Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Centro de Investigación en Comunicación y Redes
Laboratorio de Redes Móviles, Inalámbricas y Distribuidas

Java/PROSEGA: Extensión de CPN Tools para la generación de Lenguajes de Autómatas y la reducción de Grafos de Estado a Máquinas de Estado Finito.

Trabajo Especial de Grado
presentado ante la Ilustre
Universidad Central de Venezuela
Por el Bachiller
Julio César Carrasquel Gámez
para optar al título de
Licenciado en Computación

Tutor: MsC. Ana Verónica Morales Bezeira.
Co-tutor: Dra. María Elena Villapol Blanco.

Caracas, Agosto de 2015.

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Centro de Investigación en Comunicación y Redes
Laboratorio de Redes Móviles, Inalámbricas y Distribuidas

ACTA DEL VEREDICTO

Quienes suscriben, Miembros del Jurado designado por el Consejo de la Escuela de Computación para examinar el Trabajo Especial de Grado, presentado por el Bachiller Julio César Carrasquel Gámez, C.I.: V-20.653.919, con el título **Java/PROSEGA: Extensión de CPN Tools para la generación de Lenguajes de Autómatas y la reducción de Grafos de Estado a Máquinas de Estado Finito**, a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el trabajo por cada uno de los Miembros del Jurado, se fijó el día 14 de Agosto, a la 1:00 p.m., para que su autor lo defendiera en forma pública, en el aula PBIII de la Escuela de Computación, lo cual se realizó mediante una exposición oral de su contenido, y luego respondió satisfactoriamente a las preguntas que les fueron formuladas por el Jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo. En fe de lo cual se levanta la presente acta, en Caracas a los 14 días del mes de Agosto del año 2015, dejándose también constancia de que actuó como Coordinador del Jurado la Profesora Ana Verónica Morales.

_____

Profa. Ana Verónica Morales

_____ _____

Prof. Miguel Astor Prof. Eugenio Scalise

"A la verdad se llega no sólo por la razón,
sino también por el corazón"
Blaise Pascal (1623 – 1662)

A mis padres, Julio y Katiuska.

Agradecimientos

A Dios, creador de este mundo, quien junto a mis padres, me ha dado una oportunidad en esta vida, y me ha guiado en todo este camino, dándome salud y una hermosa familia. A mi madre, eterna luchadora, quien se ha abocado a enseñarme en siempre aspirar a más. A ti madre, agradeceré infinitamente todo su sacrificio. A mi padre quien no ha dejado de apoyarme y velar por mis éxitos.

A la magna e ilustre Universidad Central de Venezuela, la casa que vence las sombras, que me ha dado la posibilidad de formarme en quien he llegado a ser hasta ahora y me ha dado las bases para seguir desarrollándome. A la Facultad de Ciencias y a la Escuela de Computación que por años pasó a ser mi segunda casa. A los profesores de la Escuela de Computación, trabajando incondicionalmente por la excelencia de nosotros, los futuros profesionales del país.

Al laboratorio ICARO y a sus profesores por acogerme en su centro para el correcto desenvolvimiento de este proyecto. En particular, a mi tutora, la profesora Ana Morales quien, con su alto profesionalismo, dedicación y entrega, ha sido mi mentora y guía en este trabajo. A la profesora María Villapol, quien fue mi co-tutora y ha sido precursora en esta línea de investigación. A los investigadores experimentados de esta área, Steven Gordon y Michael Westergaard quienes me dieron consejos sobre temas específicos para el desarrollo de mi proyecto.

A todos los compañeros que he tenido durante la carrera, a quienes he podido y me han podido ayudar. Fuera de la academia, agradezco a mis familiares y amigos, en particular a mi padrino Manlio Delgado, mi novia Claudia y a Juan Ignacio Ortiz. A Wincor Nixdorf C.A, empresa que fue una escuela para mí y en general, a todos los que han contribuido en este proyecto y me han ayudado de manera incondicional.

Resumen

Las Redes de Petri Coloreadas son un lenguaje gráfico que permite modelar de una manera formal sistemas concurrentes a eventos discretos. Las Redes de Petri Coloreadas proveen una técnica poderosa de análisis mediante Grafos de Estado. En particular, CPN Tools es una de las herramientas, utilizadas por diversos investigadores y otros proyectos, para la construcción de Redes de Petri Coloreadas y el análisis de sistemas mediante Grafos de Estado. Este trabajo desarrolla, dentro de CPN Tools, un software denominado Java/PROSEGA que se encarga de reducir, a partir de algoritmos ya conocidos, los Grafos de Estado de los modelos CPN a Máquinas de Estado Finito, de manera de proveer un entorno para el análisis de sistemas concurrentes. Además, el software provee funcionalidades para la generación del lenguaje de autómatas, y también permite realizar comparaciones entre autómatas utilizando funciones de diferencia. Adicionalmente, este trabajo contempla un estudio sobre las técnicas formales en las cuales se basa el software Java/PROSEGA, así como también realiza un estudio sobre las herramientas de software utilizadas, para llevar a cabo dichas tareas de reducción de autómatas y generación del lenguaje. Para llevar a cabo las pruebas sobre el software, se utiliza como caso de estudio un trabajo de verificación del protocolo de comunicación IEEE 802.16, específicamente su proceso de gestión de conexiones a nivel de capa MAC.

Palabras Clave: Redes de Petri, Redes de Petri Coloreadas, CPN Tools, Access/CPN, Extensiones de CPN Tools, Grafos de Estado, Máquinas de Estado Finito.

Tabla de Contenido

Lista de Figuras

Lista de Tablas

Lista de Códigos

CAPÍTULO 1

Introducción

1.1 Contexto de la Investigación

La evolución tecnológica de las últimas décadas ha tenido como resultado el establecimiento de distintos sistemas modernos que son utilizados a escala mundial, tales como estaciones espaciales, satélites, redes de comunicación, sistemas de salud, plantas eléctricas, servidores de correo electrónico, banca en línea, etc. Algunos de estos sistemas se han convertido en una parte importante de nuestras vidas.

Varios de estos sistemas pueden ser considerados críticos ya sea porque son muy costosos de producir y mantener, o porque un fallo dentro de ellos acarrearía una alta pérdida económica, o porque se involucran con la seguridad de las personas. Por consiguiente, se debe utilizar una correcta metodología de desarrollo que garantice que el sistema se comportará correctamente.

Muchos de los sistemas actuales son concurrentes. Concurrente es cualquier entorno en donde esté presente una condición de simultaneidad en la cual conviven múltiples procesos y eventos. Por ejemplo, en una terminal área pueden estar aterrizando múltiples aeronaves en un determinado lapso de tiempo, al igual que, en el mismo lapso de tiempo, pueden estar despegando otro conjunto de aeronaves. Para el buen funcionamiento de esta terminal se debe entonces manejar una correcta política de sincronización. La concurrencia es por lo tanto, una característica

inherente en los sistemas actuales y que debe ser tomada en cuenta en el proceso de desarrollo, el cual usualmente involucra múltiples fases o actividades tales como, análisis de los requerimientos, diseño, implementación, pruebas y mantenimiento.

1.1.1 Modelado y validación de sistemas

Dado lo complejo que puede tornarse un sistema de alta escala que trate con ambientes concurrentes, es factible que los desarrolladores omitan ciertas variables sensibles y patrones de interacción importantes. En consecuencia, esto conllevaría a un mal funcionamiento del sistema. Para enfrentar este problema, es clave hacer énfasis en la fase de modelado.

El modelado es una técnica universal que permite a los desarrolladores tener una visión clara y un entendimiento completo del sistema a construir. La intención de tener un modelo es que se pueda validar y en algunos casos hasta verificar la funcionalidad del sistema en cuestión con la finalidad de minimizar la cantidad de errores a la hora de implementar el sistema. Usualmente para realizar esto se puede utilizar la simulación. Con el soporte de herramientas computacionales mediante ejecuciones y simulaciones del modelo se pueden reducir significativamente la cantidad de errores y defectos de un sistema. La idea es realizar diferentes casos de prueba basados en distintos escenarios para observar así, mediante las corridas, cómo se comporta el sistema, y si la salida que arroja es esperada o no.

1.1.2 Modelos CPN

Las Redes de Petri Coloreadas (*Coloured Petri Nets*) [1] son un lenguaje gráfico que permite construir modelos de sistemas concurrentes para analizar así sus propiedades. Ellas son técnicas formales que heredan sus estructuras básicas y primitivas de funcionamiento de las Redes de Petri. Las Redes de Petri, introducidas

por Carl Adam Petri en los años 60 [2], son una estructura matemática que permite representar modelos concurrentes de eventos discretos. Se dicen eventos discretos pues, aun cuando existe una condición de concurrencia o simultaneidad, la ocurrencia de cada evento se da de manera única en un instante de tiempo en particular. Las Redes de Petri proporcionan a los modelos CPN las primitivas básicas para el modelado de la concurrencia, la comunicación entre elementos y la sincronización.

Las Redes de Petri Coloreadas, además de utilizar las primitivas básicas de funcionamiento de las Redes de Petri, agregan otros elementos (tipos de datos, lenguaje de marcado, manejo de jerarquía, etc.) que enriquecen el conjunto de opciones que se provee para el modelado.

Dominios típicos de aplicación en el modelado de sistemas mediante modelos CPN son protocolos de comunicación, algoritmos distribuidos, sistemas embebidos o procesos de negocios. Sin embargo, dado el abanico de opciones que poseen dentro de sus estructuras, se pueden modelar muchos otros sistemas concurrentes.

Los modelos CPN poseen una base matemática sólida y formal. Además, permiten la utilización de una conveniente visualización gráfica de los modelos de sistemas que ayudan a los desarrolladores de sistemas asegurar y analizar las propiedades deseadas. Son una técnica madura de modelado de sistemas, que tiene como soporte varias décadas de trabajo teórico (artículos de revistas, libros e informes de investigación) y práctico (software directa o indirectamente relacionado con las Redes de Petri Coloreadas).

1.1.3 Análisis de los modelos

Una vez pasada la fase de modelado de un sistema utilizando modelos CPN, se requiere analizar si ciertas propiedades del sistema se cumplen en dicho modelo o

que otras propiedades no deseadas para el sistema son evadidas exitosamente. Este proceso de análisis es posible mediante distintas herramientas de software. Por ejemplo, una herramienta puede probar que ciertas propiedades del sistema se cumplen, y que otras propiedades no deseadas logran ser descartadas. También puede ser analizada cual es la secuencia de estados y eventos que debe poseer el sistema. La representación formal es el fundamento para la definición de varias propiedades de comportamiento y para el uso de distintos métodos de análisis.

El proceso de análisis y verificación formal de los modelos CPN puede estar soportado mediante el método del Grafo de Estados (*State Space*) [1]. Este método radica en computar todos los estados alcanzables de un modelo CPN y todos los eventos que producen dichos estados, y representarlos a través de un grafo dirigido, donde los nodos representen los estados y los arcos representen los eventos que producen estos cambios de estado.

Mediante el uso de Grafos de Estados se pueden responder varias preguntas sobre el comportamiento del sistema, tales como, ausencia de *deadlocks*, la posibilidad que el sistema siempre pueda alcanzar cierto estado en particular, o que se garantice la entrega de algún servicio del sistema en particular mediante la existencia de un camino en el grafo de un estado inicial hasta un estado que modele la entrega exitosa del servicio.

Una desventaja particular del utilizar Grafos de Estados es el problema de explosión del estado (*state explosion problem*) (véase Sección 2.4.4). Existen sistemas, incluso pequeños, que pueden tener un número astronómico o incluso infinito de estados alcanzables. Por lo tanto, realizar el cómputo del Grafo de Estados puede ser altamente costoso. Para tratar de aliviar el problema de explosión del estado, se han desarrollado diversos métodos de reducción [1]. Los métodos de reducción se basan en manipular un Grafo de Estado y convertirlo a un Grafo

reducido que represente de igual manera los estados alcanzables del sistema pero que facilite al desarrollador las labores de análisis. Algunos métodos de reducción pueden ser consultados en Jensen K. et al. [1].

Un método de reducción de particular interés es la transformación del Grafo de Estado a una Máquina de Estados Finitos (*Finite-State Machine)*. Esta técnica de reducción está publicada en Barrett W.A. et al. [3]. Un ejemplo de uso de esta técnica de reducción ha sido incluida en Billington J. et al. [4] como parte de una metodología para la verificación de protocolos de comunicación. Sin embargo, esta técnica de reducción puede ser incluida en cualquier otro procedimiento de modelado y análisis de otro tipo de sistemas.

1.1.4 CPN Tools

La aplicación práctica del modelado de Redes de Petri Coloreadas, y del uso de métodos de análisis como el *State Space*, se basa en la existencia de herramientas de software que efectivamente implementen estas funcionalidades.

CPN Tools [5] es una de las herramientas más utilizadas hoy en día para el modelado de Redes de Petri Coloreadas. Fue desarrollada originalmente por la Universidad de Aarhus en Dinamarca desde el año 2000 hasta el 2010. A partir del año 2010, la curaduría del software ha quedado a cargo del grupo AIS de la Universidad Tecnológica de Eindhoven, Países Bajos [35].

CPN Tools es una herramienta que permite la edición y simulación de las CPNs. Además, permite realizar un análisis de los modelos mediante el cálculo del Grafo de Estado. Para esto, el software comprende, entre otros elementos, un editor gráfico y un simulador. Una descripción más detallada de este software puede ser encontrada en el capítulo 2 del presente trabajo.

1.1.5 PROSEGA

El software PROSEGA (*PROtocol Sequence Generator and Analyser)* es un programa desarrollado por Villapol [10]. El objetivo de este programa es facilitar la conversión de un Grafo de Estado a una Máquina de Estado Finito y la reducción de este último usando la técnica descrita en Barrett W.A. et al. [3]. Este software ha sido utilizado exitosamente y principalmente en el proceso de validación de protocolos de comunicación.

Sin embargo, una desventaja del software es que el mismo es completamente ajeno a CPN Tools. Una visible mejora sería que el algoritmo de reducción que realiza PROSEGA pudiera ser utilizado dentro del mismo CPN Tools de manera de mejorar así la experiencia del usuario.

1.2 Planteamiento del Problema

La herramienta CPN Tools [5] es una herramienta útil para el modelado, simulación y análisis de Redes de Petri Coloreadas. Además, este software ha sido utilizado en diversos escenarios en donde se ha requerido el modelado de sistemas concurrentes a través de Redes de Petri Coloreadas. Por ejemplo, en Jensen K. et al. [1] se describen algunas aplicaciones industriales que se han beneficiado del poder elaborar modelos CPN con ayuda del software CPN Tools.

No obstante, a pesar de las múltiples aplicaciones que se le ha dado a la herramienta, CPN Tools, por sí solo, posee ciertas carencias y deficiencias enumeradas a continuación:

1. A pesar de la utilidad del análisis de protocolos de comunicación siguiendo la metodología de Billington [4], la cual propone la utilización

de Autómatas de Estado Finito, generados a partir de los Grafos de Estados, en la verificación de los mismos y que dicha metodología ha sido utilizada mayormente en desarrollos que involucran Design/CPN (predecesor de CPN Tools) y CPN Tools, esta última no soporta la conversión automática del Grafo de Estado a una Máquina de Estado Finito ni su posterior reducción usando la técnica de Barrett W.A. et al.[3].

2. CPN Tools también tiene la limitante de no poder exportar por sí solo los modelos CPN a aplicaciones externas. Esto pues, ciertas aplicaciones externas pudieran utilizar los modelos CPN desarrollados en CPN Tools para manipularlos e integrarlos con otros ambientes. Sería de alto aprovechamiento que aplicaciones externas pudieran manipular los modelos CPN mediante el establecimiento de un canal de comunicación entre CPN Tools y cualquier aplicación externa.

3. CPN Tools está desarrollado en el lenguaje de programación BETA [6], que no es un lenguaje ampliamente conocido, y en el lenguaje SML [7] que no está diseñado para ningún tipo de interacción con el usuario. Por lo tanto, se necesita que para CPN Tools exista algún tipo de *middleware* o herramienta adicional que permita integrar aplicaciones externas que estén desarrolladas en lenguajes más conocidos y utilizados por desarrolladores a nivel mundial.

Como una solución a las carencias descritas previamente han ido surgiendo diversas herramientas complementarias de apoyo al software de CPN Tools. Por ejemplo, para el problema Nº 3 se desarrolló un módulo para el manejo de extensiones de CPN Tools [8]. Mediante este módulo se pueden desarrollar programas en Java para ser embebidos dentro de CPN Tools y enriquecer así las funcionalidades del software. El problema Nº 2 fue solucionado mediante el

desarrollo de la herramienta Access/CPN [9]. Access/CPN es un framework que permite la conexión de aplicaciones externas al simulador de CPN Tools.

Sin embargo, para el problema Nº 1 no se ha desarrollado una extensión para CPN Tools que sea embebida dentro del mismo software y se encargue de la conversión del Grafo de Estados a una Máquina de Estados Finitos y su posterior reducción usando la técnica propuesta por Barrett W.A. et al. [3] que permita facilitar el proceso de análisis de un modelo cualquiera. Por otro lado, existe un problema adicional, que es la escasa documentación y trabajo que se ha realizado sobre las herramientas adicionales a CPN Tools. Muchas veces la única referencia que se posee del manejo de estas herramientas es directamente desde la fuente de los autores. Ha habido mínimas experiencias de trabajo sobre las herramientas adicionales para CPN Tools como para tener una referencia clara de cómo se puede desarrollar una solución propia al problema de reducir un Grafo de Estados a una estructura minimizada utilizando como base el uso de la herramienta Access/CPN y el manejo de las extensiones para CPN Tools.

1.3 Objetivos de la Investigación

A continuación se describe, el objetivo general del presente trabajo. Además, se describen cuáles son los objetivos específicos detallados del trabajo.

1.3.1 Objetivo general

El objetivo general del trabajo especial de grado es:

Desarrollar una nueva versión mejorada del software PROSEGA que esté embebida dentro de CPN Tools, utilizando como base las herramientas adicionales Access/CPN y el manejador de extensiones.

1.3.2 Objetivos específicos

- Aplicar el manejo de extensiones para CPN Tools y Access/CPN para el desarrollo de una herramienta de conversión de un Grafo de Estados a Autómata de Estados Finitos y su posterior reducción usando la técnica propuesta en Barrett W.A. et al. [3].

- Diseñar la herramienta de conversión de un Grafo de Estados a Autómata de Estados Finitos y reducción del autómata generado de forma tal que sea parte integrada de CPN Tools, siguiendo los lineamientos implícitos de esta última.

- Desarrollar funciones para la generación del lenguaje de los autómatas obtenidos por la herramienta a desarrollar, e implementar funciones para la comparación entre dichos autómatas con el fin de realizar labores de análisis de los modelos.

- Probar la herramienta con algún modelo de algún protocolo de comunicación desarrollado usando CPN Tools.

1.4. Justificación

La metodología de Billington ha probado ser útil para la validación y verificación de protocolos de comunicación [10] [22] [39]. La mayoría de trabajos de investigación que ha utilizado la misma han sido desarrollados con la ayuda de Design CPN y CPN Tools. Sin embargo la herramienta PROSEGA no soporta todas las actividades involucradas en dicha metodología, lo que dificulta su aplicación. Una de las fases del desarrollo de la metodología implica la conversión del Grafo de Estados a un Autómata de Estados Finito y su posterior reducción, como se mencionó anteriormente.

Por lo antes expuesto, dado el escaso trabajo que se ha realizado sobre las herramientas complementarias para CPN Tools, este trabajo pretende abarcar una investigación del correcto uso e integración de las herramientas adicionales para CPN Tools, específicamente la herramienta Access/CPN y la librería para el desarrollo de extensiones para CPN Tools para así aplicar estas herramientas al desarrollo del software requerido.

1.5 Método de desarrollo

Esta sección expone el conjunto de pasos propuestos que se utilizan para alcanzar los objetivos planteados dentro de este trabajo. Este método de desarrollo expuesto correspondió a un plan de trabajo que se concibió ante la necesidad de realizar una investigación de las herramientas complementarias para CPN Tools que fueron necesitadas para el desarrollo de Java/PROSEGA. Adicionalmente, este método de desarrollo se propuso ante la necesidad de desarrollar el software propiamente y ante la necesidad de probar dicho software con distintos casos de prueba.

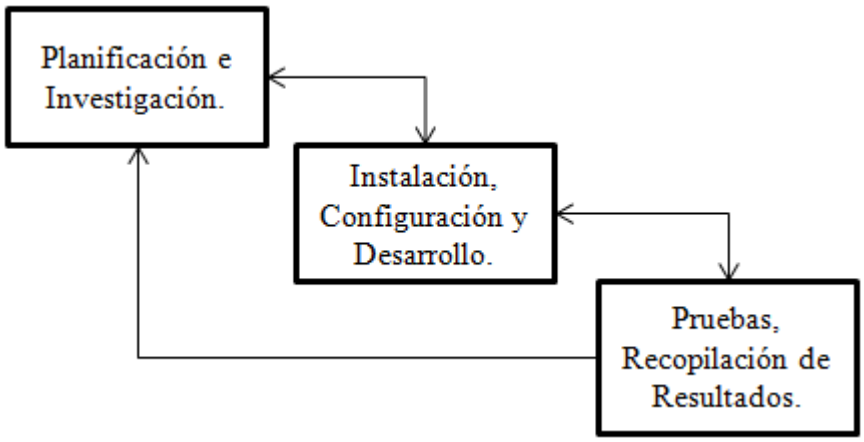

Figura 1.1. Método de desarrollo

La primera etapa correspondió a la fase de planificación e investigación. En esta fase se incluyó toda la investigación realizada sobre las técnicas formales de modelado en las que se basa este trabajo (Redes de Petri, Redes de Petri Coloreadas y su respectivo método de análisis mediante el Grafo de Estados) en conjunto con las herramientas involucradas (CPN Tools, la herramienta Access/CPN y las Extensiones para CPN Tools). De igual manera, abarcó la investigación sobre el proceso de conversión de Grafos de Estados a Máquinas de Estados Finito y que herramientas ayudan en la realización de dicha tarea. Como resultado de la investigación realizada, se realizó una planificación de cómo se desarrollaría el software Java/PROSEGA para ser integrado a CPN Tools. El resultado de esta fase puede ser apreciado en el Capítulo 2 del presente trabajo.

La segunda etapa del proyecto correspondió a la fase de instalación, configuración y desarrollo. En esta fase son utilizados, entre otros instrumentos, el framework Access/CPN y las Extensiones de CPN Tools como base para el desarrollo del software Java/PROSEGA. En conjunto con el desarrollo del software Java/PROSEGA se procedió a la integración del mismo dentro de CPN Tools permitiendo al usuario utilizar nuestro software como una funcionalidad más de CPN Tools. El desarrollo de esta etapa del proyecto es recopilado en el Capítulo 3 del presente trabajo.

La fase final del proyecto correspondió a la realización de pruebas y recopilación de resultados. Una vez que el software Java/PROSEGA se desarrolló completamente se pasó a utilizarlo como herramienta para casos de estudio. Estos casos de estudio se basaron en la verificación de algunos protocolos de comunicación siguiendo la metodología de verificación propuesta en [4]. La idea es probar Java/PROSEGA como herramienta de ayuda para estos casos de estudio específicamente en la transformación de los Grafos de Estados de los modelos CPN de dichos casos de estudio a Máquinas de Estados Finito. Finalmente, una vez se

probó exitosamente el software en esta área se procede a una de recopilación de los resultados obtenidos. El Capítulo 4 de este trabajo presenta el proceso realizado en esta fase.

El método de desarrollo del presente trabajo fue basado en el modelo de desarrollo de software en cascada propuesto por Royce W. [11]. En el caso particular de este trabajo, además del desarrollo del software Java/PROSEGA, interesa la investigación sobre las herramientas adicionales para CPN Tools puesto que el trabajo y documentación sobre estas ha sido escaso.

Otra particularidad del método de desarrollo de este trabajo radica en poder devolverse a fases anteriores. Por ejemplo, si en la fase de pruebas existió algún error o se observó que puede ser incluida alguna mejora en Java/PROSEGA, se es posible devolver a la fase de desarrollo. Incluso, si se debe realizar una investigación adicional para corregir la falla o agregar algún otro componente a la arquitectura prevista se procede a volver a la primera fase de investigación y planificación.

1.6 Antecedentes relacionados del trabajo

Como fue descrito en los objetivos del trabajo (véase Sección 1.4). Este trabajo tiene como objetivo el aprovechamiento de ciertas herramientas adicionales para CPN Tools que permitan crear un software que se integre a CPN Tools, y que se encargue este software de la reducción automática del Grafo de Estado de un modelo CPN cualquiera a una Máquina de Estados Finito.

Consecuentemente, resulta pertinente revisar los siguientes antecedentes de este trabajo mediante una revisión de la literatura disponible. Primero, revisar publicaciones que hayan tratado sobre herramientas complementarias que enriquezcan el abanico de funcionalidades que puede proveer CPN Tools. Segundo,

revisar publicaciones que hayan tratado el desarrollo de software que permita analizar y manipular Grafos de Estados y Máquinas de Estados Finito. Finalmente, revisar publicaciones que involucren el uso específico del método de reducción de Grafos de Estados a Máquinas de Estados Finito.

1.6.1 Publicaciones relacionadas con el manejo de herramientas adicionales para CPN Tools

Las siguientes publicaciones exponen un conjunto de herramientas complementarias que han sido desarrolladas para ser integradas con CPN Tools para enriquecer así las funcionalidades que este software puede ofrecer.

- En Westergaard et al. [9] se presenta la herramienta Access/CPN. La motivación de la herramienta Access/CPN es la de proveer un canal para que aplicaciones basadas en Java puedan conectarse al simulador de CPN Tools. Además, se explica en este trabajo las dos interfaces que provee la herramienta: una interfaz basada en Java para proveer al usuario del framework un conjunto de clases que permitan encapsular a los distintos elementos de los modelos CPN y otra interfaz en lenguaje SML que permite enviar al simulador expresiones en SML para ser procesadas. Esta publicación contiene un programa de ejemplo de cómo se puede utilizar la herramienta para crear una solución utilizando código Java y SML que permita encontrar los marcados muertos de un modelo CPN en particular. Sin embargo, una deficiencia de esta publicación es que no provee los pasos necesarios para poder utilizar efectivamente Access/CPN (instalación, configuración, etc.)

- En el blog de Westergaard M., específicamente en las entradas [8] [12] [13] [14] [15], se expone una exhaustiva explicación de cómo se pueden desarrollar extensiones para CPN Tools utilizando como base la librería para

el manejo de extensiones de CPN Tools. Adicionalmente, se explica como una extensión se puede comunicar con el simulador de CPN Tools utilizando como base el protocolo BIS [14] (protocolo propietario de CPN Tools para la comunicación entre el editor gráfico y el simulador).

- En Westergaard et al. [16] se presenta Grade/CPN. Esta herramienta, de carácter académico, provee un soporte a profesores para calificar modelos CPN que hayan sido desarrollados por sus alumnos tomando como base una especificación dada. Esta herramienta fue construida a partir del uso del framework Access/CPN [9].

- En Gallash et al. [17] se presentó la librería Comms/CPN. Esta librería, como se presenta en la publicación, fue desarrollada para el software predecesor de CPN Tools, Design/CPN [18]. Es una librería precursora en la creación de canales de comunicación entre modelos CPN y aplicaciones externas. Esta librería se basa en comunicarse con la herramienta Design/CPN a través de sockets. En [19] es posible conseguir un conjunto de archivos en el lenguaje C y en Java que permiten que programas escritos en C y Java respectivamente se puedan conectar a CPN Tools mediante el uso de sockets. Este último conjunto de archivos son una adaptación de Comms/CPN para ser utilizado para CPN Tools.

1.6.2 Publicaciones relacionadas con el desarrollo de Software para el análisis de Grafos de Estados (*State Space*) y Máquinas de Estados Finito.

En esta sección se exponen publicaciones que han presentado soluciones de software que fueron desarrolladas para servir como herramientas para el análisis y manipulación de Grafos de Estados y/o de Máquinas de Estados Finito.

- En Kristensen L.M. et al. [20] se presenta el software ASCoVeCo como una plataforma para el análisis de Grafos de Estados. En esta publicación se expone ASCoVeCo (*Advanced State Space Methods and Computer Tools for Verification of Communication Protocols*) como una herramienta de software específica para el análisis y verificación de protocolos de comunicación. El artículo presenta la arquitectura de la solución en conjunto de cómo los investigadores pueden utilizar la herramienta para el análisis de Redes de Petri Coloreadas mediante distintos métodos de análisis de Grafos de Estados. Esta solución, como se describe en la publicación, está construida en una interfaz gráfica propia sin guardar algún tipo de relación directa con el software CPN Tools. En [21] se consigue el enlace oficial del proyecto que provee acceso al software y la documentación necesaria para su correcta manipulación.

- En Villapol M.E. [10] se presenta el software PROSEGA como una herramienta para la reducción de Grafos de Estados a Máquinas de Estados Finito. El uso particular del software en este trabajo fue el de generar, mediante la Máquina de Estados resultante, el lenguaje de las primitivas de servicio y del propio protocolo de comunicación RSVP a fin de realizar una comparación de lenguajes siguiendo la metodología de verificación de protocolos publicada en [4]. Este software también fue utilizado en Morales A.V. [22] para obtener una Máquina de Estados Finito para el análisis del modelo CPN de la gestión de conexiones a nivel de capa MAC del protocolo de comunicación IEEE 802.16.

- Los laboratorios de investigación de la empresa de AT&T desarrollaron el software AT&T FSM Library [27]. Este software no trabaja particularmente con Grafos de Estados, sino más bien trabaja con Máquinas de Estados Finito. La librería provee una interfaz en C/C++ y una interfaz de comandos para la manipulación de Máquinas de Estados Finito. La aplicación particular

que le ha dado laboratorio al software ha sido sobre soluciones de reconocimiento de voz (*speech recognition*). Por otro lado, el software PROSEGA [10] invoca las funciones de esta librería para la reducción de Grafos de Estado (que son ajustados previamente a un formato entendible por la librería) a una Máquinas de Estado Finito. Sin embargo, el laboratorio de AT&T dejó de proveer esta librería de manera abierta.

- En Riley M. et al. Se presenta OpenFST [28]. Esta herramienta, desarrollada por los mismos investigadores que trabajaron en un primer momento con AT&T FSM Library [27], presentan ahora una librería de código abierto que está plenamente accesible para la manipulación de Transductores de Estado Finito que son un tipo particular de Máquinas de Estado Finito. En esta publicación se explica cómo se utiliza esta herramienta que posee una interfaz en C++ y otra interfaz en línea de comandos.

1.6.3 Publicaciones relacionadas que involucran el uso de métodos de reducción de Grafos de Estado a Máquinas de Estado Finito.

Esta sección presenta algunas publicaciones que han involucrado el uso de métodos de reducción de Grafos de Estados a Máquinas de Estados Finito como parte de sus respectivos trabajos a fin de analizar los sistemas que se estén modelando. En particular, en los siguientes trabajos, los sistemas en estudio son protocolos de comunicación.

- En Barrett W.A. et al. [3] se presenta un método de reducción de Máquinas de Estado Finito, sin involucrar el termino de Grafos de Estado (*State Space).* En este trabajo simplemente se refiere a la minimización de Máquinas de Estado Finito a través de una serie de pasos (remoción de *epsilons*, determinización,

minimización, etc.). En este caso la orientación principal de la minimización de Máquinas de Estado era el utilizar este proceso en el área de Compiladores.

- En Billington J. et al. [4] se presenta una metodología para la verificación de protocolos de comunicación utilizando modelos CPN. En la metodología propuesta de este trabajo se deben modelar las primitivas de servicio [23] del protocolo que se esté trabajando, generar el correspondiente Grafo de Estados y, a través de este último, la Máquina de Estado Finito resultante siguiendo el algoritmo de reducción descrito en [3]. El mismo proceso se realiza para modelar el protocolo propiamente. Finalmente, se realiza una comparación de los lenguajes de ambas Máquinas de Estado resultantes para realizar la verificación efectiva del protocolo. Este trabajo además de presentar la metodología, también presenta unos ejemplos que utilizan dicha metodología. El primer ejemplo es utilizando un modelo genérico de protocolos del tipo *Stop-and-Wait*. El segundo ejemplo presentado es el modelado de la gestión de conexiones del protocolo TCP [24].

- En Villapol M.E. [10] se utiliza la metodología pautada en [4] para la verificación el protocolo de reservación de recursos RSVP [25]. El trabajo se encarga de modelar, mediante Redes de Petri Coloreadas, tanto las primitivas de servicio como el mismo protocolo, generar los Grafos de Estado y reducirlos a las Máquinas de Estado asociadas, para finalmente realizar una comparación de los lenguajes de las Máquinas de Estado resultantes.

- En Morales A.V. [22] se realiza el modelado y análisis de los procesos involucrados en la gestión de las conexiones de la capa MAC de IEEE 802.16 [26] (conocido como WiMax) utilizando Redes de Petri Coloreadas. En este caso, el trabajo realizado fue el modelado de las primitivas de servicio siguiendo la metodología descrita en [4]. Luego, se realiza la generación del

Grafo de Estado asociado al modelo para generar la respectiva Máquina de Estado.

1.7 Estructura del trabajo

A continuación, se presenta un listado del contenido temático para cada uno de los capítulos del presente trabajo:

- **Capítulo 1: Introducción.** En esta sección inicial del trabajo se pretende introducir al lector en la línea de investigación en la que se desarrolla este trabajo. Luego, se procede a explicar el problema del escenario en el cual se ve envuelto este trabajo, la justificación del autor para proceder en el desarrollo de este trabajo, así como también cuales son los objetivos que se persigue y el alcance que se le dará. Se presenta el método de desarrollo utilizado en este proyecto. Finalmente, se presentan las publicaciones relacionadas con el presente trabajo.

- **Capítulo 2: Marco Conceptual: Técnicas Formales y Herramientas.** Esta sección expone las técnicas formales que son base de la investigación (Redes de Petri, Redes de Petri Coloreadas, Método del *State Space*, etc.) así como también las herramientas de software usadas para el desarrollo de este proyecto (CPN Tools, Access/CPN, Extensiones de CPN Tools, etc.).

- **Capítulo 3: Java/PROSEGA.** En esta sección se presenta propiamente el software desarrollado y todas las experiencias obtenidas en el desarrollo del mismo. Presenta las técnicas utilizadas para su desarrollo, así como también como debe ser utilizado. Consecuentemente, esta sección presenta una referencia técnica y una documentación exhaustiva sobre el manejo del software desarrollado.

- **Capítulo 4: Caso de estudio con Java/PROSEGA. Análisis de la gestión de conexiones a nivel de capa MAC del estándar IEEE 802.16.** Este capítulo del trabajo recopila el caso de estudio que fue utilizado para probar la aplicación Java/PROSEGA. En particular, esta sección explica cómo fue utilizado el software en el proceso de verificación del protocolo de comunicación IEEE 802.16, específicamente la gestión de conexiones a nivel de capa MAC. Además, esta sección expone los resultados arrojados a partir de las pruebas realizadas.

- **Capítulo 5: Conclusiones.** Se presentan los resultados generales de este trabajo y un conjunto de reflexiones sobre el trabajo realizado en función de los objetivos propuestos en un principio. Además, presenta los aportes realizados de este trabajo a la línea de investigación, así como también presenta los trabajos futuros que se pueden desprender del trabajo realizado.

CAPÍTULO 2

Marco Conceptual: Técnicas Formales y Herramientas

2.1 Introducción

El presente capítulo presenta una descripción de las técnicas formales que forman parte del marco teórico de este trabajo (Redes de Petri, Redes de Petri Coloreadas, etc.). Además, se describen formalmente las herramientas de software que son utilizadas en esta investigación.

2.2 Redes de Petri

Las Redes de Petri [30] son unas estructuras matemáticas y gráficas que permiten representar sistemas concurrentes a eventos discretos. Fueron definidas en los años 60 por Petri C.A [2]. Las Redes de Petri permiten describir distintos tipos de sistemas que se caracterizan por ser concurrentes, asíncronos, distribuidos, paralelos, no-determinísticos o estocásticos.

Una Red de Petri es un grafo dirigido que posee dos tipos de nodos, plazas y transiciones. Estos se conectan a través de arcos dirigidos. Los arcos conectan a las plazas con las transiciones así como a las transiciones con las plazas. Es decir, no puede haber conexión directa ni entre plazas ni entre transiciones. Por lo tanto, una transición puede estar conectada a un conjunto de plazas de entrada y a otro conjunto de plazas de salida.

Las plazas pueden contener un cierto número de marcas (tokens). Estas marcas fluyen en la red de acuerdo a las ocurrencias de las transiciones. La distribución inicial de las marcas en la red se denomina marcado inicial (denotado formalmente M_0). Los arcos pueden poseer un peso asociado. Para que se dé la ocurrencia de una transición, debe haber, para cada plaza de entrada, un número de marcas igual o mayor a lo que indique el peso del arco que conecta dicha plaza con la transición. Luego, cuando se da la ocurrencia de la transición, se toman tantas marcas de las plazas de entrada como lo indiquen sus respectivos arcos que conectan a la transición, y se producen luego en las plazas de salida tantas marcas como lo indiquen sus respectivos arcos que se conectan a la transición.

La figura 2.1 [30] modela la reacción química, $2H_2 + O_2 \rightarrow 2H_2O$. Las plazas de entrada, figura 2.1 (a), contienen las marcas que representan las 2 unidades de H_2 y la unidad de O_2, necesarias para la ocurrencia de la transición. Luego, cuando se da la ocurrencia de la transición, figura 2.1 (b), se coloca en la plaza de salida el resultado que son las dos unidades de H_2O. Después de la ocurrencia de esta transición, esta no sigue habilitada pues, según los arcos de entrada, se necesitan al menos dos unidades de H_2 y una unidad de O_2.

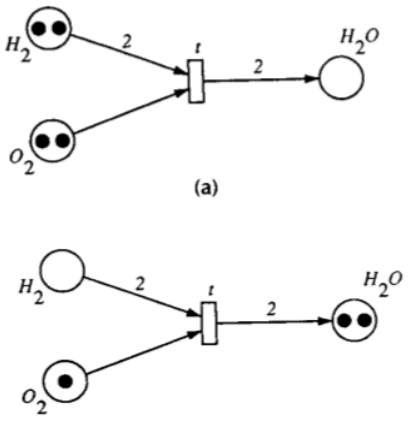

Figura 2.1. Ejemplo de habilitación y ocurrencia de una transición [30].

Con el ejemplo anterior, queda demostrado de una manera práctica como pueden ser utilizadas las plazas para modelar entidades que almacenen datos de entrada o de salida, y como las transiciones pueden modelar tareas o eventos que producen los cambios de estado del sistema.

Los estados de un sistema pueden ser modelados mediante los marcados. Los marcados en una Red de Petri denotan la distribución de las marcas dentro de la red en un momento en particular. Partiendo del marcado inicial, la ocurrencia de cada transición, genera un nuevo marcado en particular dentro del sistema.

2.2.1 Definición formal de las Redes de Petri

Las Redes de Petri puede ser definidas formalmente de la siguiente manera [30]: Una Red de Petri es una 5-tupla, $PN = (P, T, F, W, M_0)$ donde:

$P = \{p_1, p_2, \dots, p_n\}$ es un conjunto finito de plazas.
$T = \{t_1, t_2, \dots, t_n\}$ es un conjunto finito de transiciones.
$F \subseteq (P \times T) \cup (T \times P)$ es un conjunto de arcos.
$W: F \rightarrow \{1, 2, 3, \dots\}$ es una función de asignación de pesos a los arcos.
$M_0: P \rightarrow \{0, 1, 2, 3, \dots\}$ denota el marcado inicial de la red.
$P \cap T = \emptyset$ y $P \cup T \neq \emptyset$

2.3 Redes de Petri Coloreadas

A medida que los sistemas se tornan más complejos, los modelos basados en las Redes de Petri [30] pueden llegar a ser muy grandes, complejos y probablemente ilegibles. Para esto, surgieron las Redes de Petri de Alto Nivel [31]. Las Redes de Alto Nivel agregan nuevas características de funcionamiento a las Redes de Petri para enriquecer así el conjunto de opciones disponibles para el modelado de sistemas.

Dentro de este conjunto de Redes de Petri de Alto Nivel se encuentran particularmente las Redes de Petri Coloreadas. Las Redes de Petri Coloreadas (CPNs, *Coloured Petri Nets*) [1] son un lenguaje gráfico que permite la construcción de modelos de sistemas concurrentes y permite un análisis formal de sus propiedades. Las Redes de Petri Coloreadas incorporan tipos de datos para los tokens (*color sets*) así como también se apoyan en el lenguaje CPN ML [1] para la definición de estos tipos de datos y para describir la manipulación de los datos dentro de la red.

2.3.1 Estructura de la red e inscripciones

Para explicar la estructura de las Redes de Petri Coloreadas se utilizará primero como ejemplo un modelo de telefonía básica propuesto en [32]. Este modelo se encarga de modelar un sistema de telefonía básico desde la perspectiva de un usuario.

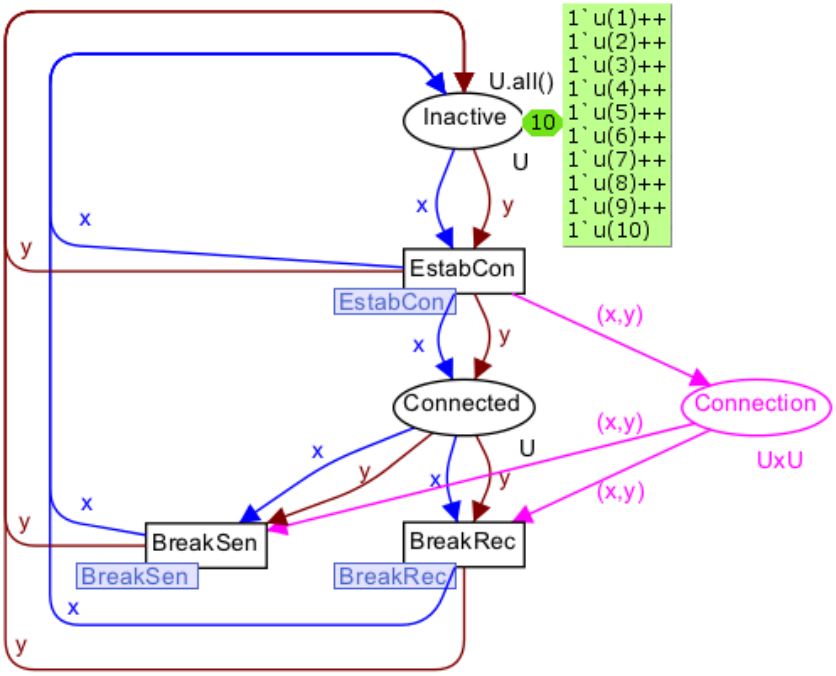

Figura 2.2. Modelo CPN que representa un modelo básico de telefonía [32].

Los teléfonos en este modelo CPN están representados en la red como tokens *u* (elementos concatenados en el recuadro verde). Estos tokens son definidos mediante el apoyo del lenguaje CPN ML (ver figura 2.2). Estos teléfonos pueden estar en dos estados, un estado inactivo o un estado conectado. Estos dos estados están modelados mediante las plazas *Inactive* y *Connected* respectivamente.

En un primer momento todos los teléfonos se encuentran en un estado inactivo y esto se modela colocando todos los tokens que representan los teléfonos en la plaza *Inactive*. Por otro lado, la plaza *Connection* se encarga de modelar y manejar la conexión entre dos teléfonos.

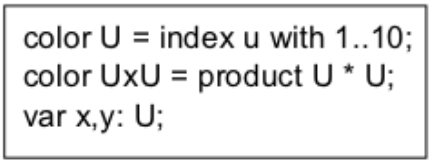

Figura 2.3. Definición de los tokens mediante CPN ML [32].

En la figura 2.2 se observa cómo se definen los *color sets* o tipos de datos que se estarán utilizando en la red mediante CPN ML [1]. El *color set* U define todos los tokens *u* que representan los teléfonos (se define un conjunto de 10 tokens). El segundo *color set* UxU es resultado de aplicar la función *product* (producto cartesiano) del conjunto U para definir así los pares ordenados que representarán la asociación entre dos teléfonos. Luego, se definen las variables x e y que son utilizadas para representar los teléfonos que fluirán a través de la red (ver figura 2.1).

Es importante mencionar que cada plaza solo puede manejar un *color set* o tipo de dato. La plaza *Inactive* indica, mediante inscripciones, que aloja solo tokens de tipo U mediante que la plaza *Connected* maneja tokens con tipo de dato UxU. Los arcos que conectan las plazas con las transiciones deben a su vez transportar

solamente el tipo de dato soportado por la plaza y para esto son utilizadas las variables *x* e *y* definidas en la figura 2.2 y que son colocadas en las inscripciones de los arcos.

Por otra parte, las transiciones en esta red modelan los cambios de estado que ocurren en las conexiones telefónicas. En el módulo principal del modelo presentado (figura 2.1) se tienen tres transiciones:

- *EstabCon:* modela el establecimiento de la conexión entre dos teléfonos. Se toman dos tokens de la plaza *Inactive* y, si la conexión es exitosa, se producirá un token de tipo (x,y) para la plaza *Connection* y dos tokens separados x e y para la plaza *Connected*. Si la conexión no es exitosa, se colocarán nuevamente estos tokens utilizados en la plaza *Inactive*. Esto se modela mediante par de arcos que se conectan nuevamente a la plaza *Inactive* desde esta transición.

- *BreakSen:* Esta transición modela el cierre de la llamada telefónica por parte del emisor. Toma un par de teléfonos que se encuentren conectados (tokens en las plazas *Connected* y *Connection*) y los coloca nuevamente en la plaza *Inactive.*

- *BreakRec:* Realiza la misma lógica que la transición *BreakSen*, pero en este caso el cierre de la conexión entre los dos teléfonos la solicita el receptor y no el emisor.

2.3.2 Habilitación y ocurrencia de las transiciones

Las expresiones de los arcos de los modelos CPN son quienes se encargan de determinar cuándo una transición dentro del modelo está habilitada o no (habilitada para que pueda ocurrir la transición). Para que una transición esté habilitada debe ser posible encontrar un enlace o asociación de las variables de los arcos que se conectan a la transición con posibles tokens que se encuentren en las correspondientes plazas de entrada a dicha transición [1].

En el modelo telefónico presentado previamente (ver figura 2.1) es fácil determinar cuándo una transición puede estar habilitada. Por ejemplo, en el marcado inicial M_0 que se presenta, la transición *EstabCon* se encuentra habilitada pues los dos arcos que van de la plaza *Inactive* a la transición *EstabCon* pueden consumir tokens que se encuentran en la plaza *Inactive* realizando una asociación efectiva entre las variables x e y (de tipo de dato U) con los tokens u (de tipo de dato U) de la plaza *Inactive*. Luego, cuando se consumen estos tokens por la transición, este colocará tokens en las plazas de salida, en este caso en las plazas *Connected* y *Connection*, de acuerdo a las expresiones de los arcos de salida.

Para el módulo principal del modelo telefónico, los arcos de entrada y de salida solo tienen como inscripciones las variables x e y. Sin embargo, en las Redes de Petri Coloreadas los arcos también pueden poseer expresiones más complejas utilizando para ello lenguaje CPN ML.

Por ejemplo, en [1] se describe una Red de Petri Coloreada que modela un protocolo básico emisor – receptor con manejo de reconocimientos o ACKs. La figura 2.4 muestra el modelo mencionado.

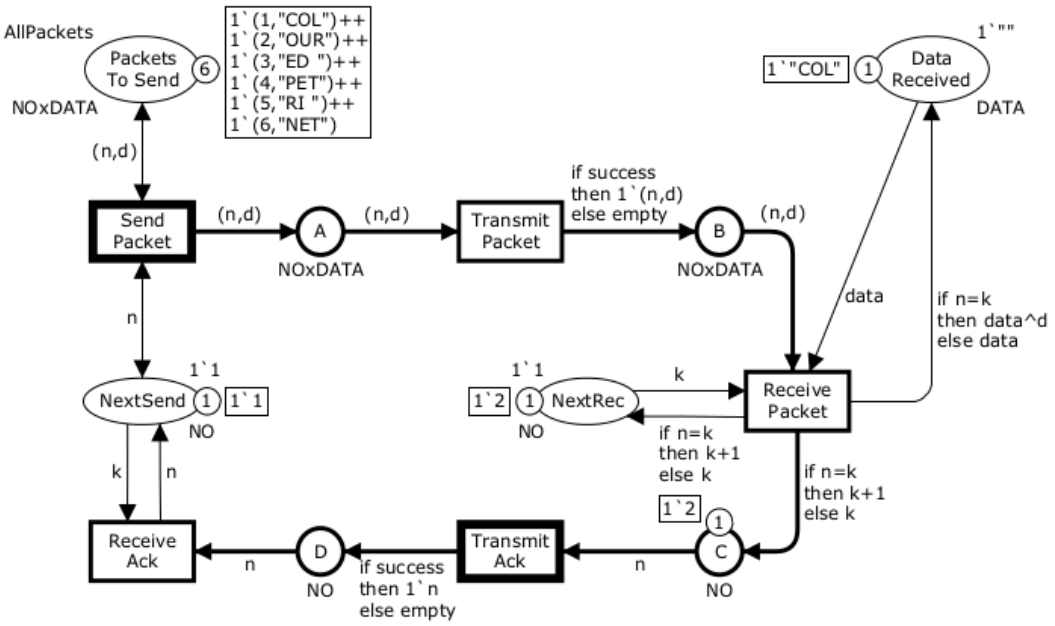

Figura 2.4. Modelo CPN de emisor-receptor con manejo de ACKs [1].

En este caso, vemos como algunos arcos de la red poseen más que variables para determinar el resultado de la transición. Por ejemplo, la transición *TransmitPacket* toma un par ordenado *(n, d)* como entrada. Sin embargo, su salida está determinada mediante un condicional que indica: si la variable booleana *success* es igual a verdadero, se colocará la variable (*n, d)* en la plaza B. Si no, se colocará la marca vacía (*empty)* en la plaza B.

Con el ejemplo de la figura 2.3 se observa cómo se utilizan expresiones más complejas que solo variables para determinar la salida de una transición, e incluso, este tipo de expresiones pueden utilizarse para determinar si una transición puede estar habilitada o no, utilizando este tipo de expresiones en los arcos de entrada.

2.3.3 Guardas

En la sección 2.3.2 se explicó que las expresiones de los arcos de entrada determinan cuando una transición está habilitada o no en un marcado en particular. Sin embargo, una transición también puede poseer una guarda. Una guarda es una expresión booleana que debe retornar verdadero para que la transición que contenga dicha guarda pueda estar habilitada. Esta expresión booleana es construida utilizando lenguaje CPN ML. De esta manera, las guardas agregan una nueva restricción para que una transición se encuentre habilitada agregando así un mayor nivel de complejidad.

A continuación se presenta un ejemplo de uso de las guardas. En la figura 2.4 se muestra una variante del módulo del receptor del modelo presentado en [1]. En este modelo dos entes (emisor y receptor) intercambian datos a través de la red representados como tokens de tipo pares ordenados, donde el primer valor representa el número de secuencia del dato (denominado *n*) y el segundo valor representa el dato o la carga útil en sí (denominado *d)*. Adicionalmente, el módulo del receptor modela una plaza denominada *NextSec* que se encarga de almacenar el número de secuencia del dato esperado por el receptor.

Para ejemplificar el beneficio de las guardas, se utilizan en esta variante (figura 2.4) dos transiciones con guardas dibujadas mediante corchetes. La transición *ReceiveNext* solo podrá estar habilitada cuando el número de secuencia esperado es exactamente igual al del dato proveniente (n = k) mientras que la transición *DiscardPacket* solo podrá estar habilitada cuando el número de secuencia del dato proveniente sea distinto al número de secuencia esperado.

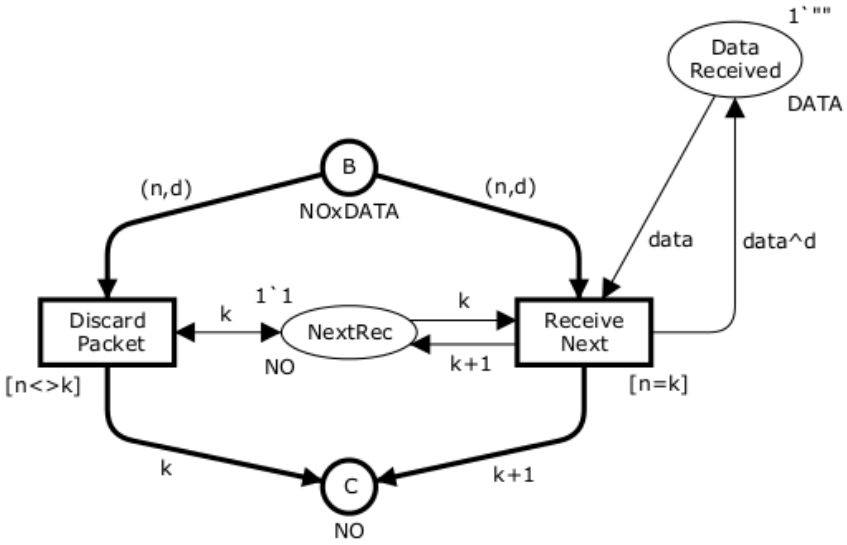

Figura 2.5. Modelo CPN ejemplificando el uso de guardas [1].

De esta manera, mediante la lógica implementada con el uso de las guardas, este modelo puede determinar cuándo un paquete puede ser desechado o recibido en función de si era el esperado o no. Con esto se demuestra de manera práctica el uso que pueden darse a las guardas.

2.3.4 Manejo de jerarquía

Esta sección explica como un modelo CPN puede estar organizado en un conjunto de páginas o módulos. Esto se realiza de una manera similar a como los programas se estructuran en una serie de rutinas. Los módulos CPN permiten trabajar a diferentes niveles de abstracción, donde en un nivel superior se tiene una vista general del sistema que se está modelando y cada parte de este módulo superior es explotado en niveles o módulos inferiores que poseen mayor nivel de detalle construyendo así un modelo jerárquico.

Para manejar la jerarquía en las Redes de Petri Coloreadas se utilizan las transiciones de substitución. Las transiciones de substitución son puntos de entrada a los sub módulos de un modelo. Estas actúan como cajas negras y encierran dentro de sí toda la lógica que haya sido diseñada en los sub módulos a los que apuntan. Estas son dibujadas como una transición pero con un doble borde y con un comentario adjunto que indica a que módulo o página se encuentra apuntando dicha transición.

Como primer ejemplo del manejo de jerarquía en los modelos CPN se citará el modelo básico de comunicación emisor-receptor con manejo de reconocimientos (ACKs) presentado en [1] (ver figura 2.4). En esta ocasión, la figura 2.6 presenta una variante de ese modelo utilizando transiciones de substitución que conllevan a los sub módulos de emisor, red y receptor. De esta manera se está modelando dicho sistema de una manera jerárquica y organizada.

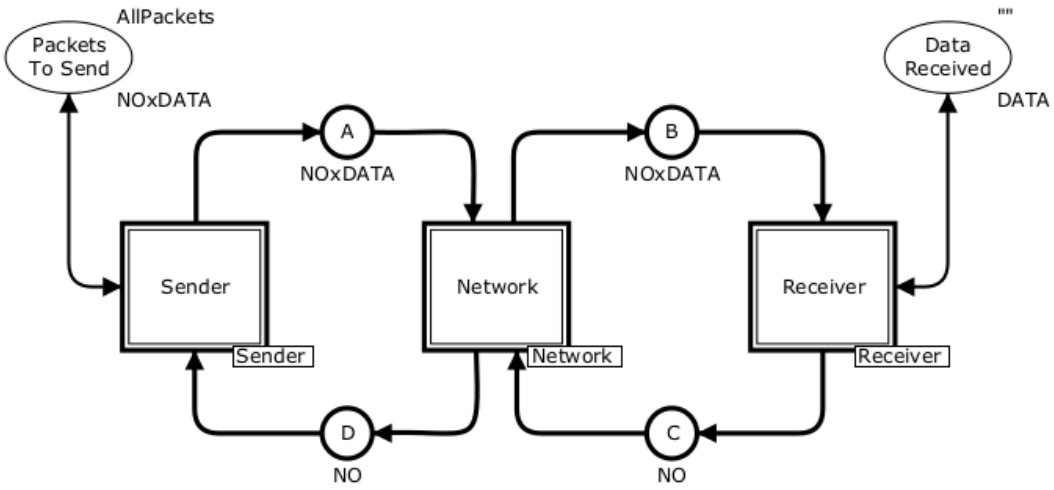

Figura 2.6. Ejemplo de un modelo CPN con manejo de jerarquía [1].

Como segundo ejemplo para el manejo de jerarquía se usará nuevamente el modelo telefónico [32] (ver figura 2.2) presentado en la sección 2.3.1, las tres transiciones son transiciones de substitución pues estas representan módulos

inferiores que contienen un mayor nivel de detalle de la red. *EstabCon* es una transición de substitución que lleva a un módulo que modela con mayor nivel de detalle cómo se establece la conexión mediante dos teléfonos. Mientras que las otras dos transiciones de substitución, *BreakSen* y *BreakRec* apuntan a dos módulos que modelan la finalización de una llamada entre dos teléfonos.

Como ejemplo, se presenta en la figura 2.7 el módulo de establecimiento de la conexión al que apunta la transición de substitución *EstabCon.*

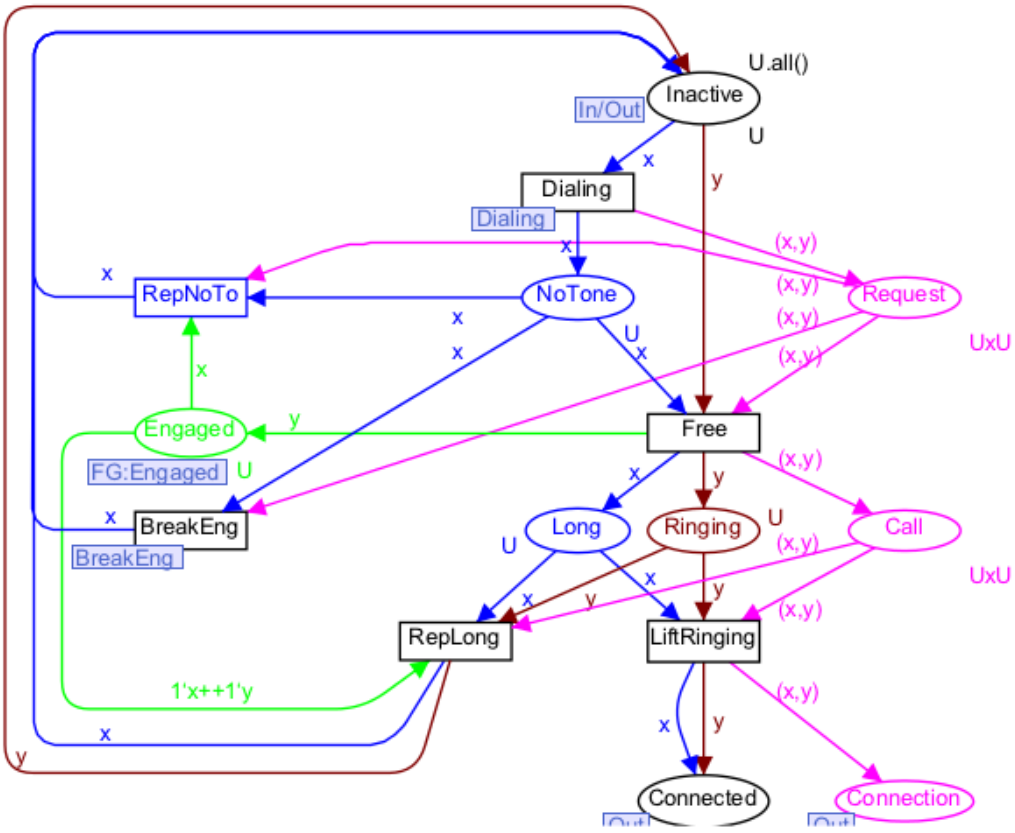

Figura 2.7. Ejemplo de sub módulo en un modelo CPN [32].

En la figura 2.7 se aprecia el módulo de establecimiento de la llamada telefónica *EstabCon*. En este sub módulo se tiene la plaza de entrada al sub módulo

Inactive que es de donde provienen los teléfonos con estado inactivo. Esta plaza también hace de plaza de salida del sub módulo puesto que este modelo contempla el caso de fallo en el establecimiento de la conexión. En caso de un establecimiento exitoso de la conexión, los tokens arribarán a las plazas de salida del sub módulo *Connected* y *Connection*. Las etiquetas *In*, *Out* e *In/Out* son las que se encargan de determinar si una plaza es de entrada, salida o entrada/salida a un sub módulo.

Para hacer una asociación adecuada entre una transición de substitución y un módulo de la red, se introducen los términos de plazas de tipo puerto y plazas de tipo socket. Las plazas de entrada y de salida que se conectan a la transición de substitución se denominan plazas sockets. Estas mismas plazas sockets deben estar presentes en el sub módulo o página al que apunta la transición de substitución y en ese lugar son denominadas plazas puerto. Por lo tanto, al colocar una transición de substitución debe estar presente esta correcta asociación y relación entre las plazas sockets y las plazas puerto.

Como ejemplo de lo anterior descrito, se toma nuevamente el modelo telefónico [32]. En la figura 2.2 se tiene el módulo superior o principal de este modelo. En este caso, las plazas *Inactive*, *Connected* y *Connection* son las plazas sockets a la transición de substitución *EstabCon* que lleva al módulo de establecimiento de la conexión. Estas plazas son dibujadas nuevamente en la figura 2.7 para hacer así una asociación efectiva entre los dos módulos y en este sub módulo dichas plazas son denominadas plazas puerto.

2.3.5 CPN ML

En las secciones anteriores se nombró a CPN ML como una herramienta de ayuda para definir *color sets* (tipos de datos), variables y expresiones dentro de las Redes de Petri Coloreadas. En esta sección se presenta de una manera formal este

lenguaje. CPN ML es un lenguaje de programación con un paradigma de carácter funcional. Está basado en el lenguaje SML (Standard ML) [7]. CPN ML integra las características del lenguaje SML añadiendo los elementos necesarios para realizar declaraciones de *color sets,* variables y otros elementos que son característicos de los modelos CPN.

La declaración de variables y *colour sets* (tipos de datos) es una de las características más importantes del lenguaje CPN ML pues define los tipos de datos que son soportados por las plazas de un modelo así como también las variables que pueden fluir a través de los arcos de una red.

```
colset DATA = string;
colset NO   = int;
colset NOxDATA  = product NO * DATA;
colset DATAPACK = record seq:NO * data:DATA;
colset ACKPACK  = NO;
colset PACKET   = union Data:DATAPACK + Ack:ACKPACK;
colset RESULT   = with success | failure | duplicate;
```

Figura 2.8. Declaración de *colour sets* en CPN ML [1].

La figura 2.8 muestra la definición de *colour sets* o tipos de datos para un modelo CPN [1]. CPN ML tiene tipos de datos predefinidos como *string*, *int*, *real* y *bool*. Los *colour sets* que se construyan, pueden apuntar a uno de estos tipos de datos predefinidos (tal es el caso de los *colour sets DATA* y NO que se definieron en la figura 2.8, o bien pueden ser el resultado de aplicar alguna operación entre otros *colour sets* ya creados. Estas operaciones provienen de la teoría de conjuntos puesto que estos *colour sets* o tipos de datos pueden ser vistos como conjuntos. Por ejemplo, se puede crear un conjunto *NOxDATA* que sea el producto de cartesiano entre el conjunto *NO* y el conjunto *DATA* (ver figura 2.8).

Además de los *colour sets*, también se pueden definir variables que tengan como tipo de dato alguno de estos colour sets. Esto servirá para definir los datos que fluirán a través de los arcos de la red y para la evaluación de dichos datos (o tokens) en funciones y expresiones.

```
var n, k    : NO;
var d, data : DATA;
var pack    : PACKET;
var res     : RESULT;
```

Figura 2.9. Declaración de variables mediante CPN ML [1].

Para definir los tokens dentro de una plaza en el marcado inicial de una red, el lenguaje utiliza dos operadores particulares ++ y ‘. El primero de estos sirve para concatenar los tokens contenidos dentro de una plaza (pues la representación de los tokens dentro de una plaza es visto como un conjunto) y el segundo operador ‘ sirve para determinar cuántos elementos de ese tipo hay dentro de dicha plaza. La figura 2.10 contempla un ejemplo de lo anteriormente descrito. La plaza A con *color set* INT contiene un elemento con valor 100, dos elementos con valor 300 y otros dos elementos con valores igual a 500 y 700 respectivamente.

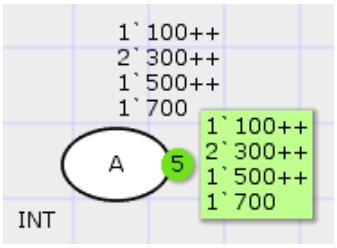

Figura 2.10. **Colocación de tokens en una plaza.**

También se pueden definir valores constantes o fijos denominados *val*. La figura 2.11 muestra un ejemplo de ello. Se declara un valor denominado *AllPackets* que contiene un conjunto de tokens determinados. Estos tokens son concatenados y

contabilizados utilizando los operadores ++ y ‘ descritos previamente. Este valor puede ser utilizado para ser asignado en plazas, ser utilizado en expresiones o incluso ser utilizado en funciones.

```
val AllPackets = 1`(1,"COL") ++ 1`(2,"OUR") ++ 1`(3,"ED ") ++
                 1`(4,"PET") ++ 1`(5,"RI ") ++ 1`(6,"NET");
```

Figura 2.11. Declaración de valores fijos en CPN ML [1].

CPN ML soporta el manejo de expresiones tal cual se realiza en el lenguaje SML. Estas expresiones pueden ser utilizadas en guardas, arcos y funciones. Por ejemplo, como se apreció en el modelo básico de comunicación emisor – receptor con manejo de reconocimientos o ACKs presentado en la Sección 2.3.2 (ver Figura 2.4) las inscripciones de los arcos pueden poseer expresiones más complejas que variables. En los arcos de los modelos CPN pueden colocarse expresiones complejas en CPN ML que determinen las salidas de una transición. Para ejemplificar esto, tomaremos la parte correspondiente a la recepción de paquetes de dicho modelo.

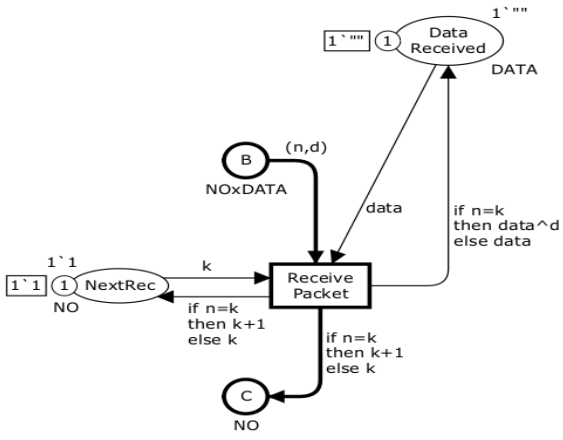

Figura 2.12. Ejemplo de uso de expresiones en CPN ML [1].

En la figura 2.12 se muestra una parte del modelo presentado en la Sección 2.3.2 que corresponde a la recepción de paquetes. En este modelo se manejan tokens

estructurados como pares ordenados en los cuales la variable *n* representa el número de secuencia del dato y la variable *d* representa el dato en sí, es decir su carga útil.

Cuando se da la ocurrencia de la transición *ReceivePacket*, los valores de los tokens a las plazas de salida estarán determinados por las expresiones de los arcos de salida. Por ejemplo, el arco que va de *ReceivePacket* a la plaza *NextRec* tiene la siguiente indicación: si el número de secuencia *n* recibido es exactamente igual al token *k* que tenía almacenado la plaza *NextRec* entonces la plaza *NextRec* tendrá ahora un token con valor $k + 1$. Otro ejemplo es el arco que va desde *ReceivePacket* hasta la plaza *DataReceived*. La expresión que contiene dicho arco indica: si el número de secuencia *n* es exactamente igual a *k*, el dato recibido *d* va a ser adjunto al conjunto de tokens *data* que almacena la plaza *DataReceived.*

El lenguaje CPN ML también soporta la declaración de funciones. Las funciones pueden ser llamadas desde guardas, expresiones de los arcos y marcados iniciales. Las funciones de este lenguaje son similares a las funciones y métodos de los lenguajes programación convencionales. Tomando como ejemplo el módulo de recepción de la figura 2.12, se pueden crear par de funciones que encapsulen las expresiones de los arcos y así, estas funciones sean invocadas cuando se da la ocurrencia de la transición *ReceivePacket.* La figura 2.13 ilustra la definición de funciones y su correspondiente llamada desde los arcos de salida de la transición *ReceivePacket*.

```
fun UpdSeq (n,k) = if n=k
                   then k+1
                   else k;

fun AddData (data,d,n,k) = if n=k
                           then data^d
                           else data;
```

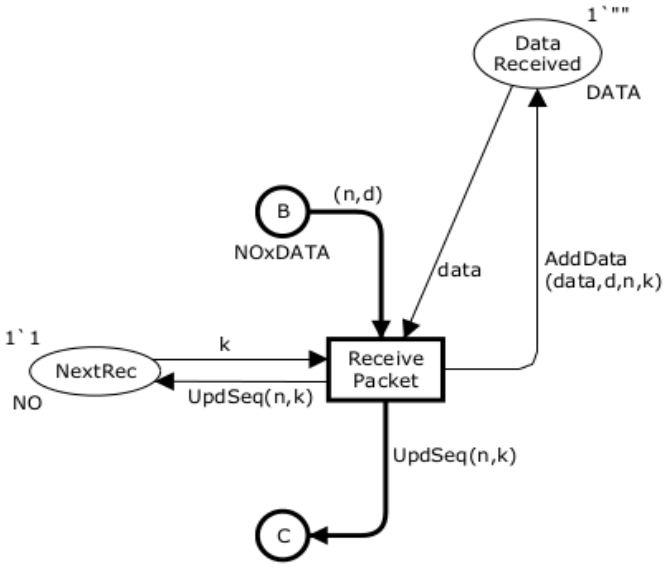

Figura 2.13. Ejemplo de uso de funciones en CPN ML [1].

Por su carácter de lenguaje de programación funcional, CPN ML incluye otras características importantes que son derivadas del lenguaje SML, como por ejemplo el uso de llamadas recursivas. Sin embargo, el alcance de esta sección pretendió solo ejemplificar los elementos básicos más utilizados que son propios de CPN ML para el manejo de los modelos CPN. En [7] es posible encontrar la definición de SML que incluye una explicación de características del lenguaje más complejas para construir así funciones y expresiones de mayor tamaño.

2.4 Análisis mediante Grafos de Estado (*State Space*)

La técnica del Grafo de Estado (o Grafo de Ocurrencias) radica en poder investigar de una manera analítica las propiedades de comportamiento de los sistemas que son modelados a través de las Redes de Petri Coloreadas. En esta sección se presenta una introducción a los Grafos de Estado, a sus propiedades de comportamiento, a la estructura derivada llamada Componentes Fuertemente Conectados (*Strongly Connected Components*) y finalmente se presentan las limitaciones que pueden poseer los Grafos de Estado.

2.4.1 Introducción a los Grafos de Estado

El procedimiento básico para el análisis de modelos a través de Grafos de Estado (*State Space)* radica en poder computar todos los estados alcanzables o marcados de un modelo y los eventos que producen dichos cambios de estado. Esto se representa mediante un grafo dirigido en el cual los nodos representan el conjunto de marcados (estados) alcanzables por el modelo CPN y los arcos de dicho grafo representan las ocurrencias de las transiciones que producen los cambios de un estado a otro.

El método del Grafo de Estado para el análisis de modelos CPN provee varias ventajas [22]: los Grafos de Estado pueden ser construidos automáticamente, lo que permite el análisis y verificación automatizada del comportamiento del sistema modelado. Esta construcción automatizada se lleva a cabo mediante soluciones de software como CPN Tools [5] que es el software utilizado en el presente trabajo. Además, un Grafo de Estado incluye mucha información sobre el comportamiento de un sistema que permite contestar a un gran conjunto de preguntas de análisis y verificación.

Para ilustrar la estructura gráfica de un Grafo de Estado y la relación que guarda con el modelo CPN del cual se genera el mismo, se utilizará como ejemplo el problema de "La Cena de Los Filósofos" (*Dinning Philosophers Problem)* [33]. Este es un problema clásico de la computación, formulado originalmente por Edsger Dijkstra en 1965, diseñado para ilustrar la sincronización entre procesos.

El problema de "La Cena de los Filósofos" consiste en 5 filósofos alrededor de una mesa circular. En el medio de esta mesa está la comida. A cada lado de los filósofos se encuentra un par de palillos chinos. Cada filósofo alterna entre pensar y comer. Para comer un filósofo necesita dos palillos chinos y solo puede utilizar los que tiene a cada lado de él. La compartición de los palillos previene que dos filósofos vecinos puedan comer al mismo tipo.

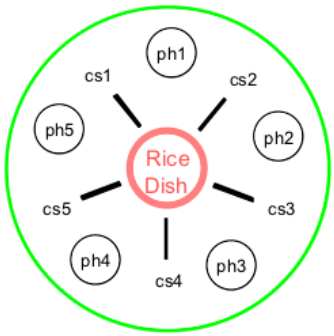

Figura 2.14. Problema de "La Cena de los Filósofos" [33].

La figura 2.15 muestra el modelo CPN de este problema. El *color set* PH representa el conjunto de filósofos, mientras que el *color set* CS representa el conjunto de palillos chinos. La función *Chopsticks* mapea a cada filósofo con los palillos chinos que se encuentran a lado de él.

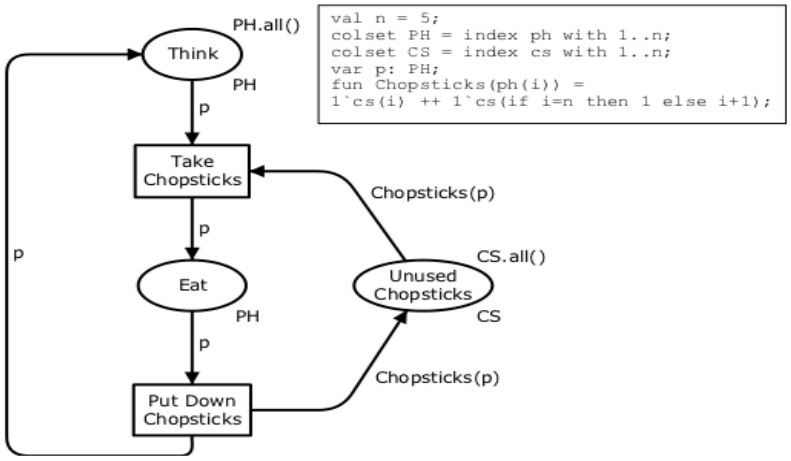

Figura 2.15. Modelo CPN de "La Cena de los Filósofos" [33].

A continuación, se muestra el Grafo de Estado generado. Cada nodo representa un estado en particular mientras que cada arco representa la ocurrencia de una transición que produce el cambio de estado.

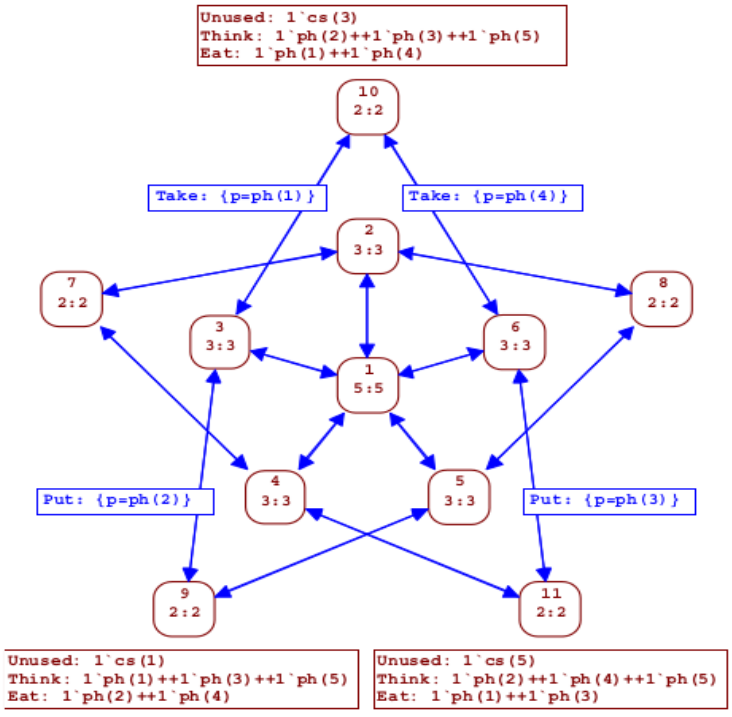

Figura 2.16. Grafo de Estado de "La Cena de los Filósofos" [33].

Cada nodo del grafo tiene un número de identificación en el tope. También, en cada nodo, hay dos números separados por dos puntos (":") que representan el número de arcos de entrada y de salida respectivamente. El nodo situado en el centro del grafo es el marcado inicial (al que hemos denominado M_0 en secciones anteriores) y tiene el número de identificación 1.

En algunos nodos del grafo se muestra un cuadro con información adicional acerca del marcado de la red para el estado correspondiente. Por ejemplo, en el estado o marcado 9 la plaza de la red *Unused* tenía un token (un palillo chino sin ser utilizado), la plaza *Think* poseía tres tokens (tres filósofos pensando) y la plaza *Eat* poseía dos tokens (dos filósofos comiendo).

Los arcos del grafo, que modelan las transiciones que producen los cambios en el marcado de la red, en este caso son bidireccionales. Es decir, entre cada par de nodos a y b del grafo existe tanto el arco (a, b) como el arco (b, a). Vale acotar que esto es un caso particular para este modelo y no es una regla en todos los grafos de estado. En algunos arcos, fue dibujada una caja que muestra la transición de la red que representa dicho arco y muestra el estado de la variable que fue procesada por la transición cuando se dio el cambio de estado. Por ejemplo, para que la red cambie del marcado 10 al marcado 3 tiene que darse la ocurrencia de la transición *Take* tomando la variable p un valor igual a $ph(1)$.

2.4.2 Propiedades de comportamiento

El Grafo de Estado de un modelo CPN permite analizar y verificar un conjunto de propiedades concernientes al comportamiento de un modelo. Ejemplos de estas propiedades incluyen el número mínimo y máximo que puede llegar a albergar una plaza, los estados en los cuales el sistema puede finalizar, o verificar que el

sistema siempre sea capaz de alcanzar cierto estado. Esta sección introduce a un conjunto de propiedades estándares de comportamiento de los Grafos de Estado. A nivel de herramientas muchas estas propiedades pueden ser comprobadas automáticamente por el software CPN Tools [5] mediante la impresión de un reporte del Grafo de Estado (*state space report*) y adicionalmente mediante funciones predefinidas en lenguaje CPN ML provistas por CPN Tools.

Alcanzabilidad (*Reachability)*

Las propiedades de alcanzabilidad (*reachability properties*) son aquellas concernientes a determinar cuándo un marcado M' puede ser alcanzado desde un marcado M. Un marcado M' es alcanzable desde un marcado M si y solo si existe un camino en el grafo de estado que va desde el nodo que representa al marcado M hasta el nodo que representa al marcado M'.

La implementación de CPN ML en CPN Tools, entre otras funciones, provee una función denominada *Reachable* que toma como parámetros dos valores *n* y *n'* que identifican dos nodos dentro del Grafo de Estado y retorna verdadero o falso en función de si el nodo *n'* es alcanzable desde el nodo *n*. A partir de allí, pueden construirse expresiones más complejas para determinar alcanzabilidad a un estado o a un conjunto de estados específicos proveyendo así al usuario un conjunto de opciones para estudiar las propiedades de alcanzabilidad en un modelo.

Acotación (*Boundedness*)

Las propiedades de acotamiento (*boundedness properties*) especifican cuantos y cuales tokens, una plaza puede llegar a albergar tomando en cuenta todos los marcados alcanzables por un modelo. De esta manera se puede permitir estudiar hasta

cuantos elementos podrían estar contenidos en una parte de un sistema, o por el otro lado, cual es el número mínimo de elementos que puede poseer un sistema. Como ejemplo de esta propiedad, se muestra una parte del reporte del Grafo de Estado (ver figura 2.17) del modelo CPN descrito en [1, Ch. 7, Sec. 1, pp. 152-153]. Este modelo es una versión con mayor nivel de abstracción del modelo de comunicación emisor – receptor con manejo de reconocimientos (ACKs) descrito en la Sección 2.3.2 del presente trabajo.

Best Integer Bounds	Upper	Lower
PacketsToSend	6	6
DataReceived	1	1
NextSend, NextRec	1	1
A, B, C, D	3	0
Limit	3	0

Figura 2.17. Ejemplo de estudio de las propiedades de acotamiento [1].

En la figura 2.17 se colocan las plazas de la red en la primera columna, en la segunda columna se coloca el número máximo de tokens que pueden tener cada una de estas plazas y la tercera columna muestra el número mínimo de tokens que llegan a poseer. Por ejemplo, la plaza *PacketsToSend* modela los tokens que son transmitidos a un receptor. En este, caso el modelo que se planteó configuró la red de manera que *PacketsToSend* nunca alterara los tokens que contiene de manera que estén disponibles para envíos y posibles retransmisiones.

De esta manera, con el estudio de esta propiedad, los modeladores de sistemas pueden controlar el estado de cada de una de las plazas en función de analizar si el número de tokens y los mismos tokens que una plaza llega a poseer a través de todos los marcados es el esperado.

Localidad (*Home*)

La propiedad de localidad radica en el estudio de los marcados locales (*home markings*). Un marcado local M_{home} es un marcado que puede ser siempre alcanzado por el resto de los marcados alcanzables [1]. El reporte del Grafo de Estado (*state space report*) generado por la herramienta CPN Tools [5] permite saber que marcados son locales dentro de un Grafo de Estado.

Mediante esta propiedad, un usuario modelador de un sistema podría determinar, si estando en cualquier estado del sistema, se puede ser siempre capaz, mediante un conjunto de eventos determinados, llegar a un estado del sistema determinado.

Vivacidad (*Liveness*)

Las propiedades de vivacidad (*liveness properties*) permiten identificar en un Grafo de Estado cuales son los marcados muertos y cuales con las transiciones vivas dentro de un modelo CPN. CPN Tools [5] imprime en su reporte del grafo de estado las propiedades de vivacidad de un modelo.

Un marcado muerto (*dead marking*) es un marcado para el cual no existen elementos de asociación habilitados, es decir, transiciones habilitadas. Por lo tanto, un sistema terminaría su ejecución en dichos marcados muertos pues no existiría algún evento habilitado capaz de producir algún cambio de estado. Los marcados muertos pueden ser o no esperados en un sistema. Esto radica en que la red tenga una terminación esperada en su ejecución, o en un caso contrario, que en el sistema siempre pueda darse la ocurrencia de un evento que produzca un cambio de estado que es lo que se denominaría una red completamente viva.

Una transición *t* está viva (*live transition*) si, comenzando desde cualquier marcado alcanzable, se puede encontrar siempre al menos una secuencia de ocurrencias que contenga a la transición *t*. Esto permite saber si será posible que pueda ocurrir al menos una vez un evento que esté siento modelado por la transición *t*.

2.4.3 Componentes Fuertemente Conectados

La generación del Grafo de Estado (o Grafo de Ocurrencias) de un modelo CPN es seguido por la generación del Grafo de Componentes Fuertemente Conectados (*Strongly Connected Components*, SCC Graph) [1]. El conjunto de Componentes Fuertemente Conectados es un grafo reducido que se deriva del Grafo de Estado y permite determinar ciertas propiedades de comportamiento de un modelo.

El Grafo de Componentes Fuertemente Conectados es obtenido realizando una división disjunta de los nodos del Grafo de Estado agrupándolos en componentes $s_0,\ s_1, s_2, \ldots$ tal que para cada uno de estos componentes se cumple la siguiente regla: Todos los nodos contenidos dentro de un componente deben ser mutuamente alcanzables. Es decir, para cada par de nodos *a* y *b* del Grafo de Estado que estén contenidos en una misma componente, debe existir un camino dentro del componente que vaya desde el nodo *a* hasta el nodo *b* y viceversa.

En la figura 2.18 se muestra un ejemplo de un Grafo de Estado que posee 10 nodos (marcados) $M_0, M_1, \ldots, M_9$ y 16 arcos. A su vez, se muestra la asociación de estos nodos en distintos componentes $s_0,\ s_1, \ldots, s_4$ tal que todos los nodos dentro de una componente son mutuamente accesibles. Estos componentes forman en sí el Grafo de Componentes Fuertemente Conectados.

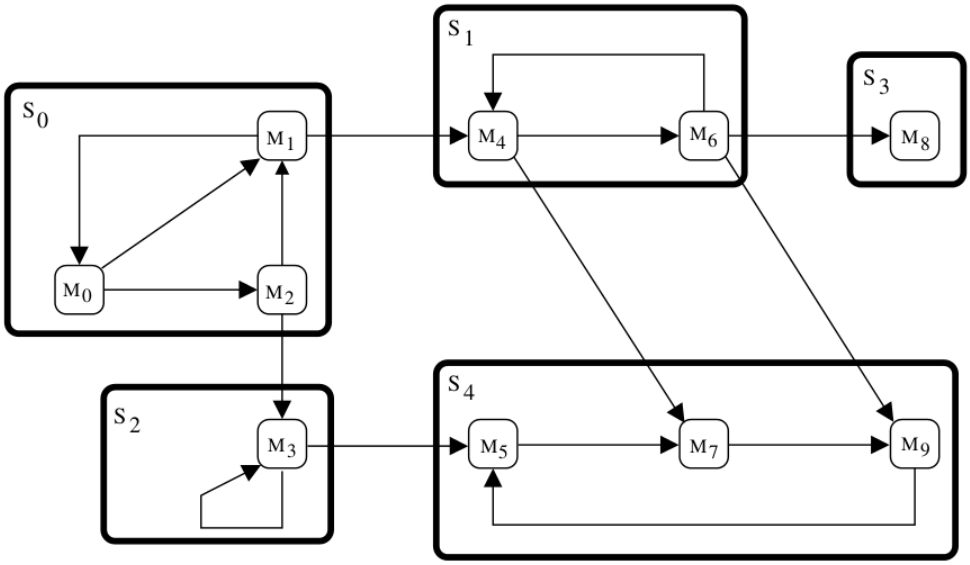

Figura 2.18. Ejemplo de Componentes Fuertemente Conectados [1].

2.4.4 Limitaciones de los Grafos de Estado

Pese a ser una técnica con varias ventajas en cuanto al análisis de sistemas se refiere, la principal desventaja del método del Grafo de Estado (o Grafo de Ocurrencias) es que cuando este grafo se hace muy grande se pueden generar problemas de cálculo en términos de consumo de tiempo y memoria. Este problema es lo que se denomina el problema de explosión del estado (*state explosion problem*) [1]. Cuando los sistemas que se modelan llegan a tener un número astronómico de estados los recursos de cómputo pueden llegar a ser limitados para realizar el cálculo de Grafo de Estado.

Sin embargo, para poder estudiar de manera analítica un modelo CPN que pueda llegar a tener un número grande de estados, en el que se torne complejo el cómputo del Grafo de Estado, como alternativa se pudiese generar un sub-grafo parcial derivado del Grafo de Estado que permita determinar ciertas propiedades específicas del modelo. No obstante, con un grafo parcial no se pueden responder

preguntas como por ejemplo el acotamiento máximo para cada una de las plazas de un modelo, pero si se pueden responder preguntas más específicas como por ejemplo: hallar al menos un marcado muerto dentro del modelo.

Para lidiar con las limitaciones de cómputo que pueden estar presente sobre Grafos de Estado de gran tamaño se han desarrollado varias técnicas de reducción. Ejemplos de estas técnicas de reducción están descritos en [1]. Mediante estas técnicas es posible todavía analizar ciertas propiedades de comportamiento del modelo a través de una representación reducida del Grafo de Estado para así estudiar analíticamente el sistema modelado.

Otra solución de reducción para el análisis de los modelos es la técnica de reducción de Grafos de Estado a Máquinas de Estado Finito [3] [4]. Esta es la técnica utilizada en el presente trabajo y es una solución para realizar un análisis del sistema modelado transformado el Grafo de Estado asociado al modelo a un autómata que logra representar de una manera minimizada el comportamiento del sistema.

2.5 CPN Tools

En esta sección se describe el software CPN Tools. Este es el utilizado en el presente trabajo para la manipulación de modelos CPN donde fue embebida la aplicación Java/PROSEGA. Además, se da una breve descripción de Design/CPN que fue el software predecesor de CPN Tools. Luego, se describen las partes básicas y fundamentales de CPN Tools: el simulador, el editor gráfico (o GUI) y el protocolo de comunicación que hace posible la interacción entre el simulador gráfico y el simulador.

2.5.1 Introducción a CPN Tools

CPN Tools [5] es un software para la edición, simulación y análisis de los modelos basados en las Redes de Petri Coloreadas [1] [32]. CPN Tools fue originalmente desarrollado por el Grupo CPN [34] de la Universidad de Aarhus en Dinamarca entre los años 2000 y 2010. Los arquitectos principales del software han sido Kurt Jensen, Søren Christensen, Lars M. Kristensen, y Michael Westergaard. En el año 2010, la curaduría y mantenimiento del software fue transferida al Grupo AIS [35] de la Universidad Tecnológica de Eindhoven en los Países Bajos.

Figura 2.19. Logotipo del software CPN Tools (versión 4.0.1) [5].

La última versión disponible para CPN Tools es la versión 4.0.1 (lanzada en Febrero de 2015). Esta es la versión que fue utilizada en el presente trabajo. Esta puede ser obtenida en [36]. Para Distribuciones Linux y para los sistemas operativos Mac de Apple la última versión de CPN Tools desarrollada fue la versión 2.3.5. Por lo que se recomienda a usuarios de Linux y Mac que descarguen la versión de Windows ya es que la más actualizada y la utilicen dentro de alguna máquina virtual.

El software CPN Tools, en su forma más básica, comprende de dos componentes principales:

- Una interfaz gráfica (o editor gráfico) para el usuario, provista con varias opciones para la creación y edición de modelos CPN permitiendo agregar y modificar elementos (plazas, transiciones, arcos, etc.). Además, esta interfaz

de usuario (o GUI) permite declarar código en lenguaje CPN ML para la declaración de *color sets*, variables, expresiones y funciones.

- Un simulador que se encarga de realizar las labores de chequeo de correctitud y de sintaxis de los modelos, generación de código asociado al modelo, y simulaciones del modelo. Se encarga de enviar a la interfaz gráficas los mensajes correspondientes para proveer *feedback* al usuario.

Además de estos dos componentes principales, a partir de la versión 4.0 de CPN Tools, fue incluido un servidor de extensiones que permite que aplicaciones externas se puedan conectar tanto al simulador como al editor gráfico para así agregar funcionalidades adicionales al software o para que las funcionalidades de CPN Tools puedan ser sean integradas con otras soluciones.

El editor gráfico, el simulador y el servidor de extensiones corren en el sistema operativo como procesos independientes. Estos se comunican entre sí mediante un protocolo privado de CPN Tools denominado BIS. Las secciones 2.5.3 – 2.5.5 describen el funcionamiento del editor gráfico, del simulador y una introducción al protocolo de comunicación BIS. Una descripción sobre el servidor de extensiones está contenida en la Sección 2.8 del presente trabajo.

2.5.2 Representación de modelos CPN en el computador

Los modelos utilizados por CPN Tools son almacenados como archivos de extensión ".cpn". Este tipo de archivos está basado en PNML (*Petri Net Markup Language)*. El estándar PNML [41] [42] (definido en la norma ISO/IEC 15909) describe la sintaxis para la representación de Redes de Petri de Alto Nivel.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE workspaceElements PUBLIC "-//CPN//DTD CPNXML 1.0//EN"
"http://www.daimi.au.dk/~cpntools/bin/DTD/5/cpn.dtd">

<workspaceElements>
  <generator tool="CPN Tools"
             version="1.5.41"
             format="5"/>
  <cpnet>
    <globbox>
      <color id="ID697059">
        <id>U</id>
        <index>
          <ml>1</ml>
          <ml>10</ml>
          <id>u</id>
        </index>
      </color>
```

Código 2.1. Extracto de un de archivo de extensión .cpn.

Específicamente, para la descripción de la estructura de los archivos de modelos CPN utilizados por CPN Tools, se utiliza la definición de tipo de documento (o DTD) contenida en [43]. Este DTD a su vez cumple con las indicaciones del estándar PNML. A través del editor gráfico, CPN Tools se encarga de crear, editar y guardar representaciones de modelos CPN que siguen los estándares mencionados de manera que para el usuario sea transparente el formato de dichos archivos.

2.5.3 Editor gráfico

El editor gráfico (o GUI) de CPN Tools es la cara visible del software para el usuario (ver figura 2.20). Consta de dos paneles (izquierdo y derecho). El panel izquierdo provee distintas opciones para la creación, manipulación y análisis de los modelos CPN en conjunto con otras opciones misceláneas, mientras que el panel derecho, un poco más grande, provee el espacio para la visualización de los modelos CPN.

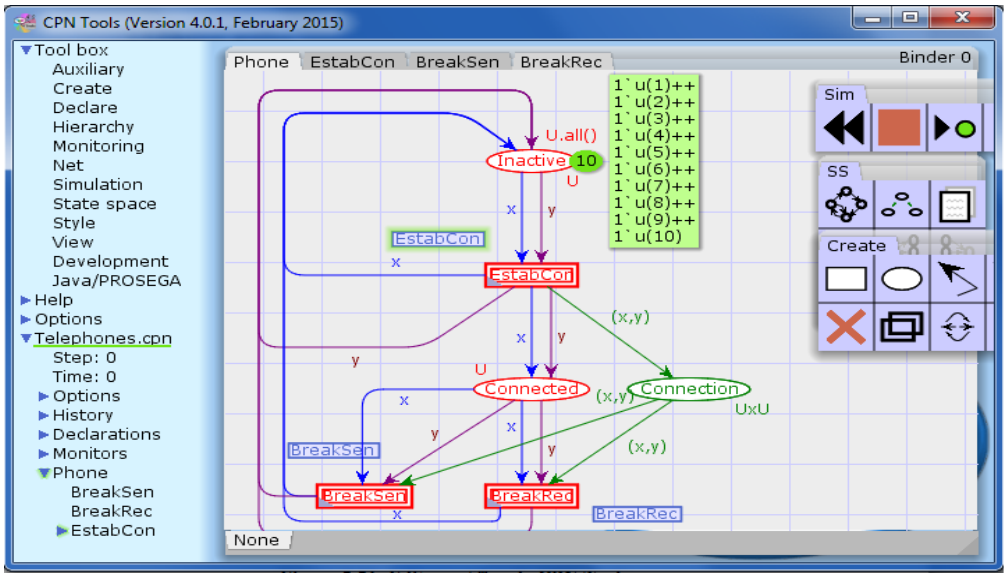

Figura 2.20. Editor gráfico de CPN Tools.

El panel izquierdo comprende en primer lugar un conjunto de cajas de herramientas que pueden ser arrastradas al panel derecho de la aplicación (ver figura 2.20) para editar las redes que se estén modelando. Es así entonces como se proveen opciones como simulación, cálculo del *State Space* y opciones para agregar elementos de red (arcos, transiciones, plazas, etc.).

Adicionalmente, el panel izquierdo provee un espacio correspondiente al modelo CPN que esté cargado en la solución en un momento dado. El software permite ver la estructura de la red (en cuanto a páginas se refiere) y permite además realizar declaraciones en lenguaje CPN ML concernientes al modelo. Es así entonces como se pueden declarar *color sets*, variables, valores constantes y/o funciones que vaya a utilizar el modelo.

El panel derecho, de mayor tamaño, es utilizado para visualizar los modelos que estén cargados en el software y de esta manera poder editarlos. Permite además el manejo de múltiples pestañas cada una identificando a una página del módulo. Es

posible también a través de esta interfaz ir visualizando la simulación que se realiza a la red. Otra característica de este panel es que permite graficar el Grafo de Estado del modelo CPN que se esté desarrollando a través del uso de la paleta de herramientas encargada del manejo del Grafo de Estado (*State Space Tool*).

2.5.4 Simulador

El simulador es el proceso maestro del software. Cuando se abre un modelo en CPN Tools, el editor gráfico envía al simulador la representación XML utilizada en el modelo (transparente al usuario) y este se encarga de autogenerar un código en lenguaje SML asociado que permite la simulación del modelo.

Antes de realizar el proceso de generación de código en SML, el simulador de CPN Tools también realiza un proceso de chequeo y análisis sintáctico para revisar la correctitud del modelo. Si existe algún error dentro del modelo, el simulador envía al editor gráfico un *feedback* que este último se encarga de mostrar al usuario. De esta manera los distintos elementos del editor gráfico pueden tener sombreados verdes, amarillos o rojos que informarán de, correctitud en las declaraciones, de advertencias o de errores respectivamente. Por ejemplo: un error de declaración en CPN ML mostrado mediante un sombreado rojo.

Después del chequeo sintáctico del modelo cargado en el editor gráfico y la posterior auto-generación del código SML asociado al modelo por parte del simulador, el usuario puede, mediante la paleta de opciones de simulación, realizar simulaciones sobre el modelo. Por ejemplo, cuando el usuario selecciona la opción de ir a un paso adelante en la simulación (esto es, ejecutar la ocurrencia de una transición habilitada dentro del modelo) el editor gráfico envía al simulador la solicitud para que se ejecute el código SML correspondiente que maneja la ejecución de dicho paso de la simulación. El simulador ejecutará dicho paso de la simulación y

envía al editor gráfico el *feedback* correspondiente para que cambie el modelo CPN a un nuevo marcado de manera de mostrar el resultado de la ejecución del paso al usuario.

El simulador también está encargado de realizar la generación del Grafo de Estado (*State Space* o Grafo de Ocurrencias). El editor gráfico provee opciones al usuario para el cálculo y manipulación del Grafo de Estado y este editor gráfico es quien redirige dichas solicitudes de cálculo y manipulación al simulador para que este último resuelva las peticiones y retorne el resultado correspondiente para ser mostrado finalmente al usuario. Así entonces es como se provee al usuario para hacer consultas (o *queries*) en CPN ML del grafo, dibujar ciertos nodos del grafo, imprimir reportes del grafo para realizar un chequeo de sus propiedades, o incluso generar el conjunto de Componentes Fuertemente Conectados (SCC Graph).

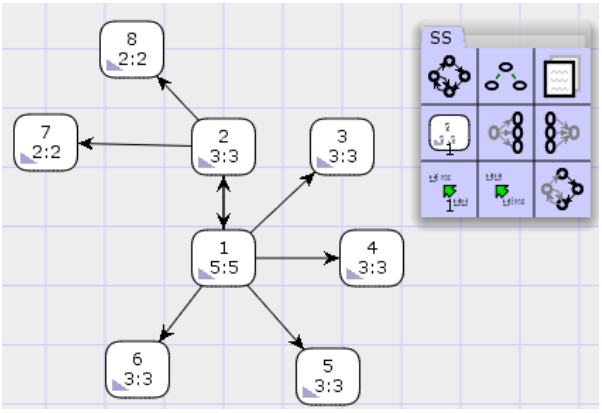

Figura 2.21. Generación de un Grafo de Estado mediante CPN Tools.

2.5.5 Protocolo de comunicación

Esta arquitectura de procesos, editor gráfico y simulador, de CPN Tools está soportada mediante el uso de un canal por el cual el editor gráfico y el simulador intercambian mensajes de solicitud y respuesta para la efectiva manipulación de los

modelos CPN. Este canal de comunicación es un protocolo propietario desarrollado por el Grupo CPN específicamente para CPN Tools.

El protocolo de comunicación es denominado BIS (*Boolean - Integer - String)* pues la carga útil de los paquetes de CPN Tools es codificada mediante una lista de booleanos, una lista de enteros y una lista de cadenas de caracteres. Además de estas tres listas, un paquete también posee un *opcode.* El *opcode* (o código de operación) es un número entero cuya función es determinar que se realizará con el paquete, como por ejemplo, cuál será el patrón de comunicación a utilizar (véase Sección 2.8.1).

La mayoría de los paquetes también poseen un comando (*command*). Este es un número que determina el tipo de tarea que se pretende resolver con el envío del paquete (manejo de simulación, edición de la red, etc.) que contenga dicho comando. A continuación, se enumeran los distintos tipos de comando [44]:

Tabla 2.1. Tipos de comandos del protocolo de CPN Tools.

Comando (cmd)	Descripción
Bootstrap (cmd = 100)	Arranque de CPN Tools.
Miscellaneous (cmd = 200)	Funciones misceláneas.
Compile Declarations (cmd = 300)	Compilación de expresiones en CPN ML.
Syntax Check Net (cmd = 400)	Chequeo de sintaxis de los modelos.
Monitor (cmd = 450)	Funciones de monitoreo.
Simulate (cmd = 500)	Funciones de simulación.
State Space (cmd = 800)	Funciones de manejo del Grafo de Estado.
Extensions (cmd = 10000)	Manipulación de extensiones.

Por último, los paquetes pueden contener un subcomando (*subcommand*). El subcomando se encarga de determinar la tarea específica que realizará el paquete

(evaluar una expresión en CPN ML, crear un arco, ejecutar una transición, etc.). Este subcomando en la práctica es embebido dentro de la lista de enteros del paquete.

Finalmente, se muestra un ejemplo de la estructura que generalmente posee un paquete del protocolo BIS de CPN Tools (ver figura 2.22). Este está compuesto del *opcode*, el comando (*command*), subcomando (*subcmd)* y las listas de booleanos (*blist)*, enteros (*ilist)* y strings (*slist)*.

Opcode	*command*
$blist = b_1, b_2, b_3, \dots, b_n$	
$ilist = subcmd, i_2, i_3, \dots, i_m$	
$slist = s_1, s_2, s_3, \dots, s_k$	

Figura 2.22. Estructura general de un paquete del protocolo BIS.

Una consideración particular es que esta es la estructura común tanto en paquetes de solicitud como en paquetes de respuesta. Sin embargo, en el caso de los paquetes de respuesta, en vez de estar colocado el subcomando (*subcmd*) en el primer elemento de la lista de enteros, se tiene una marca denominada *TERMTAG*.

La marca *TERMTAG* sirve para determinar si el paquete de solicitud previamente enviado pudo ser procesado exitosamente y el contenido del presente paquete de respuesta es el esperado. Si la marca *TERMTAG* es exactamente igual a 1 entonces hubo un error en el procesamiento del paquete de solicitud y no se produjo el resultado esperado en el paquete de respuesta. Caso contrario, el paquete de solicitud fue procesado exitosamente.

En [44] se tiene una documentación de la estructura que debe tener cada paquete en función de la tarea o procedimiento que se requiera realizar y también la

estructura de los paquetes de respuesta. A continuación, se coloca como ejemplo de dicha referencia la función de ejecutar la ocurrencia de una transición que envía el editor gráfico de CPN Tools al simulador (ver figura 2.23).

```
Extra call parameters:
  blist= nil
  ilist= transition-instance
  slist= transition-id
Return value:
  blist= nil
  ilist= TERMTAG=1
  slist= sim-step,sim-time,sim-why-stopped-message
```

Figura 2.23. Estructura de la función de la ocurrencia de una transición [44].

El comando utilizado en esta función es 500 (funciones de simulación) y el subcomando es el 12. El subcomando debe ir colocado de primero antes de los demás enteros pasados como parámetros. La lista de booleanos no contiene parámetros para esta llamada, la lista de enteros necesita (además del subcomando) un valor denominado *transition-instance* que hace mención a la instancia de la transición seleccionada y la lista de strings necesita el identificador de la transición que será ejecutada (*transition-id*).

El paquete de respuesta contiene lo siguiente: la lista de booleanos está vacía, la lista de enteros tiene la marca *TERMTAG* que determina si la función fue computada correctamente o no, y la lista de strings tiene tres valores concernientes a información de la simulación. Después de la respuesta, el editor gráfico debe (si TERMTAG es distinto de 1) actualizar el modelo desplegado en la interfaz después de la ocurrencia de la transición ejecutada.

Como conclusión, este protocolo es vital para la comunicación de los procesos de CPN Tools y como se verá posteriormente, es de gran ayuda para la comunicación entre el software Java/PROSEGA y el simulador de CPN Tools.

2.6 Java

Java [45] [46] es un lenguaje de programación de propósito general con un paradigma orientado a objetos. Desde su creación a principios de la década de los 90, ha sido ampliamente utilizado en diversos proyectos a nivel mundial. Hoy en día millones de computadores y otros dispositivos de hardware ejecutan Java para el soporte de diversos sistemas y arquitecturas de distinta índole.

La importancia de Java dentro de este trabajo radica en que es el lenguaje en el cual fue desarrollado el software Java/PROSEGA. Por esta razón, es importante mencionar dentro de la presente sección las características resaltantes de este lenguaje que fueron aprovechadas por Java/PROSEGA para cumplir los objetivos del presente trabajo. Adicionalmente, se introduce a las librerías Swing/AWT (utilizadas por Java/PROSEGA para la construcción de interfaces de usuario). Por último, se da una breve descripción de la relación que guarda el lenguaje con la propia solución de CPN Tools.

2.6.1 Características principales de Java

A continuación se describen las características más resaltantes del lenguaje de programación Java:

- **Orientado a objetos:** En el lenguaje Java todo es un objeto, desde un número entero hasta la entidad que se encarga de realizar la impresión de algún otro objeto por consola. Java hereda este paradigma orientado a objetos del lenguaje C++. Sin embargo, Java profundiza dentro de sí el enfoque orientado a objetos (o enfoque OO). Una característica resaltante de este enfoque es permitir al usuario diseñar sus propias clases de objetos (con atributos y

métodos definidos por el usuario) para que estos sean instanciados y utilizados.

- **Independencia de la plataforma:** La portabilidad es una característica fundamental del lenguaje. Esto significa que programas escritos en Java pueden ser ejecutados en cualquier tipo de hardware tal como lo indica lo indica el axioma que acompaña al lenguaje ("*write once, run anywhere*"). El ente encargado de ejecutar los programas escritos en Java es la máquina virtual de Java donde esta última si se encarga de comunicarse con el hardware de la máquina utilizada.

- **El recolector de basura:** Una ventaja que posee Java frente a lenguajes como C o C++, en cuanto a limpieza de memoria se refiere, es el recolector de basura (*automatic garbage collector*). El usuario programador determina dentro del código cuando son creados los objetos (momento en que se reserva un espacio de memoria para el almacenamiento de los atributos del objeto). Cuando en un programa no existen más referencias a un objeto creado, el recolector de basura se encarga de borrar el objeto eliminándolo de la memoria del computador.

- **Manejo de concurrencia:** Java no se queda atrás en la programación multi-hilo. A partir de objetos *Thread* o de clases similares, los programadores pueden crear un ambiente de concurrencia para que no se sature el hilo principal del programa, delegando así tareas específicas a threads asíncronos al hilo de ejecución principal del programa.

2.6.2 Librerías Swing/AWT

Existen diversos frameworks y librerías para el desarrollo de interfaces de usuario (GUI, *Graphical User Interface)* mediante Java. Sin embargo, una de las librerías más utilizadas de Java para la creación de interfaces de usuario es la librería Swing en conjunto con la librería AWT. Ambas están bien documentadas en la especificación oficial del API de Java [47].

La librería AWT (*Abstract Window Toolkit)* es un kit que provee herramientas para la construcción de gráficos e interfaces de usuario. Por otra parte, la librería Swing es una biblioteca que fue lanzada posteriormente para apoyar a la librería AWT en la labor de construcción de interfaces gráficas y widgets, para aliviar así las carencias y limitaciones que AWT poseía en un primer momento. Ambas librerías vienen integradas al conjunto de clases estándar de Java. Oracle, quien es ahora la empresa a cargo de Java, provee un conjunto de tutoriales para el desarrollo de interfaces gráficas a través de Swing (y AWT), estos tutoriales pueden ser encontrados en [48].

2.6.3 Java y CPN Tools

CPN Tools [5] es una solución poderosa para la construcción y análisis de modelos CPN. El software basa su éxito en el simulador rápido y versátil que posee. Sin embargo, manipular el simulador para aprovechar o extender las funcionalidades que provee es complicado, pues CPN Tools está desarrollado en lenguaje BETA (su editor gráfico) [6] y SML (el proceso simulador) [7].

Es en este punto donde entra en juego el lenguaje Java. Los desarrolladores a cargo de CPN Tools, primero el Grupo CPN en Dinamarca y posteriormente el Grupo AIS de la Universidad Tecnológica de Eindhoven, se han encargado, en conjunto con

demás desarrolladores de la línea de investigación, y a través de distintos proyectos, de crear canales/soluciones en lenguaje Java que permiten a usuarios extender las funcionalidades de CPN Tools mediante el desarrollo de software basado en Java.

De esta manera, se evita que nuevos desarrolladores, que requieran extender las capacidades de CPN Tools o aprovechar las bondades del simulador de CPN Tools, tengan que involucrarse con el lenguaje BETA (poco conocido) o con el lenguaje SML (complejo y no diseñado para interacción gráfica con el usuario). Los desarrolladores de CPN Tools escogieron Java como esta alternativa de extensión puesto que es un lenguaje ampliamente conocido. Ejemplos relevantes de estos proyectos en Java están presentados en [8] [9] [19] [49].

De los proyectos mencionados de Java para CPN Tools tienen especial relevancia en la presente investigación los siguientes: Access/CPN [9] y el manejo de extensiones de CPN Tools [8]. El primero pretende aprovechar las bondades del simulador de CPN Tools permitiendo integrarlo con aplicaciones externas mientras que el segundo pretende integrar software basado en Java dentro del mismo CPN Tools para enriquecer así el conjunto de funcionalidades que provee. Ambas herramientas adicionales comparten y utilizan como apoyo una implementación en Java del protocolo BIS.

Es así entonces, como herramientas desarrolladas en Java permiten potenciar el abanico de funcionalidades provistos por CPN Tools y su simulador. El software Java/PROSEGA se encarga de aprovechar estas herramientas creadas mediante Java para integrarse de esta manera a CPN Tools.

2.7 Access/CPN

Access/CPN [9] es una herramienta complementaria para el software CPN Tools [5]. Fue desarrollado por Michael Westergaard (Universidad de Aarhus, Dinamarca) y Lars Michael Kristensen (Bergen University College, Noruega). Esta herramienta adicional tiene como propósito hacer posible la manipulación de modelos CPN, construidos en CPN Tools, desde ambientes externos para así, por ejemplo, utilizar técnicas de análisis no provistas por CPN Tools o para conectar los modelos CPN con software de terceros.

En esta sección se da un una introducción a la arquitectura en la cual está envuelta Access/CPN, se listan los distintos módulos que componen esta herramienta y luego se describen las características más importantes de la herramienta (el modelo de objetos, el importador de modelos que provee y la comunicación que implementa para el intercambio de mensajes con el simulador de CPN Tools). Finalmente, se hace una comparación entre Access/CPN y el software BRITNeY Suite [49].

Adicionalmente, en el anexo "A" del presente trabajo se encuentra una descripción tipo tutorial con todos los pasos que deben ser realizados para descargar y configurar la herramienta para poder así desarrollar aplicaciones que utilicen los módulos provistos por Access/CPN.

2.7.1 Introducción a la arquitectura

Para explicar la arquitectura de Access/CPN repasaremos brevemente la arquitectura de CPN Tools [5]. Este último consiste básicamente de un editor gráfico y un simulador. El editor gráfico permite la construcción interactiva de modelos CPN. Estos modelos CPN son trasmitidos al simulador quien se encarga de chequear errores sintácticos y se encarga de transformar el modelo a una representación en

código. Al ejecutar la simulación, el editor gráfico invoca el código generado y presenta los resultados gráficamente. Adicionalmente, estos modelos CPN pueden ser guardados y cargados utilizando un formato particular basado en XML (representación PNML). La figura 2.24 (sección superior de la figura) muestra el diseño descrito de la arquitectura de CPN Tools.

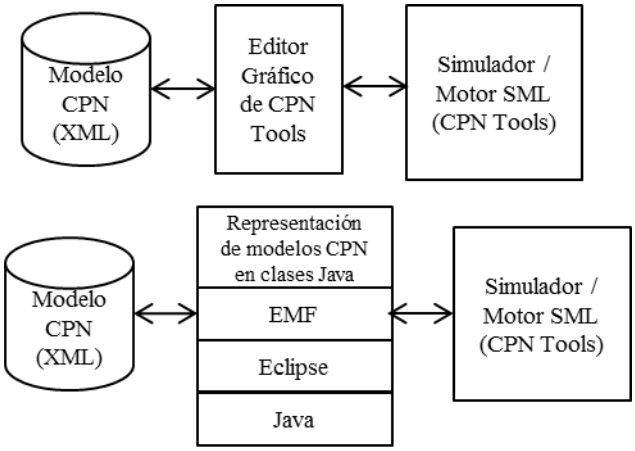

Figura 2.24. Arquitectura de CPN Tools y de Access/CPN.

En cambio, Access/CPN es una herramienta que se conecta al simulador de CPN Tools de manera directa sin tener algún tipo de interacción con el editor gráfico. De igual manera, esta herramienta es capaz de extraer los modelos CPN que estén guardados en archivos de extensión .cpn. La figura 2.24 (sección inferior de la figura) muestra la arquitectura descrita de Access/CPN, comparándola con la arquitectura de CPN Tools.

Access/CPN posee dos interfaces: una en el lenguaje Java y otra en SML [7]. La interfaz en Java consiste en una representación orientada a objetos de los modelos CPN, la habilidad para transmitir esta representación al simulador de CPN Tools y ejecutar simulaciones. También, provee una herramienta de carga que importa modelos creados en CPN Tools. Esta interfaz se apoya en gran medida del framework

EMF (*Eclipse Modeling Framework*) [55]. La interfaz en SML encapsula las estructuras de datos utilizadas en el simulador y provee una interfaz para la simulación rápida de un modelo CPN y para un análisis eficiente de los modelos. Sin embargo, esta interfaz no está diseñada para proveer interacción con el usuario, por lo que se apoya en la interfaz de Java para llevar a cabo cualquier tipo de interacción con el usuario.

2.7.2 Módulos de Access/CPN

A continuación, se presentan los módulos de Access/CPN. Estos están empaquetados en un conjunto de librerías para Java. Un usuario puede utilizar estas librerías para desarrollar sus propias aplicaciones basadas en Access/CPN.

Tabla 2.2. Módulos de Access/CPN.

Módulo	Descripción	Se necesita…
model	Provee las representaciones orientada a objetos de elementos.	Siempre
model.import	Carga modelos creados en CPN Tools.	Cuando se cargan modelos de CPN Tools.
model.export	Guarda los modelos a CPN Tools.	Cuando se exportan los modelos a CPN Tools.
engine	Provee el motor para la simulación.	Para realizar simulaciones.
engine.protocol	Representación abstracta del protocolo de bajo nivel que comunica con CPN Tools.	Para conectarse al simulador de CPN Tools.
engine.highlevel	Provee el puente entre los modelos y el *engine*.	Casi siempre.
cosimulation	Realiza una cosimulación entre los modelos CPN y los objetos en Java.	Solo si es necesario la cosimulación.
engine.proxy	Interactúa con un modelo cargado usando CPN Tools.	Raramente.
model.tests	Programas de prueba y de ejemplos.	Durante las pruebas.

2.7.3 Modelo de objetos

El modelo de objetos construido en Access/CPN tiene la función de representar los elementos de los modelos CPN con el enfoque orientado a objetos. La Red de Petri Coloreada que sea cargada desde un archivo CPN será transformada a un conjunto de clases en Java. El módulo *model* de Access/CPN contiene gran parte de las clases que componen este modelo de objetos. Access/CPN se apoya en gran medida del framework EMF [55] para lograr esta transformación de modelos CPN a representaciones orientadas a objetos.

Básicamente, se tiene en el modelo de objetos desarrollado una clase Red de Petri denominada *PetriNet* que contiene uno ó más objetos *Pages* que representan las páginas de una red. Estos objetos *Pages* pueden contener a su vez cualquier cantidad de objetos *Arcs*, *Places*, y *Transitions* que representan los arcos, plazas y transiciones respectivamente. Cada uno de estos objetos puede contener objetos *Labels* que representan las inscripciones de los elementos (nombre, tipo de plaza, expresiones de los arcos, etc.).

Los objetos *Pages* también pueden contener objetos *Instances* que corresponden a las transiciones de substitución en CPN Tools. Los objetos *Instances* contienen *ParameterAssignments* correspondientes a las conexiones puerto-sockets.

2.7.4 Importador de modelos CPN

Un modelo CPN puede ser creado de manera programada mediante Java utilizando para ello las distintas clases del modelo de objetos de Access/CPN. Sin embargo, es recomendable crear modelos utilizando el editor gráfico de CPN Tools y luego transferirlos a un ambiente orientado a objetos. Por esta razón, Access/CPN provee un importador que permite importar modelos creados con CPN Tools

contenidos en formato .cpn. Este importador realiza una labor de "*parsing"* del modelo contenido en PNML al modelo de objetos de Access/CPN.

2.7.5 Comunicación con el simulador de CPN Tools

Para entender cómo se comunica Access/CPN con el simulador de CPN Tools, se reseña el procedimiento de comunicación entre el editor gráfico de CPN Tools con el simulador. El editor de CPN Tools se comunica con el simulador utilizando un protocolo propietario, que es una implementación particular del protocolo RPC (*Remote Procedure Call*). El protocolo envía paquetes sobre un flujo TCP/IP en el formato BIS (*Boolean – Integer – String*) (véase Sección 2.5.6).

Los paquetes contienen un número entero que representa el tipo de paquete (*opcode)* y algunos contienen par de números enteros adicional para indicar la función específica a ejecutar (*command* y *subcommand*). Los comandos deben ser combinados de forma correcta de manera que el simulador pueda realizar un chequeo de sintaxis al modelo CPN y generar el código necesario para realizar la simulación del modelo.

Para comunicarse con el simulador de CPN Tools, se debió construir para Access/CPN un protocolo que implementara el formato de paquetes BIS (listas de elementos booleanos, enteros y de cadenas de caracteres) junto con componentes que permitieran tomar un modelo CPN transformado en un conjunto de clases de Java y llevarlo al simulador de CPN Tools para la realización de un chequeo de sintaxis y de la simulación. La figura 2.25 muestra cómo está implementado el protocolo con el formato BIS en la interfaz Java de Access/CPN a través de los distintos módulos, para lograr así la comunicación con el simulador de CPN Tools.

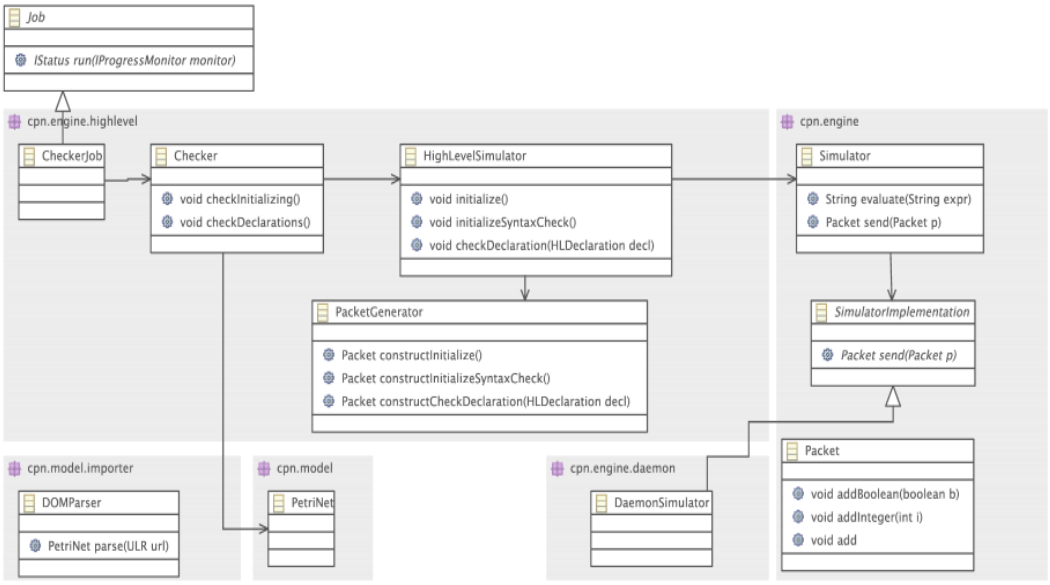

Figura 2.25. Protocolo de comunicación entre Access/CPN y el simulador [9].

En el módulo *model* se encuentra la representación orientada a objetos del modelo CPN. Los objetos *Packet*, que implementan el formato BIS, son enviados al objeto *Simulator*. El objeto *Simulator* delega al objeto *DaemonSimulator* la tarea de comunicarse con el simulador vía TCP/IP de la misma manera que lo realiza el editor gráfico de CPN Tools. *Simulator* provee comunicación al nivel de paquetes.

Una clase que hereda las características (métodos y atributos) de *Simulator* es la clase *HighLevelSimulator* que permite realizar llamadas al simulador con un grado mayor de abstracción para invocar métodos y rutinas. El simulador utiliza la clase *PacketGenerator* para crear paquetes a medida que es necesario comunicarse con el servidor para enviar peticiones.

Por otra parte, la clase *Checker* es la que se encarga de enlazar el modelo CPN ya cargado en Access/CPN con el simulador de CPN Tools. Además, *Checker* se encarga, mediante el apoyo del simulador, de realizar un chequeo de sintaxis del modelo. La clase *Checker* delega a *CheckerJob* la tarea de realizar el chequeo de

sintaxis de todas las declaraciones del modelo CPN. Este se integra con la clase *Job* de la plataforma de Eclipse.

Por otra parte, el módulo *model.importer* contiene el importador que está encargado de realizar el "*parsing"* del modelo que se encuentra en el archivo de extensión .cpn al modelo de objetos provisto por Access/CPN. Ese módulo no interactúa directamente con el simulador.

Para poder realizar la simulación de un modelo CPN que ya fue previamente cargado mediante el módulo *model.importer* se utiliza la clase *HighLevelSimulator* para enlazar la aplicación que se esté desarrollando con el simulador de CPN Tools. También se utiliza *Checker* para realizar comprobaciones del modelo previo a la simulación del mismo.

2.8 Extensiones de CPN Tools

A partir del lanzamiento de la versión 4.0 de CPN Tools [5] (Septiembre 2013) fue incluido en el software de CPN Tools, la posibilidad de añadir y gestionar extensiones. Esta fue una característica principal en el lanzamiento de dicha versión. Las extensiones permiten añadir funcionalidades extra que estén escritas en el lenguaje Java a CPN Tools. Además, las extensiones permiten añadir funcionalidades a la herramienta sin tener que involucrarse con los lenguajes Beta [6] y Standard ML [7], que son utilizados por el editor gráfico de CPN Tools y por el simulador.

En esta sección, se realiza una introducción a la arquitectura de las extensiones de CPN Tools. Además, se describe brevemente cuales son los patrones de comunicación utilizados en la comunicación de las extensiones con CPN Tools. Finalmente, se describe la interfaz utilizada en Java para la creación de extensiones.

Adicionalmente, el anexo "B" del presente trabajo contiene una guía rápida que describe los pasos realizados para la configuración y desarrollo de una extensión de CPN Tools. El anexo "C" contiene una introducción a Debug/CPN. Debug/CPN es un programa que fue desarrollado para ser una extensión de CPN Tools. A través de este programa se pueden realizar, entre otras cosas, llamadas al simulador para realizar pruebas de comunicación utilizando el protocolo BIS.

2.8.1 Arquitectura de las extensiones

CPN Tools había consistido antes de la versión 4.0, en dos procesos, un editor gráfico y un simulador que se encarga de realizar las tareas de simulación y análisis del modelo. La librería Access/CPN [9] fue desarrollada para permitir la comunicación del simulador de CPN Tools con cualquier componente escrito en Java, reemplazando así el editor gráfico. En este caso, las extensiones de CPN Tools son invocadas y ejecutadas a través de algún comando o elemento de interfaz gráfica que utilice el usuario del editor gráfico. Por lo tanto, las extensiones, además de comunicarse con el simulador de CPN Tools, deben tener una conexión al editor gráfico. Con la arquitectura planteada (ver figura 2.26) el editor gráfico tiene la capacidad de llamar procedimientos o funcionalidades contenidas en las extensiones de CPN Tools.

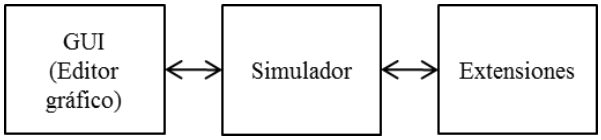

Figura 2.26. Arquitectura básica de CPN Tools con las extensiones.

Para lograr esta comunicación entre editor gráfico, simulador y extensiones, estos utilizan por debajo el protocolo de CPN Tools, BIS, y en los distintos patrones de comunicación que pueden realizarse a través de este protocolo.

2.8.2 Patrones de comunicación con CPN Tools

Los patrones de comunicación se refieren a las distintas maneras de comunicación que poseen los tres procesos de CPN Tools (editor gráfico, simulador y extensiones). En [8] se definen hasta 9 tipos de patrones de comunicación. Todos estos utilizan como forma de comunicación el protocolo de comunicación BIS.

Los patrones 1, 2 y 3 se refieren a los posibles patrones de comunicación entre el editor gráfico y el simulador. Es decir, patrones de comunicación donde no está incluido el manejo de Extensiones de CPN Tools. En este tipo de patrones, básicamente, uno de los dos procesos (simulador o editor gráfico) envía una solicitud a la otra punta que se encarga de procesar la solicitud y enviar un mensaje de *feedback*. Por ejemplo, la figura 2.27 ilustra mediante un diagrama de secuencia el patrón Nro. 1. En este tipo de patrón, el editor gráfico (o GUI) envía un mensaje de solicitud al simulador para que este, por ejemplo, realice alguna tarea (chequeo de sintaxis, simulación del modelo, etc.). Este procesará la solicitud y, si es posible, realizará la tarea indicada y a través de un mensaje de respuesta dará al editor gráfico el *feedback* correspondiente.

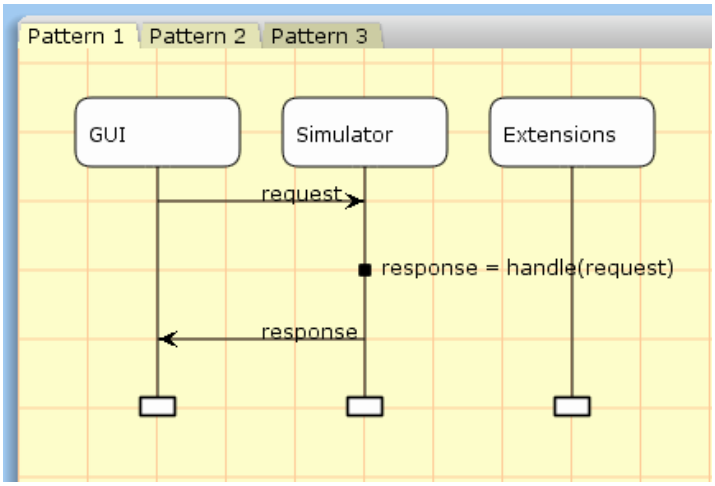

Figura 2.27 Patrón de comunicación Nro. 1 [8].

Ahora, con la introducción del manejo de extensiones de CPN Tools, además de los patrones 1, 2 y 3, se introducen otro conjunto de patrones que involucran la interacción de las extensiones de CPN Tools. Los patrones 4, 5 y 6 definen la manera en cómo se puede comunicar el simulador de CPN Tools con las extensiones dejando esta vez por fuera al editor gráfico. Como ejemplo de este tipo de patrones, colocaremos el patrón Nro. 4 (ver figura 2.28). En este caso, las Extensiones envían una solicitud al simulador, que puede ser, entre otras cosas, solicitar algún tipo de información del modelo que esté cargado en el simulador. El simulador de CPN Tools procesa esta solicitud, y de igual forma como lo realiza con el editor gráfico, envía un mensaje de respuesta a las Extensiones para proveer así un *feedback.*

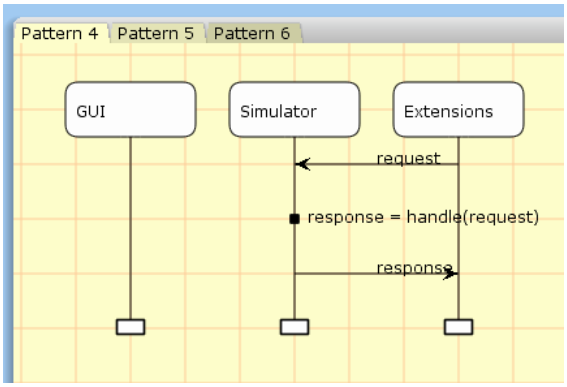

Figura 2.28. Patrón de comunicación Nro. 4 [8].

Los patrones 7 y 8 permiten al simulador redirigir mensajes entre el editor gráfico y las Extensiones sin realizar algún tipo de procesamiento. Es decir, en este caso las dos puntas que en efecto se están comunicando son el editor gráfico y el simulador. Esto sirve, por ejemplo, para enviar alguna petición a las extensiones que realicen algún tipo de tarea (abrir una ventana, ejecutar un comando externo, etc.) en el cual el simulador de CPN Tools no esté involucrado. Como ejemplo, colocamos el patrón Nro. 7 ilustra la comunicación entre el GUI y las Extensiones (ver figura 2.29).

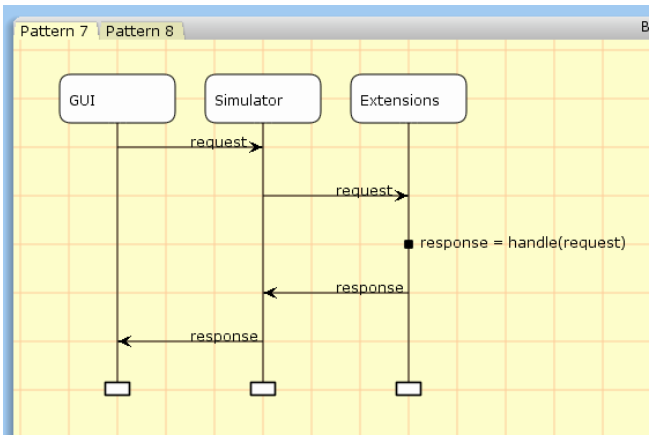

Figura 2.29. Patrón de comunicación Nro. 7 [8].

El último patrón de comunicación (Patrón 9) permite la comunicación entre toda la arquitectura de CPN Tools (editor gráfico, simulador y extensiones). Este patrón se subdivide a su vez en dos tipos de patrones, 9a y 9b. El flujo del patrón 9ª comienza cuando el editor gráfico envía una solicitud al simulador. El simulador toma la solicitud del editor, la procesa y a su vez invoca algún procedimiento de las extensiones mediante un nuevo mensaje de solicitud. Dentro de las extensiones se procesa la solicitud del simulador y se envía una respuesta al simulador, y este a su vez envía una respuesta al editor gráfico (ver figura 2.30).

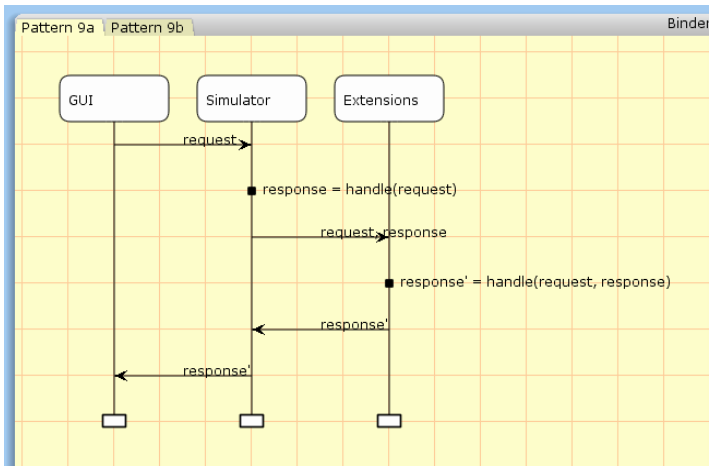

Figura 2.30. Patrón de comunicación Nro. 9a [8].

El patrón de comunicación 9b es una simplemente una variación del patrón de comunicación 9a añadiendo un nivel mayor de abstracción. En este patrón, entre otras cosas, el simulador redirige el mensaje del editor gráfico antes que este efectivamente lo procese. Cuando lo procesa creará otro flujo de intercambio de mensajes. Una explicación detallada de este patrón puede ser encontrada en [8].

La importancia de conocer los patrones de interacción de las extensiones con el editor gráfico y el simulador de CPN Tools radica en que conociendo cómo se comportan estas distintas maneras de comunicación, se puede utilizar efectivamente la interfaz desarrollada para el uso del protocolo BIS y así desarrollar una extensión que se comunique de manera efectiva con el simulador y el editor gráfico de CPN Tools.

2.8.3 Servidor de Extensiones

Hasta ahora hemos hablado de las Extensiones de CPN Tools como un conjunto de elementos adicionales que son embebidos a la solución de CPN Tools. Sin embargo, todas estas extensiones son colocadas en un proceso servidor quien es que la entidad que realmente pasa a manejar todas estas extensiones y la cual gestiona la comunicación de las extensiones con el GUI y el simulador de CPN Tools.

El Servidor de Extensiones es un proceso encargado de manejar todas las extensiones desarrolladas para CPN Tools. Está contenido en el directorio *extensions* de CPN Tools como un archivo llamado *SimulatorExtensions.jar*. Este puede ser iniciado manualmente abriendo este archivo. Sin embargo, este servidor inicia automáticamente cuando abre algún modelo en CPN Tools. Cuando inicia, cargará todas las extensiones (contenidas en el subdirectorio *plugins* de la carpeta *extensions*) y pasará a obtener un puerto y dirección para la escucha de peticiones del simulador.

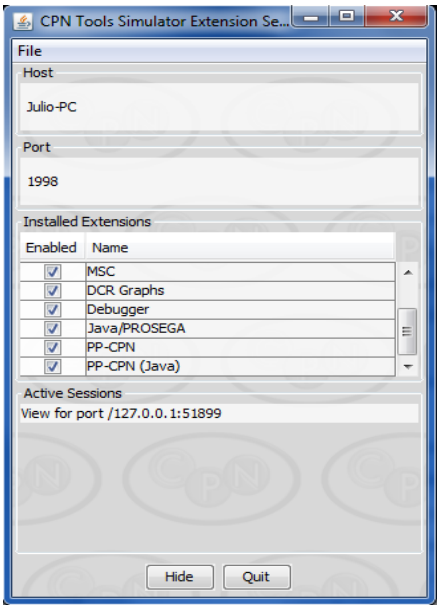

Figura 2.31. Vista del Servidor de Extensiones.

2.8.4 Interfaz de desarrollo

A continuación, se describen las interfaces y mecanismos desarrollados en Java que proveen las herramientas necesarias para el desarrollo de extensiones.

```
1  package org.cpntools.simulator.extensions;
2
3  public interface Extension {
4      int getIdentifier();
5      String getName();
6      List<Command> getSubscriptions();
7      Extension start(Channel c);
8      Packet prefilter(Packet request);
9      Packet handle(Packet p, Packet response);
10     Packet handle(Packet p);
11
12     Object getRPCHandler();
13     String inject();
14
15     List<Option<?>> getOptions();
16     <T> void setOption(Option<T> option, T value);
17
18     List<Instrument> getInstruments();
19     void invokeInstrument(Instrument i, Element e);
20 }
```

Código 2.2. Interfaz principal de una extensión [8].

El código 2.2 presenta la interfaz *Extension.* Todas las extensiones a desarrollar para CPN Tools deben heredar, en principio, de esta interfaz. Esta interfaz provee las primitivas básicas para la integración de la extensión a desarrollar con CPN Tools.

Las extensiones a desarrollar deben disponer de dos identificadores, un número entero (Código 2.2, l. 4) y un nombre que permita describir la extensión (Código 2.2, l. 5). Para pruebas, las extensiones utilizar el identificador reservado 9999. Las extensiones también deben suscribirse a mensajes remitidos desde el editor gráfico y el simulador. Esto se realiza mediante el método *getSubscriptions()* (Código 2.2, l. 6).

Al instante de ser creadas, las extensiones no se encuentran conectadas al simulador (primero se realizan configuraciones en un segundo plano como, por ejemplo, el método *getSubscriptions()*). Cuando una extensión se conecta exitosamente, se invoca el método *start()* (Código 2.2, l. 7). El parámetro canal *Channel* de dicho método es una representación de la conexión con el simulador y la interfaz de usuario (el canal del protocolo BIS).

Para un mayor nivel de integración con la interfaz gráfica de CPN Tools (el editor gráfico) se proveen los métodos de añadir opciones y añadir instrumentos (Código 2.2, ll.15-19). Estos métodos permiten modificar el editor gráfico de CPN Tools para así permitir una interacción directa entre el usuario de CPN Tools y las extensiones desarrolladas a través del editor gráfico.

AbstractExtension

La interfaz *Extension* incluye varios métodos, los cuales la mayoría no son necesarios para una operación común de las extensiones. Por esta razón, fue desarrollada la interfaz *AbstractExtension* que se hace cargo de la implementación de un conjunto de métodos de *Extension.*

De esta manera, el usuario programador que requiera desarrollar una extensión, solo deberá encargarse de dos métodos de la interfaz *Extension*: *getIdentifier()* (código 2.2, l. 4) para la definición del identificador de la extensión y *getName()* (código 2.2, l. 5) para la definición del nombre de la extensión.

El Código 2.3 muestra un ejemplo de desarrollo donde se implementan los métodos mínimos necesarios declarando a esta extensión como clase hija (a través de herencia) de *AbstractExtension*.

```
1  public class HelloWorld extends AbstractExtension {
2      public String getName() {
3          return "Hello World";
4      }
5
6      public int getIdentifier() {
7          return Extension.TESTING;
8      }
9  }
```

Código 2.3. Ejemplo de creación de una extensión [12].

Esta implementación es la mínima implementación necesaria para que el servidor de extensiones, al ejecutarse, cargue con éxito la extensión (figura 2.32).

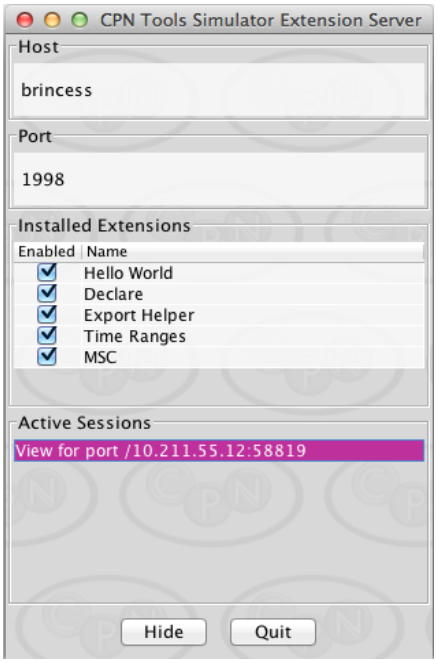

Figura 2.32. Carga de una extensión en el servidor de extensiones [12].

Options

Una de las versatilidades de las extensiones es que permiten modificar la interfaz gráfica de CPN Tools de manera que el usuario pueda tener algún tipo de interacción con las extensiones. La interfaz *AbstractExtension* provee la opción *addOption( )* que permite agregar opciones al panel izquierdo del editor gráfico de CPN Tools.

```
1  public HelloWorld() {
2      addOption(Option.create("Boolean", "boolean", Boolean.class));
3      addOption(Option.create("String", "string", String.class), "Britney");
4      addOption(Option.create("Integer", "integer", Integer.class), 1202);
5  }
```

Código 2.4. Ejemplo de agregar opciones a CPN Tools desde una extensión [12].

El código 2.4 ilustra un ejemplo de cómo pueden ser agregadas opciones al editor gráfico de CPN Tools (figura 2.33) desde el constructor de la extensión que se desarrolle.

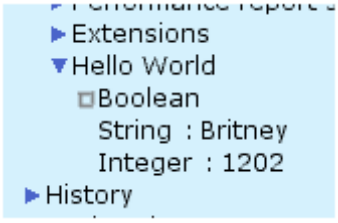

Figura 2.33. Inclusión de opciones a CPN Tools desde una extensión [12].

Instruments

Las extensiones también permiten agregar instrumentos al editor gráfico de CPN Tools. Los instrumentos son los botones contenidos en las distintas cajas de herramientas contenidas en el panel izquierdo del editor gráfico (*Tool box*). De esta manera, se podrían llamar distintas funciones a ser desarrolladas dentro de una extensión desde de botones contenidos en una caja de herramientas. Esta caja de herramientas puede ser, una ya existente de CPN Tools (*auxiliary, create, declare, hierarchy, monitoring,* etc.) o se puede crear una nueva.

El código 2.5 muestra un ejemplo de creación de dos instrumentos. El primero tiene como finalidad ser un botón para que cuando el usuario presione dicho botón la extensión maneje posteriormente la lógica necesaria para la exportación del modelo CPN cargado en el editor a una representación en código Java. El segundo instrumento es la creación del instrumento de depuración (Debug/CPN). En este caso, este instrumento de depuración es una extensión que sirve para la realización de pruebas. En particular, este segundo instrumento está efectivamente implementado

como una extensión de CPN Tools, viene incluido a partir de la versión 4.0 de CPN Tools y es descrito en el anexo "C" del presente trabajo.

```
1  exportInstrument = new Instrument(Instrument.ToolBoxes.NET,
2                                    "export_java", "Java",
3                                    "Export model as Java code");
4  addInstrument(exportInstrument);
5
6  addInstrument(new Instrument("Development", "debug",
7                               "De-\nbug", "Start protocol debugger"));
```

Código 2.5. Creación de instrumentos para el GUI de CPN Tools [12].

La figura 2.34 muestra el resultado de la ejecución del código 2.5. En el primer caso (Código 2.5, ll. 1 - 4) se agregó el botón de exportar el modelo a código Java en la caja *Net*. En el segundo caso (Código 2.5, l. 6) se creó una nueva caja denominada *Development* que contiene el instrumento *Debug*.

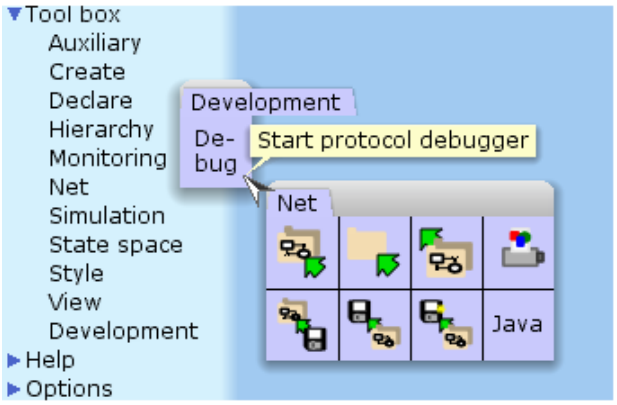

Figura 2.34. Agregado de instrumentos a CPN Tools por extensiones [12].

2.9 Introducción a la Teoría de Autómatas

La Teoría de Autómatas [50] [51] es una rama de las Ciencias de la Computación que comprende el estudio de las máquinas abstractas y los problemas que estas máquinas son capaces de resolver. El modelo principal de estas máquinas es el autómata (o máquina de estado).

Los autómatas son sistemas combinacionales que atraviesan ciertos estados de acuerdo a un conjunto de símbolos de entrada y a una función de transición (a menudo representada mediante una tabla). En función de estos cambios de estado se produce un conjunto de símbolos de salida. Los autómatas pueden extenderse a máquinas más complejas que permiten resolver problemas de mayor complejidad con un nivel mayor de abstracción.

Los modelos de la Teoría de Autómatas van desde sistemas básicos como sistemas combinacionales hasta máquinas más complejas como lo son las Máquinas de Turing (ver figura 2.35) A continuación, se mencionan brevemente las máquinas que son comúnmente estudiadas dentro de la Teoría de Autómatas:

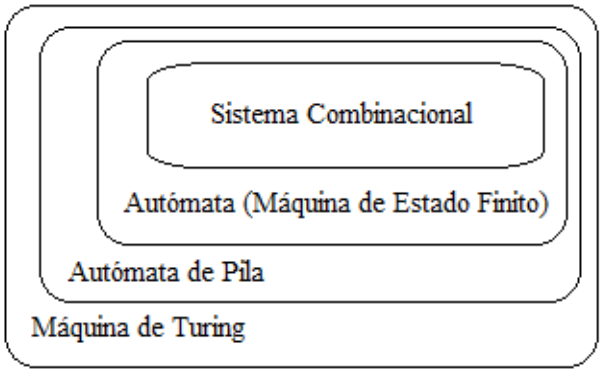

Figura 2.35. Niveles de abstracción de los distintos modelos de autómatas.

- **Sistema combinacional:** Como su nombre lo indica, son sistemas que producen una salida en función de un conjunto de posibles combinaciones de entradas. Un ejemplo claro de estos son los sistemas digitales donde se produce un resultado en función de un conjunto de posibles valores lógicos (verdadero o falso) y una función booleana.

- **Autómata (o Máquina de Estado Finito):** Es un modelo conformado por un alfabeto, un conjunto de estados finito, una función de transición, un estado inicial y un conjunto de estados terminales (o estados de parada).

- **Autómatas de Pila:** Es una máquina que hereda el funcionamiento de un autómata simple que recibe una cadena y determina si esa cadena (o palabra) es capaz de ser procesada (o aceptada) por dicho autómata de pila. Su funcionamiento se apoya adicionalmente en la existencia de una memoria auxiliar llamada pila que maneja una política del tipo LIFO (*Last In, First Out).*

- **Máquina de Turing:** Es el modelo computacional abstracto más potente. Posee una memoria infinita en forma de cinta, así como un cabezal que puede leer y cambiar esta cinta, y moverse en cualquier dirección a lo largo de la cinta, de manera de poder computar los símbolos de entrada contenidos en la cinta.

La importancia de introducir sobre la Teoría de Autómatas es debido a que uno de los objetivos de Java/PROSEGA es el de convertir Grafos de Estado (asociados a modelos CPN) que estén cargados en CPN Tools [5] a Máquinas de Estado Finito y posteriormente realizar un proceso de análisis de lenguaje. Por tal razón, se da una descripción sobre los términos de alfabetos, cadenas y lenguajes y posteriormente se introduce con mayor profundidad el concepto de autómatas.

2.9.1 Alfabetos

Un alfabeto es un conjunto finito no vacío cuyos elementos se llaman símbolos. Se denota generalmente por $\sum$. Ejemplos de alfabeto pueden ser los siguientes conjuntos:

$\sum = \{ a,\ b,\ c \}$ $\sum = \{ 0, 1 \}$ $\sum = \{ send, receive \}$ $\sum = \mathbb{Z}$

2.9.2 Cadenas

Dado un alfabeto $\sum$, una cadena (o palabra) es una secuencia finita de símbolos de dicho alfabeto. Por ejemplo, si $\sum = \{ 0, 1 \}$ posibles cadenas derivadas de este alfabeto pueden ser 0, 1, 01, 10, 101,..., etc.

Para cualquier alfabeto, puede existir la cadena vacía (o épsilon) con un número de símbolos igual a 0. La cadena vacía es representada por el carácter ε.

2.9.3 Lenguajes

Dado un alfabeto $\sum$, se define un conjunto de cadenas denotado como $\sum^*$. Este conjunto incluye todas las cadenas posibles que se pueden formar a partir de los símbolos de dicho alfabeto. Pondremos como ejemplo un alfabeto binario:

$\sum = \{0, 1\} \rightarrow \sum^* = \{ \varepsilon, 0, 1, 00, 01, 10, 11, 000, ... \} \rightarrow | \sum^* | = \infty$

Un lenguaje *L* sobre un alfabeto $\sum$, es un subconjunto de $\sum^*$. Es decir, L es un subconjunto de todas las cadenas que pueden formarse a partir de un alfabeto dado. Ejemplificando nuevamente sobre un alfabeto binario $\sum = \{0, 1\}$

$L = \{b | b \ \ es\ menor\ o\ igual\ a\ 2\ bits\} \rightarrow L = \{\ \varepsilon, 0, 1, 00, 01, 10, 11\}$

Si todas las cadenas de un lenguaje satisfacen alguna regla en común, el lenguaje puede ser definido mediante una expresión regular. Por ejemplo si se tiene el siguiente alfabeto $\sum = \{\ send, receive\ \}$ y se quiere construir un lenguaje cuyas cadenas repitan la secuencia de símbolos $send\ \ receive$ se puede definir dicho lenguaje requerido mediante la siguiente expresión regular:

$L = (send\ receive)^*$ donde el operador * se denomina Clausura o Estrella de Kleene y este denota que la secuencia de símbolos $send\ \ receive$ puede ser repetida 0 o más veces. Por lo tanto, palabras pertenecientes a este lenguaje pueden ser:

$L = \{\ \varepsilon, send\ \ receive,\ \ send\ receive\ send\ receive, \ldots\}$

De esta manera se aprecia que los lenguajes pueden ser a su vez conjuntos infinitos, por lo que su definición, si es posible, a través de expresiones regulares puede ser muy útil para determinar el patrón de las cadenas del lenguaje.

Si no fuese posible determinar una expresión regular para un lenguaje en particular, debido a la complejidad algorítmica que puede llegar a representar, una solución puede ser obtener un subconjunto de ese lenguaje tal que dicho subconjunto cumpla con una regla determinada. Otra solución visible sería obtener palabras aleatorias del lenguaje de manera de estudiar dicho lenguaje de una manera parcial.

2.9.4 Máquinas de Estado Finito

Una Máquina de Estado Finito (o autómata finito) es una 5-tupla de la forma $M = (\sum , Q , q_0 , F , \delta)$ donde:

$\sum$ es un alfabeto.
Q es un conjunto no vacío de estados
q_0 es un estado inicial. $Q_0 \in Q$.
F es un conjunto de estados terminales (o estados de parada).
Δ es una función de transición. Dado un estado de entrada y un símbolo, la función arroja un estado de salida. Δ: Q x $\sum \rightarrow$ Q.

Una Máquina de Estado Finito puede ser representada a través de un grafo dirigido cuyos nodos representen los estados del autómata y los arcos contengan símbolos del alfabeto (ver figura 2.36) de acuerdo a la función de transición que se defina. Los estados terminales son dibujados mediante un doble borde y el estado inicial (que también puede ser terminal) con un sombreado negro.

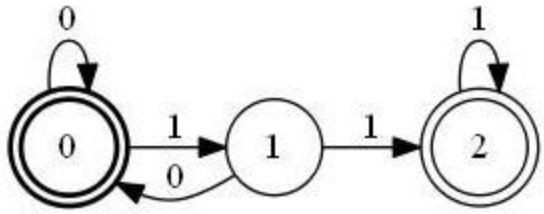

Figura 2.36. Representación de un autómata mediante un grafo.

El grafo de la figura 2.36 representa entonces la Máquina de Estado con la siguiente función de transición (ver tabla 2.3). Dado un estado de entrada q_i, i = 0, 1, 2 y un símbolo del alfabeto {0, 1} la función de transición δ produce un estado de salida q_j.

Tabla 2.3. Ejemplo de función de transición.

δ	$\mathbf{0} \in \sum$	$\mathbf{1} \in \sum$
$\mathbf{q_0 = 0}$	0	1
$\mathbf{q_1 = 1}$	0	2
$\mathbf{q_2 = 2}$	-	1

El funcionamiento general de una Máquina de Estado es el siguiente: dada una cadena de entrada *u*, la máquina de estado comenzará a procesar los símbolos de *u* a partir del estado inicial q_0. A medida que se procesa cada símbolo de la cadena, la máquina va cambiando su estado de acuerdo a lo que indique la función de transición. El procesamiento finalizará cuando se haya procesado el último símbolo de *u*. Si el estado final en el que se detuvo el procesamiento es un estado de aceptación, entonces la cadena pertenece al lenguaje que reconoce el autómata; en caso contrario, la cadena no pertenece a dicho lenguaje.

De acuerdo al tipo de máquina de estado el tipo de procesamiento del autómata puede variar. La máquina de estado clásica es aquella que se encarga de aceptar o reconocer cadenas de lenguajes. Sin embargo, existen otros tipos de procesamiento. A continuación, se mencionan los tipos principales de autómatas:

- **Aceptoras (*Acceptors*):** Son aquellas máquinas que procesan cadenas para determinar si estas pertenecen o no al lenguaje que es capaz de leer dicha máquina de estado. También se les denomina reconocedoras del lenguaje. Este tipo de autómata es el utilizado en el presente trabajo.

- **Transductores (*Transducers)*:** Son aquellas máquinas que se encargan de "traducir" una secuencia de símbolos de entrada a una secuencia de símbolos de salida. En este caso, a partir de un estado en particular y un símbolo de entrada, la máquina producirá un símbolo perteneciente a otro conjunto de

símbolos. En la figura 2.37 se aprecia un ejemplo de un transductor donde el símbolo *a* se transforma a *x*, el *b* a *y*, y el símbolo *c* a *z*. Una aplicación clásica de estas máquinas es el reconocimiento por voz (*speech recognition*) [27] [28] donde se generan cadenas de caracteres a partir de señales de audio.

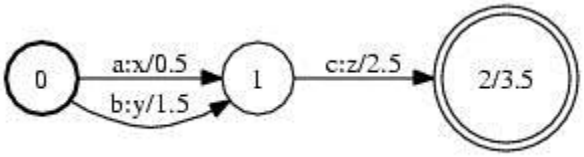

Figura 2.37. Ejemplo de un transductor [28].

En el presente trabajo, las máquinas de interés son los aceptores. No obstante, con las librerías y herramientas presentadas en este trabajo podría ser posible la construcción de transductores.

2.9.5 Lenguaje asociado a una Máquina de Estado

Como fue descrito anteriormente, las Máquinas de Estado Finito se encargan de procesar (o reconocer) cadenas de un lenguaje. Todas las cadenas que son procesadas por una máquina de estado pertenecen al lenguaje que dicha máquina reconoce. Por lo tanto, dada la estructura de una máquina de estado es posible determinar el lenguaje que la misma reconoce.

La figura 2.38 ilustra una máquina de estado finito pequeña en cuyo caso es fácil determinar las cadenas que reconoce y por lo tanto, podemos definir su lenguaje.

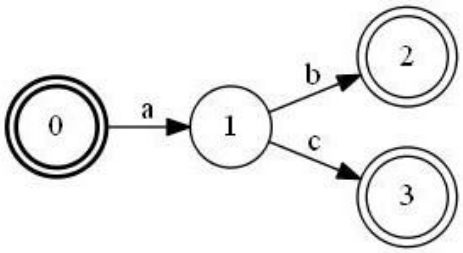

Figura 2.38. Ejemplo de máquina de estado finito acíclica.

El lenguaje de la máquina de estado de la figura 2.40 es $L = \{ab, ac\}$ pues el procesamiento inicia en el estado inicial 0 y finaliza, o en el estado 2 o en el estado 3. Desde el punto de vista de la representación del autómata como un grafo, las palabras aceptadas por un autómata son todas aquellas que pueden ser generadas a partir del procesamiento desde el nodo inicial (en este caso el nodo 0) y que este procesamiento (o camino) concluya en alguno de los nodos terminales (en este caso el nodo 2 o el nodo 3). En este ejemplo, los únicos dos caminos validos son 0 1 2 y 0 1 3. Por lo tanto, las únicas palabras que pueden ser aceptadas son *ab* o *ac.*

La máquina de estado presentada en la figura 2.38 es acíclica. Es decir, no posee ciclos. Un ciclo es un camino dentro de la máquina que inicia en un nodo y finaliza en el mismo nodo. Cuando una máquina contiene al menos un ciclo, el lenguaje de dicha máquina es infinito. Es decir, acepta un conjunto infinito de palabras. A continuación, se presenta una máquina de estado finito que si contiene ciclos (figura 2.40) y se procede entonces definir una solución para la generación del lenguaje infinito asociado.

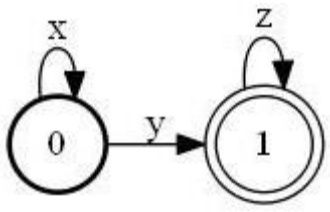

Figura 2.39. Ejemplo de máquina de estado con ciclos.

En la máquina de estado presentada en la figura 2.39, existen 2 ciclos, es decir 2 caminos que comienzan en un nodo y terminan en un mismo nodo. En este caso, estos 2 caminos son simplemente 2 lazos (en el nodo 0 y en el nodo 1). Un lazo es un camino compuesto por un único arco que parte y finaliza en un mismo nodo. Esto es suficiente para que el lenguaje que procese la máquina de estados sea infinito. Cadenas que puede aceptar dicha máquina pueden ser:

$L = \{ xy, xyz, xxyzz, xxxyzzz, xxxyzzzz, \dots \}$, es decir, el lenguaje aceptado por la máquina son todas aquellas palabras cuya secuencia puede iniciar con un conjunto de símbolos *x*, contienen un símbolo y, y pueden finalizar con un conjunto de símbolos igual a *z*. Con este ejemplo, se deduce que la cantidad de palabras con esta regla o patrón es infinita.

Dado que es imposible definir el lenguaje de esta máquina cíclica por extensión (enumerando todos sus elementos) debido a que es infinito, existen varios tipos de soluciones para la generación del lenguaje:

1. Todas las palabras del lenguaje aceptadas por el autómata cumplen con un mismo patrón o regla, por lo que se puede definir el lenguaje de la máquina utilizando una expresión regular. Esta expresión define la estructura (en cuanto a símbolos se refiere) de cada una de las palabras aceptadas por la máquina. En el caso de la figura 2.39 la expresión regular que define al lenguaje sería: $L = \{ x^* \ y \ z^* \}$. Esta expresión sintetiza que, todas las palabras aceptadas por el autómata se componen mediante: un símbolo *x* repetido 0 o más veces, un símbolo *y*, y finalizan con el símbolo *z* repetido 0 o más veces.

2. Dada la complejidad algorítmica que puede llegar a representar, el computar una expresión regular para una máquina de estado, se puede obtener un subconjunto finito del lenguaje. Este subconjunto finito del lenguaje puede estar determinado por alguna regla que permita obtener una muestra del lenguaje de manera poder analizar de manera parcial el mismo. Esta regla puede estar basada en alguna política que se hace al recorrer el grafo que representa a la máquina para poder producir así las palabras reconocidas por la máquina.

3. Generar una secuencia aleatoria de palabras del lenguaje. Este se basa en la segunda solución que es obtener un conjunto finito de palabras del lenguaje de una manera completamente aleatoria.

2.10 OpenFST

OpenFST [28] [52] es una librería de código abierto (*open source*) para la construcción, edición, manipulación y análisis de transductores de estado finito (FSTs, *Finite-State Transducers*) o simplemente transductores. Los transductores de estado finito, a diferencia de las máquinas aceptores, producen una secuencia de estados de salida a partir de una secuencia de símbolos de entrada. Sin embargo, la librería ofrece opciones para poder utilizar el conjunto de opciones que provee sobre máquinas aceptores (*acceptors).*

La importancia de OpenFST en este trabajo se debe a que esta es la librería que utiliza Java/PROSEGA para la minimización de Grafos de Estado a Máquinas de Estado Finito a través del conjunto de opciones que provee. Esta sección pretende dar una breve introducción a OpenFST. Adicionalmente, se menciona como antecedente la librería AT&T FSM Library, esta fue la librería en la cual se basó OpenFST para

su desarrollo. Luego, se mencionan un conjunto de funciones de OpenFST dando preferencia a aquellas que fueron principalmente utilizadas en el presente trabajo.

2.10.1 Introducción a OpenFST

La librería OpenFST fue desarrollada por miembros de Google Research [76] y del Courant Institute of Mathematical Sciences de la Universidad de New York [77]. Entre sus principales desarrolladores se encuentran Riley M., Mohri M. y Pereira F. quienes desarrollaron en un primer momento la librería AT&T FSM Library [27]. La motivación de haber desarrollado OpenFST radica en tener una librería tan potente y eficiente como AT&T FSM Library pero que fuese gratuita y de código abierto.

Otra motivación de haber desarrollado OpenFST sobre AT&T FSM Library es que esta última está orientada principalmente a la manipulación de máquinas reconocedoras (*acceptors*) mientras que la línea de interés de los desarrolladores de OpenFST ha estado marcado por el uso de transductores de estado finito. Sin embargo, OpenFST fue desarrollado para ofrecer de igual manera soporte a máquinas reconocedoras (como las que son utilizadas en el presente trabajo).

La librería está disponible para sistemas operativos Windows (una ventaja notable sobre AT&T FSM Library que está desarrollada solo para sistemas Linux). La última versión de la librería es la versión 1.5.0 (Julio 2015). La versión que fue utilizada para el presente trabajo es la versión 1.3.1 (Marzo 2012). Ambas pueden ser descargadas en [53].

OpenFST provee dos interfaces para el uso de sus librerías, uno desarrollado en lenguaje C++ y otra interfaz que se basa en rutinas ya compiladas listas para ser utilizadas a nivel de consola de comandos. Esta última interfaz fue la utilizada en el

presente trabajo, por lo que está fuera del alcance de esta investigación una descripción sobre la interfaz desarrollada para C++.

2.10.2 Formato de las máquinas en OpenFST

Para la representación de los FSMs, la librería OpenFST utiliza un formato *de facto*. Este formato se basa en representar la máquina de estado en un archivo de texto plano (*.txt*) donde cada línea de dicho archivo contiene la especificación de un arco de la máquina (nodo inicial, símbolo del arco, nodo destino y un peso opcional asociado). Dicho archivo también debe contener también los estados terminales. A continuación, se coloca un ejemplo del formato utilizado por la librería y posteriormente se ilustra la máquina de estado que dicho formato representaría:

```
0 1 a
1 0 b
0 0 c
1 2 d
1 3 e
2 0 f
3 4 g
0
4
```

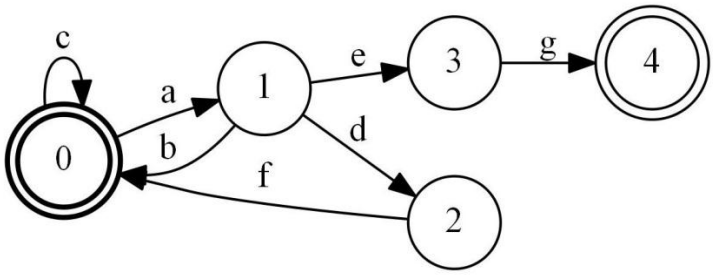

Figura 2.40. Ejemplo de máquina que utiliza el formato de OpenFST.

Cada línea del archivo representa un arco de la máquina de estados tal que cada línea contiene: un primer valor que identifica el nodo inicial de donde parte el arco, un segundo valor que identifica el nodo donde finaliza el arco y un tercer valor que indica el símbolo que contiene el arco. Finalmente, el archivo debe contener otro conjunto de líneas cada una con un único número que represente un estado terminal (o de parada).

Adicionalmente, OpenFST permite incluir otros dos valores en cada de una las líneas que representan los arcos para indicar: un símbolo de salida (para el caso de los transductores) y un peso asociado que representaría el costo de atravesar un arco determinado.

Una consideración importante es que OpenFST representa internamente los símbolos de los arcos como valores enteros. Por ello, para utilizar el archivo en formato de texto plano que representa la máquina de estado con los caracteres utilizados se debe también utilizar un archivo de símbolos donde se realice un mapeo de símbolos con números enteros. Por convención, el número entero 0 representará el símbolo épsilon (o vacío). A continuación, se muestra el archivo de símbolos utilizado por el archivo de texto descrito previamente:

```
a 1
b 2
c 3
d 4
e 5
f 6
g 7
```

Para usar la máquina de estado en las funciones de OpenFST, el archivo de texto plano que represente la máquina de estado debe ser "compilado" a un archivo binario de extensión .fst. Para esto, OpenFST posee el comando *fstcompile*. Por ejemplo, para compilar la máquina descrita anteriormente se utilizaría el comando de la siguiente manera:

```
fstcompile --acceptor=true --isymbols=syms.txt machine.txt > machine.fst
```

De esta manera se tiene como resultado una máquina compilada lista para ser manipulada por las distintas funciones que provee la librería OpenFST. La cláusula booleana -–*acceptor* determina si la máquina es una reconocedora o un transductor. La segunda cláusula --*isymbols* determina el archivo utilizado para los símbolos de los arcos. Al final, se especifica el archivo de texto de entrada (*machine.txt*) y el archivo de salida que es el archivo compilado (*machine.fst*).

Vale destacar que si se quieran trabajar con transductores, se debe agregar la cláusula --*osymbols* especificando el archivo que contiene el mapeo necesario para el conjunto de símbolos de salida.

2.10.3 Funciones principales de OpenFST

OpenFST consiste de dos interfaces. Una interfaz desarrollada en C++ y otra interfaz a nivel de consola de comandos. En este trabajo, la interfaz utilizada y de principal interés es la de nivel de consola de comandos. A continuación, se describen las funciones principalmente utilizadas de OpenFST:

- **fstcompile:** Convierte una máquina de estado finito que esté en un formato texto plano a un archivo compilado (de extensión .fst) para poder ser utilizado por el resto de los comandos de la librería.

- **fstprint:** Realiza el proceso inverso de *fstcompile*. Dado un archivo compilado que representa la máquina de estado este imprime la representación de dicha máquina en un archivo de texto plano utilizando el formato explicado en la Sección 2.10.3.

- **fstrmepsilon:** Dado una máquina de estado finito, esta función se encarga de remover todos los arcos de dicha máquina cuyo símbolo sea épsilon (ε), es decir, el símbolo vacío representado numéricamente por 0.

- **fstdeterminize:** Dado una máquina de estado finito, esta función se encarga de determinizar dicha máquina. El resultado es una máquina de estado equivalente tal que ninguno de sus estados posee dos arcos de salida con un mismo símbolo.

- **fstminimize:** Esta operación ejecuta la minimización de una máquina de estado A. Produce como salida una máquina de estado B con un número mínimo de estados tal que esta máquina de estado B es una máquina equivalente a la máquina de estado A.

- **fstarcsort:** Esta operación se encarga de ordenar el archivo de una máquina de estado finito en función de los estados. Si se utiliza la función *fstprint* después de haber ejecutado *fstarcsort* es posible apreciar cómo fueron organizados todos los arcos en el archivo de acuerdo en función de los estados.

- **fstdifference:** Esta función computa la diferencia A - B entre dos máquinas de estado finito A y B. Solo las cadenas que sean reconocidas por el primer autómata A y que no sean reconocidas por el autómata B son retenidas en el resultado. Si ambas máquinas son equivalentes, el resultado será una máquina de estado vacía.

- **fstdraw:** Dada una máquina de estado en un formato de archivo compilado, esta función se encarga de transformar la representación de esta máquina a un archivo de extensión *.dot*. Este archivo es resultante es el utilizado por el software Graphviz para la graficación de las máquinas de estado.

2.11 Graphviz

Graphviz [55] es una librería de código abierto desarrollada por los laboratorios de AT&T (AT&T Labs) para el dibujado de grafos. El software está disponible tanto para sistemas Windows como sistemas UNIX. La última versión estable publicada es la versión 2.38 (Abril 2014). Esta es la versión utilizada por el software Java/PROSEGA. Puede ser descargada a través de [29]. Graphviz basa su uso en el lenguaje de marcado denominado DOT. Este un lenguaje de marcado que define la estructura de cualquier grafo. Por ejemplo, para la máquina de estado de la figura 2.42 el archivo en lenguaje DOT utilizado por Graphviz sería el siguiente.

```
digraph FST {
rankdir = LR;
size = "8.5,11";
label = "";
center = 1;
orientation = Landscape;
ranksep = "0.4";
nodesep = "0.25";
0 [label = "0", shape = doublecircle, style = bold, fontsize = 14]
    0 -> 1 [label = "a", fontsize = 14];
    0 -> 0 [label = "c", fontsize = 14];
1 [label = "1", shape = circle, style = solid, fontsize = 14]
    1 -> 0 [label = "b", fontsize = 14];
    1 -> 2 [label = "d", fontsize = 14];
    1 -> 3 [label = "e", fontsize = 14];
2 [label = "2", shape = circle, style = solid, fontsize = 14]
    2 -> 0 [label = "f", fontsize = 14];
3 [label = "3", shape = circle, style = solid, fontsize = 14]
    3 -> 4 [label = "g", fontsize = 14];
4 [label = "4", shape = doublecircle, style = solid, fontsize = 14]
}
```

Código 2.6. Ejemplo del código de un archivo en lenguaje DOT.

En el presente trabajo, la herramienta OpenFST [52] a través de su función *fstdraw* convierte máquinas de estado a archivos en lenguaje DOT para ser utilizadas por el software Graphivz. Una de las herramientas principales de Graphviz es la función *dot*. A partir de un conjunto de opciones y de un archivo .dot, la función arroja el grafo dibujado en algún formato de imagen (Graphviz maneja distintos formatos de imagen: png, svg, ps, jpeg, etc.). Un ejemplo de uso es el siguiente:

```
dot -Tjpeg -Gdpi=300 machine.dot -o machine.jpeg
```

En este ejemplo, se genera una imagen de una máquina de estados en formato jpeg y con una resolución de 300 dpi a partir del archivo machine.dot.

CAPÍTULO 3

Java/PROSEGA

3.1 Introducción a Java/PROSEGA

Java/PROSEGA (*Java PROtocol Sequence Generator & Analyser*) es un software desarrollado en el lenguaje de programación Java. Está desarrollado para ser utilizado en sistemas operativos Windows y funciona como una extensión que se integra al software CPN Tools [5] (versión 4.0 o superior). Se comunica con el simulador de CPN Tools a través de una implementación en Java del protocolo BIS ofrecida por Access/CPN [9].

La función principal de Java/PROSEGA es la reducción de Grafos de Estados, asociados a modelos CPN y generados por la herramienta CPN Tools, a Máquinas de Estado Finito (en particular, aceptores). Adicionalmente cuenta con otros módulos para la generación del lenguaje de las Máquinas de Estado Finito resultantes y permite realizar comparaciones entre Máquinas de Estado Finito a través de funciones de diferencia. De esta manera, se provee una solución a los investigadores para el análisis de sistemas y modelos concurrentes construidos a través de las técnicas formales provistas por las Redes de Petri Coloreadas.

En los siguientes apartados de esta sección introductoria se encontrará una breve descripción de Java/PROSEGA comparándolo con su solución predecesora, PROSEGA, luego se realiza una descripción sobre la arquitectura general del software, sus funciones principales y se describe el proceso de instalación y configuración.

En las demás secciones del presente capitulo: se describen los formatos de los archivos de entrada utilizados por Java/PROSEGA, se explica en detalle el proceso de reducción de Grafos de Estado a Máquinas de Estado Finito, se describe el proceso de generación de lenguajes y la función de diferencia sobre máquinas. Además, se introduce a la librería fsm2language desarrollada por el presente autor para la generación del lenguaje. Por último, se enumeran algunas limitaciones presentes en el software.

3.1.1 Java/PROSEGA frente a PROSEGA

Java/PROSEGA está basado en las funcionalidades del software PROSEGA (ver Sección 1.1.5) que fue desarrollado en el lenguaje C y para ser utilizado en sistemas operativos basados en Unix. Esta es una limitante para los investigadores que utilicen el software de CPN Tools puesto que este último se ha mantenido actualizado hasta la fecha en sistemas operativos de Windows. Java/PROSEGA aprovecha las bondades de las últimas versiones de CPN Tools desarrolladas en Windows para integrarse a este último como una extensión ofreciendo así una solución integrada para los usuarios de CPN Tools. A continuación, se presenta un cuadro comparativo de PROSEGA y Java/PROSEGA comparando las tecnologías que utilizan para llevar a cabo las tareas requeridas.

Tabla 3.1. Comparación de Java/PROSEGA frente a PROSEGA.

	PROSEGA	**Java/PROSEGA**
Lenguaje de desarrollo	C	Java
Sistemas operativo requerido	UNIX	Windows
Herramienta de apoyo para la manipulación de FSMs	AT&T FSM Library [27]	OpenFST [52]
Herramienta para la graficación de FSMs	Graphviz	Graphviz
Herramienta de apoyo para la generación del lenguaje	Lextools [57]	fsm2language
Integrado a CPN Tools	No	Sí

Puesto que Java/PROSEGA está utilizado en un entorno Windows no se pudieron utilizar las mismas herramientas en las cuales se apoyó PROSEGA para llevar a cabo las tareas de generación de autómatas y generación del lenguaje. Por lo tanto, se llevó acabo la tarea de escoger herramientas equivalentes que pudieran ser utilizadas en Windows.

La herramienta de reducción de Grafos de Estado a Máquinas de Estado escogida fue OpenFST. En cuanto a la herramienta de graficación utilizada por Java/PROSEGA se logró mantener la utilizada en un primer momento usada por PROSEGA, esta es Graphviz [29] que posee una versión disponible para sistemas operativos Windows.

Valiéndose de estar desarrollado en lenguaje Java, el software Java/PROSEGA se implementó utilizando las librerías Swing/AWT [48] usadas para la construcción de interfaces gráficas. De esta manera, se logró cierta armonía y consistencia a nivel gráfico a la hora de integrar Java/PROSEGA con la interfaz gráfica de CPN Tools. Esto, supera a PROSEGA que su desarrollo se limitó a ser utilizado solo a nivel de consola de comandos.

3.1.2 Arquitectura general del software

A continuación, la figura 3.1 ilustra la arquitectura de Java/PROSEGA con los componentes utilizados y su interconexión para así llevar a cabo las tareas requeridas.

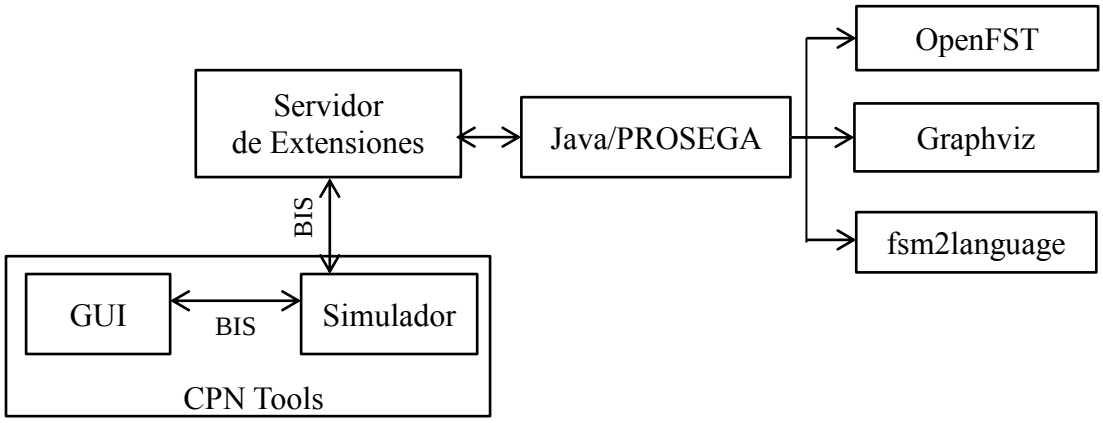

Figura 3.1. Diagrama de la arquitectura de Java/PROSEGA.

Java/PROSEGA se interconecta a CPN Tools mediante el servidor de extensiones (véase Sección 2.8.3) y se carga a este servidor de extensiones como una extensión más para CPN Tools. Para la comunicación de este con el editor gráfico (o GUI) y el simulador de CPN Tools se utiliza el protocolo de comunicación BIS. Al arrancar el servidor de extensiones se cargan los instrumentos de Java/PROSEGA en el editor gráfico de CPN Tools y cuando el usuario lo requiera hace las llamadas requeridas a Java/PROSEGA utilizando dichos instrumentos.

Internamente, Java/PROSEGA utiliza como apoyo el software OpenFST para la reducción de Grafos de Estado a Máquinas de Estado Finito y además para aplicar la función de diferencia de manera poder comparar distintas máquinas. Para la graficación de las máquinas se utiliza el software Graphviz y para la generación del lenguaje se utiliza la librería fsm2language desarrollada por el autor del presente trabajo. Para lograr la interacción con los usuarios, Java/PROSEGA utiliza como apoyo las librerías Swing/AWT provistas para la construcción de interfaces de usuario.

3.1.3 Funciones principales

Los instrumentos cargados en el editor gráfico de Java/PROSEGA proveen un medio para invocar las funciones principales del software. Estos instrumentos son empaquetados en una única caja de herramientas (*tool box*) denominada Java/PROSEGA que contiene dichas funcionalidades.

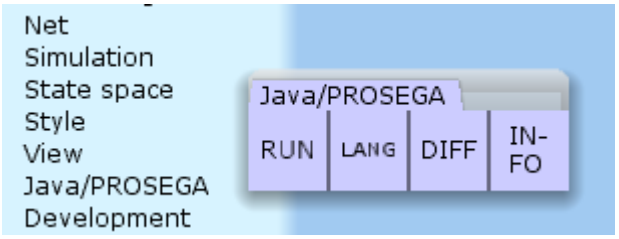

Figura 3.2. Caja de herramientas de Java/PROSEGA.

A continuación, se describen cada una de las funciones provistas dentro de la caja de herramientas de Java/PROSEGA cargada en el editor gráfico de CPN Tools:

- **RUN**: Ejecuta el proceso de reducción de un Grafo de Estado a una Máquina de Estado Finito. El Grafo de Estado (o *State Space*) debe haber sido ya

calculado por la herramienta CPN Tools delegando a esta funcionalidad solo el proceso de minimización.

- **LANG:** Genera el lenguaje de una Máquina de Estado Finito. La máquina que se quiera generar el lenguaje es importada desde un archivo externo.

- **DIFF:** Esta función permite computar la operación de diferencia entre una Máquina de Estado Finito y otra máquina B. La función despliega como resultado una máquina resultante A - B.

- **INFO:** Muestra información general acerca del software y de su autor.

3.1.4 Estructura del proyecto a nivel de clases

Al estar desarrollado en lenguaje Java, el software Java/PROSEGA fue desarrollado teniendo como base el paradigma orientado a objetos. Por lo tanto, el software fue construido a partir del desarrollo de un conjunto de clases en Java cada una con un rol o función particular dentro del software.

Adicionalmente, esta construcción de Java/PROSEGA se llevó acabo utilizando el entorno de desarrollo integrado Eclipse [58]. Eclipse provee un entorno para el desarrollo de proyectos en Java. Mediante Eclipse, todas las clases del proyecto de Java/PROSEGA fueron empaquetadas de manera jerárquica de acuerdo al tipo de función o rol que representan dentro del software. Con esto se ganó una organización y legibilidad en el proyecto. La tabla 3.2 define los paquetes del proyecto Java/PROSEGA.

Tabla 3.2. Paquetes del proyecto Java/PROSEGA

Paquete	Función de las clases que agrupa...
console	Invocación de procesos de herramientas adicionales y de ejecución de comandos por consola (Ej. invocación de OpenFST, creación de directorios, etc.).
fsm	Clases que modelan la estructura y el manejo de máquinas de estado finito.
Gui	Clases para el manejo y despliegue de las interfaces de usuario.
Instrument	Contiene la clase *Toolbox* que contiene los instrumentos a ser colocados en el editor gráfico de CPN Tools.
Language	Provee funciones para la generación del lenguaje de las máquinas y el cálculo de la diferencia.
log	Clase para el manejo de trazas.
ss	Contiene las clases que modelan la estructura y el manejo de los Grafos de Estado (o *State Spaces*)

La figura 3.3 muestra el desglose del conjunto de paquetes previamente descrito, mostrando todas las clases desarrolladas. La clase principal es la clase *JavaProsega*. Esta hereda de la interfaz *AbstractExtension* (véase Sección 2.8.4) para así poder convertir todo este proyecto en una extensión para CPN Tools.

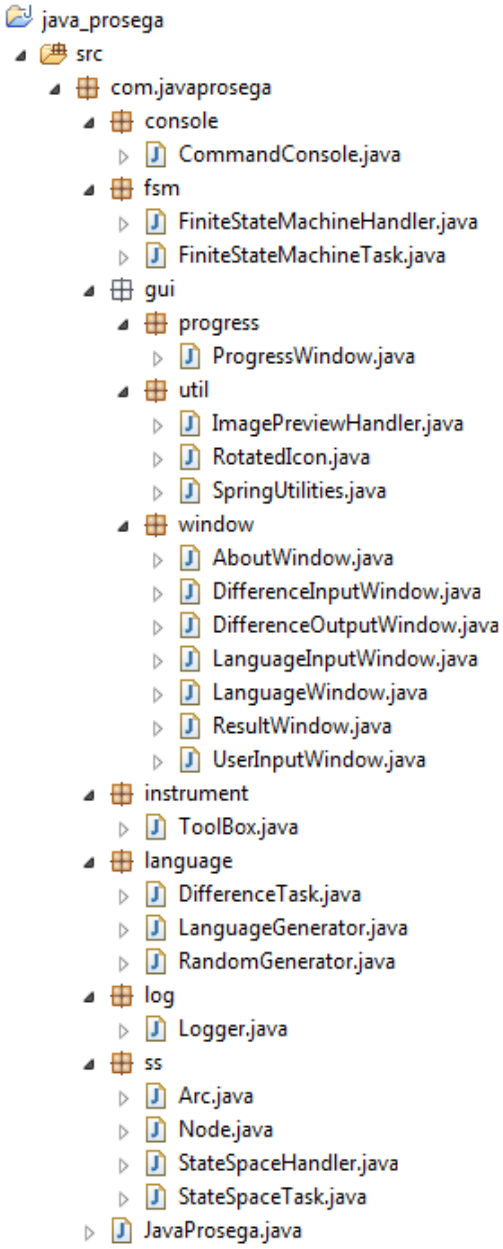

Figura 3.3. Estructura del proyecto Java/PROSEGA en Eclipse.

3.1.5 Proceso de instalación y configuración

Java/PROSEGA viene presentado en una carpeta que contiene los siguientes archivos para su correcta instalación y configuración:

- **java_prosega.jar**: Es el software Java/PROSEGA como tal. El software viene empaquetado en un archivo de extensión .jar. Este es el archivo de extensión para la compresión de proyectos desarrollados en Java. Este archivo debe ser colocado en la carpeta *plugins* del directorio *extensions* de CPN Tools. De esta manera, cuando el servidor de extensiones de CPN Tools se inicialice, este pueda cargar efectivamente a Java/PROSEGA.

- **configure.bat:** Es un script ejecutable de Windows que se encarga de configurar el directorio de Java/PROSEGA en el computador. Además, descomprime las herramientas adicionales (OpenFST, Graphviz, fsm2language) que utiliza Java/PROSEGA colocándolas en el directorio escogido para Java/PROSEGA. Adicionalmente, esta rutina agrega a la variable de entorno *PATH* de Windows la ruta de los archivos binarios de las herramientas adicionales de manera que Java/PROSEGA puede invocarlas directamente, e incluso, el usuario pueda tener acceso a estas herramientas desde la consola de comandos de Windows.

- **third_parties.jar:** Este programa desarrollado en Java es ejecutado por la rutina *configure.bat* para llevar a cabo las tareas previamente descritas para esta rutina. Contiene dentro de sí las herramientas adicionales: OpenFST, Graphviz y fsm2language.

Además, la carpeta que contiene estos archivos también incluye archivos de documentación del software que incluyen información sobre Java/PROSEGA.

Para utilizar Java/PROSEGA, se debe primero tener instalado en el computador la versión 4.0 o superior de CPN Tools que puede ser descargado en

[36]. Además, se debe poseer instalado alguna versión de Java [45] (esto es necesario también para el arranque del servidor de extensiones de CPN Tools).

Teniendo CPN Tools y Java instalados, se debe ejecutar luego la rutina *configure.bat* para que esta rutina se encargue de la configuración del directorio de Java/PROSEGA en alguna carpeta elegida por el usuario y se descompriman allí las herramientas externas (OpenFST, Graphviz, fsm2language).

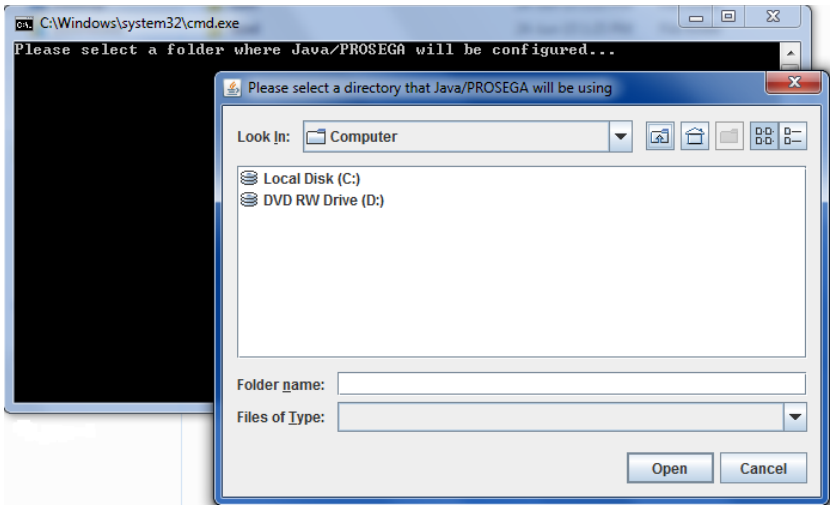

Figura 3.4. Ejecución de la rutina configure.bat.

A partir de este punto es incluso posible para el usuario el utilizar OpenFST, Graphviz o fsm2language por consola de comandos utilizando las operaciones propias de esas herramientas. Después de la ejecución de *configure.bat*, simplemente se debe colocar el archivo *java_prosega.jar* (que es Java/PROSEGA propiamente) dentro de la carpeta *plugins* localizado en el directorio *extensions* de CPN Tools.

Cuando se abre algún modelo en CPN Tools arranca automáticamente el servidor de extensiones y junto con él, el software Java/PROSEGA.

3.1.6 Estructura del directorio de Java/PROSEGA

El directorio de Java/PROSEGA, previamente configurado mediante la rutina *configure.bat* y localizada donde el usuario lo haya seleccionado, contiene a su vez los siguientes sub-directorios:

- **log**: El directorio *log* está destinado a contener archivos de trazas para la depuración y seguimiento de la ejecución de Java/PROSEGA. Contiene un archivo denominado *log.txt* que imprime la traza de la ejecución de Java/PROSEGA. Este archivo ofrece dos tipos de trazas: trazas informativas denotadas con el prefijo *[INFO]* y trazas que informan de alguna excepción o error que se haya producido en el software y se denotan mediante el prefijo *[ERROR]*.

- **out:** El directorio *out* contiene las carpetas autogeneradas para cada corrida de Java/PROSEGA. Por ejemplo, cuando se ejecuta la función *RUN* (reducción del Grafo de Estado a una Máquina de Estado Finito) se crea una carpeta que contiene los archivos resultantes para esa ejecución.

- **third_parties:** Este directorio contiene las herramientas de terceros que utiliza Java/PROSEGA para su funcionamiento. Estas herramientas contenidas en este directorio son: OpenFST, Graphviz y la librería fsm2language.

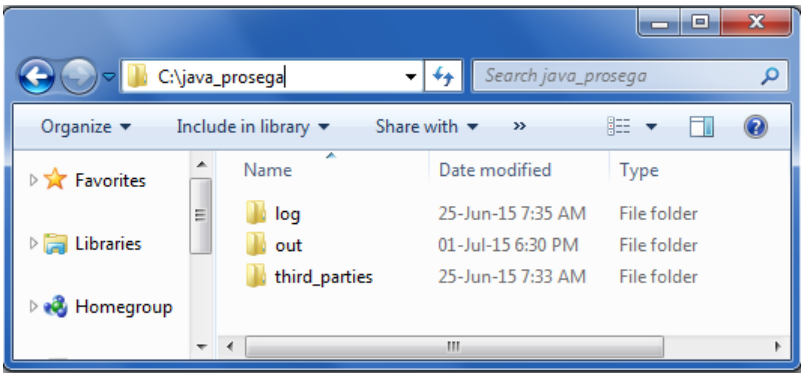

Figura 3.5. Estructura del directorio de Java/PROSEGA.

3.2 Formatos de archivos de entrada

En esta sección del presente trabajo se introducen a los formatos de los distintos tipos de archivos de entrada que son consumidos por el software Java/PROSEGA explicando la estructura que debe poseer el archivo y para qué son utilizados.

Archivos de Máquinas de Estado Finito

Estos archivos de Máquinas de Estado Finito están en texto plano (archivos de extensión .txt) y siguen los lineamientos *de facto* establecidos por OpenFST [52] y AT&T FSM Library [27] para la especificación de autómatas. En este caso nos estamos refiriendo a máquinas aceptoras (*acceptors*) con un único símbolo de entrada. Este formato de las máquinas de entrada a Java/PROSEGA es explicado en la sección 2.10.3 de la presente investigación. En particular, Java/PROSEGA utiliza este tipo de archivos para representar las máquinas que serán operadas mediante la función de diferencia y mediante la función de generación del lenguaje. Java/PROSEGA también permite como entrada para estas funciones, archivos de máquinas de estado que ya estén compiladas (es decir, archivos de máquinas con extensión .fst).

Archivos para la asignación de identificadores a los arcos del Grafo de Estado

Este tipo de archivos es utilizado para, a cada arco del Grafo de Estado asignarle un identificador. Debe poseer un formato tipo texto plano (extensión .txt). Es utilizado en el proceso de reducción del Grafo de Estado a una Máquina de Estado Finito.

Este archivo posee el siguiente formato: un conjunto de *N* líneas que representan los *N* arcos del *State Space* (o lo que es igual, las ocurrencias de las transiciones que están representadas mediante dichos arcos). Cada línea posee tres palabras, separadas por espacios, tal que la primera palabra hace referencia a la página o módulo donde está localizada la transición en el modelo, la segunda palabra es el nombre de la transición en sí y la última palabra es un valor numérico o de tipo *string* que es el identificador que representará a la transición.

Una restricción es que la primera palabra (la página) y la segunda (el nombre de la transición) no pueden contener espacios. A continuación, un ejemplo de la estructura del archivo:

Page1 Transition1 1
Page1 Transition2 2
Page2 Transition1 3
Page2 Transition2 4
Page3 Transition1 5
...

Archivos para la colocación de nodos terminales

Los nodos terminales (o estados terminales) son aquellos nodos que representan los estados de aceptación (o de parada). Este archivo debe poseer un formato en texto plano (extensión .txt). Está compuesto por un conjunto de *N* líneas cada una conteniendo un número *n* tal que $1 \leq n \leq k$ siendo *k* el número de nodos del Grafo de Estados. Por ejemplo, para un Grafo de Estados con 7 nodos este sería un archivo de nodos terminales valido:

1
2
4
7

El ejemplo anterior indicaría lo siguiente: para el Grafo de Estados que contiene 7 nodos, se definen como estados terminales los nodos 1, 2, 4 y 7.

Archivo de símbolos

Es un archivo establecido por OpenFST [52] para realizar el mapeo entre las inscripciones de los arcos de las máquinas de estado y los símbolos que efectivamente representan. Este archivo es útil puesto que OpenFST no representa internamente los valores de los arcos como *strings* sino como valores numéricos. Por lo tanto, a la hora de realizar alguna impresión de la máquina de estado a una representación en tipo texto (véase "Archivos de Máquinas de Estado Finito") o a una impresión de tipo imagen mediante Graphviz se debe contar con este archivo. Este archivo es utilizado por Java/PROSEGA para realizar las impresiones de lenguaje y también para la aplicación de la función de diferencia.

Su estructura consiste en un conjunto de *N* líneas cada una con dos valores separadas por espacio tal que el primer valor consiste en el valor de la inscripción del arco de la máquina de estado y el segundo valor consiste en el símbolo de entrada que se requiere para la máquina.

Por ejemplo, tomemos en consideración una máquina que posee inscrito en sus arcos los símbolos 1, 2 y 3. Si estos símbolos representan en realidad símbolos de tipo *string*, por ejemplo OPEN, SEND y CLOSE, entonces el archivo de símbolos debe tener la siguiente estructura:

OPEN 1
SEND 2
CLOSE 3

Si lo requerido, cuando se realice la impresión del autómata (a su versión tipo texto o a una imagen mediante Graphviz) es que se mantengan dichos símbolos numéricos (o incluso colocar otros), el archivo de símbolos puede poseer la siguiente estructura:

1 1
2 2
3 3

3.3 Reductor de Grafo de Estado a Máquina de Estado Finito

En esta sección se describe la funcionalidad desarrollada de Java/PROSEGA para la reducción de Grafos de Estado a Máquinas de Estado Finito. Este reductor puede ser utilizado por el usuario a través de la opción RUN.

En esta sección, primero se da una introducción al proceso de reducción. Luego se describe el proceso utilizado para la obtención del Grafo de Estados, los arcos y los marcados muertos asociados al modelo CPN que esté cargado en CPN Tools, luego se explica el uso de la interfaz de asignación de identificadores que provee la herramienta Java/PROSEGA y finalmente se describe como Java/PROSEGA lleva a cabo la tarea de minimización y su correspondiente interfaz donde se muestran los resultados.

3.3.1 Introducción al proceso de reducción

El proceso de reducción implementado por Java/PROSEGA de Grafos de Estado a Máquinas de Estado Finito utiliza el algoritmo de reducción descrito en [3]. Este proceso ha sido utilizado en la metodología de verificación de protocolos de comunicación utilizando Redes de Petri Coloreadas descrita en [4]. Sin embargo, este proceso de reducción permite también el análisis de otros sistemas que se deseen analizar mediante autómatas.

A continuación, se muestra un diagrama que ilustra el flujo del proceso de reducción que lleva a cabo Java/PROSEGA.

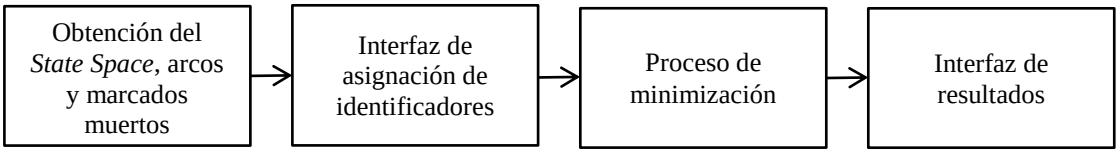

Figura 3.6. Flujo del proceso de reducción de Java/PROSEGA.

El proceso de reducción implementado por Java/PROSEGA está dividido en 4 tareas principales. Primero, se hace una comunicación con el simulador de CPN Tools para obtener el apuntador del Grafo de Estado (o *State Space*) que esté cargado en dicho simulador. Con este apuntador del Grafo de Estado, se localiza dentro del

simulador de CPN Tools los arcos del Grafo de Estado y también los marcados muertos (*dead-markings*) en caso que el usuario los requiera. Para lograr la comunicación con el simulador de CPN Tools de manera de obtener estos elementos se utiliza la implementación en Java del protocolo de comunicación BIS.

Después de este primer proceso, se despliega al usuario una interfaz de asignación de identificadores de manera que este realice el mapeo correspondiente entre los arcos del Grafo de Estado (o transiciones del modelo CPN) y un conjunto de identificadores a utilizar. También se provee un área que permite al usuario insertar los nodos terminales. Adicionalmente, se provee un conjunto de opciones adicionales tales como incluir los marcados muertos a la lista de nodos terminales y escoger el tipo de dato de los identificadores (entero o *string*).

Después de hacer las configuraciones necesarias en la interfaz de asignación de identificadores, se comienza el proceso de minimización invocando los comandos necesarios de OpenFST. Cuando este proceso concluye exitosamente, se muestra una interfaz de resultado que despliega la Máquina de Estado resultante en conjunto con información relevante de la máquina. En esta última tarea, se permite al usuario abrir la imagen generada de dicha máquina. También se permite al usuario generar el lenguaje de la máquina resultante invocando para ello la función de generación del lenguaje.

En paralelo con este proceso, Java/PROSEGA crea una carpeta en el directorio *out* (ver Sección 3.1.6) donde se generaran todos los archivos resultantes de este proceso. El nombre de las carpetas generadas por este proceso de reducción tienen el formato *fsm_<date>* tal que *date* es la fecha y hora exacta cuando inició el proceso.

3.3.2 Obtención del *State Space,* arcos y marcados muertos

La obtención del *State Space* se refiere a obtener el apuntador del Grafo de Estado en el simulador de CPN Tools. De esta manera pueden ser obtenidos los arcos del Grafo de Estado y los marcados muertos. Para lograr obtener estos datos se implementó una serie de mensajes de solicitud utilizando como base la implementación en Java del protocolo BIS desarrollada por Access/CPN [9]. Este proceso se dispara apenas el usuario presiona el instrumento RUN. Una vez finaliza la obtención de estos elementos, se despliega la ventana de asignación de identificadores. A continuación, se muestra el procedimiento realizado para obtener los elementos previamente descritos a través del protocolo BIS.

Para obtener el apuntador del Grafo de Estado, los arcos del Grafo y los marcados muertos se debe realizar la comunicación con el simulador utilizando paquetes de solicitud con *opcode* = 9 y *command* = 800. Mediante el *opcode* 9 se indica que se está utilizando el patrón Nro. 4 (comunicación extensión → simulador) mientras que con *command* 800 se está indicando que se enviarán paquetes al simulador concernientes al manejo del Grafo de Estado (ver tabla 2.1).

Obtención del *State Space*

Para obtener el Grafo de Estado del modelo CPN que esté cargado en CPN Tools, se debe primero realizar el cálculo por el método tradicional provisto por CPN Tools, una vez esté calculado este se guarda dentro del simulador de CPN Tools. Luego, para obtener una referencia a este se tiene un apuntador denominado *ssid* (*state space identifier*). El comando de la figura 3.7 muestra la documentación de CPN Tools [44] para la tarea de obtener la referencia del *State Space*.

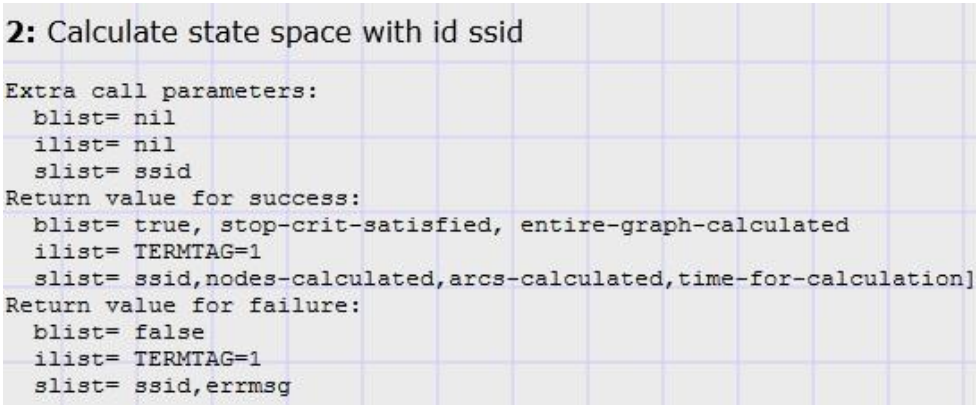

Figura 3.7. Documentación para obtención del *State Space* [44].

En este caso se requiere utilizar un subcomando (*subcommand*) con valor igual a 2. En el mensaje de solicitud la lista de booleanos viajará vacía, la lista de enteros contendrá el valor del subcomando y la lista de *strings* contendrá el identificador *ssid* que es generado por Java/PROSEGA.

La respuesta a esta tarea contiene principalmente el mismo identificador *ssid* (indicando así el enlace entre este identificador y el Grafo de Estados cargado en el simulador de CPN Tools). Además, indica otros valores tales como el número de arcos calculados y el tiempo de cómputo. Si hubiese un error en la llamada, la lista de enteros resultante contendría la marca TERMTAG con valor igual a 1.

A continuación, el código 3.1 muestra un extracto simplificado de la implementación hecha en Java/PROSEGA a partir de la documentación provista por CPN Tools de la llamada descrita previamente. Este código está contenido dentro del constructor de la clase *StateSpaceHandler*, este creará un objeto encargado de la manipulación del Grafo de Estado.

```
1  String ssid = "IDSS" + System.currentTimeMillis();
2
3  Packet p = new Packet(9, 800);
4  p.addInteger(2);
5  p.addString(ssid);
6
7  p = channel.send(p);
8
9  p.reset();
10
11 boolean successfulReponse = p.getInteger() == 1;
12
13 if(successfulReponse){
14
15     stateSpaceHandler.setId(String.valueOf(p.getString()));
16
17     String[] aux = p.getString().split("/");
18     stateSpaceHandler.setNodesCalculated(Integer.parseInt(aux[1]));
19
20     aux = p.getString().split("/");
21     stateSpaceHandler.setArcsCalculated(Integer.parseInt(aux[1]));
22
23     return stateSpaceHandler;
24
25 }else{
26     return null;
27 }
```

Código 3.1. Llamada de Java/PROSEGA para obtención del *State Space*.

Primero se calcula un identificador *ssid* (l. 1). Luego, se crea un paquete del protocolo con *opcode* = 9 y *command* = 800 (l. 3). Se añade el subcomando 2 a la lista de enteros (l. 4) y el identificador que referenciará al Grafo de Estado a la lista de strings (l. 5). Luego, se envía el paquete (l. 7) a través del objeto *channel* quien representa el canal de conexión simulador - Java/PROSEGA utilizando el método *send* que retornará un paquete de respuesta. Este paquete de respuesta se asigna sobre el mismo paquete de solicitud. En l. 11 se pregunta si la marca *TERMSTAG* es igual a 1. Si lo es, se proceden a obtener los datos del paquete (l. 15 - 23). Caso contrario, no se pudo obtener la referencia del Grafo de Estado en el simulador de CPN Tools y se devuelve la marca *null* (l. 26).

En la implementación utilizada del protocolo BIS [14] para incluir datos en las listas de booleanos, enteros y de strings de los paquetes BIS se utilizan los métodos *addBoolean( ), addInteger( ), addString( )* respectivamente. Para obtener un dato de alguna de las listas de los paquetes de respuesta se utilizan los métodos get*Boolean( ), getInteger( ), getString( )* según sea el caso. Cualquiera de estos retornará el primer valor de la lista y lo suprimirá de la lista.

Obtención de los arcos

Los arcos del Grafo de Estado son obtenidos de manera similar a como se obtiene la referencia del apuntador del Grafo de Estado del simulador de CPN Tools. En este, según la documentación [44] el subcomando necesario es 11. Este se encargará de devolver información de los arcos a partir de una lista de enteros que representan los números de los arcos. A continuación, la figura 3.8 muestra la documentación asociada para esta tarea.

```
Extra call parameters:
  blist= nil
  ilist= arc-1,...
  slist= ssid
Return value:
  blist= (see below)
  ilist= TERMTAG=1, (see below)
  slist= ssid,(see below)

num-of-arcs = pop(ilist)
repeat num-of-arcs times
  get_arc()
  if arc-exists (see get_arc above)
  then srcnode-info = get_node()
       dstnode-info = get_node()
  else (nothing more related to current arc)
```

Figura 3.8. Documentación para obtención de los arcos [44].

La figura 3.8 muestra también un algoritmo que se debe utilizar para extraer la información de los arcos provenientes del paquete de respuesta. Este algoritmo llama a unos métodos denominados *get_node()* y get_arc*()* provistos en la documentación. El código 3.2 muestra la implementación de la tarea de obtención de los arcos en Java/PROSEGA a partir de la documentación ilustrada la figura 3.8. También tuvo que ser implementado el algoritmo que acompaña a esta tarea. En cuanto a los métodos *get_node()* y *get_arc()* (figura 3.8), el anexo "E" del presente trabajo presenta la implementación realizada en Java/PROSEGA de estos métodos.

```
1   Packet p = new Packet(9, 800);
2   p.addInteger(11);
3   for(int i = 1; i <= arcsCalculated; i++){
4       p.addInteger(i);
5   }
6   p.addString(ssid);
7
8   p = channel.send(p);
9
10  p.reset();
11
12  boolean successfulReponse = p.getInteger() == 1;
13  if(successfulReponse){
14      int numberOfArcs = p.getInteger();
15      String ssid = p.getString();
16
17      for(int i = 0; i < numberOfArcs; i++){
18          Arc arc = getArc(p);
19          if(arc.getExists() == true){
20              Node sourceNode = getNode(p);
21              Node destNode = getNode(p);
22              arcs.add(arc);
23          }
24      }
25      return true;
26  }else{
27      return false;
28  }
```

Código 3.2. Llamada de Java/PROSEGA para obtención de los arcos del grafo.

El paquete de solicitud se crea con *opcode* = 9, *command* = *800* y *subcommand* = *11* (este último insertado en la lista de enteros) (ll. 1, 2). Luego, se agregan los números de los arcos a consultar del grafo, en este caso todos (ll. 3 - 5). Después, se agrega el apuntador *ssid* del Grafo de Estado al que serán consultados sus arcos (l. 6). La primera llamada (Obtención del *State Space*) permitió obtener el número de arcos *arcsCalculated* (l. 3) y el apuntador *ssid* del Grafo de Estado (l. 6).

Luego del envío del paquete (l. 8) y de la comprobación del paquete de respuesta (l. 9) se extraen, en caso de respuesta exitosa, los arcos provenientes en el paquete de respuesta implementando el algoritmo ilustrado en la figura 3.8 (ll. 14 - 24). Los arcos se insertan en la lista *arcs* (l. 22) que se utilizada posteriormente para el proceso de conversión del Grafo de Estado a una Máquina de Estado.

Obtención de los marcados muertos

La especificación del protocolo [44] provee el sub-comando 14 (figura 3.9) para evaluar una expresión en lenguaje CPN ML para retornar una lista de nodos de un Grafo de Estado en particular. Es así, como se obtienen los marcados muertos (*dead-markings*), si existen, del Grafo de Estados.

```
Extra call parameters:
  blist= nil
  ilist= nil
  slist= ssid, mlexpr
Return value for failure:
  blist= false
  ilist= TERMTAG=1
  slist= ssid,errmsg
Return value for success:
  blist= true,
  ilist= TERMTAG=1, num-of-nodes, (see below)
  slist= ssid,(see below)

repeat num-of-nodes times
   get_node()
```

Figura 3.9. Documentación para obtención de los marcados muertos [44].

El código 3.3 muestra la implementación realizada en Java/PROSEGA para la llamada necesaria (figura 3.9) para la obtención de los marcados muertos.

```
1  Packet p = new Packet(9, 800);
2  p.addInteger(14);
3  p.addString(ssid);
4  p.addString("ListDeadMarkings()");
5
6  p = channel.send(p);
7
8  p.reset();
9
10 boolean successfulReponse = p.getBoolean() == true && p.getInteger() == 1;
11
12 if(successfulReponse){
13     int numberOfNodes = p.getInteger();
14     String ssid = p.getString();
15
16     if(numberOfNodes > 0){
17         for(int i = 0; i < numberOfNodes; i++){
18             Node node = getNode(p);
19             if (node != null){
20                 deadMarkings.add(node);
21             }
22         }
23     }else{
24         logger.info("It wasn't found dead-markings on consulted State Space...");
25     }
26     return true;
27 }else{
28     return false;
29 }
```

Código 3.3. Llamada de Java/PROSEGA para obtención de los marcados muertos.

Esta implementación posee la misma estructura que las dos llamadas previas. Esta llamada en particular, envía el subcomando 14 en la lista de enteros (l. 2). Se envía en la lista de strings el identificador *ssid* para hacer saber al simulador el Grafo de Estado requerido (l. 3) y se envía también en la lista de strings la expresión en CPN ML *ListDeadMarkings( )* para filtrar como respuesta solo los nodos que sean nodos terminales (l. 4). Esta expresión en CPN ML es parte de un conjunto de opciones en CPN ML para la manipulación del Grafo de Estados [33].

Después de la comprobación que la respuesta es exitosa (l. 12) se agregan los nodos resultantes a una lista denominada *deadMarkings* (ll. 13 - 25) que podrá utilizar posteriormente el usuario para incluir, si el usuario lo desea, estos marcados muertos a la lista de nodos terminales de la Máquina de Estado Finito a construir.

Feedback al usuario de las llamadas

En paralelo con las llamadas que se realizan para la obtención del *State Space*, de los arcos y de los marcados muertos, Java/PROSEGA despliega al usuario una interfaz, una ventana de progreso (figura 3.10), que informa al usuario del proceso de comunicación que está teniendo con el simulador de CPN Tools.

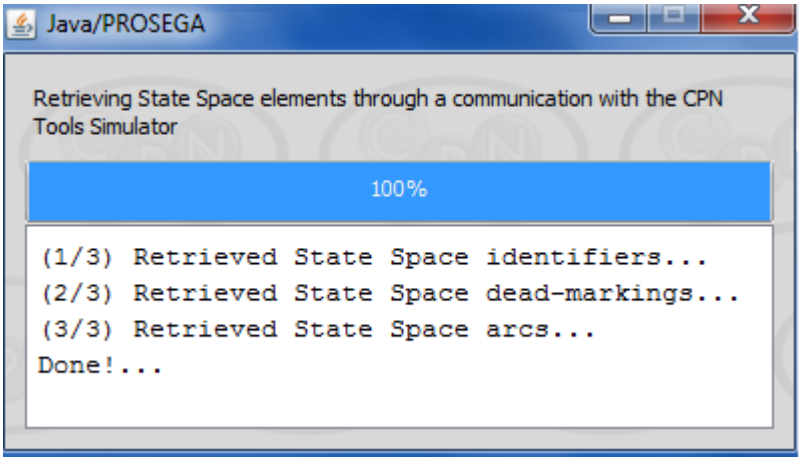

Figura 3.10. Ventana de progreso de Java/PROSEGA de llamadas al simulador.

También, si el usuario no ha calculado previamente el Grafo de Estados mediante el método tradicional provisto por el software por CPN Tools, Java/PROSEGA muestra la siguiente ventana de error al usuario (figura 3.11) indicando que primero debe calcular el Grafo de Estado del modelo cargado en CPN Tools previo al uso de Java/PROSEGA.

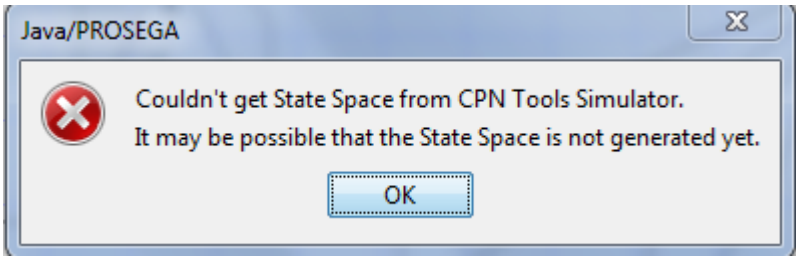

Figura 3.11. Mensaje de error de Java/PROSEGA sobre la obtención del *State Space*.

3.3.3 Interfaz de asignación de identificadores

Luego de las llamadas realizadas para la obtención del identificador del Grafo de Estado, los arcos y los marcados muertos (si posee) , si estas llamadas al simulador de CPN Tools fueron exitosas, Java/PROSEGA despliega una interfaz de asignación de identificadores. Esta interfaz se encarga de proveer al usuario un entorno que permita la asignación adecuada de parámetros y opciones para la conversión del Grafo de Estado obtenido a una Máquina de Estado Finito.

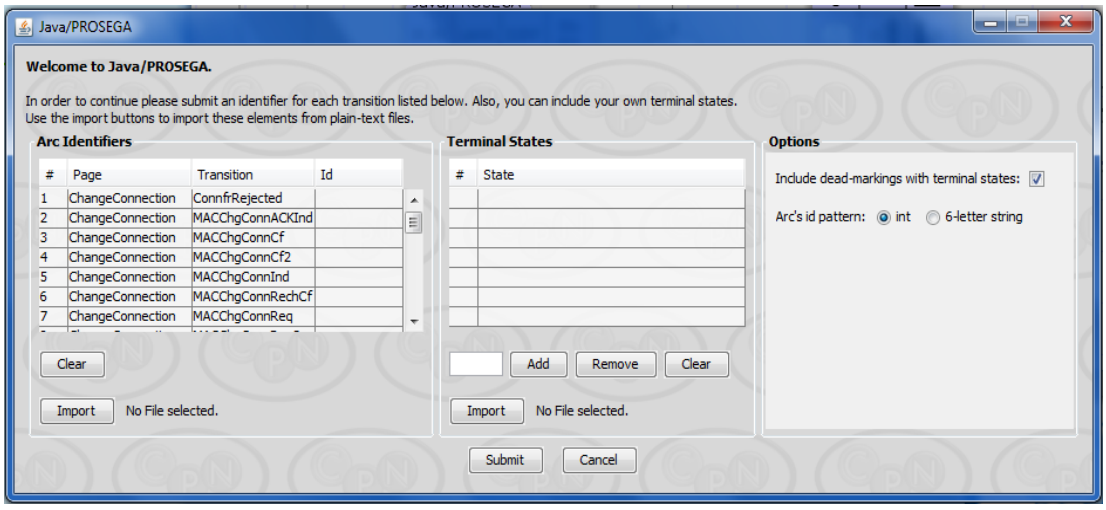

Figura 3.12. Interfaz de asignación de identificadores.

La figura 3.12 muestra la interfaz de asignación de identificadores. Está compuesta de tres secciones principales: La sección de identificadores de arcos (izquierda) que se encarga de asignar a cada arco del Grafo de Estados su correspondiente identificador, la sección de nodos terminales (centro) que permite al usuario agregar nodos terminales y una sección de opciones misceláneas (derecha).

La intención de esta interfaz radica en que el usuario pueda realizar los ajustes necesarios de manera que el Grafo de Estado obtenido asociado al modelo que esté cargado en CPN Tools sea visto como una Máquina de Estado (con sus respectivos arcos y nodos terminales) para así ejecutar efectivamente el proceso de reducción.

En la sección inferior de la interfaz se provee un botón de cancelar *(Cancel)* para abortar el proceso de reducción y un botón de aceptación (*Submit*) para ingresar de manera efectiva todas las configuraciones hechas por el usuario en la interfaz y ejecutar así el proceso de reducción del Grafo de Estado a la Máquina de Estado.

Sección de Identificadores para arcos (*Arc identifiers*)

Los arcos del Grafo de Estado pueden ser vistos como arcos de una máquina de estados, de allí que sea factible el proceso de reducción. Para hacer esta conversión, a cada arco del Grafo de Estados se le asigna un identificador.

Cada arco del Grafo de Estado representa la ocurrencia de una transición dentro del modelo CPN que esté cargado en CPN Tools. Por lo tanto, la sección de identificadores para arcos modela cada arco del Grafo de Estado como la transición que representa y la página en la cual se ubica (figura 3.18 -izquierda-) desplegando estas transiciones mediante una tabla. Esta tabla permite al usuario entonces insertar un identificador en la columna de cada fila. El identificador puede ser un valor de tipo entero positivo o de tipo string.

Esta sección incluye un botón de limpieza denominado *Clear* que provee la funcionalidad de limpiar todos los identificadores que pudo haber colocado el usuario en la tabla previamente.

También se incluye el botón *Import* que provee la funcionalidad de importar los identificadores de los arcos de un archivo de texto que siga el formato de "Archivos para la asignación de identificadores a los arcos del Grafo de Estado**"** explicado en la Sección 3.3.2 del presente trabajo. De esta manera, el usuario puede cargar todos los identificadores de manera automática evitando así la colocación manual de estos.

Sección de nodos terminales (*terminal states*)

Dado que el objetivo de esta interfaz es permitir al usuario realizar las configuraciones necesarias de manera que pueda transformar el Grafo de Estado a una Máquina de Estado Finito se provee esta sección (figura 3.18 -centro-) de nodos terminales que permite al usuario ingresar los nodos del grafo que el considere son nodos terminales (o estados terminales).

Esta sección provee una tabla donde se permite al usuario agregar manualmente los nodos. Además, provee botones para eliminar uno (*delete*) o todos (*clear*) los nodos que haya insertado en la tabla.

Adicionalmente, esta sección contiene un botón *Import* que permite importar de manera automática los nodos terminales desde un archivo de texto que siga el formato de "Archivo de nodos terminales" descrito en la Sección 3.3.2 del presente trabajo.

Sección de opciones

La sección de opciones (figura 3.18. -derecha-) provee un conjunto de opciones misceláneas para el usuario que permiten realizar distintas configuraciones concernientes al proceso de reducción del Grafo de Estado a un FSM.

Esta sección presenta actualmente dos opciones generales, sin embargo, en el trabajo a futuro que se desarrolle con el software pueden ser incluidas más opciones en función de los requisitos que se planteen. A continuación, se mencionan las dos opciones provistas en la interfaz:

- **Inclusión de marcados muertos:** Esta primera opción desplegada gráficamente mediante un *check-box* permite al usuario agregar los marcados muertos (*dead-markings*) al conjunto de nodos terminales para el momento que se ejecute el proceso de reducción.

- **Patrón del identificador del arco:** Permite escoger al usuario el tipo de dado de los identificadores del arco de la Máquina de Estado. Las opciones disponibles son: identificador de tipo entero positivo o identificador de tipo string.

3.3.4 Proceso de minimización

El proceso de minimización de Java/PROSEGA es ejecutado luego que el usuario introduzca las configuraciones deseadas a través de la previa interfaz de asignación de identificadores. En esta interfaz previa el programa permitió al usuario realizar las configuraciones necesarias para que el usuario transformara el Grafo de Estado asociado al modelo CPN que esté cargado en CPN Tools a un FSM. Una vez el usuario ingresa dichas configuraciones se procede al proceso de minimización.

El proceso de minimización de Java/PROSEGA está basado en el algoritmo desarrollado en [3] y propuesto formalmente por Billington et. al. [4] como parte del proceso de verificación de protocolos. Ha sido probado exitosamente en trabajos tales como [10] [22] [39].

El algoritmo de minimización está compuesto de tres operaciones principales: remoción de epsillons (arcos que contengan la cadena vacía), determinización del autómata y minimización, en ese orden. La herramienta OpenFST implementa estas tres operaciones a través de las funciones *fstrmepsilon*, *fstdeterminize* y *fstminimize* La explicación del funcionamiento de cada una de estas operaciones puede ser vista en la Sección 2.10.4 del presente trabajo.

Para utilizar las operaciones de Java/PROSEGA para la minimización, se toma primero el Grafo de Estados (ahora una Máquina de Estados) y se compila al formato binario de archivos de OpenFST utilizando el comando *fstcompile*. Luego de la compilación del Grafo de Estados, Java/PROSEGA se encarga de invocar estas funciones y aplicarlas sobre el Grafo de Estados (ahora una Máquina de Estados) para convertir este en una Máquina de Estado Finito minimizada.

Luego que Java/PROSEGA invoca las funciones *fstrmepsilon*, *fstdeterminize* y *fstminimize* se obtiene el FSM minimizado. Este se genera en un archivo compilado de extensión .fst (ver Sección 2.10.3). Para desplegar visualmente la Máquina de Estado en la interfaz de Java/PROSEGA se utiliza el comando *fstdraw* (ver Sección 2.10.4) para transformar el FSM minimizado a un archivo de extensión .dot para que posteriormente este archivo de extensión .dot sea consumido por el software Graphviz (ver Sección 2.11) utilizando el comando *dot* y así generar finalmente una imagen del autómata minimizado.

En resumen, se presenta la siguiente tabla con los comandos en el orden que son invocados por Java/PROSEGA para la minimización del Grafo de Estado a un FSM minimizado, mostrando como son invocados por Java/PROSEGA

Tabla 3.3. Comandos del proceso de minimización de Java/PROSEGA

#	comando	Ejemplo de invocación por Java/PROSEGA
1	**fstcompile**	fstcompile --acceptor=true --isymbols=syms.txt text.txt bin.fst
2	**fstrmepsilon**	fstrmepsilon bin.fst rmeps.fst
3	**fstdeterminize**	fstdeterminize rmeps.fst det.fst
4	**fstminimize**	fstminimize det.fst min.fst
5	**fstdraw**	fstdraw --acceptor=true min.fst dot.dot
6	**dot**	dot -Tjpeg -Gdpi=300 -Grotate=180 dot.dot -o ps.jpeg

Feedback al usuario del proceso de minimización

En paralelo con el proceso de minimización, se muestra al usuario una ventana de progreso que provee al usuario un *feedback* acerca de la ejecución del proceso de minimización informándole de las operaciones que se van ejecutando.

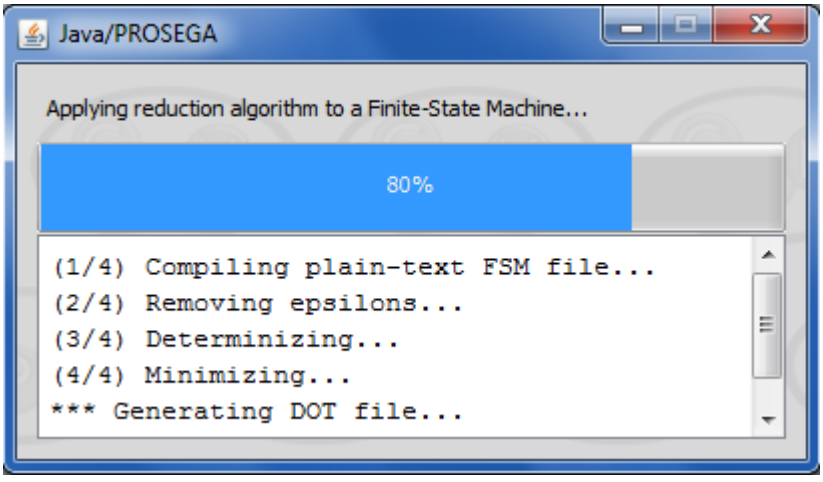

Figura 3.13. Ventana de progreso de Java/PROSEGA de minimización.

3.3.5 Archivos de salida

Cuando se ejecuta el proceso de minimización, Java/PROSEGA genera una carpeta con el nombre *fsm_<date>* donde *<date>* es la fecha y hora de la ejecución de la rutina. Esta carpeta se genera en la carpeta *out* del directorio de Java/PROSEGA. En esta carpeta se van generando los archivos utilizados en el proceso de minimización. En la figura 3.14 se ilustra un ejemplo de una carpeta generada por Java/PROSEGA y los archivos que esta carpeta contiene luego de la ejecución del proceso de minimización.

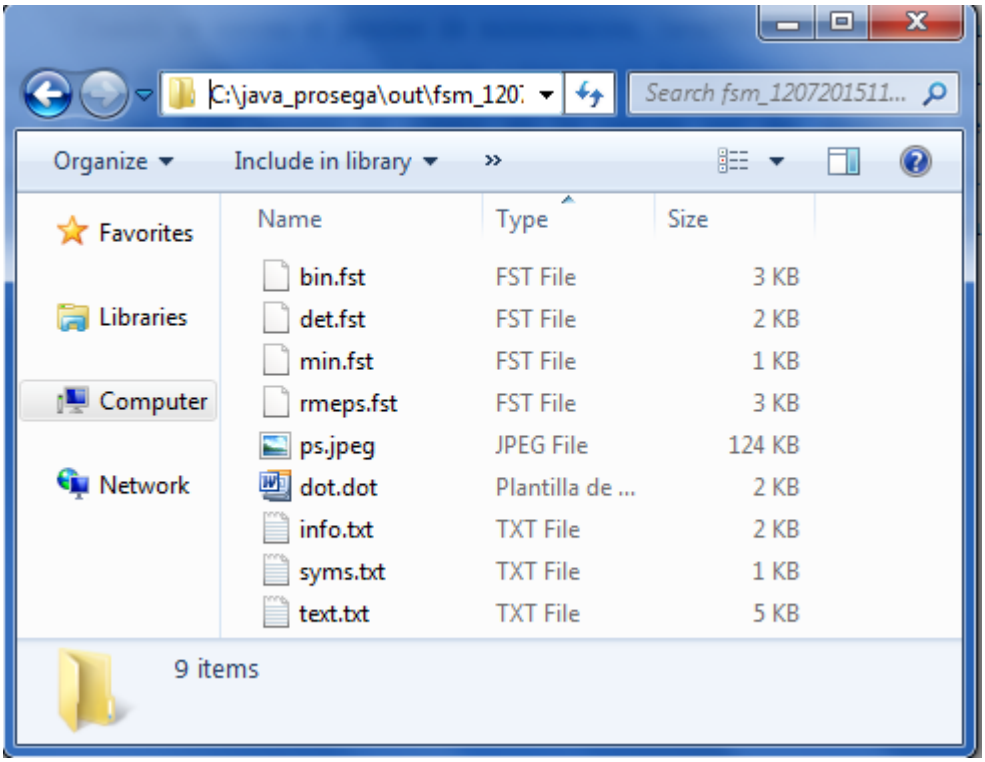

Figura 3.14. Carpeta generada por el proceso de minimización de Java/PROSEGA.

La siguiente tabla describe cada uno de los archivos que están contenidos en la carpeta que se genera en el proceso de minimización de Java/PROSEGA. Los archivos están ordenados en la tabla en función del momento en el cual fueron generados por la herramienta. La tabla 3.3 describe los comandos ejecutados por Java/PROSEGA para generar los siguientes archivos:

Tabla 3.4. Archivos generados por la rutina de minimización de Java/PROSEGA.

nombre	formato	descripción
text	txt	Es la representación del Grafo de Estado en un archivo en texto plano con el formato de representación de autómatas definido por OpenFST.
syms	txt	Es el archivo de símbolos que asocia a cada arco del Grafo de Estados un identificador numérico o de tipo *string*.
bin	fst	Es el resultado de haber aplicado la función *fstcompile* sobre el archivo *text* transformando así el Grafo de Estados (ahora en formato FSM) a un archivo compilado.
rmeps	fst	Es el resultado de haber aplicado la remoción de epsillons (*fstrmepsilon*) al archivo *bin*.
det	fst	Este archivo es el resultado de aplicar la función de determinización (*fstdeterminize*) sobre el archivo *rmeps*.
min	fst	Este archivo representa el FSM resultante de aplicar la función *fstminimize* sobre el archivo *det*. Este archivo contiene por lo tanto, el FSM final del proceso de minimización.
dot	dot	Este archivo es el archivo necesario por Graphviz para imprimir la imagen del autómata resultante. Es obtenido al aplicar la función *fstdraw* sobre el archivo *min*.
ps	jpeg	Es la imagen en formato JPEG con una resolución de 300 dpi del autómata generado del proceso de minimización. Se obtiene aplicando la función *dot* sobre el archivo *dot* (archivo que representa en lenguaje *DOT* el autómata minimizado generado en el archivo *min*)
info	txt	Es un archivo que tiene información general sobre el autómata resultante del proceso de minimización. La información de este archivo es desplegada en la interfaz de resultados.

3.3.6 Interfaz de resultados

Una vez Java/PROSEGA culmina el proceso de minimización descrito en la sección anterior, el software despliega una interfaz de resultados donde coloca el FSM minimizado resultante, información general sobre la máquina (mediante el comando *fstinfo de* OpenFST) y la asociación de los identificadores de los arcos de la máquina con los arcos del Grafo de Estado (u ocurrencias de las transiciones del modelo CPN) de manera que el usuario pueda realizar un análisis del modelo sobre la máquina generada.

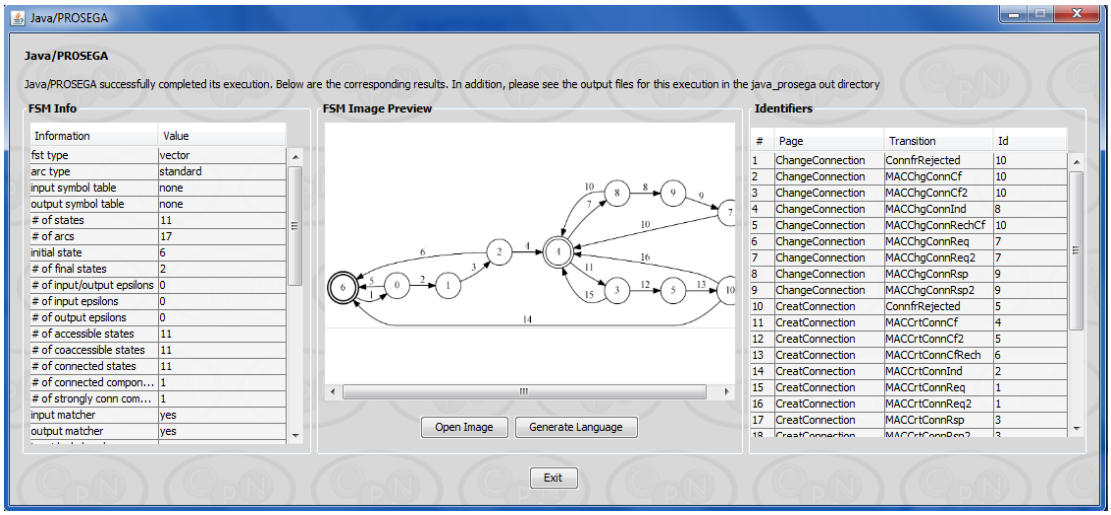

Figura 3.15. Interfaz de resultados de Java/PROSEGA.

La figura 3.15 muestra la interfaz de resultados del software Java/PROSEGA. Consta de tres secciones o paneles principales. En el panel izquierdo se muestra al usuario la información general de la Máquina de Estado Finito resultante del proceso minimización (obtenida mediante el comando *fstinfo*). En el panel central de la interfaz se muestra el autómata resultante dibujado mediante Graphviz.

El panel derecho provee una asociación entre los arcos del Grafo de Estados (o transiciones del modelo CPN) y los respectivos identificadores inscritos en los arcos del FSM permitiendo así al usuario realizar un análisis del resultado obtenido.

Además, la interfaz contiene los siguientes botones que proveen las siguientes funcionalidades: *OpenImage* que permite abrir la imagen del autómata utilizando el programa determinado de visor de imágenes que se tenga en el sistema operativo (comúnmente el visor de imágenes básico de Windows). El botón *GenerateLanguage* provee un atajo a la función de generación de lenguaje de Java/PROSEGA para así generar el lenguaje asociado a la máquina de estado generada. El botón *Exit* cierra la ventana de resultados permitiendo al usuario cerrar el proceso.

3.4 Generador de lenguajes

El generador de lenguajes es otra funcionalidad provista por el software Java/PROSEGA. Esta funcionalidad es accedida mediante el instrumento LANG de la caja de herramientas de Java/PROSEGA en el editor gráfico de CPN Tools.

El generador de lenguajes está compuesto de dos funciones principales: La generación del lenguaje de un autómata (impresión de todo el lenguaje o un subconjunto) y la generación de cadenas aleatorias que pertenecen al lenguaje reconocido por un autómata en particular.

Este generador se apoya en las funciones de la librería fsm2language desarrollada por el autor del presente trabajo para la implementación de este generador. En esta sección se describe el uso a nivel de interfaz gráfica del generador del lenguaje. Para una descripción del funcionamiento de la librería fsm2language ver la Sección 3.6 del presente trabajo.

Para acceder a cualquiera de las dos funciones (generación del lenguaje o generación de cadenas aleatorias) se debe ingresar el autómata (de tipo texto-plano o compilado) mediante la interfaz de entrada provista por este generador (ver figura 3.16).

En la sección superior de la interfaz se provee una opción que permite al usuario especificar el tipo de Máquina de Estado Finito de entrada de acuerdo al formato en que esté representada (de tipo texto-plano o compilada). La sección media de la interfaz permite al usuario ingresar la máquina y la sección inferior permite ingresar el archivo de símbolos que utiliza la máquina de estados. Este archivo de símbolos representa el mapeo entre los arcos y los símbolos de entrada, y utiliza el formato de "Archivo de símbolos" explicado en la Sección 3.2.

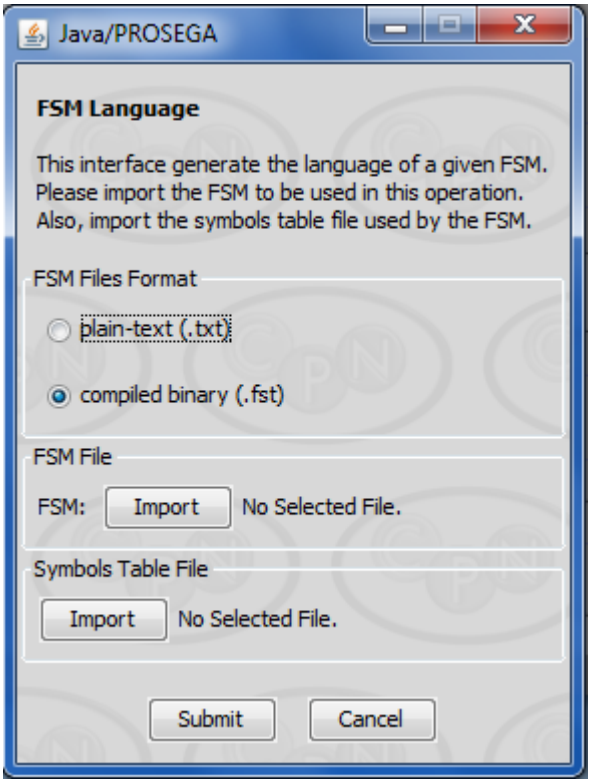

Figura 3.16. Interfaz de entrada del generador del lenguaje.

3.4.1 Generación del lenguaje

Esta función del generador del lenguaje imprime todas las cadenas pertenecientes al lenguaje de un autómata (o Máquina de Estado Finito) que sea pasado como parámetro. El autómata que es pasado como parámetro a esta interfaz debe estar contenido en un archivo ya sea en formato texto-plano siguiendo el estándar de representación autómatas de OpenFST o en un formato compilado de OpenFST en un archivo de extensión .fst (ver Sección 2.10.3)

Es posible que el lenguaje que es reconocido por una Máquina de Estado Finito sea infinito. Esto significa que existen infinitas cadenas que son aceptadas por dicha máquina. Un caso particular que motiva que esto ocurra son las máquinas que poseen ciclos (autómatas cíclicos). En este caso, la función de generación del lenguaje se encargará de imprimir un subconjunto del lenguaje reconocido por el autómata.

Para el caso de lenguajes infinitos, la interfaz de generación del lenguaje imprimirá entonces un subconjunto del lenguaje tal que las cadenas pertenecientes a este lenguaje fueron obtenidas mediante todos los posibles recorridos del autómata en que no se repitan arcos.

Esta interfaz de generación del lenguaje se apoya en gran medida de la función *fsm2language* de la librería fsm2language. Una descripción detallada de cómo esta logra la generación del lenguaje puede ser vista en la Sección 3.6 del presente trabajo.

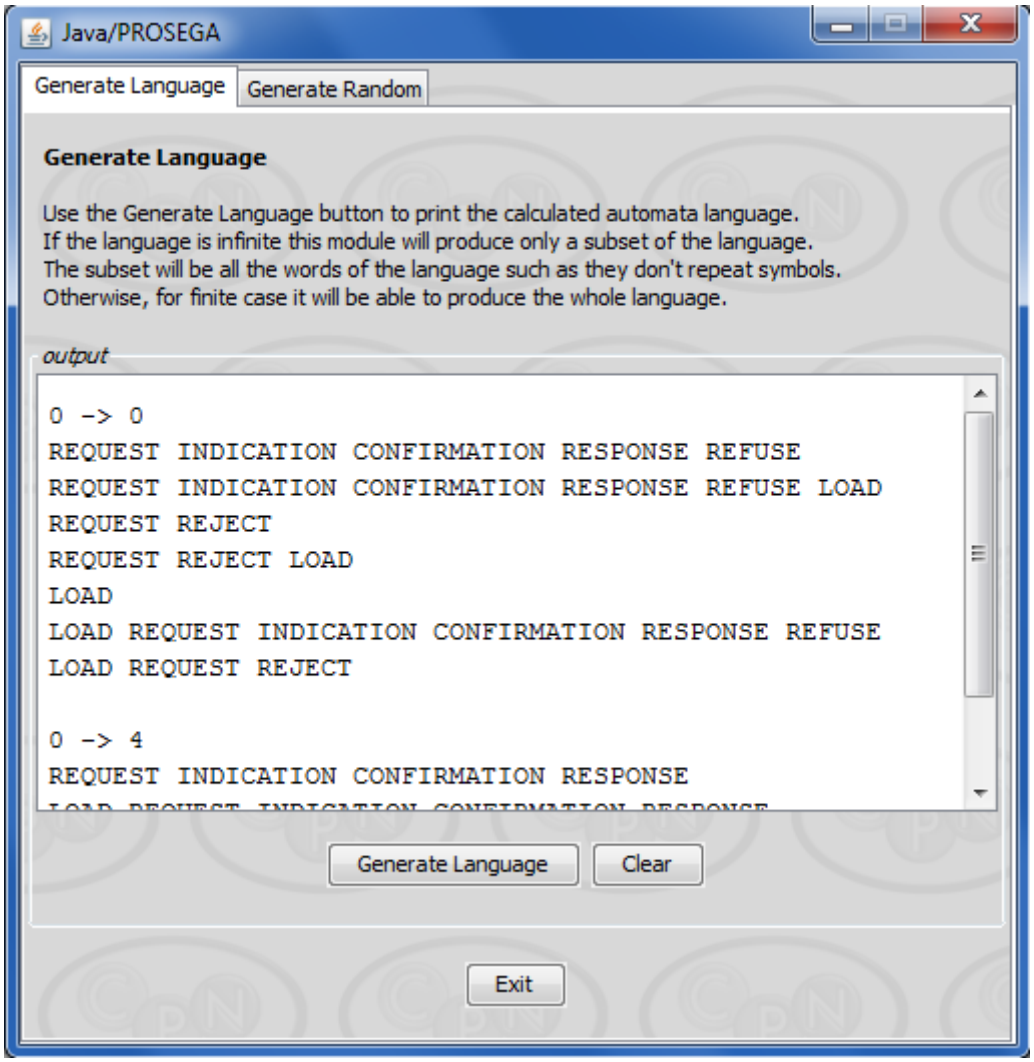

Figura 3.17. Interfaz para la generación del lenguaje.

La figura 3.17 muestra la interfaz de Java/PROSEGA para la generación del lenguaje. La sección superior de la interfaz muestra un mensaje informativo acerca del funcionamiento de esta interfaz. La sección central es un área de texto donde se imprime el lenguaje una vez se de click al botón *Generate Language*. Se disponen adicionalmente los botones *Clear* (para el borrado del área de texto) y *Exit* para salir de la interfaz.

3.4.2 Generación de cadenas aleatorias

La interfaz de generación de cadenas aleatorias (figura 3.18) es la segunda funcionalidad del generador de lenguaje de Java/PROSEGA. Como entrada utiliza el autómata que es pasado a la función de generación del lenguaje a través de la misma interfaz de entrada (ver figura 3.16). Esta función permite imprimir palabras (o cadenas) aleatorias que pertenezcan al lenguaje del autómata pasado como parámetro.

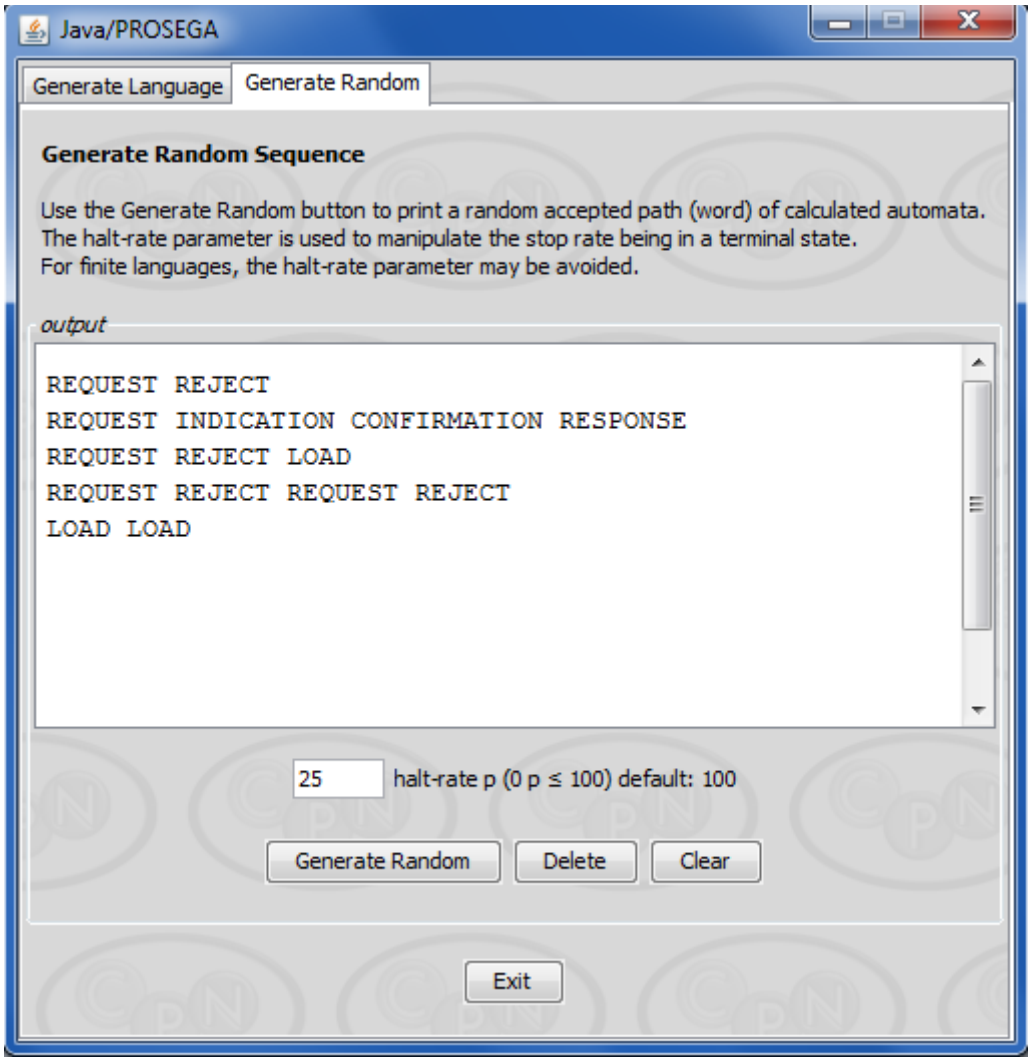

Figura 3.18. Interfaz para la generación de cadenas aleatorias.

En la sección superior de la interfaz se muestra un mensaje informativo acerca del funcionamiento de esta operación. En la sección media se despliega el área de texto donde se van generando las cadenas aleatorias. Además, se provee al usuario la posibilidad de insertar un parámetro de entrada definido como *halt-rate* (o probabilidad de parada). Este parámetro define de manera probabilística el tamaño de la cadena a generar en función del recorrido que se realiza al autómata (véase Sección 3.6.2).

La interfaz posee los siguientes botones: *GenerateRandom* que permite la generación de una cadena aleatoria, *Delete* que elimina la última cadena generada y *Clear* que elimina del área de texto todas las cadenas aleatorias que hayan sido generadas.

Esta interfaz se apoya en la función *fsm2random* de la librería fsm2language desarrollada por el autor de este trabajo. En la Sección 3.6.2 del presente trabajo se encuentra una explicación del funcionamiento de este comando y de cómo fue implementado.

3.5 Operación de diferencia sobre Máquinas de Estado Finito

La operación de diferencia sobre Máquinas de Estado Finito es una funcionalidad de Java/PROSEGA que es provista a través del instrumento DIFF de la caja de herramientas de Java/PROSEGA dentro del editor gráfico de CPN Tools. Su utilidad radica en aplicar la función de diferencia sobre máquinas de estado para así realizar análisis y comparaciones. Para llevar a cabo esta tarea, Java/PROSEGA utiliza la función de diferencia *fstdifference* provista por la librería OpenFST.

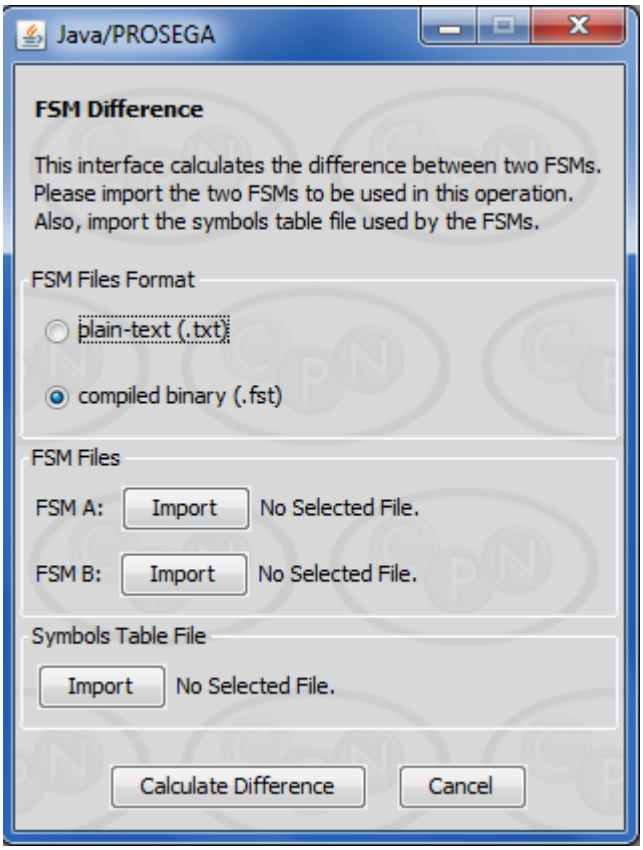

Figura 3.19. Interfaz de entrada de la función de diferencia de Java/PROSEGA.

La figura 3.19 presenta la interfaz de entrada de la función de diferencia de Java/PROSEGA. Esta interfaz de entrada es simular a la utilizada por la interfaz de entrada del generador del lenguaje, pero en esta interfaz se agrega un autómata más a la entrada. Esta interfaz permite al usuario decidir el tipo del formato de los dos autómatas de entrada (de tipo texto-plano o en un formato compilado). Se provee un área para importar los dos archivos que corresponden a los autómatas A y B a ser comparados. Además, se provee un área para importar el archivo de símbolos (que debe ser el mismo para los dos autómatas). Cuando se hace click en el botón *Calculate Difference* se procede a computar el autómata resultante (A - B).

La figura 3.20 muestra la interfaz de resultado del cómputo de la diferencia entre un autómata A y un autómata B. En la sección superior indica un cuadro informativo acerca del resultado. La sección central de la interfaz muestra la máquina resultante que es la diferencia entre el autómata A y el autómata B. Además, se provee el botón *OpenImage* para visualizar la máquina resultante en el programa de visor de imágenes predeterminado que se tenga en el sistema operativo (generalmente el visor de imágenes de Windows).

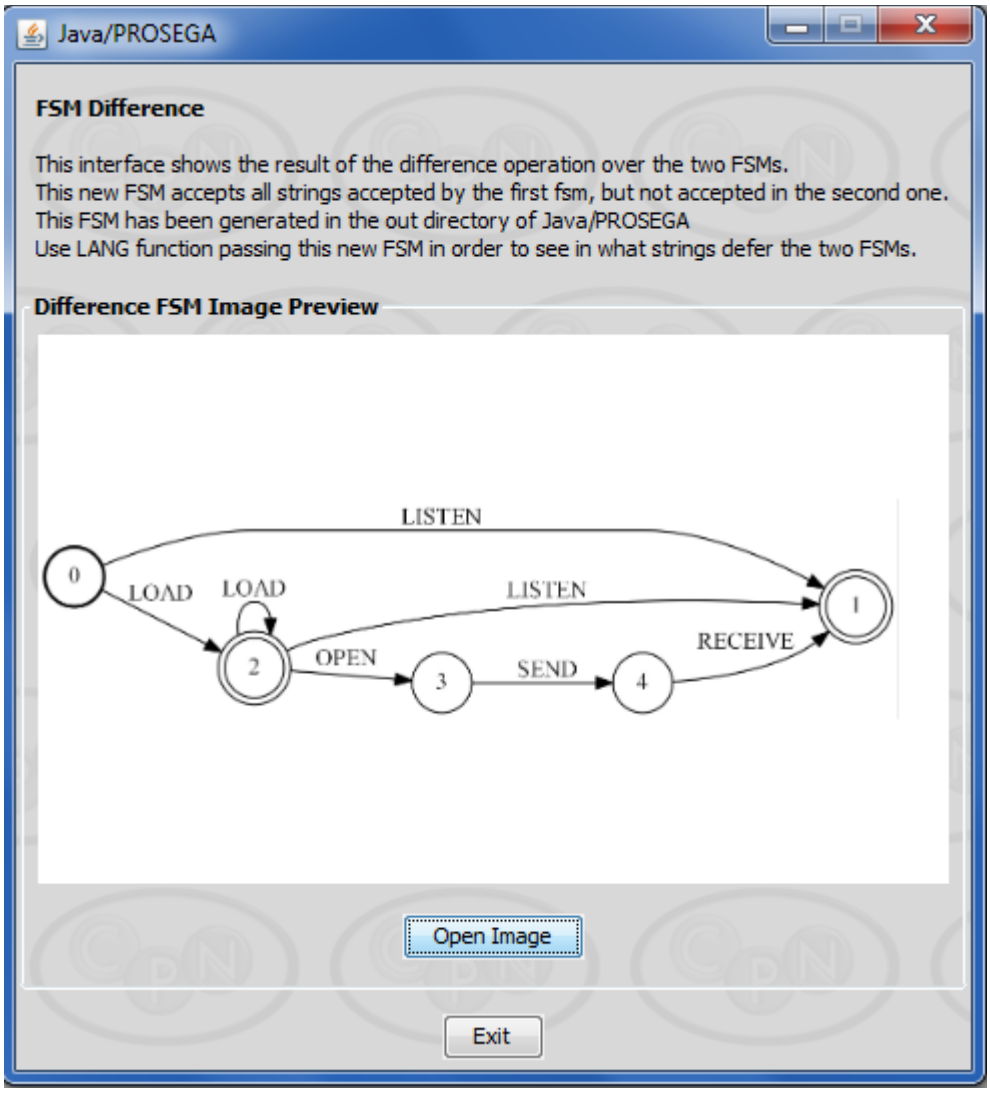

Figura 3.20. Interfaz de resultado de la función de diferencia de Java/PROSEGA.

Vale destacar, que la operación de diferencia entre autómatas puede arrojar una máquina de estado vacía, es decir, sin nodos. Esto significa que no existe una diferencia entre los dos autómatas de entrada. En caso que suceda esta condición, Java/PROSEGA muestra la siguiente interfaz (ver figura 3.21) con una indicación que corresponde a la sección central de la interfaz de resultado que informa que el resultado de aplicar la diferencia entre los dos autómatas pasados como parámetros es vacío.

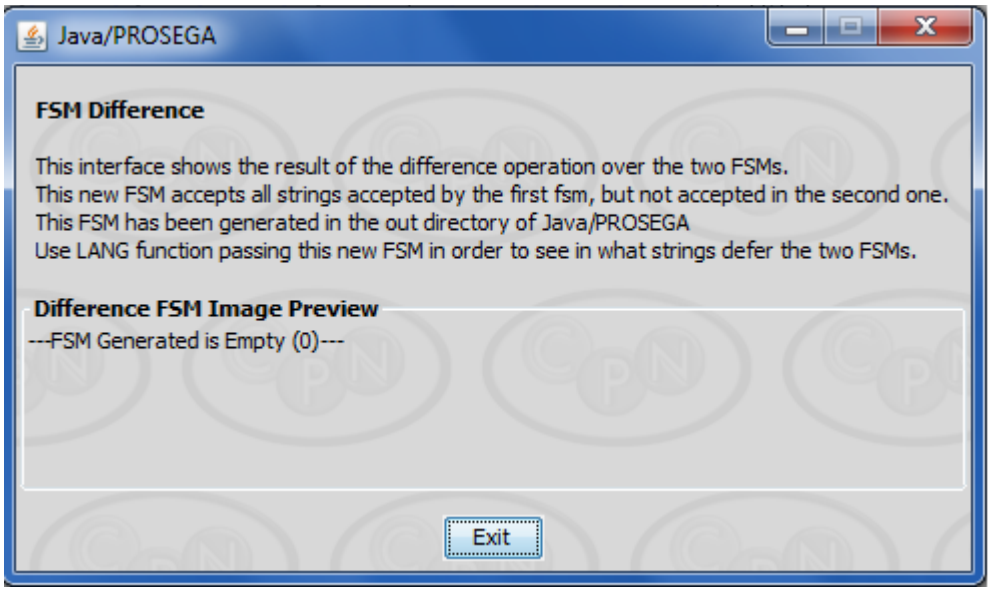

Figura 3.21. Interfaz de resultado vacío en la función de diferencia.

En paralelo con el despliegue de la interfaz del resultado, Java/PROSEGA crea una carpeta dentro del directorio *out* de Java/PROSEGA que contiene los archivos que se generan a partir del cálculo de la diferencia. El nombre de esta carpeta generada tiene el formato *diff_<date>* tal que *<date>* es la fecha y hora de la ejecución de esta función. A continuación, se muestra la figura 3.22 que muestra un ejemplo de una carpeta generada por esta función de diferencia al momento de ejecutarse.

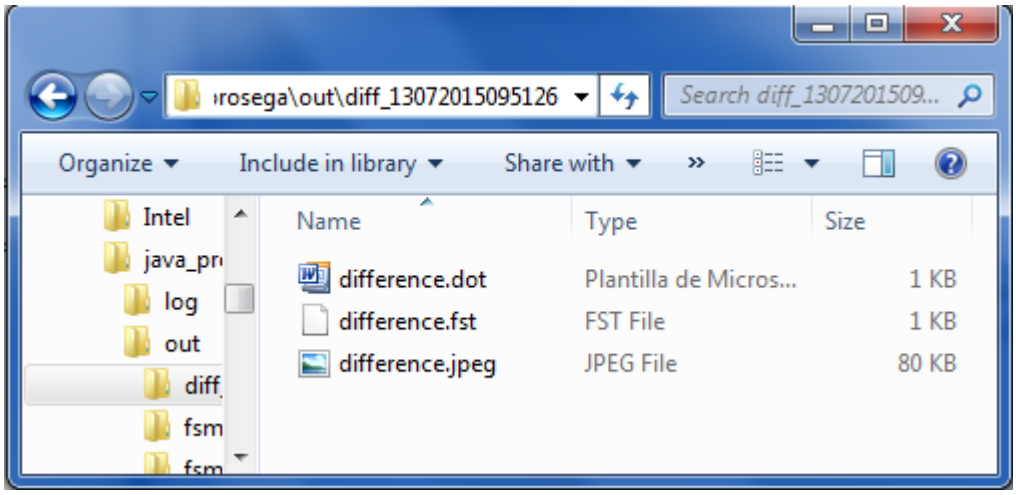

Figura 3.22. Carpeta generada por la función diferencia de Java/PROSEGA.

La carpeta que se genera a raíz de la ejecución de la función de diferencia en Java/PROSEGA contiene la máquina de estados resultante (el autómata que representa la diferencia de los dos autómatas de entrada) en tres archivos en formatos distintos: El primer archivo, y principal, es la máquina de estado en el formato compilado de OpenFST (*difference.fst*). El segundo archivo (*difference.dot)* es la máquina de estado representada en el lenguaje de marcado DOT que generó Java/PROSEGA a partir de utilizar la función *fstdraw* de OpenFST sobre el archivo *difference.fst*. Finalmente, el tercer archivo (*difference.jpeg*) es la máquina de estados en un formato de imagen JPEG que generó Java/PROSEGA utilizando la función *dot* de Graphviz sobre el archivo *difference.dot*. La generación de este directorio permite al usuario tomar la representación de la diferencia generada en cualquiera de estos formatos para utilizar el autómata generado en otro entorno o aplicación.

La Sección 4.2.3 del presente trabajo abarca una definición teórico-práctica sobre la función de diferencia *fstdifference* que utiliza Java/PROSEGA para el computo de la diferencia entre dos autómatas utilizando como base un caso de estudio en particular.

3.6 Librería fsm2language

La librería fsm2language es una librería desarrollada en el lenguaje de programación C por el autor del presente trabajo. Esta librería es utilizada por el software Java/PROSEGA para llevar a cabo las tareas de generación del lenguaje (véase Sección 3.4). Java/PROSEGA invoca las funciones de esta librería tal como invoca las demás funciones de las otras librerías de terceros utilizadas en el presente trabajo (OpenFST, Graphviz). Sin embargo, esta librería fue desarrollada para que también pueda funcionar de manera independiente, fuera del entorno de Java/PROSEGA, utilizando la consola de comandos de Windows.

La librería se compone solo de dos funciones (o comandos): *fsm2language* y *fsm2random*. La función *fsm2language* imprime el lenguaje reconocido por una Máquina de Estado Finito (o un subconjunto del lenguaje si este es infinito). La función *fsm2random* imprime una cadena aleatoria del lenguaje reconocido por una Máquina de Estado Finito.

A continuación, se explica la representación desarrollada para manipular las máquinas de estado finito en ambas funciones. Luego, se explican las dos funciones hasta ahora desarrolladas para la librería y como fueron implementadas, explicando la lógica del funcionamiento de la librería utilizando un autómata de ejemplo. Finalmente, el anexo “D” del presente trabajo muestra el código fuente en lenguaje C de las dos funciones desarrolladas para esta librería.

3.6.1 Representación de los autómatas

Los autómatas (o Máquinas de Estado Finito) que son pasados como parámetros de entrada a cualquiera de las funciones de la librería *fsm2language* están representados a través de un arreglo dinámico de listas de adyacencia [59].

La figura 3.23 muestra como ejemplo un autómata que modela el comportamiento de un modelo de comunicación en particular.

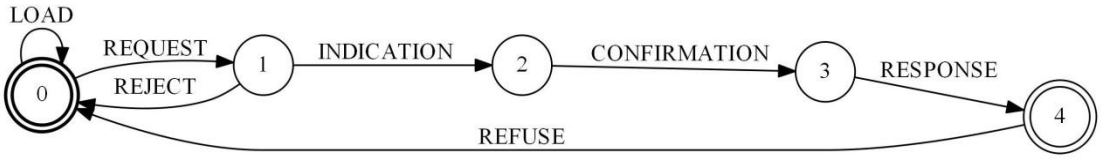

Figura 3.23. Ejemplo de autómata para la librería fsm2language.

La siguiente figura ilustra como este autómata es representado computacionalmente en las funciones de la librería fsm2language a través de un arreglo de listas de adyacencias. Para un autómata de *N* nodos, se tendrá un arreglo de dimensión *N* tal que cada entrada representa un nodo del autómata. A su vez, para un nodo *i, i = 0, 1,..., N* se tiene una lista de adyacencia *i* cuyos elementos apuntan a los nodos del autómata a los cuales el nodo *i* está directamente conectado. Además, estos elementos de la lista de adyacencia guardan el valor del arco que conecta al nodo *i* con el nodo adyacente.

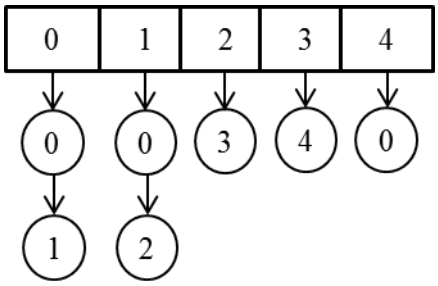

Figura 3.24. Ejemplo de representación del autómata con listas de adyacencia.

Pese a que la tarea de acceder a los arcos del autómata no tendrá una complejidad en tiempo O(1), con esta representación se realiza un ahorro importante en memoria (a diferencia de utilizar matrices ponderadas [55]) permitiendo así la construcción y manipulación de autómatas que posean un alto número de nodos.

Las funciones de la librería *fsm2language* consumen como entrada, la representación en formato tipo texto-plano de las autómatas utilizadas por la librería OpenFST. Luego, cuando se ejecuta alguna de las funciones, previo a la tarea en particular de la función, se procede a transformar la máquina de estado contenida en el archivo en formato tipo texto-plano a la representación descrita previamente.

3.6.2 Función fsm2language

La función *fsm2language*, de la librería del mismo nombre, se encarga de, dada una Máquina de Estado Finito (o FSM), imprimir el lenguaje que reconoce esta máquina. Esto es imprimir todas las cadenas que son aceptadas por dicha máquina.

Si el lenguaje de la máquina de entrada es infinito, es decir, si existen infinitas cadenas que pueden ser aceptadas por la máquina de estado entrante, entonces se imprimirá un subconjunto del lenguaje tal que dicho subconjunto contiene solo las cadenas que en su procesamiento (recorrido al autómata) no fue repetido algún arco.

Matemáticamente, la función puede definirse de la siguiente manera: Sea M una Máquina de Estado Finito y sea $L(M)$ el conjunto de palabras (o cadenas) aceptadas por M, entonces:

$$fsm2language(M) = \begin{cases} L(M) & si\ L(M) \neq \infty \\ L'(M) & si\ L(M) = \infty \end{cases}$$

Donde el subconjunto $L'(M) \subseteq L(M)$ es un conjunto de cadenas u, compuestas por los símbolos $u = x_1 x_2 \ldots x_n$ tal que x_i es el símbolo del arco que va desde el nodo a hasta el nodo b, y se cumple que $\forall x_i,\ x_j\ (a, b) \neq (c, d)\ \ i, j = 0, 1, \ldots, n$ siendo (c, d) el arco que contiene el símbolo x_j y que va desde un nodo c hasta un nodo d.

La definición anterior significa que las cadenas pertenecientes al conjunto $L'(M)$ a imprimir solo serán cadenas cuyos símbolos pertenecen a arcos distintos. Es decir, en la generación de cada palabra de este conjunto solo será visitará un arco.

Para generar una palabra aceptada por un autómata, se debe comenzar el procesamiento desde el nodo inicial hasta un nodo terminal. A medida que se va recorriendo el autómata se van guardando en un buffer los arcos visitados, y en cada visita a un arco del autómata se verifica que este no haya sido ya almacenado en el buffer. Cuando se arriba al nodo terminal deseado, se para el procesamiento y se imprime el símbolo de cada arco guardado en el buffer en el orden en el que fueron colocados de manera que en conjunto sean una palabra.

Para imprimir todas las palabras aceptadas por un autómata, o el subconjunto previamente explicado para el caso del lenguaje infinito, se deben realizar todos los posibles recorridos desde el nodo inicial del autómata hasta cada uno de los nodos terminales. Para llevar a cabo esta tarea, la función *fsm2language* implementó el algoritmo clásico de recorrido de grafos, Búsqueda en Profundidad (DFS, *Depth-First Search*) [59] con unas ligeras modificaciones.

Esta variante implementada del algoritmo DFS radica principalmente en que, en vez de imprimir los nodos del autómata, sean impresos los arcos (o más bien los símbolos que contienen) de manera imprimir así un conjunto de símbolos que representan en sí una palabra aceptada por el lenguaje.

Para una Máquina de Estado Finito que tenga k estados terminales se realizaran entonces k algoritmos de búsqueda en profundidad que permitan generar todos los caminos (palabras) posibles entre el nodo inicial de la máquina y el estado terminal i tal que $i = 1, 2, ..., k$.

Otra variación resaltante de la implementación realizada es que para reducir la complejidad en tiempo de ejecución del algoritmo también se transformó este algoritmo de naturaleza recursiva a una versión iterativa mediante el manejo de una pila. A continuación, el código 3.4 muestra una versión en pseudo-código del algoritmo desarrollado en la función *fsm2language.*

```
INT n; //number of nodes
INT start;
INT halt_states[k]; // terminal states
STRING A[n][n]; //finite-state machine

FOR i = 0; i < k; i++ DO
    PRINT_ALL_PATHS_BETWEEN(start, halt_states[i])
END_FOR;

PROCEDURE PRINT_ALL_PATHS_BETWEEN(INT start, INT halt)

    COLLECTION path;
    STACK stack;
    INT depth = 0;

    FOR i = 0; i < n; i++ DO
        IF(A[start][i] IS NOT NULL) THEN
            stack.push((start, i), depth);
        END_IF;
    END_FOR;

    WHILE(stack IS NOT EMPTY) DO
        ARC (a,b) = stack.pop().arc;
        path.push((a,b));

        INT current_node = b;
        IF(b == halt) THEN
            PRINT(path);
        END_IF;

        BOOLEAN new_depth = FALSE;
```

Código 3.4. Algoritmo de generación del lenguaje (1/2).

```
32
33          FOR i = 0; i < n; i++ DO
34              IF( A[current_node][i] IS NOT NULL
35                  AND path DOESN'T CONTAIN((current_node, i)) )THEN
36
37                  stack.push((current_node, i), depth + 1);
38                  new_depth = TRUE;
39              END_IF;
40          END_FOR;
41
42          IF(new_depth == FALSE) THEN
43              INT previous_depth = stack.peek().depth;
44
45              WHILE(size(path) - 1 >= previous_depth AND size(path) >= 1) DO
46                  path.pop();
47              END_WHILE;
48              depth = previous_depth;
49          ELSE
50              depth++;
51          END_IF;
52
53      END_WHILE;
54
55  END_PROCEDURE;
```

Código 3.4. Algoritmo de generación del lenguaje (2/2).

A continuación, se explica el código desarrollado para la generación del lenguaje, o el subconjunto previamente descrito del lenguaje, para Máquinas de Estado Finito. En ll. 1 - 4 se declara el número de nodos del autómata, el estado inicial, los estados de parada y la matriz que representa al autómata. En este pseudocódigo, A es una matriz de dimensión *n x n* tal que la entrada (*i*, *j*) de la matriz contiene el símbolo del arco que va desde el nodo *i* hasta el nodo *j*. Se asume en esta explicación que estas estructuras ya poseen valores (en la práctica estos valores son suministrados por el usuario). En la práctica, para optimizar el uso de la memoria se utiliza una representación de listas de adyacencia en vez de matrices para la representación de los autómatas. Por otra parte, en ll. 6 - 8 se realizan las llamadas para generar todas las palabras posibles (imprimir todos los caminos posibles) desde el nodo inicial hasta cada uno de los nodos terminales del autómata.

En l. 10 comienza la definición del método de impresión de todos los caminos posibles (o palabras aceptadas) entre el nodo inicial y un nodo terminal en específico. El contenedor *path* (l. 12) es un buffer que almacenará los arcos visitados, la estructura *stack* (l. 13) es una pila que contendrá arcos en conjunto con un nivel de profundidad. La variable *depth* permite saber la profundidad actual del algoritmo (que en otro contexto sería el nivel de recursividad).

En ll. 16 - 20 se incluyen en la pila todos los arcos que salgan desde el nodo inicial colocando junto a ellos la marca *depth* = 0. En l. 22 comienza el ciclo de impresión de todos los caminos. Este ciclo se detendrá hasta que no haya más elementos en la pila. En l. 23 se saca el último arco insertado en la pila, y luego se pregunta si este conduce al estado terminal deseado (l. 27) en cuyo caso se imprime todo el buffer que representa una palabra en sí perteneciente al lenguaje del autómata.

Por defecto, se asume que no podrá ser posible ir a un nuevo nivel de profundidad (l. 31). En ll. 33 - 40 se agregan a la pila todos los arcos, si existen, que salgan desde el nodo actual y que no hayan sido ya visitados por el algoritmo (es decir, que no estén contenidos en el buffer). Si se apila al menos un posible arco, la bandera *new_depth* cambia a verdadero indicando que se encontró un arco que sale del nodo actual y que no ha sido visitado en la generación de la palabra.

Por otro lado, si no existió ningún arco que saliera del nodo actual y que no fuera ya visitado, se bajando el nivel de profundidad (ll. 45 - 47) hasta el último nivel de profundidad en que todavía pueda ser posible encontrar un arco alternativo.

La implementación de este algoritmo en el lenguaje C es presentada en la sección D.1 del presente trabajo. En dicho algoritmo desarrollado en C se utiliza la representación de autómatas utilizando listas de adyacencias para un manejo eficiente de la memoria.

3.6.3 Función fsm2random

La función fsm2random se encarga de, dado un autómata, imprimir una palabra aleatoria que pertenezca al lenguaje aceptado por dicho autómata.

```
1  INT n; //number of nodes
2  INT start;
3  INT halt_states[k]; // terminal states
4  STRING A[n][n]; //finite-state machine
5  DOUBLE halt_rate; // between 0 and 1
6
7  PROCEDURE PRINT_RANDOM_PATH(INT initial_state)
8
9      COLLECTION path;
10
11     INT s = initial_state;
12     INT n;
13
14     WHILE(TRUE) DO
15         n = GET RANDOMLY ANY STATE REACHABLE FROM s;
16
17         IF(n IS NULL) THEN
18             BREAK LOOP;
19         END_IF;
20
21         INSERT A[s][n] IN path;
22
23         s = n;
24
25         IF(n IS A HALT STATE) THEN
26             IF(halt_rate > UNIFORM(0,1)) THEN
27                 BREAK LOOP;
28             END_IF;
29         END_IF;
30     END WHILE;
31
32     PRINT(path);
33
34 END_PROCEDURE;
```

Código 3.5. Algoritmo de generación de palabra aleatoria.

Para generar una palabra aleatoria del lenguaje reconocido por el autómata, se debe realizar un recorrido aleatorio al autómata. Este recorrido siempre deberá comenzar entre el nodo inicial y alguno de los nodos terminales. En el recorrido se guarda en un buffer los símbolos de los arcos visitados y cuando se finaliza la ejecución se imprime todo el buffer (que en sí es la palabra a generar). Esta implementación se explica en el código 3.5 que presenta el pseudo-código correspondiente de la función fsm2random.

Esta función utiliza las mismas estructuras utilizadas en la función fsm2language (ll. 1 - 4). Sin embargo, esta función también recibe como parámetro un número real p, $0 < p \leq 1$ (l. 5) al que denominamos probabilidad de parada (*halt-rate*). La interfaz descrita en la sección 3.4.2 provee una interfaz de entrada al usuario para que coloque este valor.

Este número real p define la probabilidad de terminar la generación de la palabra (el recorrido al autómata) al haber alcanzado un estado terminal. Por ejemplo, si $p = 1$ la generación de la palabra siempre finalizará al alcanzar el primer estado terminal. Por otro lado, si p es un número cercano a 0, será entonces pequeña la probabilidad de finalizar el algoritmo en los primeros estados terminales alcanzados. Esto motivará a que el número de símbolos que se almacena en el recorrido sea grande y por lo tanto se generará una palabra más larga. Este número permite entonces manipular el comportamiento del algoritmo para que realice recorridos cortos y largos e imprima entonces palabras compuestas por pocos o muchos símbolos.

En l. 11-12 se inicializa el recorrido, la variable s representará el estado de salida en una iteración dada (comenzando el algoritmo es el estado inicial) mientras que la variable n representa el estado de llegada en una iteración dada.

Al ejecutarse el recorrido, se toma de manera aleatoria cualquier estado alcanzable desde el estado *s* (mediante la existencia de al menos un arco de salida desde el estado *s).* Este estado aleatorio obtenido se asigna a *n*. (l. 15). Si no existiera algún estado alcanzable desde *s* se da aborta el recorrido (ll. 17 - 19).

Se inserta en el buffer (*path*) el símbolo del arco que conecta al estado de salida *s* con el estado *n* que fue seleccionado de manera aleatoria para proseguir el recorrido al autómata (l. 21).

Ahora el estado de salida *s* será el estado alcanzado *n* (l. 23). De igual manera, se pregunta si este estado seleccionado aleatoriamente es un estado terminal (l. 25). Si es un estado terminal, se generará un número aleatorio *u* entre 0 y 1 y se comparará con la probabilidad *p.* Si *p* es mayor que *u* el algoritmo proseguirá el recorrido sobre el autómata. Caso contrario, se rompe el ciclo y se imprimirá la palabra contenida en el buffer que es válida puesto que se arribó a un estado terminal.

La implementación de este algoritmo en el lenguaje C es presentada en la sección D.2 del presente trabajo.

CAPÍTULO 4

Caso de estudio con Java/PROSEGA. Análisis de la gestión de conexiones a nivel de capa MAC del estándar IEEE 802.16

4.1 Introducción al caso de estudio

El presente capitulo corresponde a la aplicación y uso del software Java/PROSEGA sobre un caso de estudio real. De esta manera, podemos presentar de manera práctica el funcionamiento del software y se pueden presentar formalmente las pruebas realizadas al software sobre un escenario práctico.

El caso de estudio escogido para realizar las pruebas de funcionamiento sobre Java/PROSEGA fue el análisis y modelado de la gestión de conexiones a nivel de capa MAC del estándar IEEE 802.16 [26]. El análisis de la gestión de conexiones a nivel de capa MAC de este estándar es un trabajo que en la actualidad está siendo desarrollado por Morales A. [22] [60] [61] y que se basa en la metodología de verificación de protocolos propuesta por Billington et. al. [4].

La función de Java/PROSEGA en dicho trabajo de investigación es servir como herramienta para las distintas tareas (conversión del Grafo de Estados a un FSM minimizado, generación del lenguaje, comparación entre FSMs, etc.) que están enmarcadas en la metodología propuesta para la verificación y análisis de protocolos.

Los siguientes apartados de esta sección ofrecen una introducción al caso de estudio. Primero, se realiza una introducción al estándar IEEE 802.16 haciendo especial énfasis en el proceso de gestión de conexiones. Luego, se describe la metodología que ha sido utilizada para llevar a cabo la tarea de análisis y verificación de este protocolo de comunicación. Después, se presenta un modelo CPN de la especificación del servicio para la gestión de conexiones y finalmente se presenta el Grafo de Estado (Grafo de Ocurrencias o *State Space*) asociado al modelo CPN mencionado.

Después de haber realizado una introducción al caso de estudio, se presenta una siguiente sección referente a las pruebas realizadas sobre el software Java/PROSEGA. En estas pruebas se utilizaron las diversas funcionalidades provistas por Java/PROSEGA para llevar a cabo las distintas tareas del proceso de análisis y verificación del estándar IEEE 802.16.

Primero, se realiza el proceso de conversión del Grafo de Estado, asociado al modelo de la especificación del servicio a nivel de capa MAC del estándar IEEE 802.16, a un FSM minimizado utilizando para ello la función de reducción provista por Java/PROSEGA. Luego, se realiza el proceso de generación del lenguaje del FSM minimizado utilizando las funciones de generación del lenguaje de Java/PROSEGA. Por último, se describen los procesos de comparación entre distintas máquinas generadas utilizando la operación de diferencia de Java/PROSEGA.

4.1.1 Estándar IEEE 802.16

El estándar IEEE 802.16 [26] [62] se encarga de especificar y describir la interfaz de los sistemas de acceso inalámbricos de banda ancha para redes de área metropolitana (WMAN, *Wireless Metropolitan Area Network*). En particular, se encarga de la descripción de las capas MAC y física (PHY).

El estándar IEEE 802.16 especifica las entidades básicas de operación en su arquitectura las cuales son

- **Estación Base (BS, *Base Station*):** Es un dispositivo que provee el servicio de control, gestión y conexión a las estaciones subscriptoras y que tiene implementado las capas MAC y física (PHY). Algunas de las funciones realizadas por este dispositivo son: asignación de ancho de banda a las estaciones subscriptoras, planificación de políticas de calidad de servicio (QoS), transmisión y recepción de datos e información de control desde y hace una o más estaciones subscriptoras, realizar control de admisión de conexiones y las funciones de gestión de conexiones, entre otras.

- **Estación Subscriptora (SS, *Subscriber Station*):** Es un dispositivo generalizado que provee conectividad entre un equipo subscriptor y una estación base, y que implementa las capas MAC y PHY. Entre algunas de las funciones que realiza este dispositivo son: identificación de la estación base, establecimiento de la conectividad básica, obtención de parámetros MAC, etc.

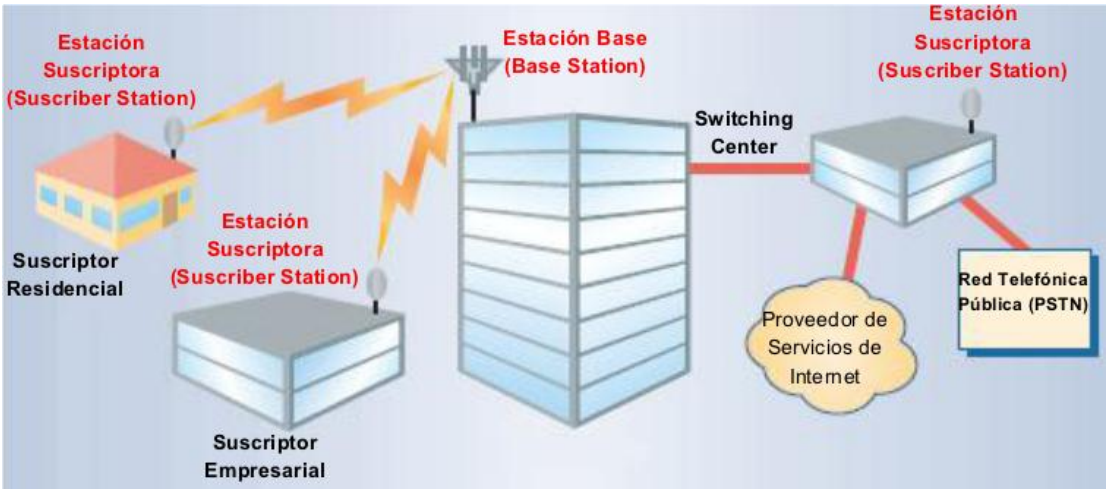

Figura 4.1. Componentes básicos de la arquitectura IEEE 802.16 [22].

Además, una revisión del estándar de 2005 [63] incluye estaciones móviles (MS, *Mobile Station*) agregando capacidad de movilidad a estaciones subscritoras.

Modelo de Referencia

El modelo estructurado por capas de IEEE 802.16 a nivel MAC se subdivide en tres subcapas, estas son: Subcapa de Convergencia (CS, *Convergence Sublayer*), Subcapa MAC de Parte Común (CPS, *Common Part Sublayer*) y Subcapa de Seguridad (*Security Sublayer*). IEEE 802.16 define un conjunto de primitivas de servicios para la comunicación entre la Subcapa de Convergencia CS y la Subcapa MAC de Parte Común CPS. Estas primitivas permiten definir la información intercambiada entre estas dos subcapas de comunicación entre pares de entidades para permitir los procesos relacionados a la gestión de conexiones (creación, cambios y terminación de conexiones) [22].

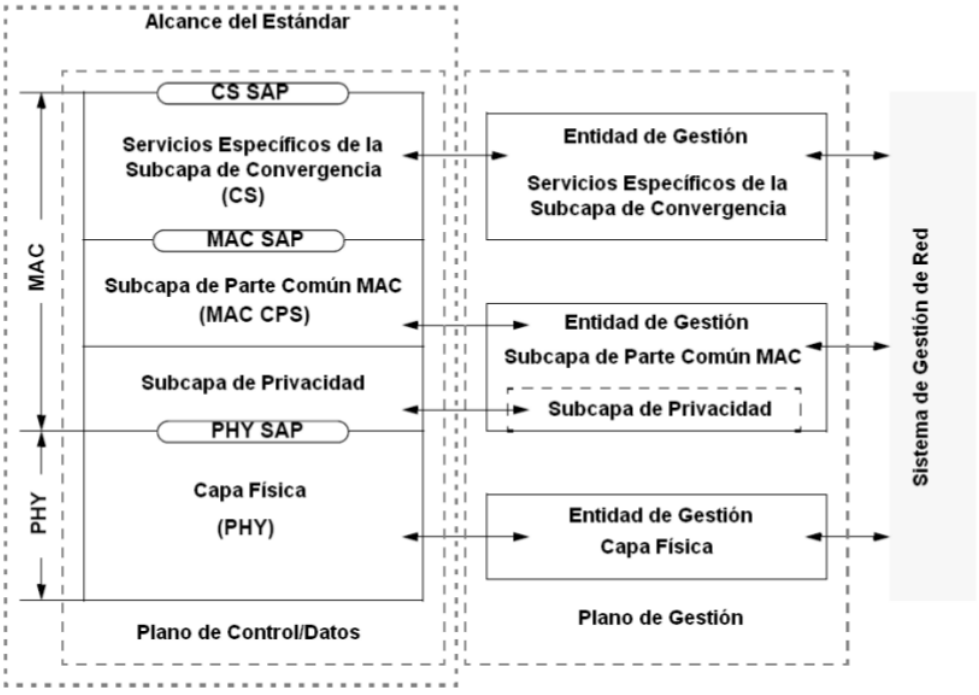

Figura 4.2. Modelo de referencia y capas de protocolos del estándar IEEE 802.16 [22].

Subcapa de Convergencia (CS)

La Subcapa de Convergencia provee la información proveniente de redes externas que se recibe a través de los puntos de acceso al servicio (SAPs, *Service Access Points*). La CS se encuentra por encima de la MAC CPS, y accede, vía el MAC SAP, a los servicios provistos por esta última. Entre las funciones de esta capa se encuentran: aceptar PDUs (*Protocol Data Units*) desde las capas superiores, clasificar las PDUs de las capas superiores (si es requerido), procesar las PDUs de capas superiores en base a su clasificación, entregar PDUs al apropiado MAC SAP y recibir PDUs desde una entidad par.

Subcapa MAC de Parte Común (MAC CPS)

Esta subcapa provee las funcionalidades básicas del sistema de acceso, asignación de ancho de banda, establecimiento (creación) de una conexión, cambios en los parámetros de una conexión (gestión de conexión) y terminación de una conexión. La MAC CPS puede recibir datos de varias CSs, a través del MAC SAP. En esta subcapa también se puede aplicar QoS para la planificación y transmisión de datos sobre la capa PHY.

Las operaciones de establecimiento, cambios y terminación de las conexiones, son descritas con mayor detalle en la Sección 4.1.2 y son modeladas a través de una Red de Petri Coloreada descrita en la Sección 4.1.4.

Subcapa de Privacidad

La subcapa de privacidad se encarga de proporcionar privacidad a los subscritores en la red inalámbrica [22] a través de servicios de autenticación e

intercambio de claves seguras y encriptación en las conexiones establecidas ente las estaciones subscriptores y la estación base.

4.1.2 Gestión de conexiones del estándar IEEE 802.16

A continuación, se describe el proceso de gestión de las conexiones a nivel de capa MAC en el estándar IEEE 802.16. La MAC CPS está orientada a conexión. Provee las funcionalidades básicas para establecer una conexión, realizar cambios en las características de las conexiones y finalizar la conexión (terminación). A continuación, se explican estos tres procesos:

- **Establecimiento de una conexión:** Luego que una estación subscriptora (SS) se registra en una estación base (BS) se procede al establecimiento de conexiones de transporte. Una conexión de transporte define las relaciones entre las subcapas de convergencias CS pares (usuarios del servicio) que utilizan la MAC. La estación BS se encarga luego de conceder el ancho de banda requerido a la estación SS. Adicionalmente, se pueden establecer nuevas conexiones en caso que se requiera cambiar las características de un servicio del cliente ya establecido.

- **Cambios en las características de la conexión:** Una vez establecida la conexión la misma puede requerir ser mantenida activamente [22]. Los requerimientos de mantenimiento varían dependiendo del tipo de servicio al que le fue provisto la conexión. En este punto pueden entonces ser realizados cambios en las características y parámetros de la conexión de acuerdo al tipo de usuario conectado y al tipo de servicio provisto.

- **Terminación de la conexión**: Finalmente, las conexiones pueden terminarse, lo cual generalmente ocurre cuando un servicio contraído por el cliente ya no es requerido y se solicita su terminación. La terminación de una conexión puede ser iniciada tanto por la estación base (BS) como por la estación subscriptora (SS) [22].

Generalmente, los protocolos de comunicación están estructurados en capas. Así dos capas pares funcionalmente iguales, pero pertenecientes a dos entidades distintas, pueden comunicarse. Para esto, se utilizan unas transacciones denominadas primitivas de servicios [23]. Estas primitivas son: *Request* (Solicitud), *Indication* (Indicación), *Response* (Respuesta), *Confirm* (Confirmación). Por su parte, el usuario del servicio (que puede ser una capa superior) accede a este conjunto de servicios a través de una dirección o identificador común conocido como el punto de acceso al servicio (SAP) (ver figura 4.3).

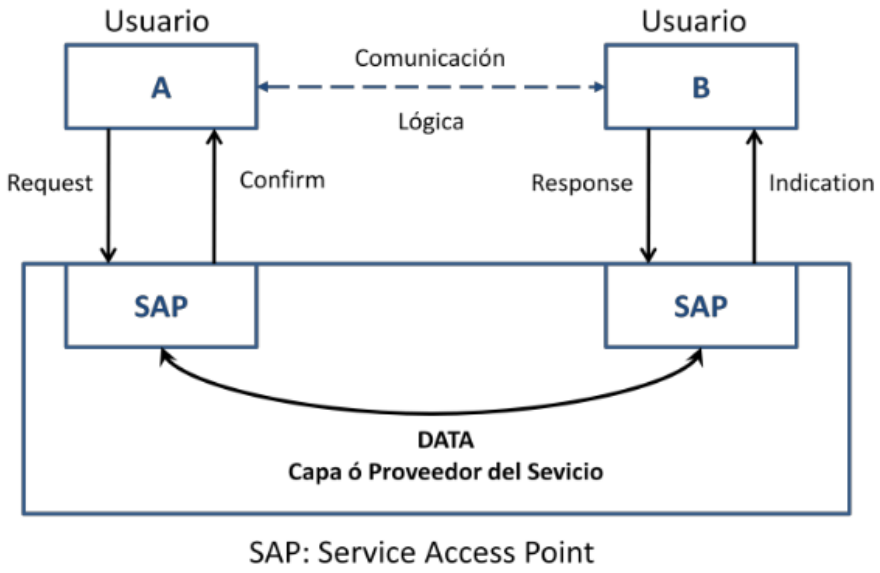

Figura 4.3. Definición de servicios entre capas [22].

Tabla 4.1. Funciones de las primitivas de servicio.

Primitiva de Servicio	Descripción
Request	El usuario del servicio solicita un servicio al proveedor.
Indication	El proveedor de servicio notifica a la entidad par receptora la solicitud de un servicio.
Response	El usuario del servicio reconoce la recepción de la primitiva de indicación del proveedor de servicio.
Confirmation	El proveedor de servicio notifica al usuario que invocó la solicitud que la actividad iniciada se ha completado.

De esta manera, se organizan las funciones (servicios) de gestión de la conexión en el estándar IEEE 802.16. Las primitivas de servicio [64] proporcionan una forma abstracta de describir la interacción entre la subcapa CS (usuario del servicio) y la subcapa MAC CPS (proveedor del servicio) de las estaciones base y subscriptoras para llevar a cabo la apertura, modificación y cierre de las conexiones.

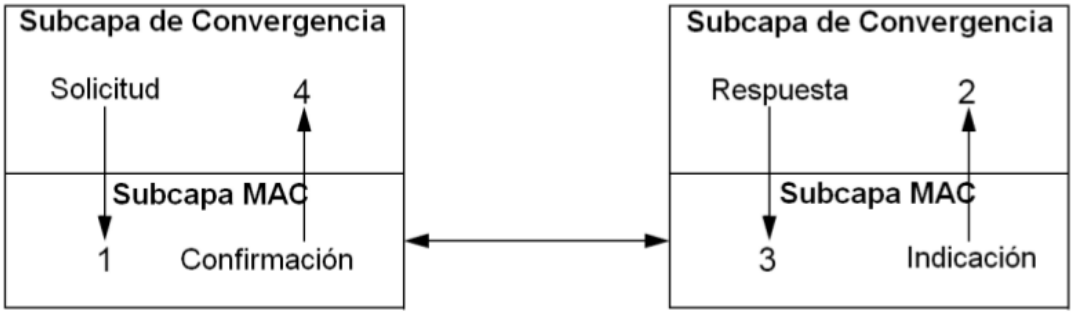

Figura 4.4. Comunicación entre entidades pares, y entre las capas CS y MAC CPS [22].

El estándar define entonces las primitivas de servicios mostradas en la tabla 4.2 para llevar así llevar a cabo la gestión de las conexiones a nivel de capa MAC.

Tabla 4.2. Primitivas de servicio de la gestión de conexiones en IEEE 802.16.

Servicio	**Primitivas de servicio**
Creación de una conexión.	MAC_CREATE_CONNECTION.Request
	MAC_CREATE_CONNECTION.Indication
	MAC_CREATE_CONNECTION.Response
	MAC_CREATE_CONNECTION.Confirmation
Cambio de las características de una conexión.	MAC_CHANGE_CONNECTION.Request
	MAC_CHANGE_CONNECTION.Indication
	MAC_CHANGE_CONNECTION.Response
	MAC_CHANGE_CONNECTION.Confirmation
Terminación de una conexión	MAC_TERMINATE_CONNECTION.Request
	MAC_TERMINATE_CONNECTION.Indication
	MAC_TERMINATE_CONNECTION.Response
	MAC_TERMINATE_CONNECTION.Confirmation

4.1.3 Metodología utilizada para la verificación del protocolo

Un protocolo de comunicación necesita satisfacer un conjunto de propiedades que están definidas para el servicio de comunicación que proporciona [22]. La verificación de que el protocolo cumpla dichas propiedades deseadas es lo que se conoce como la verificación de protocolos [4].

A pesar que el estándar IEEE 802.16 provee un conjunto de ventajas para proveer servicios de conexión inalámbrica a nivel de área metropolitana, la implementación del estándar es costosa, y se presta a múltiples ambigüedades que están presentes en su especificación a raíz del poco uso de métodos formales. Por tal motivo, Morales A.V. ha desarrollado varios trabajos de investigación [22] [60] [61] en función de realizar una verificación formal del protocolo específicamente en la gestión de conexiones a nivel de capa MAC.

Estos trabajos de investigación en torno al análisis de la gestión de conexiones a nivel de capa MAC han tenido como guía la metodología de verificación de protocolos propuesta por Billington et. al. [4]. Está metodología está basada en el uso de Redes de Petri Coloreadas [1] [32] (ver Sección 2.3) y la Teoría de autómatas (ver Sección 2.10).

A continuación, se da una breve descripción de los pasos que son realizados en esta metodología de verificación de protocolos:

- **Especificación del servicio (*Service specification*):** La especificación del servicio que se proporciona al usuario. La especificación del servicio se basa en la definición de los servicios que serán provistos al usuario (una aplicación, otro protocolo, o una capa superior del mismo protocolo). La especificación del servicio se define en un nivel más alto de abstracción que la especificación del protocolo. En este punto se desarrolla un modelo CPN basado en la especificación del servicio. Se calcula el Grafo de Estados de este modelo CPN, y se realiza un proceso de minimización del Grafo de Estados a una Máquina de Estado Finito y se genera posteriormente el lenguaje de la máquina al que se referirá como el lenguaje del servicio (L_s).

- **Especificación del protocolo (*Protocol specification*):** La especificación del protocolo incluye una descripción detallada de las características del protocolo. Dicha especificación consiste en un conjunto de reglas, de formatos y de procedimientos para que dos o más entidades se comuniquen a través de la red. Es en esta especificación donde se define la implementación de cada uno de los servicios provistos por el protocolo. En este punto se construye un modelo CPN que desarrolle la especificación del protocolo. Posteriormente, se genera el Grafo de Estado asociado al modelo CPN de la especificación del

protocolo. A partir de este modelo, se pueden analizar las propiedades de comportamiento deseadas del protocolo (ver Sección 2.4.2). También se realiza una reducción del Grafo de Estado a una Máquina de Estado y se genera posteriormente el lenguaje de esta máquina al que se referirá como el lenguaje del protocolo (L_p).

- **Comparación de la especificación del servicio con la especificación del protocolo:** Se realiza una comparación entre el lenguaje del servicio L_s y el lenguaje del protocolo L_p mediante la función de diferencia los respectivos autómatas generados para cada uno de los lenguajes. Si $L_s = L_p$ entonces el protocolo modelado es un fiel refinamiento de la especificación del servicio. Si por el contrario, $L_s \neq L_p$ entonces las secuencias de primitivas de servicio que están en el lenguaje del protocolo, pero no en el lenguaje del servicio o viceversa, necesitan ser analizadas. Estas secuencias pueden ser un resultado de, por ejemplo, un error en el modelo, un error en alguna de las especificaciones o un error de la documentación del estándar. A nivel práctico, se determina si $L_s = L_p$ si la operación de diferencia de los autómatas asociados a dichos lenguajes da vacío.

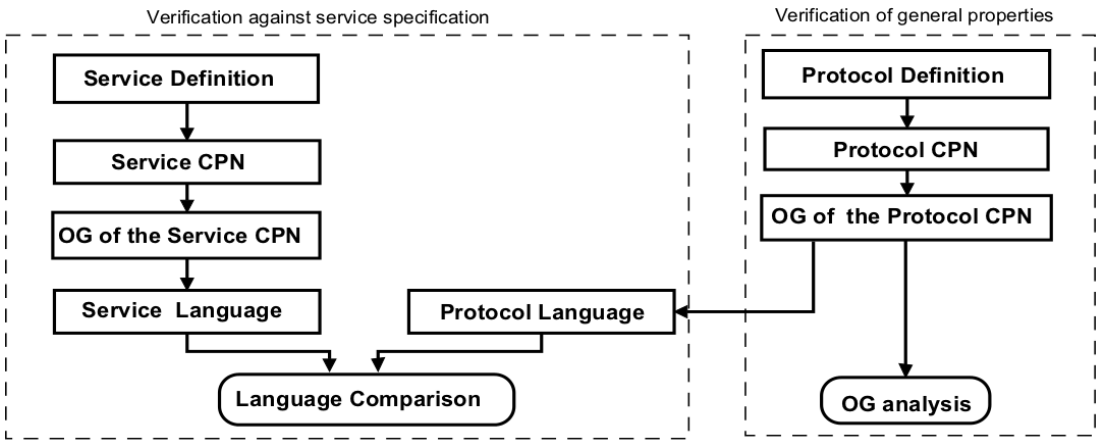

Figura 4.5. Flujo de la metodología de verificación de protocolos [4].

La primera parte de la metodología presentada correspondiente a la especificación del servicio para la gestión de conexiones a nivel de capa MAC del estándar IEEE 802.16 está contenida en el trabajo presentado en [22]. Actualmente, el estado actual (*state-of-art*) del trabajo de verificación de la gestión de conexiones a nivel de capa MAC del estándar IEEE 802.16 llevado a cabo por Morales A. V. se encuentra en la fase de desarrollo de la especificación del protocolo.

4.1.4 Modelo CPN de la especificación del servicio

A continuación, se presenta el modelo CPN de la especificación del servicio desarrollado por Morales A. V. [22]. Esta especificación del servicio abarca los servicios de creación, cambios y terminación de las conexiones invocados por las subcapas de convergencia (usuarios del servicio) y que son provistos por las subcapas MAC CPS (proveedor del servicio), estas subcapas están contenidas a su vez entre las dos entidades que se comunican (estación base y estación subscriptora). Estos servicios de creación, cambios y terminación son desglosados a su vez en un conjunto de primitivas de servicios (ver tabla 4.2).

El modelo CPN de especificación del servicio desarrollado por Morales A. V. [22] fue construido de manera jerárquica mediante el manejo de páginas y transiciones de substitución. De igual manera, cada transición del modelo CPN construido representa una primitiva de servicio (salvo un par de excepciones). Las inscripciones y demás declaraciones en lenguaje CPN ML del modelo CPN corresponden a las variables y demás datos que intercambian las estaciones base y subscriptora. En [22] se encuentra una descripción detallada de estas declaraciones, así como todas las páginas del modelo CPN.

La figura 4.6 ilustra la página *Top*, el módulo principal del modelo. En este módulo, una entidad solicitante (modelada mediante una plaza) puede generar la

ocurrencia de alguna de las siguientes transiciones: *CreatConnection* (establecimiento de la conexión), *ChangeConnection* (cambios en las características de la conexión) y *TerminateConnection* (terminación de la conexión). Estas transiciones del módulo principal, por ser transiciones de substitución, representan las páginas del mismo nombre, que a su vez contienen las primitivas de servicio correspondientes para generar una secuencia de ocurrencias de primitivas de servicio.

En el estado del modelo (figura 4.6 [22]) se muestran las pestañas que contienen las demás páginas del modelo CPN. En el estado en que se presenta el modelo (estado inicial) se muestra habilitada la transición de substitución *CreateConnection* indicando que es posible la ocurrencia de dicha transición que podría desencadenar la secuencia de ocurrencias de primitivas de servicio necesarias para la creación de la conexión.

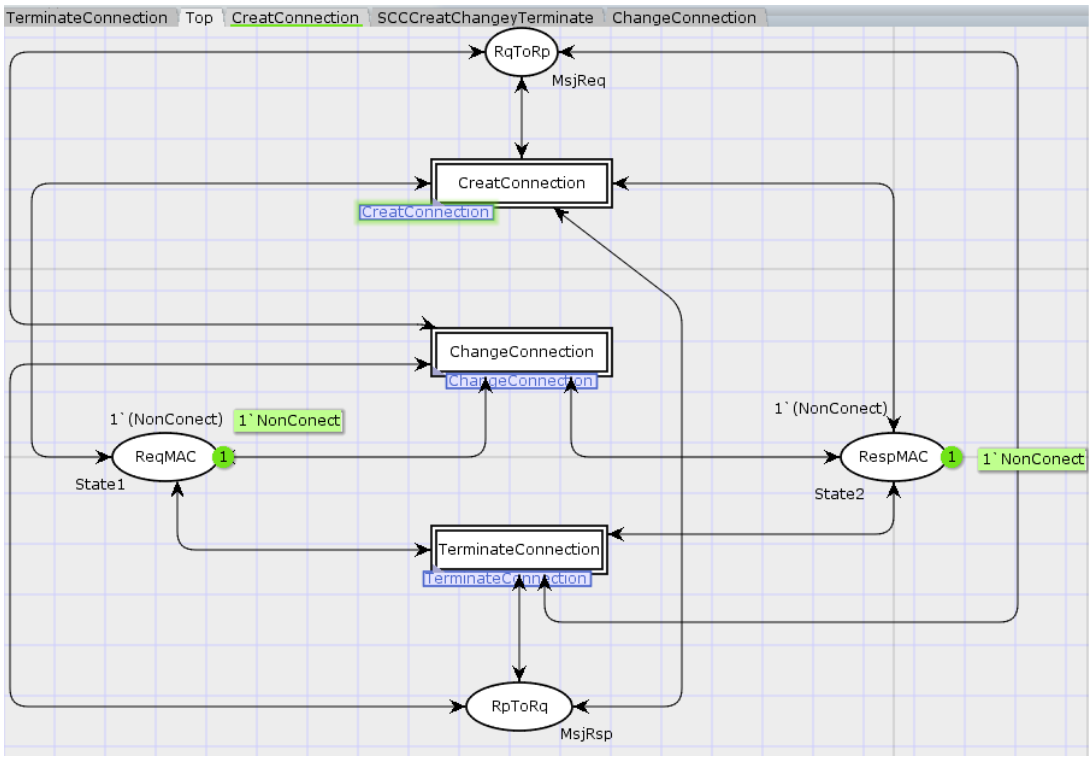

Figura 4.6. Módulo principal del modelo CPN de la especificación del servicio [22].

4.1.5 Grafo de Estado asociado al modelo CPN

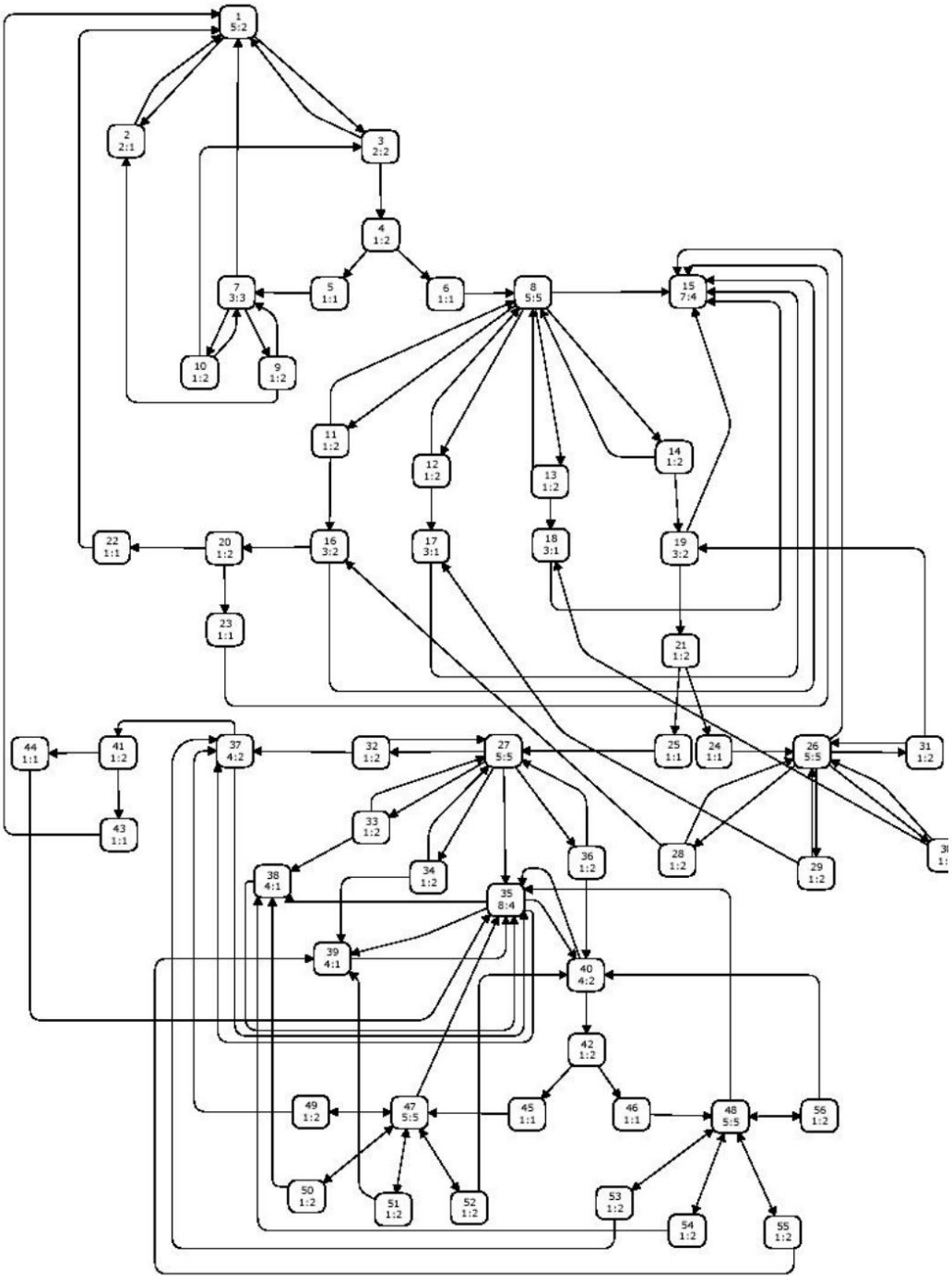

Figura 4.7. Grafo de Estado del modelo CPN de la especificación del servicio [22].

Esta sección presenta en su introducción la figura 4.7. Esta figura ilustra el Grafo de Estado (*State Space*) (desarrollado por Morales A.V. [22]) asociado al modelo CPN de la especificación del servicio de la gestión de conexiones a nivel de capa MAC del estándar IEEE 802.16. Este Grafo de Estado fue generado mediante la herramienta para el cálculo de Grafos de Estado (*State Space Tool*) provista por CPN Tools [5].

Cada nodo del grafo representa un marcado o estado del modelo mientras que los arcos del grafo representan las ocurrencias de las transiciones del modelo CPN. Dado que las transiciones del modelo CPN de la especificación del servicio representan las primitivas de servicio podríamos decir que los arcos del grafo representan las ocurrencias de las primitivas de servicio.

El Grafo de Estado sirve para hacer un chequeo general de las propiedades de comportamiento de la especificación del servicio (véase Sección 2.4.2). Dentro del proceso metodológico utilizado [4] para la verificación de protocolos este Grafo de Estado es utilizado para generar el correspondiente autómata (o Máquina de Estado Finito) minimizado para así generar posteriormente el lenguaje del servicio (L_s).

4.2 Pruebas con Java/PROSEGA

Esta sección del presente trabajo describe las pruebas realizadas con el software Java/PROSEGA utilizando como caso de estudio la verificación del proceso de gestión de conexiones a nivel de capa MAC del estándar IEEE 802.16 [26].

La sección 4.1 del presente trabajo introdujo la primera parte de la metodología empleada [4] para la verificación del protocolo que es la especificación del servicio. Esta primera parte fue desarrollada en [22].

En esta sección referente a las pruebas de Java/PROSEGA, utilizando el protocolo mencionado como caso de estudio, describe cómo se utiliza el software Java/PROSEGA para llevar a cabo las distintas funciones y tareas que son necesarias dentro de la metodología de verificación de protocolos utilizada.

El primer apartado de esta sección describe las pruebas realizadas para la generación del FSM minimizado a partir del Grafo de Estado de la especificación del servicio utilizando el reductor de Java/PROSEGA (invocado a través del instrumento RUN). El segundo apartado de esta sección se refiere a las pruebas realizadas para generar el lenguaje del FSM minimizado resultante a través del uso del generador de lenguajes de Java/PROSEGA (invocado a través del instrumento LANG). El tercer y último apartado de esta sección describe las pruebas realizadas a la función de diferencia del lenguaje de Java/PROSEGA (invocando a través del instrumento DIFF).

4.2.1 Reducción del Grafo de Estado de la especificación del servicio a una Máquina de Estado Finito

La Sección 4.1.5 muestra el Grafo de Estado generado a partir del modelo CPN de la especificación del servicio. A continuación, se utiliza el reductor de Grafos de Estado de Java/PROSEGA (véase Sección 3.3) para generar una Máquina de Estado Finito minimizada que permita posteriormente generar el lenguaje del servicio del protocolo en estudio.

Para llevar a cabo esta tarea se carga el modelo en CPN Tools [5] de manera que arranque el servidor de extensiones (véase Sección 2.8.3) y con él, la extensión Java/PROSEGA. Luego, se debe calcular el Grafo de Estados del modelo para que pueda ser ejecutado el proceso de reducción a través del instrumento RUN.

La figura 4.8 muestra el ambiente en una máquina previo a la utilización del reductor de Grafos de Estado de Java/PROSEGA. En la figura se observa el servidor de extensiones de CPN Tools encendido. De esta manera, se despliega Java/PROSEGA en el panel izquierdo del editor gráfico de CPN Tools. También, se carga en CPN Tools el modelo CPN de la especificación del servicio, se calcula el Grafo de Estados mediante la paleta *SS* y se ejecuta luego el instrumento *RUN* para iniciar el reductor de Grafos de Estado de Java/PROSEGA.

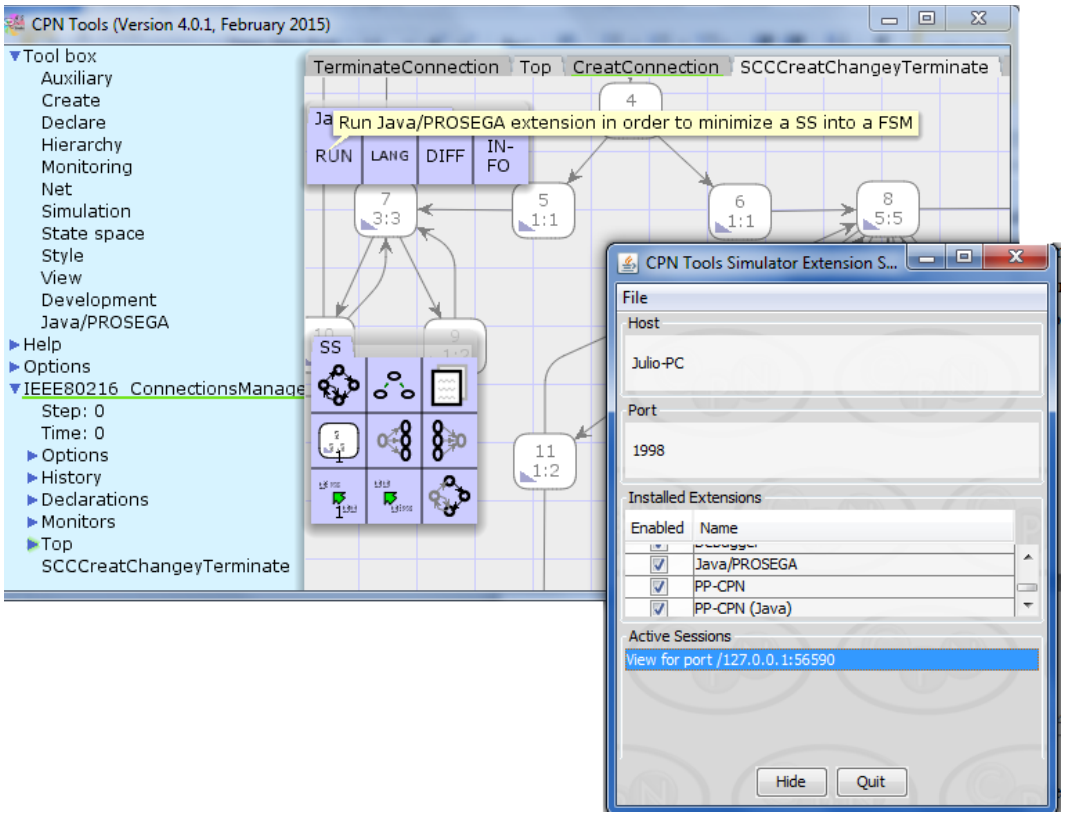

Figura 4.8. Ambiente para las pruebas con Java/PROSEGA.

Al invocarse el reductor de Java/PROSEGA, se inicia el proceso de obtención del Grafo de Estado, sus arcos y marcados muertos mediante una comunicación con el simulador de CPN Tools (Este proceso es explicado en detalle en la Sección 3.3.2)

En la interfaz de asignación de identificadores (véase Sección 3.3.3) se asigna a cada transición del modelo CPN un identificador numérico (salvo a dos transiciones que no modelan una primitiva de servicio). Esta asociación de transiciones e identificadores se realiza mediante la opción de importación de un archivo de arcos que sigue el formato de *"Archivos para la asignación de identificadores a los arcos del Grafo de Estado"* descrito en la Sección 4.2. Este archivo es mostrado en la Sección F.1 del Anexo "F" del presente trabajo. En esta asignación se siguió con la asociación planteada en [22, Ch. 7, Sec. 7.5.1, pp. 107-109].

Por otra parte, los nodos terminales del Grafo de son definidos en [22] (para este caso particular) como los nodos en los cuales se finaliza una secuencia de ocurrencia de primitivas de servicio. Esto será, cada nodo del Grafo que sea antecedido por un arco que represente una ocurrencia de una primitiva de servicio (transición) del tipo *confirmation*. La tabla 4.3 presenta una lista de los nodos terminales que cumplen con esta característica.

Tabla 4.3. Nodos terminales para el Grafo de Estado.

nodos terminales
1
7
8
13
26
27
31
47
48

En cuanto a las opciones, es indistinto seleccionar la opción "incluir marcados muertos con los estados terminales" puesto que el Grafo de Estado utilizado no posee marcados muertos. En cuanto a la opción del tipo de identificador para los arcos, se

marca la opción *int* pues se usan identificadores numéricos. La figura 4.9 muestra la interfaz previa al proceso de reducción del Grafo a un FSM minimizado. La figura 4.10 muestra la interfaz de resultado luego de ejecutarse la reducción del grafo.

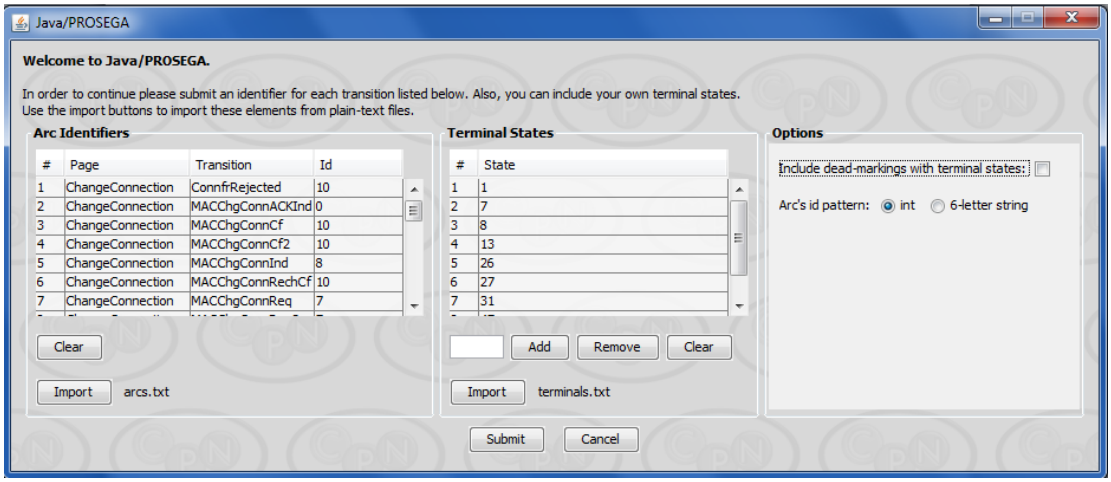

Figura 4.9. Interfaz de asignación de identificadores con las configuraciones de prueba.

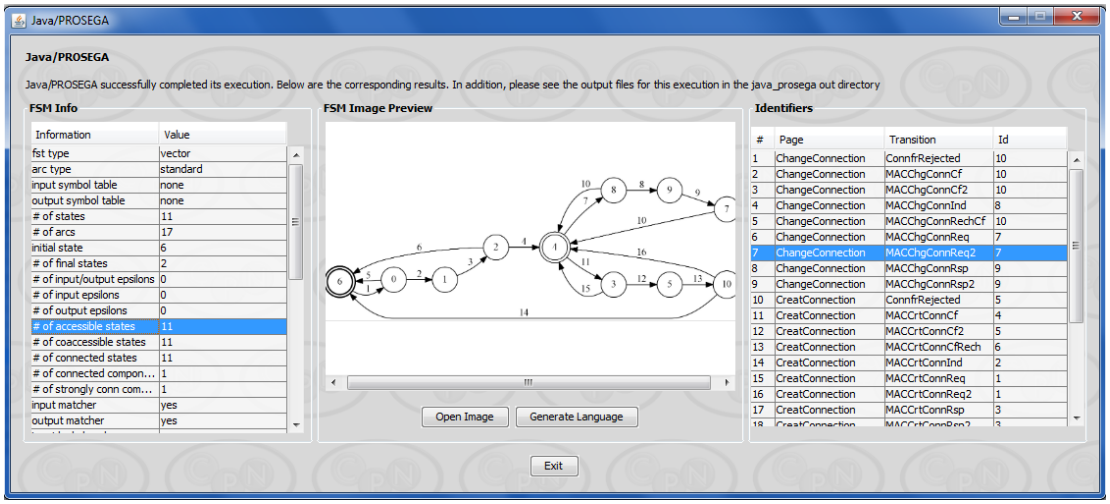

Figura 4.10. Interfaz de resultado de la minimización del FSM.

En esta interfaz de resultado (explicada en detalle en la Sección 3.3.6) se provee la información del autómata minimizado (sección izquierda), una vista previa de la imagen de la Máquina de Estado Finito generada (sección central) y la sección

de identificadores (sección izquierda). A continuación, la figura 4.11 presenta la Máquina de Estado Finito generada.

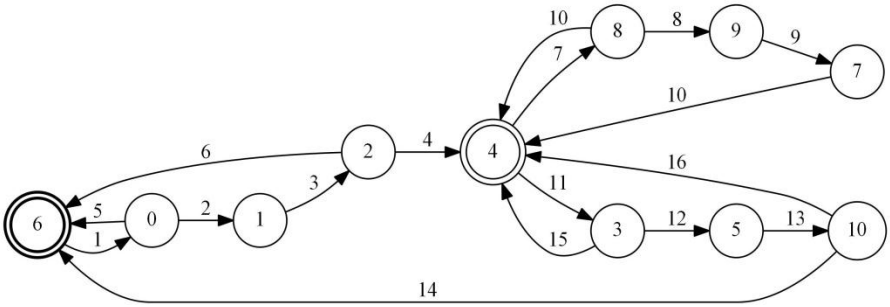

Figura 4.11 FSM minimizado obtenido mediante Java/PROSEGA.

La prueba del proceso de reducción del Grafo de Estado a una Máquina de Estado Finito, utilizando como base el Grafo de Estado asociado al modelo CPN de especificación del servicio de la gestión de conexiones a nivel de capa MAC del estándar IEEE 802.16, permitió determinar la efectividad de Java/PROSEGA de realizar el proceso de minimización ajustándose al resultado esperado comparándolo con el trabajo contenido en [22] que utilizó una herramienta ya comprobada y trabajada como lo es AT&T FSM Library [27].

4.2.2 Generación del lenguaje de la Máquina de Estado Finito

En esta prueba se lleva a cabo la generación del lenguaje del FSM minimizado obtenido en la Sección 4.2.1 utilizando para ello la función de generación del lenguaje de Java/PROSEGA. Este lenguaje generado corresponde al lenguaje del servicio. El lenguaje obtenido en esta sección es comparado contra el lenguaje del servicio obtenido en [22] a fines de comprobar la correctitud en el resultado provisto por Java/PROSEGA.

El autómata minimizado contenido en el archivo *min.fst* es pasado como parámetro a la interfaz de entrada de Java/PROSEGA para la generación del lenguaje

(ver figura 4.12). Ambos archivos se encuentran contenidos en la carpeta generada, en el directorio *out*, correspondiente a la ejecución del proceso de minimización en la Sección 4.2.1. De igual manera, la interfaz de reducción provee un acceso directo a esta función de generación del lenguaje en la interfaz de resultado a través del botón *Generate Language*.

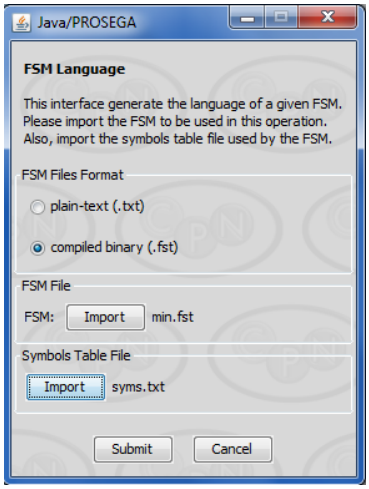

Figura 4.12. Interfaz de generación del lenguaje con los parámetros de prueba.

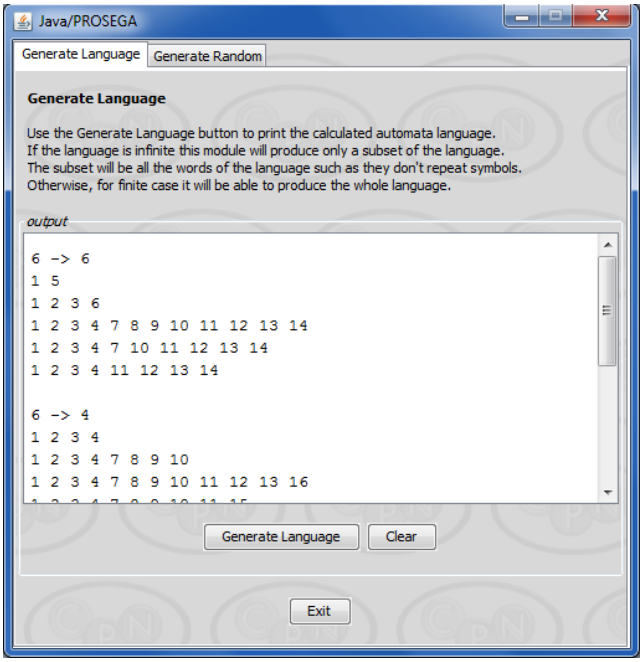

Figura 4.13. Interfaz de resultado de generación del lenguaje con el FSM de prueba.

La salida (el subconjunto del lenguaje) generada en esta función se coloca a continuación:

Tabla 4.4. Cadenas generadas pertenecientes al lenguaje del servicio.

6 → 6	1 5
	1 2 3 6
	1 2 3 4 7 8 9 10 11 12 13 14
	1 2 3 4 7 10 11 12 13 14
	1 2 3 4 11 12 13 14
6 → 4	1 2 3 4
	1 2 3 4 7 8 9 10
	1 2 3 4 7 8 9 10 11 12 13 16
	1 2 3 4 7 8 9 10 11 15
	1 2 3 4 7 10
	1 2 3 4 7 10 11 12 13 16
	1 2 3 4 7 10 11 15
	1 2 3 4 11 12 13 16
	1 2 3 4 11 12 13 16 7 8 9 10
	1 2 3 4 11 12 13 16 7 10
	1 2 3 4 11 15
	1 2 3 4 11 15 7 8 9 10
	1 2 3 4 11 15 7 10

Esta salida corresponde al subconjunto de palabras aceptadas por el FSM minimizado. En este caso de estudio en particular, estas palabras corresponden a una secuencia de ocurrencia de primitivas de servicio de la gestión de conexiones a nivel de capa MAC del estándar IEEE 802.16 (creación de conexión, modificación de parámetros de la conexión, cierre de la conexión).

La notación $x \rightarrow y$ indica el subconjunto de palabras generadas cuyo procesamiento comienza en el estado x del autómata y finaliza en el estado y. El estado x siempre corresponderá al nodo inicial del autómata (estado inicial) mientras que el estado y corresponderá a cada uno de los estados terminales. Para el FSM minimizado utilizado solo existen dos nodos terminales (véase figura 4.11).

Los resultados arrojados por Java/PROSEGA coinciden con el subconjunto del lenguaje obtenido por Morales A. V. en [22].

Adicional a esto, Java/PROSEGA permite generar cadenas aleatorias pertenecientes al lenguaje del servicio del FSM minimizado a través de la interfaz de generación de cadenas aleatorias (véase Sección 3.4.2). La figura 4.14 muestra cómo puede utilizarse esta interfaz para generar palabras (secuencias de primitivas de servicio) de manera aleatoria y perteneciente al lenguaje obtenido.

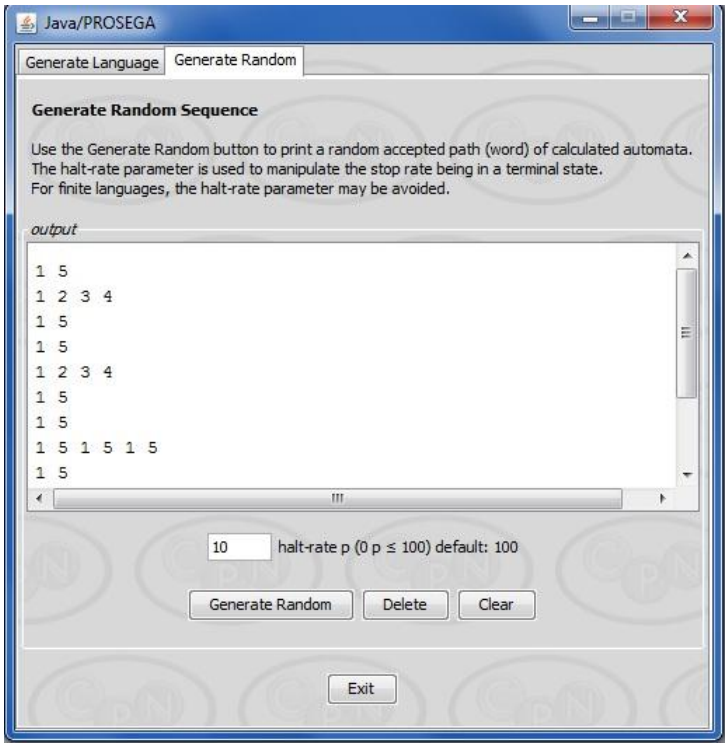

Figura 4.14. Obtención de cadenas aleatorias del lenguaje del servicio.

A continuación, se presentan un conjunto de cadenas aleatorias del lenguaje del servicio obtenidas mediante la interfaz de generación de cadenas aleatorias de Java/PROSEGA (tabla 5.5). La forma como estas cadenas son generadas se explica en la Sección 3.6.3 del presente trabajo correspondiente a la función *fsm2random*. La muestra que se presenta fue generada utilizando diversos valores de *p* (probabilidad de parada) de manera de generar cadenas con longitud variada (lo que sería igual decir secuencias con alto o bajo número de ocurrencias de primitivas de servicio).

Tabla 4.5. Generación de cadenas aleatorias pertenecientes al lenguaje del servicio.

p = 100%	1 5
	1 2 3 6
	1 2 3 6
	1 2 3 4
	1 5
p = 75%	1 5
	1 5
	1 2 3 4
	1 2 3 4
	1 2 3 4
p = 50%	1 2 3 4 11 15
	1 2 3 4 11 15
	1 5
	1 2 3 6 1 2 3 6
	1 2 3 6 1 2 3 6
p = 25%	1 2 3 4 7 8 9 10
	1 2 3 6 1 5 1 5 1 5 1 2 3 6
	1 2 3 6 1 5 1 5 1 5 1 2 3 6
	1 5
	1 5
p = 10%	1 2 3 6 1 2 3 4 7 8 9 10 11 15 11 12 13 14
	1 2 3 6 1 2 3 4 7 8 9 10 11 15 11 12 13 14
	1 5 1 2 3 6 1 5 1 2 3 4 7 10
	1 2 3 4 11 15
	1 5 1 5 1 2 3 6 1 5 1 5 1 2 3 4 11 12 13 16 7 10

Si se toma cualquiera de las palabras (secuencias de ocurrencias de primitivas de servicio) aleatorias obtenidas y se toma la Máquina de Estado Finito minimizada obtenida (ver figura 4.11) se dará cuenta que podrá ser posible realizar un recorrido valido al autómata.

De igual manera, se puede apreciar en la tabla 4.5 la probabilidad de para *p* (halt-rate) sirve al usuario como un parámetro para estimar la longitud de la cadena a generar. A medida que p $\rightarrow$ 0, el algoritmo de generación para una cadena necesitará atravesar un número considerable de estados terminales antes de finalizar su ejecución.

4.2.3 Pruebas de comparación de autómatas

En esta sección se presentan las pruebas realizadas a la operación de diferencia de Java/PROSEGA. Su utilidad dentro del caso de estudio planteado en esta sección (verificación de la gestión de conexiones a nivel de capa MAC del estándar IEEE 802.16) es la de proveer una herramienta para llevar a cabo la operación de comparación de lenguajes.

Utilizando la metodología de verificación planteada en este caso de estudio (véase Sección 4.1.3) se debe utilizar la herramienta de diferencia de Java/PROSEGA para llevar a cabo la comparación entre el lenguaje del servicio (L_s) y el lenguaje del protocolo (L_p).

El lenguaje del servicio fue obtenido a través del FSM minimizado en [22] que fue calculado nuevamente mediante Java/PROSEGA en la Sección 4.1.2. Sin embargo, el estado actual (*state-of-art*) de este trabajo de verificación no ha concluido con el desarrollo del modelo y el respectivo lenguaje del protocolo. Por tal motivo, no

se puede realizar una prueba formal de comparación entre el lenguaje del servicio y el lenguaje del protocolo.

Sin embargo, se ha utilizado la herramienta de diferencia de Java/PROSEGA para realizar comparaciones entre autómatas asociados a módulos específicos del modelo CPN del protocolo que se encuentra en fase de desarrollo por Morales A. V. contra la especificación del protocolo descrita en el estándar IEEE 802.16 [62].

Una prueba particular que se desarrolló fue la de comparar el autómata que representa el módulo de gestión de flujo de servicios a nivel de la especificación del protocolo contra, un autómata generado a partir de un modelo CPN desarrollado por Morales A. V. que busca modelar dicha especificación del protocolo. Esta comparación se realiza fines de determinar incongruencias a nivel de la especificación del protocolo en el estándar y a fines también de probar la funcionalidad de la operación de diferencia de Java/PROSEGA.

A continuación, la figura 4.15 presenta el autómata que se encuentra en el estándar que representa el módulo de gestión de flujo de servicios a nivel de la especificación del protocolo [62].

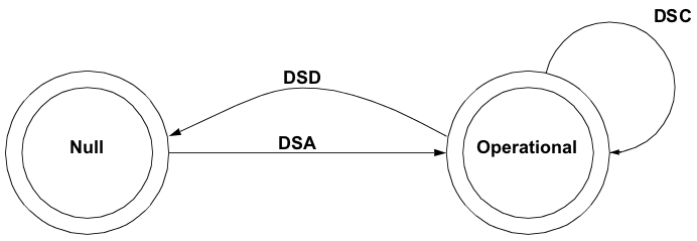

Figura 4.15. FSM asociado a la gestión de flujos de servicio a nivel de la especificación del protocolo [62].

Para realizar la comparación de este FSM del estándar contra el FSM generado a partir del modelo en desarrollo por Morales A.V se construye dicho FSM de la especificación del estándar mediante la herramienta OpenFST [28] (figura 4.16).

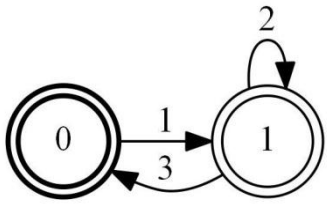

Figura 4.16. FSM construido con OpenFST del modelo especificado en [62].

Para esto se realizó también una asociación de los símbolos contenidos en el autómata con identificadores numéricos (tabla 4.6).

Tabla 4.6. Asignación de identificadores para el autómata de la especificación del protocolo que modela la gestión de flujos de servicio.

símbolo del arco	**identificador numérico**
DSA	1
DSC	2
DSD	3

Se presenta el FSM desarrollado por Morales A.V. a concerniente a este módulo en particular de la especificación del protocolo (Figura 4.17).

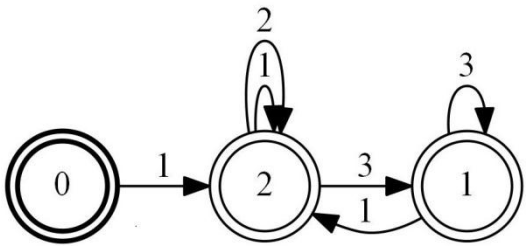

Figura 4.17. FSM obtenido por Morales A.V. a través de Java/PROSEGA.

Para efectos de esta prueba el modelo de la figura 4.16 extraído tal cual del estándar IEEE 802.16 lo denominaremos el autómata "A" mientras que el autómata de la figura 4.17 construido por Morales A.V a través de Java/PROSEGA mediante la reducción del Grafo de Estado del modelo CPN asociado a la especificación del protocolo lo denominaremos autómata "B".

Generación del lenguaje del autómata A

El lenguaje de la máquina A es infinito (la máquina contiene ciclos). Fue utilizado Java/PROSEGA para generar un subconjunto del lenguaje A. Este corresponde a todas a las cadenas del lenguaje de A tal que dichas cadenas fueron producidas sin repetir arcos en el recorrido que se le realizó al autómata para generar dichas cadenas. A continuación, un subconjunto del lenguaje de A.

```
0 -> 0
1 2 3
1 3

0 -> 1
1
1 2
```

Figura 4.18. Subconjunto del lenguaje reconocido por el autómata "A".

Debajo de 0 → 0 se indican todas las cadenas que fueron producidas realizando un recorrido desde el estado inicial 0 hasta el mismo estado 0 (pues también es un estado terminal o de parada). Debajo de 0 → 1 se indican todas las cadenas que fueron producidas realizando un recorrido desde el estado 0 hasta el estado 1. Los símbolos de estas cadenas hacen referencia a los símbolos de los arcos que fueron transitados para poder generar la cadena.

Generación del lenguaje del autómata B

Al igual que el autómata A, el lenguaje de B es infinito. Por lo que se utilizó la misma función de Java/PROSEGA para generar un sub conjunto de dicho lenguaje que fue utilizada para el autómata B. El resultado es el siguiente:

```
0 -> 0

0 -> 1
1 2 3
1 2 3 3
1 3
1 3 3

0 -> 2
1
1 2
1 2 3 1
1 2 3 3 1
1 3 1
1 3 1 2
1 3 3 1
1 3 3 1 2
```

Figura 4.19. Subconjunto del lenguaje reconocido por el autómata "B".

Aplicando diferencia A – B

Ahora es utilizada la función de Java/PROSEGA para calcular la diferencia entre estos dos autómatas. Para esto, Java/PROSEGA se basa en la función *fstdifference* de OpenFST que genera una máquina que es capaz de generar todas las cadenas que acepta el autómata A pero que no sean aceptadas por el autómata B. El lenguaje L de esta máquina resultante A – B se define de la siguiente manera.

$L(A - B) = \{ p \mid p \in L(A) \wedge p \notin L(B) \}$

Sin embargo, el resultado de aplicar A – B es vacío. Si se revisa bien, B es capaz de generar todas las cadenas (secuencias de símbolos) que puede generar el autómata A. Por lo tanto, no se genera ninguna máquina de estado puesto que no existe diferencia alguna. Como ejercicio se puede observar que en el subconjunto generado para el lenguaje de B están presentes todas las cadenas del subconjunto generado del lenguaje de A.

Aplicando diferencia B – A

Ahora se realizará el proceso inverso. Se requiere calcular una máquina que sea la diferencia B – A. Es decir, se generará una máquina de estado B – A que acepta todas cadenas que están en B y que no sean aceptadas por A. En este caso, el resultado no es vacío. Esto quiere decir que existen cadenas que son aceptadas por B y que no se encuentran en el lenguaje del autómata A. A continuación, la máquina resultante B – A (calculada por Java/PROSEGA).

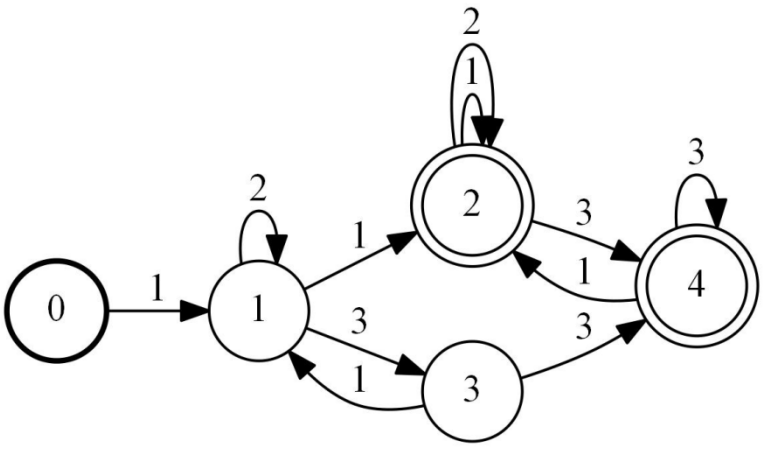

Figura 4.20. Autómata B - A.

Generación del lenguaje de B – A

Esta máquina resultante es capaz de procesar las palabras que están en el autómata B y que no se encuentran en A. Para generar este subconjunto del lenguaje del autómata B - A (diferencia de B y A) se utilizó la función de generación del lenguaje de Java/PROSEGA.

Tabla 4.7. Lenguaje de la máquina B - A.

0 -> 2		**0 -> 4**	
		1 3 3	1 2 3 3 1 3
1 3 3 3 1	1 1 2	1 3 3 3	1 2 3 3 1 3 3
1 3 3 3 1 2	1 1 2 3 3 1	1 3 3 3 1 3	1 2 3 3 1 2 3
1 3 3 1	1 1 2 3 1	1 3 3 3 1 2 3	1 2 3 3 1 2 3 3
1 3 3 1 2	1 2 3 3 3 1	1 3 3 1 3	1 2 3 1 1 3
1 3 1 1	1 2 3 3 3 1 2	1 3 3 1 3 3	1 2 3 1 1 3 3
1 3 1 1 3 3 1	1 2 3 3 1	1 3 3 1 2 3	1 2 3 1 1 2 3
1 3 1 1 3 3 1 2	1 2 3 3 1 2	1 3 3 1 2 3 3	1 2 3 1 1 2 3 3
1 3 1 1 3 1	1 2 3 1 1	1 3 1 1 3	1 2 1 3
1 3 1 1 3 1 2	1 2 3 1 1 3 3 1	1 3 1 1 3 3	1 2 1 3 3
1 3 1 1 2	1 2 3 1 1 3 3 1 2	1 3 1 1 2 3	1 2 1 2 3
1 3 1 1 2 3 3 1	1 2 3 1 1 3 1	1 3 1 1 2 3 3	1 2 1 2 3 3
1 3 1 1 2 3 1	1 2 3 1 1 3 1 2	1 3 1 2 1 3	
1 3 1 2 1	1 2 3 1 1 2	1 3 1 2 1 3 3	
1 3 1 2 1 3 3 1	1 2 3 1 1 2 3 3 1	1 3 1 2 1 2 3	
1 3 1 2 1 3 3 1 2	1 2 3 1 1 2 3 1	1 3 1 2 1 2 3 3	
1 3 1 2 1 3 1	1 2 1	1 1 3	
1 3 1 2 1 3 1 2	1 2 1 3 3 1	1 1 3 3	
1 3 1 2 1 2	1 2 1 3 3 1 2	1 1 2 3	
1 3 1 2 1 2 3 3 1	1 2 1 3 1	1 1 2 3 3	
1 3 1 2 1 2 3 1	1 2 1 3 1 2	1 2 3 3	
1 1	1 2 1 2	1 2 3 3 3	
1 1 3 3 1	1 2 1 2 3 3 1	1 2 3 3 3 1 3	
1 1 3 3 1 2	1 2 1 2 3 1	1 2 3 3 3 1 2 3	
1 1 3 1			
1 1 3 1 2			

Lo anterior colocado corresponde a un subconjunto de todas las palabras aceptadas por el autómata B y que no pueden ser aceptadas o reconocidas por el autómata A. Se puede comprobar realizando lo siguiente, elija cualquier palabra generada de este lenguaje generado del autómata B – A e intente procesarlo en el autómata A y verá como no se puede realizar dicho proceso.

CAPÍTULO

5

Conclusiones

A continuación, se presentan las conclusiones del presente trabajo en función de los objetivos que se plantearon.

Mediante el presente trabajo logró desarrollar una solución denominada Java/PROSEGA embebida dentro del software CPN Tools [5] para la conversión de Grafos de Estado a Máquinas de Estado Finito y su posterior reducción mediante el algoritmo de reducción de Barrett et. al. [3] y que utiliza Billington et. al. [4] en su proceso de verificación de protocolos. Para llevar a cabo el desarrollo de este software integrado a CPN Tools se logró utilizar adecuadamente la herramienta complementaria Access/CPN [9] y el manejo de extensiones de CPN Tools [8].

A través del manejo de extensiones de CPN Tools se logró integrar el software Java/PROSEGA a la interfaz gráfica de CPN Tools. Access/CPN fue útil para la utilización del protocolo BIS que permitió obtener el Grafo de Estado previamente calculado en CPN Tools y manipularlo a fines de realizar la conversión de este a un autómata finito y aplicar sobre este último el algoritmo de reducción citado previamente. Uno de los problemas encontrados en la utilización de estas herramientas fue la escasa documentación que estas herramientas poseían. Por tal motivo, este trabajo se encargó de realizar una investigación sobre el correcto uso de dichas herramientas. El capítulo 2 del presente trabajo presenta dicho trabajo sobre Access/CPN y el manejo de extensiones.

Adicionalmente, se utilizaron las librerías OpenFST [28] y Graphviz [54] para la manipulación y graficación de los autómatas. En particular, OpenFST provee un conjunto de comandos que Java/PROSEGA invoca para llevar a cabo las distintas funciones relacionadas con el manejo de los autómatas.

Como apoyo al proceso metodológico propuesto por Billington et. al. [4] para la validación de sistemas, en particular de protocolos de comunicación, se logró desarrollar dentro de Java/PROSEGA las funciones de generación del lenguaje y de comparación de autómatas. Para la generación del lenguaje el autor del presente trabajo desarrolló una librería en lenguaje C denominada *fsm2language*. Una completa descripción sobre cómo fue implementada esta librería y los comandos que provee de manera de generar el lenguaje de los autómatas se encuentra en el capítulo 2 del presente trabajo. Por otra parte, para la comparación entre autómatas se integró exitosamente la función *fstdiferrence* de OpenFST dentro de Java/PROSEGA para llevar a cabo una comparación entre el lenguaje de autómatas.

Como caso de prueba, el software Java/PROSEGA ha podido ser utilizado como una herramienta de apoyo dentro del proceso de análisis y verificación del estándar IEEE 802.16 específicamente en el proceso de gestión de conexiones a nivel de capa MAC [22] [60] [61].

5.1 Aportes

A continuación, se describen algunos de los aportes obtenidos a partir del desarrollo del software Java/PROSEGA

- **Estudio y documentación acerca de las herramientas adicionales para CPN Tools:** El desarrollo del software Java/PROSEGA tuvo como antesala un estudio exhaustivo de todas las herramientas adicionales

existentes para CPN Tools. De las herramientas más interesantes y útiles para el desarrollo del presente trabajo resaltaron Access/CPN [9] y las Extensiones de CPN Tools [8]. Mientras que Access/CPN permite la integración de programas con el potente simulador de CPN Tools, las extensiones de CPN Tools permite que cualquier software desarrollado sea embebido dentro del mismo CPN Tools. Estas dos herramientas quedan fuertemente documentadas en este trabajo para futuros usos.

- **Selección de nuevas herramientas de software de apoyo al proceso de verificación y análisis, propuesto por Billington et. al. [4]:** Los proyectos anteriores (como [10] [22] [39]) basados en esta metodología de análisis y verificación utilizan software que ya no está disponible libremente y que no está disponible en sistemas Windows (como es el caso de AT&T FSM Library [27] y Lextools [57]). Previo al desarrollo de Java/PROSEGA, se tuvo que realizar una investigación sobre nuevas herramientas *open-source* y disponibles para Windows que permitieran llevar a cabo las tareas dentro de la metodología de verificación utilizada. Es así, como fue escogida la herramienta OpenFST [28] utilizada por Java/PROSEGA para la manipulación de autómatas. De igual manera, al no encontrar herramientas disponibles para la generación de lenguajes que reemplazaran a Lextools, el autor de este trabajo desarrolló una librería de generación de lenguajes a la que se denominó fsm2language. Por otra parte, para la graficación de autómatas fue mantenido el software Graphviz [29] [54] al encontrarse de manera gratuita y estar disponible para sistemas operativos Windows.

- **Desarrollo de una librería propia para la generación del lenguaje de autómatas:** Como se explicó anteriormente, al no encontrar un sustituto de código abierto y disponible en Windows para Lextools, se desarrolló

una librería propia para la generación del lenguaje reconocido por autómatas a la que el autor del presente trabajo denominó *fsm2language*. Incluso, esta librería supera a Lextools (específicamente en la función *lexfsmstrings* [57]) pues para el caso de autómatas con lenguajes infinitos se desarrolló un método que permite imprimir un subconjunto del lenguaje reconocido por el autómata para así llevar a cabo las tareas de análisis. Una completa explicación de cómo fue implementada esta librería se encuentra en la Sección 3.6 del presente trabajo.

- **Apoyo de Java/PROSEGA al proceso de verificación del estándar IEEE 802.16:** Actualmente, Java/PROSEGA está siendo utilizado como parte del trabajo de análisis y verificación del estándar IEEE 802.16 [26], específicamente en el análisis de la gestión de conexiones (apertura, modificación y cierre de las conexiones) entre entidades solicitantes y subscriptoras a nivel de la capa MAC. Morales A. V. Este trabajo ha tenido como guía el proceso de verificación de protocolos de Billington et. al. [4]. Actualmente, luego de haber desarrollado la especificación del servicio [22], este trabajo se encuentra en la fase de especificación del protocolo. En esta fase Java/PROSEGA ha sido utilizado para generar los autómatas asociados a diversos módulos de la especificación del protocolo. Al final de este trabajo, Java/PROSEGA podrá ser utilizado para realizar comparaciones entre el lenguaje del servicio y el lenguaje del protocolo a fin de determinar incongruencias o errores presentes en el estándar IEEE 802.16.

5.2 Trabajo Futuro

A continuación, se presentan los posibles trabajos a futuro que pudieran estar relacionados con el software Java/PROSEGA, o relaciones con la línea de investigación en la cual está enmarcado el software Java/PROSEGA.

- **Actualización de Java/PROSEGA:** A medida que sea utilizado el software Java/PROSEGA para llevar a cabo las tareas de minimización de Grafos de Estado y análisis de lenguaje, pueden surgir consideraciones que motiven la actualización de Java/PROSEGA a una nueva versión. En particular, Java/PROSEGA seguirá siendo parte del trabajo de verificación del estándar IEEE 802.16. A medida que se avance en este trabajo es posible que surjan ciertas consideraciones o escenarios que motiven mejores, actualizaciones al software e incluso agregación de nuevas características.

- **Desarrollo de una versión de Java/PROSEGA no-integrada a CPN Tools:** Si surge la necesidad, es posible desarrollar una versión de Java/PROSEGA que no esté integrada a CPN Tools. Es decir, una versión del software que no esté embebida dentro del editor gráfico de CPN Tools. De esta manera, esta versión alternativa de Java/PROSEGA no dependería su puesta en marcha a partir del arranque de CPN Tools y del servidor de extensiones sino que más bien podría, a través del protocolo de comunicación BIS, conectarse desde un ambiente externo al simulador de CPN Tools cuando se requiera, y funcionar así de una manera independiente.

- **Conexión de Java/PROSEGA con Wireshark:** Dado que el caso de estudio en el cual fue concebido el software Java/PROSEGA fue el análisis y verificación de protocolos, puede desarrollarse un proyecto en cual se realice una integración el analizador de paquetes de protocolos Wireshark [65] para

así realizar una verificación interactiva entre autómatas construidos mediante Java/PROSEGA y paquetes de comunicación de distintos protocolos capturados mediante Wireshark.

- **Redes de Petri Coloreadas como Servicios (CPNaaS, *Coloured Petri Nets as a Service*)**: Este trabajo a futuro no está directamente relacionado con Java/PROSEGA pero se coloca aquí al estar dentro de la línea de investigación que ha sido trabajada por el autor del presente trabajo. Este es un proyecto, que inicialmente está propuesto por Westergaard M en [66] y que se encuentra en fase de desarrollo. El proyecto radica en el desarrollo de un API (o *middleware*) para utilizar las Redes de Petri Coloreadas en un paradigma de servicios web a través de internet en el que el simulador de CPN Tools sea utilizado como el motor de un servidor de aplicaciones web. De esta manera, puede ser posible desarrollar aplicaciones web teniendo como base el uso de Redes de Petri Coloreadas y el simulador de CPN Tools. El anexo "G" del presente trabajo introduce brevemente este concepto para trabajos futuro.

Referencias

[1] Jensen K. and Kristensen L. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer-Verlag, Berlin, 2009.

[2] Petri C.A. *Communication with Automata*. Tech. Rep. RADC-TR-65-377, Rome Air Dev. Center. New York, 1966.

[3] Barrett W.A. and Couch J. *Compiler Construction: Theory and Practice.* Science Reasearch Associates. Chicago, 1979.

[4] Billington J., Gallasch G. and Bing H. *A Coloured Petri Net Approach to Protocol Verification.* Lectures on Concurrency and Petri Nets, vol. 3098. 2004. pp. 210-290.

[5] CPN Tools Homepage. http://www.cpntools.org

[6] Knudsen L., Löfgren M., Madsen O. L., and Magnusson B. *Object-Oriented Environments – The Mjølner Approach*. Prentice Hall, 1994. Ch. 6, pp. 100-118.

[7] Milner R., Tofte M., Harper R., and MacQueen D. *The Definition of Standard ML (Revised).* MIT Press, 1997. ISBN 0-262-63181-4.

[8] Westergaard M. *CPN Tools 4 Extensions: Part 1: Basics*. 29 September 2013. [Blog entry]. Available: https://westergaard.eu/2013/09/cpn-tools-4-extensions-part-1-basics/.

[9] Westergaard M. and Kristensen L. *The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator*. Proceedings of the 30th International Conference on Applications and Theory of Petri Nets, Paris, 2009. pp. 313-322.

[10] Villapol M.E. *Modelling and Analysis of the Resource Reservation Protocol Using Coloured Petri Nets*. Doctoral Thesis, University of South Australia, November 2003.

[11] Royce W. *Managing the Development of Large Software Systems: Concepts and Techniques.* Technical Papers of Western Electronic Show and Convention (WesCon). Los Angeles, August 1970.

[12] Westergaard M. *CPN Tools 4 Extensions: Part 2: Getting Started and Basic Abstractions*. 29 September 2013. [Blog entry]. Available: https://westergaard.eu/2013/09/cpn-tools-4-extensions-part-2-getting-started-and-basic-abstractions/

[13] Westergaard M. *CPN Tools 4 Extensions: Part 3: Graphics and Callbacks*. 1 October 2013. [Blog entry]. Available: https://westergaard.eu/2013/10/cpn-tools-4-extensions-part-3-graphics-and-callbacks/

[14] Westergaard M. *CPN Tools 4 Extensions: Part 4: Advanced Communication and Debugging*. 5 November 2013. [Blog entry]. Available: https://westergaard.eu/2013/11/cpn-tools-4-extensions-part-4-advanced-communication-and-debugging/

[15] Westergaard M. *CPN Tools 4 Extensions: Part 5: Extension Overview*. 13 December 2014. [Blog entry]. Available: https://westergaard.eu/2014/12/cpn-tools-4-extensions-part-5-extension-overview/

[16] Westergaard M., Fahland D. and Stahl C. *Grade/CPN: Semi-automatic support for teaching Petri nets by checking many Petri nets against one specification.* Proceedings of the International Workshop on Petri Nets and Software Engineering. Hamburg, June 2012. pp. 32-46.

[17] Gallasch G. and Kristensen L.M. *Comms/CPN: A Communication Infrastructure for External Communication with Design/CPN.* Proceedings of the 3rd Workshop on Practical Use of Coloured Petri Nets and the CPN Tools. Department of Computer Science, University of Aarhus, 2001. pp. 79-93.

[18] Christensen S., Jørgensen J.B. and Kristensen L.M. *Design/CPN – A Computer Tool for Coloured Petri Nets*. Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS '97). Springer-Verlag, London, 1997. pp. 209-223.

[19] CPN Tools Homepage. Comms/CPN. http://cpntools.org/documentation/concepts/external/external_communication_wi

[20] Kristensen L.M. and Westergaard M. *The ASCoVeCo State Space Analysis Platform: Next Generation Tool Support for State Space Analysis.*

[21] ASCoVeCO Project Homepage. http://www.cs.au.dk/~ascoveco/index.html.

[22] Morales A.V. *Modelado y Análisis de los Procesos Involucrados en la Gestión de las Conexiones en la Capa MAC IEEE 802.16 utilizando Redes de Petri Coloreadas (CPNs).* Trabajo de Ascenso a la Categoría de Asistente. Escuela de Computación, Universidad Central de Venezuela. Caracas, 2010.

[23] Black U. *OSI: A Model of Computer Communications Standards.* Prentice-Hall, New Jersey, 1991.

[24] Postel J. *Transmission Control Protocol – DARPA Internet Program Protocol Specification.* RFC 793, DARPA, September 1981. Available: http://tools.ietf.org/html/rfc793.

[25] Braden R., Zhang L., Berson S., Herzog S. and Jamin S. *Resource Reservation Protocol (RSVP) – Version 1 Functional Specification.* RFC 2205, September 1997.

[26] IEEE Sta. 802.16-2001. *Local and Metropolitan Area Network, Part 16: Air Interface for Fixed Broadband Wireless Access Systems.* October 2002.

[27] AT&T Labs. *AT&T FSM Library Factsheet*. Available: http://www.research.att.com/export/sites/att_labs/library/documents/licensing_data_sheets/fsmlibrary_factsheet_20090925.pdf. 2009.

[28] Riley M., Schalkwyk J., Skut W. and Mohri M. *OpenFST: A General and Efficient Weighted Finite-State Transducer Library (Extended Abstract of an Invited Talk).* Proceedings of the 12th International Conference on Implementation and Application of Automata. Prague, Czech Republic, July 2007. pp. 11-23.

[29] Graphviz. Graph Visualization Software. http://www.graphviz.org.

[30] Murata T. *Petri Nets: Properties, Analysis and Applications.* Proceedings of the IEEE, Vol. 77, No. 4, April 1989. pp. 541-580.

[31] ISO/IEC 15909-1:2004. *Systems and Software Engineering – High-level Petri Nets – Part 1: Concepts, Definition and graphical notation.* 2004.

[32] Jensen K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* Monographs on Theoretical Computer Science, vol.1: Basic Concepts. Springer-Verlag, Berlin, 1997.

[33] Jensen K, Christensen S. and Kristensen L.M. *CPN Tools State Space Manual*. University of Aarhus, January 2006.

[34] Aarhus University. Department of Computer Science. Coloured Petri Nets Group. http://cs.au.dk/cpnets/.

[35] Eindhoven University of Technology. Chair Architecture of Information Systems (AIS Group). http://www.win.tue.nl/ais/doku.php.

[36] CPN Tools Homepage. Download Section. http://cpntools.org/download.

[37] Design/CPN Online. Home Section. http://www.daimi.au.dk/designCPN/.

[38] Design/CPN Online. Examples of Industrial Use of CP-nets. http://daimi.au.dk/CPnets/proxy.php?url=/CPnets/intro/example_indu.

[39] Gordon S.D. *Verification of the WAP Transaction Layer using Coloured Petri Nets*. Doctoral Thesis, University of South Australia, November 2001.

[40] CPN Tools Homepage. Feedback/Support for CPN Tools Section. *Can CPN Tools open models from Design/CPN?*. Available: http://support.cpntools.org/knowledgebase/articles/102344-can-cpn-tools-open-models-from-design-cpn.

[41] ISO/IEC 15909-1:2004. *Systems and software engineering - High-level Petri nets - Part 1: Concepts, definitions and graphical notation.* December 2004.

[42] ISO/IEC 15909-2:2011. *Systems and software engineering - High-level Petri nets - Part 2: Transfer Format*. February 2011.

[43] CPN Tools Homepage. Documentation Section. *DTD for net files.* Available: http://cpntools.org/documentation/concepts/external/dtd_for_netfiles.

[44] CPN Tools Homepage. Internal Documentation. *Communication Architecture.* Available: http://cpntools.org/cpn2000/apn_ml_protocol_manual.

[45] Oracle Corporation. Java Software. https://www.oracle.com/java/index.html.

[46] Eckel B. *Thinking in Java: The definitive introduction to object-oriented programming in the language of the World Wide Web*. (4TH Edition). Prentice Hall, February 2006. ISBN-10: 0131872486.

[47] Oracle Corporation. Java™ Platform, Standard Edition 7. API Specification. http://docs.oracle.com/javase/7/docs/api/.

[48] Oracle Corporation. The Java™ Tutorials. *Trail: Creating a GUI with JFC/Swing (also known as The Swing Tutorial).*
Available: https://docs.oracle.com/javase/tutorial/uiswing/index.html.

[49] Westergaard M. *The BRITNeY Suite: A Platform for Experiments.* Proceedings of the 7th Workshop on Practical Use of Coloured Petri Nets and the CPN Tools. Aarhus, Denmark, October 2006.

[50] Carrasco M. Class Lecture, Topic*: Teoría de Autómatas. Matemáticas Discretas III (Semestre Lectivo 01-2010).* Escuela de Computación, Facultad de Ciencias, Universidad Central de Venezuela. Caracas, Venezuela, April 2010.

[51] Hopcroft J.E., Motwani R. and Ullman J.D. *Introduction to Automata Theory, Languages and Computation.* (3RD Edition). Pearson, July 2006.
ISBN-10: 0321455363.

[52] OpenFST Library. Homepage. http://www.openfst.org/.

[53] OpenFST Library. Download Section.
http://www.openfst.org/twiki/bin/view/FST/FstDownload.

[54] Gansner E.R., Koutsofios E, North S. and Vo K-P. *A Technique for Drawing Directed Graphs.* IEEE Transactions on Software Engineering Vol.19, Issue.3. AT&T Bell Laboratories. Murray Hill, New Jersey, USA, March 1993.

[55] Eclipse. Eclipse Modelling Framework (EMF).
http://www.eclipse.org/modeling/emf.

[56] BRITNeY Suite. Tutorial of the BRITNeY Suite. http://britney.szm.com/en/.

[57] Sproat R. *Lextools: Tools for finite-state linguistic analysis.* Technical Report 11522-951108-10TM, Bell Laboratories, 1995.

[58] Eclipse. Eclipse Homepage. https://www.eclipse.org/.

[59] Coto E. *Algoritmos Básicos de Grafos*. Lectures on Computer Science. ND 2003-02. Universidad Central de Venezuela, Facultad de Ciencias, Escuela de Computación, Laboratorio de Computación Gráfica. Caracas, February 2003. ISSN: 1316-6239.

[60] Morales A., and Villapol M.E. *Modelado y análisis inicial de la especificación del servicio de la capa MAC del IEEE 802.16 utilizando Redes de Petri Coloreadas* (*CPN*). Proceedings on the 33th Latin American Computing Conference (CLEI 2007). Costa Rica, 2007.

[61] Morales A., and Villapol M.E. *Reviewing the Service Specification of the IEEE 802.16 MAC Layer Connection Management: A Formal Approach*. CLEI Electronic Journal, vol. 16, number. 2, paper. 2. Montevideo, August 2013.
ISSN: 0717-5000.

[62] IEEE Sta. 802.16-2004. *Local and Metropolitan Area Network, Part 16: Air Interface for Fixed Broadband Wireless Access Systems.* October 2004.

[63] IEEE Sta. 802.16e-2005. *Local and Metropolitan Area Network, Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems, Amendment 2: Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands and Corrigendum 1.* February 2006.

[64] ITU-T Convention for the definition of OSI Services. *Recommendation X.210.* November 1993.

[65] Wireshark. Official Homepage. https://www.wireshark.org/.

[66] Westergaard M. *CPNaaS – Colored Petri Nets as a Service.* 6 May 2015. [Blog entry]. Available: https://westergaard.eu/2015/05/cpnaas-colored-petri-nets-as-a-service/.

[67] Apache Subversion. Online home of the Apache Subversion™ software project. https://subversion.apache.org/.

[68] Eclipse. Eclipse Subversive - Subversion (SVN) Team Provider. http://www.eclipse.org/subversive/.

[69] Eclipse. Download Section. Eclipse Java Luna SR2 v4.4.2 Win32. Available: https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/luna/SR2/eclipse-java-luna-SR2-win32.zip.

[70] Eclipse. Updated releases of EMF.
Available: http://download.eclipse.org/modeling/emf/updates/releases/.

[71] Polarion Software. Subversive. Available:
http://community.polarion.com/projects/subversive/download/eclipse/4.0/luna-site/.

[72] Subversion. CPN Tools Repository. Access/CPN.
Available: https://svn.win.tue.nl/repos/cpntools/AccessCPN.

[73] EMF Commons Library. Available:
http://www.java2s.com/Code/Jar/o/Downloadorgeclipseemfcommon290v201305280742jar.htm.

[74] EMF Core Library. Available:
http://www.java2s.com/Code/Jar/o/Downloadorgeclipseemfecorexmi290v201305280742jar.htm.

[75] GitLab Repository. CPN Tools. Simulator Extension Packages.
Available: https://gitlab.westergaard.eu/cpn-tools/simulator/tree/versions/4.0.0.

[76] Google Research. https://research.google.com.

[77] New York University. Courant Institute of Mathematical Sciences.
https://cims.nyu.edu.

ANEXO

A

Instalación y configuración de Access/CPN

Access/CPN [9] es una herramienta adicional para CPN Tools [5] que provee un conjunto de módulos que hacen posible desarrollar aplicaciones basadas en Java que se conecten al simulador de CPN Tools. En este anexo se describen los pasos necesarios, en base a la experiencia particular del autor del presente trabajo de investigación, para llevar cabo la instalación y configuración de la herramienta Access/CPN de manera de desarrollar aplicaciones en Java que utilicen esta herramienta.

La herramienta Access/CPN basa su uso en el entorno de desarrollo integrado Eclipse [58] y el framework EMF [55]. Por lo tanto, se explica primero en el presente anexo, la instalación de Eclipse y EMF, y posteriormente la descarga y configuración de la herramienta Access/CPN. La herramienta Access/CPN se encuentra alojada en un repositorio de Subversion [67]. Por esta razón, también es explicado el uso de la herramienta Subversive [68] de Eclipse para descargar los distintos módulos que componen la herramienta Access/CPN.

En la experiencia de este trabajo, por facilidad, se creó una carpeta llamada TESIS en la raíz del disco local del computador utilizado para el presente trabajo (C:/TESIS). En esta carpeta se incluyen todas las herramientas y proyectos a utilizar.

A.1 Instalación de Eclipse

Eclipse [58] es un entorno de desarrollo integrado que permite la gestión y desarrollo de proyectos de desarrollo de software. La versión de Eclipse utilizada en el presente trabajo es la versión Luna SR2 (4.4.2) para sistemas operativos Windows de 32-bits. Esta versión se puede descargar directamente a través del enlace contenido en [69].

El software Eclipse en la versión utilizada viene comprimido en un archivo de extensión .zip. Una vez descargado el archivo, este es extraído en una carpeta llamada "Eclipse". En la experiencia realizada para el presente trabajo, la herramienta fue colocada en la ruta C:/TESIS/Eclipse.

A.2 Configuración del espacio de trabajo

El espacio de trabajo (*workspace*) es el directorio donde estarán alojados todos los proyectos a utilizar o desarrollar mediante el uso de Eclipse y, en general, es donde se alojan todos los códigos fuentes utilizados en el presente trabajo. Este espacio de trabajo fue creado en la ruta C:/TESIS/Workspace.

Cuando se procede a ejecutar Eclipse, la aplicación pregunta al usuario que espacio de trabajo se utilizará. En este caso se colocará el directorio mencionado C:/TESIS/Workspace. Luego de esto, Eclipse procede a cargar el espacio de trabajo en el panel izquierdo de la aplicación. Además, Eclipse provee un panel para el manejo de archivos y un conjunto de consolas para visualizar las trazas de las aplicaciones y para funciones de depuración. La figura A.1 muestra una vista general de Eclipse luego que se coloca el espacio de trabajo por defecto.

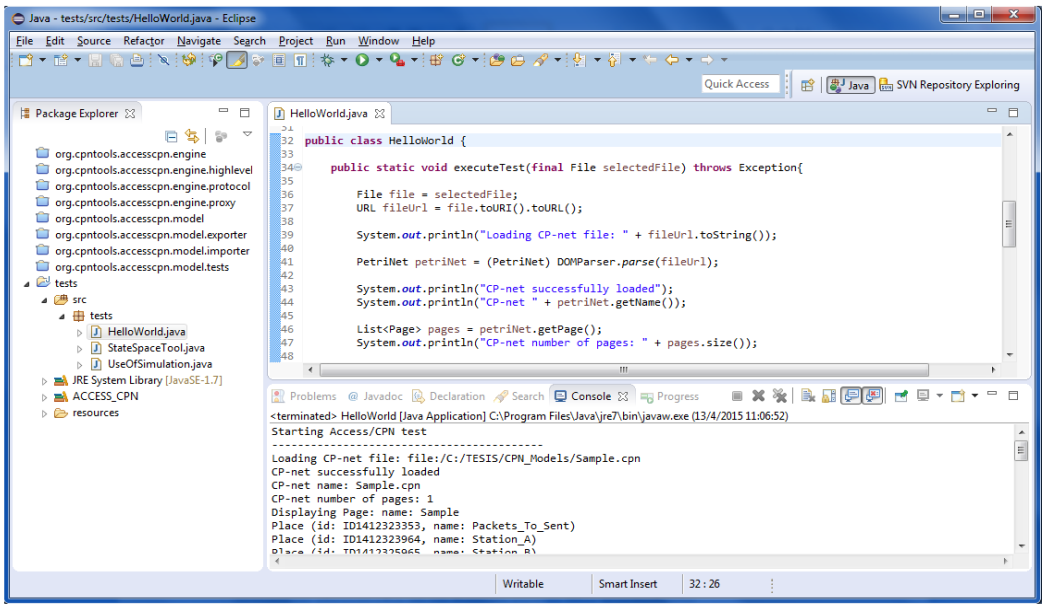

Figura A.1. Vista general de Eclipse con el espacio de trabajo cargado.

A.3 Instalación de EMF

EMF (*Eclipse Modeling Framework)* [55] es un framework de modelado que permite transformar un modelo representado gráficamente a una clase en Java. Este framework es una librería utilizada por los distintos módulos de la herramienta Access/CPN. Access/CPN utiliza este framework para, entre otras cosas, transformar un modelo CPN a una representación orientada a objetos.

EMF puede ser descargado a través de la herramienta de descarga de software adicional de Eclipse. En la pestaña "Ayuda" (*Help)* de Eclipse se hace click en la opción "Instalar Nuevo Software" (*Install New Software*). Se coloca como fuente el enlace citado en [70]. La figura A.2 muestra la ventana de instalación de software adicional para Eclipse que permite la descarga de EMF. Allí se debe descargar el paquete *EMF – Eclipse Modelling Framework SDK*. Particularmente, en el presente trabajo se utiliza la versión 2.10.2 de EMF.

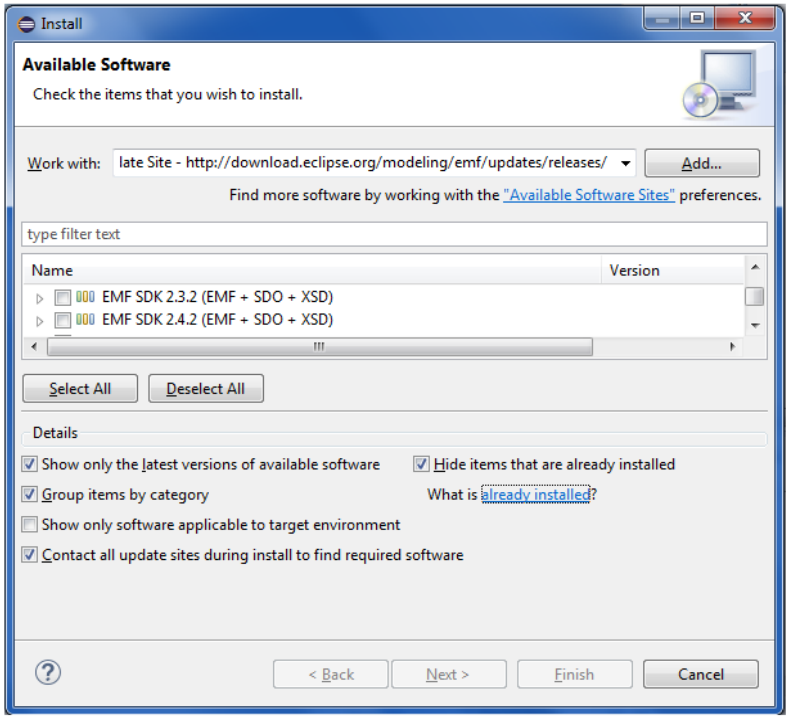

Figura A.2. Instalación de EMF a través de Eclipse.

A.4 Subversive

Access/CPN viene empaquetado en un conjunto de proyectos en Java. Este conjunto de proyectos o módulos pueden ser descargados en la sección de descarga de la página oficial de CPN Tools [5].

Sin embargo, una limitante de esta opción de descarga es que los módulos vienen comprimidos en formato JAR, por lo que los módulos no pueden ser manipulados o modificados libremente. Una mejor alternativa es descargar los módulos de un repositorio en Subversion donde se encuentran almacenados en un formato para su libre manipulación donde es visible el código fuente. De esta manera se podrán editar los módulos de Access/CPN en caso que sea necesario y se podrá tener acceso a los diferentes archivos que componen cada módulo.

Subversion es un repositorio en línea para el alojamiento de versiones de proyectos de software. Para descargar los módulos de Access/CPN que están alojados en Subversion se utiliza, dentro del ambiente de Eclipse, la herramienta Subversive.

Subversive [68] es una aplicación cliente embebida dentro de Eclipse que se conecta al servidor de Subversion para la descarga de proyectos. Para instalar Subversive se utiliza, al igual que para la instalación de EMF, la herramienta de instalación de software adicional de Eclipse. La fuente a utilizar es el enlace citado en [71]. Los paquetes que se deben instalar para el correcto funcionamiento de Subversive son los colocados en la tabla A.1.

Tabla A.1. Paquetes de Subversive instalados en Eclipse.

Paquete de Subversive a instalar	Versión
Subversive SVN Connectors	v4.1.3.I20150214-170
Subversive SVN Team Provider	v2.0.4.I20150123-1700
Subversive SVN Team Provider Sources	v2.0.4.I20150123-1700

Luego de instalar Subversive, se reiniciará Eclipse. Cuando Eclipse arranca nuevamente se debe instalar el siguiente paquete que ofrece justo después del reiniciarse: *SVNKit v1.7.12 Implementation*.

A.5 Descarga de Access/CPN

Finalmente, luego de haber instalado Subversive, es posible descargar los módulos de Access/CPN. Para esto, se debe cambiar la perspectiva de Eclipse a la funcionalidad de búsqueda de repositorios de Subversion (*SVN Repository Exploring*). La figura A.3 muestra cómo se puede acceder a dicha perspectiva. En dicha perspectiva de búsqueda de repositorios, se debe hacer click en la función de buscar nuevo repositorio.

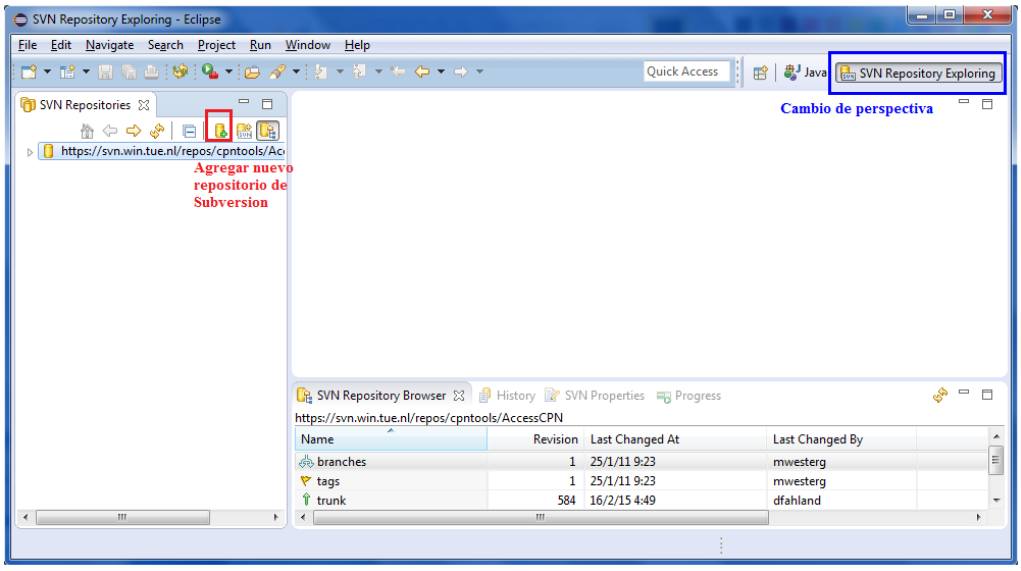

Figura A.3. Perspectiva de exploración de repositorios en Eclipse.

Cuando se usa la opción de agregar nuevo repositorio, se abre una ventana donde se colocará el URL del repositorio de Subversion donde se encuentran alojados los distintos de Access/CPN. El URL utilizado es [72] (ver Figura A.4).

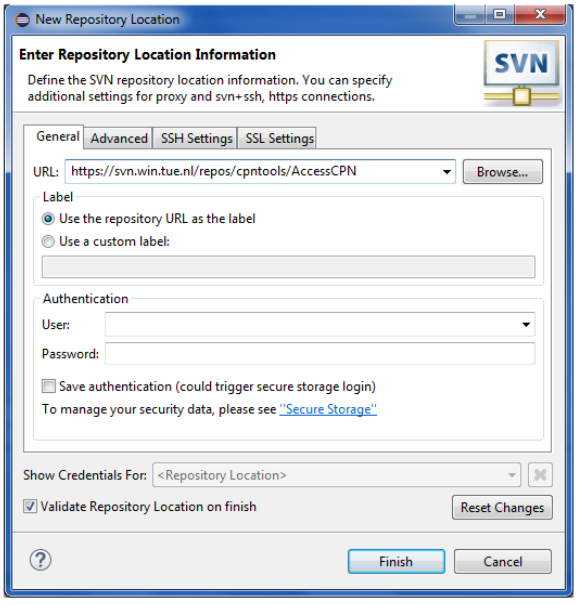

Figura A.4. Búsqueda del repositorio de Access/CPN mediante Eclipse.

Una vez conectado al repositorio, los módulos de Access/CPN que se encuentran en el repositorio podrán ser visualizados en el panel izquierdo de Eclipse. Para poder trabajar con estos módulos de Access/CPN, estos deben ser exportados al espacio de trabajo local (en el computador local). En el presente trabajo, estos módulos fueron exportados al directorio local C:/TESIS/Workspace.

Para exportar cada uno de los modelos al espacio de trabajo local, se debe hacer click derecho en cada uno de estos y seleccionar la opción de exportar ("*Export*"). Luego, debe colocarse la ruta a donde será exportado el módulo seleccionado (ver figura A.5).

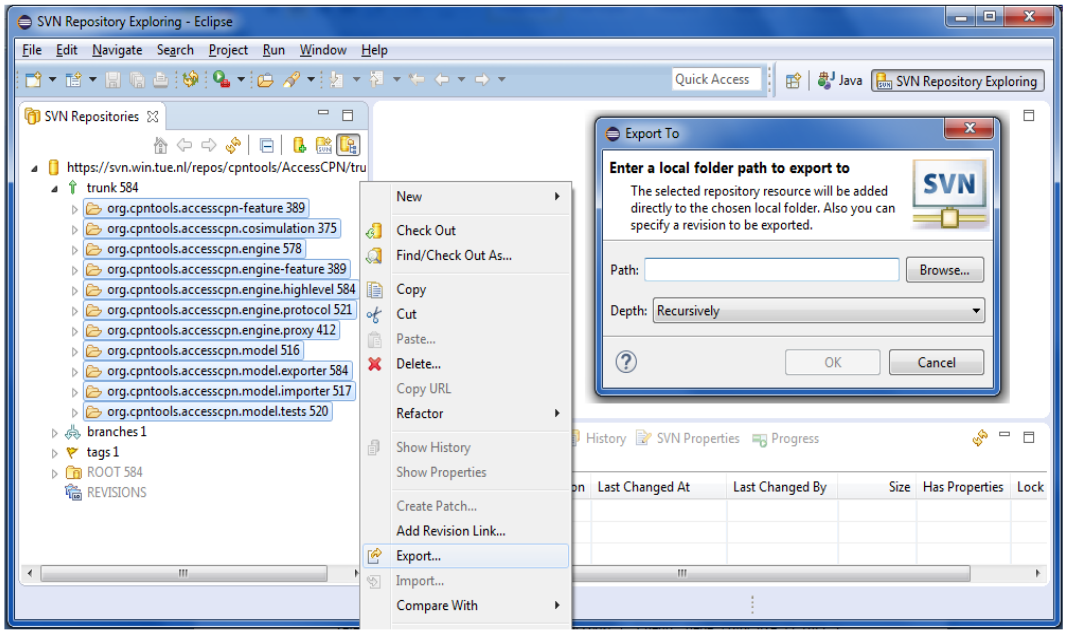

Figura A.5. Exportación de Access/CPN al espacio de trabajo local.

Este es un proceso que puede variar unos cuantos minutos dependiendo de la velocidad de descarga. La motivación de realizar esta descarga de los módulos de Access/CPN es que se están descargando los módulos en código fuente, lo que hace posible su manipulación y análisis. Esto a diferencia de descargar los módulos en

formato JAR a través de la página de CPN Tools, en donde no es posible la modificación o visualización de la manera que se construyeron los módulos de Access/CPN. Una vez realizado el proceso de exportación se puede pasar a trabajar nuevamente en la perspectiva de trabajo por defecto de Eclipse.

A.6 Configuración de los módulos

En este punto ya se poseen los módulos de Access/CPN en el directorio local C:/TESIS/Workspace. Sin embargo, para ser trabajados con Eclipse, estos deben ser importados desde la herramienta Eclipse a través de la opción de importar proyectos ("*Import"*). En el presente trabajo los proyectos fueron importados uno por uno.

Es recomendable que el primer proyecto a importar sea el módulo *model*, luego pueden importarse los módulos *engine, engine.highlevel, engine.protocol,* y después los módulos *model.exporter* y *model.importer*. Finalmente, pueden ser importados los módulos restantes. Esto se realiza en este orden puesto que el módulo *model* no tiene dependencia alguna con los demás módulos de Access/CPN.

Entre los módulos CPN existen relaciones de uso entre las clases que componen dichos módulos, por lo que es probable, que a medida que se importen los módulos, surjan ciertos errores. Pueden surgir a la hora de importar, errores en un módulo, debido a que las clases que componen dicho módulo en específico no encuentran cierta clase o método que utilizan y que se encuentra en otro módulo.

La solución será crear en Eclipse una librería llamada ACCESS_CPN donde se coloquen los archivos comprimidos en formato JAR de cada uno de los módulos de Access/CPN y que está librería sea utilizada por cada uno de los módulos de Access/CPN. De esta manera se resuelve el problema de dependencia entre módulos.

La figura A.6 muestra como exportar a formato JAR los módulos que van quedando exentos de errores, la figura A.7 como agregar la librería ACCESS_CPN a un módulo en particular, y la figura A.8 muestra cómo crear librería ACCESS_CPN y como agregar los módulos en formato JAR a la librería.

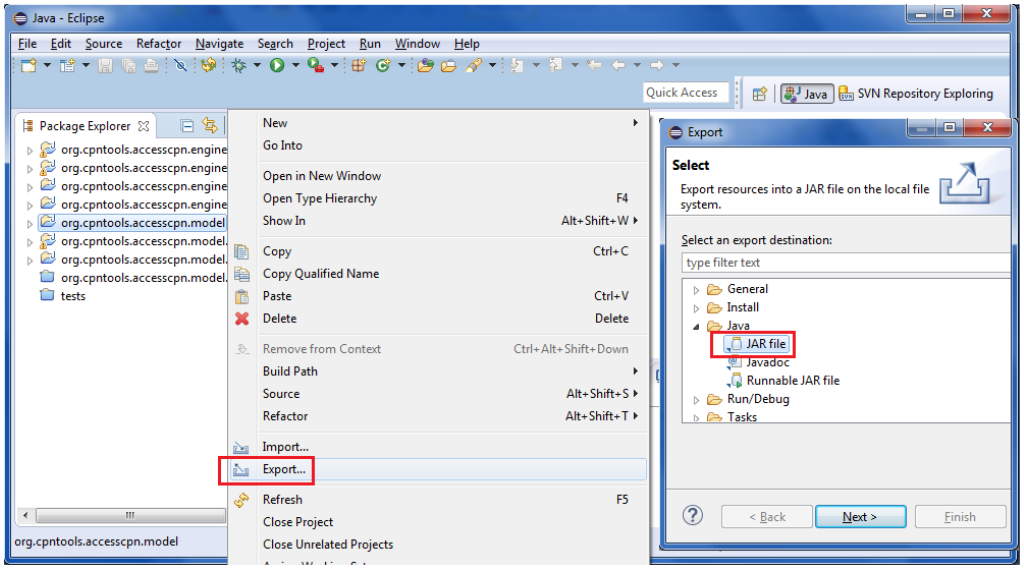

Figura A.6. Exporte de un módulo CPN a formato JAR.

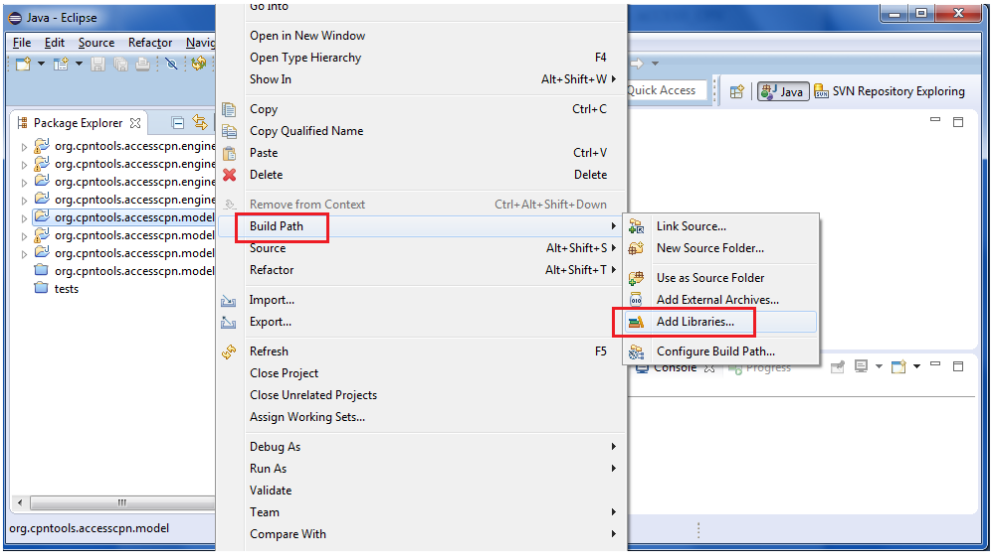

Figura A.7. Añadir la librería ACCESS_CPN a crear en un módulo en particular.

El objetivo que se busca es que esta librería sea utilizada por cada uno de los módulos de Access/CPN para solventar así los errores surgidos por las relaciones de dependencia.

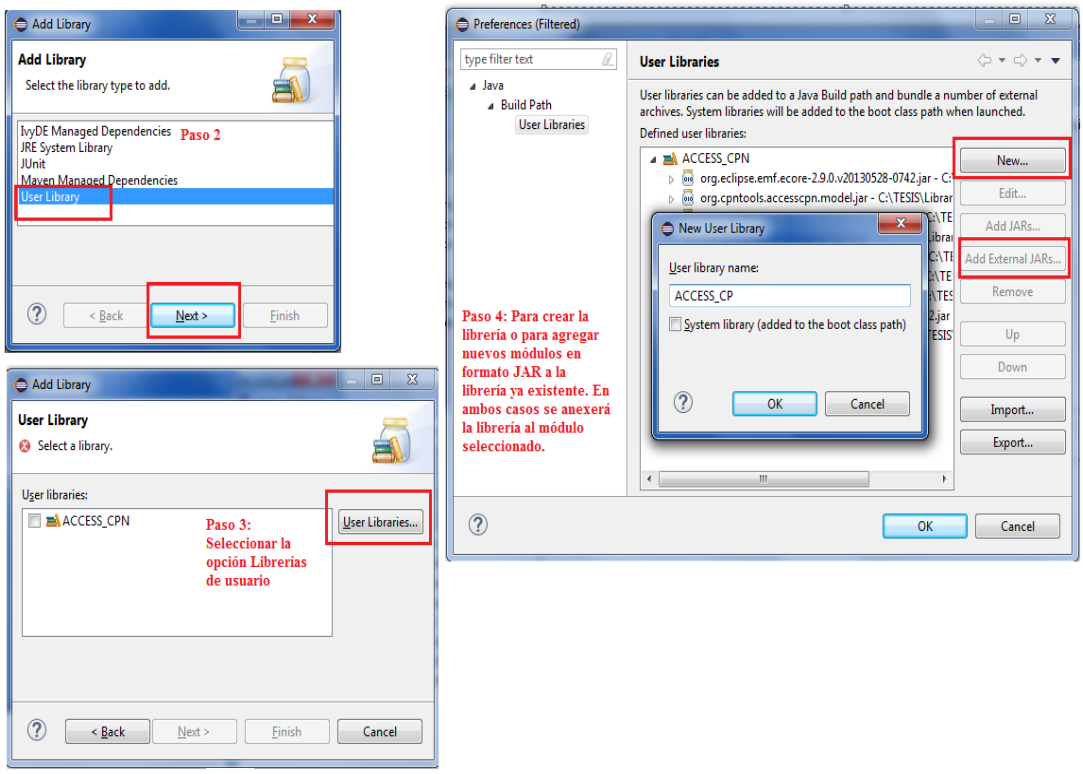

Figura A.8: Creación de la librería ACCESS_CPN y agregación de los módulos en JAR.

Adicionalmente, en esta experiencia de configuración fue necesaria la descarga de dos archivos JAR de librerías faltantes de EMF que pudieron no ser provistas en la instalación de EMF. Estos archivos son los siguientes:

- org.eclipse.emf.common-2.9.0.v20130528-0742.jar [73].
- org.eclipse.emf.ecore-2.9.0.v20130528-0742.jar [74].

Finalmente, la figura A.9 muestra cómo queda estructurada la librería ACCESS_CPN que es utilizada por los distintos módulos CPN. Vale resaltar que esta librería también deberá ser utilizada por las aplicaciones a desarrollar que utilicen Access/CPN puesto que contiene empaquetados los distintos módulos de Access/CPN.

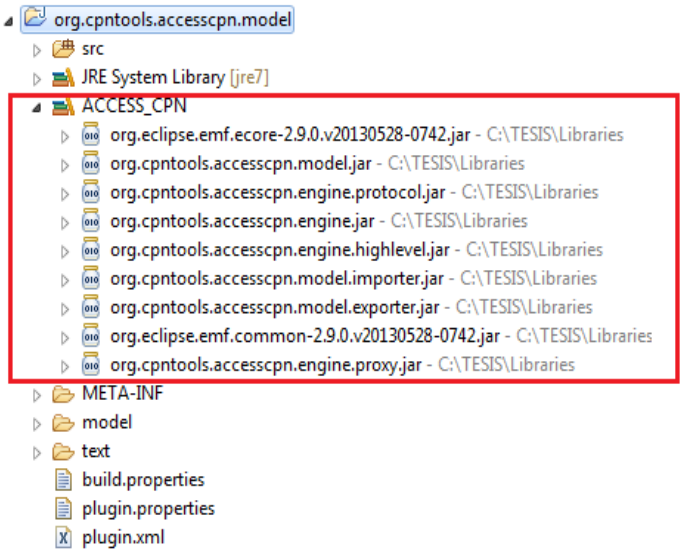

Figura A.9: Estructura de la librería ACCESS_CPN dentro de un módulo.

A partir de este punto, con los módulos de Access/CPN descargados y ya embebidos dichos módulos dentro de una librería, es posible el desarrollo de programas utilizando la herramienta Access/CPN.

A.7 Ejemplo de programa que utiliza Access/CPN

Para finalizar este tutorial de configuración de Access/CPN, se presenta un ejemplo (código A.1) extraído de [9] que ilustra cómo se carga un modelo CPN y luego utiliza la interfaz en SML de Access/CPN para ejecutar un algoritmo de exploración del Grafo de Estado.

```
1  import java.io.File;
2  import java.net.InetAddress;
3  import java.net.URL;
4
5  import org.cpntools.accesscpn.engine.DaemonSimulator;
6  import org.cpntools.accesscpn.engine.Simulator;
7  import org.cpntools.accesscpn.engine.highlevel.HighLevelSimulator;
8  import org.cpntools.accesscpn.engine.highlevel.checker.Checker;
9  import org.cpntools.accesscpn.model.PetriNet;
10 import org.cpntools.accesscpn.model.importer.DOMParser;
11
12
13 public class StateSpaceTool {
14     public static void main(final String[] args) throws Exception {
15
16         File file = new File(args[0]);
17         URL fileUrl = file.toURI().toURL();
18
19         final PetriNet petriNet = DOMParser.parse(fileUrl);
20         final HighLevelSimulator s = HighLevelSimulator.getHighLevelSimulator();
21
22         try {
23             final Checker checker = new Checker(petriNet, null, s);
24             checker.checkEntireModel();
25             s.evaluate("use \"C:/TESIS/Workspace/org.cpntools.accesscpn.model.tests/resources/simple-dfs.sml\"");
26             System.out.println(s.evaluate("let
27                                             val (state, storage) = dfs dead (CPNToolsModel.getInitialStates())
28                                           in
29                                            (state, HashTable.numItems storage)
30                                           end"));
31         } finally {
32             s.destroy();
33         }
34     }
35 }
```

Código A.1. Programa de ejemplo que utiliza Access/CPN [9].

El programa mostrado es una simple herramienta de línea de comandos que utiliza un algoritmo desarrollado en SML [9, Ch. 2, Sec. 2] para chequear la existencia de *dead-locks* en el modelo pasado como parámetro. En ll. 5-10 se importan los módulos de Access/CPN utilizados por el programa. La ruta del modelo CPN utilizado es pasado como parámetro (ll. 14-16). En l. 19, el modelo CPN se carga dentro del objeto *PetriNet* a través del importador de modelos (véase Sección 2.7.4) y luego en l. 20 se crea el objeto *HighLevelSimulator* que es el simulador de CPN Tools en sí. En ll. 23-24 se realiza un chequeo de sintaxis y análisis del modelo. En l. 25 se carga el algoritmo en SML [9, Ch. 2, Sec. 2] para el chequeo de *dead-locks* en el simulador a través del método *evaluate*( ) y finalmente en ll. 26 - 30 se imprime por consola el resultado de invocar dicho método.

ANEXO

B

Configuración del manejo de extensiones para CPN Tools

Para configurar extensiones de CPN Tools se debe realizar una configuración de los paquetes de Extensión en el entorno de desarrollo integrado Eclipse [58]. De esta manera, se podrán trabajar en Eclipse con estos paquetes para desarrollar así aplicaciones en Java como extensiones para la herramienta CPN Tools. Estos paquetes de Extensión son de código abierto y pueden ser descargados a través del enlace [75].

Estos paquetes están contenidos en un archivo comprimido de extensión .zip. A continuación, la tabla B.1 presenta la lista de paquetes que trabajan el manejo de Extensiones.

Vale destacar que uno de estos paquetes (paquete *protocol*) pertenece a los módulos de Access/CPN [9] y es utilizando en el manejo de Extensiones para la comunicación con el servidor (implementación del protocolo BIS).

Tabla B.1. Paquetes para el desarrollo de Extensiones en CPN Tools.

Paquetes del manejo de Extensiones	**Descripción**
org.cpntools.accesscpn.engine.protocol (Access/CPN)	Provee estructuras para la comunicación con el simulador.
org.cpntools.simulator.extensions	Paquete principal. Incluye las interfaces centrales y el manejo de comunicación con el servidor.
org.cpntools.simulator.extensions.launcher	El lanzador actual para las extensiones.
org.cpntools.simulator.extensions.declare	Implementan tres funcionalidades extras para CPN Tools colocadas como extensiones: funcionalidad *declare,* exporte de PNML y manejo de gráficos.
org.cpntools.simulator.extensions.export	
org.cpntools.simulator.extensions.graphics	
org.cpntools.simulator.extensions.ranges	Implementa la funcionalidad de manejo de intervalos de tiempo para CPN Tools.

Una vez descargados todos estos paquetes a través del enlace [75**]**, estos se deben importar al espacio de trabajo de Eclipse (véase Sección A.2). Para que un proyecto a desarrollar en Java utilice todos estos paquetes, se deben empaquetar y exportar todos estos paquetes juntos a un formato de compresión .jar (esto se puede realizar mediante Eclipse, ver figura B.1 -derecha-).

De esta manera, proyectos en Java que tengan como fin ser una extensión de CPN Tools, utilizan estas librerías para utilizar las interfaces adecuadas (véase Sección 2.8). La figura B.1 -izquierda- muestra como un proyecto posee añadido una librería en formato .jar (org.cpntools.simulator.extensions.jar) que agrupa todos los paquetes del manejo de Extensiones (tabla B.1).

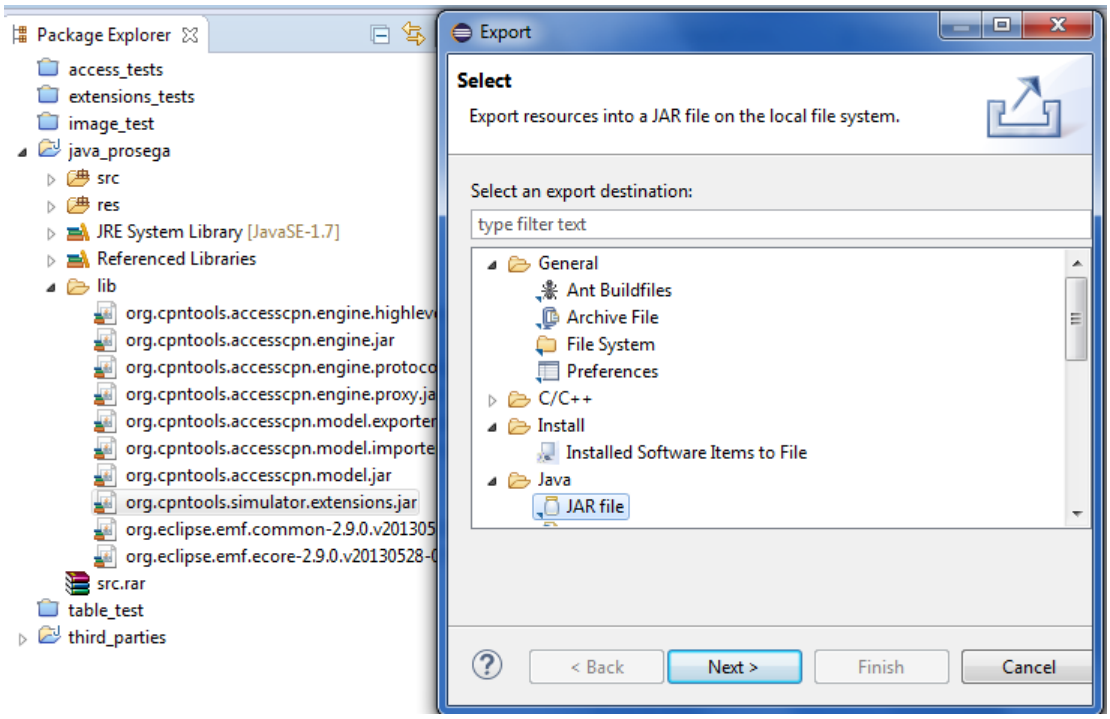

Figura B.1. Exportación de módulos a un JAR y agregación a un proyecto en Eclipse.

ANEXO

C

Debug/CPN

Debug/CPN es una útil herramienta de depuración provista por CPN Tools [5] (versión 4.0 o superior). Es una herramienta que fue agregada a CPN Tools a partir de la versión 4.0 porque es agregada como una extensión más (el archivo .jar de esta extensión se incluye, como cualquier otra extensión desarrollada por un usuario, en la carpeta *plugins* del directorio *extensions* de CPN Tools.

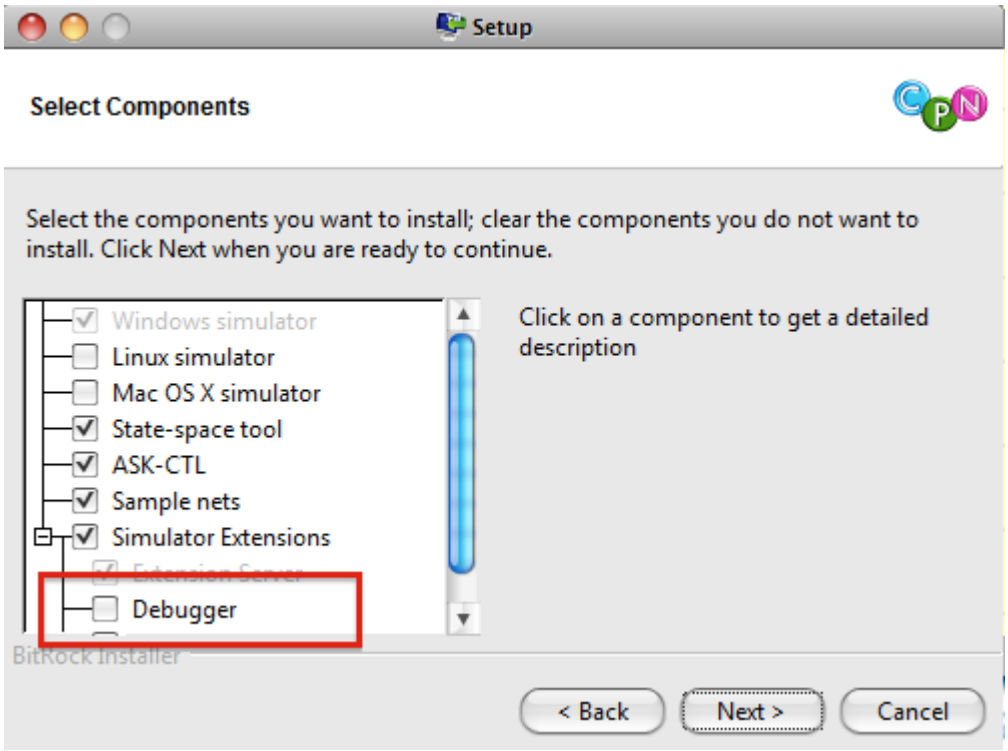

Figura C.1. Instalación de Debug/CPN.

Para que esta extensión esté presente en el directorio de extensiones de CPN Tools, se debe seleccionar la opción *debuuger* al momento de realizar la instalación de CPN Tools (véase figura C.1). Luego, para utilizarlo, una vez encendido CPN Tools y el servidor de extensiones, se accede a través del instrumento *Debug* en la caja de herramientas *Development*.

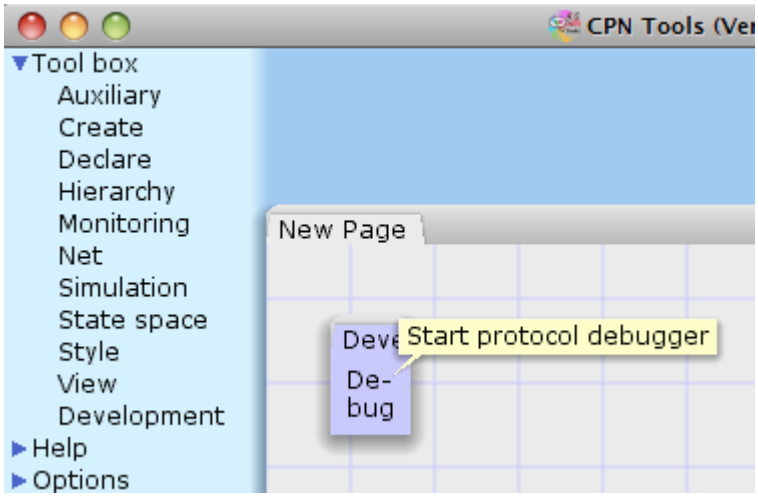

Figura C.2. Invocación de Debug/CPN en CPN Tools.

Debug/CPN provee, entre otras cosas, una interfaz para experimentar y realizar pruebas de comunicación con el simulador de CPN Tools utilizando como base el protocolo de comunicación BIS [19] (véase Sección 2.5.6).

Utilizando esta herramienta se puede entender como funciona el traspaso de paquetes entre un usuario y el simulador de CPN Tools a través del manejo de comandos (*command*), subcomandos (*subcommand*) y el manejo del formato BIS (listas booleana, de enteros y de *strings*). En este anexo se describe particularmente el funcionamiento de la pestaña *SendPackets* que permite enviar paquetes en el formato BIS al simulador de CPN Tools (ver figura C.1).

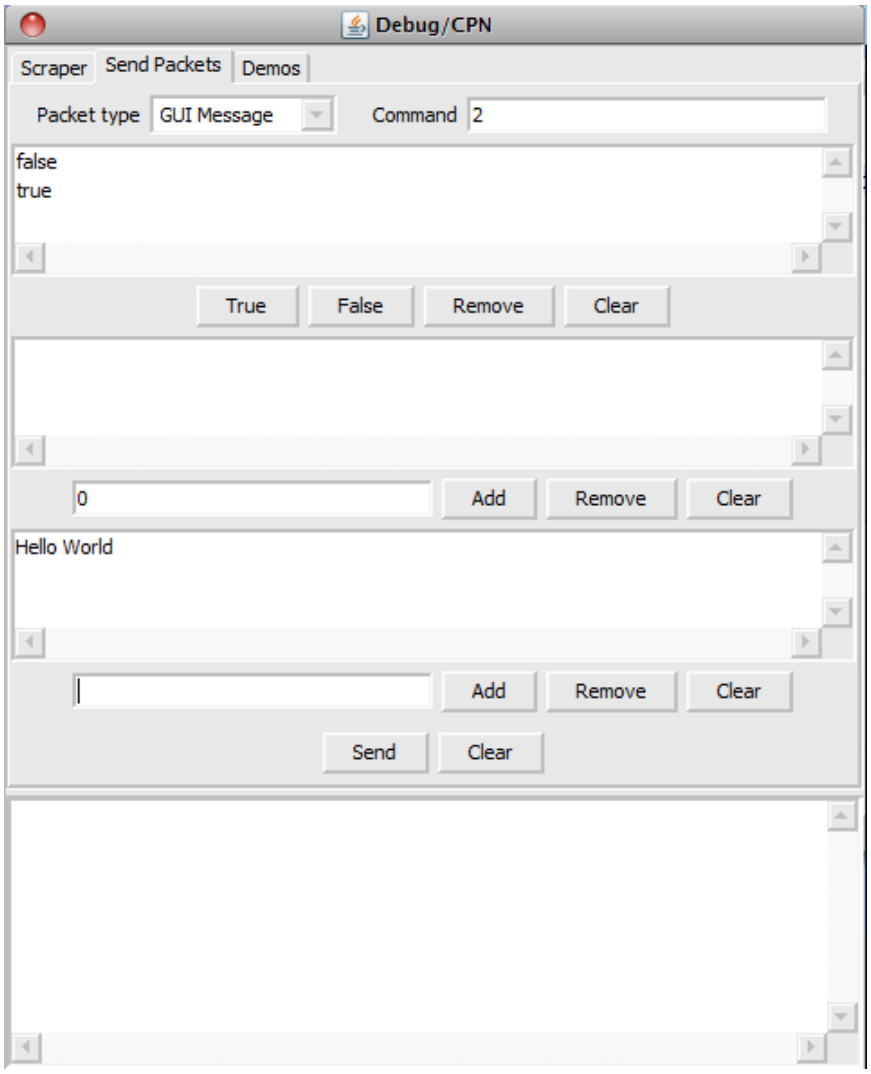

Figura C.3. Debug/CPN.

En la figura C.3 se ilustra la interfaz de la funcionalidad de envío de paquetes al simulador por parte Debug/CPN. En la sección superior se coloca el tipo de comando a utilizar y el número del comando a utilizar. Luego, la interfaz provee tres interfaces para agregar elementos a la lista de booleanos, a la lista de enteros y la lista de strings de un paquete BIS. Al final, el botón *Send* permite realizar el envío del paquete construido. En el cuadro inferior de la interfaz se provee un área de texto en la que se obtiene la información del paquete de respuesta.

Como ejemplo, enviaremos, mediante Debug/CPN, al simulador de CPN Tools el siguiente mensaje (ver figura C.4) que es un mensaje del tipo de comandos 400 (comandos para el chequeo de sintaxis, véase Tabla 2.1).

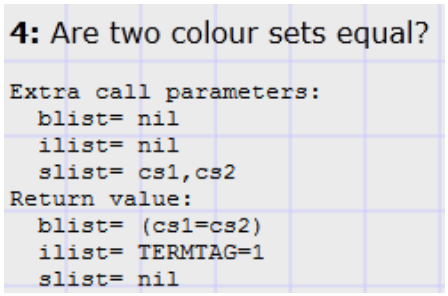

Figura C.4. Ejemplo de paquete para utilizar en Debug/CPN [19].

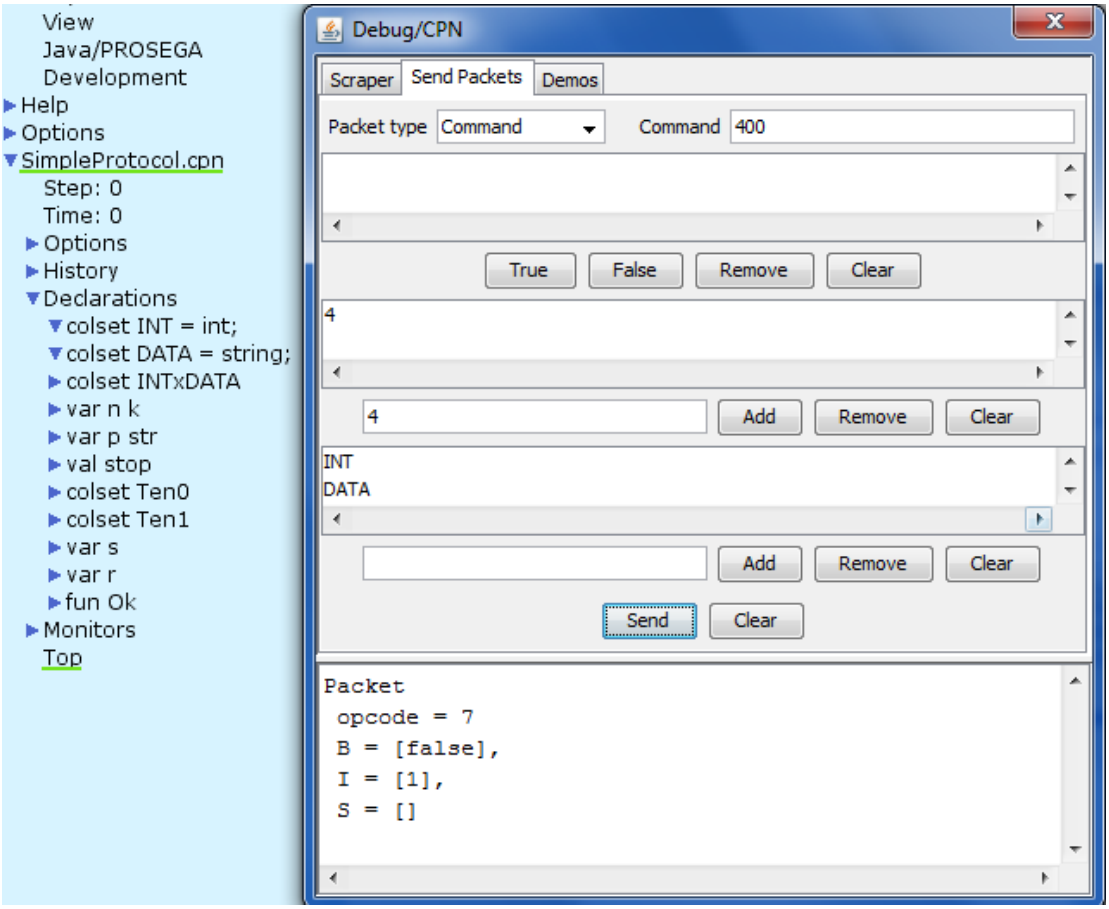

Figura C.5. Envío y recibo de un paquete BIS en CPN Tools.

El paquete a enviar escogido para ejemplo, perteneciente al grupo de paquetes relacionados con el chequeo de sintaxis de los modelos (cmd=400), se encarga de verificar si dos *color sets* son iguales. Para esto se carga y utiliza cualquier modelo CPN en el editor gráfico de CPN Tools

La figura C.5 muestra cómo se envió dicho paquete a través de Debug/CPN. El tipo de paquete seleccionado es *command*. Cuando se elige el tipo de paquete la herramienta Debug/CPN se encarga internamente de seleccionar el *opcode* adecuado. En el campo *command* se escribe el número del grupo de comandos al que pertenece el paquete a enviar (en este caso, 400).

Ahora nos encargamos de rellenar las tres listas del paquete de acuerdo a la especificación provista en la figura C.4. La lista de booleanos viajará vacía según lo indicado en la documentación, la lista de enteros solo se le es agregado el subcomando respectivo (en este caso, 4) para indicar al simulador de CPN Tools la tarea específica que se quiere realizar. Finalmente, en la lista de strings se insertarán los nombres de los dos *color sets* a comparar y luego se presiona el botón *Send*.

El área inferior de la interfaz de Debug/CPN colocará la información del paquete de respuesta que envía el simulador de CPN Tools de acuerdo a la documentación provista (figura C.4). En la lista de booleanos viaja un único booleano indicando si los dos *color sets* consultados son iguales, la lista de enteros envía la marca *TERMSTAG* indicando que se procesó exitosamente el paquete de respuesta previamente enviado. La lista de strings del paquete de respuesta no es devuelta con algún valor y esto también es esperado de acuerdo a la documentación provista para esta tarea en la figura C.4.

ANEXO

D

Código en C de la librería fsm2language

A continuación, se provee el código en C de las funciones de la librería fsm2language (véase Sección 3.6): *fsm2language* y *fsm2random*. Esta librería es utilizada por Java/PROSEGA para realizar la generación del lenguaje.

D.1 fsm2language

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lib/fsm.h"
#include "lib/stoi.h"
#include "lib/stack.h"
#include "lib/collection.h"
#define TRUE 1
#define FALSE 0

/** Check for an arc whose origin is node a and its end is node b.
In other words, check the adjacent list of the node a, in search for direct connection with b **/
char* get_arc(unsigned int a, unsigned int b){
    int found = FALSE;
    ADJACENT_NODE *p = adjacent_list_first(fsm[a].adjacent_list);
    while(p != NULL){
        NODE *q = adjacent_node_getnode(*p);
        if(q->number == b){
            found = TRUE;
            break;
        }else{
            p = adjacent_node_next(*p);
        }
    }
    return found == TRUE ? adjacent_node_getvalue(*p) : "\0" ;
}

void collection_print(COLLECTION collection){
    int i;
    for(i = 0; i < collection_size(collection); i++)
        printf("%s ",collection.container[i].pointer_value);
    printf("\n");
}
```

```
34
35  void print_paths(unsigned int start, unsigned int halt){
36      COLLECTION path;
37      collection_init(&path);
38      STACK stack;
39      stack_init(&stack);
40
41      unsigned int depth = 0;
42      unsigned int i;
43      for(i = 0; i < number_states; i++){
44          //Check adjacent nodes and put them in the stack
45          char* value = get_arc(start, i);
46          if(strcmp(value,"\0") != 0){
47              /// *** Old comparation: if(strcmp(fsm[start][i],"\0") != 0)
48              struct stack_element element;
49              element.in_state = start;
50              element.out_state = i;
51              element.level = depth;
52              element.pointer_value = value;
53              stack_push(&stack, element);
54          }
55      }
56
57      while(stack_size(stack) > 0){
58          struct stack_element x = stack_pop(&stack);
59          struct collection_element c;
60          c.start = x.in_state;
61          c.end = x.out_state;
62          c.pointer_value = x.pointer_value;
63          collection_push(&path, c);
64
65          unsigned int current_node = x.out_state;
66          if(current_node == halt){
67              collection_print(path);
68          }
69          int new_depth = FALSE;
70
71          for(i = 0; i < number_states; i++){
72              char* value = get_arc(current_node, i);
73              if(strcmp(value, "\0") != 0 && collection_contains(path, current_node, i) == FALSE){
74                  struct stack_element new_stack_element;
75                  new_stack_element.level = depth + 1;
76                  new_stack_element.in_state = current_node;
77                  new_stack_element.out_state = i;
78                  new_stack_element.pointer_value = value;
79
80                  stack_push(&stack, new_stack_element);
81                  new_depth = TRUE;
82              }
83          }
84
85          if(new_depth == FALSE){
86              struct stack_element peek_element = stack_peek(stack);
87              unsigned int previous_depth = peek_element.level;
88
89              while(collection_size(path) - 1 >= previous_depth && collection_size(path) >= 1){
90                  collection_pop(&path);
91              }
92              depth = previous_depth;
93          }else{
94              depth++;
95          }
96      }
97      collection_free(&path);
98      stack_free(&stack);
99  }
```

```
100
101 int main(int argc, char *argv[]){
102     number_states = stoi(argv[1], 1, 10);
103     number_halt_states = stoi(argv[2], 1, 10);
104     initial_state = stoi(argv[3], 1, 10);
105     input_file = fopen(argv[4],"r");
106     init_fsm();
107     load_fsm();
108
109     int i;
110     for(i = 0; i < number_halt_states; i++){
111         printf("%u -> %u\n", initial_state, halt_states[i]);
112         print_paths(initial_state, halt_states[i]);
113     }
114     printf("\n");
115     free_fsm();
116     return 0;
117 }
```

Código D.1. Código en C de la función fsm2language.

En ll. 4- 7 se agregan otras librerías (o archivos de cabecera) desarrollados por el autor del presente trabajo. En particular, se agrega el archivo *fsm* que contiene los métodos para la inicialización y carga del autómata en una representación matricial. Se agrega el archivo *stack* que contiene la estructura y métodos necesarios para el manejo de pilas. También se agrega el archivo *collection* que contiene la estructura y métodos necesarios para el manejo de una lista implementada mediante un arreglo dinámico.

En ll. 101-117 comienza la ejecución del programa, en donde se toman los parámetros de entrada (número de estados del autómata, número de estados terminales, estado inicial y el archivo que contiene los arcos y nodos terminales del autómata). En l. 106 y l. 107 se inicializa la máquina de estado finito a partir del archivo en texto plano suministrado. Luego, en ll. 109 - 114 se lleva a cabo la impresión del lenguaje imprimiendo todas las palabras que son procesadas desde el estado inicial hasta cada uno de los estados terminales. La sección que corresponde a ll. 35-99 es fiel copia del pseudocódigo desarrollado por el autor del presente trabajo (ver Sección 3.6.1) que define el algoritmo de impresión de todas las cadenas que pueden ser generadas entre un estado inicial y un estado terminal.

D.2 fsm2random

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "lib/stoi.h"
#include "lib/fsm2random/collection.h"
#include "lib/fsm.h"
#define STRING_SIZE 6
#define TRUE 1
#define FALSE 0

double halt_rate; //probabilidad de parada

int times; // despreciar el primer camino

char* get_arc(unsigned int a, unsigned int b){
    int found = FALSE;
    ADJACENT_NODE *p = adjacent_list_first(fsm[a].adjacent_list);
    while(p != NULL){
        NODE *q = adjacent_node_getnode(*p);
        if(q->number == b){
            found = TRUE;
            break;
        }else{
            p = adjacent_node_next(*p);
        }
    }
    return found == TRUE ? adjacent_node_getvalue(*p) : "\0" ;
}

double uniform(){
    return (double)rand() / (double)RAND_MAX ;
}

unsigned int next_state(unsigned int state){

    int i;
    int reachables[number_states];
    int n = 0;
    for(i = 0; i < number_states; i++){
        char* value = get_arc(state, i);
        if(strcmp(value,"\0") != 0){
            reachables[n] = i;
            n++;
        }
    }

    if(n == 0){
        return -1;
    }else{
        int u = round(uniform() * (n-1));
        unsigned int k = reachables[u];
        return k;
    }
}

int is_halt_state(unsigned int n){
    int found = FALSE;
    int i;
    for(i = 0; i < number_halt_states; i++){
        if(n == halt_states[i]){
            found = TRUE;
            break;
        }
    }

    return found;
}
```

```
70
71  void print_random_word(unsigned int initial_state){
72
73      COLLECTION path;
74      collection_init(&path);
75
76      unsigned int s = initial_state;
77      unsigned int n;
78
79      while(TRUE){
80          n = next_state(s);
81          if(n == -1) break;
82
83          struct collection_element c;
84          strcpy(c.data, get_arc(s,n));
85          collection_push(&path, c);
86
87          s = n;
88
89          if(is_halt_state(n) == TRUE){
90              if(halt_rate > uniform()){
91                  break;
92              }
93          }
94      }
95
96      if(times > 0){
97          collection_print(path);
98      }
99      collection_free(&path);
100 }
101
102 int main(int argc, char *argv[]){
103     number_states = stoi(argv[1], 1, 10);
104     number_halt_states = stoi(argv[2], 1, 10);
105     initial_state = stoi(argv[3], 1, 10);
106     input_file = fopen(argv[4],"r");
107     int h = stoi(argv[5], 1, 10); // 0 < h <= 100
108     halt_rate = (double) h / (double) 100.0;
109
110     init_fsm();
111     load_fsm();
112
113     srand(time(0));
114
115     print_random_word(initial_state);
116
117     free_fsm();
118
119     return 0;
120 }
```

Código D.2. Código en C de la función fsm2random.

En ll. 1-11 se cargan las librerías (o archivos de cabecera) utilizados por este programa. En particular, se importa la librería *stack* para de la estructura pila y el archivo *fsm* que define la estructura y métodos para el manejo de las Máquinas de Estado Finito.

En ll. 32-34 se implementa la función *uniform* que se encarga de retornar un número aleatorio que sigue una distribución uniforme entre 0 y 1.

El programa comienza en la sección correspondiente a ll. 102-120 en donde se extraen los parámetros de entrada para el programa, el número de estados, numero de estados terminales, el estado inicial, el archivo que define la estructura e información del autómata, y el número *p* (*halt-rate*) que define la probabilidad de parada. En l. 115 se realiza la llamada a la función *print_random_word* ( ) que se encargará de realizar un recorrido aleatorio al autómata desde el estado inicial hasta algunos de los estados terminales.

En ll. 71-100 comienza el algoritmo de generación de una palabra aleatoria del lenguaje reconocido por el autómata basado en el pseudocódigo explicado en la Sección 3.6.3.

En ll. 36-56 se implementa una rutina que se encarga de, a partir de un nodo inicial, seleccionar un nodo destino, de manera aleatoria a través de la distribución uniforme en donde cada nodo destino tendrá igual chance de ser escogido. Si no es posible seleccionar un nodo destino (por ejemplo, debido a que el nodo inicial no posee arcos de salida) la rutina devolverá -1.

ANEXO

E

Implementación de *get_node* / *get_arc* en Java/PROSEGA

En esta sección se expone la implementación hecha en Java para las funciones *get_node* y *get_arc* provistas en la documentación de CPN Tools [19] para extraer los nodos y arcos provenientes en los paquetes de respuesta del protocolo BIS.

Para obtener los arcos de un Grafo de Estado en específico se realiza una llamada al simulador de CPN Tools (véase Sección 3.3.2). Luego, cuando el simulador responde a Java/PROSEGA este envía en el paquete de respuesta la información de cada uno de los arcos del grafo y la información de los nodos que componen el arco (nodo inicio, nodo destino). Sin embargo, extraer esta información del paquete de respuesta no es algo trivial.

La documentación provista [19] establece un conjunto de algoritmos (*get_node* y *get_arc*) que manipula el paquete BIS de respuesta para obtener los arcos y nodos del Grafo de Estado. A continuación, la figura E.1 muestra dichos algoritmos contenidos en la documentación de CPN Tools. El código E.1 muestra luego como se implementaron este par de algoritmos en Java/PROSEGA para obtener así los arcos del Grafo de manera que posteriormente Java/PROSEGA pudiera realizar el proceso de minimización del Grafo de Estados a una Máquina de Estado Finito.

The following pseudo-code describes how to retrieve information about a node from boolean, integer, and string lists.

```
get_node()
  current-node = pop(ilist)
  node-exists = pop(blist)
  if node-exists
  then processed = pop(blist)
       fully-processed = pop(blist)
       x-coordinate = pop(ilist)
       y-coordinate = pop(ilist)
       num-predecessors = pop(ilist)
       num-successors = pop(ilist)
       node-descriptor = pop(slist)
  else (nothing more for current node)
```

The following pseudo-code describes how to retrieve information about an arc from boolean, integer, and string lists.

```
get_arc()
  current-arc = pop(ilist)
  arc-exists = pop(blist)
  if arc-exists
  then num-of-bendpoints = pop(ilist)
       repeat num-of-bendpoints times
          x-coordinate = pop(ilist)
          y-coordinate = pop(ilist)
       srcnode = pop(ilist)
       dstnode = pop(ilist)
       arc-descriptor = pop(slist)
  else (nothing more for current arc)
```

Figura E.1. Documentación de los métodos *get_arc* y *get_node* [19].

```java
private Arc getArc(Packet p) {
    Arc arc = new Arc();

    arc.setNumber(p.getInteger());
    arc.setExists(p.getBoolean());

    if(arc.getExists()){
        arc.setBendpoints(p.getInteger());
        for(int i = 0; i < arc.getBendpoints(); i++){
            int xCoordinate = p.getInteger();
            int yCoordinate = p.getInteger();
        }
        arc.setSourceNode(p.getInteger());
        arc.setDestNode(p.getInteger());
        arc.setDescriptor(p.getString());
    }

    return arc;
}

private Node getNode(Packet p) {
    Node node = new Node();
    node.setNumber(p.getInteger());

    boolean nodeExists = p.getBoolean();
    if(nodeExists){
        boolean processed = p.getBoolean();
        boolean fullyProcessed = p.getBoolean();
        int xCoordinate = p.getInteger();
        int yCoordinate = p.getInteger();
        node.setNumPredecessors(p.getInteger());
        node.setNumSuccessors(p.getInteger());
        node.setDescriptor(p.getString());
        return node;
    }

    return null;
}
```

Código E.1. Implementación de los métodos *get_arc* y *get_node*.

ANEXO

F

Archivos utilizados en las pruebas Java/PROSEGA

F.1. Archivo de identificadores para el Grafo de Estado del modelo CPN de la especificación del servicio

A continuación, se muestra el contenido del archivo de identificadores para el Grafo de Estado del modelo CPN de la especificación del servicio. Este archivo es utilizado en la interfaz de asignación de identificadores del proceso de minimización de Java/PROSEGA. El archivo permite realizar una asociación entre las transiciones del modelo CPN y un identificador numérico. Cada línea del archivo se compone de: el nombre de la página, el nombre de la transición y el identificador numérico que la asociará (separado por espacios).

```
ChangeConnection MACChgConnReq 7
ChangeConnection MACChgConnReq2 7
ChangeConnection MACChgConnInd 8
ChangeConnection MACChgConnRsp 9
ChangeConnection MACChgConnRsp2 9
ChangeConnection MACChgConnCf 10
ChangeConnection MACChgConnCf2 10
ChangeConnection ConnfrRejected 10
ChangeConnection MACChgConnRechCf 10
CreatConnection MACCrtConnReq 1
CreatConnection MACCrtConnReq2 1
CreatConnection MACCrtConnInd 2
CreatConnection MACCrtConnRsp 3
```

CreatConnection MACCrtConnRsp2 3
CreatConnection MACCrtConnCf 4
CreatConnection MACCrtConnCf2 5
CreatConnection ConnfrRejected 5
CreatConnection MACCrtConnCfRech 6
TerminateConnection MACTerConnReq 11
TerminateConnection MACTerConnReq2 11
TerminateConnection MACTerConnInd 12
TerminateConnection MACTerConnRsp 13
TerminateConnection MACTerConnRsp2 13
TerminateConnection MACTerConnCf 14
TerminateConnection MACTerConnCf2 15
TerminateConnection ConnfrRejected 15
TerminateConnection MACTerConnCfRech 16

F.2 Archivo generado por Java/PROSEGA del Grafo de Estado utilizando el formato de Máquinas de Estado Finito

A continuación, se presenta el archivo que genera Java/PROSEGA en donde se transforma el Grafo de Estado a una representación de autómata que maneja la librería OpenFST. La asociación del nombre de las transiciones con identificadores númericos se logra mediante el archivo presentado en la Sección F.1.

1 2 CreatConnection'MACCrtConnReq
1 3 CreatConnection'MACCrtConnReq2
2 4 CreatConnection'MACCrtConnInd
2 1 CreatConnection'ConnfrRejected
3 1 CreatConnection'MACCrtConnCf2
4 5 CreatConnection'MACCrtConnRsp
4 6 CreatConnection'MACCrtConnRsp2
5 7 CreatConnection'MACCrtConnCf
6 8 CreatConnection'MACCrtConnCfRech
7 9 TerminateConnection'MACTerConnReq
7 10 TerminateConnection'MACTerConnReq2
7 11 ChangeConnection'MACChgConnReq
7 12 ChangeConnection'MACChgConnReq2
7 13 CreatConnection'MACCrtConnACKInd
8 14 CreatConnection'MACCrtConnReq

8 1 CreatConnection'MACCrtConnACKInd
8 15 CreatConnection'MACCrtConnReq2
9 7 TerminateConnection'ConnfrRejected
9 16 CreatConnection'MACCrtConnACKInd
10 7 TerminateConnection'MACTerConnCf2
10 17 CreatConnection'MACCrtConnACKInd
11 7 ChangeConnection'ConnfrRejected
11 18 CreatConnection'MACCrtConnACKInd
12 7 ChangeConnection'MACChgConnCf2
12 19 CreatConnection'MACCrtConnACKInd
13 16 TerminateConnection'MACTerConnReq
13 17 TerminateConnection'MACTerConnReq2
13 18 ChangeConnection'MACChgConnReq
13 19 ChangeConnection'MACChgConnReq2
14 2 CreatConnection'MACCrtConnACKInd
14 8 CreatConnection'ConnfrRejected
15 3 CreatConnection'MACCrtConnACKInd
15 8 CreatConnection'MACCrtConnCf2
16 20 TerminateConnection'MACTerConnInd
16 13 TerminateConnection'ConnfrRejected
17 13 TerminateConnection'MACTerConnCf2
18 21 ChangeConnection'MACChgConnInd
18 13 ChangeConnection'ConnfrRejected
19 13 ChangeConnection'MACChgConnCf2
20 22 TerminateConnection'MACTerConnRsp
20 23 TerminateConnection'MACTerConnRsp2
21 24 ChangeConnection'MACChgConnRsp
21 25 ChangeConnection'MACChgConnRsp2
22 1 TerminateConnection'MACTerConnCf
23 13 TerminateConnection'MACTerConnCfRech
24 26 ChangeConnection'MACChgConnCf
25 27 ChangeConnection'MACChgConnRechCf
26 28 TerminateConnection'MACTerConnReq
26 29 TerminateConnection'MACTerConnReq2
26 30 ChangeConnection'MACChgConnReq
26 31 ChangeConnection'MACChgConnACKInd
26 32 ChangeConnection'MACChgConnReq2
27 33 TerminateConnection'MACTerConnReq
27 34 TerminateConnection'MACTerConnReq2
27 35 ChangeConnection'MACChgConnReq

27 13 ChangeConnection'MACChgConnACKInd
27 36 ChangeConnection'MACChgConnReq2
28 26 TerminateConnection'ConnfrRejected
28 37 ChangeConnection'MACChgConnACKInd
29 26 TerminateConnection'MACTerConnCf2
29 38 ChangeConnection'MACChgConnACKInd
30 39 ChangeConnection'MACChgConnACKInd
30 26 ChangeConnection'ConnfrRejected
31 37 TerminateConnection'MACTerConnReq
31 38 TerminateConnection'MACTerConnReq2
31 39 ChangeConnection'MACChgConnReq
31 40 ChangeConnection'MACChgConnReq2
32 40 ChangeConnection'MACChgConnACKInd
32 26 ChangeConnection'MACChgConnCf2
33 27 TerminateConnection'ConnfrRejected
33 16 ChangeConnection'MACChgConnACKInd
34 27 TerminateConnection'MACTerConnCf2
34 17 ChangeConnection'MACChgConnACKInd
35 18 ChangeConnection'MACChgConnACKInd
35 27 ChangeConnection'ConnfrRejected
36 19 ChangeConnection'MACChgConnACKInd
36 27 ChangeConnection'MACChgConnCf2
37 41 TerminateConnection'MACTerConnInd
37 31 TerminateConnection'ConnfrRejected
38 31 TerminateConnection'MACTerConnCf2
39 42 ChangeConnection'MACChgConnInd
39 31 ChangeConnection'ConnfrRejected
40 31 ChangeConnection'MACChgConnCf2
41 43 TerminateConnection'MACTerConnRsp
41 44 TerminateConnection'MACTerConnRsp2
42 45 ChangeConnection'MACChgConnRsp
42 46 ChangeConnection'MACChgConnRsp2
43 1 TerminateConnection'MACTerConnCf
44 31 TerminateConnection'MACTerConnCfRech
45 47 ChangeConnection'MACChgConnCf
46 48 ChangeConnection'MACChgConnRechCf
47 49 TerminateConnection'MACTerConnReq
47 50 TerminateConnection'MACTerConnReq2
47 51 ChangeConnection'MACChgConnReq
47 31 ChangeConnection'MACChgConnACKInd

47 52 ChangeConnection'MACChgConnReq2
48 53 TerminateConnection'MACTerConnReq
48 54 TerminateConnection'MACTerConnReq2
48 55 ChangeConnection'MACChgConnReq
48 31 ChangeConnection'MACChgConnACKInd
48 56 ChangeConnection'MACChgConnReq2
49 47 TerminateConnection'ConnfrRejected
49 37 ChangeConnection'MACChgConnACKInd
50 47 TerminateConnection'MACTerConnCf2
50 38 ChangeConnection'MACChgConnACKInd
51 39 ChangeConnection'MACChgConnACKInd
51 47 ChangeConnection'ConnfrRejected
52 40 ChangeConnection'MACChgConnACKInd
52 47 ChangeConnection'MACChgConnCf2
53 48 TerminateConnection'ConnfrRejected
53 37 ChangeConnection'MACChgConnACKInd
54 48 TerminateConnection'MACTerConnCf2
54 38 ChangeConnection'MACChgConnACKInd
55 39 ChangeConnection'MACChgConnACKInd
55 48 ChangeConnection'ConnfrRejected
56 40 ChangeConnection'MACChgConnACKInd
56 48 ChangeConnection'MACChgConnCf2
1
7
8
13
26
27
31
47
48

ANEXO

G

Trabajo Futuro: Redes de Petri Coloreadas como Servicios (CPNaaS)

Este anexo tiene como propósito introducir a un posible futuro trabajo dentro de esta línea de investigación que está siendo desarrollado en la actualidad por Westergaard M. [66]. Este trabajo puede dar inicio a que futuros proyectos de aplicaciones web utilicen como base el uso de las Redes de Petri Coloreadas.

Este trabajo se basa en el desarrollo de un API (o middleware) llamado CPNaaS (*Coloured Petri Net as a Service*) [66] con la finalidad de manipular las Redes de Petri Coloreadas en un paradigma de Servicios Web a través de internet.

Se provee una aplicación cliente (móvil, web, etc.) que se conecte a través de esta API a través del protocolo HTTP siguiendo el enfoque REST (solicitudes de recursos a través de métodos HTTP GET, POST, DELETE, etc.) y esta se encargará de redirigir las solicitudes (a través del protocolo BIS de CPN Tools) al core del API quien no es más que el mismo simulador de CPN Tools.

La arquitectura prevista también incluye otros componentes (Bases de datos para persistencia, etc.). Esto podría dar entonces entrada a realizar proyectos de aplicaciones con tecnología internet que utilicen como base modelos CPN. La figura G.1 muestra la arquitectura inicial planteada por Westergaard M. [66].

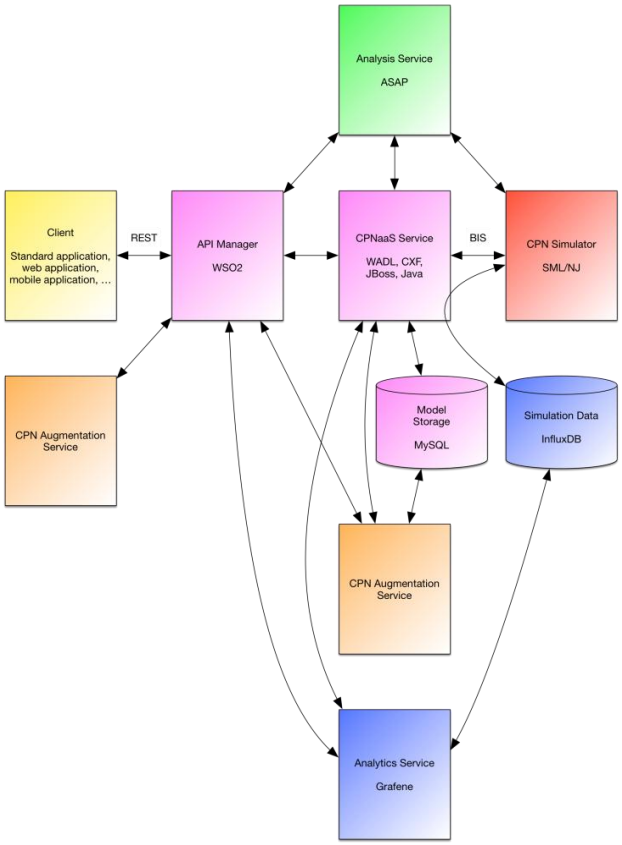

Figura G.1. Arquitectura inicial planteada para CPNaaS [66].

La tabla G.2 muestra algunos de los ejemplos que se muestran en [66] acerca de cómo podrían ser accedidos los recursos de Redes de Petri Coloreadas a través de URLs siguiendo el enfoque REST.

Tabla G.1. Ejemplos de posibles llamadas con CPNaaS [66].

Solicitud de Recurso	Explicación
POST /nets	Crea una nueva red
GET /nets/1	Obtiene información acerca de la red 1
POST /nets/1/pages	Crear una página en la red 1
GET /nets/1/pages/3/places/3	Obtener información de la plaza 3 en la página 3 de la red 1.
POST /nets/1/pages/2/places	Crea una plaza en la página 2 de la red 1