra vacía o *false* en caso contrario.

Entonces se puede resumir el comportamiento de un monitor en la figura 16², donde existen una cola de entrada y una cola distinta por cada variable de condición (en el caso de la imagen, **a.q** es la cola asociada a la variable de condición **a** y **b.q** aquella asociada a **b**). En la cola de entrada se encuentran los procesos que están esperando la exclusión mutua del monitor, mientras que en el resto se encuentran aquellos que estén esperando eventos de diferente naturaleza, como puede ser la extracción de datos de un *buffer* saturado. Los procesos entran a las colas de condición cuando realizan una operación *wait(variable)*, y salen de las mismas cuando otro proceso realiza una

² Theodore Norvell, 2008 [Online]. Disponible: <https://commons.wikimedia.org/w/index.php?curid=21038822>.

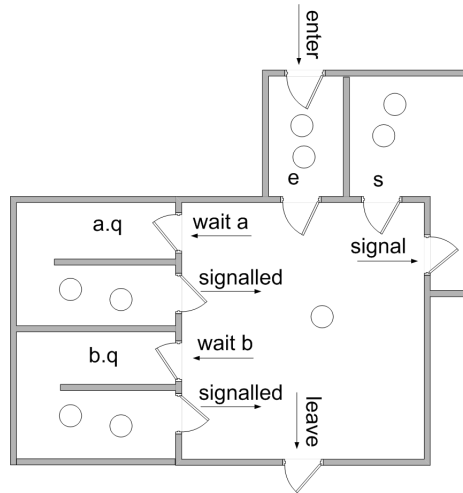


Fig. 16: Diagrama de un monitor

operación *resume(variable)* (aunque la imagen muestra la palabra *signal*, la misma es equivalente a *resume*). Es importante destacar que los procesos que se encuentren en las colas asociadas a las variables de condición siempre tienen prioridad sobre aquellos que se encuentran en la de entrada.

2.5 REDES SPN Y GSPN

Una *stochastic Petri net* (SPN) se obtienen al asociar a cada transición de una red de Petri una variable aleatoria con distribución exponencial que expresa el retardo desde la habilitación hasta el disparo de la misma. En términos formales una *Stochastic Petri net* está definida por una tupla de 2 elementos (2-tupla) de la siguiente manera:

$$SPN = \{N, L\} \quad (31)$$

Donde:

- $N = \{P, T, I^-, I^+, M_0\}$ es la estructura de la red.
- $L = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$ es el conjunto finito de los valores que representan el *rate*³ de la transición t_i .

Como se mencionó anteriormente el tiempo de disparo de una transición t_i está asociado a una distribución exponencial. La misma define un número aleatorio X_i mediante la siguiente ecuación:

$$F_{x_i}(x) = 1 + e^{-\lambda_i x} \quad (32)$$

Un típico ejemplo de una *stochastic Petri net* es ilustrado en la figura 17, la cual posee un marcado inicial $M_0 = [1, 0, 0, 0, 0]$. Si se observa esta figura se puede apreciar que la transición t_1 se encuentra sensibilizada, y la misma tiene asociada una distribución exponencial con

³ El *rate*, o también conocido como λ , es el valor característico asociado a la distribución exponencial.

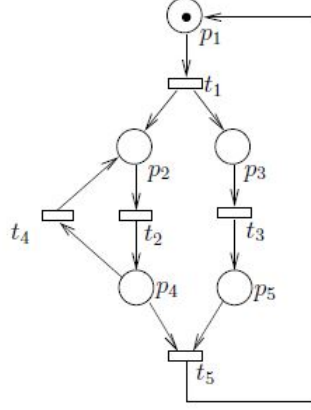


Fig. 17: Red de Petri estocástica

un valor de *rate* λ_1 ; por lo tanto el tiempo medio para que t_1 se dispare es de $1/\lambda$.

Una vez que t_1 se dispara se obtiene un nuevo marcado $M_1 = [0, 1, 1, 0, 0]$. En este punto se puede observar que tanto t_2 como t_3 se encuentran sensibilizadas concurrentemente. Si la transición t_2 se dispara primero, el marcado será $M_3 = [0, 1, 0, 0, 1]$, si en cambio el disparo lo realiza la transición t_3 el marcado que se obtendría sería $M_3 = [0, 1, 0, 0, 1]$. La decisión de cual de estas dos transiciones será disparada primero se define de acuerdo a las siguientes ecuaciones.

La probabilidad que t_2 sea disparada primero está dado por:

$$\begin{aligned}
 P[t_2 \text{ fires first}] &= P[X_2 < X_3] \\
 &= \int_0^{\infty} \left(\int_0^x \lambda_2 e^{-\lambda_2 y} dy \right) \lambda_3 e^{-\lambda_3 x} dx \\
 &= \int_0^{\infty} (1 - e^{-\lambda_2 x}) \lambda_3 e^{-\lambda_3 x} dx \\
 P[t_2 \text{ fires first}] &= \frac{\lambda_2}{\lambda_2 + \lambda_3} \tag{33}
 \end{aligned}$$

De la misma forma:

$$P[t_3 \text{ fires first}] = \frac{\lambda_3}{\lambda_3 + \lambda_2} \tag{34}$$

En una *general stochastic Petri net* (GSPN) existen dos clases de transiciones: transiciones inmediatas y transiciones temporales. Cuando una transición inmediata está sensibilizada, ésta es disparada instantáneamente. En cambio, una temporal se dispara después de un valor aleatorio exponencialmente distribuido, este es el caso de las *stochastic Petri net*.

En la figura 18 se representa un productor-consumidor que modela una *general stochastic Petri net*. Las transiciones temporales son

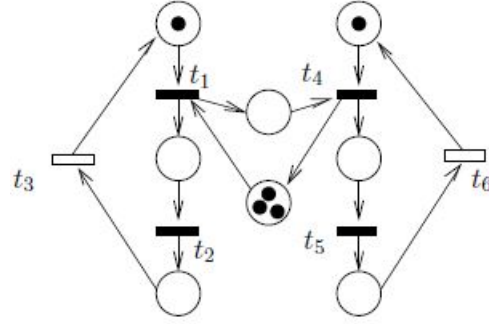


Fig. 18: Red de Petri estocástica generalizada

representadas por las barras vacías, mientras que las transiciones inmediatas por las rellenas. Dado que se asume que producir o consumir un elemento requiere mayor tiempo que agregar o quitar uno del *buffer*, se decidió modelar las transiciones de consumo y de producción como temporales (t_3, t_6), siendo las demás inmediatas.

Como siempre, varias transiciones se pueden sensibilizar dado un marcado M_i . Si el conjunto de transiciones sensibilizadas, denotado por $S_t(M)$, comprende solo transiciones temporales, entonces $t_i \in S_t(M)$ sera disparada con una probabilidad:

$$= \frac{\lambda_i}{\sum_{t_j \in S_t(M)} \lambda_j} \quad (35)$$

Si en cambio el conjunto de transiciones sensibilizadas $S_t(M)$ comprende tanto transiciones temporales como inmediatas, aquellas inmediatas tendrán prioridad sobre las temporales.

En términos formales una *general stochastic Petri net* está definida por una tupla de 4 elementos (*4-tupla*) de la siguiente manera:

$$GSPN = \{N, T_1, T_2, L\} \quad (36)$$

Donde:

- $N = \{P, T, I^-, I^+, M_0\}$ es la estructura de la red.
- $T_1 \subseteq T$ es el conjunto de transiciones temporales, donde $T_1 \neq \emptyset$.
- $T_2 \subset T$ es el conjunto de transiciones inmediatas.
 $T_1 \cap T_2 = \emptyset, T = T_1 \cup T_2$.
- $L = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$ es el conjunto finito de los valores que representan el *rate* de las transiciones.

Cabe aclarar que si $T_2 = \emptyset$, entonces la definición de *GSPN* coincide con la definición de *SPN*.

NOTA: En la presente sección, tanto las *general stochastic Petri net* como las *stochastic Petri net* tienen asociada distribuciones exponenciales. Sin embargo es posible asociar otro tipos de distribuciones que serán explicadas en el capítulo 10.

Parte II

DESARROLLO

En esta sección se abordarán temas directamente vinculados con el desarrollo de las funcionalidades especificadas en los requerimientos. En el capítulo 3 se documentará el funcionamiento básico del editor seleccionado y se reforzará el marco teórico desarrollado en la sección anterior a medida que surja la necesidad de profundizar nuevos conceptos aún no tratados. De la misma manera, en el resto de los capítulos se definirán los objetivos y se explicará detalladamente el proceso de desarrollo para cada una de las iteraciones propuestas.

INTRODUCCIÓN

En este capítulo se explicarán brevemente el funcionamiento básico y la arquitectura del editor seleccionado, así como la manera de utilizar el monitor. La información cubierta no abarcará detalles sobre implementación, sino que simplemente servirá como paso previo para la comprensión del desarrollo de las funcionalidades especificadas en los capítulos anteriores.

Como se mencionó en la sección 1.2, el producto final que se desea alcanzar constará de tres componentes principales: un editor, un monitor y un conjunto de algoritmos para el análisis de las *redes de Petri*. El editor seleccionado para realizar el resto del proyecto es *PNEditor*¹. La selección del mismo fue el resultado de un análisis exhaustivo realizado durante la práctica profesional supervisada. Las principales características del editor elegido son:

- Una interfaz intuitiva que permite la creación, edición, almacenamiento y exportación de *redes de Petri*.
- La posibilidad de utilizar tanto arcos comunes como arcos inhibidores.
- Código escrito en *java*.
- Código abierto y bajo licencia GPL.

Por otro lado, el editor no cuenta con ciertas funcionalidades que se consideran necesarias en una herramienta de este tipo, y son exactamente aquellas que se pretende desarrollar a lo largo de este proyecto. Entre ellas se destacan:

- Análisis de *redes de Petri*: La herramienta no cuenta con ningún tipo de análisis de redes (generación de grafos, obtención de matrices, cálculo de trampas y sifones, etc).
- Simulación de la red: Tampoco existe una funcionalidad que permita el disparo de las transiciones. Cuenta sin embargo con la posibilidad de disparar una única transición al presionar la misma.

PNEditor utiliza un patrón de diseño denominado *command pattern*² (patrón por comandos), el cual se clasifica como patrón de comportamiento. El patrón *command* define la manera de establecer la comunicación entre clases o entidades. El mismo se utiliza para expre-

¹ Martin Riesz, *PNEditor*, (2010) [Online]. Disponible: <http://www.pneditor.org/>.

² Richard Carr, *Command design pattern*, (2009) [Online]. Disponible: <http://www.blackwasp.co.uk/command.aspx>.

sar pedidos o *requests* (incluyendo la llamada y el paso de parámetros) en un objeto *Command*. El objeto *Command* no posee la implementación de la funcionalidad que debe ejecutar, sino que la misma está contenida dentro de los objetos *Receiver*, de esta forma eliminando el vínculo entre la definición de los comandos y la funcionalidad en sí.

Sin embargo, ni el *Receiver* ni el *Command* son responsables de determinar el momento de ejecución del mismo. Esta tarea se lleva a cabo por el objeto *Invoker*, quien se encarga de llamar al método *execute()* del objeto *Command* que recibe como argumento.

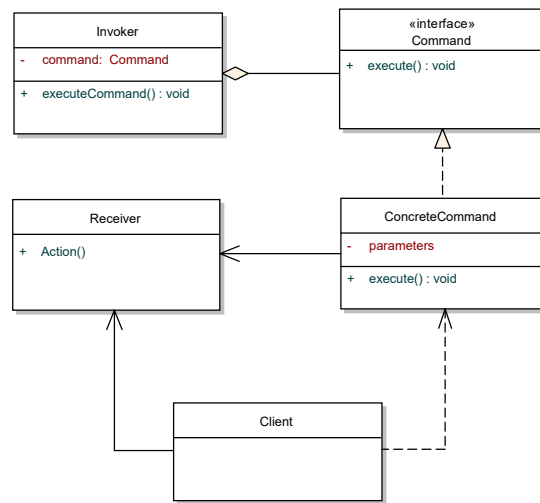


Fig. 19: Diagrama de clases típico de un *command pattern*

El diagrama de la figura 19 representa los vínculos entre las clases que conforman este tipo de patrón. Se puede observar que **existen cinco clases diferentes**:

- **Client** ó cliente: Es la clase consumidora del patrón. Crea los objetos y realiza los vínculos apropiados entre los *Commands* y los *Receivers*.
- **Receiver** ó receptor: Estos objetos contienen los métodos que se ejecutan cuando uno o más comandos se invocan. Esto permite que la verdadera funcionalidad se encuentre separada de la definición de los comandos.
- **Command** ó comando: Es una clase abstracta que representa la base de todos los objetos *Command*. Generalmente define un campo protegido que almacena el *Receiver* que está vinculado con el *Command*. De la misma forma, esta clase define un método abstracto *execute()* que el *Invoker* utiliza para ejecutar comandos.

- **ConcreteCommand** ó comando concreto: Son las subclases de la clase abstracta *Command*, por lo que definen la implementación del método *execute()*.
- **Invoker** ó invocador: Es el objeto que lanza la ejecución de un comando. El mismo puede ser por ejemplo controlado por el cliente o bien estar separado del mismo, dependiendo de la implementación.

La organización de *PNEditor* es muy similar a lo mencionado, sólo que en este caso las clases más importantes serán:

- **Root:** Es la clase principal, encargada de crear todos los objetos y definir los enlaces entre los mismos.
- **ConcreteAction:** Las acciones concretas (todas heredan de *AbstractAction*), están vinculadas a los botones del editor. Representa el cliente anteriormente mencionado ya que es responsable de llamar al invocador e informarle que comando ejecutar.
- **Command:** De la misma manera que en el ejemplo anterior, se tiene una interfaz *Command*.
- **ConcreteCommand:** Representa los diferentes comandos que implementan el método abstracto *execute()*.
- **UndoManager:** Representa el *Invoker*. Recibe del cliente el comando que debe ejecutar y llama al método *execute()* en el momento apropiado.

Un ejemplo particular puede observarse en el diagrama de clases ilustrado en la figura 20. En este caso la acción que se desea realizar es la de cortar un elemento presente en el *worksheet*, por lo que la acción ejecutada es *CutAction*. El método *actionPerformed()* de la clase *CutAction* realiza, entre otras tareas, lo siguiente:

```

1 public class CutAction extends AbstractAction
2 {
3     private Root root;
4
5     // Constructor
6
7     public void actionPerformed(ActionEvent e)
8     {
9         // Some tasks to get selected element
10        root.getUndoManager().executeCommand(new CutCommand(element));
11        // Other tasks
12    }
13 }
```

A través de un campo que contiene un objeto de la clase *Root*, se obtiene el objeto *Invoker* y se ejecuta el método *executeCommand()* del mismo, pasándole como parámetro el objeto de tipo *Command* que desea ejecutar. Es la clase *UndoManager* (la cual representa el *Invoker*) quien se encarga de ejecutar el comando. Esto se puede observar en el siguiente segmento de código:

```

1  public class UndoManager
2  {
3      private List<Command> executedCommands = new ArrayList<Command>();
4      private int currentCommandIndex = -1;
5      private Root root;
6      private UndoAction undoAction;
7      private RedoAction redoAction;
8
9      public UndoManager(Root root, UndoAction undoAction, RedoAction redoAction)
10     {
11         this.root = root;
12         this.undoAction = undoAction;
13         this.redoAction = redoAction;
14     }
15
16     public void executeCommand(Command command)
17     {
18         executedCommands.add(command);
19         currentCommandIndex = executedCommands.size() - 1;
20         command.execute();
21         refresh();
22     }
23 }

```

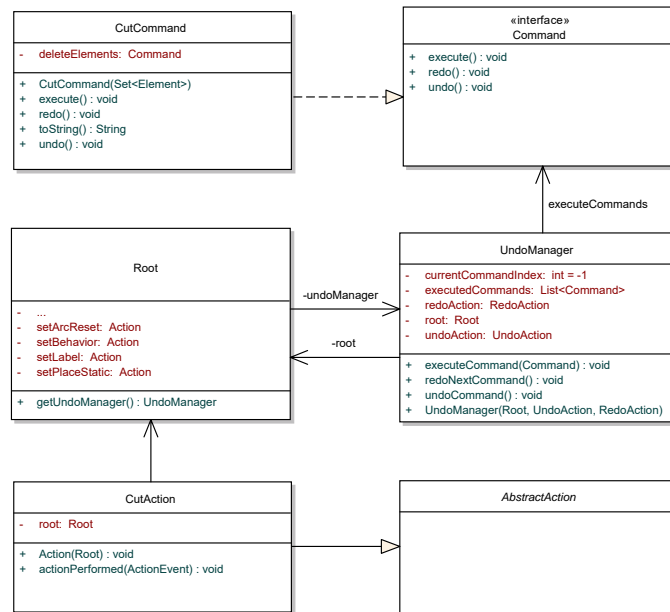


Fig. 20: diagrama de clases del patrón de diseño de PNEditor

Aquí se observa que, además de realizar ciertas tareas relacionadas con las funcionalidades de rehacer y deshacer, se ejecuta el comando en la línea número 20. Esto se realiza simplemente invocando al método *execute()* del comando que se pasó como argumento. A continuación, la responsabilidad de ejecutar la tarea es del comando *CutCommand*.

ITERACIÓN 1: IMPORTACIÓN Y EXPORTACIÓN DE ARCHIVOS PNML

4.1 INTRODUCCIÓN

En esta iteración se desea añadir al editor la funcionalidad de exportar la red de Petri a un formato compatible con el resto de los componentes del sistema. Como se mencionó en la sección 1.2, el editor se comunica tanto con los algoritmos como con el monitor. Para que pueda realizarse la comunicación entre los algoritmos y el editor, este último debe permitir la exportación de la red en un formato *.xml*. De la misma manera, para la comunicación entre el simulador y el editor es necesario que la exportación se realice en formato *.pnml* con un dialecto específico.

4.2 OBJETIVOS

Los objetivos para la iteración fueron planteados en el capítulo 1 y, al ser ésta la primera, no se presentaron aún situaciones que pudieran requerir de cambios en el plan trazado. Por lo tanto, los mismos serán:

1. Realizar la exportación de la red en formato *.pnml* con dialecto compatible con el monitor.
2. Realizar la exportación de la red en formato *.xml* con dialecto compatible con los algoritmos.
3. Modificar el almacenamiento y lectura del formato *.pflow* para que el mismo incorpore las nuevas características y propiedades agregadas en la red.
4. Permitir importar un archivo *.xml* con el dialecto utilizado por la herramienta PIPE.

4.3 DESARROLLO

PNEditor permite exportar la red en un archivo *.pnml* con un formato denominado *VipToolDialect*¹. Sin embargo, ni los algoritmos ni el monitor son compatibles con el mismo. Por este motivo, el formato de exportación que brinda el editor no puede utilizarse para establecer la comunicación. Como consecuencia de esto se necesita realizar la exportación de la red en dos dialectos diferentes. Por un lado, un archivo *.xml* (el mismo que es utilizado por el *software PIPE*) y por

¹ FernUniversität in Hagen, *VipTool*, [Online]. Disponible: http://www.fernuni-hagen.de/sttp/forschung/vip_tool.shtml.

otro, un archivo *.pnml* cuyo dialecto sea compatible con el *software TINA*. A continuación se detalla la forma de generar estos archivos, así como los elementos principales que los componen.

La estructura básica de un archivo *.pnml* es muy similar a la de uno *.xml*. El siguiente código muestra dicha estructura, en donde se omitieron las características de los elementos de la red para simplificar la misma.

```

1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <pnml>
3    <net id="Net-One" type="P/T net">
4      <!-- Declaration of places -->
5      <place id="p1">
6      </place>
7      <place id="pn">
8      </place>
9      <!-- Declaration of transitions -->
10     <transition id="t1">
11     </transition>
12     <transition id="tm">
13     </transition>
14     <!-- Declaration of arcs -->
15     <arc id="" source="pn" target="tm">
16     </arc>
17     <arc id="" source="p1" target="t1">
18     </arc>
19   </net>
20 </pnml>

```

Se puede observar que el archivo se separa en tres secciones. En primer lugar se realiza la declaración de las plazas de la red, luego de las transiciones, y por último de los arcos. Todos los elementos (ya sean plazas o transiciones) tienen asociado además de las características propias de cada uno de ellas un identificador y una posición, los cuales se utilizan para ubicar los elementos dentro del espacio gráfico del editor.

A grandes rasgos, para exportar un archivo *.pnml* o *.xml* es necesario utilizar un documento de estilo *xslt*. Este documento es la entrada de un procesador *xslt* y describe la estructura del archivo de salida que se desea generar. La disposición de las partes de un documento de este tipo se muestra a continuación:

```

1  <xsl:for-each select="place">
2    <place>
3      # information about places
4    </place>
5  </xsl:for-each>
6
7  <xsl:for-each select="transition">
8    <transition>
9      # information about transitions
10   </transition>
11 </xsl:for-each>
12
13 <xsl:for-each select="arc">

```

```

14     <arc>
15         # information about arcs
16     </arc>
17 </xsl:for-each>

```

El documento *xslt* anterior se divide en tres secciones: plazas, transiciones y arcos. Cada una de estas secciones posee diferentes *items* que hacen referencia a las particularidades de cada uno de los elementos. Por ejemplo, para el caso de las plazas en el dialecto de TINA se tienen cuatro *items* diferentes (*id*, *name tokens*, *position*). De esta forma, queda definido el archivo de salida:

```

1  <place id="place_id">
2      <name>
3          <text>placeName</text>
4      </name>
5      <initialMarking>
6          <text>placeMarking</text>
7      </initialMarking>
8      <graphics>
9          <position x="" y="" />
10     </graphics>
11 </place>

```

Para que el procesador *xslt* pueda generar este archivo (*.xml* o *.pnml* en su dialecto correspondiente), es necesario que el usuario seleccione el formato deseado y defina el nombre del mismo. Asumiendo que se selecciona el formato por defecto *VipToolDialect*, se ejecutará la función *save*.

```

1  public void save(Document document, File file) throws Exception
2  {
3      try
4      {
5          final InputStream xslt;
6          xslt = getClass().getResourceAsStream("/xslt/pnml-export.xslt");
7          PetriNet petriNet = document.getPetriNet();
8          Marking m = petriNet.getInitialMarking();
9          new DocumentExporter(document, m).writeToFileWithXslt(file, xslt);
10     }
11     catch ( ... )
12     {
13         // Handling of captured exceptions.
14     }
15 }

```

Como se observa en el anterior segmento de código, la función *save* recibe dos parámetros: *document*, que hace referencia a la red de Petri que se exportará; y *file*, que indica el nombre que tomará el archivo de salida. Por último, se crea una instancia de la clase *DocumentExporter*, la cual se utilizará para obtener la información sobre los elementos de la red, almacenándola en diferentes variables. El procesador *xslt* tomará esta información para generar el archivo de salida.

La información obtenida por la clase *DocumentExporter* consiste en los *items* definidos en el documento *xslt* y la misma se obtiene realizando consultas a diferentes clases. En la figura 21 se ilustra la secuencia de las mismas.

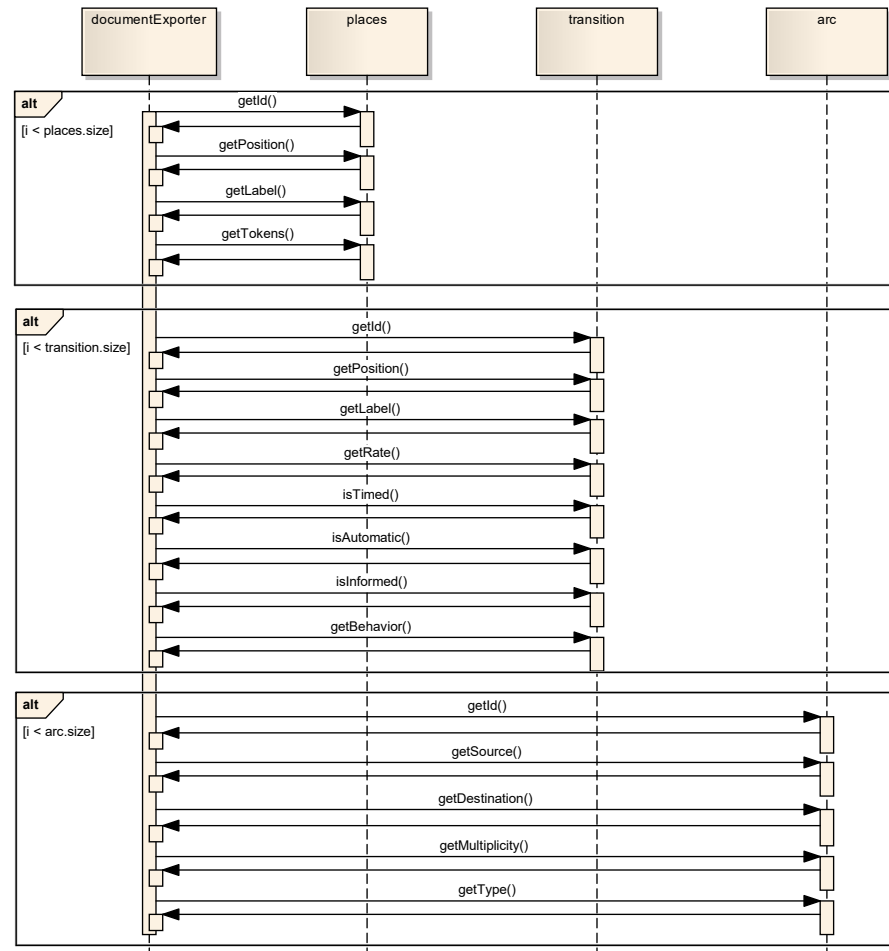


Fig. 21: Diagrama de secuencia de la obtención de datos

Como se mencionó anteriormente, el editor ya cuenta con la opción de exportar y lo hace con el dialecto *VipTool*, de manera que se extendió esta funcionalidad creando dos nuevas clases para los nuevos dialectos y aplicando las modificaciones necesarias.

Para cumplir con el objetivo asociado a permitir la exportación a un dialecto compatible con el monitor, se realizaron las siguientes modificaciones:

1. Se modificaron ciertas etiquetas, ya que por ejemplo el dialecto de *TINA* declara los valores entre etiquetas "`<text> </text>`" en lugar de "`<value> </value>`" como lo hacen el resto de los formatos.
2. Se omitió la información irrelevante dentro de la declaración gráfica de los elementos. Entre ellas se encuentran *fill*, *line* y *font*, las cuales son usadas para definir el relleno, color de contorno y fuente a la plaza y/o transición.
3. Se agregaron las propiedades de las transiciones temporales. El dialecto *TINA* expresa el tiempo de disparo entre llaves, estableciendo un valor mínimo y uno máximo. En nuestro sistema el

disparo de la transición temporal no se realiza de esta forma, sino que el mismo es expresado por un único valor.

4. Se agregó el comportamiento (*behavior*) de la transición en el campo *label* de la misma.

Por otro lado para cumplir con el objetivo de permitir la exportación compatible con los algoritmos (*PIPE*), se realizaron las siguientes modificaciones:

1. En la sección de plazas se agregó el campo *capacity*, colocando el valor por defecto cero.
2. Al igual que en el formato *TINA*, se omitió la información irrelevante dentro de la declaración gráfica (*fill*, *line* y *font*).
3. Ya que el *software PIPE* utiliza transiciones rectangulares en lugar de transiciones cuadradas, es necesario establecer la orientación de las mismas. Por este motivo se agregó el campo *orientation* (colocando el valor de 90° por defecto). Esto implica que todas las exportaciones a este formato mostrarán las transiciones con orientación horizontal.
4. Se agregaron las propiedades de las transiciones temporales. Esto incluye el campo *timed*, el cual indica si la misma es temporal o no. Además, se incluyeron los campos *rate* y *priority* utilizados para determinar el tiempo de disparo y el análisis *GSPN* sobre el cual se profundizará en el capítulo 2.

El editor cuenta además con un formato propio denominado *pflow*. A medida que se agregaron funcionalidades, se debió de igual manera agregar nuevas propiedades a los elementos que conforman la red. Por este motivo, es necesario actualizar el almacenamiento de archivos *pflow* para que el mismo contenga las características añadidas y no se pierda información en el proceso de guardado y reapertura de una red. Para realizar esto, en el bloque de transiciones del archivo *save.xslt* se añadieron las siguientes líneas:

```

1 <rate><xsl:value-of select="rate"/></rate>
2 <automatic><xsl:value-of select="automatic"/></automatic>
3 <informed><xsl:value-of select="informed"/></informed>
4 <enableWhenTrue><xsl:value-of select="enableWhenTrue"/></enableWhenTrue>
5 <guard><xsl:value-of select="guard"/></guard>

```

Por otro lado, para realizar la importación de una red en formato *.xml* compatible con *PIPE* se debe efectuar el proceso inverso al realizado para la exportación en este mismo formato. En este caso se tuvieron en cuenta los puntos de quiebre de los arcos, por lo que la distribución de los elementos de una red importada desde el *software PIPE* será exactamente la misma tanto en el sistema origen como el destino.

4.4 TESTING

Debido a que las tareas finalizadas en la presente iteración no representan una funcionalidad en sí, las mismas serán puestas a prueba en los capítulos posteriores, cuando el sistema requiera de la exportación para brindar el análisis de la red haciendo uso del formato compatible con *PIPE* o para realizar la simulación con el monitor utilizando el formato *.pnml*.

4.5 CONCLUSIONES

Esta iteración es simplemente un primer paso para realizar el resto, ya que tanto la segunda (simulación de la red de Petri) como aquellas relacionadas con el análisis necesitan de la exportación de la red para su implementación. Por este motivo, no resulta posible extraer conclusiones de la implementación realizada ya que, si bien es necesaria, la misma no representa una funcionalidad en sí. El análisis concreto de sus resultados será una combinación de las conclusiones obtenidas en las iteraciones posteriores.

ITERACIÓN 2: SIMULACIÓN DE REDES DE PETRI NO TEMPORALES

5.1 INTRODUCCIÓN

En esta iteración se pretende añadir al editor la funcionalidad de simular el comportamiento de una red. Como se explicó en la sección 2.2, el comportamiento de una red de Petri está dado por el disparo de sus transiciones. Una vez implementada esta funcionalidad, el usuario podrá presionar un botón de simulación, el cual proporcionará un diálogo donde se deberá ingresar:

- La cantidad de transiciones a disparar.
- El tiempo que debe transcurrir entre el disparo de una transición y el de la siguiente.

El monitor es quien llevará a cabo la simulación en sí, recibiendo la red como parámetro e informando al observador de las transiciones cada vez que una de ellas se dispare. De esta manera se acumulan los eventos para luego reproducirlos en el editor.

5.2 OBJETIVOS

Los objetivos para la iteración se plantearon en el capítulo 1 y no se dieron situaciones en las iteraciones anteriores que pudieran provocar la necesidad de modificar los mismos. Por lo tanto, éstos serán:

1. Desarrollar una ventana que le permita al usuario ingresar la cantidad de transiciones a disparar y el tiempo entre el disparo de cada una de ellas.
2. Crear los hilos para aquellas transiciones que no sean automáticas, de manera que cada uno intente disparar una única transición cada un período aleatorio de tiempo.
3. Implementar un *observer* y asociarlo a las transiciones de manera que el mismo reciba un evento cada vez que una transición se dispare. Los eventos se almacenarán en una lista ordenada que representará la evolución de la red durante el tiempo de simulación.
4. Realizar la simulación, proveyendo al monitor con la red y el resto de información necesaria.
5. Generar en el editor la representación de la simulación una vez que se hayan generado en el *observer* la cantidad de eventos especificada por el usuario.
6. Mostrar en el editor una lista con los eventos a medida que éstos vayan ocurriendo.

5.3 DESARROLLO

Para la implementación, se creó una nueva clase llamada *SimulateAction* que hereda de *AbstractAction* (*javax.swing*) y que está vinculada al botón *Simulate*. El comportamiento de la clase puede representarse en el diagrama de secuencia de la figura 22.

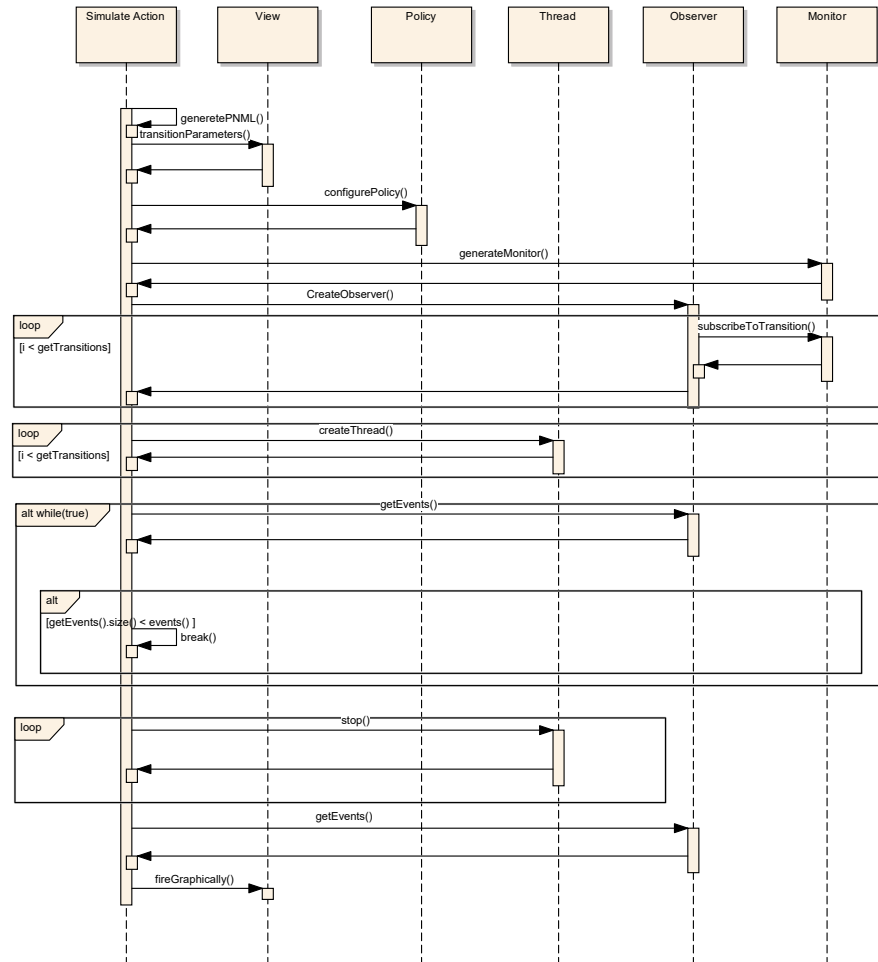


Fig. 22: Diagrama de secuencia de la clase *SimulateAction*

Como se puede observar en el diagrama, se comienza generando un archivo temporal *.pnml* con el formato especificado en el capítulo 4. Este archivo toma por nombre *tmp.pnml* y se genera de la siguiente manera:

```

1 | FileChooserDialog chooser = new FileChooserDialog();
2 | chooser.addChoosableFileFilter(new TinaPnmlFileType());
3 | chooser.setAcceptAllFileFilterUsed(false);
4 | chooser.setCurrentDirectory(root.getCurrentDirectory());
5 |
6 | File file = new File("tmp" + System.getProperty("file.separator")
7 | + "tmp" + "." + "pnml");
8 | FileType chosenFileType = (FileType) chooser.getFileFilter();
9 |
10 | try
11 | {
  
```



```

12     chosenFileType.save(root.getDocument(), file);
13 }
14 catch (FileNotFoundException e1)
15 {
16     System.out.println("The tmp file could not be saved\n");
17 }

```

Una vez generado el mismo, se abre el diálogo para que el usuario ingrese la cantidad de transiciones que desea disparar, así como el tiempo que debe transcurrir entre el disparo de cada una de ellas. Luego de obtener estos dos argumentos, se procede a la creación e inicialización del monitor, así como de sus variables asociadas. El procedimiento para realizar esto se encuentra explicado en la documentación¹ del monitor. A grandes rasgos, es necesario:

- Creación de una instancia de la clase *PetriNetFactory*, cuyo constructor recibe como parámetro el *path* al archivo temporal.
- Creación de la red de Petri como una instancia de la clase definida en el monitor. La función *makePetriNet()* recibe como parámetro el tipo de red que se desea crear (normal, temporal, etc).
- Creación de la política que el monitor utilizará para sincronizar los eventos.
- Creación del monitor, especificando como argumentos la red de Petri y la política creada.
- Inicialización de la red de Petri .

Este procedimiento se puede observar en el siguiente segmento de código:

```

1  /*
2   * Create monitor, petri net, and all related variables.
3   */
4
5  PetriNetFactory factory = new PetriNetFactory("tmp/tmp.pnml");
6  RootPetriNet petri;
7
8  try
9  {
10     petri = factory.makePetriNet(petriNetType.PLACE_TRANSITION);
11 }
12 catch (DuplicatedNameError e)
13 {
14     JOptionPane.showMessageDialog(null,
15     "Two places or transitions cannot have the same label");
16     stop = false;
17     setEnabled(true);
18     return; // Don't execute further code
19 }
20
21 TransitionsPolicy policy = new FirstInLinePolicy();

```

¹ Ariel Rabinovich, Juan Arce, "Framework de sincronización de tareas", (2017) [Online] Disponible: https://github.com/airabinovich/java_petri_engine/blob/master/README.md#java_petri_engine.

```

22 | PetriMonitor monitor = new PetriMonitor(petri, policy);
23 |
24 | petri.initializePetriNet();

```

Para comenzar, el tipo de red pasado como parámetro a la función `makePetriNet()` en la línea 10 es `place_transition`, lo cual hace referencia a un tipo de red normal (no temporal). Por otro lado, esta misma función lanza una excepción `DuplicateNameError` si encuentra dos plazas o dos transiciones con el mismo nombre. En este caso se informa al usuario que la red no puede crearse tal como está y que debe realizar las modificaciones pertinentes.

La política seleccionada en este caso es la de `FirstInLinePolicy`, la cual funciona simplemente como una cola *Wikipedia*, `FIFO`² (*first in, first out*). Hay otras políticas disponibles en el monitor y entre ellas se encuentran:

- `RandomPolicy`
- `TransitionPolicy`

De la misma manera, existe la posibilidad de crear una nueva política de forma simple y sin realizar modificaciones en el resto del monitor. Las instrucciones para realizar esto pueden encontrarse en la documentación del monitor anteriormente mencionada.

Una vez generadas todas las variables necesarias para establecer la comunicación con el monitor, fue necesario realizar tres pasos:

1. Implementación de una clase `ConcreteObserver` que implemente la interfaz `rx.Observer`. Esta clase debe sobrescribir el método `onNext(String event)`. El monitor invocará a este método cada vez que una transición se dispare.
2. Suscripción del `Observer` a todas las transiciones informadas.
3. Creación de los hilos que intentarán disparar las transiciones cada un tiempo aleatorio.

La implementación de la clase `ConcreteObserver` es la siguiente:

```

1 | public class ConcreteObserver implements Observer<String>
2 | {
3 |     private ArrayList<String> eventsRecieved;
4 |     Root root;
5 |
6 |     public ConcreteObserver(Root root)
7 |     {
8 |         super();
9 |         this.root = root;
10 |         eventsRecieved = new ArrayList<String>();
11 |     }
12 |
13 |     // Other overwritten methods that are not currently being used

```

² `FIFO`, [Online]. Disponible: [https://en.wikipedia.org/wiki/FIFO_\(computing_and_electronics\)](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)).

```

14
15     @Override
16     public void onNext(String event) { eventsRecieved.add(event); }
17
18     public ArrayList<String> getEvents() { return eventsRecieved; }
19 }

```

Como se puede observar, los únicos métodos implementados fueron `onNext()` y `getEvents()`. El método `onNext()` se ejecuta por el monitor cada vez que se dispara una transición y recibe como parámetro un objeto `string` que contiene información sobre el evento (como el nombre y el `id` de la transición que fue disparada). Por otro lado, cada vez que se recibe un evento se lo agrega a una lista y es el método `getEvents()` el responsable de retornar la misma. De esta manera, en la clase `SimulateAction`, se suscribe el `observer` a todas las transiciones informadas existentes en la red:

```

1  /*
2   * Subscribe to all transitions
3   */
4  Observer<String> observer = new ConcreteObserver(root);
5  for(int i = 0; i < petri.getTransitions().length; i++)
6  {
7      MTransition t = petri.getTransitions()[i];
8      Subscription subscription = monitor.subscribeToTransition(t, observer);
9  }

```

A continuación, se crean los hilos asociados a cada una de las transiciones no automáticas:

```

1  /*
2   * Create one thread per transition, start them all to try and fire them.
3   */
4  List<Thread> threads = new ArrayList<Thread>();
5  for(int i = 0; i < petri.getTransitions().length; i++)
6  {
7      if(!(root.getDocument().petriNet.getRootSubnet().getTransition
8          (petri.getTransitions()[i].getId()).isAutomatic()))
9      {
10         Thread t = createThread(monitor, petri.getTransitions()[i].getName());
11         threads.add(t);
12         t.start();
13     }
14 }

```

Como se puede apreciar en el código anterior, los hilos se crearon con la función `createThread(Monitor, String)`. Esto se debe a que todos los hilos tendrán el mismo comportamiento aunque cada uno disparará una transición diferente. La función en cuestión es la siguiente:

```

1  Thread createThread(PetriMonitor m, String id)
2  {
3      Thread t = new Thread(new Runnable() {
4          @Override
5          public void run()
6          {
7              while(true)

```

```

8      {
9          try
10         {
11             Thread.sleep((new Random()).nextInt(50));
12             m.fireTransition(id);
13         } catch (IllegalTransitionFiringError e) {
14             e.printStackTrace();
15         } catch (IllegalArgumentException e) {
16             e.printStackTrace();
17         } catch (PetriNetException e) {
18             e.printStackTrace();
19         } catch (InterruptedException e) {
20             e.printStackTrace();
21         }
22     }
23 }
24 });
25 return t;
26 }

```

La función crea un objeto *Runnable* cuyo método *run()* se implementó como un bucle infinito que intenta disparar la transición cuyo *id* se pasó como parámetro cada un tiempo aleatorio que varía entre 1 y 50 milisegundos. Como se puede observar, las instrucciones dentro del bucle pueden ocasionar cuatro excepciones diferentes. Las más importantes son aquellas lanzadas por el monitor: *IllegalTransitionFiringException* y *PetriNetException*. La primera se da cuando se intenta disparar una transición automática (no *fired*), aunque esta situación no se presentará nunca ya que a la hora de crear los hilos sólo se crean para aquellas transiciones no automáticas. La segunda excepción lanzada por el monitor es simplemente un aviso de que la red de Petri se encuentra en un estado donde no es posible continuar la ejecución.

Una vez realizados estos pasos, solo es necesario esperar que ocurra la cantidad de eventos (disparos de transiciones) que el usuario especificó al presionar el botón simular. Para ello basta con corroborar el tamaño de la lista de eventos contenida dentro del *observer* periódicamente como se observa a continuación:

```

1  while(true)
2  {
3      if(observer.getEvents().size() >= numberOfTransitions)
4          break;
5
6      else
7      {
8          try {
9              Thread.currentThread().sleep(10);
10             } catch (InterruptedException e1) {
11                 e1.printStackTrace();
12             }
13
14             if(checkAllAre(petri.getEnabledTransitions(), false))
15             {
16                 JOptionPane.showMessageDialog(root.getParentFrame(),
17                     "The net is blocked, " + ((ConcreteObserver) observer).

```

¿Que hace la función *checkAllAre()*

```

18         getEvents().size() + " transitions were fired.");
19         break;
20     }
21 }
22 }

```

Mientras no se haya dado la cantidad de eventos requerida se duerme el hilo que realiza la espera durante 10 milisegundos y se corrobora que la red no se encuentre bloqueada. Esto último indica que aún quedan transiciones por dispararse. Cuando se alcanza la cantidad de disparos necesaria, se interrumpe el ciclo *while*. En este momento, se detienen todos los hilos en ejecución y se llama a la función *fireGraphically()*, la cual se encarga de representar los eventos en el editor, disparando las transiciones en el orden en que las mismas se informaron al *observer* y esperando entre cada una el tiempo definido por el usuario. Las acciones principales de esta función pueden resumirse en el diagrama de secuencia de la figura 23.

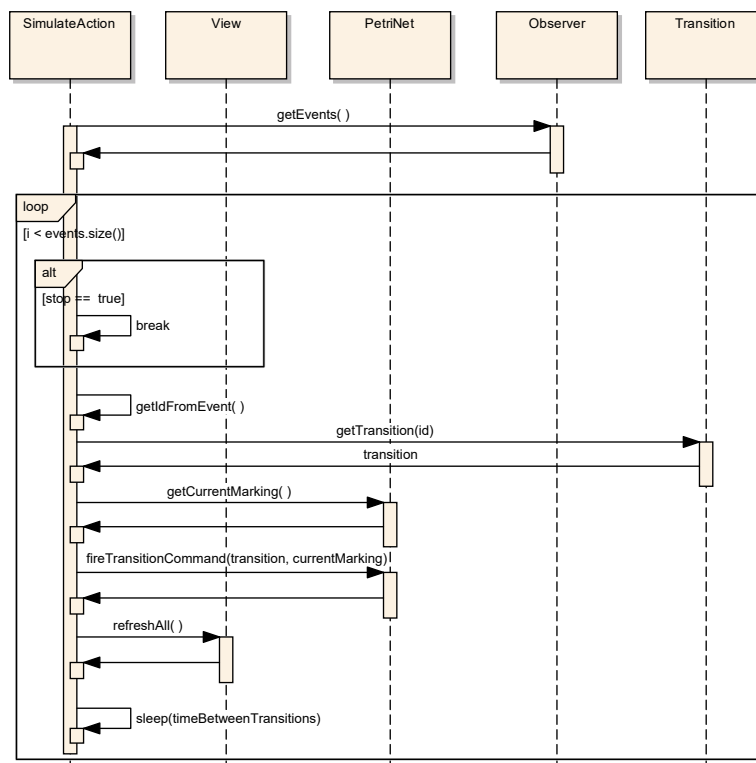


Fig. 23: Diagrama de secuencia de la función *fireGraphically()*

La función *fireGraphically()* comienza obteniendo la lista de eventos del *observer* e itera por cada uno de sus elementos. Dentro de cada iteración se analiza el evento para obtener el *id* de la transición cuyo disparo lo generó. A continuación se pide a la red de Petri el objeto *Transition* asociado a ese número de *id* y el marcado actual de la red para generar el disparo y refrescar la vista de modo que el usuario observe los cambios.

Asimismo esta función es responsable de actualizar la lista de eventos de la vista (aunque esto no se encuentra representado en el dia-

grama de la figura 23 para evitar confusiones entre la lista de eventos del *observer* y aquella de la vista). El código de esta función es el siguiente:

```

1  void fireGraphically(List<String> list,
2                          int timeBetweenTransitions,
3                          int numberOfTransitions)
4  {
5      for(String event : list)
6      {
7          /*
8           * Check if stop button has been pressed
9           */
10         if(stop)
11         {
12             stop = false;
13             setEnabled(true);
14             list.clear();
15             System.out.println(" > Simulation stopped by user");
16             return;
17         }
18
19         List<String> transitionInfo = Arrays.asList(event.split(", "));
20         String transitionId = transitionInfo.get(2);
21         transitionId = transitionId.replace("id:", "");
22
23         Transition transition = root.getDocument().
24                                 petriNet.getRootSubnet().
25                                 getTransition(transitionId);
26         Marking marking = root.getDocument().petriNet.getInitialMarking();
27         FireTransitionCommand fire;
28         fire = new FireTransitionCommand(transition, marking);
29         fire.execute();
30
31         root.getEventList().addEvent((transition.getLabel()
32 + " was fired!"));
33
34         root.refreshAll();
35
36         try
37         {
38             Thread.currentThread().sleep(timeBetweenTransitions);
39         } catch (InterruptedException e) {
40             e.printStackTrace();
41         }
42     }
43 }

```

Se puede observar de igual manera en este segmento de código que dentro de cada iteración se corrobora el estado de la **variable booleana *stop***. En caso de ser *true*, se rehabilita el botón de simular, se vacía la lista de eventos y se abandona la función. La encargada de modificar esta variable es la clase *StopSimulationAction*, quien también hereda de *AbstractAction*, y cuya única tarea es modificar el valor de la variable estática *stop* de la clase *SimulateAction* cada vez que el botón de *stop* se presiona.

5.4 TESTING

Para corroborar el correcto funcionamiento de esta funcionalidad, bastó con implementar una red simple y realizar la simulación del disparo de algunas transiciones. Para esto se tomó un ejemplo de una red que representa el comportamiento de un sistema productor-consumidor y se la graficó en el editor, como puede apreciarse en la figura 24.

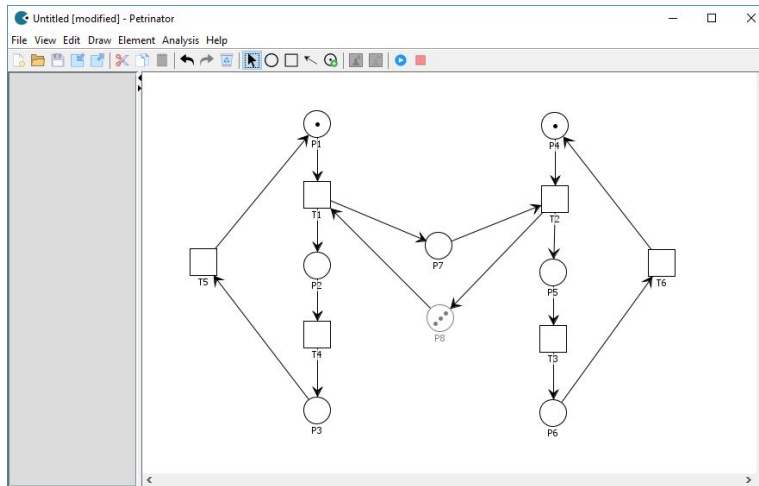


Fig. 24: Sim: red a simular

La red de Petri de un sistema productor-consumidor consta de tres partes principales:

- **Productor:** Representado por el conjunto de nodos $P = \{P_1, P_2, P_3, T_1, T_4, T_5\}$. Éste se encarga de introducir elementos en el *buffer*.
- **Consumidor:** Representado por el conjunto de nodos $C = \{P_4, P_5, P_6, T_2, T_3, T_6\}$. El mismo intenta extraer elementos del *buffer*.
- **Un *buffer* limitado:** Representado por las plazas $B = \{P_7, P_8\}$. Representa el lugar físico donde se encuentran los elementos que el productor produce y que el consumidor extrae. P_7 representa el *buffer* en sí (notar que comienza vacío) y P_8 indica la capacidad máxima del mismo (puede verificarse que el *buffer* jamás contendrá más elementos que la cantidad de *tokens* de P_8).

Para comenzar la simulación de la red, debe presionarse el botón *play* como se resalta en la figura 25.

Al presionar este botón, se abrirá un diálogo que requerirá que el usuario ingrese la cantidad de transiciones cuyo disparo desea simular, así como el tiempo que desea esperar entre el disparo de cada una de ellas. Este diálogo se observa en la figura 26, donde se ingresó una cantidad de cinco transiciones, esperando medio segundo entre

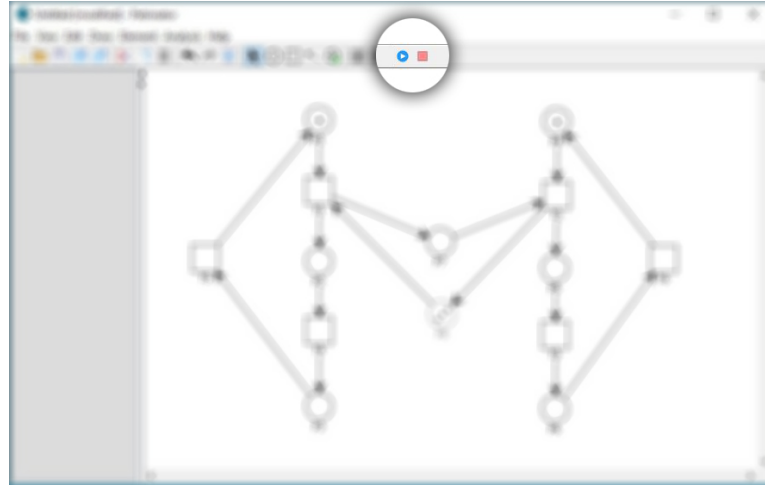


Fig. 25: Sim: botones

cada una de ellas.

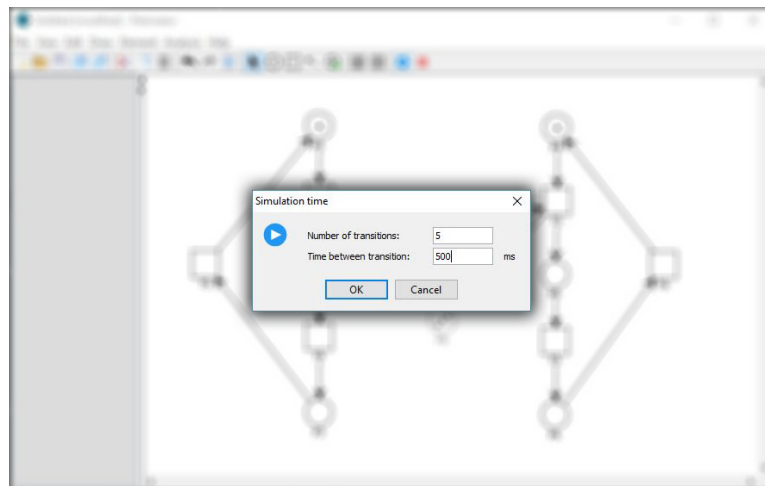


Fig. 26: Sim: ingreso de parámetros

Por último, para corroborar que el resultado de la simulación fue correcto, simplemente se analizaron las transiciones que fueron disparadas (indicadas en la columna izquierda del editor) para verificar que en efecto el disparo de esas transiciones produce el estado presente en el *worksheet* una vez finalizada la simulación. En este caso, los disparos realizados corresponden a las transiciones:

$$T_1 \rightarrow T_4 \rightarrow T_2 \rightarrow T_5 \rightarrow T_3 \quad (37)$$

El disparo de T_1 seguido de T_4 indica que el productor ha insertado un nuevo elemento en el *buffer*. Por este motivo puede dispararse a continuación T_2 con lo cual el consumidor ha extraído el elemento que el productor acaba de insertar. Luego, los disparos de T_5 y T_3 generan el marcado apreciado en la figura 27.

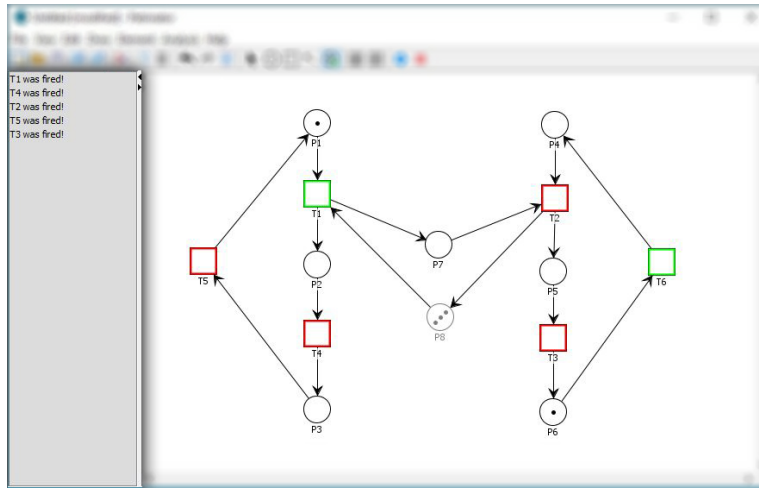


Fig. 27: Sim: resultado de simulación

Las transiciones resaltadas en verde son aquellas que se encuentran sensibilizadas para el marcado actual, mientras que aquellas en rojo no lo están. Recordar que el concepto de sensibilizado de una transición se encuentra detallado en la sección 2.2.1.

5.5 CONCLUSIONES

La implementación de esta funcionalidad no presentó grandes dificultades con lo cual el resultado de esta iteración no requiere en lo absoluto de modificaciones en el plan trazado en la sección 1.5. Por supuesto se mantiene abierta la posibilidad de encontrar *bugs* en las iteraciones posteriores, con lo cual no se considera la iteración como cerrada sino que éstos deberán ser corregidos a lo largo del proyecto. Por otro lado, los *testings* realizados en esta iteración también comprueban el correcto funcionamiento de una parte de la iteración desarrollada en el capítulo 4. El sistema cuenta ahora con las funcionalidades combinadas de las dos primeras iteraciones.

ITERACIÓN 3: CLASIFICACIÓN DE REDES DE PETRI

6.1 INTRODUCCIÓN

En este capítulo se explicarán las tareas realizadas para la implementación de las funcionalidades propuestas para la iteración 3. Como se planteó en el capítulo 1, se considera de gran importancia la clasificación en profundidad de una *red de Petri*. Para realizar esto se debieron seleccionar aquellas propiedades y características asociadas a las *redes de Petri* que se consideran necesarias para brindar una clasificación útil y detallada al usuario. Por este motivo se decidió realizar un análisis sobre varios conceptos, determinando si la red analizada cumple con los requisitos para ser:

- Una máquina de estados ó *state machine*.
- Un grafo marcado ó *marked graph*.
- Una red de libre elección ó *free choice net*.
- Una red simple ó *simple net*.
- Una red acotada ó *bounded*.
- Una red segura ó *safe*.
- Una red que presenta interbloqueo ó *deadklock*.

Por otro lado, se desean calcular algunas matrices asociadas a las *redes de Petri*, entre las cuales se encuentran:

- Matriz *post* ó I^+ .
- Matriz *pre* ó I^- .
- Matriz de incidencia.
- Matriz de inhibición.

6.2 OBJETIVOS

Entonces, según lo explicado en la introducción, esta iteración tiene como objetivos:

1. Determinar si la *red de Petri* presente en el *worksheet* cumple con las propiedades especificadas con anterioridad (máquina de estados, red de libre elección, etc).
2. Calcular las matrices asociadas, así como las transiciones sensibilizadas y el marcado actual.

6.3 DESARROLLO

Debido a que este es el primer bloque de análisis que se añadirá al proyecto (denominado algoritmos de clasificación para fines prácticos), se procederá a explicar la manera en que un nuevo algoritmo

es insertado en el mismo.

Los algoritmos se implementaron como *actions*, motivo por el cual los mismos se encuentran en el paquete *org.petrinator.editor.actions .algorithms* y heredan de la clase abstracta *AbstractAction*. Se observa a continuación la manera de implementar un algoritmo estándar:

```

1  package org.petrinator.editor.actions.algorithms;
2
3  // Imports
4
5  public class ClassificationAction extends AbstractAction
6  {
7      private Root root;
8      // Other fields
9
10     public ClassificationAction(Root root)
11     {
12         String name = "Net classification";
13         this.root = root;
14         putValue(NAME, name);
15         putValue(SHORT_DESCRIPTION, name);
16         putValue(SMALL_ICON, GraphicsTools.getIcon("classif.png"));
17     }
18
19     public void actionPerformed(ActionEvent e)
20     {
21         // Algorithm implementation
22     }
23 }

```

De esta manera, la acción *ClassificationAction* está vinculada a un botón con la leyenda "Net classification", el cual se añadió al menú superior del editor desde la clase *Root*:

```

1  JMenuBar menuBar = new JMenuBar();
2  mainFrame.setJMenuBar(menuBar);
3
4  JMenu algorithmsMenu = new JMenu("Analysis");
5  algorithmsMenu.setMnemonic('A');
6  menuBar.add(algorithmsMenu);
7
8  algorithmsMenu.add(new ClassificationAction(this));
9  algorithmsMenu.add(new GSPNAction(this));
10 algorithmsMenu.add(new InvariantAction(this));
11
12 // Rest of the desired algorithms

```

Por otro lado es importante mencionar que los algoritmos desarrollados en *PIPE* utilizan la clase *PetriNetView* para representar la red de Petri. Esta clase tiene un constructor que recibe como parámetro una variable de tipo *string* indicando el directorio en que se encuentra el archivo *pnml*, a partir del cual se desea crear el objeto que representará la red. Es aquí donde entra en juego la exportación desarrollada en el capítulo 4, ya que entre las primeras tareas que realiza el algoritmo se incluyen las de:

1. Exportar un archivo temporal con el dialecto requerido por *PIPE*.

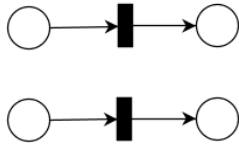


Fig. 28: (a)

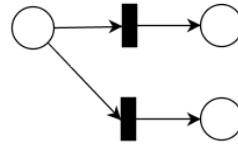


Fig. 28: (b)

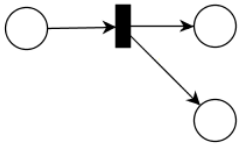


Fig. 28: (c)

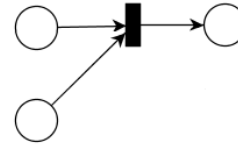


Fig. 28: (d)

2. Crear el objeto *PetriNetView* a partir del archivo temporal generado.

Una vez explicada la manera de agregar un nuevo algoritmo (o conjunto de algoritmos) al proyecto, se procederá a desarrollar la implementación de las funcionalidades propuestas. Cabe aclarar que la mayor parte de las funciones que se explicarán a continuación forman parte de la herramienta *PIPE* y solo fueron integradas y/o adaptadas al código del proyecto.

6.3.1 MÁQUINA DE ESTADOS

Una *red de Petri* es una máquina de estados si y solo si todas las transiciones que la componen tienen como máximo una entrada y una salida. Esto quiere decir que:

$$SM \text{ sii } \forall t \in T \Rightarrow |\bullet t| \leq 1 \wedge |t \bullet| \leq 1 \quad (38)$$

Este concepto puede ser comprendido observando las diferentes alternativas planteadas en la figura 28. Si se aprecian las mismas, se puede concluir que las subfiguras (a) y (b) representan efectivamente *redes de Petri* que cumplen con los requisitos necesarios para ser máquinas de estado, puesto que en ambos casos no existe ninguna transición que posea más de una entrada o más de una salida. Por el contrario, este no es el caso de las subfiguras (c) y (d), ya que la primera está compuesta por una transición que presenta dos salidas, mientras que en la última ocurre lo mismo, sólo que en este caso con dos entradas.

La implementación consta de una función llamada `stateMachine()` dentro de la clase `ClassificationAction`. La misma retorna un valor `booleano` y recibe como parámetro un objeto `PetriNetView`, el cual fue creado con el archivo `.pnml` generado a partir de la red.

El comportamiento de esta función se encuentra ilustrado en el diagrama de secuencia de la figura 29. Se comienza por obtener la cantidad de transiciones existentes en la red para luego recorrer cada una de ellas y realizar la verificación correspondiente. Para realizar esta verificación se llama a las funciones `countTransitionInputs` y `countTransitionOutputs`, las cuales devuelven un valor entero indicando la cantidad de arcos entrantes y la cantidad de arcos salientes a la transición pasada como parámetro respectivamente. Luego, sólo queda comprobar que ninguno de los valores retornados por estas funciones exceda el valor de uno.

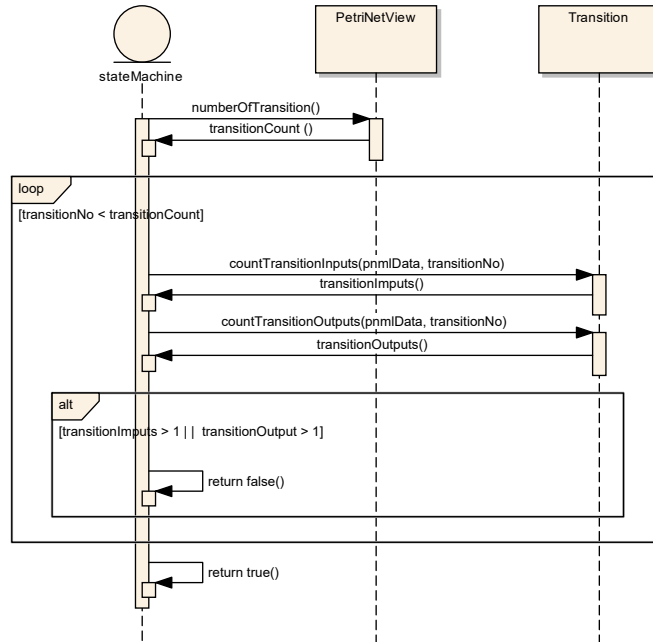


Fig. 29: Diagrama de secuencia de la función `stateMachine()`

6.3.2 GRAFO MARCADO

Una red de Petri es un grafo marcado si y solo si todas las plazas que la componen tienen como máximo una entrada y una salida. Por lo tanto esta condición puede expresarse como:

$$MG \text{ sii } \forall p \in P \Rightarrow |\bullet p| \leq 1 \wedge |p \bullet| \leq 1 \quad (39)$$

Si observamos el conjunto de figuras 30 se puede apreciar que las subfiguras (a) y (b) son en efecto grafos marcados, ya que en ambos casos no existen plazas con más de una entrada o más de una salida. Sin embargo, en el caso de las subfiguras (c) y (d), no se

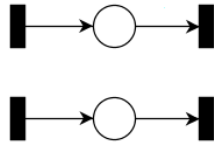


Fig. 30: (a)

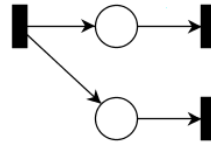


Fig. 30: (b)

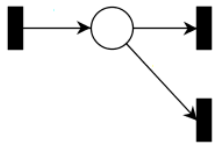


Fig. 30: (c)

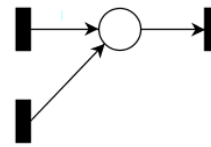


Fig. 30: (d)

cumplen los requisitos. La red de la figura (c) cuenta con una plaza con dos salidas, mientras que la de la figura (d) presenta una plaza con dos entradas.

La implementación para la detección de grafos marcados es análoga a la realizada para las máquinas de estado. La función que realiza esta tarea toma por nombre `markedGraph()` y al igual que la anterior, **retorna un valor booleano y recibe un objeto *PetriNetView*** como argumento. Por otra parte, en este caso se deseará obtener la cantidad de plazas presentes en la red en lugar de las transiciones, y la verificación se realizará sobre los valores retornados por los métodos `countPlaceInputs` y `countPlaceOutputs`, como se observa en la figura 31.

6.3.3 LIBRE ELECCIÓN

Una red de Petri es una red de libre elección (ó una *free choice net*) si y solo si ninguna transición recibe entradas de un par de plazas, a menos que ambas plazas tengan solamente una salida.

$$FCN \text{ sii } \forall p, p' \in P / p \neq p' \Rightarrow (p \bullet \cap p' \bullet = \emptyset \vee (|p \bullet| = |p' \bullet| \neq 1)) \quad (40)$$

El conjunto de figuras 32 aclara este concepto. En la subfigura (a) no existe ninguna transición que reciba entradas de más de una plaza, con lo cual la misma puede ser considerada como una *free choice net*. Por otro lado observamos que en la figura (b), la única transición que compone a la red tiene en efecto dos entradas, con lo cual no cumple la primer condición. Sin embargo, las dos plazas pertenecientes al conjunto de entrada de dicha transición poseen una y solo una salida, con lo cual se cumple la segunda condición y la red puede ser

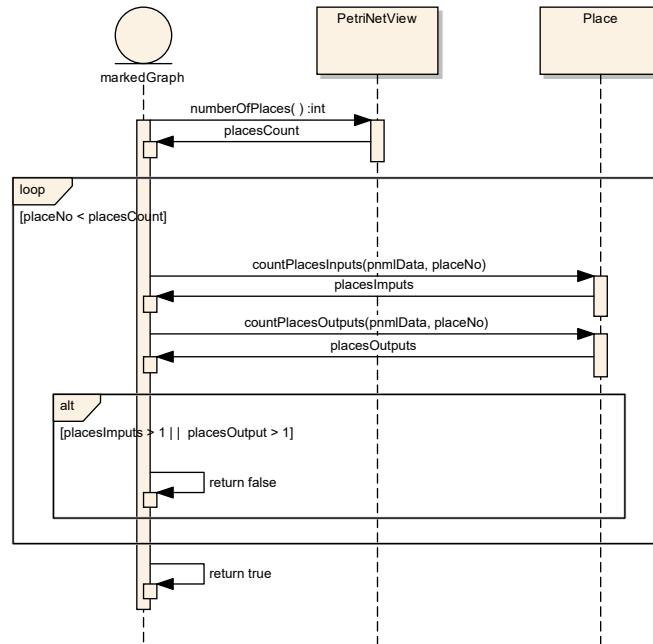
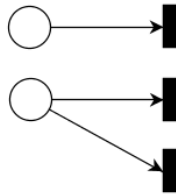
Fig. 31: Diagrama de secuencia de la función *markedGraph()*

Fig. 32: (a)

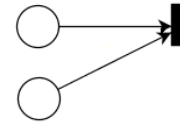


Fig. 32: (b)

considerada como una *free choice net*. Por último, las figuras (c) y (d) representan ejemplos de redes que **no** son de libre elección.

El análisis de la red para realizar esta verificación está contenido dentro de la función *freeChoiceNet()*, también incluida dentro de la clase *ClassificationAction*. El diagrama de la figura 33 ilustra el procedimiento para realizar dicha tarea. Se comienza por obtener la cantidad de plazas existentes en la red. Luego se recorren todas las plazas y se analiza cada una de ellas con respecto al resto, verificando que se cumplan las dos siguiente condiciones:

1. Hay intersección entre el conjunto de salida de una de las plazas con el conjunto de salida de la otra.
2. alguna de las plazas tiene más de una salida.

Si se cumplen ambas condiciones, la red **no** cuenta con los requisitos para ser una *free choice net*, con lo cual se retorna *false*. Si se termina de analizar todas las plazas y nunca se cumplieron estas condiciones, la función retorna *true* indicando que se trata de una *free choice net*.

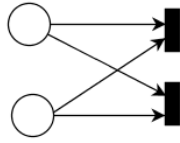


Fig. 32: (c)

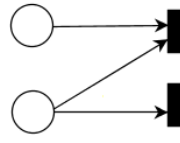
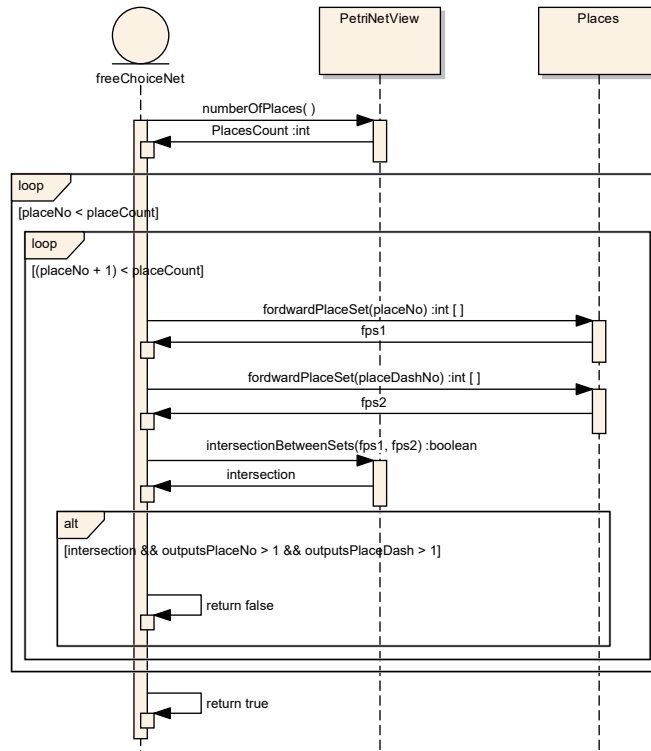


Fig. 32: (d)

Fig. 33: Diagrama de secuencia de la función *freeChoiceNet()*

6.3.4 LIBRE ELECCIÓN EXTENDIDA

Como el título lo infiere, este concepto está muy relacionado con el de libre elección. Una *red de Petri* es una *red de libre elección extendida* si y solo si ninguna transición recibe entradas de un par de plazas, a menos que **ambas** plazas tengan **exactamente** las mismas salidas.

$$EFCN \text{ sii } \forall p, p' \in P / p \neq p' \Rightarrow (p \bullet \cap p' \bullet = \emptyset \vee (p \bullet = p' \bullet)) \quad (41)$$

Es decir, en caso de haber dos plazas que ingresan a la misma transición, no es necesario que las mismas sólo contengan **una** salida como en el caso de *free choice*, sino que éstas pueden ser más de una siempre y cuando sean **las mismas**. Debido a esto, la subfigura 32 (c) que no cumplía con los requisitos para ser una *free choice net*, sí los cumple para ser una *extended free choice net*. Como corolario, se puede afirmar que todas las *free choice nets* son *extended free choice nets* pero

no a la inversa.

El procedimiento es exactamente igual que para el caso anterior, solo que la condición *if* para retornar *false* requerirá que se cumplan las dos siguientes condiciones:

1. Hay intersección entre el conjunto de salida de una de las plazas con el conjunto de salida de la otra.
2. Los conjuntos no son exactamente iguales.

6.3.5 RED SIMPLE

Una *red de Petri* es una red simple si y solo si ninguna transición recibe entradas de un par de plazas, a menos que alguna de las plazas tenga **solamente** una salida.

$$SPLN \text{ sii } \forall p, p' \in P / p \neq p' \Rightarrow \quad (42)$$

$$(p \bullet \cap p' \bullet = \emptyset \vee |p \bullet| \leq 1 \vee |p' \bullet| \leq 1) \quad (43)$$

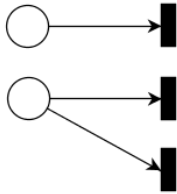


Fig. 34: (a)

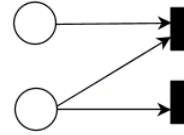


Fig. 34: (b)

Esta definición es muy similar a la de *free choice net*, aunque menos exigente. En vez de requerir que ambas plazas tengan solamente una salida, basta con que una la tenga para calificar como una *simple net*. Esto se ilustra en la figura 34, donde las subfiguras (a) y (b) son redes simples, ya que en (a) no hay transiciones con más de una entrada y en (b), aunque las hay, una de las plazas tiene solo una salida. Por el contrario, las subfiguras (c) y (d) ilustran redes donde ninguna de las plazas tiene sólo una salida, con lo cual no pueden considerarse como *simple nets*.

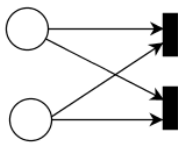


Fig. 34: (c)

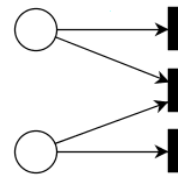


Fig. 34: (d)

Se representa el comportamiento del método *simpleNet()* en la figura 35, cuyo funcionamiento es totalmente análogo a las detalladas en el resto de este capítulo.

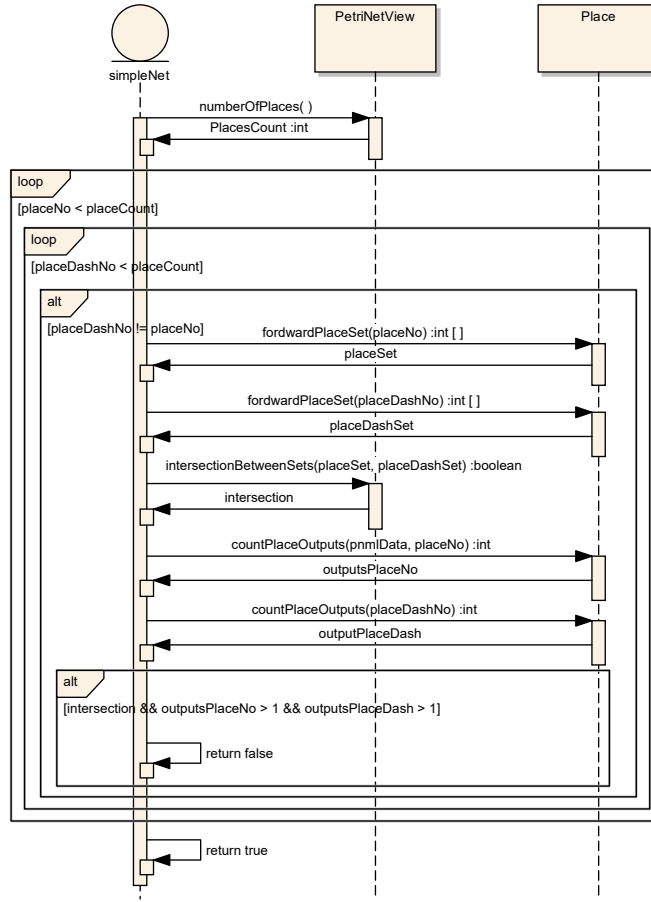


Fig. 35: Diagrama de secuencia de la función *simpleNet()*

6.3.6 RED SIMPLE EXTENDIDA

Una *red de Petri* es una red simple extendida si y solo si ninguna transición recibe entradas de un par de plazas, a menos que el conjunto de salida de una de las plazas esté contenido o sea igual al de la otra.

$$ESPLN \text{ sii } \forall p, p' \in P / p \neq p' \Rightarrow \quad (44)$$

$$(p \bullet \cap p' \bullet = \emptyset \vee p \bullet \subseteq p' \bullet \vee p' \bullet \subseteq p \bullet) \quad (45)$$

La implementación es muy similar a la realizada para detectar redes simples, sólo que se modifica la condición para devolver *true*.

6.3.7 SAFENESS

En la sección 2.3.1 se definió el concepto de *safeness*. Una red es segura o *safe* cuando todas sus plazas son 1-limitadas. Para realizar esta

verificación simplemente se recorren todas las plazas que componen la red y se comprueba que en ningún caso el marcado de las mismas sea mayor a uno. Esto se encuentra representado en el diagrama de secuencia de la figura 36.

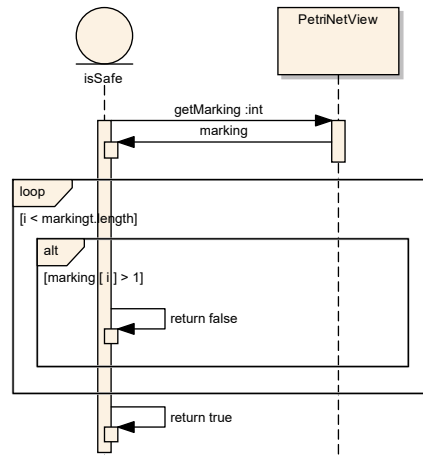


Fig. 36: Diagrama de secuencia de la función `isSafe()`

6.3.8 BOUNDEDNESS E INTERBLOQUEO

Los conceptos de *boundedness* e interbloqueo están asociados el uno con el otro puesto que ambos pueden obtenerse dentro del mismo algoritmo. El algoritmo para detectar si una red de Petri es acotada o no consiste en calcular su grafo de alcanzabilidad. Cuando la red es en efecto acotada, el algoritmo para la obtención del grafo converge. Sin embargo, éste no finaliza si la red es no acotada ó *unbounded*.

Calcular el grafo de alcanzabilidad consiste en obtener todos los marcados alcanzables por la red, lo cual se realiza partiendo de un nodo raíz, disparando todas las transiciones posibles y agregando los nuevos marcados como nodos del árbol. La función que se encarga de comenzar este proceso es `getTree()`. La misma crea un objeto `Node` a partir del marcado inicial de la red e invoca al método `analyzeTreeRecursively()` sobre el mismo. Esta última es una función recursiva que:

1. Obtiene las transiciones sensibilizadas para el marcado actual.
2. Dispara una por una.
3. Agrega los nuevos marcados obtenidos al árbol.
4. Realiza el mismo procedimiento para cada uno de los nuevos nodos encontrados.

Este procedimiento se ilustra en la figura 37. A grandes rasgos, se comienza obteniendo las transiciones sensibilizadas para luego ingresar en un ciclo *for* que recorrerá cada una de ellas. Al disparar una transición y calcular un nuevo estado, se agrega el mismo como un nuevo nodo del árbol. Una vez añadido se realizan algunas comprobaciones:

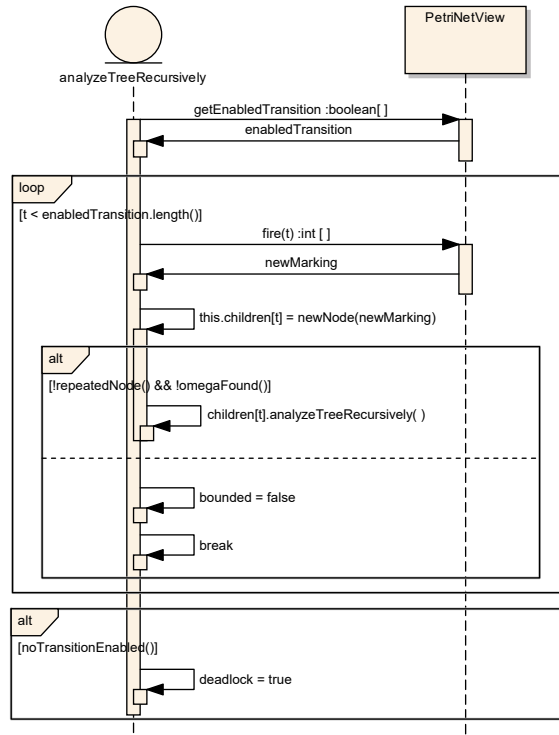


Fig. 37: Diagrama de secuencia de la función *anaylzeTreeRecursively()*

1. Por un lado se obtienen las transiciones sensibilizadas a partir del nuevo marcado. Si no hay ninguna transición sensibilizada, se valúa la variable *deadlock* en *true*, ya que esto significa que la red puede alcanzar un estado a partir del cual nunca podrá continuar; con lo cual la misma presenta interbloqueo.
2. Por otro lado se verifica si existen *omegas* (ω) en el árbol calculado hasta ese punto. Como se mencionó en la sección 2.3.4, un ω indica que existe un número muy grande o infinito de estados similares, pero para los cuales algunas de las plazas incrementan su cantidad de *tokens* de manera constante. La existencia de un ω indica que la red es *no acotada*.
3. Por último se verifica que el nuevo nodo no exista ya en el árbol (ya que es posible que el marcado sea alcanzable desde varios estados).

Si no se encontraron ω en el último marcado agregado al árbol y si el nodo no existía ya en el mismo, se llama recursivamente a la función *analyzeTreeRecursively()* sobre este marcado. De esta manera se calcula el grafo de alcanzabilidad, se verifica si una red es acotada y se determina la posibilidad de interbloqueo con el mismo algoritmo.

6.3.9 OBTENCIÓN DE MATRICES

La forma de calcular las matrices asociadas a una red fueron desarrolladas en la sección 2.1.2. A continuación se explicará la manera de obtener la matriz *post* ó I^+ , ya que la obtención del resto de las

matrices (exceptuando la de incidencia) es completamente análoga.

Como se puede apreciar en la figura 38, la función que se encarga de obtener la matriz es *forwardIncidenceMatrix()*. La misma comienza obteniendo de la *red de Petri* la cantidad de plazas y de transiciones existentes en la misma. A continuación crea un objeto *Matrix*, indicándole la cantidad de columnas (número de transiciones) y la cantidad de filas (número de plazas).

Una vez creada la matriz (con todos sus elementos iguales a cero), se obtienen todos los arcos de la red y se analizan iterativamente. Por cada arco se comprueba que el destino del mismo sea una plaza y que el origen sea una transición. En caso de cumplirse se modifica el valor de la matriz correspondiente llamando al método *set()* de la clase *Matrix*, el cual recibe como parámetros el *id* de la transición de origen, el *id* de la plaza destino, y el peso del arco que se analizó.

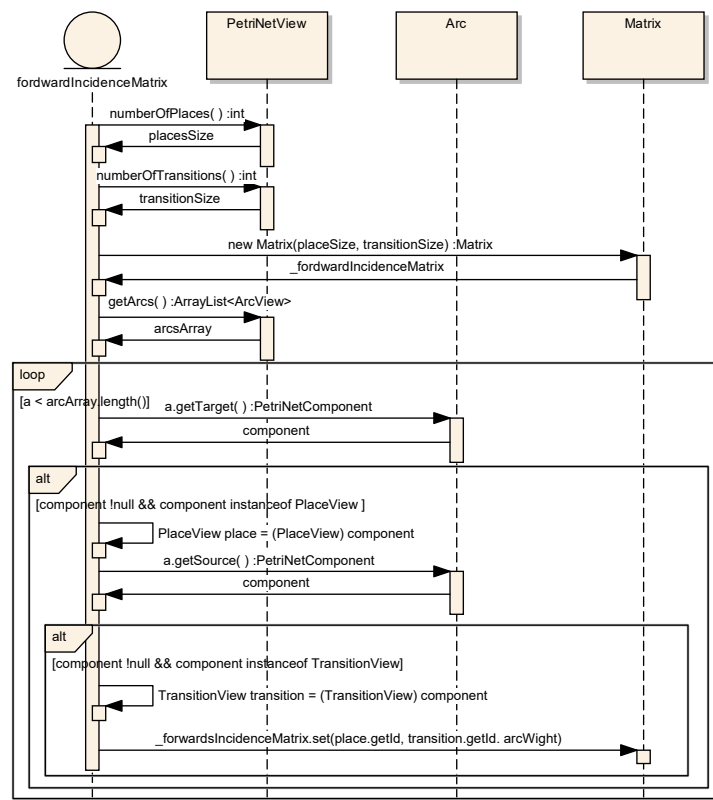


Fig. 38: Diagrama de secuencia de la función *forwardIncidenceMatrix()*

Por otro lado el procedimiento para obtener la matriz *pre* ó I^- es exactamente igual al recién mencionado, sólo que la comprobación en este caso será que el origen del arco sea una plaza y el destino una transición. Por último, el cálculo de la matriz de incidencia consiste en realizar la resta entre las dos matrices anteriores.

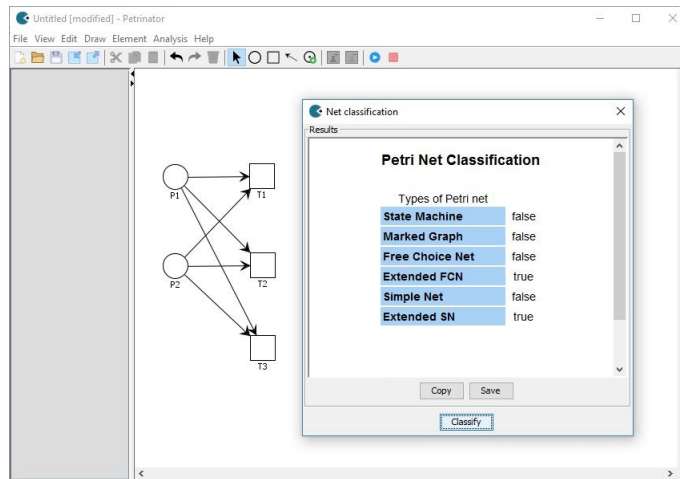


Fig. 39: Clasificación de una red

6.4 TESTING

Para realizar el *testing* de la clasificación se creó una red en el editor y se verificó que los resultados obtenidos tuvieran coherencia de acuerdo a la definición de las propiedades que se detallaron a lo largo de este capítulo.

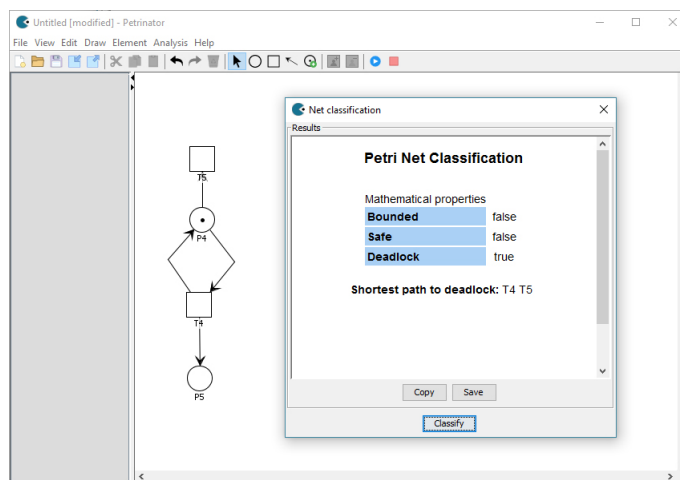
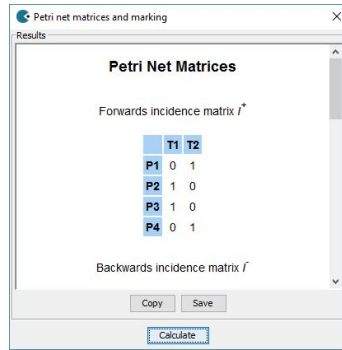


Fig. 40: Propiedades de una red

Se destaca que no sólo se puede observar que la red presenta interbloqueo sino que se especifica también cual es la secuencia de disparos de transiciones para alcanzar al mismo.

Por otra parte, para verificar el correcto funcionamiento de la funcionalidad que calcula las matrices se creó en el editor un ejemplo que coincide con aquél que fue planteado en los capítulos anteriores (figura 3). De esta manera se puede verificar simplemente que el resultado arrojado por el *software* coincide con aquél calculado en la



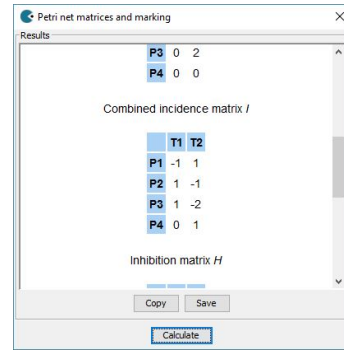
Petri Net Matrices

Forwards incidence matrix I^+

	T1	T2
P1	0	1
P2	1	0
P3	1	0
P4	0	1

Backwards incidence matrix I^-

Copy Save Calculate

Fig. 41: Matriz *post*


Petri net matrices and marking

Results

Combined incidence matrix I

	T1	T2
P3	0	2
P4	0	0

Inhibition matrix H

	T1	T2
P1	-1	1
P2	1	-1
P3	1	-2
P4	0	1

Copy Save Calculate

Fig. 42: Matriz de incidencia

sección 2.1.2. Las figuras 41 y 42 muestran algunos de los resultados obtenidos.

6.5 CONCLUSIONES

Nuevamente no se encontraron impedimentos a la hora de adaptar y/o integrar los algoritmos desarrollados en *PIPE* con el sistema. Sin embargo, cabe destacar que todo lo realizado en esta iteración para la detección de redes acotadas e interbloqueo servirá para la obtención del grafo de cobertura a tratar en el capítulo 9.

Por otro lado, visto y considerando que en el presente capítulo se desarrolló con detalle el proceso de integración e inclusión de un algoritmo al proyecto, los capítulos 7, 8 y 9 servirán exclusivamente para documentar los algoritmos implementados por *PIPE* (mencionando por supuesto las modificaciones que se hayan realizado para adaptar los mismos), evitando de esta forma repetir las tareas realizadas para añadir los algoritmos al proyecto.

ITERACIÓN 4: CÁLCULO DE SIFONES, TRAMPAS E INVARIANTES

7.1 INTRODUCCIÓN

En este capítulo se explicarán los algoritmos utilizados para la implementación de las funcionalidades propuestas en la iteración 4. Dentro del análisis que puede ser efectuado sobre una *red de Petri*, el cálculo de trampas, sifones e invariantes puede ser considerado como uno de los más importantes. Este tipo de análisis permite corroborar, a través de las ecuaciones provistas por estos algoritmos, si el comportamiento de la red es el deseado.

7.2 OBJETIVOS

De acuerdo a las tareas que se plantearon dentro del capítulo 1 y teniendo cuenta que no existen actividades pendientes que correspondan a iteraciones anteriores, los objetivos asociados a esta iteración son los siguientes:

1. Cálculo del conjunto de plazas que cumple con las condiciones para ser sifón.
2. Cálculo del conjunto de plazas que cumple con las condiciones para ser trampa.
3. Cálculo del conjunto de plazas que forman un *p-invariante*.
4. Cálculo del conjunto de transiciones que forman un *t-invariante*.
5. Obtención de las ecuaciones asociadas a los *p-invariantes*.

7.3 DESARROLLO

En la sección 6.3 se explicó la manera en que los algoritmos son integrados y añadidos al proyecto. Puesto que este procedimiento se repetirá para las iteraciones que siguen, sólo se abordarán los temas relacionados con los algoritmos en sí, pretendiendo documentar sus modos de operación y su funcionamiento.

7.3.1 INVARIANTES DE PLAZAS

El algoritmo utilizado para la obtención de las invariantes de plaza es conocido como *The Farkas Algorithm* y fue desarrollado por J. Farkas en 1902. Este algoritmo permite encontrar los vectores x que cumplan con la siguiente igualdad:

$$C \cdot x = 0 \quad (46)$$

donde C corresponde a la matriz de incidencia de la red analizada, cuya dimensión es $N \times M$.

Para realizar el procedimiento es necesaria además una matriz identidad con dimensión $N \times N$. Las operaciones que se realicen sobre la matriz C deben efectuarse de igual manera sobre la matriz identidad. Cabe aclarar que la matriz de incidencia de la red es llamada C para evitar confusiones entre la misma y la matriz identidad (representada por la letra I).

El algoritmo se basa en iteraciones. En cada una de ellas se busca anular una columna realizando operaciones elementales, generando así matrices intermedias hasta llegar al resultado final. Para ejemplificar el procedimiento, se considera la red representada en la figura 43.

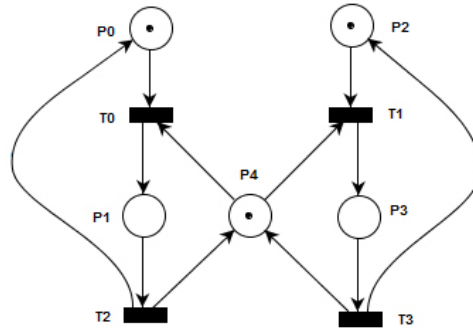


Fig. 43: Red de Petri productor y consumidor

La matriz de incidencia C con dimensiones 5×4 asociada a la red es la siguiente:

$$C = \begin{pmatrix} -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ -1 & -1 & 1 & 1 \end{pmatrix} \quad (47)$$

Como se mencionó anteriormente, es necesario realizar las mismas operaciones sobre las matrices C e I , por lo que se tendrá una nueva matriz, representada por $C | I$, conformada por ambas.

$$C | I = \left(\begin{array}{cccc|ccccc} -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \\ -1 & -1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array} \right) \quad (48)$$

Para la primera iteración, el análisis se enfoca sobre los valores de la primera columna de la matriz $C | I$. Se genera una nueva matriz con todas aquellas filas cuyo valor en la primera columna sea nulo. De esta forma, las primeras filas de la matriz intermedia MI_1 son:

$$\left(\begin{array}{cccc|ccccc} 0 & -1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \end{array} \right) \quad (49)$$

Para aquellos valores de la columna que no sean nulos, se debe realizar una combinación lineal entre las filas, siempre y cuando los mismos sean de signo opuesto. El vector resultante se inserta en la matriz correspondiente a la iteración (en este caso MI_1). Por lo tanto se deberá sumar las filas 1-2 y 2-5, obteniéndose la matriz MI_1 completa:

$$MI_1 = \left(\begin{array}{cccc|ccccc} 0 & -1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{array} \right) \quad (50)$$

Se continúa iterando sobre la siguiente columna de la matriz intermedia MI_1 , realizando el mismo procedimiento aplicado sobre la primera. Por lo tanto, la matriz resultante de la siguiente iteración es la siguiente:

$$MI_2 = \left(\begin{array}{cccc|ccccc} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array} \right) \quad (51)$$

De esta forma se logró que todas las columnas correspondientes a la matriz C sean nulas. Los invariantes de plazas son entonces el resultado de aplicar las mismas operaciones sobre la matriz identidad. Quedan definidos los siguientes vectores como p-invariantes:

$$p - \text{invariantes} = \left(\begin{array}{ccccc} P_0 & P_1 & P_2 & P_3 & P_4 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{array} \right) \quad (52)$$

Por otro lado, para calcular las ecuaciones asociadas a los p-invariantes, se utilizan los vectores obtenidos en la matriz 52, donde los componentes de la ecuación estarán dados por el marcado inicial de las plazas cuyo valor dentro del vector es igual a uno. Por lo tanto, las ecuaciones obtenidas serán:

$$\begin{aligned} M(P_0) + M(P_1) &= 1 \\ M(P_2) + M(P_3) &= 1 \\ M(P_1) + M(P_3) + M(P_4) &= 1 \end{aligned} \quad (53)$$

7.3.2 INVARIANTES DE TRANSICIÓN

Como se explicó en la sección 2.3.6, un invariante de transición indica cual es la secuencia de disparos para retornar al estado inicial. Esta secuencia se representa mediante un vector x , cuyos elementos toman el valor uno para aquellas transiciones que se disparan y un cero para las que no. De esta forma, son considerados t-invariantes todos los vectores que cumplen con la siguiente ecuación:

$$C^T \cdot x = 0 \quad (54)$$

donde C^T corresponde a la matriz de incidencia transpuesta asociada a la red de Petri. Por lo tanto, el algoritmo utilizado para calcular los t-invariantes es el mismo que el aplicado para las p-invariantes (*the Farkas algorithm*) con la diferencia que el proceso se realiza sobre la matriz de incidencia transpuesta.

Siguiendo con el ejemplo de la figura 43 planteado en la sección anterior. Se tiene que la matriz de incidencia transpuesta (C^T), cuya dimensión es de 4×5 , es la siguiente:

$$C^T = \begin{pmatrix} -1 & 1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 & -1 \\ 1 & -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & -1 & 1 \end{pmatrix} \quad (55)$$

Nuevamente, se denomina como C^T para evitar confusiones entre la misma y la matriz identidad. Puesto que las operaciones deben realizarse sobre ambas matrices, se obtiene lo siguiente:

$$C^T | I = \left(\begin{array}{ccccc|cccc} -1 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & -1 & 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 1 \end{array} \right) \quad (56)$$

Realizando el mismo procedimiento que en la sección anterior sobre la primera columna, se obtiene la matriz intermedia TI_1 como resultado de la primera iteración.

$$TI_1 = \left(\begin{array}{ccccc|cccc} 0 & 0 & -1 & 1 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{array} \right) \quad (57)$$

Para la siguiente, se enfoca sobre la segunda columna, efectuando las mismas operaciones y obteniendo la matriz TI_2 :

$$TI_2 = \left(\begin{array}{ccccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{array} \right) \quad (58)$$

Dicho resultado corresponde a la matriz final, por lo que las *t-invariantes* pueden ser representados de la siguiente manera:

$$t - invariantes = \begin{pmatrix} T0 & T1 & T2 & T3 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad (59)$$

7.3.3 SIFONES Y TRAMPAS

En la sección 2.3.5 se explicaron con detalle los requerimientos que un conjunto de plazas debe reunir para considerarse como un sifón o como una trampa. El algoritmo propuesto para la obtención de estos conjuntos consiste principalmente de tres pasos:

1. Generar todos los posibles conjuntos (dos o más plazas) a partir de las plazas existentes en la red. La cantidad de conjuntos en una red de n plazas es:

$$q = (2^n) - (n + 1) \quad (60)$$

La cantidad de iteraciones necesarias para la obtención del resultado es definitivamente el punto débil del algoritmo.

2. Iterar sobre cada conjunto generado, verificando si el mismo cumple con los requisitos para ser un sifón.
3. En caso de que el conjunto sobre el cual se itera no sea un sifón, comprobar si el mismo es una trampa.

La incógnita del algoritmo reside en como determinar si un subconjunto de plazas es un sifón o no. Recordemos que el requisito para considerarse a S como tal es que el subconjunto de transiciones entrantes a S esté contenido dentro del subconjunto de transiciones salientes. Para verificar esto se utilizan las matrices I^+ e I^- , ya que las mismas contienen la información sobre las transiciones entrantes y salientes de cada plaza.

La fila I^+_i contiene ceros en las posiciones correspondientes a las transiciones que **no** son entrantes a la plaza P_i y valores distintos de cero para aquellas transiciones que sí lo sean. Visto y considerando que sólo nos interesa la existencia de un arco entre cierta transición y la plaza P_i (y no el peso de mismo), es necesario reemplazar todos estos valores distintos de cero por unos, obteniendo un vector binario con información sobre las transiciones entrantes a la plaza P_i . Para obtener el subconjunto de transiciones entrantes a un **grupo** de plazas, simplemente se necesita realizar una operación *or* entre todos estos vectores (uno asociado a cada plaza). Se llamará al vector resultante de esta operación V_i .

El mismo procedimiento debe realizarse para calcular el vector asociado a las transiciones salientes de un conjunto de plazas (V_o), sólo

que en este caso la información la contiene la matriz I^- .

En este punto se tienen dos vectores: uno asociado a las transiciones entrantes del conjunto de plazas S y el otro a las transiciones salientes del mismo conjunto. Para verificar que el conjunto de entrada está contenido dentro del de salida simplemente se necesita realizar una operación *and* entre los vectores V_i y V_o y comprobar que el resultado sea exactamente igual que V_i . Esta condición es suficiente para demostrar que el conjunto de plazas es un sifón. Para el caso de las trampas sólo se deberá comparar el resultado con V_o .

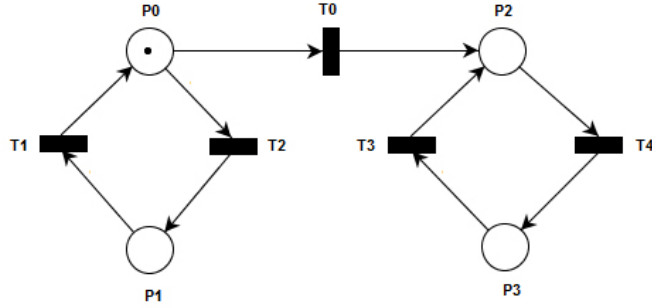


Fig. 44: Red de Petri con trampa y sifón

EJEMPLO Dentro de la red representada en la figura 44 podemos extraer el subconjunto de plazas $\{P_0, P_1\}$. Según lo explicado anteriormente, el vector de transiciones salientes de P_0 se obtiene a partir de la matriz I^- y es:

$$[1 \ 1 \ 1 \ 0 \ 0] \quad (61)$$

Por otro lado, aquel correspondiente a la plaza P_1 es:

$$[0 \ 1 \ 1 \ 0 \ 0] \quad (62)$$

Realizando una operación *or* entre ambos se obtiene V_o , vector asociado a las transiciones de salida del subconjunto de plazas $\{P_0, P_1\}$. El mismo será entonces:

$$V_o = [1 \ 1 \ 1 \ 0 \ 0] \quad (63)$$

De la misma manera se calcula el vector V_i :

$$V_i = [0 \ 1 \ 1 \ 0 \ 0] \quad (64)$$

De esta forma, si realizamos una operación *and* entre ambos, obtendremos un vector cuyos elementos coinciden completamente con aquellos de V_i , demostrando entonces que el subconjunto $\{P_0, P_1\}$ es efectivamente un sifón. Cabe aclarar que, en caso de no haberlo sido, se habría realizado la comparación con V_o para determinar si el mismo conjunto es o no una trampa. Este procedimiento debe realizarse por cada posible subconjunto de plazas en la red.

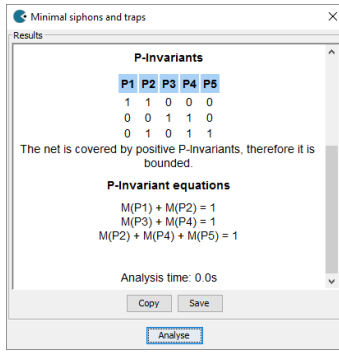


Fig. 45: P – invariantes

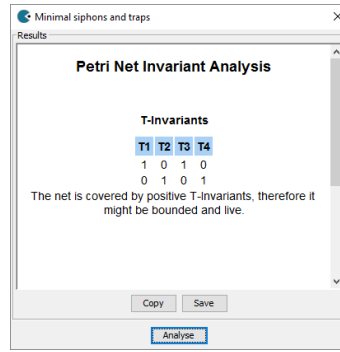


Fig. 46: T – invariantes

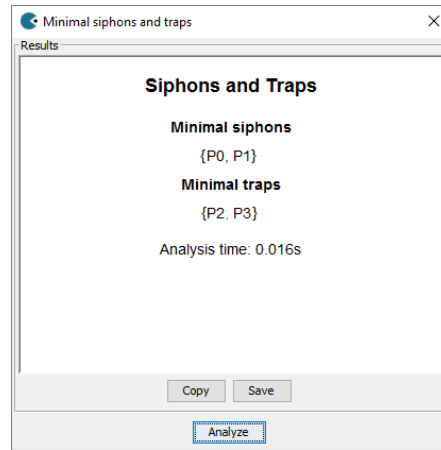


Fig. 47: Sifones y trampas de una red de Petri simple

7.4 TESTING

Para realizar el *testing* correspondiente a los invariantes de plaza y transición, se creó en el editor la red ilustrada en la figura 43 y se verificó que los resultados obtenidos tras aplicar dicho análisis sean los mismos que se adquirieron luego de realizar la descripción de estos algoritmos. Las figuras 45 y 46 muestran los resultados desde el editor, los cuales coinciden con las expresiones 52, 53 y 59.

De la misma manera, para efectuar el *testing* correspondiente a los sifones y trampas, se creó la red ilustrada en la figura 44, cuyos resultados se muestran en la figura 47. La mismos se compararon con los obtenidos en la sección 2.3.5, verificando que el resultado arrojado por el *software* coinciden con aquél calculado en la sección.

7.5 CONCLUSIONES

El proceso de adaptación de algoritmos entre *PIPE* y nuestro editor no presentó inconvenientes. Por este motivo se considera que el mayor desafío de este capítulo se basó en el hecho de entender el funcionamiento de los algoritmos utilizados, así como documentar los

mismos de una manera comprensible para el lector.

ITERACIÓN 5: ANÁLISIS DE REDES ESTOCÁSTICAS

8.1 INTRODUCCIÓN

En este capítulo se explicarán las tareas realizadas para la implementación de las funcionalidades propuestas para la iteración 5. Como se mencionó en el capítulo 2, las condiciones necesarias que una red debe cumplir para poder realizar este tipo de análisis son: que la misma sea acotada y que posea al menos una transición temporal.

El análisis de las *redes de Petri* estocásticas incluye un conjunto de conceptos que permiten definir algunas de las características de la red. Por lo tanto, haciendo énfasis en lo anteriormente mencionado, el análisis de una *general stochastic Petri net* se conforma por:

1. Estados tangibles de la red.
2. Distribución en estado estacionario de los estados tangibles.
3. Promedio de *tokens* sobre una plaza.
4. Densidad de probabilidad de *tokens* por plaza.
5. Rendimiento de las transiciones temporales.
6. Tiempo de permanencia sobre los estados tangibles.

8.2 OBJETIVOS

Entonces, según lo explicado en la introducción y de acuerdo a las tareas planteadas en un comienzo para esta iteración, la misma tiene como objetivos:

1. Obtener y mostrar en pantalla la siguiente información sobre la red presente en el editor:
 - Conjunto de estados tangibles.
 - Distribución en estado estacionario de los estados tangibles.
 - Promedio de *tokens* por plazas.
 - Densidad de probabilidad de los *tokens*.
 - Productividad de las transiciones temporales.
 - Tiempo de permanencia para los estados tangibles.
2. Exportar en un archivo *.html* la información obtenida.

8.3 DESARROLLO

Al igual que en las iteraciones anteriores solo se abordarán los temas relacionados con los algoritmos en sí, pretendiendo documentar sus modos de operación y su funcionamiento. El proceso de análisis se realizará en orden de acuerdo a los objetivos planteados.

8.3.1 ESTADOS TANGIBLES

Antes de explicar la implementación del algoritmo es necesario mencionar que **un estado es considerado tangible si el mismo no posee transiciones inmediatas sensibilizadas**.

El ejemplo de la figura 48 representa una red de Petri con un estado tangible y uno efímero. Por un lado $S_0 = (2, 0)$ y por otro $S_1 = (1, 1)$. Desde el estado S_0 se puede alcanzar S_1 si la transición T_1 se dispara, siendo esta temporal y la única sensibilizada, con lo cual el estado S_0 se considera tangible. Por otra parte, desde S_1 se puede alcanzar el estado S_0 por dos caminos, disparando la transición T_0 o T_1 . Puesto que ambas están sensibilizadas, siendo una temporal y la otra inmediata, este estado no forma parte del conjunto de estados tangibles.

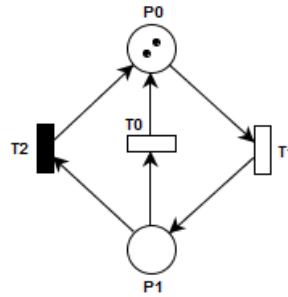


Fig. 48: Red de Petri de dos estados

En base a lo planteado anteriormente, el algoritmo utilizado opera revisando todos los estados de la red (ya sean tangibles o no) y seleccionando aquellos en donde las transiciones sensibilizadas sean sólo del tipo temporal.

8.3.2 DISTRIBUCIÓN DE LOS ESTADOS TANGIBLES

La distribución en estado estacionario de los estados tangibles está definida por el vector π , el cual se calcula a partir de la siguiente ecuación:

$$\pi \cdot Q = 0 \quad (65)$$

Para la obtención del vector π es necesario calcular previamente una matriz Q de dimensión $N \times N$, siendo N la cantidad de estados tangibles. El procedimiento para obtener Q parte del grafo de alcanzabilidad de la *red de Petri*, donde las columnas de la misma están asociadas al estado tangible y las filas reflejan la interacción con el resto de los estados.

Sobre la diagonal principal de la matriz Q se tiene la suma de los *rates* (con signo negativo) de las transiciones que deben dispararse para llegar al estado en cuestión. Por otro lado, los *rates* de aquellas transiciones que deben dispararse para alcanzar otros estados

partiendo del analizado, serán colocados en la posición asociada al estado alcanzado.

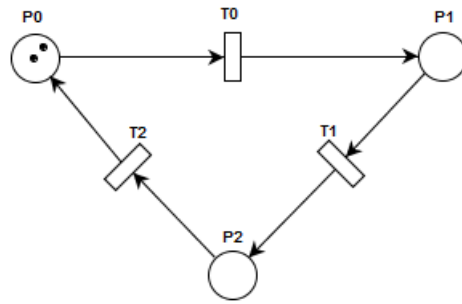


Fig. 49: Red de Petri estocástica simple

EJEMPLO Para una mejor comprensión se realiza el análisis en base al siguiente ejemplo: considerando la red de Petri estocástica ilustrada en la figura 49 y suponiendo que las tasas de disparo promedio son $\lambda_1 = 1$, $\lambda_2 = 1$ y $\lambda_3 = 1$, se genera el grafo de alcanzabilidad mostrado en la figura 50. Como se puede observar, el mismo contiene seis estados tangibles y un total de nueve transiciones. Por este motivo, la matriz Q tiene una dimensión de 6x6.

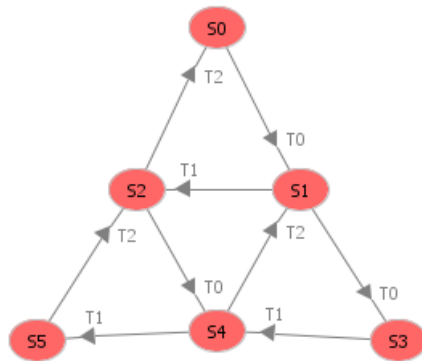


Fig. 50: Grafo de alcanzabilidad de una red de Petri estocástica

Partiendo del estado S_0 se puede generar la primer columna de la matriz según el procedimiento desarrollado a continuación para luego obtener de forma iterativa el resto de las mismas. Los marcados

asociados a cada uno de los estados tangibles pueden apreciarse en la matriz 66.

$$\begin{pmatrix} & P0 & P1 & P2 \\ S0 & 2 & 0 & 0 \\ S1 & 1 & 1 & 0 \\ S2 & 1 & 0 & 1 \\ S3 & 0 & 2 & 0 \\ S4 & 0 & 1 & 1 \\ S5 & 0 & 0 & 2 \end{pmatrix} \quad (66)$$

Para comenzar, se analizan los estados a través de los cuales se puede alcanzar aquel que se analiza; es decir, todos los estados entrantes a S_0 . Por lo tanto, para este caso sólo se tiene un estado entrante, el cual corresponde a S_2 por medio del disparo de la transición T_2 . De esta manera se coloca en la posición 2 de la primer columna el *rate* asociado a esta transición.

A continuación deben analizarse los estados alcanzables desde S_0 . En este caso, sólo si T_0 se dispara se obtiene un nuevo estado, con lo cual en la posición 0 de la primera columna se coloca el *rate* asociado a dicha transición (con signo negativo). Para el caso en que la cantidad de flechas salientes sea mayor a uno, se debe colocar en dicha posición (numero del estado de origen de las flechas) la sumatoria de todos los *rates* correspondientes a las transiciones que intervienen. De acuerdo a lo anterior, la primera columna de la matriz queda definida de la siguiente forma:

$$\begin{pmatrix} -1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (67)$$

Siguiendo este procedimiento con los demás estados se obtiene la matriz Q , la cual queda representada de la siguiente manera:

$$Q = \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -2 & 1 & 1 & 0 & 0 \\ 1 & 0 & -2 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & 0 & -2 & 1 \\ 0 & 0 & 1 & 0 & 0 & -1 \end{pmatrix} \quad (68)$$

Luego de esto, la distribución de los estados tangibles π se obtiene aplicando un método iterativo de resolución de ecuaciones lineales

(Gauss-Seidel¹) sobre la ecuación 65, en donde debe satisfacerse la siguiente expresión:

$$\sum_{i=1}^s \pi_i = 1 \quad (69)$$

Luego de aplicar este método iterativo, la distribución de estados tangibles para este ejemplo es:

$$\pi = [0.166, 0.166, 0.166, 0.166, 0.166, 0.166] \quad (70)$$

donde cada elemento del arreglo corresponde a un estado alcanzable de la red. El vector π será de suma importancia para el desarrollo los algoritmos que se detallarán en el resto del capítulo.

8.3.3 DENSIDAD DE PROBABILIDAD DE TOKENS

La matriz de densidad de probabilidad de *tokens* se compone por elementos enteros que representan la probabilidad de que una determinada plaza contenga una cierta cantidad de *tokens*. Por este motivo, la matriz tendrá tantas filas como plazas existan en la red analizada y un número de columnas n igual a la máxima cantidad de *tokens* que una plaza pueda adquirir.

Lo primero que realiza el algoritmo que calcula esta matriz es recorrer todos los estados alcanzables para determinar cuál es el máximo número de *tokens* que una plaza (de cualquier estado) puede adquirir. Este valor sumado en uno determina la cantidad de columnas de la matriz de salida. Es decir, la primera columna corresponde a cero *tokens*, la segunda a un *token* y la última a n .

A continuación se calcula la probabilidad para cada una de las plazas y para cada valor de *tokens* entre cero y el máximo obtenido. Para obtener la probabilidad de que una plaza P_i contenga una cantidad μ de *tokens*, el algoritmo realiza la suma de los elementos del vector π cuyas posiciones corresponden a los estados donde la plaza P_i contiene μ *tokens*.

EJEMPLO Siguiendo con el ejemplo de la figura 50, se pretende calcular la probabilidad de que la plaza P_o contenga 1 *token*. Como los únicos estados para los cuales la plaza P_o tiene un marcado de uno son S_1 y S_2 (esto puede observarse en la matriz 66), esta probabilidad se calcula de la siguiente manera:

$$\begin{aligned} P(P_o | \mu = 1) &= \pi[1] + \pi[2] \\ P(P_o | \mu = 1) &= 0.166 + 0.166 = 0.332 \end{aligned} \quad (71)$$

El cálculo del resto de las probabilidades es análogo al anterior, con lo cual la matriz se completará repitiendo este procedimiento para cada una de las plazas.

¹ Steven C. Chapra, Raymond P. Canale (2007). "Numerical Methods for Engineers", 310-318.

8.3.4 PROMEDIO DE TOKENS POR PLAZA

El procedimiento para obtener la cantidad de *tokens* promedio en una plaza es muy similar al realizado para calcular la densidad de probabilidad. El diagrama de secuencia del algoritmo puede apreciarse en la figura 51.

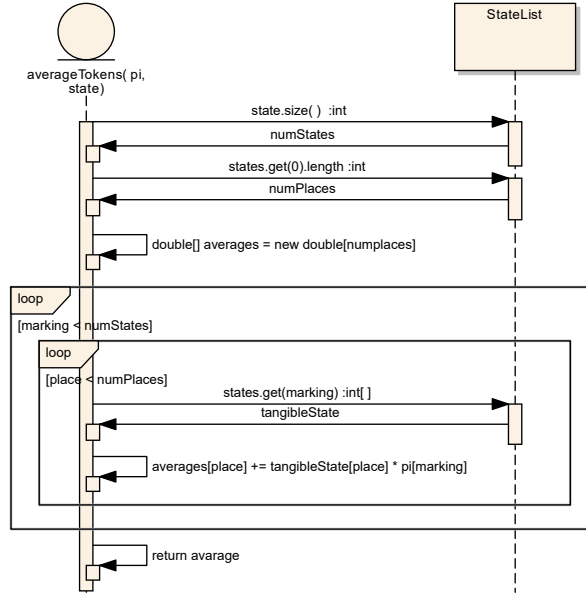


Fig. 51: Diagrama de secuencia de la función *averageTokens()*

Para comenzar, se tiene un arreglo vacío (todos sus elementos inicializados en cero) de tamaño igual al número de plazas existentes en la red. Se itera por cada estado tangible encontrado, sumando a los elementos del arreglo de salida el producto entre el marcado de las plazas para el estado en el cual se itera y el valor correspondiente del vector π . Si se quiere obtener el promedio de *tokens* para la plaza P_i , el resultado será:

$$\sum_{j=0}^n P_i(S_j) \cdot \pi_j \quad (72)$$

Donde n es la cantidad de estados y $P_i(S_j)$ es el marcado de la plaza P_i para el estado S_j .

EJEMPLO Continuando con el ejemplo de la figura 50 y calculando el promedio de *tokens* en la plaza P_o , se puede aplicar la propiedad distributiva para obtener la siguiente expresión:

$$\begin{aligned} & [P_o(S_0) \cdot \pi_0 + P_o(S_1) \cdot \pi_1 + \dots + P_o(S_5) \cdot \pi_5] \\ & [2 + 1 + 1 + 0 + 0 + 0] \cdot 0.166 = 0.664 \end{aligned} \quad (73)$$

8.3.5 PRODUCTIVIDAD DE LAS TRANSICIONES

El vector de productividad es un arreglo que contiene tantos elementos como transiciones hay en la red. El algoritmo para su obten-

ción es similar a los desarrollados en el resto de esta sección. Nuevamente se itera sobre todos los estados tangibles encontrados, sólo que en este caso el algoritmo realiza un análisis sobre el vector de transiciones sensibilizadas para cada uno de los mismos.

Para cada estado tangible, el procedimiento consiste en multiplicar todos los elementos del vector de transiciones sensibilizadas con aquellos del arreglo de *rates*. Esta operación produce un nuevo vector que contiene el *rate* de las transiciones sensibilizadas en las posiciones correspondientes y cero para aquellas que no lo están. Además, se multiplica cada valor del vector calculado por el elemento del arreglo π que corresponde al estado tangible sobre el cual se itera. De esta manera, la productividad de la transición T_i será:

$$\sum_{j=0}^n T_i(S_j) \cdot \pi_j \cdot \lambda_i \quad (74)$$

donde $T_i(S_j)$ es el elemento i del vector de transiciones sensibilizadas para el estado S_j (el cual toma el valor de uno o cero), λ_i es el *rate* de la transición T_i y n es la cantidad de estados tangibles.

EJEMPLO Para continuar con el ejemplo de la figura 50, se considera el procedimiento para calcular la productividad de la transición T_0 . El cálculo será:

$$\begin{aligned} & [T_0(S_0) \cdot \pi_0 + T_0(S_1) \cdot \pi_1 + \dots + T_0(S_5) \cdot \pi_5] \cdot \lambda_0 \\ & [1.0, 166 + 1.0, 166 + 1.0, 166 + 0.0, 166 + 0.0, 166 + 0.0, 166] \cdot 1 = 0.5 \end{aligned} \quad (75)$$

8.3.6 TIEMPO DE PERMANENCIA PARA ESTADOS TANGIBLES

En la sección anterior se mencionó que el vector correspondiente a la productividad de las transiciones es un arreglo que contiene tantos elementos como transiciones existen. Para el caso del tiempo de permanencia en estados tangibles, la cantidad de elementos asociados al vector está dada por la cantidad de estados tangibles de la *red de Petri*.

El algoritmo implementado para el cálculo de estos valores utiliza los estados tangibles de la red para obtener un vector de transiciones sensibilizadas T_j por cada estado. Además, se requiere un vector λ con los *rates* asociado a las transiciones. Para obtener el tiempo de permanencia del estado tangible M_j se debe realizar la inversa del producto escalar entre T_j y λ , como se observa en la ecuación 76.

$$M_j = \frac{1}{T_j \cdot \lambda} \quad (76)$$

EJEMPLO Para el caso del estado M_0 del ejemplo de la figura 49, los vectores T_0 y λ quedan definidos de la siguiente manera:

$$\overline{T_0} = [1, 0, 0] \quad \overline{\lambda} = [1, 1, 1] \quad (77)$$

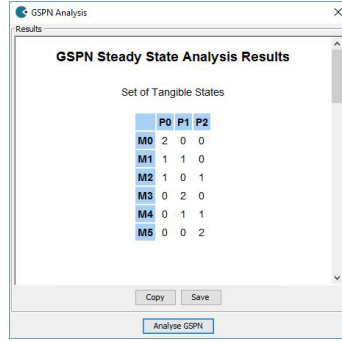


Fig. 52: (a)

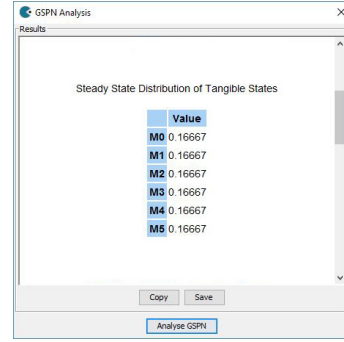


Fig. 52: (b)

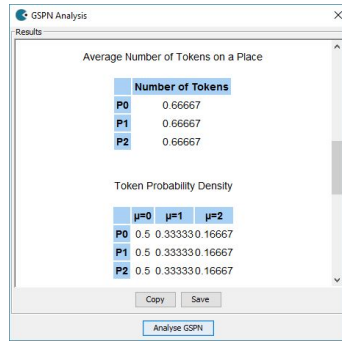


Fig. 52: (c)

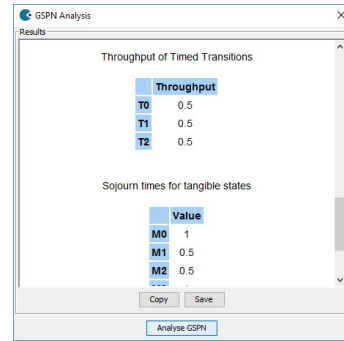


Fig. 52: (d)

Al aplicar el producto escalar entre ambos se obtiene el siguiente resultado:

$$\bar{T}_j \perp \bar{\lambda} = 1 \quad (78)$$

Por lo tanto, la inversa de este valor representa el *sojourn time* del estado M_0 . El mismo proceso es realizado para obtener el resto de los elementos del vector de salida.

8.4 TESTING

Para verificar el correcto funcionamiento de los algoritmos planteados en este capítulo se creó la red propuesta en la figura 49 en el editor. Se ejecutó la funcionalidad de análisis GSPN para comprobar que los resultados obtenidos se corresponden con aquellos calculados en los ejemplos planteados en las secciones anteriores.

Los resultados pueden observarse en el grupo de figuras 52, donde: el conjunto de estados tangibles se encuentra representado en la subfigura (a); la distribución de los mismos en la subfigura (b); el promedio y la densidad de probabilidad de *tokens* por plaza en la subfigura (c); y la productividad de transiciones y tiempo de permanencia de estados tangibles en la (d).

8.5 CONCLUSIONES

El desarrollo de la iteración no presentó dificultades y se lograron incorporar al editor las funcionalidades propuestas en los objetivos

de este capítulo. Hasta este punto, el plan de iteraciones trazado en el capítulo 1 fue efectuado sin desvíos ni retrasos, con lo cual no es necesaria una actualización del mismo.

ITERACIÓN 6: CÁLCULO DE ÁRBOLES DE ALCANZABILIDAD Y COBERTURA

9.1 INTRODUCCIÓN

En este capítulo se explicará el procedimiento realizado para la creación del grafo, ya sea de cobertura o de alcanzabilidad, propuesto para la iteración 6. Como se mencionó anteriormente, en el caso de que la red sea acotada, ambos grafos son exactamente iguales, con lo cual el cálculo del de alcanzabilidad es suficiente. Sin embargo, un algoritmo estándar no converge para el caso de una red no acotada, con lo cual es necesario realizar ciertas comprobaciones a medida que se itera sobre los estados alcanzables.

9.2 OBJETIVOS

El único objetivo planteado para esta iteración consiste en adaptar el algoritmo que calcula y gráfica el grafo asociado a los estados de una red con el editor desarrollado.

9.3 DESARROLLO

Las tareas realizadas para alcanzar el objetivo son bastante similares a las desarrolladas para otras iteraciones. Las mismas consisten principalmente en exportar la red presente en el editor al formato *.pnml* para luego crear un objeto de la clase *PetriNetView* a partir del archivo temporal generado. Luego se aplica el algoritmo desarrollado por *PIPE* sobre el objeto creado.

El algoritmo utilizado por *PIPE* fue desarrollado por el Dr. William Knottenbelt en su tesis de master titulada *Generalised Markovian Analysis of Timed Transition Systems*¹ en 1996. Este algoritmo procesa la información brindada por la función *myTree()* y elimina los estados efímeros en caso de que la red no sea acotada.

En la sección 6.3.8 se explicó brevemente el funcionamiento de la función *myTree()*, la cual obtiene todos los estados alcanzables en forma de árbol y genera los ω (omegas) presentes en la red, determinando también de esta manera si la red es o no acotada. Asimismo, la representación gráfica del árbol se lleva a cabo haciendo uso de la

¹ William J. Knottenbelt (1996). "Generalised markovian analysis of timed transitions systems".

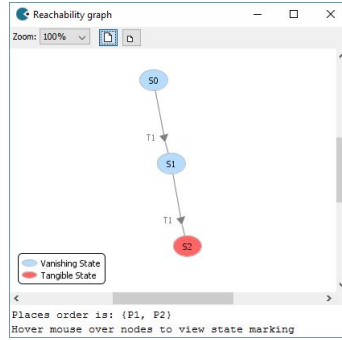


Fig. 53: Grafo de alcanzabilidad

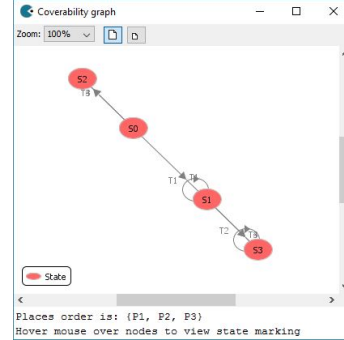


Fig. 54: Grafo de cobertura

librería *JPowerGraph*².

9.4 TESTING

Para verificar el correcto funcionamiento de esta funcionalidad se decidió obtener los grafos asociados a las redes de las figuras 6 y 8, ambas presentadas como ejemplos en el capítulo 2 y una de las cuales representa una red acotada mientras que la otra, por el contrario, una no acotada. De esta forma se puede comprobar que los resultados calculados coinciden con aquellos obtenidos empíricamente en las figuras 7 y 9 respectivamente.

Los resultados pueden ser apreciados en las figuras 53 y 54. La primera imagen representa el grafo de alcanzabilidad correspondiente a la red de la figura 6. Se aclara que el mismo es un grafo de **alcanzabilidad** puesto que la red es acotada. Esto puede ser comprobado al colocar el *mouse* sobre cada uno de los estados, lo que permite ver el marcado de los mismos.

Por otro lado, la segunda imagen muestra el grafo de la red de la figura 8 y el mismo es de **cobertura** debido a que la red en cuestión no es acotada. Esto puede ser observado en la ventana de *display* del grafo cuando se coloca el *mouse* sobre el estado S1 o S3, ya que el marcado de ambos contendrá un ω (*omega*).

9.5 CONCLUSIONES

Con la implementación del cálculo de grafos de alcanzabilidad y/o cobertura se concluye la parte del proyecto integrador que tiene como objetivo principal el análisis de las redes, puesto que las iteraciones consecuentes requieren de cambios en el monitor para permitir la simulación de redes estocásticas. Por este motivo, se puede concluir con seguridad que la incorporación de los algoritmos de análisis al editor no ha presentado inconvenientes mayores que pudieran poner

² Mike Kerrigam, *JPowerGraph*, (2013) [Online]. Disponible: <https://sourceforge.net/projects/jpowergraph/>.

en peligro los planes (de viabilidad o de tiempo) trazados al inicio de proyecto.

ITERACIÓN 7: DISPARO DE TRANSICIONES ESTOCÁSTICAS

10.1 INTRODUCCIÓN

En el presente capítulo se explicarán las tareas realizadas para la simulación de redes estocásticas. De acuerdo a los requerimientos planteados en el capítulo 1, es necesario que cada transición temporal tenga asociada una distribución probabilística que permita calcular el tiempo de disparo al realizar la simulación. Las distribuciones seleccionadas son:

1. Exponencial
2. Normal
3. Uniforme

Por otro lado se realizarán ciertas modificaciones en el monitor para que el mismo dispare transiciones estocásticas. Los tiempos de disparo para este tipo de transiciones serán muestras generadas de acuerdo a la distribución de probabilidad seleccionada para cada una de las transiciones. Además, dado que las transiciones estocásticas adquieren nuevas propiedades, será necesario modificar tanto los archivos de importación como los de exportación para preservar las mismas y mantener la coherencia en la comunicación con el monitor.

10.2 OBJETIVOS

De acuerdo a lo mencionado en la introducción del presente capítulo y a las tareas planteadas para la iteración 7 en la sección 1.5, se definen los siguientes objetivos:

1. Realizar modificaciones en los procedimientos de exportación e importación para incorporar a los archivos las nuevas propiedades asociadas a las transiciones.
2. Realizar las modificaciones necesarias en la función principal del monitor para permitir el disparo de transiciones estocásticas.
3. Calcular el tiempo de disparo de una transición estocástica a partir de una muestra obtenida de la función de distribución de probabilidad seleccionada por el usuario.

10.3 DESARROLLO

10.3.1 EXPORTACIÓN DE ARCHIVOS

Como se mencionó en la introducción del presente capítulo, cada transición temporal posee dos nuevas propiedades que definen la distribución asociada:

1. Distribución probabilística: Indica una de las cuatro disponibles.
2. Valores asociados: Almacena los valores característicos de la distribución para una transición determinada. Por ejemplo, para el caso de una distribución normal, estos valores corresponden a la media (μ) y a la varianza (σ^2).

De acuerdo con lo mencionado en la sección 4.3, para insertar un nuevo *item* en el archivo a exportar es necesario que la clase *documentExporter* consulte a cada transición obteniendo las propiedades requeridas para luego agregarlas al procesador *.xslt*, el cual es responsable de generar el archivo de salida.

El resultado de agregar dichas propiedades pueden apreciarse en el siguiente fragmento del archivo exportado:

```

1  <distribution>
2    <text>Normal</text> Tipo de distribucion
3  </distribution>
4  <delay>
5    <values>
6      <cn>4.0</cn>
7      <cn>1.0</cn>
8    </values>
9  </delay>
```

u = mediasigma = desviacionEntonces primero va el valor de la me

En este caso la transición posee una distribución normal con $\mu = 4$ y $\sigma^2 = 1$.

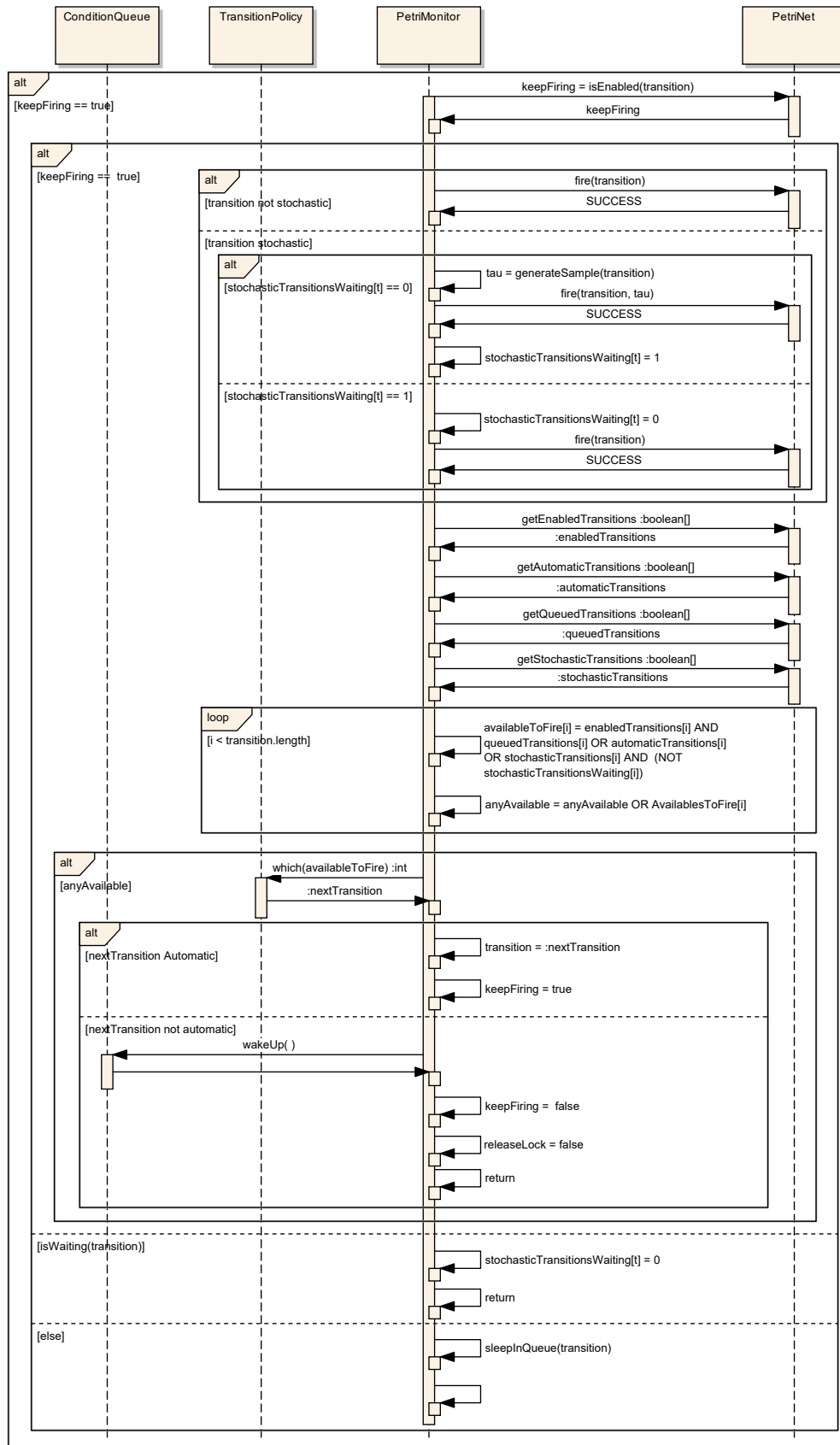
10.3.2 DISPARO DE TRANSICIONES ESTOCÁSTICAS

Se comenzó por modificar la función *internalFireTransition()*¹ del *monitor* para que la misma responda al comportamiento representado en el diagrama de secuencia de la figura 55.

Una vez que el hilo obtuvo el *mutex* del *monitor*, la variable *keep-Firing* comienza valuada en *true*. Se comprueba si la transición que el hilo desea disparar se encuentra sensibilizada. En caso de estarlo, se procede a realizar el disparo. En este punto pueden darse tres casos, aunque se comenzará por analizar los primeros dos:

1. La transición a disparar no es estocástica: En este caso simplemente se realiza el disparo de la *red de Petri* sin realizar modificaciones en el comportamiento de la función original.

¹ Ariel Rabinovich, Juan Arce (2017). "Framework de sincronización de tareas, coordinado por *redes de Petri*", 89–93.

Fig. 55: Diagrama de secuencia de la función *internalFireTransition()*

2. La transición a disparar es estocástica y la misma no se encuentra esperando. Esto último indica que es la primera vez que esta transición se sensibiliza y que el tiempo asociado a la misma no está transcurriendo. Por lo tanto, se genera una muestra de la función de probabilidad asociada a la transición y se invoca el método *fire(transition, τ)*, donde τ es la muestra generada. Además, se modifica el vector *stochasticTransitionsWaiting* para informar que el tiempo asociado al disparo de esta transición está transcurriendo y que la misma no debe considerarse como una transición disponible hasta que el mismo finalice.

La función *fire(transition, τ)* realiza dos tareas:

- Crear un hilo que se duerme a sí mismo durante el tiempo especificado por el parámetro τ .
- Retornar el valor *SUCCESS* para que el monitor pueda continuar con la ejecución de otras transiciones sensibilizadas.

De esta manera, se ha lanzado la ejecución de la transición (el tiempo asociado a la misma se encuentra transcurriendo). A continuación, el monitor genera el vector de transiciones disponibles. El mismo se obtiene a partir del siguiente conjunto de vectores:

- *enabledTransitions*: Transiciones que se encuentran sensibilizadas.
- *queuedTransitions*: Transiciones no automáticas para las cuales hay al menos un hilo esperando para dispararlas.
- *automaticTransitions*: Transiciones automáticas.
- *stochasticTransitions*: Transiciones estocásticas (las cuales son automáticas por definición).
- *stochasticTransitionsWaiting*: Transiciones estocásticas cuyo tiempo se encuentra transcurriendo. Este vector debe negarse, ya que para determinar las transiciones disponibles, solo es importante conocer las transiciones estocásticas que **no** están esperando.

De esta manera, realizando operaciones lógicas entre los vectores, se obtiene un nuevo arreglo con las transiciones disponibles. A continuación, se consulta a la política cual de estas transiciones es la próxima a disparar.

Si la transición retornada por la política es una transición automática (ya sea estocástica o no), simplemente se asigna el objeto asociado a la misma en la variable *transition* y se valúa *keepFiring* en *true*. Por el contrario, si la misma no es automática, se despierta al hilo que espera la sensibilización de la transición en cuestión y se abandona el monitor.

Se mencionó al comienzo de esta sección que, una vez comprobado que la transición a disparar se encuentra sensibilizada, pueden darse

tres casos, dos de los cuales ya se detallaron en este mismo apartado. El tercer caso implica que la transición es estocástica y que la misma se encuentra esperando para ser disparada (su tiempo está transcurriendo). Este caso sólo se dará cuando el hilo en cuestión sea aquel creado por la función *fire(transition, τ)*, y el hecho de que el mismo se encuentre activo implica que el tiempo asociado a la transición ya ha transcurrido. En caso de que la transición se encuentre aún sensibilizada, se procede a disparar la misma y a modificar el vector *stochasticTransitionsWaiting* para informar que esta transición ya ha sido disparada y que no hay ningún hilo dormido que intentará dispararla.

Por otro lado, si una transición estocástica no se encuentra sensibilizada una vez transcurrido el tiempo (la misma ha dejado de estar sensibilizada entre el inicio y la finalización del período), la misma no puede dispararse. Ante esta situación, se modifica el vector *stochasticTransitionsWaiting* y se libera el *mutex* del monitor.

10.3.3 FUNCIONES DE DISTRIBUCIÓN DE PROBABILIDAD

En la sección 2.5 se desarrolló el análisis de una *general stochastic Petri net* asumiendo que las transiciones temporales tienen asociadas una distribución exponencial. Sin embargo, se pretende incorporar en la presente iteración la posibilidad de seleccionar otros tipos de distribuciones entre las cuales se encuentran la normal y la uniforme.

DISTRIBUCIÓN NORMAL También conocida como distribución de Gauss, es una distribución continua que se utiliza frecuentemente para modelar fenómenos físicos. La gráfica de su función de densidad tiene forma de campana simétrica como puede observarse en la figura 56.

La función de densidad de probabilidad de una variable aleatoria normal con media (μ) y varianza (σ^2) esta dada por

$$F(x) = \frac{1}{\mu \cdot \sqrt{2\pi}} \cdot e^{-(x-\sigma)^2 / 2 \cdot \sigma^2} \quad (79)$$

DISTRIBUCIÓN UNIFORME Es el modelo continuo más simple. Una variable aleatoria sólo puede tomar valores comprendidos entre dos extremos a y b, de manera que todos los intervalos de una misma longitud tienen la misma probabilidad. La función de densidad de probabilidad de una variable aleatoria uniforme que se encuentre dentro del rango [a, b] esta dada por

$$F(x) = \frac{1}{b - a} \quad (80)$$

En la figura 57 se ilustra la densidad de probabilidad anteriormente mencionada.

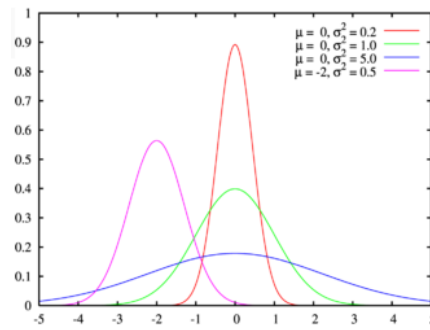


Fig. 56: Distribución normal

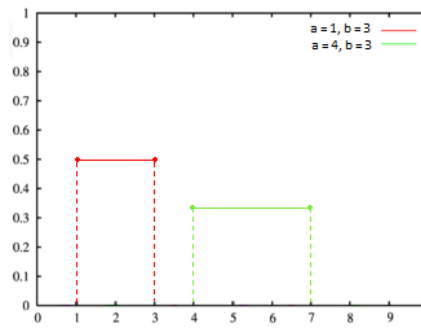


Fig. 57: Distribución uniforme

Para la implementación de estas funciones se utilizó *Apache Commons Math*². La misma es una librería conformada de componentes matemáticos y estadísticos que abordan los problemas más comunes que no se encuentran disponibles en el lenguaje de programación Java. Este paquete prioriza componentes pequeños y fácilmente integrados, motivo por el cual la misma se seleccionó.

Las distribuciones de probabilidad de esta librería se generan a partir de las variables correspondientes a cada distribución. Por ejemplo, para caso de una distribución de probabilidad normal, es necesario crear una instancia de la clase *NormalDistribution* colocando en el constructor de la misma los valores asociados. Estos valores serán la **media (μ)** y la **varianza (σ^2)**, los cuales se obtienen al analizar el archivo *.pnml* de entrada. Una vez generado la instancia de la clase, para obtener una muestra solo es necesario invocar al método *sample()*. Este método se encarga de generar y retornar un valor aleatorio muestreado de esta distribución, que luego será utilizado como tiempo de disparo de la transición temporal.

10.4 CONCLUSIONES

La presente iteración consistió a grandes rasgos en realizar modificaciones sobre iteraciones pasadas o sobre componentes ajenos al editor. Los cambios realizados en el monitor requirieron de un previo análisis del mismo, así como de la lectura del documento que lo detalla. Los resultados obtenidos son satisfactorios, aunque los mismos no se han integrado aún con el editor para visualizar la simulación en el mismo. Por este motivo, no es posible extraer conclusiones definitivas.

De la misma manera, no es posible realizar pruebas. Las mismas se llevarán a cabo en la siguiente y última iteración, puesto que la funcionalidad de simulación de redes estocásticas no será utilizable en su totalidad hasta que ambas iteraciones hayan finalizado.

² Apache, *Commons Math*, [Online]. Disponible: <http://commons.apache.org/proper/commons-math/>.

ITERACIÓN 8: SIMULACIÓN DE REDES GSPN

11.1 INTRODUCCIÓN

Una vez realizadas las modificaciones en el monitor, es necesario implementar en el editor la funcionalidad de ejecutar transiciones estocásticas. En la sección 5.3 se detalló el comportamiento de la clase responsable de ejecutar la simulación, con lo cual las tareas de esta iteración consistirán principalmente en modificar la misma para permitir el disparo de transiciones estocásticas. De la misma manera, se implementará una nueva funcionalidad para graficar, una vez finalizada la simulación, el comportamiento de cada una de las plazas durante la misma. Esto implica generar un gráfico de dos dimensiones cuyos ejes corresponden al tiempo y a la cantidad de *tokens* para cada instante.

11.2 OBJETIVOS

Los objetivos planteados no divergen en gran medida de aquellos propuestos en la introducción del presente documento, aunque se han incorporado ciertos detalles para mejorar la visualización de la simulación. Los mismos serán:

1. Modificar el editor para que el mismo pueda crear los hilos necesarios que intenten disparar transiciones de una *red de Petri* estocástica.
2. Lanzar la simulación en el monitor.
3. Generar la simulación en base a la lista de eventos almacenada en el *observer*, esperando el tiempo correspondiente para el disparo de las transiciones estocásticas. Cabe aclarar que en estos casos, el evento informado al *observer* incluye también la muestra de tiempo generada por el monitor al momento de la sensibilización de la misma.
4. Mientras se espera el tiempo asociado a una transición estocástica, mostrar sobre la misma los milisegundos que restan para el disparo, actualizando la cifra periódicamente.
5. Para cada plaza, generar un gráfico de dos dimensiones que represente su comportamiento durante la simulación. Este comportamiento hace referencia a la cantidad de *tokens* que contuvo para cada instante de tiempo.

11.3 DESARROLLO

La creación de las variables asociadas al monitor, los hilos responsables de disparar las transiciones y el lanzamiento de los mismos no sufre modificaciones con respecto a lo desarrollado en el capítulo 5. La principal diferencia es el comportamiento de la función *fireGraphically()*.

En la figura 23 se representó el diagrama de secuencia de dicha función. Como se puede observar, lo que esta realiza es simplemente recorrer la lista de eventos del *observer* y realizar lo siguiente para cada uno de estos:

1. Obtener el *id* de la transición a partir del evento.
2. Obtener la transición de la red a partir del *id*.
3. Disparar la transición.
4. Refrescar la vista del editor.
5. Esperar el tiempo ingresado por el usuario antes de disparar la próxima transición (variable *timeBetweenTransitions*).

Cuando se tiene una red con transiciones estocásticas, el procedimiento cambia en algunos aspectos. En este caso, no existe un tiempo común para esperar entre los distintos disparos, si no que habrá:

- Un tiempo distinto por cada transición estocástica.
- Un tiempo igual a cero para transiciones no estocásticas (también llamadas transiciones atómicas).

El tiempo asociado a cada transición estocástica se encuentra almacenado en el mismo evento. De esta manera, puede observarse el nuevo comportamiento de la función *fireGraphically()* en el diagrama de secuencia de la figura 58.

Como se observa en esta figura, al analizar un evento de la lista se comienza por obtener el *id* de la transición cuyo disparo lo generó y el tiempo asociado a la misma. Si la transición es estocástica, se debe esperar el tiempo apropiado antes de dispararla. Para ello se invoca al método *countDown()* y se duerme el hilo actual durante el tiempo indicado por el evento. Una vez realizado esto, se dispara la transición. Cabe aclarar que, en caso de no ser estocástica, el hilo no se duerme, indicando que la transición es atómica y que su disparo será imperceptible para el usuario.

Puede darse el caso de que la red no contenga transiciones estocásticas, realizándose esta comprobación antes de finalizar el bucle. Cuando esto sucede, no existen los conceptos de transiciones

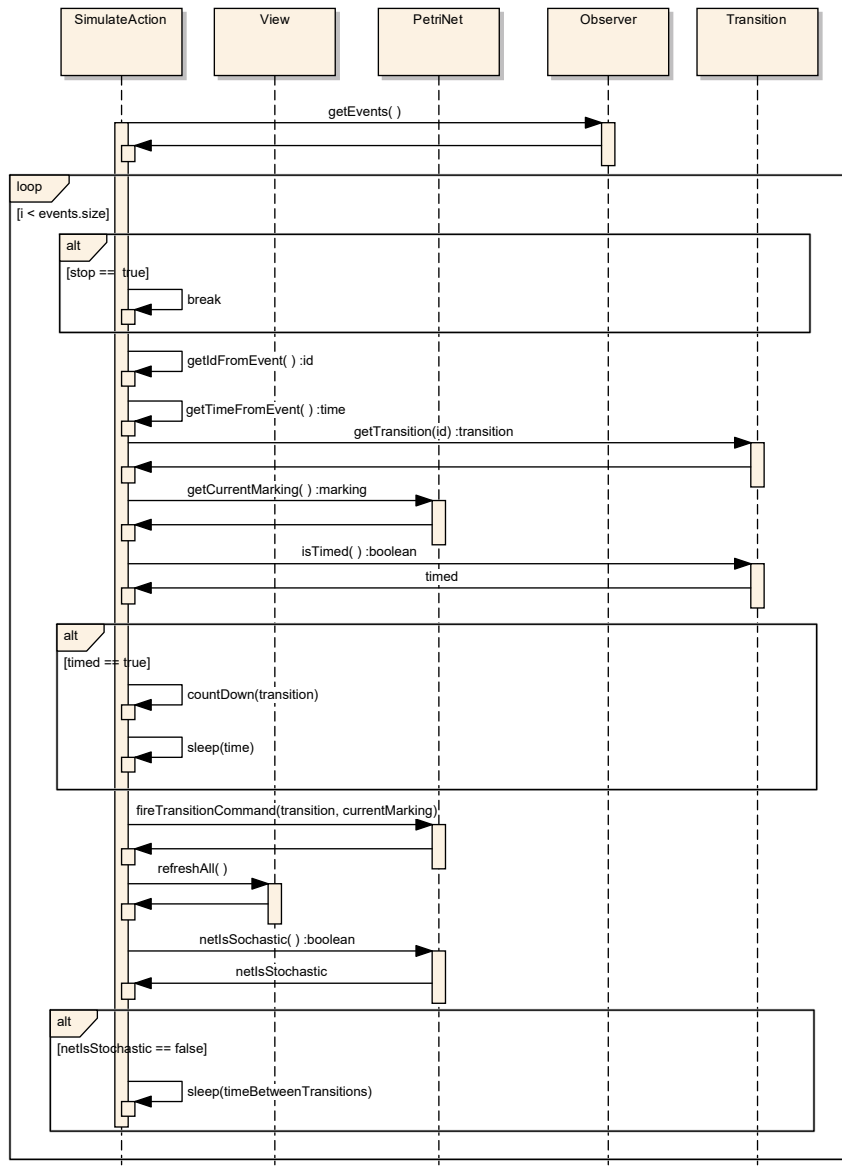


Fig. 58: Diagrama de secuencia de la función *internalFireTransition()* para redes estocásticas

estocásticas y atómicas, por lo que, como se desarrolló en los capítulos anteriores, se espera siempre el mismo período de tiempo entre el disparo de una transición y la siguiente. Por otra parte, el método *countDown()* crea un nuevo hilo que muestra y actualiza periódicamente el tiempo restante para el disparo de la transición estocástica en curso.

Por último, sólo resta representar el comportamiento de las plazas durante la simulación. Esto se realiza generando un gráfico de dos dimensiones donde el eje de las abscisas representa el tiempo y el de ordenadas representa el marcado de una plaza en particular. Para realizar esto se creó una clase llamada *GraphPanel*, quien hereda de *javax.swing.JPanel* y cuyo constructor recibe dos parámetros:

- Una lista que contiene vectores de números de punto flotante, donde todos los vectores tienen la misma cantidad de elementos. El primer vector debe contener las coordenadas del eje x (el cual representa el tiempo), mientras que el resto de los vectores deben contener las coordenadas en y (marcado) para cada una de las plazas que se desee graficar.
- Una lista con los nombres o etiquetas de las plazas correspondientes a los vectores contenidos en la lista.

De esta manera, se representará en un mismo espacio gráfico el comportamiento de todas las plazas deseadas.

11.4 TESTING

Para verificar el correcto funcionamiento de la funcionalidad para la simulación de redes estocásticas se abrió en el editor uno de los ejemplos propuestos por *PIPE*. Este ejemplo es el mismo que el que se utilizó en la sección 5.4, solo que en este caso las transiciones que representan las tareas de "producir" y "consumir" son estocásticas. La red de Petri en cuestión puede observarse en la figura 59.

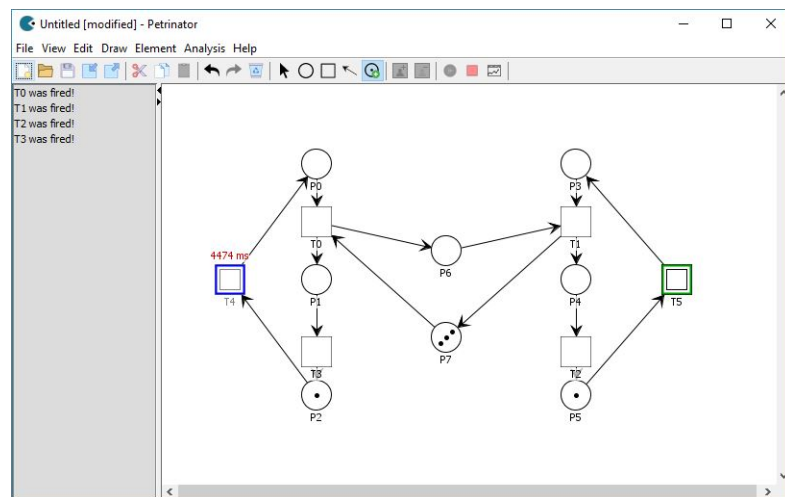


Fig. 59: Sim: red de Petri estocástica

Por otra parte, la figura 60 muestra el diálogo de configuración para la transición T_4 . Aquí se puede apreciar que el tiempo de disparo de la misma posee una distribución normal con una media de 5 segundos y con una varianza de 0,2.

Volviendo a la figura 59, se puede observar que las transiciones T_0 y T_3 se han disparado, sensibilizando de esta manera T_4 . Sin embargo, esta transición no puede dispararse hasta que el tiempo asociado a la misma haya transcurrido. Este estado de una transición se representa con el color azul y una leyenda encima de la misma indicando la cantidad de milisegundos que restan para que ésta pueda dispararse.

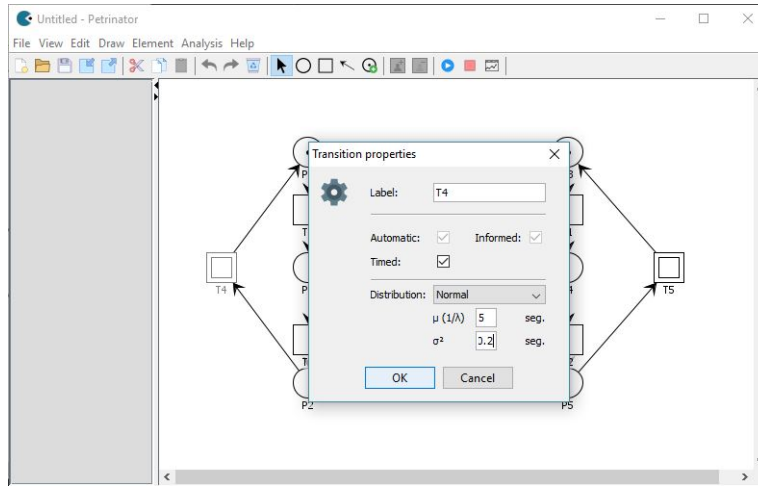


Fig. 60: Sim: configuración de una transición

Por otra parte, se graficó la historia de la plaza P_5 durante la simulación. El resultado obtenido puede apreciarse en la figura 61 y se observa como el marcado de dicha plaza varía entre cero y un *token*.

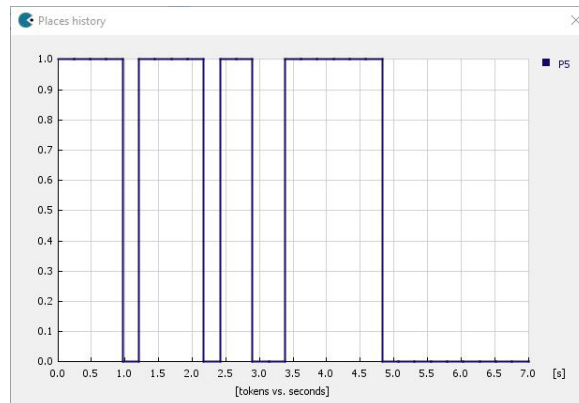


Fig. 61: Sim: historia de una única plaza

Por último, la figura 62 representa la historia de un conjunto de plazas $P = \{P_3, P_4, P_5, P_6\}$ perteneciente a una red de Petri distinta a la del ejemplo anterior; verificando de esta manera la correcta representación de la historia de múltiples plazas en un mismo espacio gráfico.

11.5 CONCLUSIONES

Con las tareas realizadas en la presente iteración se concluye el plan trazado para cubrir los requerimientos del proyecto. La implementación de las mismas no presentó dificultad alguna, puesto que éstas consistieron en gran parte en la modificación de algunos componentes desarrollados en iteraciones anteriores, como por ejemplo la clase *SimulateAction*. Sin embargo, la realización de estas tareas requirió sin duda de un previo estudio de redes de Petri estocásti-

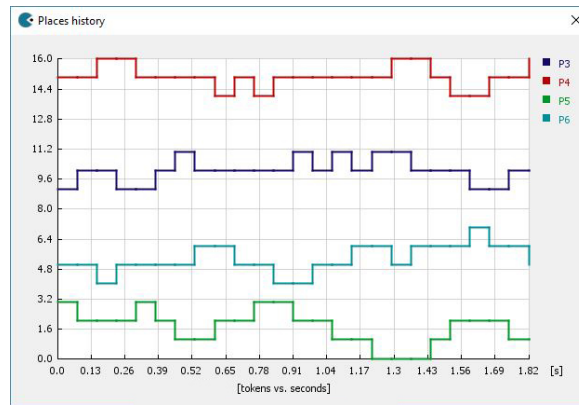


Fig. 62: Sim: historia de múltiples plazas

cas y de las funciones de distribución de probabilidad más utilizadas. Asimismo, se debió realizar un análisis de las librerías estadísticas disponibles en *java*, así como de sus características, alcances y restricciones.

Los objetivos planteados en el capítulo introductorio del documento se han alcanzado sin dificultades significativas. La estimación de tiempo resultó aproximada, así como la de esfuerzo. Las conclusiones generales del proyecto se detallarán en el capítulo 12.

Parte III

CONCLUSIONES

CONCLUSIONES

En el capítulo introductorio del presente documento se especificaron los objetivos planteados para el proyecto desarrollado. Una vez finalizadas las tareas asociadas a los mismos, es posible extraer algunas conclusiones.

Por un lado, se logró integrar el editor y el monitor independientemente de la implementación de este último. Asimismo, el monitor fue modificado y adaptado para el disparo de transiciones estocásticas, permitiendo extender el mismo a la simulación de *general stochastic Petri nets*. De la misma manera, se adaptaron todos los algoritmos de análisis de redes de Petri especificados en los objetivos. La integración de estos tres componentes (editor, monitor y conjunto de algoritmos) permitió crear una herramienta completa y formal para la simulación y análisis de *redes de Petri*. Este análisis puede ser tanto de carácter estático (realizado sobre la estructura de la red de Petri en cuestión) como dinámico (basado en el comportamiento de la misma durante el período de simulación).

Por otro lado, se extendió la funcionalidad de simulación de redes estocásticas para permitir la selección de funciones de distribución de probabilidad distintas de la exponencial. Esta característica posibilita la utilización de redes de Petri para modelar fenómenos físicos con mejores aproximaciones. El resultado obtenido de la combinación de las tareas llevadas a cabo en cada una de las iteraciones planteadas para este proyecto es entonces una herramienta modular, completa y formal para la edición, simulación y análisis de redes de Petri generales y estocásticas.

TRABAJOS FUTUROS

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

BIBLIOGRAFÍA

- [1] Juan Arce Ariel Rabinovich. *Framework de Sincronización de Tareas Coordinado por Redes de Petri*. 2017.
- [2] G.W. Brams. *Las Redes de Petri: Teoría y Práctica*. Masson.
- [3] Richard Carr. «Command Design Pattern.» In: (2009). <http://www.blackwasp.co.uk/command.aspx>.
- [4] Pieter S Kritzinger Falko Bause. *Stochastic Petri Nets*. Universität Dortmund, University of Cape Town, 2002.
- [5] Javier Esparza Jorg Desel. *Free Choice Petri Nets*. Cambridge Tracts in Theoretical Computer Science.
- [6] Fernando Sánchez Figueroa Alexis Quesada Arencibia José Palma Méndez María del Carmen Garrido Carrera. *Programación Concurrente*. Thomson, 2003.
- [7] William J. Knottenbelt. *Generalised Markovian Analysis of Timed Transitions Systems*. 1996.
- [8] G. Conte S. Donatelli M. Ajmone-Marsan G. Balbo and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing, 1995.
- [9] Tadao Murata. *Petri Net: Properties, Analysis and Applications*. IEEE, 1989.
- [10] Hassane Alla Rene David. *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems*. Prentice-Hall, 1992.
- [11] Raymond P. Canale Steven C. Chapra. *Numerical Methods for Engineers*. 6th edition. McGraw-Hill, 2007, pp. 310–318.