UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ANÁLISIS Y VISUALIZACIÓN DE ÁRBOLES DE PROCESOS CONFIGURABLES

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

ELISA PAZ KAUFFMANN FIGUEROA

PROFESOR GUÍA:
JOCELYN SIMMONDS WAGEMANN

MIEMBROS DE LA COMISIÓN:
ALEXANDRE BERGEL
PATRICIO INOSTROZA FAJARDIN

SANTIAGO DE CHILE
2018

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: ELISA PAZ KAUFFMANN FIGUEROA
FECHA: 2018
PROFESOR GUÍA: JOCELYN SIMMONDS WAGEMANN

ANÁLISIS Y VISUALIZACIÓN DE ÁRBOLES DE PROCESOS CONFIGURABLES

Hoy en día la disponibilidad masiva de información permite a las organizaciones tomar mejores decisiones en base a registros de los procesos que ejecuta. Este es el contexto en el que se desarrolla la minería de procesos. A través de técnicas que incorporan tanto minería de datos como aprendizaje computacional, se puede extraer conocimiento a partir de registros de eventos que comúnmente están disponibles en los sistemas de información de las organizaciones.

El equipo de GEMS del Departamento de Ciencias de la Computación de la Universidad de Chile actualmente se encuentra investigando nuevas técnicas de minería de procesos que permitan descubrir familias de procesos similares. Incorporando la noción de variabilidad, los procesos obtenidos como resultados de sus experimentos se pueden modelar usando árboles de procesos configurables (CPT), los cuales agrupan en una sola representación diversas variantes de un solo proceso.

Sin embargo surgen varios problemas al momento de analizar, manipular y visualizar dichos modelos. Varias revisiones han sido hechas por el equipo de GEMS y aún no se ha encontrado herramientas que provean estas funcionalidades de manera adecuada. Incluso el framework ProM6, una de las más populares herramientas de apoyo a la ejecución de algoritmos de descubrimiento de procesos, carece de las funcionalidades requeridas por el equipo para llevar a cabo su proyecto.

En este Trabajo de Título se propone diseñar y desarrollar una herramienta que permita al equipo de GEMS agilizar el proceso de análisis y publicación de los resultados de su trabajo. Para ello se propone implementar módulos que provean estas funcionalidades, incorporándolas a ProM6.

En específico, la solución implementada incorporó nuevos plug-ins de ProM6 que permiten al usuario importar al framework archivos con la codificación en texto de un CPT, y visualizar la representación gráfica del modelo. A su vez, dichos plug-ins permiten extraer las variantes de un CPT de manera individual, para luego exportar sus representaciones gráficas a archivos de formato `svg`. Esto proporciona una manera de post-procesar las figuras usando algún editor de imágenes vectoriales apropiado, facilitando el proceso de visualización y publicación de los resultados obtenidos.

La solución fue validada estudiando casos de uso aplicados sobre CPTs sintéticos y reales generados por el equipo de GEMS. Se constata que la solución implementada constituye un aporte al trabajo realizado por el equipo de GEMS, según los requerimientos planteados, disminuyendo sustancialmente la cantidad de tiempo destinado al análisis, visualización y publicación de resultados.

Tabla de Contenido

Índice de Ilustraciones

Índice de Códigos Fuente

Índice de Tablas

Capítulo 1

Introducción

1.1. Antecedentes

Diferentes organizaciones ejecutan diversos procesos para, por ejemplo, manufacturar sus productos, ofrecer determinados servicios, realizar compras y ventas de artículos, etc. Para estas organizaciones, es importante definir dichos procesos formalmente, de manera que se faciliten no solo su ejecución, si no que también las operaciones administrativas. Esto puede incluir la reutilización de dichos procesos, replicación de estrategias exitosas, y mejora en base a resultados anteriores, lo que puede repercutir fuertemente en los costos y desempeño de la organización. Desde una mirada más expandida, al definir estos procesos de negocio formalmente, se pueden hacer comparaciones entre organizaciones que proveen servicios similares, lo que permite analizar diferencias, aprender buenas prácticas y mejorar la toma de decisiones.

Existen variados lenguajes y notaciones para definir procesos formalmente, como Petri nets [3], el estándar BPMN (Business Process Model Notation) [4], álgebras de procesos, como CSP [5], entre otros. Sin embargo, la tarea de formalizar una definición de un proceso es difícil y propensa a errores, dada la naturaleza "intangible" [6] de éstos. Desde luego, para hacer un buen análisis de un determinado proceso, éste se debe modelar a partir de lo que realmente se hace en la organización. Es por ello que muchas organizaciones mantienen registros de algún tipo que indican, por ejemplo, el nombre y/o descripción de las actividades que se llevan a cabo, los recursos que cada una necesita, marcas de tiempo y duración, personas involucradas, lugar en el que se llevan a cabo, etc. [1].

Levantar la especificación de un proceso de forma manual es un trabajo que llevan a cabo los ingenieros de procesos, generalmente utilizando alguna de las notaciones anteriormente mencionadas, y usando como base registros de eventos y actividades reales de la organización. En las últimas décadas, con la explosión de datos, el escenario es distinto, y es en este contexto donde emerge el área de investigación de minería de procesos. Nace como el puente entre el análisis e ingeniería de procesos tradicional, con la minería de datos y aprendizaje computacional. Su objetivo es descubrir, monitorear y mejorar procesos a través de la extracción de conocimiento a partir de registros de eventos [7] (de ahora en adelante, *logs*). La

minería de procesos busca llegar a comprender diferentes perspectivas del proceso, no solo la del control de flujo, sino que también la perspectiva organizacional, la de desempeño, la de datos, entre otras.

Se han propuesto numerosas y variadas técnicas para el descubrimiento de procesos en base a logs, usando distintas notaciones, medidas de verificación, criterios, etc. Por ejemplo, existe el algoritmo *alpha* [8], que usa Petri nets para representar los modelos que descubre. También existen algoritmos genéticos, como el ETM (*Evolutionary Tree Miner*) [7], y algoritmos que descubren modelos de procesos con variantes, como el algoritmo ETMc [1].

A grandes rasgos, las técnicas de minería de procesos se pueden clasificar en dos categorías importantes: aquéllas que descubren un solo modelo de procesos, y aquéllas que descubren familias de procesos, es decir, un conjunto de modelos que no difieren tanto entre sí y que comparten un tronco común. Una manera de representar variantes de procesos similares en un solo modelo es a través de árboles de proceso configurables (CPT) [1], los cuales serán el tema central en el presente trabajo. En términos generales, los CPTs se construyen como un árbol, donde las hojas representan actividades de un proceso, y los nodos internos, operaciones sobre ellas. La variabilidad del proceso modelado se visualiza en forma de etiquetas que pueden decorar individualmente los nodos, especificando sus posibles cambios en cada variante del proceso.

En comparación a las técnicas de descubrimiento de un solo proceso, existen muy pocas aproximaciones al descubrimiento de variantes, a pesar de que la presencia de variabilidad en los procesos de organizaciones es evidente. Por lo mismo, son pocas las herramientas en la industria que permiten el análisis y visualización de familias de procesos. Dichas herramientas son sumamente importantes a la hora de poder exponer el modelo que se está investigando y es por esto que su escasez presenta un problema a la hora de presentar resultados a la organización o publicarlos a la comunidad científica. La idea es poder facilitar el trabajo de la persona que está realizando la especificación del proceso, pues si éste no puede entender ni comunicar lo que se extrae de los registros entonces estos datos son inútiles.

1.2. Motivación

El equipo de GEMS [9] del Departamento de Ciencias de la Computación de la Universidad de Chile actualmente se encuentra investigando sobre modelos de procesos configurables. En particular está desarrollando algoritmos de descubrimiento de familias de procesos usando representaciones Petri net [10]. Sin embargo, últimamente esta notación ha complejizado considerablemente el trabajo, dado que no tiene una forma de representar variabilidad en los procesos. Es por esto que ha optado por usar árboles de procesos configurables para sus trabajos más recientes.

Varias revisiones han sido hechas por el equipo de GEMS, y no se han encontrado herramientas que permitan visualizar y manipular de manera adecuada diagramas de árboles de procesos configurables y sus variantes de manera individual. En particular ProM6 [11], uno de los frameworks más populares utilizados para realizar descubrimiento de procesos, carece

de plug-ins que incorporen de manera integral funcionalidades y operaciones sobre árboles configurables de procesos. Es deseable para el equipo de GEMS tener una herramienta que genere visualizaciones de los resultados que obtiene de los algoritmos que está desarrollando, y poder exportarlas a algún formato conveniente. Esto es necesario, pues los árboles de procesos configurables y las variantes que se obtienen a partir de logs reales, pueden llegar a tener una cantidad inmanejable de nodos y niveles, dificultando su representación de forma manual y su comprensión. Por lo tanto también se vuelve difícil la validación de los procesos descubiertos por parte de los ingenieros de procesos. Dichas visualizaciones deberían ser, a su vez, fácilmente exportables a algún formato cómodo para editar, como `svg`, para agilizar el proceso de presentación o publicación de resultados. Esto, pues actualmente las figuras se están generando manualmente utilizando complejas bibliotecas de LaTeX, cuyo uso no es fácil a la hora de producir este tipo de visualizaciones, teniendo además una sintaxis poco legible y no reutilizable. En la figura 1.1 se observa un ejemplo de representación gráfica de un árbol de procesos configurable generada por el equipo de GEMS. Las actividades del proceso modelado están indicadas con letras del alfabeto, y algunas de ellas presentan cuatro variantes, lo cual se indica en las respectivas etiquetas de color gris. Un modelo de este tamaño podría fácilmente corresponder a solo una parte de un árbol más grande. Sin embargo, el tiempo utilizado en escribir las líneas de código que producen esta representación es no despreciable, y genera retrasos no contemplados por los integrantes del equipo de GEMS, afectando directamente el desarrollo del proyecto de investigación.

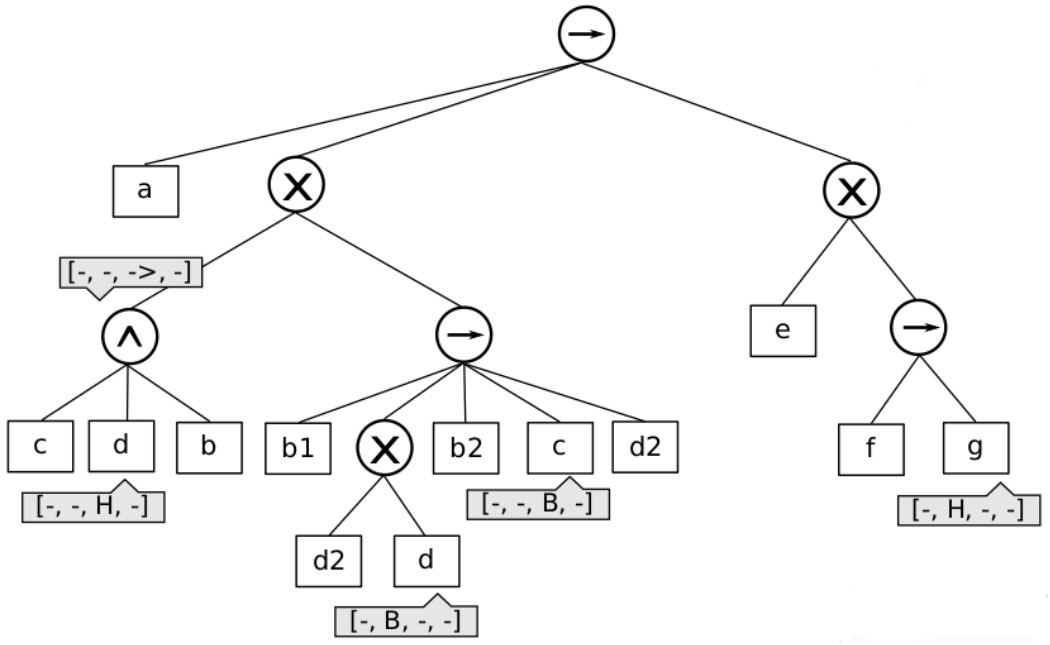


Figura 1.1: Ejemplo de un árbol de procesos configurable utilizado por el equipo de GEMS para mostrar sus resultados.

1.3. Objetivos

A continuación se presenta el objetivo general y los objetivos específicos de este trabajo de título.

1.3.1. Objetivo general

Diseñar e implementar un paquete para el framework ProM6 que genere y manipule diagramas de árboles de procesos configurables a partir de una representación en texto previamente generada por otro plug-in ya existente.

1.3.2. Objetivos específicos

1. Diseñar e implementar un módulo que procese la representación en texto de un CPT y los parsee a una estructura de árbol de procesos configurable interna de ProM6. Esta representación ya se encuentra definida y es entregada como output de otro plug-in ya existente en el framework.

2. Construir una representación visual general de un árbol de procesos configurable con sus etiquetas, manteniendo un layout general que sea visualmente cómodo frente árboles de procesos configurables con demasiados nodos, etiquetas y niveles. En particular, los elementos del árbol no deben traslaparse a un nivel que no permita la lectura.

3. Permitir seleccionar y visualizar de manera individual las variantes posibles de un árbol de procesos configurable.

4. Permitir al usuario exportar la figura del árbol de proceso configurable a formato `svg`, de tal forma que pueda editarse posteriormente usando algún editor de imágenes vectoriales, como Inkscape [12].

1.4. Metodología

La metodología de trabajo se dividió en cuatro fases, las cuales se describen a continuación.

- Investigación: Esta parte del desarrollo de este trabajo se dedicó al estudio del funcionamiento del framework ProM6, sus componentes principales y formas de compilarlo localmente. También se investigó sobre el desarrollo de plug-ins y paquetes, y cuáles son los ya existentes que pueden ser de interés y que pueden ser reutilizados o extendidos. Además se estudió sobre las posibles librerías de Java que faciliten el trabajo a desarrollar, sobretodo el de visualización de grafos.

- Análisis: En esta etapa se decidió cuáles fueron las librerías y paquetes de ProM6 que se pueden usar y/o se extender. También se tomaron las decisiones sobre diseño y estructura de la implementación, incluyendo la estimación de tiempos a destinar a cada tarea.

- Desarrollo: Teniendo en cuenta los estándares del desarrollo de paquetes de ProM6 y haciendo uso de desarrollo incremental por módulos, en esta etapa se construyó el software propuesto como solución que se decidió en las etapas anteriores de la metodología.

- Validación: A lo largo del desarrollo se estuvo validando con el equipo de GEMS que el software cumpliera con las expectativas. Para probar el funcionamiento se usaron archivos de representación en texto de los CPT sintéticos.

1.5. Estructura de la memoria

El resto del documento se estructura como sigue: en el capítulo 2 se aborda con más detalle la teoría sobre árboles de procesos configurables y se realiza una descripción del uso y desarrollo de plug-ins de ProM6. Por su parte, en el capítulo 3 se describe la situación actual en la que se manifiestan los problemas que afectan al equipo de GEMS, mientras que en el capítulo 4 se describen el análisis y diseño de dos prototipos de solución. En los capítulos 5 y 6 se explica cómo fue realizada la implementación y la validación de la solución final respectivamente. Finalmente se expone la conclusión del trabajo realizado y se proponen posibles mejoras y lineamientos en el desarrollo de trabajos futuros.

Capítulo 2

Marco Teórico

En este capítulo se exponen los elementos relevantes para la completa comprensión de este documento. En las primeras dos secciones se explica con mayor detalle lo que es un modelo de procesos y su importancia en el área de la minería de procesos. En particular se describirá la representación de un proceso a través de los árboles de procesos configurables. En la tercera sección se entregan mayores detalles sobre ProM6, un framework utilizado para llevar a cabo técnicas de minería de procesos. Finalmente, en la cuarta sección se describe la metodología estándar para crear y extender funcionalidades de esta plataforma, mediante el desarrollo de plug-ins.

2.1. Árboles de procesos

En el contexto de la minería de procesos, una de las actividades centrales es el descubrimiento de un modelo general y repetible de un proceso a partir de observaciones reales proporcionadas en forma de logs de eventos. Para representar, manipular y analizar estos modelos, existen variadas notaciones. Cada uno de estos lenguajes proporciona la noción de elección, es decir, muestran distintas formas de llevar a cabo el proceso que modelan dependiendo del contexto en el que éste se realiza y las decisiones que se toman durante su ejecución. Específicamente, estos lenguajes de notación deben abarcar al menos tres distintas perspectivas de un proceso [6]: La del flujo de actividades y operaciones, la de los actores, y la del contexto y variables que afectan al proceso.

Una posible forma de representar modelos es a través de árboles de procesos [7]. Para describir un proceso desde la perspectiva del flujo de actividades, los árboles de procesos se construyen como un árbol, es decir, un grafo acíclico. En éstos, las hojas corresponden a actividades en el proceso y los nodos internos a operaciones de control de flujo entre actividades. Los nodos internos indican instrucciones o decisiones que deben tomarse con respecto a sus nodos hijos, los cuales pueden ser a su vez otras operaciones. En la figura 2.1 se observa un ejemplo de un árbol de procesos desde la perspectiva del flujo de actividades. La raíz del árbol corresponde a la operación de paralelismo de actividades ($\wedge$), e indica que ambos subárboles que nacen de éste deben ejecutarse obligatoriamente y al mismo tiempo. Existen

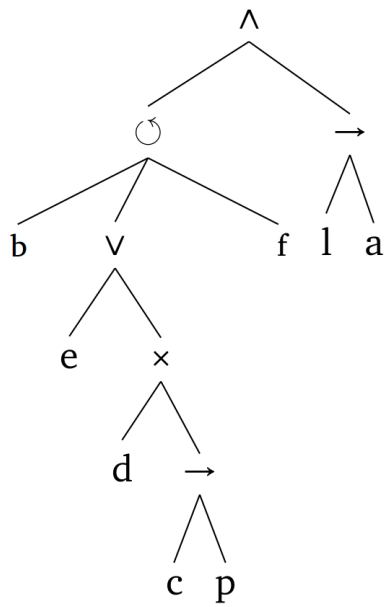


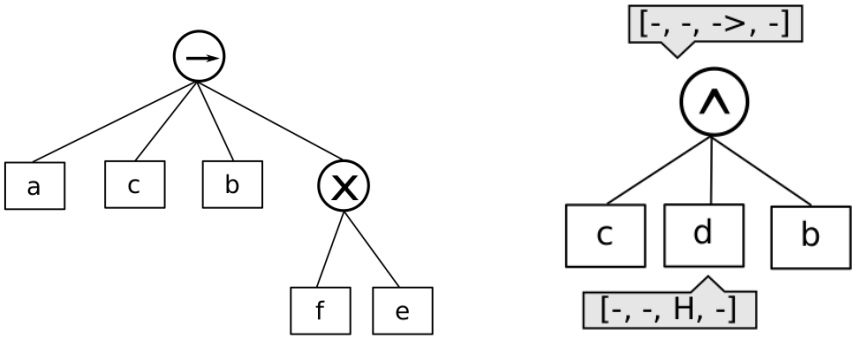
Figura 2.1: Ejemplo de un árbol de procesos.

Símbolo	Operación sobre subárboles
$\wedge$	Paralelismo
$\vee$	Disyunción no exclusiva
$\rightarrow$	Secuencia ordenada (de izquierda a derecha)
$\circlearrowleft$	Iteración. Solo puede tener tres hijos
$\times$	Disyunción exclusiva

Tabla 2.1: Notación de las operaciones en un árbol de procesos.

otros tipos de operadores, como el de iteración o *loop* ($\circlearrowleft$), que representa una repetición de actividades que termina cuando cierta condición se cumple. El operador loop debe tener exactamente tres hijos en el árbol. Los primeros dos, de izquierda a derecha, representan dos actividades o subprocesos que se ejecutan sucesivamente una determinada cantidad de veces ('do' y 'redo'). Mientras que el tercero, es decir, el de más a la derecha, indica la actividad final o de salida del ciclo ('exit'). En el ejemplo de la figura 2.1 se puede observar que el hijo izquierdo de la raíz del árbol corresponde a un operador de iteración. Éste se interpreta como una secuencia repetida en donde la actividad b y el subproceso que empieza con $\vee$ se ejecutan uno después del otro una cantidad de veces hasta que alguna condición de término aparece. En tal caso, se ejecuta la actividad f, y esa rama del subproceso termina. El resto de los operadores posibles en un árbol de proceso se describen en la tabla 2.1.

Los arcos y los nodos de un árbol de proceso pueden poseer información adicional sobre el proceso que modelan, como los recursos utilizados para alguna actividad o transición, los actores involucrados y otras variables de contexto. Por ejemplo, la información que indica cuándo una iteración de actividades debe terminar puede estar contenida en el arco entre el operador $\circlearrowleft$ y el tercer hijo ('exit'). Para efectos de este trabajo de título, el desarrollo estuvo



(a) Ejemplo de un árbol de proceso.

(b) Ejemplo de un árbol de proceso configurable.

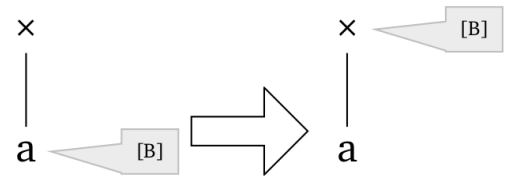
Figura 2.2: Ejemplos de un árbol de proceso no configurable y otro que sí lo es. Este último muestra cuatro variantes.

centrado solamente en la perspectiva del flujo de actividades que un árbol de procesos puede modelar, es decir, no se consideró la existencia de información adicional o metadata en los arcos y nodos de la notación.

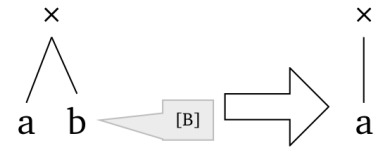
2.2. Árboles de procesos configurables

Un árbol de procesos configurable (CPT) [1] es un árbol de procesos que representa un conjunto de modelos que no difieren tanto entre sí y que comparten un tronco común. Es decir, engloba en una sola representación una familia de modelos de procesos. A cada nodo, ya sea una actividad o una operación, se le puede asignar una o más opciones de configuración, que pueden ser: no configurado (−), bloqueado (`B`), escondido (`H`). Por su parte, en el caso particular de un operador, éste se puede configurar para transformarse en otro operador. En la figura 2.2 se puede comparar un árbol de proceso simple y otro configurable con cuatro variantes. La diferencia está en las etiquetas que acompañan a algunos nodos, representando los cuatro posibles tipos que puede tomar cada nodo dependiendo de la configuración que se aplica. Cada nodo configurable tomará el primer tipo del arreglo en su etiqueta cuando se aplique la primera configuración al árbol; el segundo valor del arreglo para la segunda configuración; y así sucesivamente.

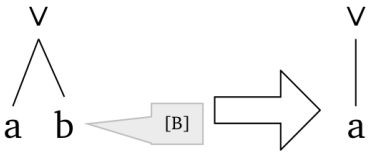
Un nodo etiquetado con '−' implica que su tipo no cambiará para la configuración respectiva. Por su parte, un nodo que cambia a tipo bloqueado indica que el camino representado por ese nodo ya no puede ser tomado, y por tanto el subproceso que le sigue ya no puede ser ejecutado. Por otro lado, un nodo de tipo escondido significa que puede ser ignorado, pues no es relevante para el proceso, y la mayoría de las veces estos nodos ni siquiera representan alguna acción (no se ejecuta nada). Una vez que cambian al estado escondido, llevarán por nombre el símbolo τ. Estos nodos de tipo escondido, también llamados silenciosos, se incorporan a la notación de un CPT para que sea posible una futura traducción de este modelo a otros lenguajes, como BPMN.



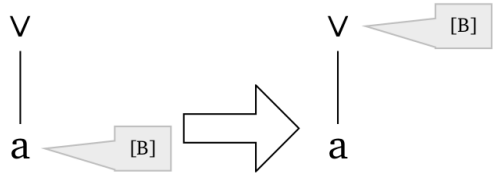
(a) Bloqueo del último nodo hijo de un operador ×.



(b) Bloqueo de un nodo hijo de un operador ×.



(c) Bloqueo de un nodo hijo de un operador ∨.



(d) Bloqueo del último nodo hijo de un operador ∨.

Figura 2.3: Reglas de bloqueo para operadores × y ∨.

Cuando se aplica una configuración de bloqueo a algún nodo, distintas situaciones pueden ocurrir [1]:

1. En el caso en que el nodo padre sea un operador × o ∨:

   (a) Si el nodo bloqueado es el único hijo que queda, entonces el padre mismo se bloquea (figuras 2.3a y 2.3d).

   (b) Si no, el nodo bloqueado se quita del padre (figuras 2.3b y 2.3c).

2. En el caso en que el nodo padre sea un operador → o ∧, ese operador también se bloquea, dado que obliga a la ejecución de todos sus hijos (figuras 2.4a y 2.4b).

3. En el caso en que el nodo padre sea un operador ↻:

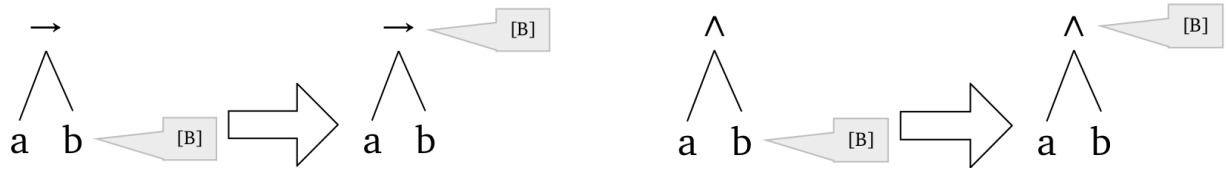
   (a) Si el nodo bloqueado es el primero ('do') o el tercero ('exit'), entonces el padre también se bloquea (figuras 2.5a y 2.5c).

   (b) Si el nodo bloqueado es el segundo ('redo'), entonces el padre es reemplazado por un operador → con el primer y tercer nodo como hijos (figura 2.5b).

4. En el caso en que el nodo bloqueado es la raíz del árbol de proceso configurable, entonces ésta es reemplazada por un nodo escondido (τ), pues el árbol de proceso no permite ningún comportamiento (figura 2.6).

Dado que el bloqueo de un nodo puede tener un efecto en nodos de niveles de más arriba en el árbol de proceso, las configuraciones deben ser aplicadas recursivamente. Una vez que el bloqueo ha sido efectuado es posible, si se desea, hacer una reducción del árbol resultante borrando todos los caminos que han quedado bloqueados.

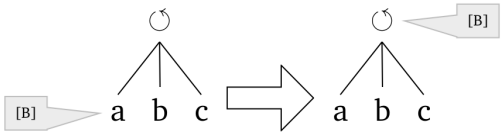
Un operador puede ser configurado transformándolo en otro operador de menor jerarquía, según las reglas mostradas en la figura 2.7. 'Rebajar', o hacer *downgrade* a un nodo, restringe



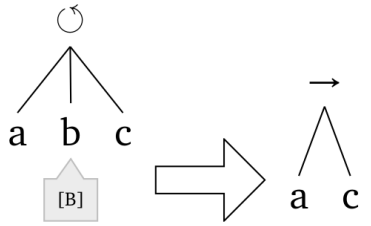
(a) Bloqueo de un nodo hijo de un →.

(b) Bloqueo de un nodo hijo de un ∧.

Figura 2.4: Reglas de bloqueo para operadores → y ∧.



(a) Bloqueo del nodo hijo 'do' de un ↻.



(b) Bloqueo del nodo hijo 'redo' de un ↻.

(c) Bloqueo del nodo hijo 'exit' de un ↻.

Figura 2.5: Reglas de bloqueo para operadores ↻

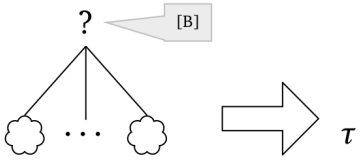


Figura 2.6: Bloqueo de la raíz de un árbol de proceso configurable.

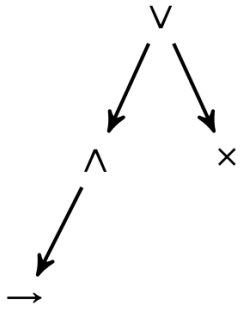


Figura 2.7: Jerarquía de opciones de *downgrade* de operadores.

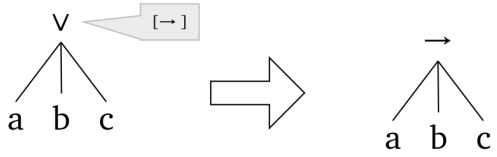


Figura 2.8: Ejemplo de *downgrade* de un operador $\wedge$ a un $\rightarrow$.

su comportamiento a un subconjunto de posibles comportamientos iniciales. Un operador de disyunción ($\vee$) puede ser rebajado a uno de los siguientes: $\wedge$, $\times$ o $\rightarrow$. Un operador $\wedge$ puede ser rebajado a $\rightarrow$. Finalmente un operador $\circlearrowleft$ puede ser rebajado a uno de $\rightarrow$, cuando su segundo hijo es bloqueado, como se explicó anteriormente. Un ejemplo de *downgrade* de un nodo se observa en la figura 2.8, donde un nodo $\vee$ es rebajado a un $\rightarrow$.

Las opciones de configuración se muestran solo para nodos donde al menos hay una opción que no es '$-$'. Por ejemplo, solo dos nodos (operador $\wedge$ y actividad d) del árbol de la figura 2.2b tienen etiquetas de configuración, mostradas en cajas (usualmente de color gris) conectadas a su nodo correspondiente. Aquí, cada etiqueta de configuración posee cuatro opciones, una para cada una de las cuatro variantes del modelo. El nodo $\wedge$ ha sido etiquetado con $[-, -, \rightarrow, -]$, lo que significa que quedará tal cual para las configuraciones 1, 2 y 4, pero es rebajado a $\rightarrow$ en la tercera configuración. A su vez, la actividad d, quedará escondida en la tercera variante de este CPT. El resto de las configuraciones, es decir, 1, 2 y 4, son exactamente iguales. Esto tiene sentido puesto que este árbol corresponde a una sección de otro más grande.

En las figuras 2.9 y 2.10 se muestra otro ejemplo, un poco más grande, de un árbol de proceso configurable, junto a sus posibles configuraciones. Es importante hacer notar que una vez aplicadas las configuraciones, si se lo desea, se puede llevar a cabo una etapa final de reducción. Esto implica, tanto el borrado de los nodos que han sido bloqueados, como los nodos escondidos que producen redundancias en la estructura del CPT.

En la figura 2.10a se observan la primera y cuarta configuraciones obtenidas del CPT de la figura 2.9, las cuales no presentan cambios pues sus variantes son solamente '$-$'. Son idénticas, lo cual tiene sentido puesto que este CPT corresponde a una sección de un CPT más grande. Por su parte, en la figura 2.10b se observa la segunda variante. Nótese que en esta última ha sido bloqueada y borrada la actividad d, que era hija del operador $\times$. Dado que una vez aplicada la configuración el nodo $\times$ pasa a tener un solo hijo, es redundante mostrar este arco. Esto es así porque la disyunción exclusiva obliga a escoger un y solo un camino, que en este caso sería siempre el de la actividad d2. Es por esta razón que d2 sube de nivel, reemplazando al nodo $\times$ y quedando como hijo directo de $\rightarrow$, al mismo nivel que $b1$, $b2$, c y otra instancia de d2.

A su vez, en esa misma configuración (figura 2.10b) se puede observar la reducción del

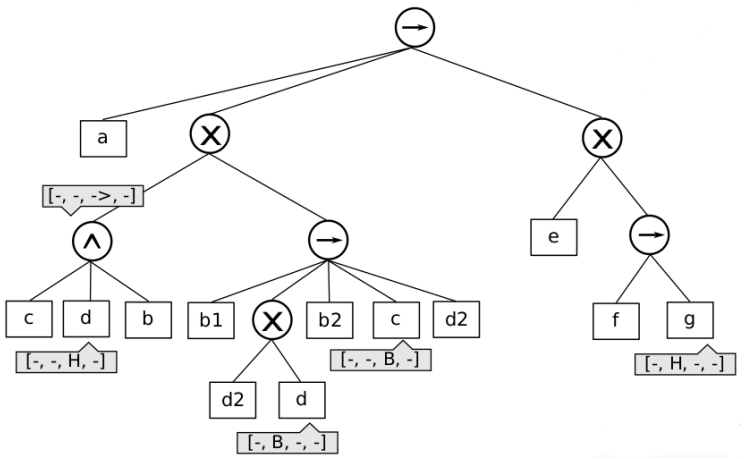


Figura 2.9: Ejemplo de árbol de procesos configurable con cuatro variantes.

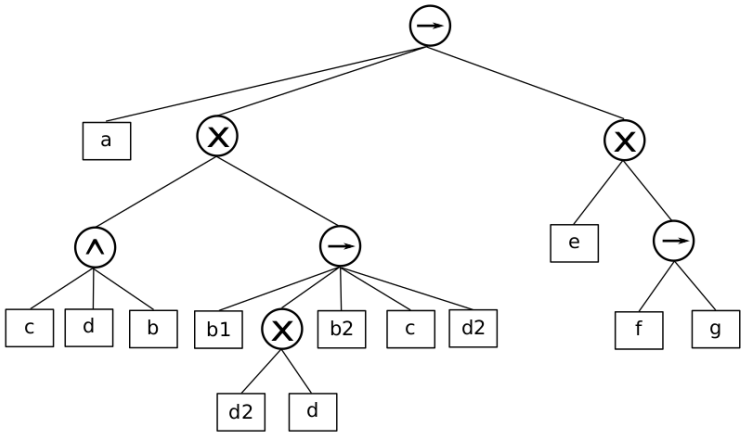
nodo $\rightarrow$ que tenía como hijos las actividades f y g. Dado que en esta configuración la actividad g aparece escondida, f puede tomar el lugar de su padre, pues una secuencia donde solo participa una sola actividad es equivalente a ejecutar esa actividad por sí sola. Así, la actividad f puede subir de nivel, reemplazando al operador $\rightarrow$, quedando como hijo del operador $\times$ y al mismo nivel que la actividad e.

Finalmente, en la figura 2.10c se muestra la tercera configuración del CPT de la figura 2.9. Es interesante notar que en esta variante, toda la rama de nodos hijos del operador $\rightarrow$, es decir, la rama que contiene las actividades $b1$, d2, d, $b2$ y c, ha sido borrada. Esto es así pues al propagar las reglas de bloqueo, la actividad c provoca que el operador $\rightarrow$ se bloquee. Por otra parte, el operador $\wedge$ ha cambiado por un $\rightarrow$, y dado que ya no tiene más hermanos, puede reducirse subiéndolo de nivel.

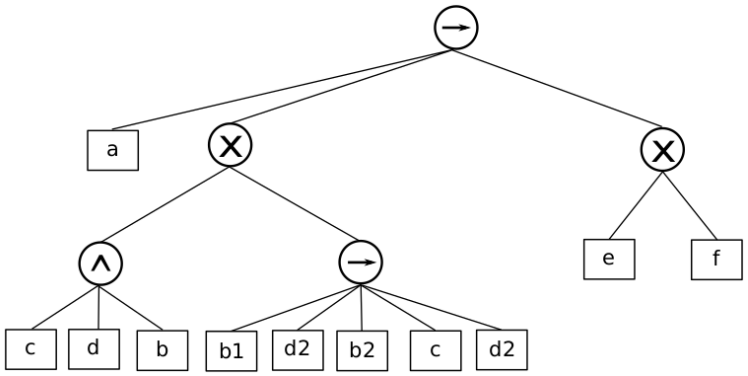
2.3. ProM6

Dentro del área de la minería de procesos ProM6 es una de las herramientas más populares. ProM6 es un framework open source, gratuito y extensible que ofrece una amplia variedad de técnicas de minería de procesos en forma de plug-ins. Soporta algoritmos de descubrimiento de procesos, visualización de modelos precalculados, edición de diagramas, animaciones, entre muchas otras funciones. Este framework además es portable, dado que está implementado en el lenguaje de programación Java.

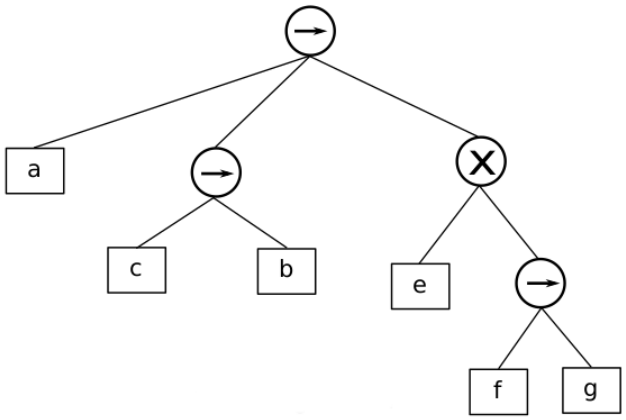
La interfaz de usuario de ProM6 posee tres vistas principales, las cuales se pueden seleccionar desde las pestañas en la parte superior, como se observa en la figura en el apéndice B.1. La de la izquierda corresponde al *workspace* (ver figura 2.11), donde se almacenan y manipulan todos los recursos disponibles para uso y visualización en el framework. Estos recursos son objetos (objetos Java) que representan los modelos o estructuras de datos que participan en un determinado algoritmo. Éstos pueden ser importados desde un archivo local o pueden haber sido producidos por la ejecución de algún algoritmo. En la misma figura 2.11 se pueden ver en el listado de recursos algunas estructuras de datos disponibles como Petri



(a) Primera y cuarta variantes.



(b) Segunda variante.



(c) Tercera variante.

Figura 2.10: Variantes del árbol de proceso configurable de la figura 2.9.

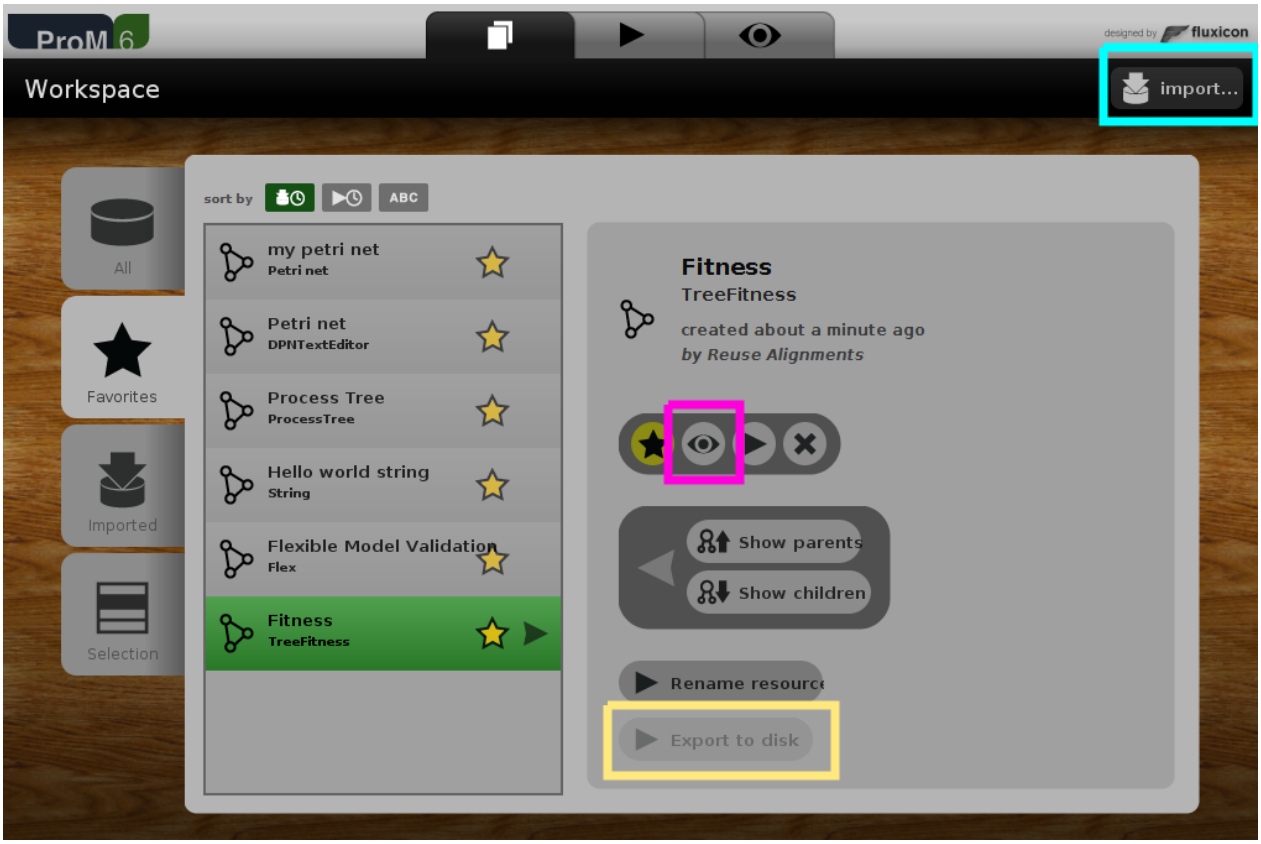


Figura 2.11: Vista que muestra el listado de recursos disponibles para ejecutar plug-ins en ProM6. Marcados en color cyan, magenta y amarillo se observan los botones para importar, visualizar y exportar, respectivamente.

nets, árboles de procesos e incluso un String de Java.

En la pestaña del medio se accede a la vista de acciones (figura 2.12). En ésta se puede ver un listado de todos los plug-ins disponibles en el framework. En el panel de la izquierda se pueden escoger recursos desde el workspace para usarlos como input de algún plug-in que está en el listado, y a la derecha se observan los tipos de output de dicho plug-in. Una acción del listado aparece de color verde cuando tiene los inputs necesarios para ejecutarse. Si no, aparecerá de color amarillo. Para iniciar la ejecución del plug-in se debe clickear el botón verde de 'Start'.

Finalmente la pestaña de la derecha corresponde a la vista de visualización de alguno de los recursos disponibles. En el ejemplo de la figura 2.13 se escogió una Petri net del workspace para visualizar. Algunos plug-ins que tienen objetos como output activan inmediatamente esta pestaña para visualizar los resultados, si es que el objeto tiene alguna forma de visualización implementada.

Una característica muy importante de ProM6 es que está construido de manera tal que permite y fomenta a investigadores y desarrolladores a contribuir con nuevas funcionalidades mediante paquetes. Se han publicado algunos tutoriales oficiales [13] [14] [15] e incluso se llevó a cabo en 2012 un taller presencial [16] de desarrollo de plug-ins de ProM6, con el objetivo

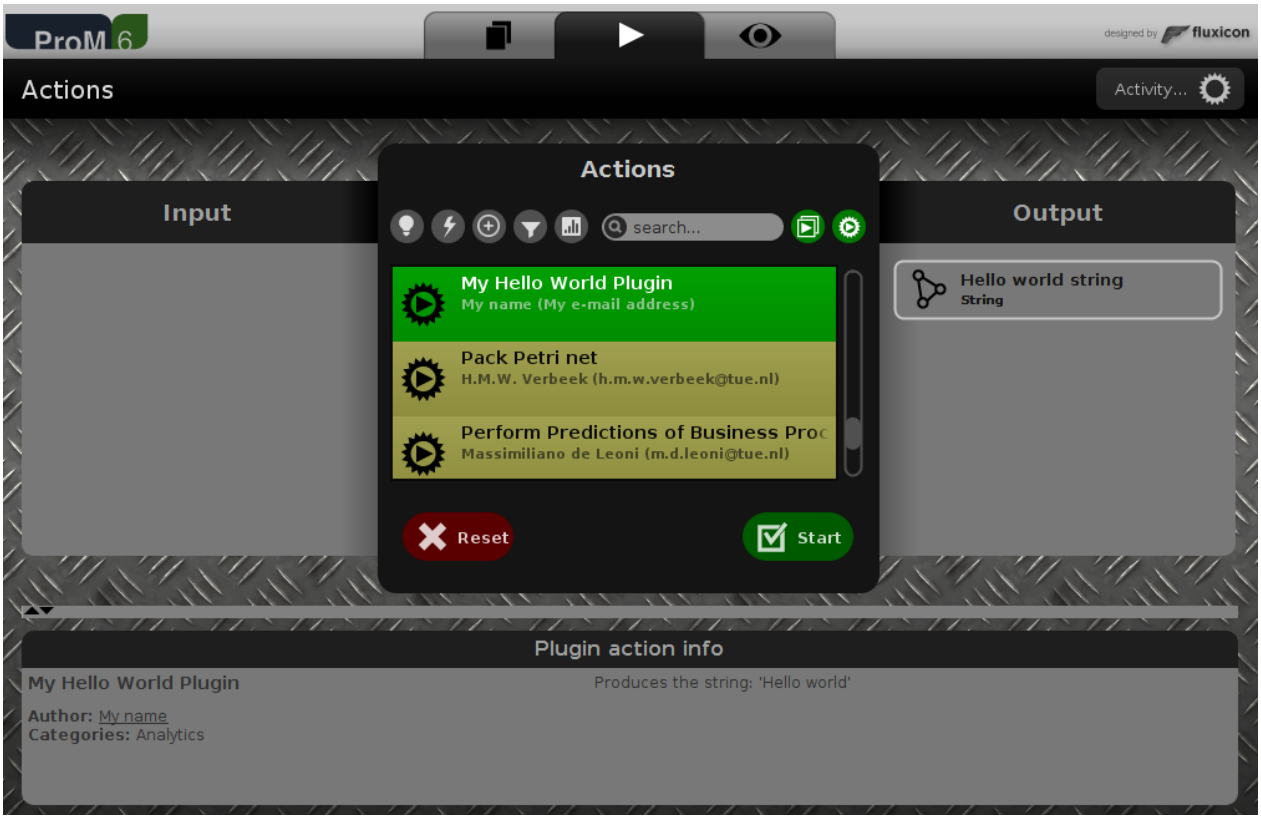


Figura 2.12: Vista de acciones, con el listado de plug-ins disponibles. Al seleccionar uno, aparece su descripción, inputs necesarios y tipo de outputs

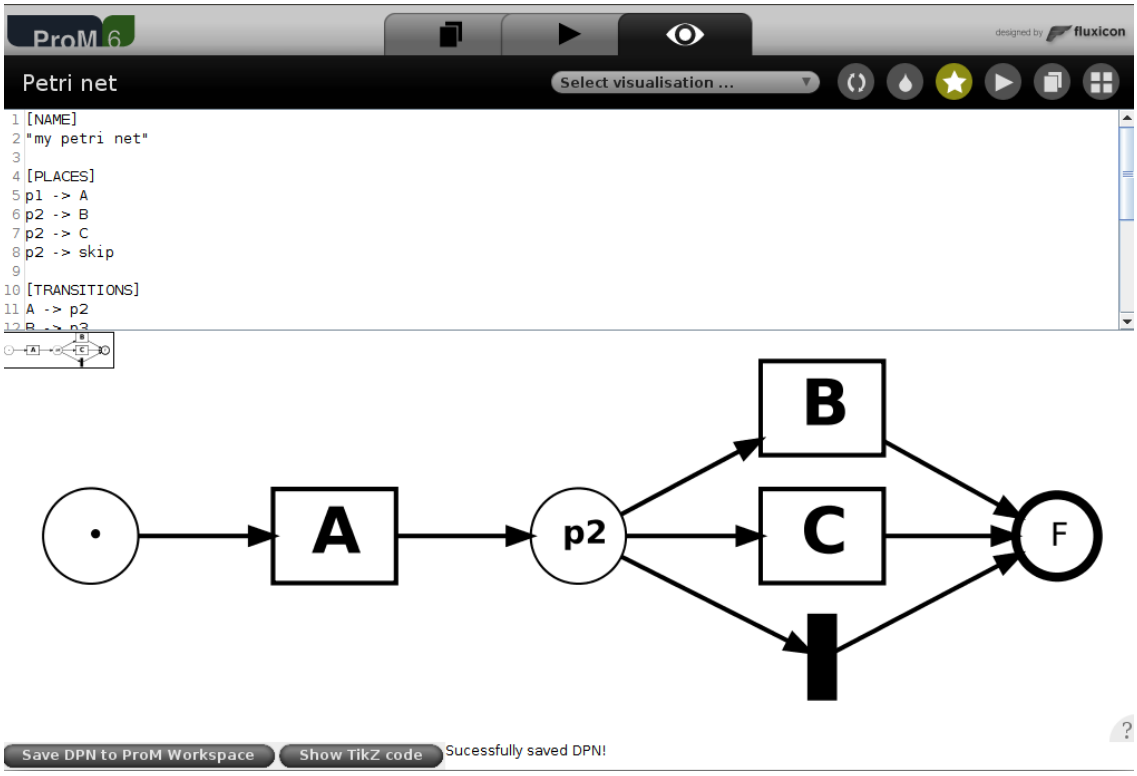


Figura 2.13: Vista de la pestaña de visualización de ProM6, con una representación visual de una red de Petri de ejemplo.

de ampliar la comunidad que contribuye a este framework.

2.3.1. Paquetes de ProM6

Un paquete de ProM6 se puede entender como el trabajo de una persona particular, que aporta una o varias funcionalidades nuevas al framework. Cualquier desarrollador que quiera implementar algo, y que no forme parte del equipo interno de ProM6, debe desarrollar un paquete nuevo, y será el responsable de que éste funcione, compile, y de su publicación, en caso de que así lo desee. Un paquete de ProM6 representa una funcionalidad del framework, en el más amplio sentido de la palabra. A través de uno o más plug-ins definidos en el paquete, éste puede implementar, por ejemplo, un algoritmo de minería de datos, y/o construir y visualizar alguna estructura de datos abstracta. De esta manera, toda la lógica del área de la minería de datos queda definida en los paquetes, quedando aislada del funcionamiento interno del framework.

Los componentes más importantes de un paquete de ProM6 son los plug-ins. Un plug-in es un módulo del paquete que ejecuta un algoritmo de minería de procesos, usando todos los recursos, clases, configuraciones, implementaciones de algoritmos, etc., que están disponibles y definidos dentro del paquete. Un paquete puede implementar uno o más plug-ins, y son estos "la cara visible" del paquete ante el usuario, quien los puede ver listados en la interfaz gráfica (figura 2.12) y ejecutarlos.

A veces ciertos componentes tienen código que puede ser reutilizado por otros, por ejemplo, las estructuras de datos. En tal caso, ProM6 permite que los paquetes puedan tener dependencias entre sí. Por ejemplo, existe el paquete `PetriNets`, el cual es utilizado por muchos otros paquetes, abstrayéndose así de la implementación del modelo de una red de Petri, y centrándose solamente en el desarrollo de nuevos algoritmos que las utilicen.

2.3.2. Desarrollo de plug-ins

Hay variados componentes que pueden conformar un paquete de ProM6, los cuales no son más que clases típicas de Java. ProM6 ofrece una plantilla oficial [17] para el desarrollo de nuevos paquetes. Este template incorpora Apache Ivy [18] para el manejo de dependecias y provee un launcher para ejecutar ProM6 con su interfaz gráfica. Con esto se puede probar de manera local el paquete que se está desarrollando.

La implementación de un plug-in de ProM6 queda definida en un solo método dentro de una clase Java. Dentro de este método, se hace uso de los modelos, los algoritmos y otras clases, y se integra en una sola funcionalidad. En general el plug-in deriva la ejecución de algoritmos u otros procesos a métodos de otras clases, y solo se encarga de la lógica general, la comunicación entre elementos del paquete en el que se encuentra (o de otras dependencias), y la interacción directa con el usuario, por ejemplo, a través de la interfaz gráfica.

Los plug-ins se deben definir usando variadas anotaciones específicas de Java provistas por

el framework. Éstas ofrecen una manera de describir cada plug-in en muchos ámbitos, por ejemplo, lo que hace, de qué tipo es, qué parámetros recibe y retorna, quién es su autor, etc. A continuación se describen las anotaciones más relevantes para el desarrollo de este trabajo.

- `@Plugin:` denota que determinado método constituye un plug-in para el framework. Los atributos de esta anotación permiten explicitar el nombre del plug-in, las etiquetas de los parámetros de entrada y salida, el tipo de los objetos que retorna, entre otras cosas. Es muy importante explicitar de la forma más específica los tipos de retorno para poder hacer que los plug-ins interactúen entre sí. Cualquier inconsistencia entre lo declarado en la anotación y lo que el plug-in realmente retorna, levantará errores en tiempo de ejecución. El nombre y las etiquetas de los parámetros de entrada y salida son textos que además se mostrarán en la interfaz gráfica de usuario (ver figura 2.12).

- `@UITopiaVariant:` detalla información adicional que se mostrará en la interfaz de usuario de ProM, incluyendo, por ejemplo, información sobre el autor del plug-in, como su nombre, su afiliación y su correo.

  En el snippet de código que se encuentra en el apéndice A.1 se muestra un ejemplo de plug-in básico utilizando las notaciones `@Plugin` y `UITopiaVariant`. El plug-in se llama "My Hello World Plugin" y retorna el string "Hello World". En la figura 2.12 se muestra cómo se ve dicho plug-in en la lista de plug-ins disponibles de la interfaz gráfica de ProM6, junto con su nombre, el nombre del autor y la afiliación. Esta información también es desplegada en la parte inferior cuando el plug-in se encuentra seleccionado, como lo está en esta figura. En la parte derecha se observa el tipo (`String`) del output y la etiqueta (`Hello world string`) de lo que retorna el plug-in. En la parte izquierda se observa que no se lista ningún input, pues este plug-in no lo necesita (línea 4 del snippet de código A.1). Nótese que al definir la función `helloWorld` (línea 15), a pesar de que el plug-in no necesita parámetros, el método de todas formas recibe uno de tipo `PluginContext`. Éste es obligatorio para todos los métodos que definan cualquier plug-in en ProM6, pues es el objeto que provee de información adicional a todos los plug-ins sobre el contexto y estado del framework en tiempo de ejecución.

- `@Visualizer:` no tiene atributos de anotación. Implementa una visualización de alguna estructura de datos del framework. Como parámetros, debe recibir un `PluginContext`, y otro objeto Java, el cual va a ser visualizado. Retorna un solo objeto de tipo `JComponent`, perteneciente a la librería Java Swing, y que será desplegado en el panel de visualización. Este plug-in no aparece en el listado de acciones del framework, pues no se usa directamente. El plug-in visualizador se ejecuta de dos maneras: automáticamente, después de que un plug-in termina retorna un objeto como output, éste se visualizará en la pestaña de visualización. Si no, la visualización de determinado objeto se puede ejecutar manualmente clickeando el botón marcado en la figura 2.11 con color magenta, en la vista del workspace.

- `@PluginVariant:` la anotación `@Plugin` también se puede colocar al comienzo de la definición de una clase para definir varios métodos que constituyan variantes de un solo plug-in. En tal caso, la anotación `@PluginVariant` especifica que un método representa una variante de un conjunto de plug-ins que hacen lo mismo. Se suele utilizar cuando se hace sobrecarga de un plug-in, es decir, cuando se quiere permitir un mismo com-

portamiento frente a diferente cantidad y/o tipo de parámetros. Un ejemplo típico es un plug-in que puede aceptar un archivo directamente como parámetro y una variante que, a través de una ventana de diálogo, le solicita al usuario cargar el archivo. La anotación debe definir los parámetros requeridos, que deben ser un subconjunto de lo definido por la anotación `@Plugin` de la clase.

- `@UIImportPlugin` **y** `@UIExportPlugin`**:** ambas anotaciones indican que los plug-ins son de importación o exportación de archivos locales y objetos hacia y desde el framework. Los atributos que conforman ambas anotaciones indican la descripción del plug-in y las extensiones soportadas. Al igual que un plug-in visualizador, los plug-ins de importación y exportación no aparecen en el listado de acciones disponibles. Estos plug-ins son invocados al presionar los botones de importación o exportación en la vista del workspace, como se muestra en la figura 2.11, marcados en cyan y amarillo respectivamente.

Capítulo 3

Situación Actual

En este momento el equipo de GEMS se encuentra investigando técnicas de descubrimiento de variabilidad de procesos en base a logs. El objetivo es lograr incorporar información adicional sobre el contexto de un proceso para obtener modelos más simples considerando la variabilidad y sin perder precisión.

Para ello, GEMS está utilizando árboles de proceso configurables para representar los resultados de los algoritmos de descubrimiento, y ProM6 para llevar a cabo los experimentos. En específico, para algunos de sus experimentos se está utilizando el plug-in "Mine a Configured Process Tree with ETMc" del paquete `EvolutionaryTreeMiner` [2], y obtener un árbol de proceso configurable a partir de logs. Sin embargo no existe hasta ahora una forma adecuada de representar un árbol de proceso configurable en ProM6. Actualmente, en ProM6 sólo existe el paquete `ProcessTree` [19] para representar, analizar, visualizar y manipular árboles de proceso, pero no para CPTs, lo cual es insuficiente al momento de representar variabilidad de procesos.

Por el momento, los plug-ins de `EvolutionaryTreeMiner` que descubren árboles de proceso configurables retornan un solo output de tipo `ProcessTree` que representa solamente la estructura del árbol, pero sin la visualización de sus posibles configuraciones. Aún así, esta información sí es retornada al usuario, pero solo se imprime en la consola, y usando una notación específica. Por lo tanto los resultados no son evidentes en una primera instancia dado que la representación en texto no permite obtener conclusiones de manera comprensible y rápida, a diferencia de visualizaciones como las que se han mostrado en la sección 2.2.

Por otra parte, una vez interpretados los resultados, es muy importante para el equipo de GEMS tener una manera de visualizarlos al momento de publicarlos y difundirlos. Por ahora se está utilizando el paquete TikZ para generar gráficos directamente en un documento LaTeX, sin embargo se hace cada vez más necesario alguna herramienta más conveniente para editar imágenes vectoriales que facilite este trabajo.

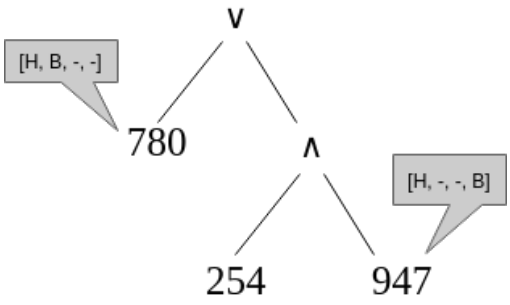


Figura 3.1: Ejemplo de un pequeño CPT con cuatro posibles configuraciones.

3.1. Representación de CPTs en texto

El paquete `EvolutionaryTreeMiner` utiliza una notación en texto para representar árboles de proceso configurables. Ésta consiste en una secuencia de caracteres dividida en dos grupos: El primero describe la estructura del árbol, es decir, los nodos y arcos del grafo, mientras que el segundo representa las configuraciones del árbol.

El primer grupo de la representación en texto de un CPT consiste en una estructura anidada de nodos de la forma `nombre_del_nodo(hijo_1, hijo_2, ..., hijo_n)`, donde los hijos se ordenan de acuerdo a su aparición de izquierda a derecha en el árbol. Primero se lee el nombre del nodo, el cual puede ser un operador o una actividad, y luego aparecerán sus hijos encerrados entre paréntesis '`()`'. Si el nodo no posee hijos, entonces es una hoja del árbol, por lo tanto no se agregarán paréntesis, y a su nombre se le antepondrá el texto '`LEAF:`'. Obsérvese el CPT de la figura 3.1 y su correspondiente representación en texto en el código fuente 3.1. La raíz del árbol corresponde a un operador $\vee$, el cual transcribe como '`OR`', y posee dos hijos. El de la izquierda es una hoja, y por tanto es una actividad, denotada por un número (780), luego en la representación en texto aparecerá escrita como '`LEAF:780`'. Por su parte, el hijo de la derecha es un nodo interno, el operador $\wedge$, el cual se transcribe como '`AND`'. En la tabla 3.1 se muestran las reglas de transcripción de los nodos de un CPT a su representación en texto.

En la representación en texto de este árbol, la hoja `LEAF:780` se escribe como el primer hijo de la raíz, pues es el nodo hijo que aparece más a la izquierda. En seguida aparece transcrito el siguiente hijo a la derecha, el operador $\wedge$. De manera análoga, y utilizando una estructura anidada, se escriben los hijos del operador $\wedge$, primero el hijo izquierdo `LEAF:254` y luego el derecho `LEAF:947`.

```
OR(LEAF:780, AND(LEAF:254, LEAF:947)) [[-,H,-,-,H][-,B,-,-,-][-,-,-,-,-][-,-,-,-,B]]
```

Código Fuente 3.1: Representación en texto del CPT de la figura 3.1.

El segundo grupo de caracteres de la representación en texto está escrito entre corchetes '[]', y corresponde a las configuraciones posibles del CPT. Cada configuración se encuentra a su vez también encerrada entre corchetes '[]'. En el ejemplo, el CPT puede ser configurado de cuatro formas distintas, por lo tanto en la representación aparecerán cuatro subgrupos de caracteres separados por corchetes. Cada subgrupo tiene la configuración de cada nodo

Nodo	Transcripción
$\wedge$	`AND`
$\vee$	`OR`
$\rightarrow$	`SEQ`
$\circlearrowleft$	`LOOP`
$\times$	`AND`
Actividad de nombre ‘ooo’	`LEAF:ooo`

Tabla 3.1: Transcripción a texto de los nodos de un CPT.

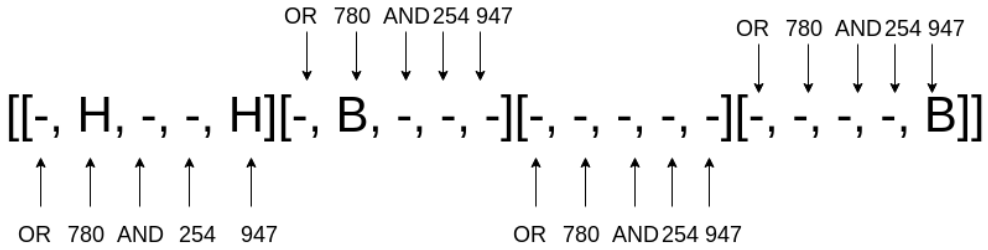


Figura 3.2: Correspondencia de las configuraciones de cada nodo del CPT (figura 3.1) en su representación en texto.

para esa variante del CPT, separadas por comas, y en el mismo orden en que aparecen en la representación de la estructura. En el ejemplo, la primera variante del CPT se transcribe como `[H, -, -, H]`, pues en esta configuración las actividades 780 y 947 deben ser escondidas.

La configuración para la primera variante de la actividad 780 aparece en la segunda posición. Esto es así pues cada subgrupo se construye leyendo los nodos de la estructura desde arriba hacia abajo y de izquierda a derecha, al igual que cuando se transcribe la primera parte de la representación en texto. La raíz aparecerá primero, y en segundo lugar, su primer hijo más a la izquierda, es decir, la actividad 780. Por su parte, la configuración de la actividad 947 en esta variante aparece en la última posición, pues es el nodo que se encuentra más abajo y más a la derecha de la estructura del CPT. La correspondencia entre los nodos y sus respectivas configuraciones en la notación se puede apreciar en la figura 3.2. Ahí se observa con más claridad que todas las configuraciones posibles de cada nodo corresponden en posición con lo transcrito en la parte estructural de la notación. Las configuraciones del nodo de la operación $\vee$ quedan definidas con el primer elemento de cada subgrupo; las de la actividad 780, con el segundo elemento de cada subgrupo, y así secuencialmente.

3.2. Visualización de CPTs

El equipo de GEMS necesita generar figuras de CPTs para poder publicar resultados de sus experimentos. Actualmente esto se lleva a cabo usando la librería TikZ [20], una herramienta compleja y poderosa para crear elementos gráficos en LaTeX. Pero hereda algunas de las desventajas de un sistema de tipografía como TeX: una empinada curva de aprendizaje, dificultades propias de dibujar sin una herramienta WYSIWYG, es decir, el código no muestra cómo se verán las cosas realmente, y finalmente, pequeños cambios requieren de largos tiempos

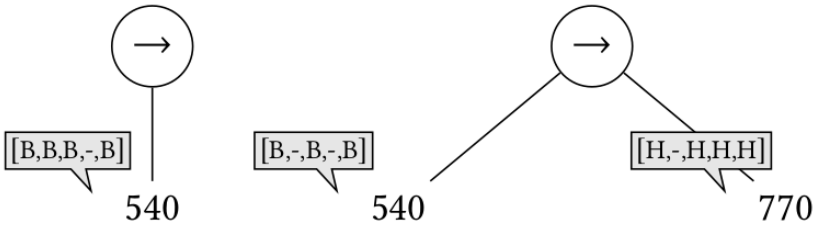


Figura 3.3: Ejemplo de una de las visualizaciones más básicas generadas por el equipo de GEMS para sus publicaciones.

de compilación.

Esto conlleva gastos excesivos en tiempo y esfuerzo solo en generar visualizaciones de los resultados obtenidos, en vez de dedicar estos recursos a la investigación misma. Por otra parte, el código generado para ello es poco reusable y casi ilegible. A continuación se describen algunos ejemplos de los problemas con los que el equipo de GEMS tiene que lidiar al momento de presentar resultados.

En la figura 3.3 se observan dos CPTs muy simples generados por el equipo de GEMS usando el paquete TikZ, con dos y tres nodos respectivamente. El primero consiste en un operador $\rightarrow$ que tiene un solo hijo etiquetado con sus variantes, la actividad 540. El segundo árbol también posee como raíz un operador $\rightarrow$, el cual tiene dos actividades como hijos: 540 y 770, ambos con etiquetas de configuración. Para lograr estas visualizaciones, primero se deben definir los estilos del grafo, como se ejemplifica en el código fuente 3.2. Allí se definen tres estilos de nodo, usando nombres poco significativos para algún otro lector: `circ`, `circ2` y `circ3`. Ya aquí se puede apreciar lo poco reusable que es esta herramienta, pues se puede notar que los tres estilos definen las mismas reglas salvo por pequeñas diferencias, como la orientación (`east` y `west`).

Solo una vez definidos los estilos, se puede comenzar a describir la estructura de la figura del CPT. En el código fuente 3.3 se muestran los comandos necesarios para poder mostrar los CPTs de la figura 3.3, haciendo uso del estilo *circ*2 (líneas 6, 13 y 14) definido en el código fuente 3.2. Obsérvese que para generar una figura más o menos simple como ésta, las instrucciones son complejas, y requieren de un cierto grado de dominio de la librería. Por lo demás, las medidas en centímetros, por ejemplo el largo de los arcos y la posición de las etiquetas, han sido definidas luego de cierto tiempo no despreciable de ensayo y error.

Todo el trabajo descrito anteriormente fue realizado para figuras simples. Pero cuando los resultados de los experimentos arrojan CPTs más complejos el problema se vuelve aún más difícil. Obsérvese la figura 3.4, en la que se muestra un CPT de seis niveles y más de trienta nodos. El código necesario para generarlo se puede ver en el código fuente 3.4, (dentro de un contexto LaTeX), donde claramente se observa una alta complejidad y poca legibilidad. Según los miembros del equipo de GEMS, el tiempo utilizado en definir esta figura fue tal, que se optó por no continuar usando los estilos predefinidos. Tampoco valía la pena seguir gastando tiempo en ello, sobretodo en la tarea de evitar que las etiquetas se traslapen.

```
\forestset{
  circ/.style={
    label={[rectangle callout, fill=gray!20, draw, inner sep=.3ex,
    label distance=-18.5pt, anchor=east, above=0.3cm, callout pointer
    width=1.2 mm, callout pointer shorten=3mm]east:{\scriptsize #1}},
  },
  circ2/.style={
    label={[rectangle callout, fill=gray!20, draw, inner sep=.3ex,
    label distance=-28.5pt, anchor=east, above=0.3cm, callout pointer
    width=1.2 mm, callout pointer shorten=3mm]east:{\scriptsize #1}},
  },
  circ3/.style={
    label={[rectangle callout, fill=gray!20, draw, inner sep=.3ex,
    label distance=-28.5pt, anchor=west, above=0.3cm, callout pointer}
    width=1.2 mm, callout pointer shorten=3mm]west:{\scriptsize #1}},
  }
}
```

Código Fuente 3.2: Estilos de la librería TikZ definidos por el equipo de GEMS para visualizar CPTs.

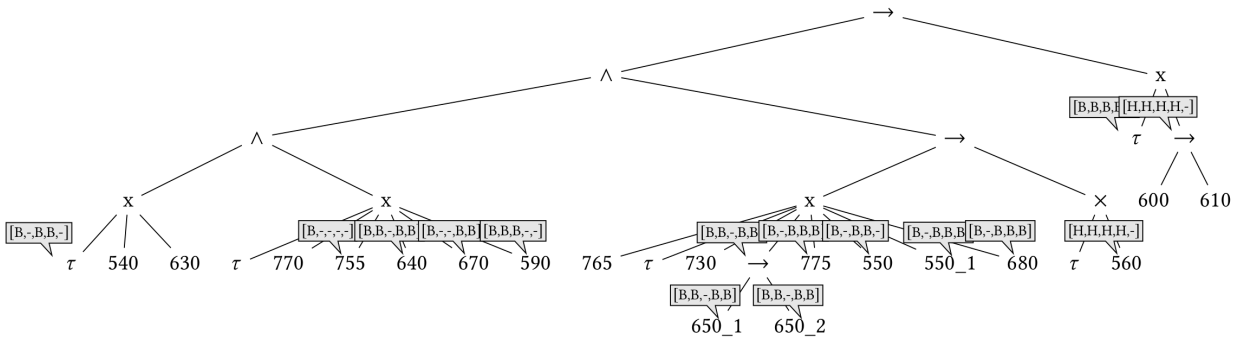


Figura 3.4: Ejemplo de una visualización de un CPT más complejo generado por el equipo de GEMS para sus publicaciones.

En efecto, el problema del traslape de etiquetas no lo tiene solamente el equipo de GEMS. En otras publicaciones que involucran árboles de procesos configurables, se observan figuras altamente complejas, donde se puede intuir que el posicionamiento de etiquetas se hizo probablemente de forma manual. Esto significa que no hay alguna solución adecuada para la visualización de estos modelos. En las figuras 3.5 y 3.6 se observan ejemplos de CPTs complejos que se han encontrado en otras publicaciones [1] [2], donde el traslape de etiquetas y arcos dificulta la comprensión de la visualización.

3.3. Discusión

Considerando todo lo anteriormente descrito, se han identificado dos importantes obstáculos para el equipo de GEMS al momento de analizar los resultados de sus experimentos. En primer lugar, la representación en texto de los CPTs generados por ProM6 es difícil de

```
\begin{figure}
\centering
  \subfloat[]{
    \Forest{
     [$\rightarrow$, circle, draw
     [540, l=1.3cm, circ2={$\lbrack$B,B,B,-,B$\rbrack$}]]
    } \label{initial_tree}}
    \quad
    \subfloat[]{
    \Forest{
    [$\rightarrow$, circle, draw,
    s sep = 2.4cm
    [540, circ2={$\lbrack$B,-,B,-,B$\rbrack$},l=1.3cm]
    [770, l=1.3cm, circ2={$\lbrack$H,-,H,H,H$\rbrack$}]]
    }
    \label{first_gen}}
\end{figure}
```

Código Fuente 3.3: Comandos utilizados por el equipo de GEMS para generar los CPTs de la figura 3.3.

manipular e interpretar; y en segundo lugar, el proceso para generar las visualizaciones de estos resultados es engorroso y consume mucho tiempo.

Es razonable pensar en una alternativa para generar visualizaciones de CPTs directamente a partir de la representación en texto. Sería deseable que además éstas pudiesen ser ajustadas posteriormente a través de alguna herramienta gráfica tradicional o un editor de imágenes vectoriales. Esto facilitaría enormemente el trabajo de investigación, agilizando el proceso de análisis y visualización de resultados, así como su posterior publicación a través de imágenes que pueden ser importadas al documento directamente.

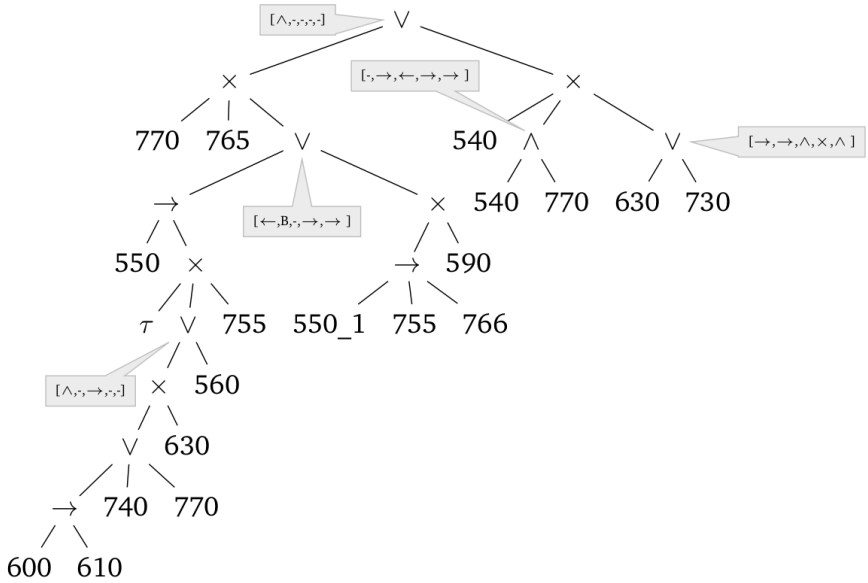


Figura 3.5: Visualización de un CPT complejo utilizada en otras publicaciones [1].

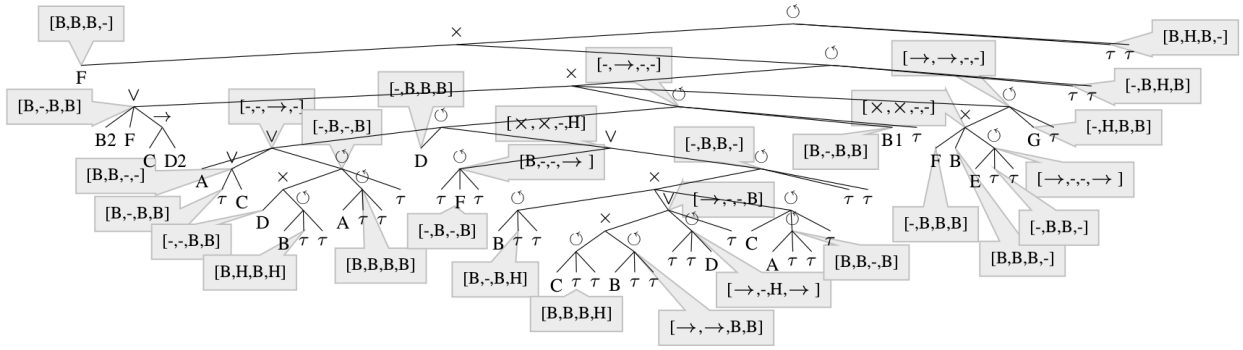


Figura 3.6: Visualización de un CPT complejo utilizada en otras publicaciones [2].

```
\Forest{
  [$\rightarrow$ [$\land$ [$\land$ [x
  [$\tau$,circ={$\lbrack$B,-,B,B,-$\rbrack$}] [540] [630]]
  [x [$\tau$] [770] [755, circ={$\lbrack$B,-,-,-,-$\rbrack$}]
  [640,circ={$\lbrack$B,B,-,B,B$\rbrack$}]
  [670, circ={$\lbrack$B,-,-,B,B$\rbrack$}]
  [590,circ={$\lbrack$B,B,B,-,-$\rbrack$}]]]
  [$\rightarrow$ [x [765] [$\tau$] [730]
  [$\rightarrow$,circ={$\lbrack$B,B,-,B,B$\rbrack$}
  [650\_1, circ={$\lbrack$B,B,-,B,B$\rbrack$}]
  [650\_2, circ={$\lbrack$B,B,-,B,B$\rbrack$}]]
  [775,circ={$\lbrack$B,-,B,B,B$\rbrack$}]
  [550,circ={$\lbrack$B,-,B,B,-$\rbrack$}]
  [550\_1,circ={$\lbrack$B,-,B,B,B$\rbrack$}]
  [680,circ={$\lbrack$B,-,B,B,B$\rbrack$}]]
  [$\times$[$\tau$][560, circ={$\lbrack$H,H,H,H,-$\rbrack$}]]]]
  [x [$\tau$,circ={$\lbrack$B,B,B,B,-$\rbrack$}]
  [$\rightarrow$,circ={$\lbrack$H,H,H,H,-$\rbrack$} [600] [610]]]]
}
```

Código Fuente 3.4: Comandos utilizados por el equipo de GEMS para producir el CPT de la figura 3.4.

Capítulo 4

Análisis y Diseño de la Solución

Durante esta etapa del trabajo se hizo una primera aproximación a la solución extendiendo el paquete `ProcessTree` de ProM6 mencionado en los capítulos anteriores, con la idea de reutilizar código y seguir los guidelines del framework. Pero debido a la poca mantenibilidad de esta librería y el exceso de funcionalidades que no se requerían en el escenario de este problema, se decidió descartar esta alternativa antes de profundizar el desarrollo.

Como segunda aproximación se propuso como solución el desarrollo de un paquete completamente nuevo que implementase su propia estructura de datos para representar y manipular CPTs de acuerdo a los requerimientos. A su vez se decidió utilizar una librería más adecuada y conveniente que la que usan la mayoría de los paquetes de ProM6 para la creación de archivos `svg`.

4.1. Primer prototipo

Para llevar a cabo el desarrollo de la solución, primero se hizo una revisión de la documentación de ProM6 y de paquetes ya existentes, en búsqueda de funcionalidades que actuaran como punto de partida del desarrollo de este trabajo. Como se mencionó en los capítulos anteriores, ya se desarrolló un paquete para el análisis y manipulación de árboles de procesos (no configurables), llamado `ProcessTree` [19], el cual parecía un buen punto de partida.

4.1.1. Paquete ProcessTree

`ProcessTree` implementa de manera muy completa la representación de procesos a través de árboles de procesos. Define el modelo de un árbol de proceso en sus tres perspectivas mencionadas en la sección 2.1: flujo de actividades, recursos (o actores), y data. La primera consiste en la estructura misma del árbol, es decir, sus nodos, actividades, arcos, y métodos para construir y manipular estos grafos. Las otras dos perspectivas corresponden a todo lo que se refiere a la metadata y el contexto del proceso que se está modelando con el árbol,

como los actores (en este contexto, originadores) y variables involucradas. Como se indicó en esa misma sección, para efectos de esta memoria, estas últimas dos perspectivas no fueron consideradas, pues el énfasis se puso solamente en la visualización de la estructura de un CPT.

En la figura 4.1 se puede observar un diagrama de clases de la implementación de la estructura de datos para un árbol de proceso del paquete `ProcessTree`. Se divide en tres módulos, uno para cada perspectiva del modelo del árbol de procesos. En particular en color rojo se observan las clases que implementan lo necesario para la manipulación de un árbol de procesos a nivel estructural.

El modelo intenta seguir el patrón de diseño 'Composite', dada la naturaleza de árbol de esa estructura, pero debido a la presencia de las otras dos perspectivas, es decir, la de las variables y la de los actores (en azul y verde), se requieren de más clases, las cuales de alguna forma ensucian la estructura del patrón tradicional. Por ejemplo, se introduce una clase para encapsular el comportamiento de los arcos en el árbol, `Edge` e interactuar con la perspectiva de las variables (azul). Por otro lado las actividades, es decir, la clase `Task`, no se encuentra al mismo nivel que los operadores como se podría esperar (OR, XOR, AND, etc.), pues debe comunicarse con el módulo de recursos (verde).

A partir del modelo de clases para representar un árbol de proceso, `ProcessTree` implementa variados y complejos módulos que ofrecen un abanico de funcionalidades en torno a esta estructura de datos. Esto incluye la importación, exportación y conversión entre objetos Java y otros archivos, como `ptml`[1]. Junto a lo anterior, también provee un panel de visualización y edición de árboles de procesos. En la figura 4.2 se muestra la vista del editor de árboles de procesos de `ProcessTree`, el cual está dividido en tres secciones. Destacado en color amarillo en la figura, se observa el panel visualizador de árboles de proceso. Éste permite tanto el ajuste manual de la posición de los nodos y arcos, como también cambios en sus nombres desplegados. En color magenta se muestra el panel de edición, con tres pestañas independientes. Cada una de ellas ofrece una interfaz que permite al usuario añadir y borrar nodos, y otra metadata del árbol, como recursos, variables y actores. Finalmente, en color cyan, se observa el panel de propiedades, que muestra la información que contiene un nodo cuando éste es seleccionado en el panel de visualización.

Las secciones del panel que son de interés para este trabajo de memoria son el panel visualizador de árboles y la pestaña de control de flujo ('Control-Flow') del panel de edición, a la derecha. En esta pestaña se permite agregar, editar y eliminar nodos, arcos y actividades del árbol de proceso que se observa en el panel contiguo. Otra funcionalidad interesante del editor es que permite colapsar secciones del árbol: Al hacer click derecho en algún nodo, todos sus hijos correspondientes se esconden. Esto podría facilitar la comprensión de la estructura de árboles demasiado complejos

Para el desarrollo de este primer prototipo se decidió utilizar el paquete `ProcessTree` como base para implementar las funcionalidades que se buscan en este trabajo. Esto, pues incorpora de manera integral todos los conceptos de un árbol de proceso, los cuales se pueden reusar y extender para definir un árbol de proceso configurable. Además el paquete está siendo

[1]Formato de archivo para representar árboles de procesos en lenguaje XML.

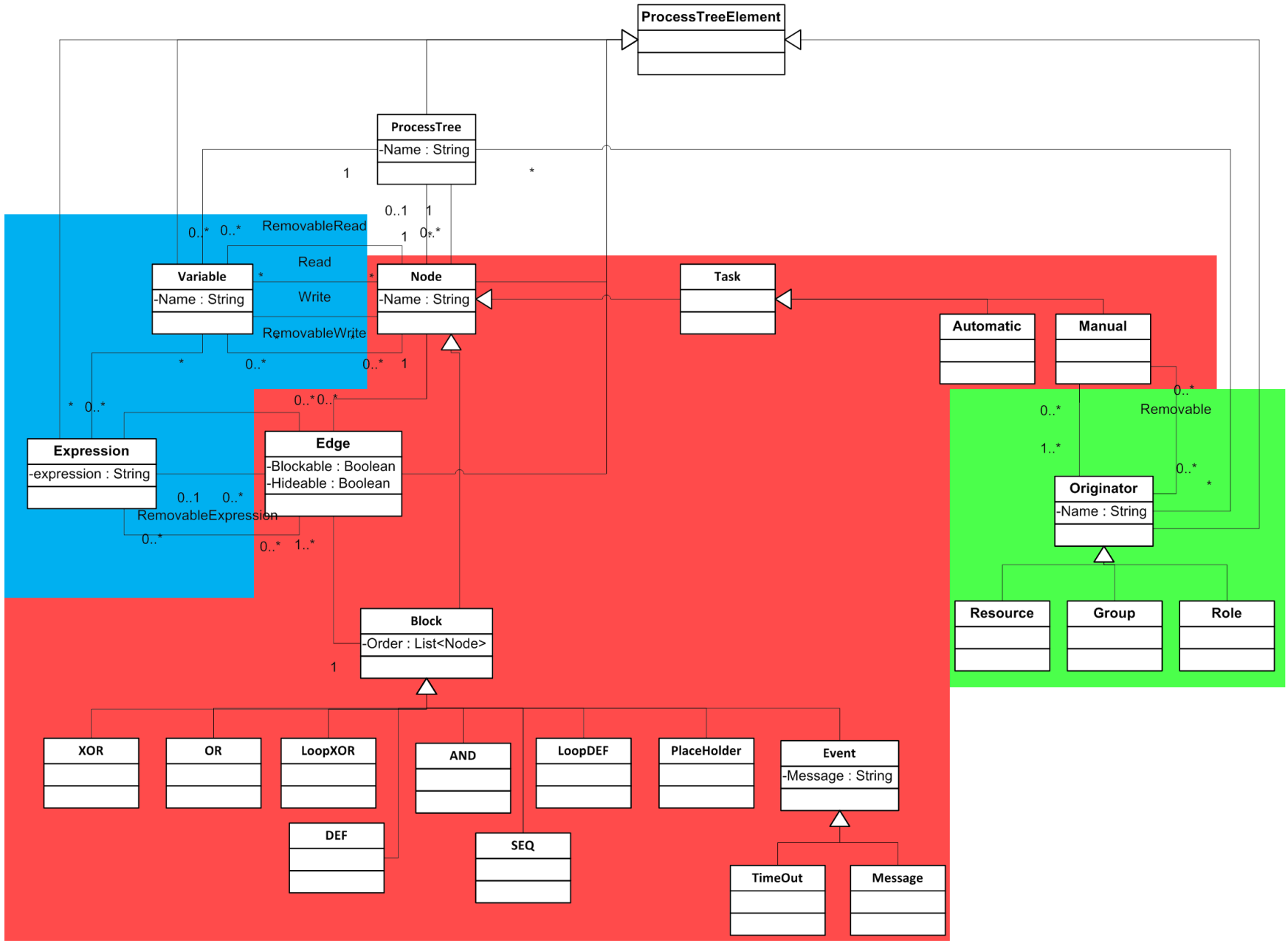


Figura 4.1: Meta-modelo del paquete `ProcessTree`. La parte roja está relacionada al modelo de la estructura del árbol en sí, mientras que los otros colores corresponden a los componentes que modelan metadatos del árbol.

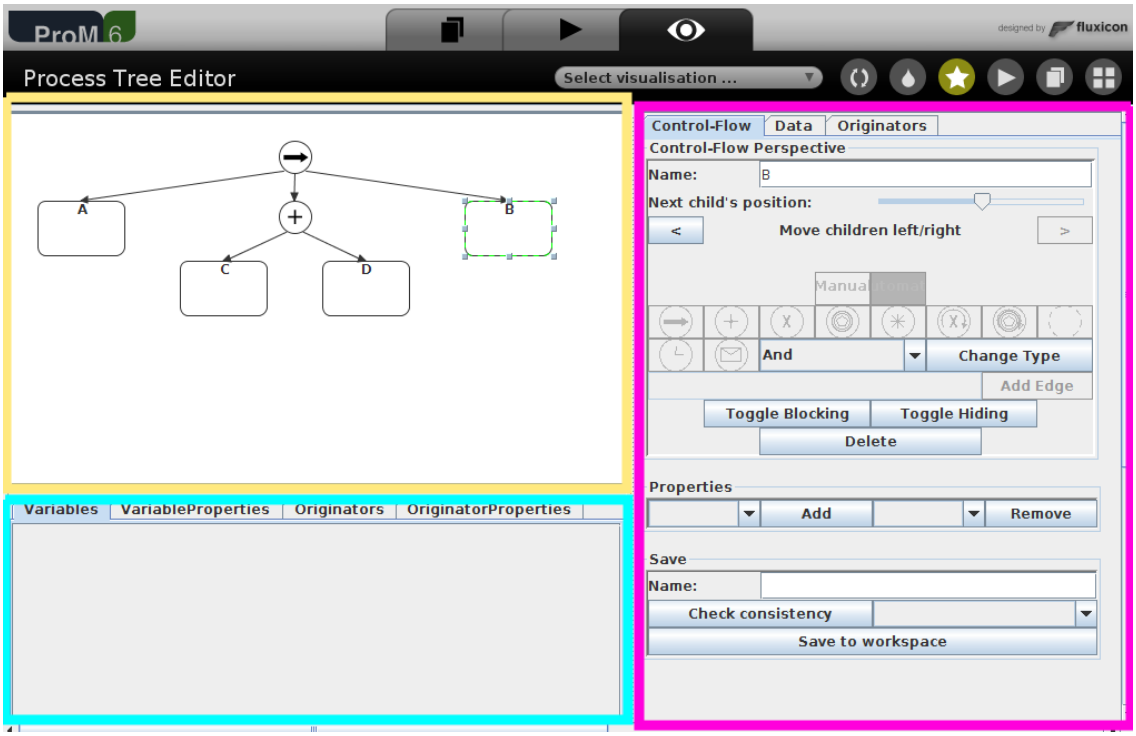


Figura 4.2: Editor de árboles de procesos del paquete `ProcessTree`. En color amarillo se destaca el panel visualizador de árboles. En magenta el panel de edición y en cyan el panel de propiedades.

actualizado constantemente, y posee una extensa documentación [19], pues es ampliamente usado en la comunidad de ProM6.

4.1.2. Limitaciones de esta alternativa

Se comenzó a desarrollar un pequeño prototipo de solución que extendía el paquete `ProcessTree`, en particular, las clases que implementaban `Block`. Pero desde el primer momento se observó que la estructura de clases era bastante compleja, y presentaba también muchos "code smells".

Por una parte, el constructor de un objeto `ProcessTree` utilizaba una enorme sentencia condicional para decidir el tipo de elemento a retornar. Un extracto de este código se puede ver en el listado A.2, donde se observa el uso de múltiples condicionales y abuso de `instanceof` y casts.

Por otra parte, el método que lleva a cabo el render de un árbol de proceso en el panel visualizador también implementa un gran switch case para pintar cada componente (ver listado A.3).

Se deduce entonces que el código presenta al menos dos code smells: "Type-checking" para el primer caso, y "God class" para el segundo. De lo contrario, cada tipo de nodo implementaría sus propios métodos de construcción y dibujo. Esto implica que la mantenibilidad del

código es compleja y su reutilización, muy difícil.

Además de lo anterior, existen demasiados elementos que no son necesarios en el contexto de este trabajo, y que más bien entorpecen el desarrollo. Como por ejemplo, los bloques `Def`, `LoopDef`, `PlaceHolder`, `Event` y sus subclases. Los arcos, y todas las variables y métodos que guardan relación con las perspectivas de data y recursos tampoco son de utilidad para la solución del actual problema.

Finalmente, la funcionalidad para exportar una figura de un árbol de procesos a formato `svg` no está implementada en este paquete, por lo que habría que comenzar este trabajo desde cero, lidiando con los problemas mencionados anteriormente.

En definitiva, el paquete `ProcessTree` es demasiado complejo para los requerimientos de este trabajo. Dado que además éste presenta poca cohesión en sus clases, y bastante acoplamiento con las perspecivas de data y recursos, se optó por desarrollar una solución propia. Esto significa implementar lo justo y necesario, pero centrándose en la extensibilidad, para que problemas como los que presenta `ProcessTree` no ocurriesen en este nuevo paquete.

4.2. Solución final

Dadas las dificultades explicadas anteriormente, se tomó la decisión de recomenzar el desarrollo partir de cero, es decir, de implementar un paquete de ProM6 completamente nuevo. Éste no depende de ningún otro paquete del framework, e implementa su propia estructura de datos para un CPT, sin extender del ya existente `ProcessTree`. Además de éste, se desarrolló un módulo que se encarga de parsear archivos con la representación en texto de un CPT descrita en la sección 3.1 e importarlos como la estructura de datos para ProM6. También se implementó un módulo para exportar el modelo a un imagen en formato `svg`, para lo cual se utilizó la librería de diagramación JGraphX [21]. Finalmente se desarrollaron cuatro plug-ins de ProM6 que hacen uso de estos módulos para importar y exportar CPTs hacia y desde el framework, visualizarlos, y aplicar las configuraciones y así obtener las variantes del CPT en cuestión.

En las siguientes subsecciones se describirá el diseño de los módulos más importantes de este paquete. En primer lugar se hablará de la estructura de datos para representar un árbol de proceso, mientras que en la segunda subsección se entregarán más detalles sobre el módulo que parsea los archivos de representación de un CPT. Luego se explicará el código que se encarga de la exportación del modelo a archivos formato `svg`, y finalmente se hablará de la implementación de los plug-ins que integran todas las funcionalidades al framework ProM6.

4.2.1. Estructura de datos para un CPT en ProM6

En la figura 4.3 se presenta un diagrama de clases de la estructura de un CPT para esta solución, mostrando solo los métodos de clase más relevantes para entender el funcionamiento. Se observa un patrón de diseño 'Composite' que se justifica por la naturaleza de árbol

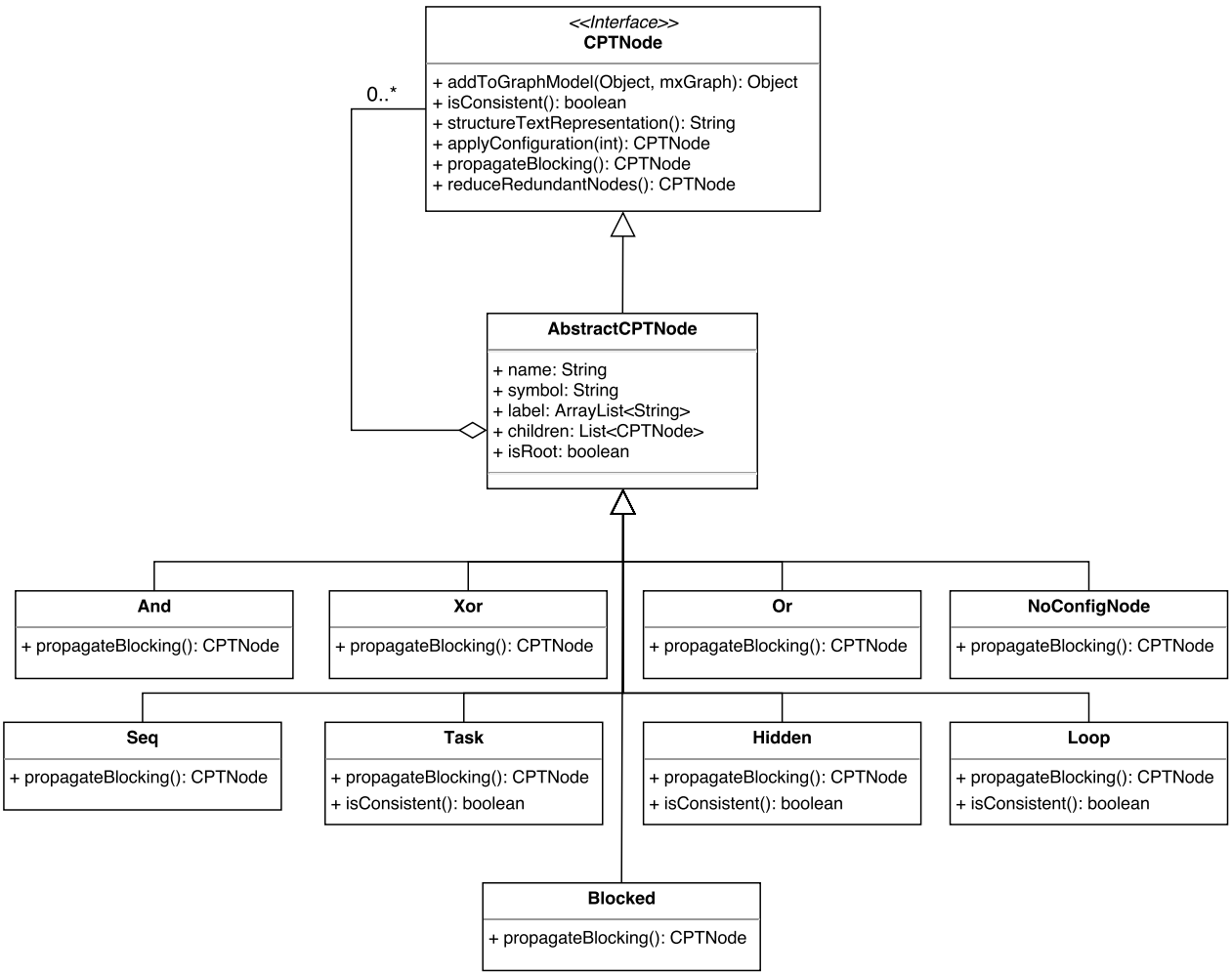


Figura 4.3: Diagrama de clases de la estructura de un CPT.

de un CPT. Se tiene la interfaz `CPTNode` que representa un nodo del árbol, el cual puede ser un operador, una actividad, un nodo bloqueado o escondido, entre otros. Expone al cliente los métodos necesarios para manipular, dibujar y aplicar las configuraciones de un CPT (`isConsistent`, `applyConfiguration`, `propagateBlocking`, etc.). También incluye los métodos que implementan la conversión de CPT a `svg` (`addToGraphModel`) y a su representación en texto (`structureTextRepresentation`).

Por su parte, la clase `AbstractCPTNode` implementa directamente la interfaz mencionada anteriormente, aunando el comportamiento en común de todos los tipos de nodos que existen en la estructura. Esta clase es abstracta, sin embargo implementa todos los métodos de la interfaz, y no posee ningún método abstracto. Se mantiene como abstracta para que no pueda ser instanciada, dado que no existe un nodo con el comportamiento de esta clase. Algunos métodos son compartidos por todos los nodos que heredan de la clase `AbstractCPTNode`, y para los métodos que no, esta clase ofrece un comportamiento por defecto. Por ejemplo, el método `isConsistent` verifica que el nodo tenga una cantidad de hijos que sea consistente con su definición. En el caso de una actividad, este nodo no debe tener ningún hijo, mientras que en el caso de un operador ⟲, éste debe tener tres y solo tres hijos. Así, las clases `Task` y `Loop` respectivamente sobreescriben el método `isConsistent`. El resto de los nodos heredan

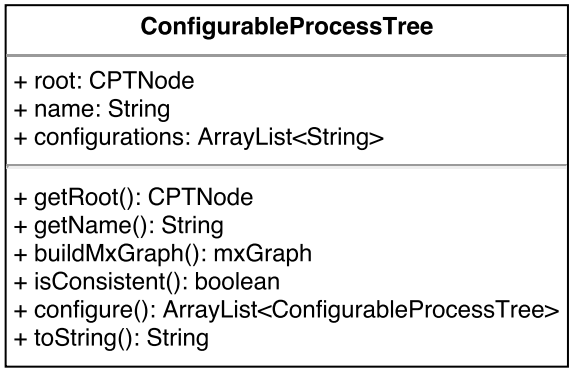


Figura 4.4: Diagrama de clase de `ConfigurableProcessTree`.

el método definido en la clase padre `AbstractNode`, que verifica que la cantidad de hijos sea mayor a uno.

Dados los requerimientos del problema, a diferencia de la implementación del paquete `ProcessTree`, la noción de arcos en este módulo no es necesaria, y por lo tanto es omitida pues las conexiones entre nodos se deducen del patrón de diseño utilizado.

Para el manejo de las variantes del árbol, cada nodo guarda en el campo `label` sus propias configuraciones, en un arreglo de nodos también de tipo `CPTNode`. Este arreglo es relevante al momento de querer obtener una variante en particular del CPT. De esto se encarga el método `applyConfiguration`, que recibe un índice que representa la i-ésima variante del árbol, y al aplicarlo sobre su nodo raíz, se ejecuta recursivamente hacia los hijos. El método retorna el nodo configurado, reemplazando cada nodo por el i-ésimo elemento en el arreglo de configuraciones que posee.

Luego se aplican los métodos que propagan recursivamente el bloqueo y redundancia de nodos, como se explicó en la subsección 2.2. Para el caso particular en el que el nodo no cambia para alguna variante, existe un nodo especial en la i-ésima posición del arreglo `label`. Este corresponde a un nodo de tipo `NoConfigNode` que cubre este caso, que retornar una copia del nodo que no necesita configurarse para determinada variante del árbol. Esto se hace así para que las variantes obtenidas sean árboles nuevos, evitando mutar el CPT original[2].

Envolviendo toda la implementación de un CPT anteriormente descrita, existe la clase `ConfigurableProcessTree`, la cual representa la estructura de datos con la que trabajará directamente ProM6 (ver figura 4.4). Esta clase contiene como miembros un objeto `CPTNode` correspondiente a la raíz del árbol mismo, y otros campos que facilitan su manipulación dentro del framework. Por ejemplo, el nombre del archivo desde el cual se importó y la representación en texto de sus configuraciones, lo cual optimiza su exportación, como se explicará más en detalle en las próximas subsecciones.

[2]Esto se hace así debido a requerimientos de ProM6, que no permiten a los plug-ins mutar los objetos que participan dentro del framework [13].

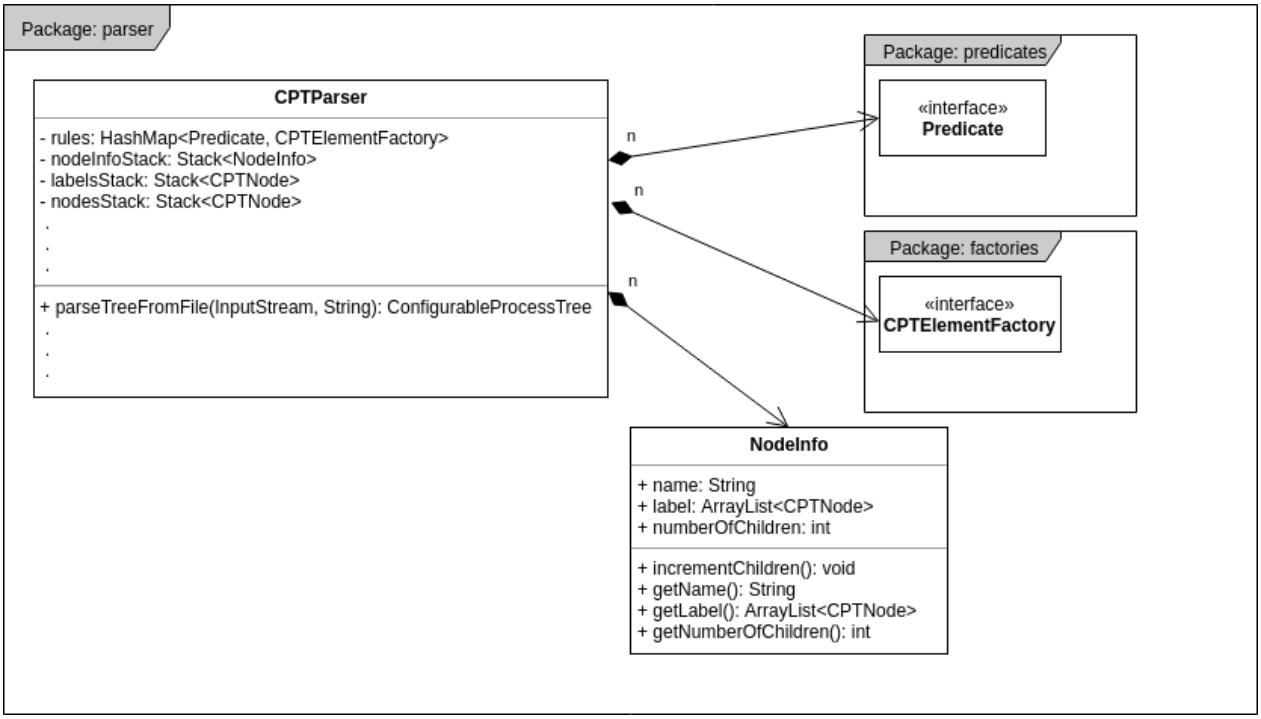


Figura 4.5: Módulo que implementa el parser de representaciones en texto de un CPT.

4.2.2. Parser de representación en texto de CPTs

Este módulo implementa las funcionalidades que permiten construir un objeto de tipo `ConfigurableProcessTree` a partir de la representación en texto de un CPT descrita en la sección 3.1. Uno de los objetivos principales del desarrollo de este módulo fue lograr que fuese lo más extensible posible en el caso, por ejemplo, de agregar nuevos tipos de nodos al modelo de un CPT.

El parser está implementado en la clase `CPTParser` que se muestra en el diagrama de la figura 4.5. Dentro de sus campos, posee tres stacks y un set de reglas de parseo para llevar a cabo el proceso. En términos generales, el parser actúa usando estos elementos mientras va leyendo caracter a caracter la representación en texto del CPT. Para facilitar la implementación, se tomó la decisión de permitir en la representación en texto, actividades cuyo nombre fuesen únicamente dígitos, para evitar colisiones entre nombres de actividades y nombres de los operadores.

Primero que todo, se obtiene el grupo de caracteres que representan las variantes del árbol y se procesa, reagrupando las configuraciones de acuerdo al nodo que le corresponden, en vez de la configuración. Las etiquetas de cada nodo resultantes se insertan en el stack `labelsStack` que participa en el proceso de parseo de la estructura del árbol posteriormente. El stack `nodeInfoStack` contiene información parcial sobre los nodos a construir a medida que se va leyendo el texto caracter a caracter, mientras que el stack `nodesStack` tiene los nodos que ya se han construido.

En la tabla 4.1 se observan los pasos más importantes en el proceso de parseo del texto

```
AND(XOR(LEAF:1, LEAF:2), LEAF:3) [[-,H,-,-,H][-,B,-,-,B]]
```

Código Fuente 4.1: Representación en texto de un CPT de ejemplo para explicar los pasos del proceso de parsing

de ejemplo que se tiene en el código fuente 4.1.

1. Al comenzar, es decir, cuando aún no se ha leído ningún caracter del texto, se inserta en `nodeInfoStack` un nodo fantasma para prevenir excepciones al final del proceso. Por su parte, se han preprocesado las configuraciones del CPT que vienen en el archivo de texto y se insertan en `labelsStack`. Así, este stack contiene todas las etiquetas de cada nodo, en orden de aparición. Se observa que `nodesStack` se encuentra vacío pues aún no se construye ningún nodo.

2. Se comienza a leer caracter a caracter hasta que se encuentra un '`(`'. Allí se extrae la primera etiqueta de `labelsStack` y se inserta en `nodeInfoStack` la información que se tiene hasta el momento: el nombre del nodo encontrado ('`AND`'), su etiqueta ('`-,-`') y su contador de hijos, que comienza en 0 pues no se ha visto ninguno hasta ahora.

3. Se siguen leyendo caracteres, y de nuevo se encuentra un '`(`'. De forma análoga al paso anterior, se extrae otra etiqueta de `labelsStack` y se inserta un nuevo nodo en `nodeInfoStack`: un nodo de nombre '`XOR`', etiqueta '`H,B`' y 0 hijos.

4. Se encuentra el texto '`LEAF:`', por lo tanto se crea un nuevo nodo de actividad. Dado que no hay que esperar a seguir leyendo el texto para completar la información, pues una actividad no posee hijos, el nodo se crea y se guarda inmediatamente en `nodesStack`. Éste es un nodo tipo `Task` con el nombre '1' y y su etiqueta es la siguiente que aparece en `labelsStack`, '`-,-`'. Finalmente, dado que se ha creado un nodo, se aumenta el contador de hijos del último elemento en el stack `nodeInfoStack`, es decir, el del nodo `XOR`.

5. En este paso se vuelve a encontrar una hoja, y de la misma forma crea la actividad `Task` de nombre '2', con su correspondiente etiqueta `-,-` y se inserta en `nodesStack`. Luego se vuelve a aumentar el contador del último nodo de `nodeInfoStack`.

6. Luego de haber creado las dos hojas, se encuentra un '`)`', que indica que se ha terminado de definir los hijos de un nodo, el cual corresponde siempre al último nodo del stack `nodeInfoStack`. Ahora que se tiene tanto la información del nodo como sus hijos, se extrae '`XOR`' y se extrae la cantidad de nodos en `nodesStack` indicada en su contador de hijos. Se crea el nodo de tipo '`Xor`', con su etiqueta `H,B` y sus dos hijos, las actividades 1 y 2. El nuevo nodo construido se vuelve a insertar en `nodesStack` y se aumenta el contador de hijos del primer nodo de `nodeInfoStack`, es decir, el de 'AND'.

7. Se vuelve a encontrar la secuencia '`LEAF:`' y por lo tanto se crea una nueva actividad, con su etiqueta, su nombre y se inserta en `nodesStack`. Como siempre, se aumenta el contador de hijos del nodo en `nodeInfoStack`.

8. Finalmente se encuentra el último caracter, '`)`', el cual indica que se tiene toda la

información para crear el último nodo, 'AND'. Se extraen dos hijos de `nodesStack`, dado que su contador tiene valor 2, y se crea el nodo '`And`', con su etiqueta y los dos hijos, la actividad 3 y el nodo `Xor` construido anteriormente.

Si el texto tenía un formato correcto, `nodeInfoStack` siempre debería contener un solo elemento, el fantasma; `labelsStack` debería quedar vacío, y `nodesStack` debería contener un solo elemento también: la raíz del árbol parseado.

En el caso en que las condiciones anteriores no se cumplieren, se lanzará la excepción `IncorrectCPTStringFormat`. La misma excepción se lanza cuando existe un `EmptyStack-Exception` en el proceso, pues implica que también hubo un problema de formato de texto, como paréntesis desbalanceados.

Cuando se desea parsear y construir un objeto a partir de un string, llega un momento en el que inevitablemente se deben tomar decisiones. En este caso, se debe escoger qué tipo de nodo instanciar frente a qué secuencia de caracteres. Por ello, se hace necesario utilizar sentencias condicionales para decidir construir un nodo tipo `And` cuando se encuentra la palabra '`AND`' en el archivo, o un `Loop` cuando aparece la palabra '`LOOP`'.

Para evitar replicar los grandes bloques de sentencias condicionales como los que se mencionaron en la subsección 4.1.2, y para lograr una mejor mantenibilidad del código, se diseñaron dos submódulos que utilizará `CPTParser` para tomar este tipo de decisiones: `predicates` y `factories` (ver figuras 4.6 y 4.7).

Dentro de `predicates` se tiene la interfaz `Predicate`, la cual expone un solo método, `checkPredicate(String word)`. Éste toma una secuencia de caracteres que recibe del texto a parsear, y retorna `true` o `false` dependiendo de si el predicado que contiene cumple con la condición o no. Por su parte, en el paquete `factories` se encuentra la interfaz `CPTElementFactory`, que expone dos métodos: `buildNode`, que construye un nodo particular para incorporarlo a la estructura del árbol, y `buildNodeForLabel`, el cual construye un nodo para insertar en el arreglo de variantes de cada elemento en el CPT. Por cada tipo de nodo, existirá un predicado y una factory correspondiente.

Las clases que implementan `Predicate` y `CPTElementFactory` interactúan entre sí para permitir a `CPTParser` decidir qué nodo construir a partir de cierta secuencia de caracteres encontrada en la representación de texto.

Al momento de instanciar el objeto `CPTParser`, recibe como parámetros un set de reglas de parseo el cual guarda como miembro de clase en el campo `rules` (ver figura 4.5). Éste corresponde a un `HashMap` donde sus llaves son predicados y los valores son factories. Cuando se encuentra una secuencia de caracteres a partir del cual se debe decidir construir cierto tipo de nodo, el parser recorre iterativamente las reglas, evaluando uno a uno los predicados. En el caso en que uno de ellos retorne `true`, se accede al valor del hashmap, es decir, a la factory, y se ejecuta el método `buildNode`, el cual retorna el nodo correspondiente al tipo que ha sido leído en el texto.

De esta manera, se puede evitar tener un bloque condicional para cada caso, y así permitir que el código sea más mantenible. Esto, pues en el caso en que quiera agregarse un

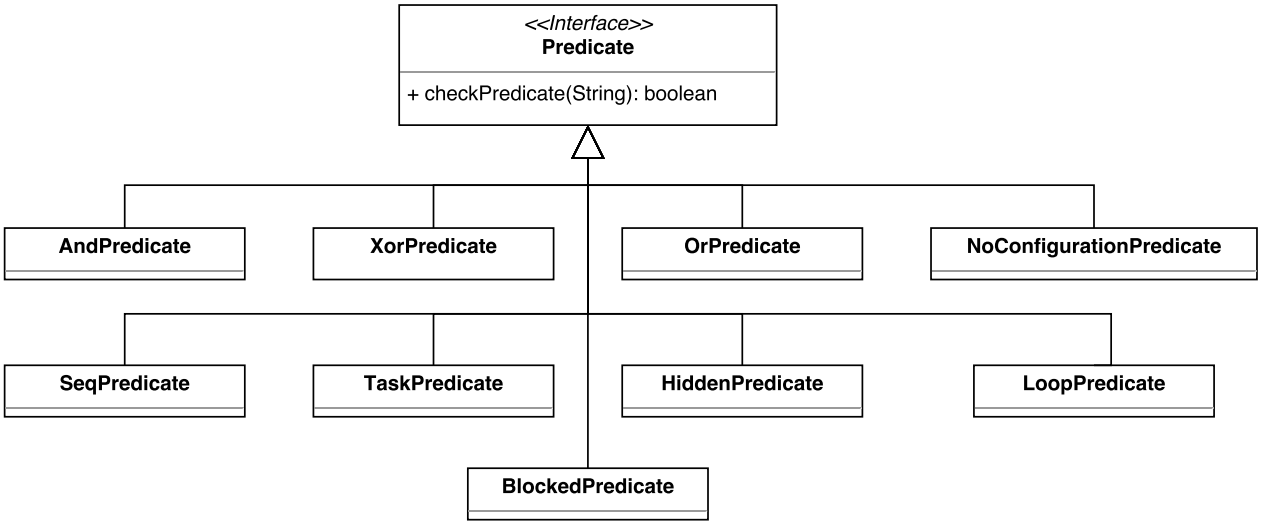


Figura 4.6: Diagrama de clases del módulo `predicates`

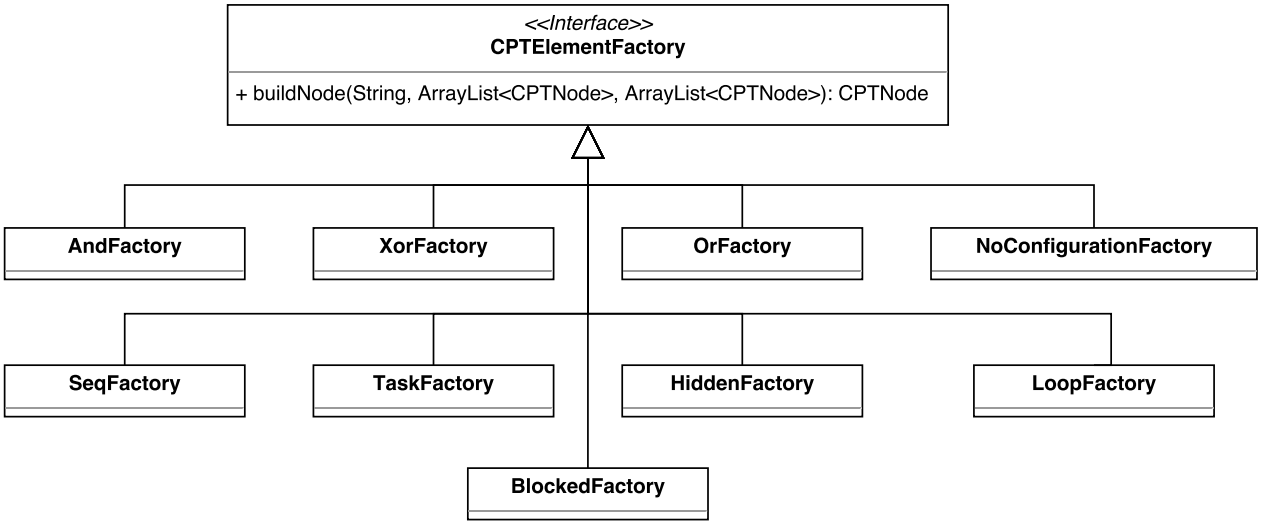


Figura 4.7: Diagrama de clases del módulo `factories`

nuevo tipo de nodo, tan solo se deben agregar dos nuevas clases que implementen `Predicate` `CPTElementFactory`, y luego agregarlas al set de reglas del parser.

4.2.3. Visualización y Exportación a formato `svg`

Para implementar esta funcionalidad se optó por utilizar la librería de visualización de diagramas JGraphX [21] [22], de la familia mxGraph[3]. Esta librería entrega un completo conjunto de funcionalidades para permitir a aplicaciones mostrar diagramas y grafos de manera interactiva.

Son varias las razones por las cuales se escogió usar esta librería:

[3]También se encuentra implementada para JavaScript, y en sus versiones alfa y beta para .NET y ActionScript

- Provee un manual de usuario muy completo, autocontenido y con ejemplos (implementados). Además el código está completamente documentado [23].

- Está siendo actualizada permanentemente (fecha de último release al momento de escribir este documento: 19 de enero de 2017).

- Alta participación de usuarios de la librería en distintas comunidades. Incluso tiene su propio tag en StackOverflow [jgraphx].

- Eficiencia. Antes de comenzar a utilizar la librería se probó construyendo y visualizando sucesivas veces un árbol binario de 2^8 nodos. La visualización se generaba correctamente en un promedio de 660 ms, lo cual es suficiente para efectos de este trabajo.

- Facilidad de uso. JGraphX maneja su propio modelo de datos de manera interna, por lo cual agregar nodos y arcos a la estructura se reduce al uso de solo dos métodos: `insertVertex` e `insertEdge`. Permite la visualización de diagramas utilizando modelos desarrollados por el propio usuario. Para ello basta sobreescribir el método `toString()` de los distintos componentes, como nodos y arcos.

- Flexibilidad en los estilos de dibujo. Utiliza stylesheets altamente personalizables para generar distintos tipos de visualizaciones, layouts de los diagramas, interacción con el usuario y manejo de otros estilos, como colores y formas.

- Soporte para exportación de los diagramas a formato `svg`. Esto cumple con uno de los requisitos del problema a resolver.

4.2.4. Plug-ins

Para incorporar todas las funcionalidades descritas anteriormente a ProM6, se implementarán cuatro plug-ins utilizando la metodología de desarrollo de plug-ins descrita en la subsección 2.3.2:

- Importación: utilizando la notación `@UIImportPlugin` se implementará un plug-in que permita importar archivos con la representación en texto de un CPT al framework a través de la interfaz de usuario (ver figura 2.11). Para ello se utilizara el módulo de parseo descrito anteriormente en la subsección 4.2.2

- Visualización: usando la notación `@Visualizer`, se desarrollorá un plug-in que permita ver en la pestaña de visualización de ProM6 un determinado CPT, haciendo click en el botón de visualización (ver figura 2.11). Para ello se utilizarán las funcionalidades de la librería JGraphX.

- Obtener variantes: haciendo uso de la notación `UITopiaVariant` se implementará un plug-in que aparecerá en el listado de acciones en la pestaña de plug-ins disponibles en ProM6 (ver figura 2.12). Al ejecutarlo generará todas las variantes del CPT y las guardará en el workspace.

- Exportación: usando la notación `@UIExportPlugin` se desarrollará un plug-in para exportar un CPT de ProM6 a una imagen formato `svg` y guardarla en un archivo local, a través de la interfaz de usuario (ver figura 2.11).

4.3. Resumen

Durante el análisis y diseño de la solución se pensó en una primera instancia extender el ya existente paquete de ProM6 `ProcessTree`. Pero debido a problemas de mantenibilidad y extensibilidad de éste, se optó por recomenzar el desarrollo, implementando un completamente nuevo paquete.

Se pretende llevar a cabo el desarrollo centrándose en la mantenibilidad, y tratando de no repetir los mismos 'code smells' encontrados en la librería `ProcessTree`.

Para visualizar y exportar los diagramas se decidió utilizar la librería de Java Swing, JGraphX, debido a su relativa facilidad de uso y extensa comunidad de usuarios y desarrolladores.

Se desarrollarán cuatro plug-ins, los cuales permitirán importar, visualizar, exportar y obtener las configuraciones de un CPT, integrando las soluciones propuestas anteriormente al framework de ProM6.

Paso	Texto leído	nodeInfoStack	labelsStack	nodesStack
1	-	dummy	- - H B - - - - H B	
2	AND(	AND -,- 0 dummy	H B - - - - H B	
3	XOR(	XOR H,B 0 AND -,- 0 dummy	- - - - H B	
4	LEAF:1,	XOR H,B 1 AND -,- 0 dummy	- - H B	Task(1)
5	LEAF:2,	XOR H,B 1 AND -,- 0 dummy	H B	Task(2) Task(1)
6	),	AND -,- 1 dummy	H B	Xor(Task(1), Task(2))
7	LEAF:3	AND -,- 2 dummy		Task(3) Xor(Task(1), Task(2))
8	)	dummy		And(Xor(Task(1), Task(2)), Task(3))

Tabla 4.1: Pasos del proceso de parseo de la representación en texto del código fuente 4.1, al modelo de CPT.

Capítulo 5

Implementación

En este capítulo se abordará el desarrollo de la solución[1] planteada en la sección 4.2. Se expondrán más detalles sobre la implementación, aplicando el diseño propuesto y mostrando algunos casos de uso.

Se presentará lo anterior en cuatro secciones, una para cada uno de los ejes de este trabajo: En la sección 5.1 se hablará del módulo encargado de importar archivos de representación en texto de un CPT al framework. Por su parte, en las secciones 5.2 y 5.3 se expondrán los módulos de visualización de un CPT y la obtención de sus variantes. Finalmente en la sección 5.4 se mostrará el funcionamiento del módulo de exportación de CPTs a archivos locales en formato `svg`.

En cada sección se incluyen ejemplos de uso desde la perspectiva de un usuario que utiliza ProM6 a través de la interfaz gráfica[2], junto a los respectivos screenshots de la herramienta.

5.1. Importación de archivos de texto

Como se explicó en la subsección 2.3.2, al desarrollar plug-ins de ProM6 existen varias notaciones Java que indican al framework cómo manipular determinado plug-in implementado. Para el caso de los plug-ins de importación, la notación utilizada corresponde a `@UIImportPlugin` y expone, entre otras cosas, la extensión del archivo que soporta. Para los archivos de representación en texto de CPTs se decidió utilizar la extensión `cpt`.

Al seleccionar el botón para importar archivos, como se muestra en la figura 2.11, se despliega un diálogo de sistema que permite escoger el recurso a importar (ver figura 5.1). Al seleccionar cierto tipo de archivo, el framework reconoce su extensión y ejecuta el plug-in de importación correspondiente que lo soporta, según lo indicado en la anotación del método que lo define.

[1] El código puede ser encontrado en el repositorio Git `https://github.com/ekauffmann/configurable-process-tree`.

[2] Para más detalles sobre el uso de la herramienta ProM6 se sugiere revisar la sección 2.3.

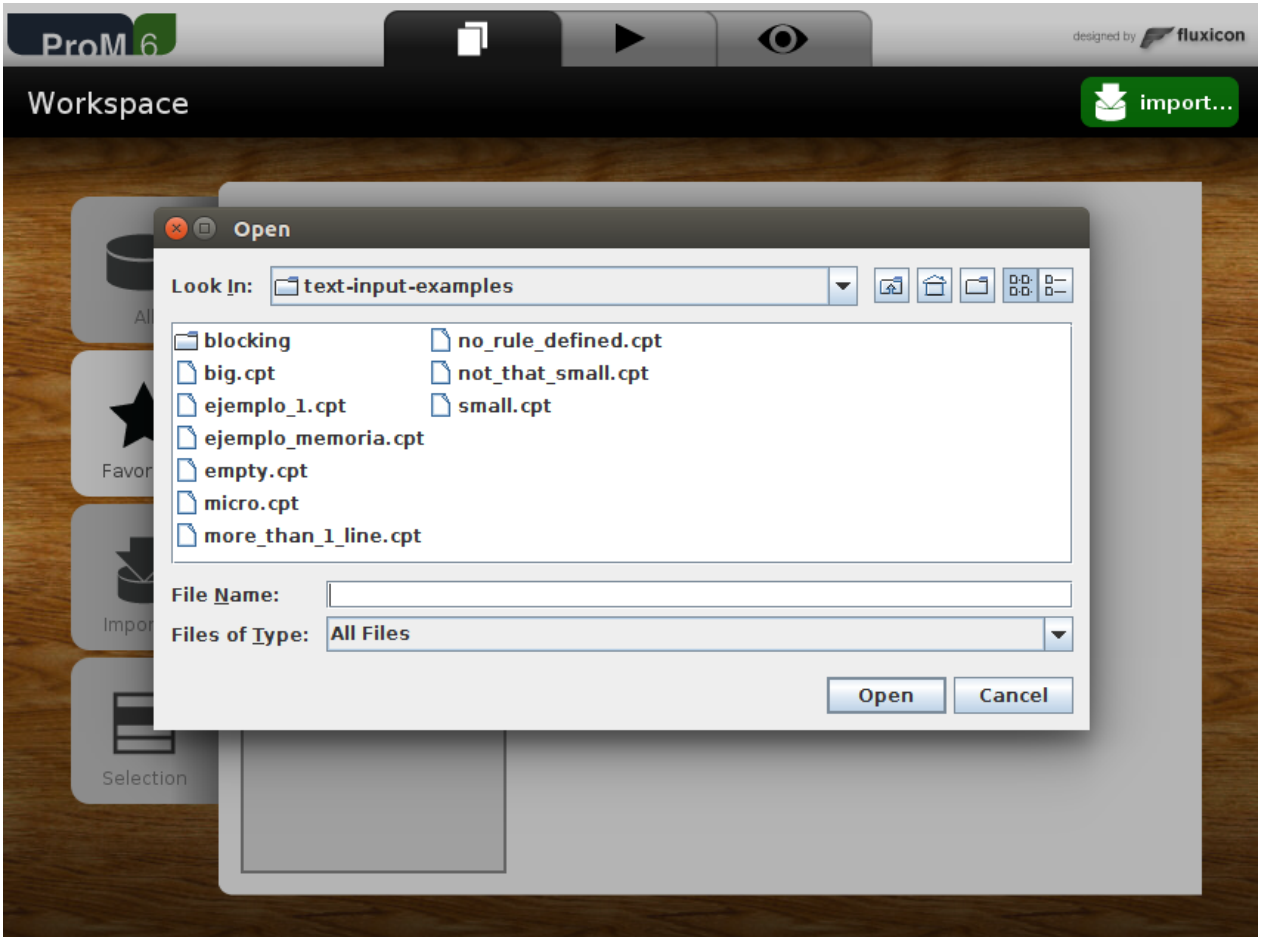


Figura 5.1: Cuadro de diálogo de sistema que se despliega al importar archivos locales al framework.

```
AND() [[OR][H]]
```

Código Fuente 5.1: Representación en texto de un CPT inconsistente, pues un operador $\wedge$ debe tener al menos un hijo.

Para la implementación de este plug-in, se cubrieron ciertos casos en que los archivos `cpt` estuviesen corruptos. Para ello se crearon cinco tipos de excepciones Java que se encargan de manejarlos, impidiendo que el framework importe estos archivos. Además se despliega un mensaje de error en pantalla, retroalimentando al usuario sobre el tipo de problema que hubiere ocurrido, como se muestra en la figura 5.2.

EmptyFileException se lanza cuando se intenta importar un archivo de extensión `cpt` sin contenido. El mensaje de error muestra el nombre del archivo, indicando que éste está vacío.

MoreThanOneLineFileException maneja el caso en que el archivo contiene más de una línea. Por ahora el paquete no soporta la definición de múltiples árboles en un solo archivo.

IncorrectCPTStringFormat indica que el la representación en texto del CPT a importar tiene un formato incorrecto. Esto puede ser causado por parentización incorrecta, número de configuraciones que no coinciden con la estructura del árbol, uso de delimitadores no reconocidos, etc.

RuleNotFoundException se lanza cuando no se ha encontrado un predicado que reconozca el tipo de nodo a representar. Esto ocurre cuando no se ha encontrado una regla que satisfaga la secuencia de caracteres al momento de construir determinado tipo de nodo.

IncosistentCPTException maneja el caso en que, una vez construido el CPT, éste no cumpla con las reglas de consistencia de un árbol de proceso configurable. Para ello se ejecuta el método '`isConsistent`' descrito en la subsección 4.2.1. Un ejemplo de un error de este tipo se observa en el código fuente 5.1.

Solo una vez que el proceso de importación ha finalizado de manera correcta, el objeto Java que representa un CPT dentro de ProM6 aparecerá como disponible en el listado de recursos de la vista del workspace.

5.2. Visualización de un CPT en ProM6

Para desarrollar esta funcionalidad se implementó un plug-in utilizando la notación `@Visualizer`, como se explicó en la subsección 2.3.2. El método que se define toma un objeto de tipo `ConfigurableProcessTree` y retorna un `JComponent`, el cual el framework sabe desplegar en la vista de visualización. Según lo descrito en la subsección 4.2.3, se utilizó la librería de Java Swing, JGraphX, para convertir el modelo de un CPT a su representación visual.

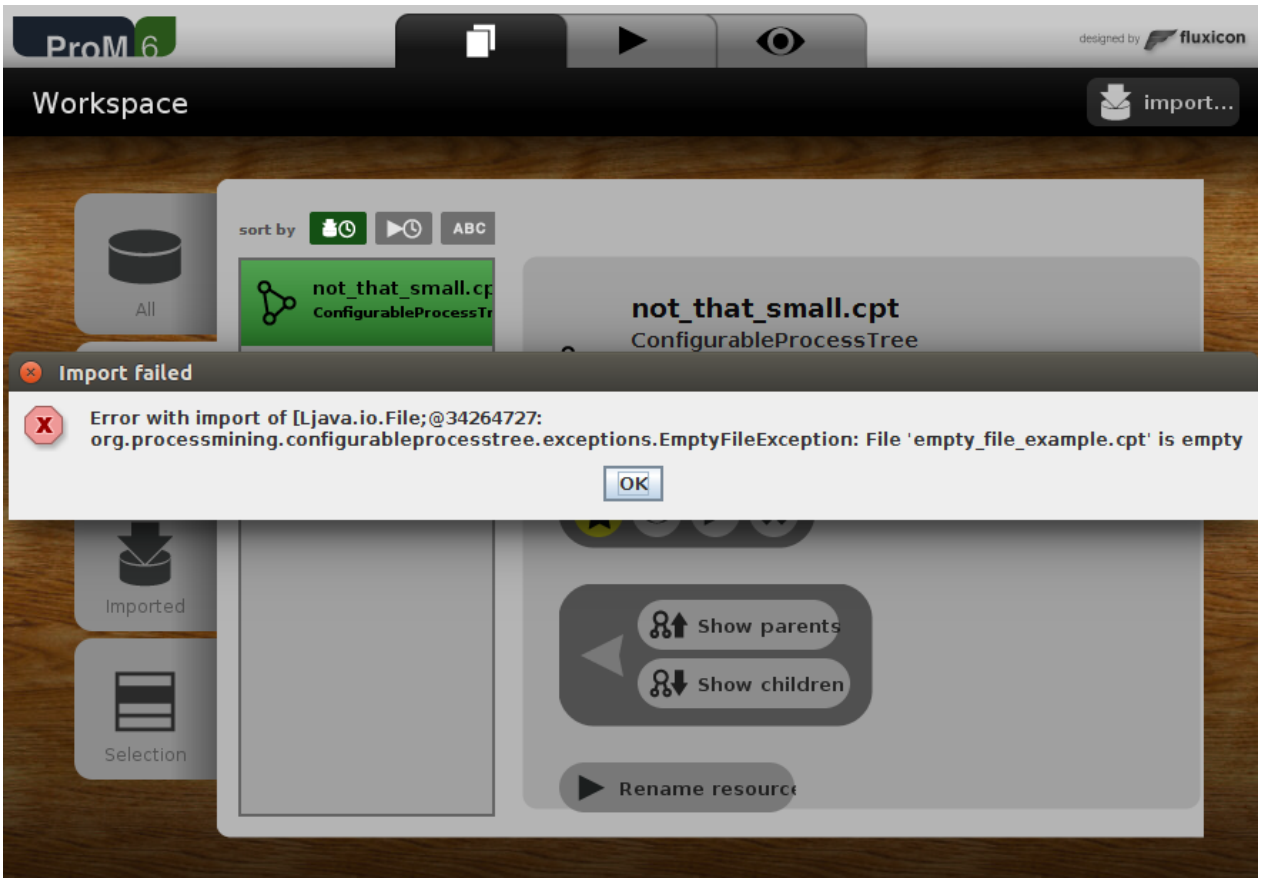


Figura 5.2: Mensaje de error desplegado por ProM6 cuando hay un problema al momento de importar un archivo al framework.

JGraphX utiliza su propia estructura de datos, `mxGraph`, para representar nodos y arcos de un grafo, y a partir de este modelo, se encarga de crear el layout adecuado. Este modelo permite al usuario manipular variadas configuraciones sobre el diagrama, como comportamiento, interactividad con el cliente, estilos, tipo de layout, etc. La ventaja de esto, es que JGraphX se encarga completamente de la disposición de los elementos en el panel al momento de la renderización del diagrama. Esto libera al usuario de tener que hacer cualquier tipo de ajuste manual o cálculos para la distribución espacial de los componentes. El usuario solo debe preocuparse de integrar su propio modelo al de la estructura de datos propia de la librería, y definir las configuraciones y estilos bajo las que el diagrama debe visualizarse.

Para construir este objeto de tipo `mxGraph` a partir del modelo de un CPT, cada nodo `CPTNode` implementa el método `addToGraphModel`, el cual recibe como uno de sus parámetros al objeto `mxGraph`. Cada nodo se inserta a sí mismo en el grafo, y de manera recursiva llama al método sobre sus nodos hijos, para que éstos se inserten al modelo también.

Una vez insertados todos los nodos del CPT a la estructura de la librería, simplemente se debe instanciar la clase `mxGraphComponent` de la librería. Ésta construye el componente de Java Swing que visualiza el diagrama a partir del modelo descrito en el objeto `mxGraph`. Éste componente de Java Swing es el que ProM6 sí sabe cómo manipular y desplegar en pantalla a través de la interfaz gráfica.

Para visualizar CPTs que estén disponibles dentro de ProM6, se debe seleccionar uno de la lista, en la vista de workspace, y presionar el botón para visualizarlo, como se muestra en la figura 2.11. ProM6 sabe qué plug-in visualizador ejecutar para tal tipo de objeto (`ConfigurableProcessTree`) gracias a la anotación Java `@Visualizer` antes mencionada. El framework redirige al usuario a la pestaña de visualización de la interfaz gráfica y despliega la instancia de la clase `JComponent` que ha sido retornada por el plug-in.

Un ejemplo de cómo es la visualización de un CPT desde ProM6 se observa en la figura 5.3. Allí se muestra el árbol generado por el plug-in de visualización que representa el que ha sido mostrado en la figura 2.2b anteriormente. Los nodos `c`, `d` y `b` han sido reemplazados por los números 1, 2 y 3 respectivamente, dados los requerimientos del parser explicados en la subsección 4.2.2.

5.3. Obtención de variantes de un CPT

En el contexto de este trabajo, una variante de un árbol de proceso configurable se representa como un CPT también, pero sin configuraciones posibles. Visualmente, éstos son equivalentes a un árbol de proceso, como el que se ha mostrado en la figura 2.2a anteriormente. La manera en que un CPT sin etiquetas de configuración se representa en texto, es incluyendo una sola posible variante para cada nodo, correspondiente a un '−'. Como se explicó previamente, un nodo con esta etiqueta indica que para esa variante el nodo permanece igual. Así, se define una determinada configuración de un CPT como un árbol de proceso configurable también, pero con una sola variante, la cual no indica cambios de tipo en ningún nodo. Un ejemplo de representación en texto de un CPT sin configuraciones se observa en

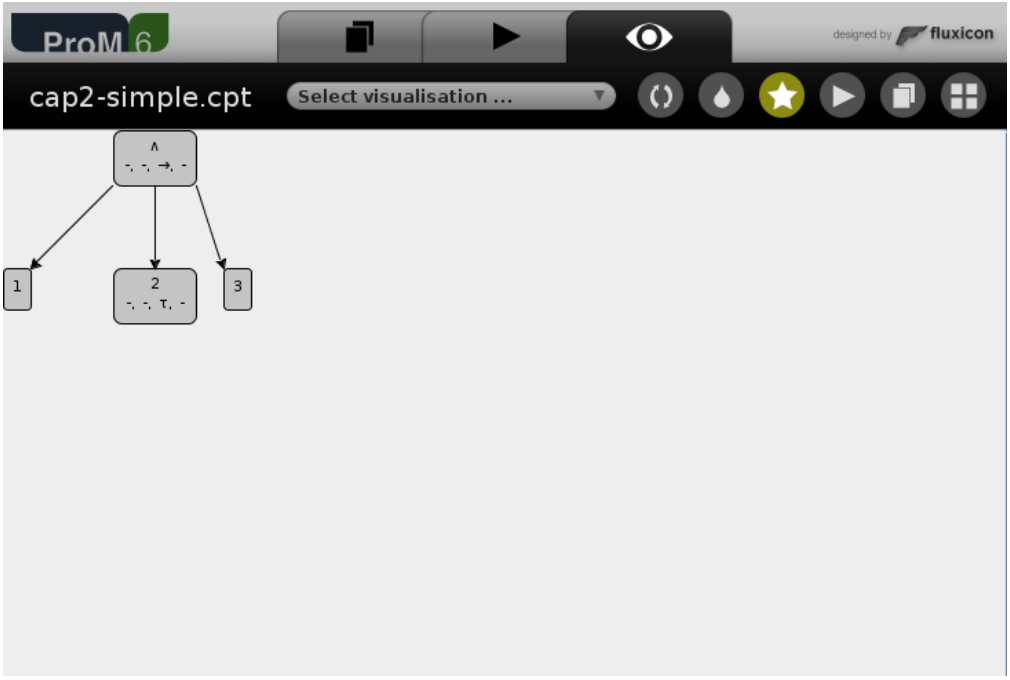


Figura 5.3: Visualización de un CPT desde la interfaz de ProM6, equivalente al de la figura 2.2b.

```
SEQ(LEAF:1, LEAF:2, LEAF:3, XOR(LEAF:4, LEAF:5)) [[-,-,-,-,-,-,-]]
```

Código Fuente 5.2: Representación en texto de un CPT sin configuraciones.

el código fuente 5.2. El árbol posee siete nodos, y una sola configuración entre '`[]`', donde ninguno de sus elementos cambia de tipo.

Un árbol sin configuraciones también se puede visualizar en ProM6 como se ha explicado en la sección anterior, salvo que ninguno de sus nodos muestra posibles variantes. A pesar de que se representa cada nodo con una sola posible configuración '`-`', ésta no se muestra, para que el modelo no sea redundante.

En la figura 5.4 se observa la visualización del CPT sin configuraciones correspondiente al código fuente 5.2. Este árbol es equivalente al árbol de proceso no configurable de la figura 2.2a.

Para obtener todas las variantes posibles de un CPT en ProM6, se se debe ejecutar el plug-in "Apply Configurations of a Configurable Process Tree", el cual recibe como parámetro un objeto de tipo `ConfigurableProcessTree`.

Dado que se implementó utilizando la anotación Java `@UITopiaVariant`, éste es el único de todos los otros plug-ins creados que se puede encontrar en el listado de acciones del framework (ver figura 5.5). Se muestra el nombre del plug-in, el autor y su correo, y en la parte inferior se muestra información adicional sobre éste, como su categoría y una descripción de lo que lleva a cabo. Toda esta información se provee al momento de definir el método que implementa el plug-in, específicamente en la anotación Java utilizada. Obsérvese en la figura que el plug-in

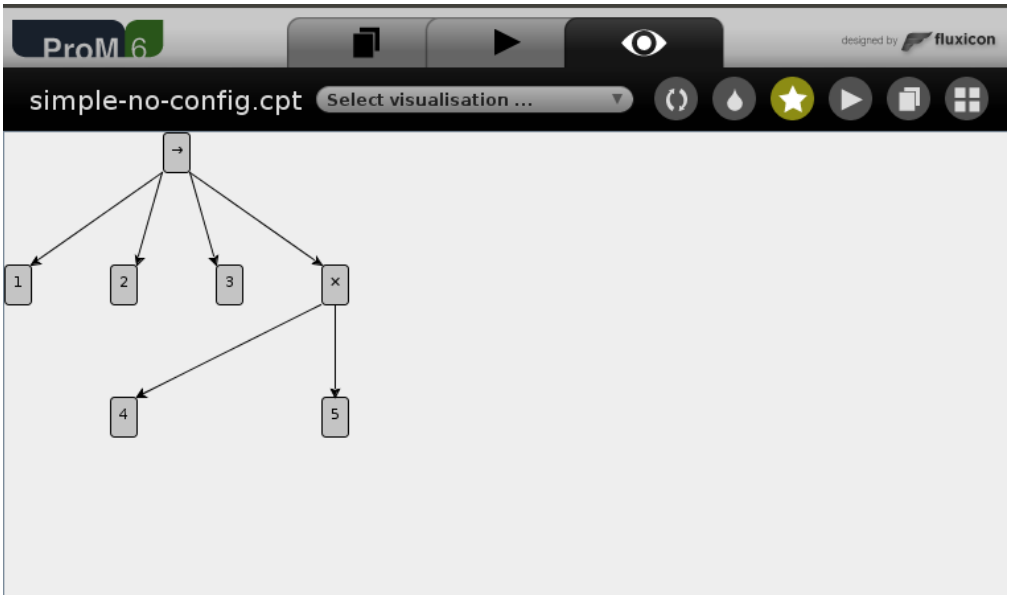


Figura 5.4: Visualización de un árbol de proceso (no configurable) desde la interfaz de ProM6, equivalente al de la figura 2.2a.

está destacado en color amarillo en la herramienta pues no se ha asignado un recurso como input para poder ejecutarlo. Al hacer click en el campo de input en la sección izquierda se puede seleccionar un objeto `ConfigurableProcessTree` para ejecutar el plug-in y luego ejecutarlo presionando el botón 'Start' de abajo a la derecha, en color verde. Más screenshots que ilustran estos pasos se pueden encontrar en las figuras B.2 y B.3 del apéndice.

El plug-in funciona ejecutando el método `configure` perteneciente a la clase `ConfigurableProcessTree` (ver diagrama de clase en la figura 4.4). Éste, por su parte, ejecuta `applyConfiguration` (ver subsección 4.2.1) a la raíz del árbol de proceso configurable sucesivas veces, una para cada una de de las n variantes posibles del CPT, y se obtiene un arreglo con todas las configuraciones obtenidas. Finalmente, el plug-in se encarga de guardar cada uno de ellos en el workspace, apareciendo en la lista de recursos disponibles como se observa en la figura 5.6. Cada variante tiene el mismo nombre que el CPT del cual se generó, acompañado de un número identificador correspondiente a su índice.

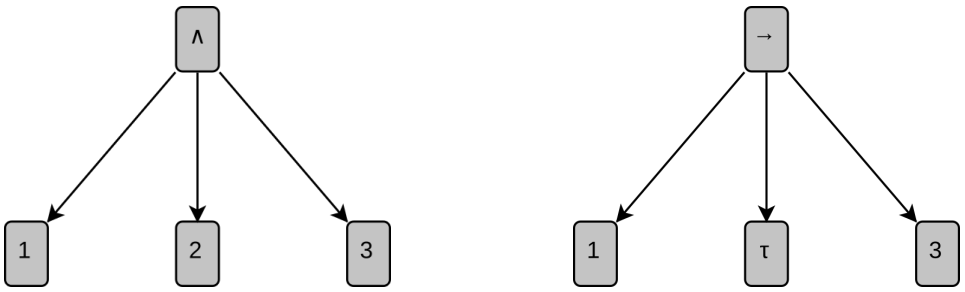
Como se mencionó anteriormente, dado que las variantes obtenidas después de ejecutar el plug-in también son representadas como objetos de tipo `ConfigurableProcessTree`, éstas también pueden ser visualizadas en ProM6. En la figura 5.7 se muestran las variantes obtenidas del CPT de la figura 5.3 a través del plug-in de ProM6.

5.4. Exportación de un CPT a archivos locales

Para desarrollar el plug-in de exportación se utilizó la anotación `@UIExportPlugin`, la cual, entre otras cosas, especifica la extensión del archivo al cual se convertirá el objeto en cuestión. El método que lo define recibe como parámetro el objeto a exportar, en este caso, uno de tipo `ConfigurableProcessTree`, y un objeto `File` nuevo. Este último ya viene preconfigurado

Figura 5.5: Plug-in para obtener las variantes de un CPT aparece en el listado de acciones disponibles en el framework.

Figura 5.6: Configuraciones resultantes después de ejecutar el plug-in que los obtiene se guardan en el listado de recursos de ProM6.



(a) Primera, segunda y cuarta variantes.

(b) Tercera variante.

Figura 5.7: Variantes del CPT de la figura 5.3 generadas por el plug-in implementado.

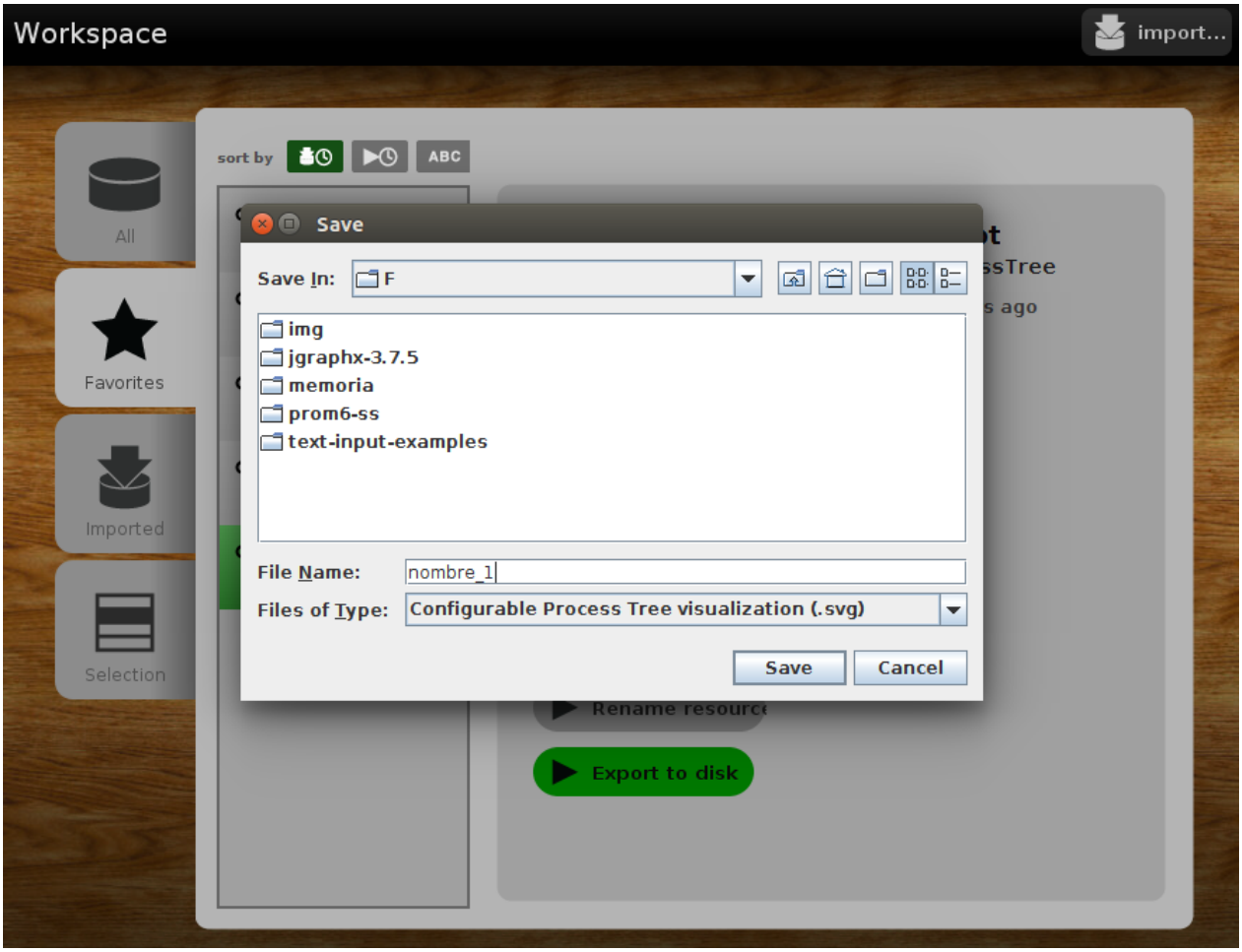


Figura 5.8: Diálogo de sistema desplegado al exportar a un archivo local un objeto en el workspace de ProM6.

por el framework con la información proveída por el usuario al momento de ejecutar el plug-in, como el directorio donde se guardará y la extensión del archivo. En la figura 5.8 se muestra la ventana de diálogo de sistema que se despliega al exportar un objeto del workspace a un archivo local. Dado que se ha escogido un objeto de tipo `ConfigurableProcessTree` para exportar, en la ventana de diálogo se permite solo guardar archivos de extensión `svg`. Esto es así pues la firma del método que define el plug-in expone el tipo de objetos que soporta, y eso permite a ProM6 ofrecer al usuario el plug-in que le corresponde al objeto seleccionado.

Para generar el documento SVG, primero se debe construir la misma estructura de datos `mxGraph` descrita en la sección 5.2, utilizando el método `addToGraphModel` sobre el CPT. Luego, simplemente se crea el objeto `mxSvgCanvas` propio de la librería, a partir del modelo construido anteriormente, lo cual genera un XML con el resultado deseado. Finalmente se escribe este contenido en el archivo vacío que provee el framework al ejecutar el plug-in.

Por otra parte, y dado que su implementación era simple, se incorporó otro plug-in que exportase un CPT a un archivo de extensión `cpt` con su representación en texto. Esto permite guardar localmente las variantes de CPT generadas a través de ProM6.

Cada nodo del CPT posee el método `structureTextRepresentation`, que de manera recursiva genera el string que representa la estructura del árbol, pues cada nodo sabe como representarse a sí mismo en texto. Luego se genera la representación de la configuración. En el caso de CPTs sin configuraciones, es decir, una variante, tan solo se construye una única representación de configuración, con tantos '−' como nodos tenga el árbol.

También se puede exportar un árbol que sí posee configuraciones a un archivo `cpt`. Por ahora, siempre que se tenga disponible un CPT en el framework es porque éste ha sido importado desde un archivo `cpt`. Para facilitar el proceso de exportación, durante el parseo del árbol se guarda la representación string de sus configuraciones directamente en un campo de la clase. Luego cuando se quiere exportar, simplemente se ejecuta el método para transformar la estructura del CPT a texto, y luego se concatena con estas configuraciones que se habían guardado previamente cuando se importó dicho CPT al framework.

Capítulo 6

Validación

En este capítulo se expone la manera en que se puede comprobar que la solución diseñada efectivamente ayuda a resolver los problemas que surgen al momento de manipular árboles de proceso configurables. Para demostrarlo, se estudió el desempeño del paquete de ProM6 implementado frente a distintos casos críticos que afectan al equipo de GEMS en sus quehaceres académicos.

Se realiza la validación analizando tres ejes de la solución propuesta al problema planteado en este trabajo: visualización automatizada de CPTs a partir de su representación en texto, aplicación de configuraciones y obtención de variantes, y exportación a archivos `svg`.

Se constata que la solución implementada constituye un aporte al trabajo realizado por el equipo de GEMS, según los requerimientos planteados.

6.1. Visualización partir de la representación en texto

Como se ha explicado anteriormente en el capítulo 3, los CPTs obtenidos por el equipo de GEMS a través de sus experimentos están representados usando una notación en texto específica. Esto dificulta y enlentece el proceso de análisis y comprensión de los resultados, dado que primero se hace necesario construir manualmente una representación gráfica de estos antes de poder estudiar la información obtenida.

Una vez implementado el parser y el visualizador de CPTs a través de plug-ins de ProM6, el trabajo de interpretación de resultados se reduce considerablemente, por el simple hecho de eliminar el paso intermedio de construcción manual de la visualización. Al automatizar la generación de una imagen que representa un CPT, el equipo de GEMS ahorra tiempo valioso al momento de analizar resultados.

Por ejemplo, la representación en texto que se muestra en el listado 6.1 es de un CPT real obtenido por el equipo de GEMS. Observando directamente el texto no se puede concluir mucha información, por cuanto ni siquiera se deducen las configuraciones individuales de

```
SEQ(LEAF:1, XOR(AND(LEAF:2, LEAF:3, LEAF:4), SEQ(LEAF:5, XOR(LEAF:42, LEAF:4),
LEAF:22, LEAF:3, LEAF:42)), XOR(LEAF:5, SEQ(LEAF:6, LEAF:7)))
[[-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-][-,-,-,-,-,-,-,-,-,-,-,B,-,-,-,-,-,-
,-,H][-,-,-,SEQ,-,H,-,-,-,-,-,-,-,B,-,-,-,-,-,-][-,-,-,-,-,-,-,-,-,-,-,-,-,-,-
,-,-,-,-,-]]
```

Código Fuente 6.1: Representación en texto de un CPT obtenido por el equipo de GEMS en sus experimentos.

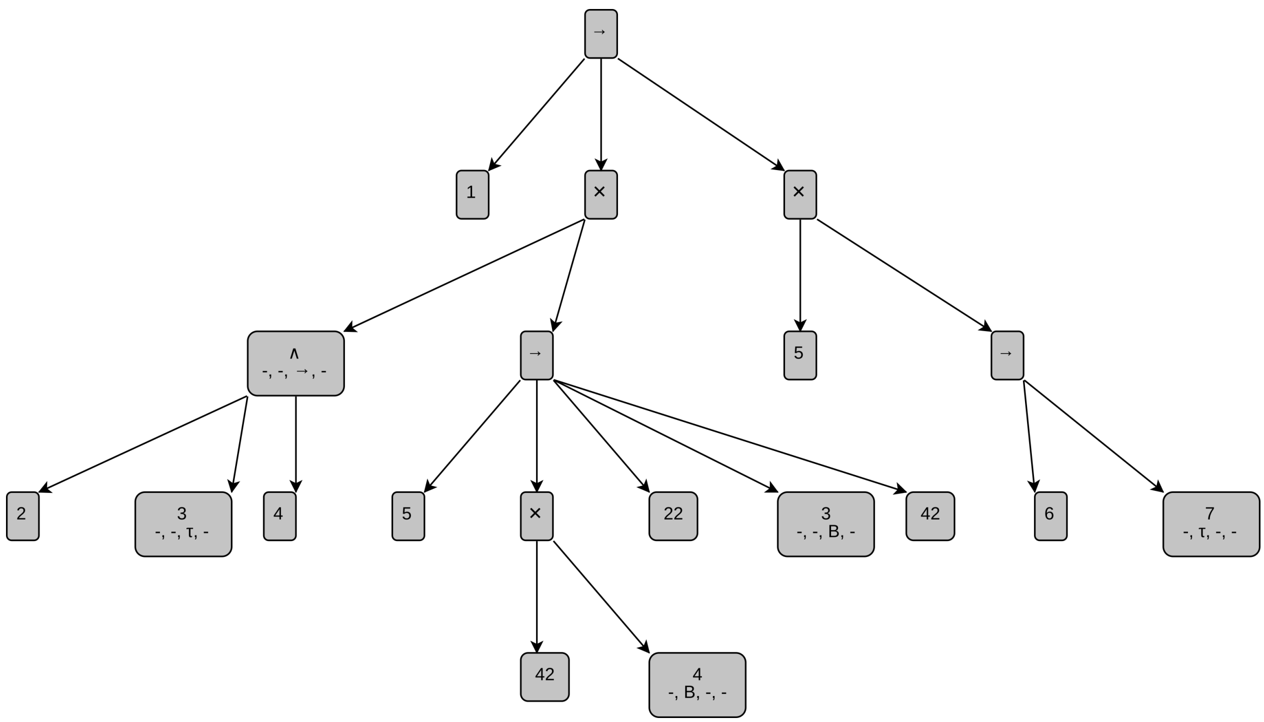


Figura 6.1: Visualización de CPT representado por el texto del listado 6.1, generado en ProM6.

cada nodo. Es claro que una representación gráfica ayudaría en este caso, una en la que se muestre la estructura convencional de un árbol y las etiquetas correspondientes a cada nodo.

La visualización del CPT correspondiente al texto mencionado se puede observar en la figura 6.1. Ésta fue generada a través de los plug-ins implementados, y en promedio el proceso de importación y generación de la visualización tomó 150 ms.

Es importante mencionar que las configuraciones de cada nodo ya no se visualizan como etiquetas externas. Éstas en cambio, se despliegan en el interior del nodo, evitando que hayan eventuales traslapes de etiquetas sobre los arcos, como es el caso que se observó en la figura 3.6. Esto mejora considerablemente la legibilidad de la información al momento de publicar y difundir resultados, que es una de las dificultades a las que se enfrenta el equipo de GEMS en su proyecto.

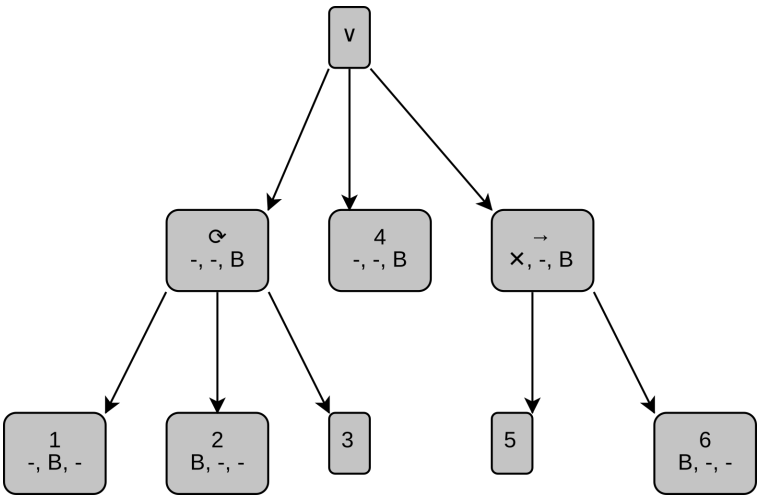


Figura 6.2: Ejemplo de un CPT con tres configuraciones.

6.2. Aplicación de configuraciones

Hasta ahora el equipo de GEMS no cuenta con ninguna herramienta que genere todas las variantes de un CPT de manera individual. El paquete de ProM6 desarrollado en este trabajo provee esta funcionalidad, aplicando, además de las configuraciones, las reglas de bloqueo y reducción pertinentes explicadas en la subsección 2.2. En la figura 6.2 se muestra un CPT con tres configuraciones, al cual se le aplicó el plug-in implementado para obtener sus variantes.

En la figura 6.3 se observa la primera variante obtenida. El plug-in además aplicó las reglas de bloqueo correctamente, como se observa en el nodo $\rightarrow$. Anteriormente éste era un $\circlearrowleft$, pero dado que en esta configuración su segundo hijo fue bloqueado (hijo 'redo'), su tipo debe cambiar a $\rightarrow$, como se muestra en las reglas de bloqueo en la figura 2.5b. Por otra parte, en esta variante el nodo $\rightarrow$ se configura a un operador $\times$, a la vez que su segundo hijo, la actividad 6 es bloqueada. Lo anterior provoca que, según las reglas de reducción aplicadas por el plug-in, la actividad 5 suba un nivel, reemplazando a su nodo padre. Esto, pues es redundante mostrar un operador $\times$ con un único hijo.

La figura 6.4 se muestra la segunda variante del CPT de la figura 6.2 generada por el plug-in. Obsérvese la desaparición del nodo $\circlearrowleft$, la cual fue provocada por la aplicación de las reglas de bloqueo, al configurarse su primer hijo ('do') como bloqueado para esta variante.

Finalmente en la figura 6.5 se observa la tercera variante del CPT de la figura 6.2. El árbol completo se ha configurado como escondido, según las reglas de bloqueo, puesto que los tres hijos de la raíz se bloquean en esta variante.

6.3. Exportación y post-procesamiento

El paquete implementado también provee la funcionalidad de exportación de visualizaciones de CPTs a imágenes `svg`, con la finalidad de poder ser post-procesados usando alguna

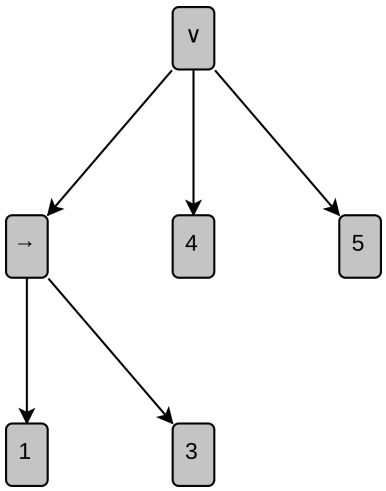


Figura 6.3: Primera variante del CPT de la figura 6.2.

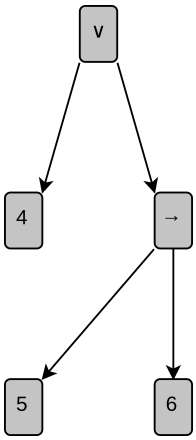


Figura 6.4: Segunda variante del CPT de la figura 6.2

Figura 6.5: Tercera variante del CPT de la figura 6.2

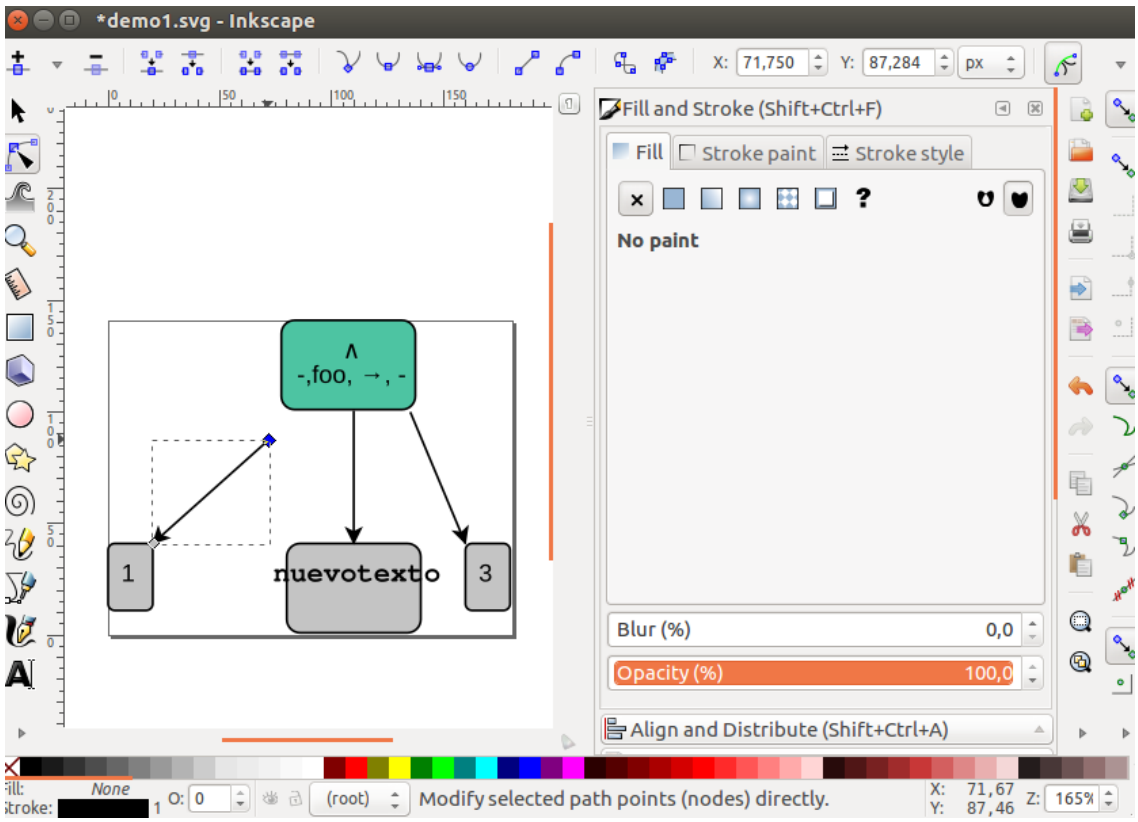


Figura 6.6: Imagen `svg` del CPT de la figura 5.3 siendo ajustada en el editor Inkscape.

herramienta externa.

Se comprueba que los archivos `svg` generados por ProM6 se han construido de manera correcta y consistente, dado que es posible abrirlos y modificarlos a través de un editor WYSIWYG[1] como Inkscape. En la figura 6.6 se muestra el mismo CPT de la figura 5.3 siendo editado en Inkscape, una vez exportado desde ProM6. Gracias a este formato, a través de este editor se pueden hacer ajustes más detallados sobre el árbol, como reposicionar elementos, cambiar estilo de los componentes (color, tamaño, fuentes, etc.), renombrar los nodos, entre otros.

[1]What You See Is What You Get `https://en.wikipedia.org/wiki/WYSIWYG`.

Conclusión

El equipo de GEMS del DCC se ve enfrentado a ciertas dificultades en el desarrollo de su investigación en el área de la minería de procesos. En particular, se presentan obstáculos al momento de interpretar analizar, visualizar los modelos de árboles de proceso configurables obtenidos como resultados de sus experimentos. En torno a ese contexto, el trabajo realizado durante esta memoria consistió en diseñar e implementar una herramienta que proveyera las funcionalidades requeridas por el equipo de GEMS para facilitar el desarrollo de sus proyectos de investigación.

Según lo descrito a lo largo del documento, se puede concluir que los objetivos específicos propuestos se han cumplido. Se diseño e implementó un plug-in capaz de procesar la codificación en texto de un CPT y construir a partir de éste un modelo propio de ProM6 que lo represente. Una vez creado este modelo dentro del framework, se permite al usuario visualizar una representación gráfica del CPT en cuestión, utilizando un layout adecuado y evitando el traslape de las configuraciones del árbol con otros elementos. Por su parte, se implementó otro plug-in que construyese todas las posibles variantes de un CPT, aplicando consistentemente las reglas de bloqueo pertinentes. Finalmente, se permite al usuario exportar las visualizaciones de los CPTs generados a través de ProM6, a archivos de formato `svg`, dando así la posibilidad de llevar a cabo un post-procesamiento de la representación gráfica utilizando algún editor de imágenes vectoriales.

Dado lo anterior, el objetivo general se ha cumplido. Las funcionalidades requeridas por el equipo de GEMS se han implementado mediante la construcción de un nuevo paquete para el framework ProM6. Los plug-ins incluídos permiten generar y manipular árboles de procesos configurables a partir de archivos de texto.

El desafío más interesante que hubo durante el desarrollo de este trabajo fue el hecho de enfrentarse a código de terceros. Durante el proceso de diseño y desarrollo se planteó un primer prototipo de solución que incorporaba y extendía un paquete de ProM6 ya existente. El momento crítico ocurrió cuando se tomó la decisión de comenzar el desarrollo de un segundo prototipo, desechando por completo la primera solución propuesta. Esto, pues se consideró que la librería original, por un lado, no estaba construida de una manera tal que permitiese su extensión con relativa facilidad, mostrando dificultades en la mantenibilidad de su código. Por otro lado, su implementación contemplaba demasiadas funcionalidades que no interesaban en el contexto del problema a solucionar en este trabajo, lo cual complejizaba aún más el desarrollo. El principal aprendizaje rescatado es que la implementación de todo código debe realizarse considerando siempre su extensibilidad y mantenibilidad, factor que

guió las líneas de desarrollo del segundo prototipo.

En retrospectiva, hubo algunos elementos que pudieron hacerse de mejor manera desde un comienzo. En particular hizo falta una metodología de testing que acompañara el proceso de desarrollo de la solución. Por desgracia, se encontró muy poca documentación sobre pruebas en ProM6, lo cuál significó tomar la decisión de no destinar más tiempo a ello. Sin embargo es un factor que definitivamente debe considerarse en desarrollos posteriores de esta herramienta.

Desde su inicio la solución final fue desarrollada en miras a que pudiese ser extendida, pues solo representa una base de partida para seguir desarrollando funcionalidades que permitan operaciones más complejas sobre árboles de proceso configurables. Variados otros plug-ins de ProM6 pueden ser creados a partir de la representación de CPTs propuesta en esta solución, los cuales pueden incorporar nuevas implementaciones de algoritmos de descubrimiento de procesos.

Por ahora el paquete implementado no permite la edición de CPTs dentro del mismo framework de ProM6. Sin embargo, la librería JGraphX para visualizar diagramas que se utilizó fue justamente escogida pensando en el desarrollo de un editor interno a la herramienta. JGraphX incluye variadas funcionalidades que permiten la interacción del usuario con la representación gráfica, lo cual podría sentar la base para próximos desarrollos.

Otra posible línea de trabajo futuro podría incluir la representación de CPTs usando el lenguaje DOT [24]. Esto permitiría la generación de visualizaciones de grafos utilizando herramientas mucho más poderosas y populares, como Graphviz [25].

Bibliografía

[1] Joos C. A. M. Buijs. *Flexible evolutionary algorithms for mining structured process models*. PhD thesis, Technische Universiteit Eindhoven, 2014.

[2] Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Mining configurable process models from collections of event logs. In *Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings*, pages 33–48, 2013.

[3] James L. Peterson. Petri Nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

[4] BPMN online quick guide. `http://www.bpmnquickguide.com/view-bpmn-quick-guide/`. [En línea], último acceso: Abril de 2017.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, Inc., 1985.

[6] Wil M. P. van der Aalst. Process discovery: Capturing the invisible. *IEEE Comp. Int. Mag.*, 5(1):28–41, 2010.

[7] Maikel L. van Eck, Joos C. A. M. Buijs, and Boudewijn F. van Dongen. Genetic Process Mining: Alignment-Based Process Model Mutation. In *Business Process Management Workshops - BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7-8, 2014, Revised Papers*, pages 291–303, 2014.

[8] Wil M. P. van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.*, 16(9):1128–1142, 2004.

[9] GEMS: Gestión Experimental de la mejora de Software. Proyecto FONDEF IDeA IT13I20010, CONICYT. *Departamento de Ciencias de la Computación, Universidad de Chile*. `https://www.dcc.uchile.cl/gems/`. último acceso: Abril de 2017.

[10] Fabian Rojas Blum, Jocelyn Simmonds, and María Cecilia Bastarrica. The *v-algorithm* for discovering software process lines. *Journal of Software: Evolution and Process*, 28(9):783–799, 2016.

[11] ProM Tools. `http://www.promtools.org`. último acceso: Abril de 2017.

[12] Inkscape, open-source vector graphics editor. https://inkscape.org/en/about/overview/. último acceso: Noviembre de 2017.

[13] Michael Westergaard. *ProM Tutorial. Intergalactic Michaelpedia of the World.* https://westergaard.eu/category/prom-tutorial/, [En línea] último acceso: Noviembre de 2017.

[14] *How to create plug-ins in ProM?* https://svn.win.tue.nl/trac/prom/wiki/setup/HowToCreatePluginsInProM, [En línea] último acceso: Julio de 2017.

[15] H.M.W. Verbeek. *Developing and running ProM plug-ins on your local computer.* http://www.win.tue.nl/~hverbeek/blog/2017/06/19/developing-and-running-prom-plug-ins-on-your-local-computer/, último acceso: Julio de 2017.

[16] Workshop on Programming with/in ProM. https://svn.win.tue.nl/trac/prom/wiki/Workshop. Hasselt, Belgium, Sept 28, 2012.

[17] Template de ProM6 para nuevos paquetes. https://svn.win.tue.nl/repos/prom/Packages/NewPackageIvy/Trunk. último acceso: Julio de 2017.

[18] Apache Ivy package manager. http://ant.apache.org/ivy/. último acceso: Julio de 2017.

[19] *Documentación de paquete `ProcessTree` de ProM6.* https://svn.win.tue.nl/trac/prom/browser/Documentation/ProcessTree, último acceso: Noviembre de 2017.

[20] Till Tantau. *The TikZ and PGF Packages Manual for version 3.0.1a.* Institut für Theoretische Informatik, Universität zu Lübeck, 8 2015. http://mirror.utexas.edu/ctan/graphics/pgf/base/doc/pgfmanual.pdf, último acceso: Noviembre de 2017.

[21] Java Swing graph visualization library. https://github.com/jgraph/jgraphx. último acceso: Octubre de 2017.

[22] *JGraphX (JGraph6) User Manual.* https://jgraph.github.io/mxgraph/docs/manual_javavis.html, último acceso: Octubre de 2017.

[23] *mxGraph 3.8.0 API Specification.* https://jgraph.github.io/mxgraph/java/docs/index.html, último acceso: Noviembre de 2017.

[24] The DOT Language. https://graphviz.gitlab.io/_pages/doc/info/lang.html, último acceso: Diciembre de 2017.

[25] Graphviz - Graph Visualization Software. http://www.graphviz.org/about/, último acceso: Diciembre de 2017.

Apéndice A

Extractos de código del paquete `ProcessTree`

En el listado A.1 se observa la implementación de un ejemplo básico de un plug-in de ProM6. Su nombre es "My Hello World Plugin", no recibe ningún input y retorna el string "Hello World"

En el listado A.2 se muestra el constructor de la clase que representa un CPT en el paquete `ProcessTree`. Se destaca el gran bloque condicional que construye distintos tipos de nodos utilizando typechecking de manera intensiva.

Por su parte, en el listado A.3 también se observa otro extracto de código de la implementación del paquete `ProcessTree`, donde se observa poca mantenibilidad. El code smell detectado aquí es el de 'God Class', pues un solo método es el que se encarga de renderear todos los distintos tipos de nodos que pueden existir en un CPT.

```
1  public class HelloWorld {
2          @Plugin(
3                  name = "My Hello World Plugin",
4                  parameterLabels = {},
5                  returnLabels = { "Hello world string" },
6                  returnTypes = { String.class },
7                  userAccessible = true,
8                  help = "Produces the string: 'Hello world'"
9          )
10         @UITopiaVariant(
11                 affiliation = "My company",
12                 author = "My name",
13                 email = "My e-mail address"
14         )
15         public static String helloWorld(PluginContext context) {
16                 return "Hello World";
17         }
18 }
```

Código Fuente A.1: Ejemplo de plug-in básico que retorna el string "Hello World".

```
public ProcessTreeImpl(ProcessTree tree){
        super(tree);
        HashSet<Manual> manualTasks = new HashSet<Manual>();
        HashSet<Block> blocks = new HashSet<Block>();
        for(Node n: tree.getNodes()){
                Node np = null;
                if(n instanceof Seq){
                        np = new AbstractBlock.Seq((Seq)n);
                }
                if(n instanceof And){
                        np = new AbstractBlock.And((And)n);
                }
                if(n instanceof Xor){
                        np = new AbstractBlock.Xor((Xor)n);
                }
                if(n instanceof Def){
                        np = new AbstractBlock.Def((Def)n);
                }
                if(n instanceof Or){
                        np = new AbstractBlock.Or((Or)n);
                }

        (...)

                if(n instanceof Block){
                        blocks.add((Block) n);
                }
                if(n instanceof Manual){
                        np = new AbstractTask.Manual((Manual)n);
                        manualTasks.add((Manual)n);
                }
                if(n instanceof Automatic){
                        np = new AbstractTask.Automatic((Automatic)n);
                }

        (...)
}
```

Código Fuente A.2: Constructor de un árbol de procesos. Obsérvese el uso de un gran y solo bloque condicional.

```
protected void paintComponent(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;

        GeneralPath gatewayDecorator = new GeneralPath();
        String path = "";
        double s = (getWidth() - 2.0 * borderWidth) / 33.;
        switch (type) {
                case AND :
                        path = "M 8.5 15 L 8.5 17 L 15 17 L 15 23.5 L 17 23.5" +
                        "L 17 17 L 23.5 17 L 23.5 15 L 17 15 L 17 8.5" +
                        "L 15 8.5 L 15 15 L 8.5 15";
                        break;
                case OR :
                        path = "M 8.5,16 L 23.5,16 M 16,8.5 L 16,23.5 M 10.5,10.5 L" +
                        "21.5,21.5 M 10.5,21.5 L 21.5,10.5";
                        break;
                case LOOPDEF :
                        g2d.drawArc((int) (5 * s), (int) (5 * s),
                            (int) (25 * s), (int) (25 * s), -30, 240);
                        path += "M 26 24 L 31 19 L 25 17.5 L 26 24 ";

                (...)

                case SEQ :
                        path = "M 4 14 L 22 14 L 22 11 L 30 16.5 L 22 22" +
                        "L 22 19 L 4 19 L 4 14 ";
                        break;
                case TIMEOUT :
                        path = "M 17 8 L 16 16 L 24 16";
                        break;
                case PLACEHOLDER :
                case AUTOTASK :
                case MANTASK :
                        break;
        }
(...)
}
```

Código Fuente A.3: Método que renderea un árbol de proceso, usando un gran y solo bloque switch. Se observa un comportamiento de 'God class' en esta clase, dado que un solo método se encarga de dibujar todos los tipos de nodos.

Apéndice B

Screenshots adicionales de la interfaz gráfica de ProM6

En figura B.1 se observa la vista de acciones disponibles en ProM6. Encerrado en un óvalo rojo se destacan las pestañas de navegación que permiten moverse de una vista a otra en la interfaz.

En la figura B.2 se muestra el cuadro de diálogo desplegado por el framework que permite escoger uno de los recursos disponibles en el sistema para utilizar como input de algún plug-in que haya sido seleccionado previamente.

En la figura B.3 se ve la lista de plug-ins disponibles en ProM6. Uno de ellos está pintado de color verde, pues es el único que puede ser aplicado cuando se tiene como input un objeto de tipo `ConfigurableProcessTree`. Los demás están en color rojo puesto que no aceptan ese tipo de input. Para ejecutar el plug-in utilizando el input proveído, se debe hacer click en el botón de 'Start' abajo a la derecha, en color verde.

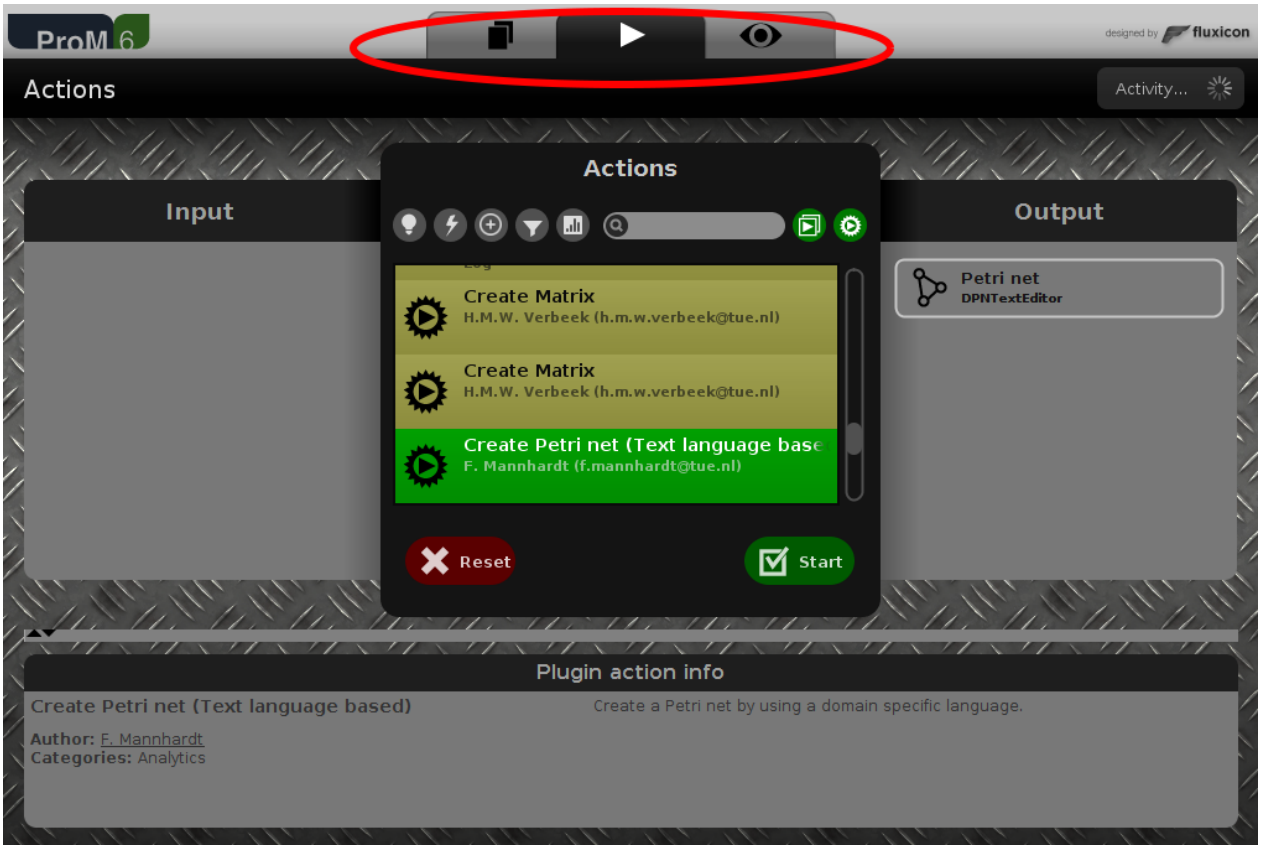


Figura B.1: Pestañas de navegación entre las tres vistas principales de ProM6: workspace, plug-ins y visualizaciones (de izquierda a derecha)

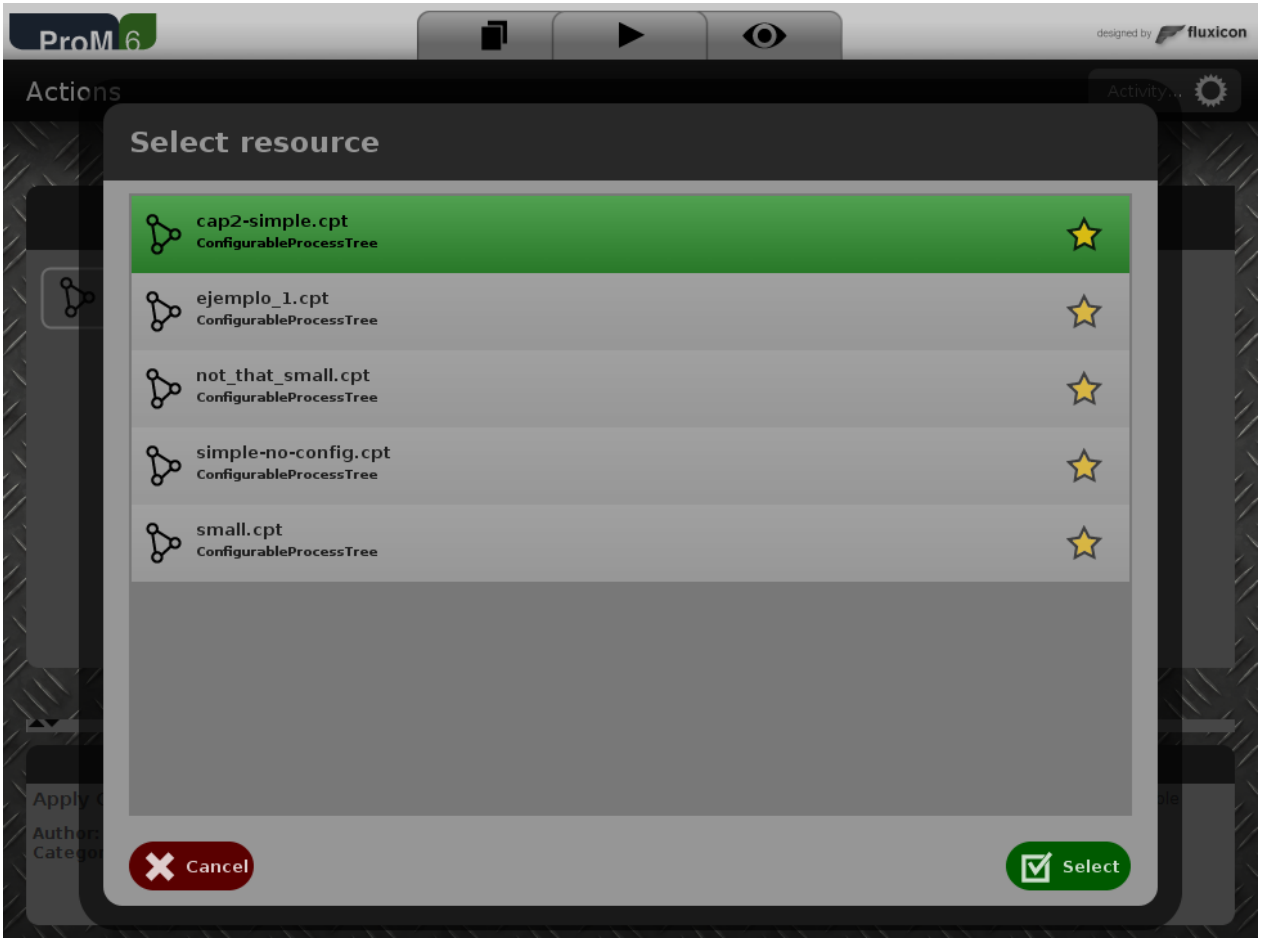


Figura B.2: Listado de recursos disponibles que se despliega al hacer click en campo de input que requiere un plug-in para ejecutarse.

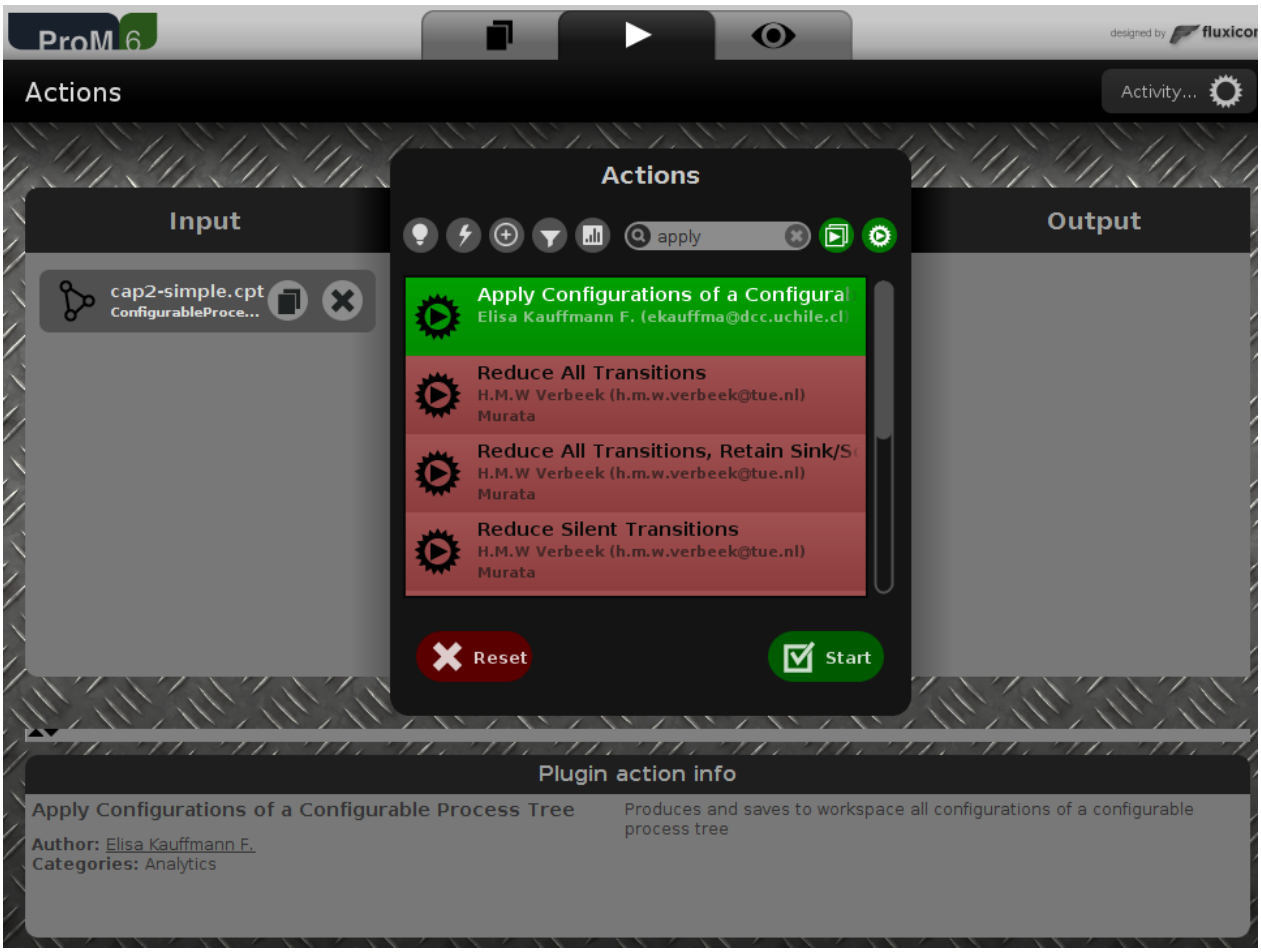


Figura B.3: Una vez seleccionado el input a utilizar se muestran en color verde todos los plug-ins que se pueden ejecutar dada la cantidad y tipo de inputs que se han proveído. En este caso, solo el plug-in que obtiene las variantes de un CPT puede utilizarse cuando el input es de tipo `ConfigurableProcessTree`.