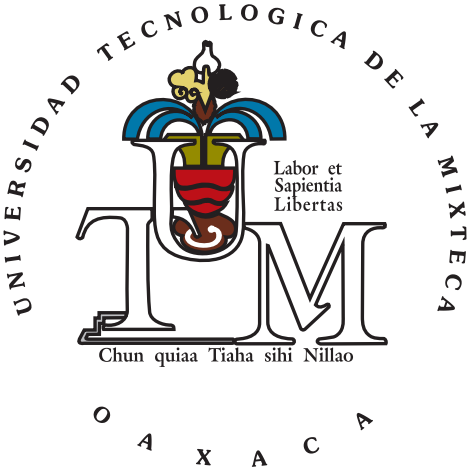


UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

Diseño e implementación de un plug-in para el modelado de diagramas de flujo de tareas

TESIS
PARA OBTENER EL TÍTULO DE:
INGENIERO EN COMPUTACIÓN

PRESENTA:
José Angel Quintanar Morales

DIRECTOR DE TESIS:
Dr. CARLOS ALBERTO FERNÁNDEZ Y FERNÁNDEZ

Huajuapan de León, Oaxaca. Mayo 2013

Dedicatoria

A mis padres.

Por ser el pilar fundamental en todo lo que soy. En especial a mi madre, por darme la vida, quererme mucho, creer en mi y apoyarme en todo momento.

A mis hermanas Ana, Lucia y en especial a mi hermano Gabriel por apoyarme siempre, los quiero mucho.

A Lucero, espero ser un ejemplo a seguir.

A mis amigos.

Por el apoyo mutuo en este largo y difícil camino. Aunque por ahora nos encontremos lejos, seguimos siendo amigos.

Todos aquellos familiares y amigos que no recordé al momento de escribir esto. Ustedes saben quiénes son.

Agradecimientos

Agradezco principalmente a mi director de tesis, el Dr. Carlos Alberto Fernández y Fernández, por todo el tiempo dedicado así como sus consejos y sobre todo su paciencia casi infinita para ayudarme a concluir este trabajo.

De igual manera le doy las gracias a mis sinodales M.C. Wendy Yaneth García Martínez, M.C. David Martínez Torres, Dr. Moisés Homero Sánchez López y al M.A.E. Rodolfo Maximiliano Valdés Dorado por su tiempo y disposición al revisar este trabajo, así como su valiosa contribución para el mejoramiento del mismo.

A mis maestros.

Todos y cada uno de ellos que con sus enseñanzas marcaron cada etapa de mi camino universitario.

Finalmente a las personas que contribuyeron de una u otra forma en la conclusión de este trabajo.

Resumen

La verificación formal demuestra que el programa desarrollado satisface su especificación. Pero se tiene el argumento en contra de la necesidad de requerir notaciones especializadas. Hoy en día existen herramientas que facilitan el desarrollo de software, la herramienta dominante es UML, sin embargo, presenta algunos inconvenientes, sobre todo porque no está fundamentado en una notación formal. Una alternativa es el Método *Discovery*, el cual también es una metodología para el desarrollo orientado a objetos, una limitante en su utilización y difusión, es que no existen herramientas de software como en el caso de UML. Por ello, en el presente trabajo se propone el diseño e implementación de un plug-in de Eclipse, enfocado a generar el álgebra de tareas a partir de diagramas de tareas.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Modelado de sistemas

El Modelado de Sistemas de Software es una técnica utilizada para disminuir la complejidad propia a este tipo de sistemas. Una de las principales razones para su utilización, es que ayuda a los ingenieros de software a visualizar el sistema a implementar; otra gran ventaja es que, si representa un grado de abstracción mayor, puede ser presentada a los clientes y de esta manera tener una comunicación eficiente con ellos. Por último, las herramientas de modelado y las de Ingeniería de Software Automatizada, pueden ayudar a verificar la corrección del modelo [8].

Un modelo de ingeniería eficaz debe satisfacer ciertas características:

- Abstracto: enfatiza a los elementos importantes y oculta a los irrelevantes.
- Comprensible: fácil de comprender por los observadores.
- Preciso: representa de forma fiel el sistema que modela.
- Predictivo: se pueden usar para deducir conclusiones sobre el sistema que modela.
- Barato: mucho más barato y sencillo de construir que el sistema que modela

Sin embargo, para detectar errores u omisiones en el diseño, antes de comprometer recursos para la implementación se requiere analizar y experimentar, investigar y comparar soluciones alternativas.

1.2. Verificación formal.

Un modelo debe ser una representación de la realidad, pero, ¿Cómo asegurar que un sistema de cómputo realiza sólo las funciones para las cuales fue implementado?, algo para asegurarnos de ello sería la automatización de la verificación formal [4].

El área de Verificación formal estudia los fundamentos teóricos y la implementación de técnicas de verificación de los sistemas de cómputo. Consta de tres etapas [30, 31]:

- Modelación: Se construye un modelo matemático de los posibles comportamientos del sistema.

- Especificación: Se especifica en un lenguaje formal el comportamiento deseables del sistema.
- Verificación: Se encarga de conocer si el modelo satisface la especificación.

En general, la verificación formal demuestra que el programa desarrollado satisface su especificación, por lo que los errores de implementación no comprometen la confiabilidad.

El argumento en contra del uso de la verificación formal es que requiere notaciones especializadas, las cuales sólo pueden ser utilizadas por personal entrenado y pueden no ser comprensibles por todos los expertos del dominio [16]. Por lo tanto, los problemas con los requerimientos del sistema pueden estar cubiertos por la formalidad. Los Ingenieros de Software no reconocen dificultades potenciales con los requerimientos, debido a que no tienen una compresión global del dominio. Además, la aplicación manual de los métodos formales es larga, tediosa y susceptible a los errores humanos.

Verificar un software no trivial consume una gran cantidad de tiempo y requiere tradicionalmente herramientas especializadas, tales como demostradores de teoremas y expertos matemáticos. Por lo tanto, es un proceso relativamente caro[40]. En consecuencia, gran cantidad de desarrolladores piensan que la verificación formal no es muy rentable.

Adicionalmente se puede lograr un nivel aproximado de confianza en el sistema de forma relativamente económica, utilizando otras técnicas de validación como las inspecciones y pruebas de sistemas. Sin embargo, requiere mayor tiempo debido a que estas pruebas no pueden automatizarse de manera óptima, por lo regular el proceso se extiende, es decir, suele superar los costos de la verificación formal [31, 37].

1.3. Lenguaje Unificado de Modelado

Actualmente para el desarrollo de software se utilizan herramientas de diseño, una de las más populares es UML (*Unified Modeling Language*) por sus siglas en inglés, en gran parte debido a que unifica principalmente los métodos de Booch, *Rumbaugh* OMT (*Object-Modeling Technique*) y Jacobson.

Es en octubre de 1994 cuando oficialmente surge el UML de *Rational Corporation*. La idea original fue unificar los modelos existentes de la época. Es hasta noviembre del año 1997 cuando el OMG (*Object Management Group)* por sus siglas en inglés, acepta formalmente a UML como un estandar[7].

La razón principal de la concepción de UML, fue establecer un lenguaje de modelado independiente a cualquier otro método [19]. La característica principal que lo hace atractivo a los desarrolladores de software es la forma gráfica con la cual se crean sus diagramas, pero sobre todo el conocimiento y manejo de esos diagramas que poseen debido a la gran difusión de UML. Dichos diagramas se pueden utilizar en una gran variedad de aplicaciones y procesos diferentes al diseño y desarrollo de software[20, 33]. A ultimas fechas, se han diseñado bases de datos orientadas a objetos completamente modeladas en UML, facilitando así la comprensión del esquema [26].

UML define un total de 13 diagramas así como el significado de los mismos [12]. La razón por la cual existen estos diagramas es mejorar la comunicación en el proceso de desarrollo del software, ya que todos los involucrados tienen una misma notación[12, 19].

La ventaja principal de UML sobre otras notaciones Orientadas a Objetos, es que elimina la diferencia entre semánticas y notaciones[19]. Sin embargo, no resuelve todos los problemas, otra notación podría subsanar por completo esta problemática.

1.4. Método Discovery.

El Método Discovery (MD) es una Metodología Orientada a Objetos utilizada principalmente para el Modelado de Sistemas de Negocios. Utiliza una notación simple y consistente, en algunos casos similar a la descrita en UML (esto para aprovechar el conocimiento previo sobre la notación), conservando su propósito original.

El método Discovery es una metodología para el desarrollo orientado a objetos propuesta formalmente en 1998 por Anthony J. H. Simons [16, 34], el autor lo considera un método centrado en el proceso técnico. Como ya se mencionó, Discovery es usado principalmente para el modelado de sistemas de negocios[16], en el cual - a diferencia de UML -, el modelo de tareas posee una representación semántica formal utilizando notaciones simples, donde el modelo visual plasmado en el diagrama de tareas genera un álgebra de tareas. Como utiliza algunas notaciones similares [34], no resulta difícil familiarizarse con él, en cambio se puede utilizar dicha álgebra para verificar formalmente los modelos desarrollados.

A pesar de tener más de una década de existencia su uso no ha sido masivo, una de las razones es que no posee herramientas de software para el modelado de tareas. El compilador existente trabaja mediante archivos de texto y carece de herramientas que faciliten su uso. En contra parte, UML tiene una gran cantidad de software (tanto libre como privado), el cual ayuda a generar los diferentes tipos de diagramas de los que se compone. Algunos ejemplos de diagramas son [19, 37]:

- Diagrama de paquetes.
- Diagrama de casos de uso.
- Diagrama de comunicación.
- Diagrama de actividad.
- Diagrama de secuencia.
- Diagrama de clases.
- Diagramas de estados.

Este trabajo de investigación se enfoca a la implementación de una herramienta para el diagrama flujo de tareas del Método Discovery, no se abarcará ninguna otra de las 4 fases.

1.4.1. Principios del Método Discovery

Existen 4 principios fundamentales sobre los cuales se basa Discovery:

1. Dirección: establece una secuencia para el diseño de actividades y productos.
2. Selectividad: establece la necesidad de seleccionar la notación apropiada según la técnica de diseño.
3. Transformación: la búsqueda de una mejor y más amplia comunicación entre el desarrollador y el cliente.
4. Compromiso: Tiene el fin de establecer una mejor notación y las actividades inherentes que sean adecuadas.

1.4.2. Herramienta existente.

En la sección 1.4 se mencionó que Discovery supera los diez años de existencia, a pesar de ello y de las ventajas expuestas anteriormente no se han desarrollado herramientas suficientes para la creación de las distintas actividades de las cuales se compone. A la fecha existe una herramienta llamada *Discovery Method CASE Tool* (DMCT), dicha herramienta fue implementada por Thom Parkes y dirigida por Simons[29].

Los artefactos de los que se compone el DMCT en su versión final, ofrece la posibilidad de uso y modificación del código. A pesar de no especificar bajo qué licencia es liberada y debido a la naturaleza académica del proyecto, se sobreentiende que se tienen los permisos para realizar actividades inherentes al código libre como lo es: modificación y redistribución.

1.4.3. Nuevas herramientas.

Las nuevas herramientas se deben construir basándose en modelos estructurados y lenguajes libres, de tal manera que se pueda obtener un conjunto de aplicaciones que apoyen las diferentes etapas de las que se compone el método Discovery, de esta manera se reducirán costos, específicamente se propone una herramienta con la cual se podrá realizar verificación formal visual a partir de diagramas de tareas.

A continuación se enumeran las necesidades no funcionales que debe contar el editor de diagramas de flujo de tareas propuesto: .

- Componentes visuales: Para facilitar la compresión, se requiere utilizar elementos gráficos similares a los que posee UML, esto con la finalidad de aprovechar la mayor difusión y uso que tiene en la actualidad.
- Lenguaje liberado bajo licencias libres: El lenguaje sobre el cual se construirá, debe estar amparado bajo alguna licencia libre.
- No debe ser dependiente del sistema operativo.

1.5. Descripción del problema.

Hoy en día todo desarrollo de software serio debe estar sustentado por una documentación adecuada, donde cada uno de los artefactos describan y mejoren la reutilización del código de calidad [40]. Es por ello que desde mediados de la década de los 90's, se buscó estandarizar los componentes del proceso de desarrollo.

Sin embargo, es un lenguaje de modelado y no un método, la mayor parte de los métodos consisten -al menos en principio-, en un lenguaje y un proceso para modelar[16], en cambio UML es una especie de meta-modelo[1] mediante el cual se modela al resto de UML [32]. Existen otras notaciones menos generales, orientadas a determinados tipos de sistemas y por consiguiente a tener menos problemas de ambigüedad, tal como *Fusion* y *Object-Z* [34].

Una alternativa a UML es la notación del Método Discovery, pero como se vio en la sección 1.4 sólo posee una herramienta de modelado. Por ello es imprescindible dotar de herramientas software a esta metodología. Se necesita mejorar el proceso de creación de diagramas de flujo de tareas con una o más herramientas cómodas e intuitivas, ya que actualmente los diagramas se traducen de forma manual, siendo esta una tarea tediosa y propensa a los errores.

Sin embargo, toda propuesta debe contemplar la facilidad de modificación, mantenimiento y compresión del código para lograr ser de utilidad a los desarrolladores.

Los desarrollos independientes suelen implementar soluciones de una manera más lenta, es decir, los productos software que mayor éxito poseen son los que se sustentan bajo una comunidad de desarrollo[38]; caso contrario a lo propuesto en la descripción de la herramienta DMCT de la sección 1.4.2.

1.6. Solución propuesta.

Para subsanar la debilidad dada por la falta de herramientas de software del Método *Discovery*, se debe implementar una herramienta que genere el álgebra de tareas a partir de un diagrama.

Para ello se propone el diseño e implementación de una herramienta que permita a los usuarios modelar diagramas de flujo de tareas, y a partir de estos diagramas obtener de forma automática la expresión del álgebra de tareas. Se pretende llevarla a cabo bajo el auspicio del Entorno de Desarrollo Integrado Eclipse, debido a que cumple con los criterios establecidos en la sección 1.4.3, así la nueva herramienta podrá ser liberada bajo Licencia Pública General (GPL), siguiendo como método de desarrollo OpenUP. Cabe mencionar que el producto final, se limita a implementar un plug-in el cual se integrará al IDE Eclipse, con todos los artefactos generados en el proceso de elaboración descritos por el método de desarrollo OpenUp; además de toda la información necesaria para su instalación, compresión, manipulación y modificación de la herramienta propuesta.

[1] Un meta-modelo es un modelo que define el lenguaje para expresar a otros modelos.

1.7. Objetivos.

A continuación se presentan los objetivos a desarrollar en este proyecto.

1.7.1. Objetivo general.

Desarrollar un plug-in visual de Eclipse para la generación del álgebra de flujo de tareas a partir de diagramas de tareas del método Discovery, el cual se construirá bajo modelos integrados y desarrollados en el *Graphical Modeling Framework* (GMF) y *el Eclipse Modeling Framework* (EMF) respectivamente.

1.7.2. Objetivos específicos.

- Diseño de una interfaz basada en un modelo estructurado.
- Diseñar y generar un modelo de EMF a partir de los componentes del álgebra de tareas para la estructuración de datos mediante documentos *Extensible Markup Languaje* (XML).
- Diseñar y generar un modelo de GMF a partir del modelo EMF para el desarrollo del área de trabajo (editor visual).
- Generar la documentación necesaria referente al modelo, siguiendo el proceso descrito por OpenUP.
- Generar el código del álgebra correspondiente, derivado de los diagramas de tareas.
- Grabar el código de los diagramas de tareas en formato *Xml Metadata Interchange* (XMI).

Capítulo 2

El Lenguaje Unificado de Modelado y el Método Discovery

2.1. UML

El éxito de los proyectos de desarrollo de aplicaciones o sistemas se debe a que sirven como enlace entre quien tiene la idea y el desarrollador. El lenguaje Unificado de Modelado (UML) es una herramienta que cumple con esta función, ya que sirve de ayuda para capturar la idea de un sistema y de esta manera poder comunicarla a quien esté involucrado en su proceso de desarrollo, todo ello se lleva a cabo mediante un conjunto de símbolos y diagramas. Cada diagrama tiene fines distintos dentro del proceso de desarrollo, donde todo forma parte de un modelo.

En las disciplinas de la ingeniería se hace evidente la importancia de los modelos ya que describen el aspecto y la conducta de uno o todos los componentes del mismo. Ese comportamiento o descripción pueden ya existir, estar en desarrollo o estar en estado de planeación.

Es en este momento cuando los diseñadores del modelo deben investigar los requerimientos del producto terminado y dichos requerimientos pueden incluir áreas tales como funcionalidad, rendimiento y confiabilidad. Además, a menudo el modelo es dividido en un número de vistas cada una de las cuales describe un aspecto específico del producto o sistema en construcción[31].

El modelado sirve no solamente para los grandes sistemas, aún en aplicaciones pequeñas se obtienen beneficios, sin embargo, es un hecho que entre más grande y complejo es el sistema, más importante es el papel que juega la descripción del modelado[8].

En la especificación de UML se puede comprobar que una de las partes que lo componen es un metamodelo formal[23]. Un metamodelo es un modelo que define el lenguaje para expresar otros modelos. Un modelo en el Diseño Orientado a Objetos es una abstracción cerrada semánticamente de un sistema y un sistema es una colección de unidades conectadas que son organizadas para realizar un propósito específico[19]. Un sistema puede ser descrito por uno o más modelos, posiblemente desde distintos puntos de vista.

Una parte del UML define entonces, una abstracción con significado de un lenguaje para expresar otros modelos (otras abstracciones de un sistema o conjunto de unidades conectadas que se organizan para conseguir un propósito). Lo que en principio puede parecer complicado no lo es tanto si pensamos que uno de los objetivos de UML es llegar a convertirse en una forma para

definir modelos, no sólo establecer una forma de modelo, de esta manera simplemente estamos diciendo que UML, define un lenguaje con el que podemos abstraer cualquier tipo de modelo [19, 33].

UML es una técnica de modelado de objetos y como tal supone una abstracción de un sistema para llegar a construirlo en términos concretos. El modelado no es más que la construcción de un modelo a partir de una especificación, aunque el modelo también resulta una especificación.

Un modelo es una abstracción de algún objeto, que se elabora para comprender ese objeto antes de construirlo. El modelo omite detalles que no resultan esenciales para la comprensión del original y por lo tanto facilita dicha comprensión.

Algo similar se puede realizar utilizando la *Object Modeling Technique* (OMT por sus siglas en inglés o Booch) un antecesor de UML. Por ejemplo, intenta abstraer la realidad utilizando tres clases de modelos OO: el modelo de objetos, que describe la estructura estática; el modelo dinámico, con el que describe las relaciones temporales entre objetos; y el modelo funcional que describe las relaciones funcionales entre valores. Mediante estas tres fases de construcción de modelos, se consigue una abstracción de la realidad que tiene en sí misma información sobre las principales características de ésta [6].

¿Por qué utilizar UML? hoy en día es necesario contar con un plan de desarrollo fundamentado en un buen análisis, sobre todo en las aplicaciones comerciales, el cliente debe comprender que trabajo se llevará a cabo por parte de los desarrolladores, de esta manera, cuando se presenten modificaciones en los requerimientos, estos podrán ser marcados de una manera clara y adecuada. A su vez el desarrollo es una tarea en equipo, por lo que cada uno de los miembros tiene que saber que lugar toma su trabajo en la solución final así como tambien conocer la solución global.

Conforme aumenta la complejidad del mundo, los sistemas de cómputo crecen de igual manera, ya que contienen cantidades enormes de información, por ello es imprescindible organizar el diseño de tal manera que los analistas, clientes, desarrolladores y otras personas involucradas en el desarrollo del sistema lo comprendan y sirva de modelo. UML proporciona ese tipo de organización.

Una de las principales ventajas que ofrece el basar un desarrollo en un modelo bien definido, es la reducción en el tiempo de desarrollo [12]. Cuando se calendariza un proyecto es escencial contar con un diseño sólido.

2.2. Antecedentes de UML.

El UML es la creación de Grady Booch, James Rumbaugh e Ivar Jacobson entre otros. Todos ellos laboraban en empresas distintas a mediados de los ochentas y principios de los noventas, cada uno diseñó su propia metodología para el análisis y diseño Orientado a Objetos. Sus metodologías predominaron sobre las de sus competidores. A mediados de los años noventas iniciaron un intercambio de ideas sobre sus métodologías, a partir de ese intercambio, es que decidieron desarrollar su trabajo en conjunto[7].

En 1994, Rumbaugh ingresó a Rational Software Corporation, donde se encontraba laborando Booch. Jacobson ingresó a Rational un año después. Es ahí donde comienza la gestación y posterior desarrollo de UML.

De esta manera la notación de UML deriva y unifica a las tres metodologías de análisis y diseños más extendidas:

- Metodología de Grady Booch para la descripción de conjuntos de objetos y sus relaciones.
- Técnica de modelado orientada a objetos de James Rumbaugh (OMT: Object - Modelling Technique).
- Aproximación de Ivar Jacobson (OOSE: Object- Oriented Software Engineering) mediante la metodología de casos de uso (use case) [19].

De las tres metodologías de partida, las de Booch. y Rumbaugh pueden ser descritas como centradas en objetos, ya que sus aproximaciones se enfocan hacia el modelado de los objetos que componen el sistema, su relación y colaboración. Por otro lado, la metodología de Jacobson es más centrada al usuario, ya que su método se deriva de los escenarios de uso.

Los anteproyectos de UML comenzaron a circular en la industria del software y las reacciones provenientes de los primeros usuarios trajeron consigo una gran cantidad de modificaciones. A medida que diversas corporaciones comenzaron a utilizar UML en sus proyectos de desarrollo, se creó un consorcio para el Lenguaje Unificado de Modelado. Entre los miembros de dicho consorcio se encuentran DEC, Hewlett-Packard, Intellicorp, Microsoft, Oracle, Texas Instruments y Rational. En 1997 el consorcio produjo la versión 1.0 de UML y fue puesto a disposición de la OMG como respuesta a su propuesta para un lenguaje de modelado estandar.

El consorcio aumentó y generó la versión 1.1, misma que se puso nuevamente en consideración del OMG. El grupo adoptó esta versión a finales de 1997. El OMG se encargó de la conservación de UML y produjo otras dos versiones en 1998. El UML ha llegado a ser el estándar en la industria del software[7].

La superestructura de la versión de la especificación de UML 2.0 (es decir, la versión estable y actualmente en uso) ha sido liberada, y está disponible para todos los usuarios en descarga gratuita. Esta superestructura la conforman tres partes separadas de UML 2.0 - la infraestructura (es decir, el núcleo para la creación de otros modelos), Object Constraint Language, y el soporte para intercambio de diagrama [23].

2.3. Conceptos básicos.

El UML debe entenderse como un estándar para modelado y no como un estándar de proceso software. Aunque UML debe ser aplicado en el contexto de un proceso, la experiencia ha demostrado que organizaciones y dominios del problema diferentes requieren diferentes procesos. Por ello se han centrado los esfuerzos en un meta-modelo y una notación común que proporcione una representación de esas semánticas. Los autores de UML fomentan un proceso guiado por casos de uso, centrado en la arquitectura, iterativo e incremental [7]. Bajo estas líneas genéricas proponen el proceso software definido en una de las extensiones del UML (*Objectory Extension for Software Enginnering*), pero en general es fuertemente dependiente de la organización y del dominio de aplicación.

2.3.1. Conceptos y modelos de UML

Los conceptos y modelos de UML pueden agruparse en las siguientes áreas conceptuales [10, 12]:

Estructura estática: Cualquier modelo preciso debe primero definir su universo, esto es, los conceptos clave de la aplicación, sus propiedades internas y las relaciones entre cada una de ellas. Este conjunto de construcciones es la estructura estática. Los conceptos de la aplicación son modelados como clases, cada una de las cuales describe un conjunto de objetos que almacenan información y se comunican para implementar un comportamiento. La información que almacena se modelada como atributo; la estructura estática se expresa con diagramas de clases y puede usarse para generar las declaraciones de estructuras de datos en un programa.

Comportamiento dinámico: Hay dos formas de modelar el comportamiento, una es la historia de la vida de un objeto y la forma como interactúa con el resto del mundo y la otra es por los patrones de comunicación de un conjunto de objetos conectados, es decir la forma en que interactúan entre sí. La visión de un objeto aislado es una máquina de estados, muestra la forma en que el objeto responde a los eventos en función de su estado actual. La visión de la interacción de los objetos se representa con los enlaces entre objetos junto con el flujo de mensajes y los enlaces entre ellos. Este punto de vista unifica la estructura de los datos, el control de flujo y el flujo de datos[19].

Construcciones de implementación: Los modelos UML tienen significado para el análisis lógico y para la implementación física. Un componente es una parte física reemplazable de un sistema y es capaz de responder a las peticiones descritas por un conjunto de interfaces. Un nodo es un recurso computacional que define una localización durante la ejecución de un sistema. Puede contener componentes y objetos.

Mecanismos de extensión: UML tiene una limitada capacidad de extensión pero que es suficiente para la mayoría de las extensiones que requiere el día a día sin la necesidad de un cambio en el lenguaje básico. Un estereotipo es una nueva clase de elemento de modelado con la misma estructura que un elemento existente pero con restricciones adicionales.

Organización del modelo: La información del modelo debe ser dividida en piezas coherentes, para que los equipos puedan trabajar en las diferentes partes de forma concurrente. El conocimiento humano requiere que se organice el contenido del modelo en paquetes de tamaño modesto. Los paquetes son unidades organizativas, jerárquicas y de propósito general de los modelos de UML. Pueden usarse para almacenamiento, control de acceso, gestión de la configuración y construcción de bibliotecas que contengan fragmentos de código reutilizable[7].

Elementos de anotación: Los elementos de anotación son las partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clasificar y hacer observaciones sobre cualquier elemento de un modelo. El tipo

principal de anotación es la nota que simplemente es un símbolo para mostrar restricciones y comentarios junto a un elemento o un conjunto de elementos.

2.3.2. Relaciones en UML

Existen cuatro tipos de relaciones entre los elementos de un modelo UML. Dependencia, asociación, generalización y realización. Estas se describen a continuación:

Dependencia: Es una relación semántica entre dos elementos en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (elemento dependiente). Se representa como una línea discontínua, posiblemente dirigida, que a veces incluye una etiqueta.

Asociación: Es una relación estructural que describe un conjunto de enlaces, los cuales son conexiones entre objetos. La agregación es un tipo especial de asociación y representa una relación estructural entre un todo y sus partes. La asociación se representa con una línea contínua, posiblemente dirigida, que a veces incluye una etiqueta. A menudo se incluyen otros adornos para indicar la multiplicidad y roles de los objetos involucrados.

Generalización: Es una relación de especialización / generalización en la cual los objetos del elemento especializado (el hijo) pueden sustituir a los objetos del elemento general (el padre). De esta forma, el hijo comparte la estructura y el comportamiento del padre. Gráficamente, la generalización se representa con una línea con punta de flecha vacía.

Realización: Es una relación semántica entre clasificadores, donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Se pueden encontrar relaciones de realización en dos sitios: entre interfaces y las clases y componentes que las realizan, y entre los casos de uso y las colaboraciones que los realizan. La realización se representa como una mezcla entre la generalización y la dependencia, esto es, una línea discontínua con una punta de flecha vacía.

2.4. UML 2.0

En junio del 2003, la Junta de Arquitectura de la OMG aprobó la propuesta de la primera revisión importante de UML, es decir actualizar a la versión 2.0 con ello se pretendió hacer que UML 2.0 se convirtirera en un hito importante en la evolución de las tecnologías de desarrollo de software.

Esto se debe a la principal fuerza impulsora detrás de UML 2.0 la cual es el desarrollo orientado a modelos, un acercamiento al desarrollo de software que cambia el enfoque del desarrollo desde el código a los modelos, y para mantener automáticamente la relación entre los dos. La esencia del software de modelado (como en todos los modelos) es la abstracción: la eliminación de los detalles evitando la distracción de las tecnologías de aplicación, así como el uso de conceptos que permiten la expresión más directa de los fenómenos en el dominio del problema

[12].

En las versiones previas de UML se hacía énfasis en que este no era un lenguaje de programación. Un modelo creado mediante UML no podía ejecutarse. En el UML 2.0, esta asunción cambió de manera drástica y se modificó el lenguaje, de manera tal que permitiera capturar un comportamiento más real del sistema modelado. De esta forma, se permitió la creación de herramientas que soporten la automatización y generación de código ejecutable, a partir de modelos UML[33].

Objetivos de UML 2.0: Al momento de desarrollar el nuevo estándar, la OMG se propuso, entre otros, dos objetivos principales debido a la influencia de éstos en la versión final del estándar. Estos objetivos son:

1. Hacer el lenguaje de modelado mucho más extensible de lo que era.
2. Permitir la validación y ejecución de modelos creados mediante UML.

UML 2.0 se desarrolla sobre la base de estos dos objetivos, causando un hito respecto a versiones anteriores. Para entender esta razón y el por qué de esta evolución, habría que profundizar en la historia y definición misma de UML [33].

Reestructuración del Lenguaje: Para lograr los objetivos enunciados, varios aspectos del lenguaje fueron reestructurados y/o modificados. La especificación se separó en cuatro especificaciones (paquetes) bien definidas, tal como se muestra en la Figura 2.1. Es interesante destacar que UML 2.0 puede definirse a sí mismo. Es decir, su estructura y organización es modelable utilizando el propio UML 2.0; de esta manera, se da un ejemplo de utilización en un dominio distinto al del desarrollo de software. En este caso, cada paquete del diagrama representa cada una de las cuatro especificaciones que componen al lenguaje[12].

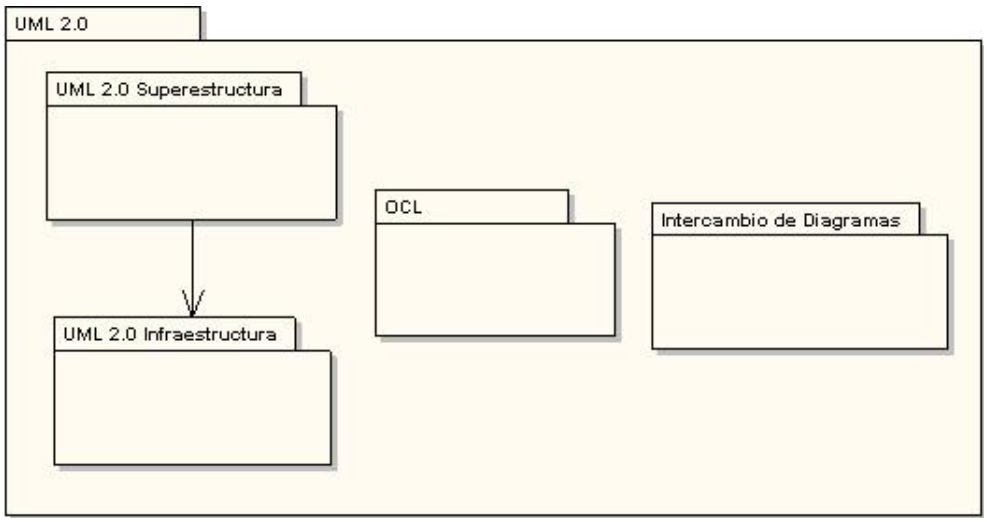


Figura 2.1: Especificaciones principales del UML 2.0 [23]

Estándares que conforman al UML:

- Superestructura: Es la especificación que usamos todos los días. Aquí se encuentran todos los diagramas que la mayoría de los desarrolladores conocen.
- Infraestructura: conceptos de bajo nivel. Meta-Modelo da soporte a la superestructura, entre otras.
- OCL (del inglés Object Constraint Language): De utilidad para especificar conceptos ambiguos sobre los distintos elementos del diagrama.
- XMI / Intercambio de diagramas: Permite compartir diagramas entre diferentes herramientas de modelado UML.

A continuación, se presenta de manera general cada una de las principales especificaciones que componen UML 2.0.

Superestructura: La superestructura de UML es la definición formal de los elementos de UML. Únicamente esta definición contiene más de 640 páginas. La superestructura es comúnmente utilizada por los desarrolladores de aplicaciónes. Es aquella que proviene de versiones anteriores de UML.

Es aquí dónde se definen los diagramas y los elementos que los componen. La Superestructura se encuentra dividida en niveles. Estos niveles se conocen como [23]:

- Básico (N1): Contiene los elementos básicos del UML 2.0 entre ellos: Diagramas de clases, Diagramas de actividades, Diagramas de Interacciones, y Diagramas de Casos de Uso.
- Intermedio (N2): Contiene los siguientes diagramas: Diagramas de estado, Perfiles, Diagramas de Componentes y Diagramas de despliegue.
- Completo (N3): Representa la especificación del UML 2.0 completa, como por ejemplo: las acciones, las características avanzadas entre otras.

Es importante destacar que basta con que una herramienta implemente el nivel de conformidad Básico (L1), para que se considere UML 2.0 compatible. Por eso, es común ver una disparidad de características bastante amplia entre dos herramientas distintas, aunque éstas sean UML 2.0 compatibles.

El bloque de construcción básico de UML es el diagrama. La estructura de los diagramas UML se muestra en el diagrama de la figura 2.2, de acuerdo con la especificación de UML 2.0 del Object Development Group. Los detalles sobre estos diagramas específicos se organizan de acuerdo a esta estructura taxonómica, que da la perspectiva a los diagramas y a sus interrelaciones. Los diagramas de interacción comparten propiedades y atributos similares, como lo hacen los diagramas estructurales y de comportamiento[12].

Infraestructura: En la infraestructura de UML se definen los conceptos centrales y de más bajo nivel. La Infraestructura es un meta-modelo (un modelo de modelos) mediante la cual se modela el resto de UML. Generalmente, la infraestructura no es utilizada por usuarios finales de UML; pero provee la base fundamental sobre la cual la Superestructura se define. Esta última sí es la utilizada por el común de los usuarios. La Infraestructura brinda también varios mecanismos de extensión, que hacen de UML un lenguaje configurable. Para los usuarios normales de UML basta con saber si la infraestructura existe y cuáles son sus objetivos[32].

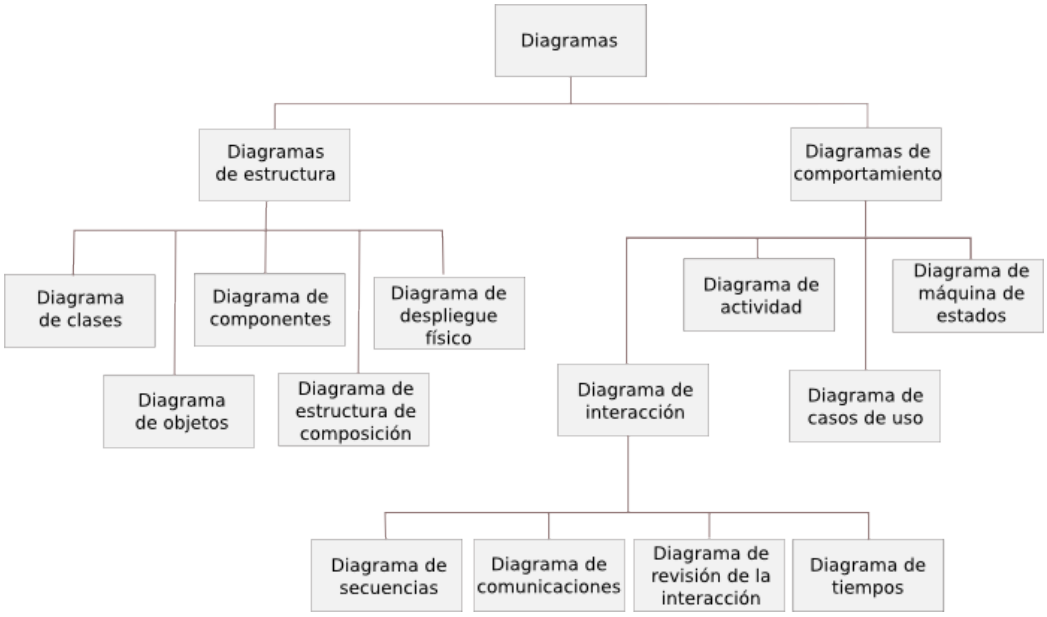


Figura 2.2: Bloque de construcción básico de UML 2.0[23]

Especificación para el Intercambio de Diagramas: La especificación para el intercambio de diagramas fue escrita para facilitar una manera de compartir modelos, realizados mediante UML, entre diferentes herramientas de modelado. En versiones anteriores de UML, se utilizaba un esquema XML para capturar los elementos utilizados en el diagrama; pero este esquema no decía nada acerca de la manera en que el modelo debía graficarse. Para solucionar este problema, se desarrolló una nueva especificación para el intercambio de diagramas utilizando un nuevo esquema XML, que permite construir una representación SVG (*Scalable Vector Graphics*). Esta especificación se denomina con las siglas XMI, que en inglés significa: XML *Metadata Interchange*; y en español se traduce como: XML de Intercambio de Metadatos (datos que representan datos). Típicamente esta especificación es solamente utilizada por quienes desarrollan herramientas de modelado UML[32].

OCL: OCL son siglas en inglés que significan: *Object Constraint Language* y que en español se traducen como: Lenguaje de Restricciones de Objetos. El OCL define un lenguaje simple, para escribir restricciones y expresiones sobre elementos de un modelo. El OCL suele ser útil cuando se está especificando un dominio particular mediante UML y es necesario restringir los valores permitidos para los objetos del dominio. OCL brinda la posibilidad de definir en los elementos de un diagrama, invariantes, precondiciones, poscondiciones y restricciones. OCL fue incorporado al UML en la versión 1.1.

2.5. Diversos diagramas existentes en UML 2.0

El Lenguaje Unificado de Modelado se compone de diversos elementos gráficos que se combinan para conformar diagramas. Debido a que UML es un lenguaje cuenta con reglas para

combinar dichos elementos.

La finalidad de los diagramas es presentar diversas perspectivas de un sistema, a las cuales se les conoce como modelo. El modelo UML de un sistema es similar a un modelo a escala de un edificio el cual debe quedar listo para su construcción. Es importante destacar que un modelo UML describe lo que hará el sistema, pero no dice como implementar dicho sistema [10, 23, 33].

A continuación se describen los diagramas más comunes utilizados en UML, así, como los conceptos que representan.

Diagrama de clases: Muestra una colección de elementos de modelado declarativo (estáticos), tales como clases, tipos, sus contenidos y relaciones.

Diagrama de componentes: Representa los componentes que integran una aplicación, sistema o empresa. Los componentes, sus relaciones, sus interacciones y sus interfaces públicas.

Diagrama de estructura de composición: Representa la estructura interna de un clasificador (tal como una clase, un componente o un caso de uso), incluyendo los puntos de interacción de clasificador con otras partes del sistema.

Diagrama de despliegue físico: Muestra cómo y dónde se desplegará el sistema. Las máquinas físicas y los procesadores se representan como nodos. La construcción interna puede ser representada por nodos o artefactos embebidos. Como los artefactos se ubican en los nodos para modelar el despliegue del sistema, la ubicación es guiada por el uso de las especificaciones de despliegue.

Diagrama de objetos: Presenta los objetos y sus relaciones en un punto del tiempo. Un diagrama de objetos se puede considerar como un caso especial de un diagrama de clases o un diagrama de comunicaciones.

Diagrama de paquetes: Presenta cómo se organizan los elementos de modelado en paquetes y las dependencias entre ellos, incluyendo importaciones y extensiones de paquetes.

Diagrama de actividades: Representa los procesos de negocios de alto nivel, incluidos el flujo de datos. También puede utilizarse para modelar lógica compleja y/o paralela dentro de un sistema.

Diagrama de comunicación: Interacción entre líneas de vida, donde es central la arquitectura de la estructura interna y cómo corresponde con el paso de mensajes. La secuencia de los mensajes se da a través de un esquema numerado de la secuencia.

Diagrama de revisión de la interacción: Se enfocan a la revisión del flujo de control, donde los nodos son interacciones u ocurrencias de interacciones. Las líneas de vida de los mensajes no aparecen en este nivel de revisión

Diagrama de secuencias: Representa una interacción, poniendo énfasis en la secuencia de los mensajes que se intercambian, junto con sus correspondientes ocurrencias de eventos en las Líneas de Vida.

Diagrama de máquinas de estado: Ilustra cómo un elemento (muchas veces una clase), se puede mover entre estados que clasifican su comportamiento de acuer-

do con disparadores de transiciones, guardias de restricciones y otros aspectos de los diagramas de máquinas de estados, que representan y explican el movimiento y el comportamiento.

Diagrama de tiempos: El propósito primario del diagrama de tiempos es mostrar los cambios en el estado o la condición de una línea de vida (representando una Instancia de un Clasificador o un Rol de un clasificador) a lo largo del tiempo lineal. El uso común es mostrar el cambio de estado de un objeto a lo largo del tiempo, en respuesta a los eventos o estímulos aceptados. Los eventos que se reciben se anotan, a medida que muestran cuándo se desea mostrar el evento que causa el cambio en la condición o en el estado.

Diagrama de casos de uso: Muestra las relaciones entre los actores y el sujeto (sistema), y los casos de uso.

2.6. Diagrama de actividades.

Como se mencionó en la sección 1.3 se pretende aprovechar el conocimiento previo de UML, así, el diagrama de actividades guarda cierta similitud con el diagrama de flujo de tareas – este último posee una semántica formal –; a continuación se detalla el comportamiento y manejo del diagrama de actividades [10, 12, 19].

2.6.1. Componentes

Todos los diagramas de actividades tienen una serie de elementos básicos, su comportamiento se define mediante la integración de los elementos descritos a continuación.

Acción: Es la unidad básica de este tipo de diagramas, pueden especificar pre y postcondiciones. Una acción puede ser cualquiera de lo siguiente:

- Obtener o establecer un valor de atributo
- La invocación de la operación de otra clase
- Llamar a una función
- La invocación de una actividad que contiene las acciones
- El envío de una señal o notificación de un evento a un grupo de objetos.

La notación visual que utiliza UML es un rectángulo redondeado. Se coloca el nombre del comportamiento simple como texto dentro del rectángulo.

Actividad: Contiene secuencias de acciones y/u otras actividades. A nivel de una clase orientada a objetos, puede utilizar una actividad para representar el método de una operación. También puede usarse para representar las tareas que componen un proceso de negocio. La notación visual para este componente UML es un rectángulo redondeado con el nombre de la actividad en el interior (como en una acción). También se puede mostrar el resultado de las actividades en un gran rectángulo redondeado que contiene secuencias complejas de acciones, actividades,

flujos de objetos y control de flujos. La forma compleja de una actividad también le permite mostrar los parámetros, las condiciones previas, las postcondiciones y las propiedades de la misma.

Nodos de control: Son utilizados para controlar el flujo de los objetos a través de un conjunto de actividades y acciones. Los nodos de control son los siguientes:

- Inicio: Es el inicio de una secuencia de actividades o acciones. se representa como un punto grande, figura 2.3.

Figura 2.3: Nodo Inicio.

- Final de actividad: Pone fin a todos los flujos de control y los flujos de objetos en una actividad. Se representa visualmente con un símbolo de diamante cerrado, figura 2.4.

Figura 2.4: Nodo final de actividad.

- Final de flujo: Finaliza algunos flujos dentro de una actividad, se representa visualmente al colocar al final de un flujo un pequeño círculo con una X dentro, figura 2.5.

Figura 2.5: Nodo final de flujo.

- Decisión: Un nodo de decisión utiliza una prueba para asegurarse de que un flujo de objeto o de control tome sólo un camino. Similar a una selección *si-entonces-sino* de una ruta de ejecución. Un nodo de decisión se representa visualmente con forma de diamante de gran tamaño, en la cual los criterios de decisión se colocan entre corchetes para cada ruta de acceso en la línea de control de flujo, figura 2.6.
- Mezcla: Es utilizado para fusionar flujos hacia un nivel específico, dicha unión no responde a una jerarquía definida. Se trata de la misma forma visual que un nodo de decisión.
- Bifurcación: Componente mediante el cual se pueden realizar tareas en paralelo para dividir el comportamiento en las operaciones simultáneas. Vi-

Figura 2.6: Nodo decisión

sualmente se representa con una línea de la cual parten las diferentes actividades, figura 2.7.

Figura 2.7: Nodo Inicio

- Fusión: Utilizada al momento de unir los flujos paralelos de las operaciones separadas mediante una bifurcación, se utiliza un símbolo similar a la bifurcación con la excepción de que los flujos entran hacia el componente.
- Conector: Utilizado para trasladar el flujo a otro espacio físico del diagrama. Visualmente se utiliza un conector a un círculo pequeño con una etiqueta en el interior. El conector indica que el flujo direcciona a otro lugar en el diagrama o en otra página donde se encuentra un conector con la misma etiqueta.

Objeto nodo: Las operaciones de clase pueden tomar parámetros y generar un retorno de resultados, así como transformar objetos en otros objetos. Visualmente se utiliza un cuadro de clase con el nombre de la clase del objeto para mostrar el resultado de un nodo. También puede describir el estado del objeto, incluyendo el nombre del estado entre corchetes debajo del nombre de la clase.

Flujo de objetos: Anteriormente era conocido como "flujo de datos". El flujo de objetos se utiliza en los diagramas de actividad para mostrar el resultado del flujo de objetos de una actividad o acción a otra. Visualmente representado como una línea con punta.

Control de flujo: Conecta las acciones y actividades en conjunto, muestra la secuencia de ejecución. Conecta actividades y acciones con una línea que tiene una punta de flecha para indicar la dirección en la que el control fluye.

2.6.2. Usos de los diagramas de actividad.

A continuación se presentan los usos de los diagramas de actividad así como una breve descripción de los mismos.

- Operaciones de alto nivel: Descripción de una clase con una operación compleja que involucra muchos pasos. Se utiliza un diagrama de actividades para mostrar los pasos como una secuencia de actividades.

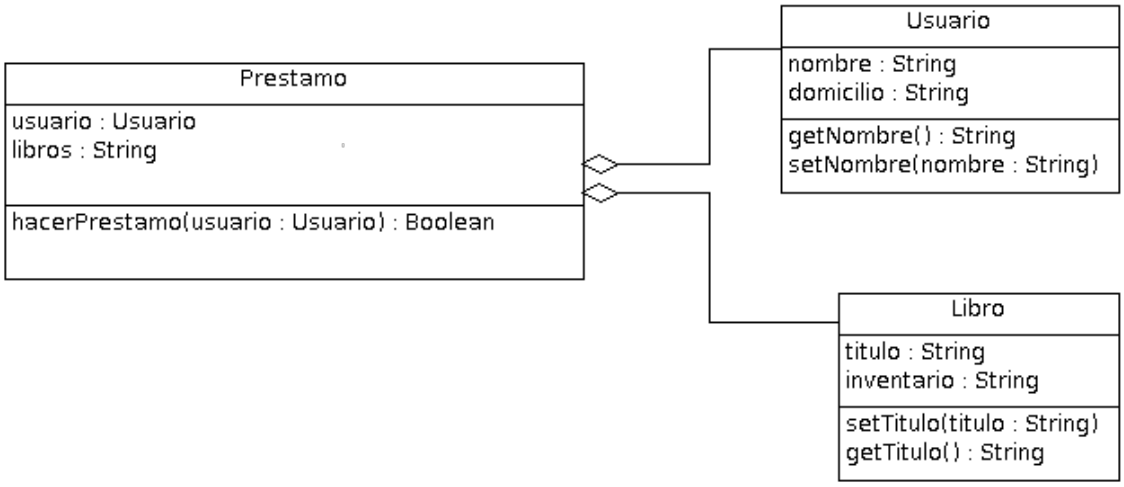


Figura 2.8: Diagrama de clases ejemplo para la construcción de un diagrama de actividad.

- Casos de uso a detalle: Cuando un caso de uso es un grupo de pasos realizados al mismo tiempo se utiliza una forma de diagrama de actividad que muestra el flujo de la interacción entre el escenario principal y el éxito de cualquier escenario alternativo.
- Flujo de trabajo o Proceso de negocio: Ideales para modelar procesos de negocio, no sólo las operaciones de software. Muestra que existen actividades que deben tomar decisiones y que documento del proceso de negocio genera.
- Modelado de procesos: Utilizados para modelar cualquier proceso. Muestra el modelo en forma de pasos de un proceso como actividades y muestra el resultado de la secuenciación de los flujos de control y los nodos de control.
- Método resumido para los diagramas de secuencia: Si en el proceso de desarrollo se genera una gran cantidad de diagramas de secuencia para cada caso de uso, estos pueden ser tratados como un diagrama de actividades. Ya que el comportamiento complejo de un caso de uso puede ser más fácil de entender visto como un diagrama de actividad.

La figura 2.8 es un ejemplo básico de un diagrama de clases para documentar una operación de alto nivel. El ejemplo se centra en la operación hacerPrestamo de la clase Prestamo. La operación toma un parámetro (usuario:Usuario), cuando se invoca la operación hacerPrestamo se pasa un instancia del tipo Usuario, cuando hacerPrestamo se ha completado este retorna un valor booleano.

Para este ejemplo, partimos del supuesto de que una persona realizó una reservación de un libro en una biblioteca, consultando en un sistema en línea, para ello debe seleccionar un libro y contar con su credencial de la biblioteca, finalmente, será notificada de la fecha de entrega del o los libros, cabe resaltar que si dicha persona tiene una sanción, no podrá llevarse ningún libro. La figura 2.9 captura el comportamiento para el préstamo de un libro en un diagrama de actividad.

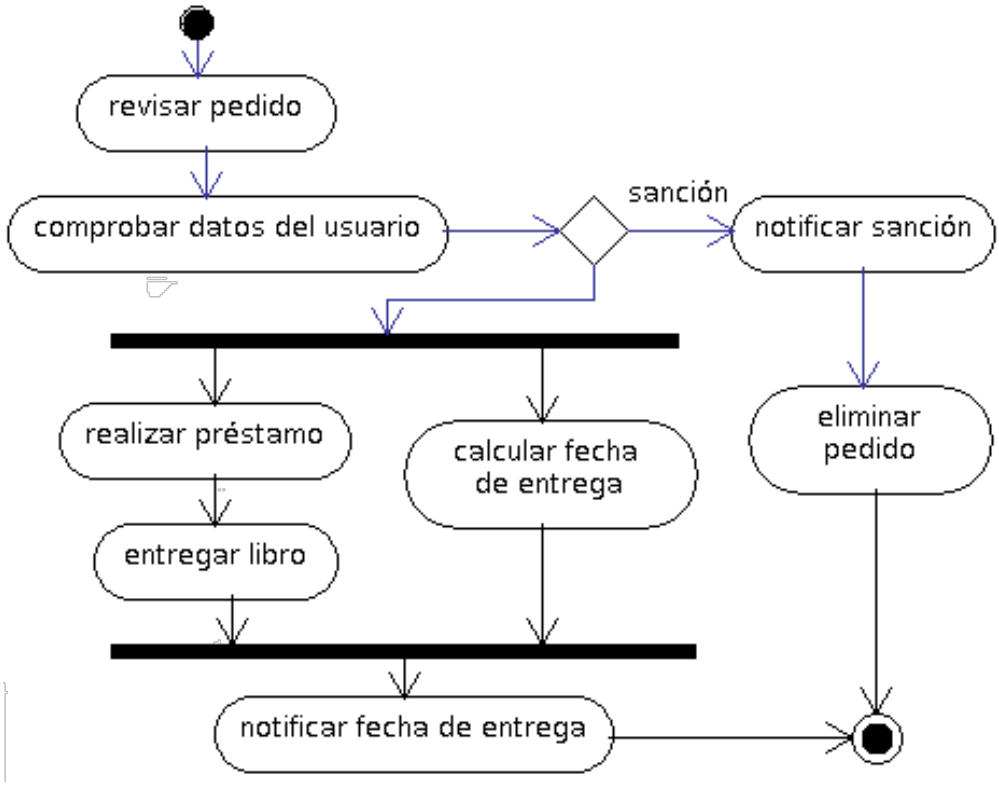


Figura 2.9: Diagrama de actividad para obtener prestado un libro.

2.6.3. Otros usos de los diagramas de UML

Debido a su simplicidad y amplia difusión, algunos diagramas de UML han sido utilizados para mostrar gráficamente otros aspectos en el desarrollo de software, como es el diseño de bases de datos. Así mismo, su uso se ha extendido fuera de este medio siendo acogido en los ambientes empresariales para modelar procesos de negocios.

Al momento de iniciar un proyecto de desarrollo de software, los analistas realizan el modelado del proceso de negocios de la empresa, en ocasiones ese modelo suele mostrar información útil a la empresa ya que se registran las operaciones fundamentales para el correcto funcionamiento de la organización.

En ocasiones este tipo de análisis, no contribuye de una manera fundamental al buen desarrollo del software, sin embargo sí genera un beneficio para la empresa, ya que esta puede utilizar el modelo para establecer mejoras en los procesos analizados [24].

2.6.4. Inconsistencias de UML

El Lenguaje de Modelado Unificado actualmente es la notación estandar del modelado visual, posee 13 diferentes diagramas que se pueden utilizar para representar un sistema de software detallando diferentes aspectos así como diversas perspectivas.

Desde el año de 1997 y bajo la supervisión del OMG ha evolucionado. Sin embargo, su

notación se ha vuelto más compleja con la finalidad de abarcar todo el proceso completo para el modelado de sistemas así como de procesos [20].

Esto ha originado algunas críticas respecto a su composición semántica y las diferentes formas de interpretar los modelos.

El problema principal, es la ambigüedad con la que se pueden crear los diversos diagramas existentes en el lenguaje, esto provoca que se carezca de un sentido específico así como que la validez del mismo no pueda establecerse con la claridad necesaria [20, 33].

Por esta razón, se han llevado a cabo diversos intentos por formalizar algunas partes de UML, incluyendo los trabajos realizados por el pUML (del inglés Precise UML group) el cual intentó clarificar mediante la creación de una notación semántica para UML y sus respectivas herramientas para los rigurosos análisis de los modelos de UML [21]. De manera conjunta con IBM y pUML, propusieron un Framework de Meta-Modelado (MMF) como una alternativa al metamodelo original del OMG.

Algunos trabajos proponían utilizar el lenguaje Z, usado por Bruel y France al presentar una transformación de diagramas de clase UML a especificación Z Kim y Carrington presentaron un mapeo formal para la transformación de diagramas de clases de UML a especificación object-Z .

Después se tomó otro camino para formalizar a UML, en esta variante se utilizó el lenguaje formal Alloy. Bordbar y Anastasakis propusieron una herramienta llamada UML2Alloy, en la cual un modelo es transformado de un Metamodelo UML a un metamodelo Alloy [36].

Otros trabajos se enfocaron al desarrollo de revisores de modelos y sus respectivas herramientas para UML, sin embargo esto aún es inmaduro y la integración con UML se ha mostrado lenta.

En octubre de 2003, OCL fue adoptado por el OMG como parte de UML 2.0 con ello se pretenden construir expresiones en OCL para UML donde se puedan representar invariantes, precondiciones, postcondiciones, inicializaciones, guardias, reglas de derivación, así como consultas a objetos para determinar sus condiciones de estado. Su papel principal es el de completar los diferentes artefactos de la notación UML con requerimientos formalmente expresados [22].

A pesar de todos los intentos enfocados a formalizar UML, no se han logrado avances substanciales en cuanto a la mejora de este tipo de problemas, principalmente porque las características propias del lenguaje son muy amplias, ya que abarcan diversas vistas y perspectivas.

Una posible mejora radicaría en una notación orientada a objetos más sencilla y pequeña, fácil de aprender, que sea más exacta y replicable. Dicha notación debe estar soportada por un lenguaje formal para representar la semántica de una manera precisa de de tal forma que los modelos se puedan verificar unos con otros.

Una posible solución para tal notación es restringir un perfil UML, denotado por el Método Discovery, el cual pone especial atención al minimalismo y la coherencia [16].

2.7. Método Discovery

2.7.1. ¿Qué es el Método Discovery?

El Método Discovery es una metodología orientada a objetos propuesta formalmente en 1998 por Anthony Simons, considerado como un método centrado en el proceso técnico [34]. Desde su

primera versión este ha utilizado una notación simple y semánticamente clara basada en UML, pero cambiando algunos modelos, para mejorar el proceso de formalización. Fue probado en diversos proyectos por estudiantes de posgrado de la Universidad de Sheffield [35]. La notación simple y sin ambigüedades lo hace una opción viable y consistente en comparación con UML.

Existen 4 principios básicos en el Método Discovery .

- Dirección: Especifíca la lógica para el diseño de actividades y productos.
- Selección: Escoge sólo aquellas notaciones dadas por la técnica de diseño.
- Transformación: Abarca las transformaciones del modelo.
- Compromiso: Incrementa la comunicación entre el cliente y el desarrollador mediante anotaciones claras y actividades adecuadas.

Para su organización, el Método Discovery se estructura en 4 fases las cuales son:

- Modelado de Negocios: en esta fase se trata de obtener conocimiento sobre el contexto del sistema, así como capturar y analizar los requisitos para llegar a una decisión sobre el alcance del contrato y su posible costo.
- Modelado de Objetos: Se tiene como finalidad identificar los objetos y las unidades modulares del diseño.
- Modelado del Sistema: En ella se analiza la relación entre límites y la dependencia interna (acoplamiento y cohesión), además de identificar los sub sistemas naturales.
- Modelado de Software: En esta fase se realiza la traducción de los modelos en algún lenguaje de programación específico.

De forma contraria a lo recomendado por UML, el Método Discovery retarda (en la medida de lo posible) la creación de objetos en las primeras etapas del desarrollo, esto porque los objetos iniciales subsisten a lo largo de las demás etapas de desarrollo, trayendo consigo una perspectiva no muy amplia de las necesidades totales y reales del proyecto.

El método Discovery se fundamenta en técnicas y notaciones tomadas o adaptadas de otros métodos ya existentes. Selecciona y evalúa cada técnica elegida y su notación correspondiente, de esta manera se crea un meta-modelo gradual con las mejoras de cada evaluación.

Uno de los puntos más importantes que presenta el Método Discovery es que además de utilizar una notación simplificada ésta resulta coherente en todo el método, es decir, si un elemento tiene un significado específico, dicho elemento tendrá el mismo significado en todos los demás diagramas y en cualquier otro lugar donde el método lo use.

Un ejemplo claro pueden ser los símbolos de agregación y generalización, estos dos componentes tienen el mismo significado semántico en el modelo de datos como en el modelo de estructuras de tareas, la figura 2.10 muestra gráficamente el ejemplo. De esta manera es como se puede eliminar la confusión proveniente de UML creada por <<include>> y <<extend>> ello en el diagrama de casos de uso, esta relación muestra un comportamiento consistente. La ventaja que conlleva esta característica, es que los diagramas resultan más comprensibles ya que se elimina la posible confusión derivada de la notación [16].

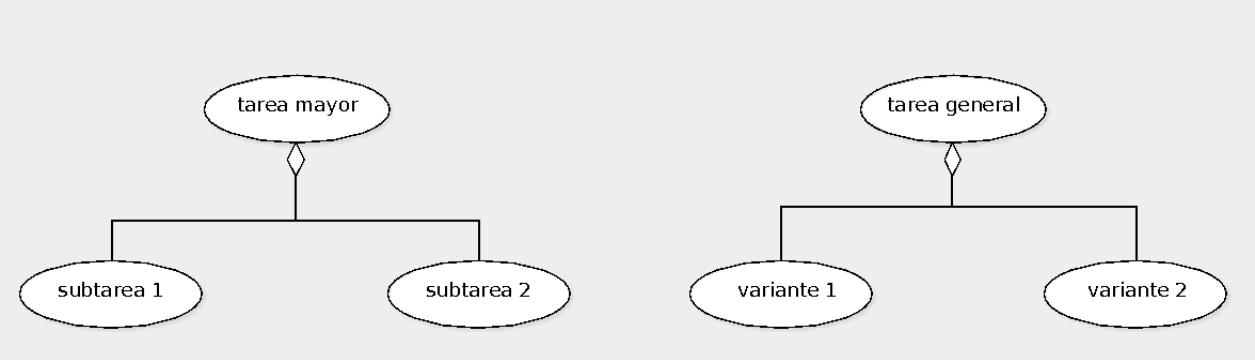


Figura 2.10: Generalización para diagramas de datos y de estructuras de tareas.

Sin embargo, debido al alcance del proyecto sólo se detallará la primera fase del método: la fase de modelado de negocios.

2.7.2. Modelado de negocios

El modelado de negocios es la primera fase del Método Discovery. Esta fase consiste primordialmente en explorar y representar los requisitos del cliente en un modelo estructurado dentro del contexto de la empresa y donde se establece el funcionamiento del sistema.

Consiste en identificar las tareas de negocios y el apoyo a los objetivos del cliente, paralelamente es importante la elaboración de un contrato donde se describa la entrega gradual del sistema. Para esta fase se requiere llevar a cabo las siguientes actividades:

- Entrevistas.
- Análisis del dominio.
- Análisis de tareas.
- Planificación del contrato.

Las entrevistas son realizadas por el *desarrollador,* este término se refiere a uno o varias personas designadas por la empresa desarrolladora, se aplican a el *cliente* en el entendido de que puede ser una o más personas desde directores hasta usuarios potenciales del sistema.

2.7.3. Análisis de tareas.

El análisis de tareas es parte integral del Modelado de Negocios que consiste en tres técnicas:

Task Sketching:

Se identifican las tareas de negocio a gran escala. La idea es capturar tareas de negocios más allá de los detalles del procesos del sistema. También marca el orden de ejecución de las tareas mediante una flecha, llamada flujo de control. Por último, en esta parte se identifican los actores y su relación con las tareas.

Narrative Modelling:

Es utilizada para describir las tareas en el modelo de negocios. Cabe resaltar que es el desarrollador quien las captura a partir de lo que mencionan los clientes, así se identifican el propósito

de la tarea pudiendo de esta manera establecer los actores y sus objetos correspondientes. Dicha descripción puede incluir alternativas y excepciones, además de condiciones previas y postcondiciones las cuales afectan a la tarea que las incluye.

Task Modelling.

Es esta parte del modelado de tareas la que se realiza con mayor detalle, es decir, es en este proceso donde se lleva a cabo un encadenamiento temático, una reestructuración lógica de tareas, y la creación de flujos alternativos de tareas.

Podemos entender el encadenamiento temático como el proceso mediante el cual se agrupan diversas tareas en tareas con mayor abstracción (creación de supertareas).

Esta agrupación está dada por dos criterios básicos:

- Mediante la identificación de tareas con un objetivo común.
- Mediante la identificación de tareas con actores y objetos comunes así como su respectiva participación en dichas tareas.

La finalidad de reestructurar las tareas consiste en mejorar el diseño estructural del modelado de tareas, sobre todo se busca reducir la dependencia de tareas y aislar a los actores que participan en varias de ellas.

Finalmente, los flujos alternativos de las tareas es un procedimiento mediante el cual el desarrollador describe los flujos desde la perspectiva de un objeto particular o actor [16].

2.7.4. Diagrama de flujo de tareas.

El Método Discovery define una tarea como "algo que tiene el sentido específico de una actividad con propósitos de negocios llevada a cabo por los stakeholders (involucrados en el desarrollo del sistema)". A partir de esta afirmación, el Método Discovery propone dos tipos de diagramas: el diagrama de estructuras de tareas y el diagrama de flujos de tareas. Sin embargo el proyecto se limita a trabajar con el diagrama de flujo de tareas.

En algún momento - durante el proceso de identificación de tareas y la estructuración de relaciones para las diferentes perspectivas de los actores en el modelo-, será necesario representar el flujo de las relaciones de trabajo. El Método Discovery representa este flujo de trabajo por medio del Diagrama de flujo de tareas.

Básicamente, un diagrama de flujo de tareas representa el orden en que las tareas se llevan a cabo en la organización, también denota la relación lógica entre todas las tareas que componen el diagrama.

Mientras que la notación utilizada en el Método Discovery se basa en el diagrama de actividades de UML, la figura de una elipse para la representación de una tarea se mantiene constante [16]. A continuación se describen los diferentes elementos a partir de los cuales se construyen los diagramas de flujo de tareas, algunos ejemplos de estos diagramas se pueden observar en la figura 2.11.

Task: Es el elemento básico el cual contiene información respecto a una tarea específica .

Choice: Se representa con un diamante, es el componente encargado de llevar a cabo un operación de selección dadas dos condiciones.

Exception: Se representa por la mitad de un diamante, puede definirse como un caso especial

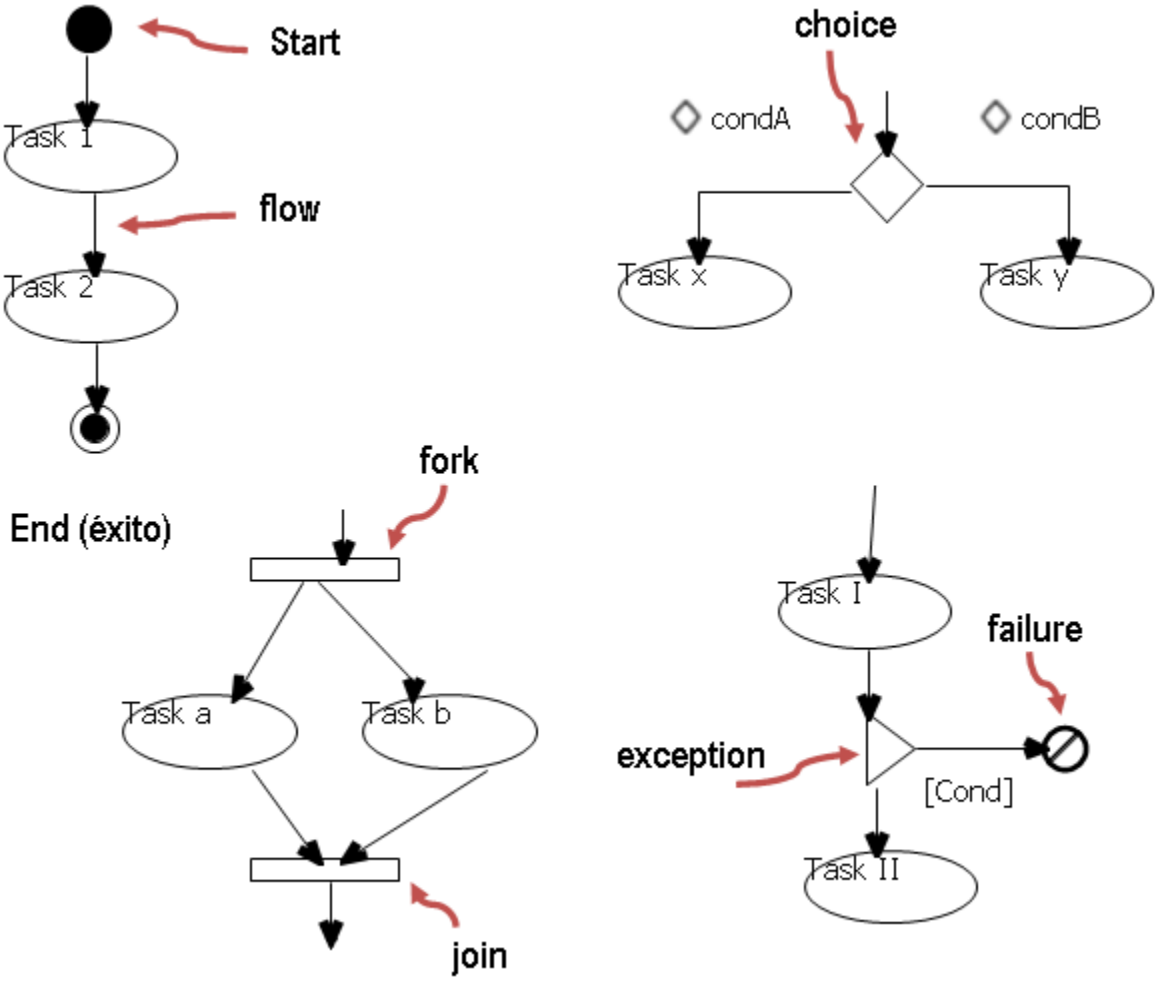


Figura 2.11: Elementos del diagrama de flujo de tareas [16].

de un Choice, únicamente posee una condición con dos flujos de salida, tomando como hecho que la segunda condición implícita es la negación de la primer condición.

Failured: Se representa por un círculo tachado con una sola línea diagonal, además indica que el proceso terminó en un estado incorrecto.

Start: Corresponde al ícono inicio ya conocido en los diagramas de estado de UML.

End: Corresponde al ícono fin ya conocido en los diagramas de estado de UML. A diferencia de Failured, End indica que el proceso descrito por el diagrama terminó con éxito.

Fork: Permite el manejo de tareas concurrentes tal como lo hace el diagrama de actividades de UML, esto debido a que en diversas ocasiones (en los procesos de negocios) las acciones no son todas secuenciales ni dependientes, por ello pueden llevarse a cabo en forma paralela con cierto grado de independencia. Es decir, pueden iniciar al mismo tiempo en un Fork y terminar con un grado de diferencia para posteriormente reunir los flujos en un Join, en el entendido de que los flujos paralelos deben provenir de un Fork y terminar en un Join todo ello de manera balanceada, debe existir un Join por cada Fork. Gráficamente los componentes Fork y Join son representados por una barra horizontal.

Ahora se detallará la semántica del diagrama de flujo de tareas fundamentada por el álgebra de tareas. Este punto es importante ya que dicha álgebra otorga la representación semántica de la que carece UML.

2.7.5. Del diagrama de flujo de tareas al álgebra de tareas.

La representación formal del diagrama de flujo de tareas está dada por el álgebra de tareas. Esta álgebra es una traducción o representación abstracta de dicho diagrama. Las estructuras básicas de las que se compone el álgebra son las siguientes:

- **Secuencia:** Una secuencia de tareas se representa en la figura 2.12, su traducción al álgebra de tareas corresponde a {a;b;c}.

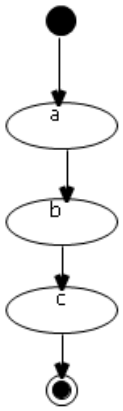


Figura 2.12: Secuencia de tareas en el diagrama de flujo de tareas.

En este punto se debe resaltar que los componentes Start y End no tienen una traducción en el álgebra, implícitamente las llaves cumplen ese cometido. Además, el orden de ejecución está

dado por los flujos, la posición en la expresión depende del nombre de la tarea, ejecutándose de izquierda a derecha y separada por puntos y comas.

- **Selección:** Esta estructura presenta guardias que son mútuamente excluyentes y exahustivas. La figura2.13 muestra su uso con dos opciones. En el proyecto se contempla el uso de Choice de forma binaria. La traducción de la figura 2.13 correspone a {a+b}.

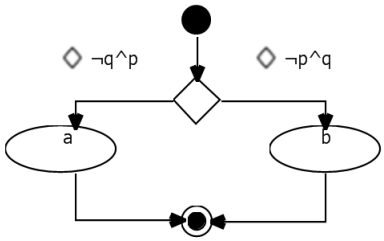


Figura 2.13: Selección binaria del Método Discovery implementada en el proyecto.

Cabe mencionar que las condiciones del componente *Choice* no tienen una representación directa en la sintaxis abstracta, sin embargo en la versión actual existe una propuesta para su representación [15].

- **Paralelismo:** La imagen de la figura 2.14 muestra la estructura del paralelismo mediante las tareas secuenciales a;b y c;d. La representación paralela se compone de manera similar a una selección, pero usando el símbolo || así la traducción a la sintaxis abstracta corresponde a { (a;b) || (c;d) }.

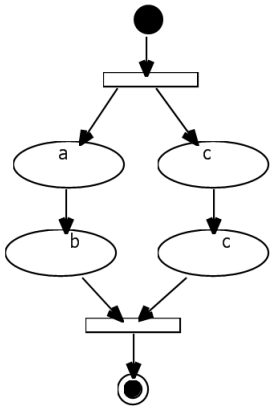


Figura 2.14: Estructura del paralelismo en el Método Discovery.

- **Repetición:** El Método Discovery contempla dos tipos de ciclos: *while* y *until*. La figura 2.15 muestra la representación gŕafica del ciclo *while*. La figura 2.16 muestra la representación gráfica del ciclo *until*. Su traducción al álgebra de tareas queda de la siguiente forma:

  μx.(a; ε + x) y μx.(ε + a; x) respectivamente.

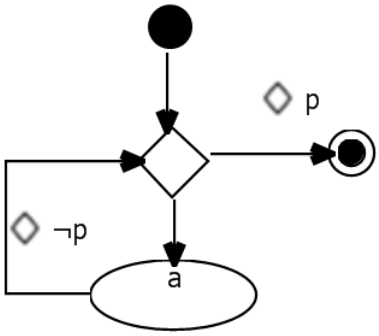


Figura 2.15: Ciclo while

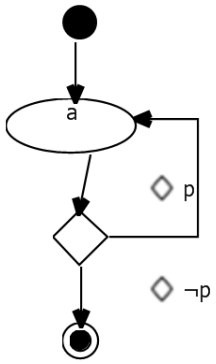


Figura 2.16: ciclo until

Las guardias de ambos ciclos no poseen una representación directa en la sintaxis abstracta.

El Método Discovery considera la encapsulación como medio para delimitar el ámbito de los diagramas. Otros componentes utilizados por el Método Discovery y que poseen una representación en la sintaxis abstracta son:

- ε: Utilizado por las estructuras repetitivas y la tarea vacía (Epsilon).
- σ: Terminación exitosa de un proceso (Sigma).
- Φ: Terminación abrupta de un proceso, (Phi).

Finalmente la imagen de la figura 2.17 muestra la sintaxis abstracta definida en la forma Backus Naur. Esta es una estructura básica usada para los diagramas de flujo de tareas. Como se mencionó en la sección 1.4.3, se requiere una herramienta para la automatización de la traducción.

La herramienta desarrollada se construyó sobre diferentes frameworks de Eclipse, de esta manera se pretende disminuir el tiempo de implementación así como el esfuerzo necesario para el respectivo mantenimiento [13].

```
Activity ::= ε                  --  Actividad vacía
        |σ                      -- Exito
        |ϕ                      -- Fallo
        | Task                  -- Tarea
        | Activity ; Activity   -- Secuencia de actividades
        | Activity + Activity   -- Selección de actividad
        | Activity || Activity  -- Actividades paralelas
        | μx.(Activity ;  ε + x)  -- Ciclo hasta
        | μx.( ε + Activity ; x)  -- Ciclo mientras

Task ::= Simple                 -- Tarea simple
        | { Activity }          -- Actividad encapsulada
```

Figura 2.17: Sintaxis abstracta del álgebra de tareas del Método Discovery [16].

2.7.6. IDE Eclipse.

Eclipse es un Entorno de Desarrollo Integrado (IDE por sus siglas en inglés) de código abierto multiplataforma construido para desarrollar lo que el propio proyecto llama "Aplicaciones de Cliente Enriquecido". Esta plataforma, se ha usado para desarrollar entornos de desarrollo integrados como el IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse). Sin embargo, también se puede usar para otros tipos de aplicaciones cliente como BitTorrent o Azureus. Eclipse es también una comunidad de usuarios, extendiendo constantemente las áreas de aplicación cubiertas. Un ejemplo es el recientemente creado *Eclipse Modeling Project*, cubriendo casi todas las áreas de *Model Driven Engineering* (proyecto sobre el cual se fundamenta este trabajo) [28].

La arquitectura base para Eclipse es la Plataforma de cliente enriquecido (del Inglés Rich Client Platform RCP). Cuyos componentes son:

- Plataforma principal - inicio de Eclipse, ejecución de plugins.
- OSGi - una plataforma comercial estándar.
- El Standard Widget Toolkit (SWT) - Un widget toolkit portable.
- JFace - manejo de archivos, manejo de texto, editores de texto .
- El Workbench de Eclipse - vistas, editores, perspectivas, asistentes.

Los widgets de Eclipse están implementados por una herramienta para widget de Java llamada SWT, a diferencia de la mayoría de las aplicaciones Java, que usan las opciones estándar Abstract Window Toolkit (AWT) o Swing. La interfaz de usuario de Eclipse también tiene una capa GUI intermedia llamada JFace, la cual simplifica la construcción de aplicaciones basadas en SWT.

El IDE Eclipse se construye a base de plug-ins (módulos) para proporcionar toda su funciona-

lidad al frente de la plataforma de cliente enriquecido, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no.

Este mecanismo de módulos genera una plataforma ligera para componentes de software. Podemos definir a un plug-in como la unidad mínima de funcionalidad de Eclipse que se puede distribuir de manera separada.

Las herramientas pequeñas se escriben como un único plug-in, mientras que en las complejas la funcionalidad radica en varios plug-ins. Existe un pequeño núcleo de la plataforma Eclipse que se encarga de coordinar al resto de la plataforma la cual está implementada como plug-ins.

El proyecto construido como un plug-in se integra a Eclipse, de esta manera se aprovechan las funcionalidades otorgadas por el IDE, particularmente la construcción mediante el patrón de desarrollo Modelo-Vista-Controlador. Además, se reducen tiempos de mantenimiento, se tiene un cierto grado de independencia del sistema operativo y se automatiza el proceso de distribución [11].

Otra de las características fundamentales sobre las cuales se construyó el proyecto, es que Eclipse provee *frameworks* robustos para el desarrollo de aplicaciones gráficas, así como definición y manipulación de modelos de software, aplicaciones web, etc. Por ejemplo, GEF (*Graphic Editing Framework - Framework* para la edición gráfica) es un plug-in de Eclipse para el desarrollo de editores visuales.

Este tipo de construcción responde al patrón de diseño Modelo-Vista-Controlador, de esta manera se divide la aplicación en partes separadas que se comunican entre sí de forma específica, es decir, el modelo de datos por separado (modelo), la interfaz gráfica de usuario (la vista) y la lógica de negocio (controlador) [11, 28].

El proyecto se compone de dos *frameworks* fundamentales, uno es GEF para la visualización gráfica de los componentes, el otro es EMF (*Eclipse Modeling Framework, Framework* para el desarrollo de herramientas basadas en un modelo estructurado):

GEF Framework utilizado para la creación de editores gráficos que pueden ir desde procesadores de texto wysiwyg (What You See Is What You Get en inglés, "lo que ves es lo que obtienes") hasta editores de diagramas UML, interfaces gráficas para el usuario (GUI), etc. Dado que los editores realizados con GEF se integran a Eclipse de forma transparente al usuario, estos se pueden utilizar conjuntamente con otros plug-ins. Hacen uso de su interfaz gráfica personalizable y profesional. Por lo general, GEF se utiliza en combinación con GMF (del inglés *Graphical Modeling Framework*), que une a GEF y a EMF para crear el código necesario para implementar el modelo de datos que origina un editor de diagramas. Los editores generados con el GEF constan de los siguientes componentes [27]:

- Barras de herramientas.
- Figuras con representación gráfica derivada de un modelo de datos.
- Mapeo de figuras y su significado en el modelo de datos.
- Los objetos capturan ciertos eventos en respuesta a la interacción con los usuarios.
- Comandos con las sentencias hacer y deshacer.

EMF es un framework de modelado diseñado específicamente para desarrollar aplicaciones basadas a partir de un modelo estructurado descrito bajo la especificación XMI. La aplicación resultante brinda soporte mediante la implementación de clases adaptadoras en lenguaje Ja-

va, las cuales permiten la visualización y edición de los comandos descritos en el modelo. Sin embargo, su mayor aporte y trascendencia, es que establece una base para la inter operabilidad con otros plug-ins, ya que entre sus clases adaptadoras se crea la estructura que regirá la lógica de negocios, esto es, el comportamiento que implementará el desarrollador [8, 28].

Mediante estos dos frameworks se identifica el papel que desempeñará el componente controlador en el proyecto. Para abstraer del modelo visual al álgebra de tareas se construyó un sistema similar a un compilador.

2.8. Compiladores.

Como se mencionó en la sección anterior, para que el proceso de abstracción pueda llevarse a cabo es necesario darle tratamiento a los diagramas de flujo de tareas, el software indicado para ello son los compiladores. Un compilador es un programa que traduce de un lenguaje a otro. Un compilador toma como entrada un programa escrito en un lenguaje fuente y produce un programa equivalente escrito en un lenguaje objetivo [1].

Por lo regular, el lenguaje fuente es un lenguaje de alto nivel tal como C o C++, mientras que el lenguaje objetivo es código objeto (también llamado en ocasiones código máquina) para la máquina objetivo, es decir, código escrito en las instrucciones de máquina correspondientes a la computadora en la cual se ejecutará.

Los compiladores resultan ser muy complejos y aunque requieren un gran esfuerzo el implementarlos estos se utilizan en casi todas las formas de la computación[25]. En la figura 2.18 podemos apreciar las fases y los componentes auxiliares divididos en piezas independientes mediante las cuales se lleva a cabo un proceso de compilación (aunque en la práctica suelen implementarse de manera conjunta y no aislada) [25].

A continuación de describen brevemente cada una de las fases que describe la imagen de la figura 2.18.

- Analizador léxico: Efectúa la lectura del programa fuente, recolecta secuencias de caracteres en unidades significativas denominadas tokens.
- Analizador sintáctico (parser): El analizador sintáctico recibe el código fuente en forma de tokens provenientes del analizador léxico y realiza el análisis sintáctico el cual determinará los elementos estructurales del programa y sus relaciones, dando como resultado un árbol sintáctico.
- Analizador semántico: Trata de determinar el tipo de los resultados intermedios, comprobar que los argumentos que tiene un operador pertenecen al conjunto de los operadores posibles y si son compatibles entre sí, etc. En definitiva, comprobará que el significado de lo que se va leyendo es válido.
- Optimizador de código fuente: Los compiladores a menudo incluyen varias etapas para el mejoramiento u optimización del código. Sin embargo, también puede referirse de manera general a cualquier representación interna para el código fuente utilizado por el compilador.

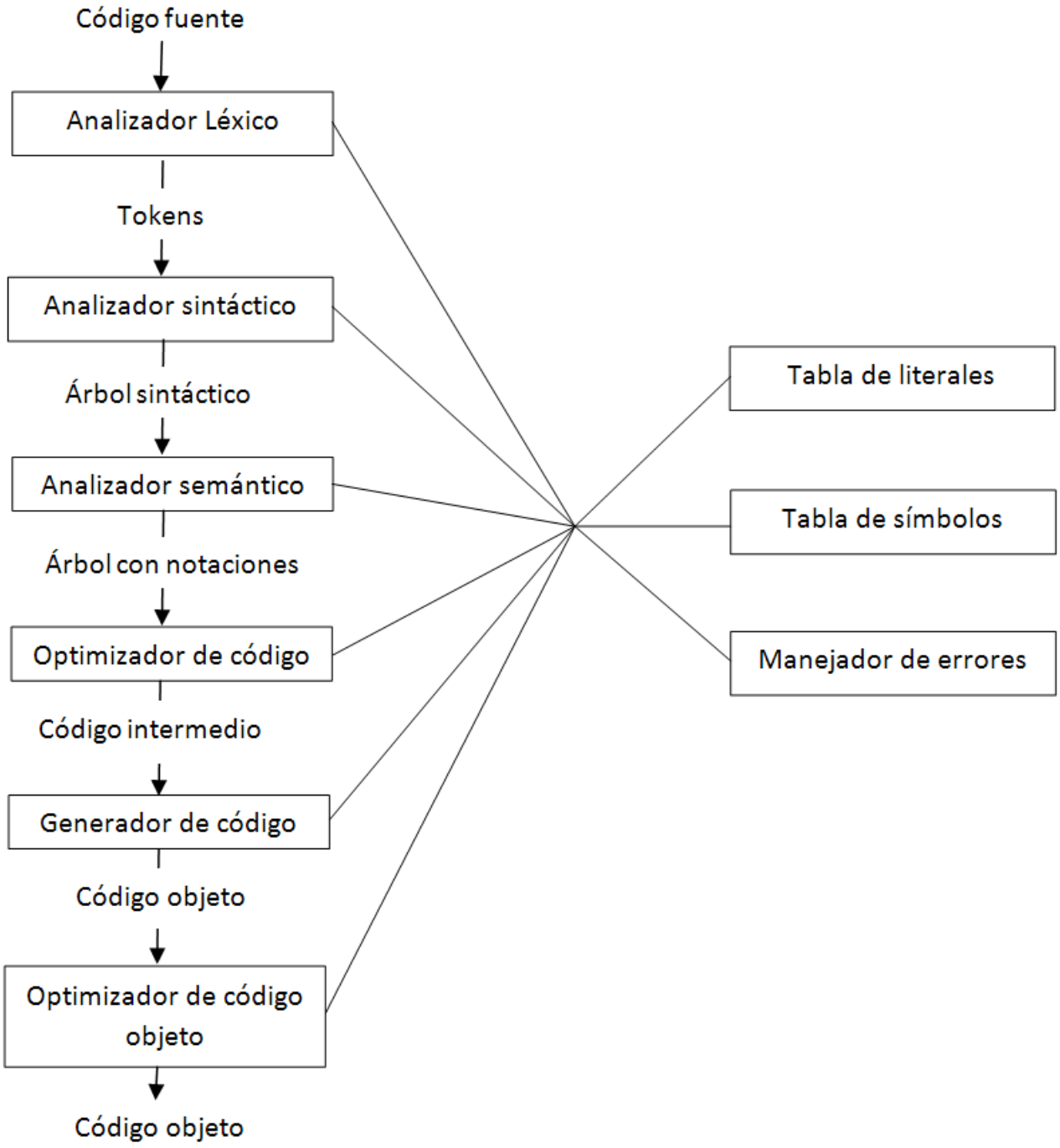


Figura 2.18: Fases de un compilador.

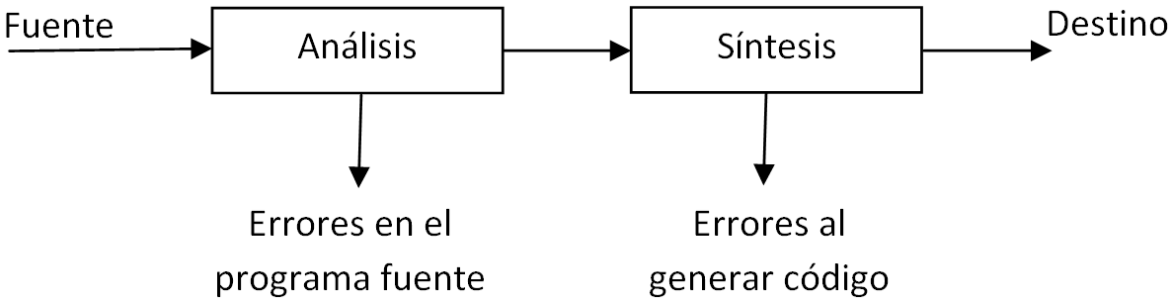


Figura 2.19: Esquema por etapas de un traductor.

- Generador de código: El generador de código toma el código intermedio o RI y genera el código para la máquina objetivo.
- Optimizador de código objetivo: En esta fase el compilador intenta mejorar el código objetivo generado en la etapa anterior. Dichas mejoras incluyen la selección de modos de direccionamiento para mejorar el rendimiento.

Debido a la naturaleza del proyecto, ciertos parámetros no se ajustan al sentido tradicional de un compilador, siendo de manera general un traductor.

2.8.1. Traductor

Aunque podemos hablar de forma general de traductores (englobando compiladores e interpretes) y definirlo como un programa que traduce o convierte desde un texto o programa escrito en un lenguaje fuente hasta un texto o programa equivalente escrito en un lenguaje destino produciendo mensajes de error [25]. Un traductor se divide en dos etapas:

- Análisis: Analiza la entrada y genera estructuras intermedias.
- Síntesis: Sintetiza la salida a partir de dichas estructuras.

Por lo tanto, el esquema de un traductor se presenta en la figura 2.19.

Básicamente los objetivos de la etapa de análisis son:

- Controlar la corrección del programa fuente.
- Generar las estructuras necesarias para comenzar la etapa de síntesis.

La etapa de análisis consta de las siguientes fases:

- Análisis lexicográfico: Divide el programa fuente en los componentes básicos del lenguaje a compilar. Cada componente básico es una subsecuencia de caracteres del programa fuente, perteneciente a una categoría gramatical.
- Análisis sintáctico: Comprueba que la estructura de los componentes básicos sea las correcta según las reglas gramaticales del lenguaje que se compila.
- Análisis semántico: Comprueba que el programa fuente respeta las directrices del lenguaje que se compila (todo lo relacionado con el significado) revisión de tipos, rangos de valores, existencia de variables, etc.

Cualquiera de estas fases puede emitir mensajes de error derivados de fallas cometidos por el programador en la redacción de los textos fuente. Mientras más errores controle un compilador, menos problemas dará un programa en tiempo de ejecución [39].

La etapa de síntesis construye el programa objeto deseado (equivalente semánticamente al fuente) a partir de las estructuras generadas por la etapa de análisis. Para ello se compone de tres fases fundamentales:

- Generación del código intermedio.
- Generación del código máquina.
- Fase de optimización.

Actualmente la tendencia de los sistemas es construir software cada vez más grande y poderoso, y no sólo eso sino que además los clientes quieren que éste se construya lo más rápido posible. Por tal motivo, los desarrolladores se enfrentan a los problemas del análisis y diseño del software que tienen mucho que ver con las dificultades al momento de coordinar las múltiples actividades que envuelve un proyecto de desarrollo de software [2]. Debido a lo anterior, se utilizó como método y proceso de desarrollo a OpenUP basado en el patrón Modelo-Vista-Controlador.

2.9. OpenUP

2.9.1. ¿Qué es OpenUP?

OpenUP es un método y un proceso de desarrollo de software propuesto por un conjunto de empresas de tecnología quienes lo donaron en el año 2007 a la Fundación Eclipse. La fundación lo ha publicado bajo una licencia libre y lo mantiene como método de ejemplo dentro del proyecto Eclipse Process Framework.

Es un proceso mínimo, suficiente y extensible [17]. Detallando estos tres puntos tenemos:

- Mínimo: Sólo incluye el contenido del proceso fundamental.
- Suficiente: Puede ser utilizado como proceso entero para construir un sistema.
- Extensible: Puede ser utilizado como base para agregar o adaptar más procesos.

Se fundamenta en 4 principios básicos:

- Se centra en la arquitectura de forma temprana para tener un buen desarrollo y así minimizar el riesgo en el proyecto.
- Desarrollo evolutivo para tener retroalimentación y poder llevar a cabo mejoras contínuas.
- Motivación a la colaboración y convergencia de intereses para que de esta manera se pueda compartir conocimiento.
- Equilibra las prioridades para maximizar el beneficio obtenido por los involucrados en el proceso de desarrollo (*stakeholders*).

Mediante OpenUP se obtienen beneficios tales como:

- Disminución de las posibilidades de fracaso del proyecto, por tal motivo incrementa las probabilidades de éxito.
- Detección de errores tempranos a través de un ciclo iterativo.

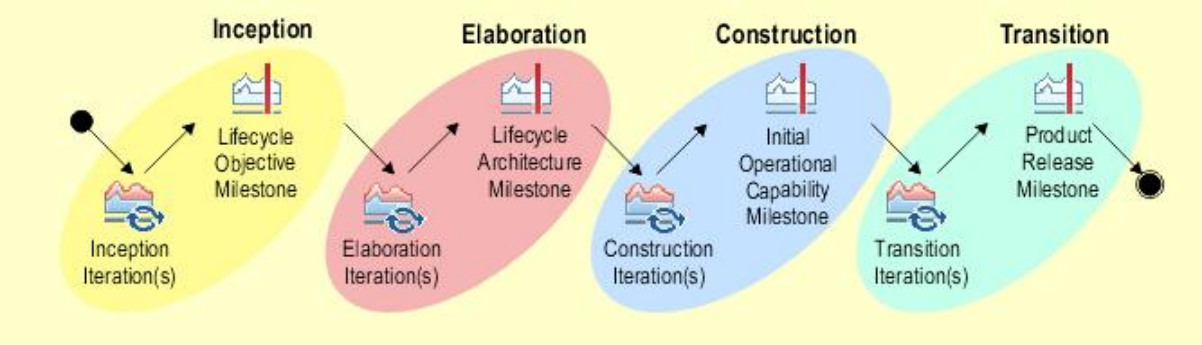


Figura 2.20: Proceso de entrega para el ciclo de Vida del OpenUP/Basic [18].

- Decremento en la elaboración de documentación, diagramas e iteraciones innecesarias requeridos por la metodología RUP.
- Como metodología ágil tiene un enfoque centrado en el cliente y establece iteraciones cortas.

2.9.2. Proceso de entrega

Como proceso de entrega define un proceso de desarrollo de software que se fundamenta en los principios básicos de OpenUP. Está diseñado para funcionar en equipos pequeños - de 3 a 6 integrantes - trabajando sobre proyectos de corta duración, la figura 2.20 muestra el proceso de entrega [17].

A continuación se detallan las actividades llevadas a cabo en las diferentes fases del proceso de entrega.

Concepción

- Realizar el planteamiento del problema.
- Elaborar la justificación.
- Definir objetivos generales y objetivos específicos.
- Obtener la información relacionada con el proyecto.
- Capturar las necesidades de los stakeholders en los objetivos del ciclo de vida para el proyecto.

Elaboración

- Definir los riesgos significativos para la arquitectura.
- Establecer la base para la elaboración de la arquitectura.

Construcción

- Diseñar, implementar y realizar las pruebas de las funcionalidades para desarrollar el sistema completo.
- Finalizar el desarrollo del sistema basado en la arquitectura definida.

Transición

- Asegurar que el sistema sea entregado a los usuarios.
- Evaluar la funcionalidad y rendimiento del último entregable de la fase de construcción.

.

El siguiente capítulo describe el proceso llevado a cabo durante el análisis y diseño del plug-in. El capítulo inicia haciendo una breve reseña del proceso de experimentación previa que antecedió al desarrollo del proyecto de investigación.

Capítulo 3

Análisis y diseño del plug-in

3.1. Introducción

En el presente capítulo se describen los requerimientos del plug-in, los pasos previos que se dieron y ayudaron a realizar la fase de análisis, así como los elementos del diseño, de igual forma se presenta la estructura funcional de las clases del sistema, los casos de uso de acuerdo al lenguaje UML y la aplicación del patrón Modelo-Vista-Controlador.

3.2. Análisis del sistema

En el análisis del sistema se involucra la descripción de los conceptos implementados en el sistema, los cuales definen su funcionalidad, la descripción de los requerimientos funcionales del sistema y del usuario[24].

De acuerdo a lo descrito en la sección 2.7.5 se tiene la necesidad de una herramienta que facilite el proceso de traducción de los diagramas de flujos de tareas a su respectiva *álgebra de tareas*. La herramienta actual descrita en la sección 1.4.2 se encuentra en fase de prototipo, por tal motivo la generación del álgebra y su posterior uso en el compilador que generá las trazas no se puede realizar de manera óptima.

Los diagramas de flujos de tareas no tienen una representación visual estricta, específicamente el componente *Choice*, ya que este puede formar parte de tres estructuras distintas, es decir, comportarse como un *Or*, representar un ciclo mientras o representar un ciclo hasta.

La meta del diseño de los diagramas de flujos de tareas, es aprovechar el comportamiento de los diagramas de actividad de UML, con la intención de brindar una facilidad a los usuarios y tener una mejor aceptación y difusión.

Esta decisión tiene puntos a favor y en contra, se puede tomar como ejemplo práctico el lenguaje HTML. *"HTML es muy flexible y los exploradores HTML aceptan cualquier cosa que se parezca a HTML, esta característica ayudó a la temprana adopción de HTML, pero ahora es un problema"* la solución para esta problemática en particular fue la aparición de lenguajes más estrictos [5].

En resumen, implementar la automatización de la traducción de un diagrama no estructurado,

significa no poder utilizar métodos o técnicas que si podrían utilizarse en diagramas estructurados, por tal motivo se debe emplear otro tipo de técnica.

3.2.1. Descripción general

El plug-in para la generación del álgebra de tareas implementa las siguientes actividades:

- Creación y edición de diagramas.
  - Los usuarios del plug-in podrán crear uno o varios diagramas de flujos de tareas, así como su posterior edición, todo ello mediante un entorno visual.
- Revisión del diagrama.
  - Se realizará una revisión para detectar si un diagrama se puede traducir a su respectiva álgebra de tareas, en caso de que se encuentren inconsistencias se mostraran al usuario .
- Generación del álgebra de tareas.
  - Se realizará la traducción literal de un diagrama, en caso de que esta no pueda ser generada, se mostrarán al usuario los posibles problemas que impiden generarla.
- Exportación
  - El diagrama creado será traducido al estándar XMI.

3.2.2. Requerimientos funcionales

1. Creación y edición de diagramas.
   a) Creación de diagramas. Mostrar un área de trabajo vacía y una paleta de componentes, dicha paleta contendrá los componentes visuales del diagrama de flujos de tareas, es decir, Flujo, *Start, Task, Fork, Join, Exception, Failure, Choice* y *End*.
   b) Edición de diagramas. Mostrar un diagrama específico creado con anterioridad para su posible edición, contemplando la misma paleta que en la actividad creación de diagramas.
2. Revisión del diagrama.
   a) Se revisarán los flujos existentes en el diagrama, los que no sean correctos serán reportados al usuario en un área específica en donde se darán a conocer los posibles problemas encontrados.
3. Generación del álgebra de tareas.
   a) Se realizará la traducción literal del diagrama, en caso de que no pueda llevarse a cabo una traducción, los problemas encontrados serán reportados en un área específica destinada para ello.

b) Cuando un diagrama se traduzca correctamente, se abrirá (en la zona de trabajo) un archivo nuevo, el cual contendrá el álgebra respectiva lista para ser enviada al compilador.

4. Exportación

   a) El usuario seleccionará una ruta en donde el diagrama activo será exportado en formato XMI.

3.2.3. Requerimientos no funcionales

El plug-in debe cumplir con ciertas características como son:

1. Uso similar a Eclipse: El sistema contará con una interfaz integrada a Eclipse, con ello se simplificará su uso al aprovechar el conocimiento previo del IDE.
2. Interfaz amigable: El sistema tendrá una interfaz que facilite la comprensión de la creación de los diagramas.
3. Interacción adecuada: El sistema requerirá un conocimiento básico del lenguaje UML, en específico el diagrama de actividades. Esto facilitará en gran medida el proceso de interacción.
4. Rapidez: El sistema realizará la tarea específica en el menor tiempo posible.

3.2.4. Requerimientos del sistema

El plug-in funciona dentro del IDE Eclipse, se recomienda la edición de modelado (*Eclipse Modeling Tools*) sin embargo, puede funcionar con otras ediciones, las cuales deben contener a los plug-ins GMF y EMF.

El cuadro 3.1 especifica los requerimientos mínimos del sistema.

Componente	Características
Sistema operativo	*Windows* xp, *Windows* Vista, *Windows Seven*, Mac OS, linux
java Runtime Environment	JRE-ES 7.0
IDE Eclipse	3.1 o superior.
plug-ins	GMF 3.0.1, EMF 2.0

Cuadro 3.1: Requerimientos mínimos del sistema

3.3. Experimentación previa

Antes de iniciar formalmente el proyecto, se llevó a cabo una etapa de experimentación con las tecnologías involucradas, es decir, se probaron diversas funcionalidades que GMF y EMF proporcionaban, así como otros *frameworks* necesarios para el desarrollo de plug-ins.

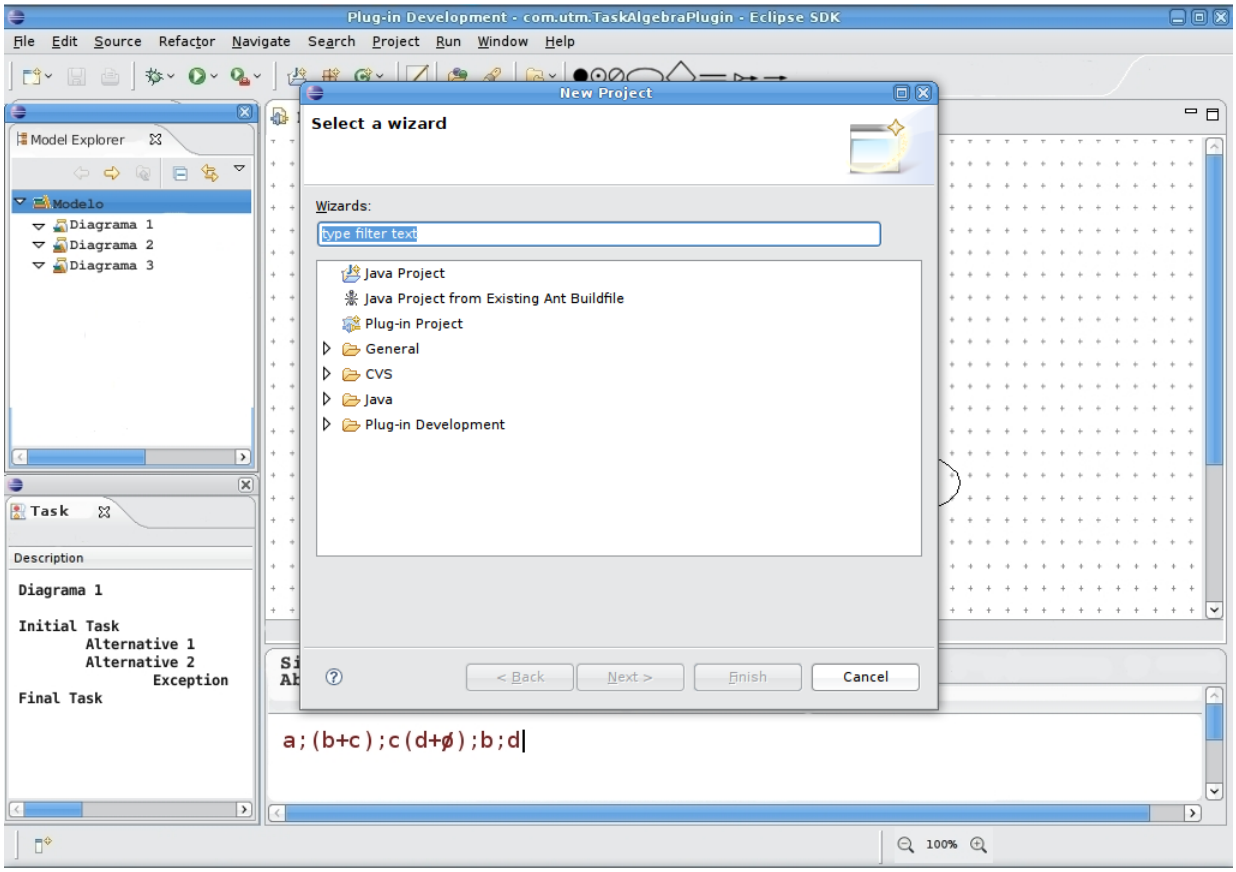


Figura 3.1: Prototipo resultante de la experimentación con las herramientas de desarrollo.

Como resultado, se llegó a la conclusión de que ciertas funciones elementales para el adecuado funcionamiento de la herramienta se cumplían de forma automática, esto por las características que los *frameworks* y otros plug-ins aportan al desarrollo del proyecto. El prototipo resultante se puede apreciar en la figura 3.1, las características utilizadas se presentan en el cuadro 3.2.

3.3.1. Análisis para el editor de diagramas de flujos de tareas

La construcción de los diagramas se lleva a cabo de forma visual y se almacenan en archivos que residen dentro de proyectos creados para tal fin. Desafortunadamente, la herramienta entrega información en archivos XML que no contienen una estructura que facilite su manejo, tal como se observa en el cuadro 3.4 en donde se muestra información correspondiente a la estructura lógica del diagrama identificada con la extensión **"*.tfe"** y el cuadro 3.5, el cual muestra un extracto de la información visual correspondiente a formas, colores y otras etiquetas que no afectan a la lógica del diagrama, este archivo se identifica con la extensión **"*.tfd"**.

Una vez que se ha expuesto la estructura de los archivos de los cuales se obtendrá información, así como la problemática que estos presentan para abstraer información, se llegó a la conclusión de que las primeras pretensiones de obtener directamente un árbol estructurado no llegarían a buen termino. Sin embargo, una vez que se tiene el conocimiento de la forma en que

Herramienta	Funcionalidad
plug-in Development Enviroment	Herramientas para la creación, desarrollo, depuración y despliegue de artefactos para el plug-in (menú, botones, barras de estado, etc).
EMF	Soporte del modelo de datos a un modelo visual.
GMF	Soporte para la implementación básica del modelo visual.

Cuadro 3.2: Características identificadas como resultado de la experimentación con las herramientas de desarrollo.

NOMBRE	TIPO	ID	OP1	OP2	ENTRA T	SAL T	MARCA
0	1	2	3	4	5	6	7

Figura 3.2: Estructura dinámica para abstraer los datos de los componentes.

el plug-in presenta la información, se propusieron dos estructuras dinámicas las cuales contendrán información de los componentes del diagrama, así como de los flujos que lo componen.

Para simplificar el trabajo con los componentes gráficos, se asigna a cada uno de ellos un valor único que lo asociará con un tipo. A continuación se muestra el cuadro 3.3 donde se describe esta asignación.

Componente visual	Nombre	Identificador numérico
	*Start*	0
	*Task*	1
	*Fork*	2
	*Join*	3
	*Exception*	4
	*Failure*	5
	*Choice*	6
	*End*	7

Cuadro 3.3: Asignación de identificadores numéricos a cada uno de los siete tipos de componentes.

Las imágenes de las figuras 3.2 y 3.3 muestran el formato de las estructuras dinámicas propuestas.

A continuación se describen los datos que almacenará cada uno de los campos de la estructura 3.2.

```
<?XML version="1.0" encoding="UTF-8"?>
<taskflow:ShapeDiagram XMI:version="2.0" XMLns:XMI="http://www.omg.org/XMI"
XMLns:xsi="http://w3.org/2001/XMLSchema-instance" XMLns:taskflow="http://taskflow/1.0">
<hasComponents xsi:type="taskflow:Start"/>
<hasComponents xsi:type="taskflow:Choice" condv="2" condf="1"/>
<hasComponents xsi:type="taskflow:Task" task="tarea 2"/>
<hasComponents xsi:type="taskflow:Task" task="tarea 3"/>
<hasComponents xsi:type="taskflow:Choice" condv="4" condf="3"/>
<hasComponents xsi:type="taskflow:Task" task="tarea 5"/>
<hasComponents xsi:type="taskflow:Task" task="tarea 6"/>
<hasComponents xsi:type="taskflow:End"/>
<hasComponents xsi:type="taskflow:Failure"/>
<hasFlow source="//@hasComponents.0" target="//@hasComponents.1"/>
<hasFlow source="//@hasComponents.1" target="//@hasComponents.3"/>
<hasFlow source="//@hasComponents.1" target="//@hasComponents.2"/>
<hasFlow source="//@hasComponents.3" target="//@hasComponents.4"/>
<hasFlow source="//@hasComponents.4" target="//@hasComponents.8"/>
<hasFlow source="//@hasComponents.4" target="//@hasComponents.5"/>
<hasFlow source="//@hasComponents.2" target="//@hasComponents.6"/>
<hasFlow source="//@hasComponents.6" target="//@hasComponents.7"/>
<hasFlow source="//@hasComponents.5" target="//@hasComponents.6"/>
</taskflow:ShapeDiagram>
```

Cuadro 3.4: Información abstracta de un diagrama de flujo de tareas entregado por el plug-in

```
<?XML version="1.0" encoding="UTF-8"?>
<notation:Diagram XMI:version="2.0" XMLns:XMI="http://www.omg.org/XMI"
XMLns:notation="http://www.eclipse.org/gmf/runtime/1.0.2/notation" XMLns:taskflow="http://taskflow/1.0"
XMI:id="_yncRQTj1EeKuNv7zTDZ3iA" type="Taskflow" name="pr_or_t1.tfd" measurementUnit="Pixel">
<children XMI:type="notation:Node" XMI:id="_1_rYYDj1EeKuNv7zTDZ3iA" type="2001">
<styles XMI:type="notation:DescriptionStyle" XMI:id="_1_rYYTj1EeKuNv7zTDZ3iA"/>
<styles XMI:type="notation:FontStyle" XMI:id="_1_rYYjj1EeKuNv7zTDZ3iA" fontName="Sans"/>
<element XMI:type="taskflow:Start" href="pr_or_t1.tfe#//@hasComponents.0"/>
<layoutConstraint XMI:type="notation:Bounds" XMI:id="_1_rYYzj1EeKuNv7zTDZ3iA" x="525" y="20"/>
</children>
<children XMI:type="notation:Node" XMI:id="_3NdaIDj1EeKuNv7zTDZ3iA" type="2002">
<children XMI:type="notation:DecorationNode" XMI:id="_3NeoQDj1EeKuNv7zTDZ3iA" type="5001">
<layoutConstraint XMI:type="notation:Location" XMI:id="_3NeoQTj1EeKuNv7zTDZ3iA" y="5"/>
</children>
<styles XMI:type="notation:DescriptionStyle" XMI:id="_3NdaITj1EeKuNv7zTDZ3iA"/>
<styles XMI:type="notation:FontStyle" XMI:id="_3NdaIjj1EeKuNv7zTDZ3iA" fontName="Sans"/>
<element XMI:type="taskflow:Choice" href="pr_or_t1.tfe#//@hasComponents.1"/>
<layoutConstraint XMI:type="notation:Bounds" XMI:id="_3NdaIzj1EeKuNv7zTDZ3iA" x="540" y="65"/>
</children>
<children XMI:type="notation:Node" XMI:id="_4rsSsDj1EeKuNv7zTDZ3iA" type="2005">
<children XMI:type="notation:DecorationNode" XMI:id="_4rs5wDj1EeKuNv7zTDZ3iA" type="5003"/>
<styles XMI:type="notation:DescriptionStyle" XMI:id="_4rsSsTj1EeKuNv7zTDZ3iA"/>
<styles XMI:type="notation:FontStyle" XMI:id="_4rsSsjj1EeKuNv7zTDZ3iA" fontName="Sans"/>
<element XMI:type="taskflow:Task" href="pr_or_t1.tfe#//@hasComponents.2"/>
<layoutConstraint XMI:type="notation:Bounds" XMI:id="_4rsSszj1EeKuNv7zTDZ3iA" x="615" y="65"/>
</children>
...
<edges XMI:type="notation:Edge" XMI:id="_EyeeUDj2EeKuNv7zTDZ3iA" type="4001" source="_7M-
jsDj1EeKuNv7zTDZ3iA" target="_8Y4WIDj1EeKuNv7zTDZ3iA">
<styles XMI:type="notation:RoutingStyle" XMI:id="_EyeeUTj2EeKuNv7zTDZ3iA"/>
<styles XMI:type="notation:FontStyle" XMI:id="_EyeeUjj2EeKuNv7zTDZ3iA" fontName="Sans"/>
<element XMI:type="taskflow:Flow" href="pr_or_t1.tfe#//@hasFlow.5"/>
<bendpoints XMI:type="notation:RelativeBendpoints" XMI:id="_EyeeUzj2EeKuNv7zTDZ3iA" points="[7, 0, -
58, -26]$[72, 0, 7, -26]$[72, 26, 7, 0]"/>
<sourceAnchor XMI:type="notation:IdentityAnchor" XMI:id="_EyivwDj2EeKuNv7zTDZ3iA"
id="(0.7666666666666667,0.4666666666666667)"/>
<targetAnchor XMI:type="notation:IdentityAnchor" XMI:id="_EyivwTj2EeKuNv7zTDZ3iA"
id="(0.4714285714285714,0.0)"/>
</edges>
<edges XMI:type="notation:Edge" XMI:id="_GviA4Dj2EeKuNv7zTDZ3iA" type="4001" sour-
ce="_4rsSsDj1EeKuNv7zTDZ3iA" target="_9H0fwDj1EeKuNv7zTDZ3iA">
<styles XMI:type="notation:RoutingStyle" XMI:id="_GviA4Tj2EeKuNv7zTDZ3iA"/>
<styles XMI:type="notation:FontStyle" XMI:id="_GviA4jj2EeKuNv7zTDZ3iA" fontName="Sans"/>
<element XMI:type="taskflow:Flow" href="pr_or_t1.tfe#//@hasFlow.6"/>
<bendpoints XMI:type="notation:RelativeBendpoints" XMI:id="_GviA4zj2EeKuNv7zTDZ3iA" points="[0, 0, 91,
-140]$[-74, 133, 17, -7]"/>
<sourceAnchor XMI:type="notation:IdentityAnchor" XMI:id="_GvkdIDj2EeKuNv7zTDZ3iA"
id="(0.44285714285714284,1.0)"/>
<targetAnchor XMI:type="notation:IdentityAnchor" XMI:id="_GvkdITj2EeKuNv7zTDZ3iA"
id="(0.7571428571428571,0.26666666666666666)"/>
</edges>
</notation:Diagram>
```

Cuadro 3.5: Información visual de un diagrama de flujo de tareas entregado por el plug-in

TIPO	ORIGEN	TIPO	DESTINO	RAMA	MARCA	ENT_T	SAL_T
0	1	2	3	4	5	6	7

Figura 3.3: Estructura dinámica para abstraer los datos de los flujos del diagrama

- NOMBRE: El nombre del componente gráfico con un valor de texto.
- TIPO: El tipo de identificador conforme al cuadro 3.3.
- ID: Identificador único asignado por el plug-in.
- OP1: Campo destinado a información introducida por el usuario, en este caso para los componentes *Task*, *Choice* y *Exception*.
- OP2: Campo destinado a información introducida por el usuario, en este caso para los componentes *Choice*, *Exception*.
- ENTRA T: Número total de entradas del componente.
- SAL T: Número total de salidas del componente.
- MARCA: Campo auxiliar libre y sin representación de información.

Ahora se detallará la información que contendrá la estructura dinámica 3.3 que representa los flujos del sistema.

- TIPO: El tipo de identificador del componente origen conforme el cuadro 3.3.
- ORIGEN: Identificador único del componente origen.
- TIPO: El tipo de identificador del componente destino conforme el cuadro 3.3.
- DESTINO: Identificador único del componente destino.
- RAMA: Campo para validación de ramas.
- MARCA: Campo auxiliar libre y sin representación de información.
- ENT_T: Número total de entradas del componente.
- SALT_: Campo auxiliar libre y sin representación de información.

Una vez que se tienen capturados los datos en las estructuras anteriores, pueden manipular los diagramas y la información que contiene.

3.3.2. Archivos de configuración básicos de un plug-in

Existen dos archivos que contendrán información sobre el comportamiento y las capacidades del plug-in, los cuales son plug-in.XML y MANIFEST.mf. En ellos se declaran las dependencias y servicios de los cuales se compondrá el plug-in.

En el archivo de manifiesto o MANIFEST.mf, se definen el nombre, versión y las dependencias (*require-bundle*) del plug-in a desarrollar. Para nuestro caso en específico los *frameworks* EMF y GMF. Dos de las entradas principales de la estructura de manifiesto son:

- *Bundle-Activator*. Es una clase que nos permite acceder a los recursos estáticos del plug-in, preferencias u otra información específica sobre el estado del mismo.
- *Bundle-ActivationPolicy*. Permite indicar si se necesita cargar el plug-in al comienzo de

Eclipse.

El archivo plug-in.XML debe ser único en el plug-in. Es el archivo que le indica a Eclipse qué contiene el plug-in. En particular, le indica qué puntos de extensión tiene, ya que en él se puede definir uno o más puntos de extensión. Es bajo esta característica que se indica la clase que implementará toda la lógica del proyecto, es decir el funcionamiento de la parte controlador.

En la siguiente sección se describe el análisis del editor visual y posteriormente el análisis de las clases necesarias para implementar la parte "controlador" del patrón Modelo-Vista-Controlador.

3.3.3. Diseño del editor visual

Como se mencionó en la sección 2.7.6 para desarrollar la parte "vista" correspondiente al patrón Modelo-Vista-Controlador, se requiere diseñar un modelo estructurado descrito bajo la especificación XMI. La aplicación resultante brinda soporte mediante la implementación de clases adaptadoras en lenguaje java, las cuales sirven como base para la implementación automatizada de la interfaz gráfica de usuario (GUI). El editor resultante construido mediante GMF se integra a Eclipse, siendo necesaria la personalización adecuada para el proyecto. Hasta este punto, el editor generado con EMF y GMF consta de los siguientes componentes:

- Barras de herramientas (aún sin personalización de íconos).
- Figuras con representación gráfica derivada de un modelo de datos (sin la debida correspondencia con los componentes del diagrama de flujo de tareas).
- Mapeo de figuras y su significado en el modelo de datos.
- Comandos con las sentencias hacer y deshacer.

El modelo de datos necesario para llevar a cabo esta parte del análisis se crea conforme a un diagrama de clases UML, el cual se muestra en la figura 3.4; cabe resaltar que después de diseñar el modelo de datos correcto (diagrama de clases UML) bastará con una personalización adecuada para tener todos y cada uno de los componentes del diagrama de flujo de tareas.

Hasta este punto contamos con un editor visual completo pero no personalizado, para su posterior comunicación con el editor mediante clases adaptadoras.

3.3.4. Arquitectura del plug-in patrón Modelo-Vista-Controlador

Como se vio en la sección 3.3.3 después de generar el editor visual mediante el modelo estructurado, el diseño restante para finalizar el proyecto debe corresponder a la arquitectura típica de los plug-ins de eclipse, es decir, se requiere integrar nuestro desarrollo a las clases adaptadoras implementadas automáticamente. La figura 3.5 muestra dicha arquitectura.

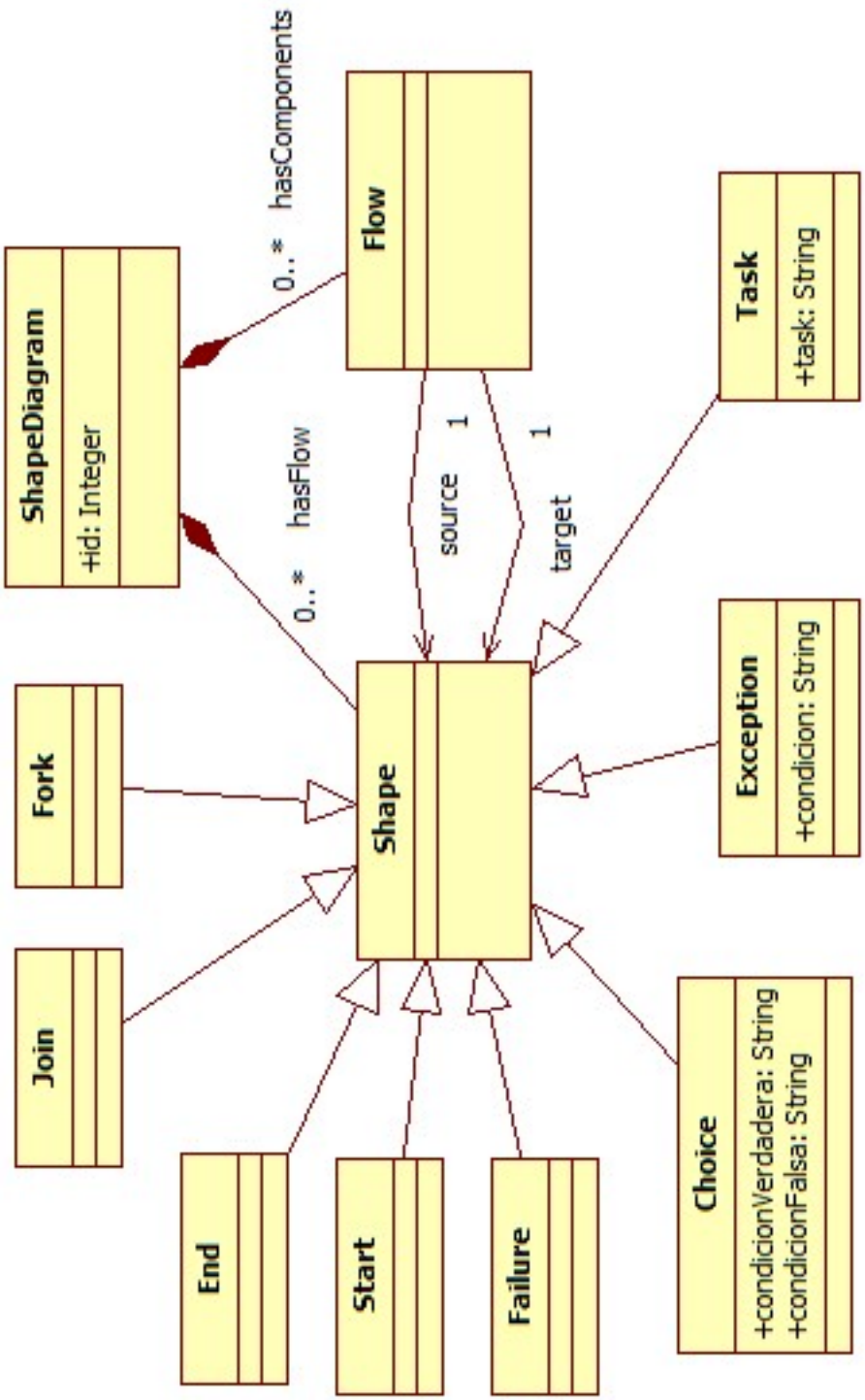


Figura 3.4: Diagrama de clases UML, como modelo estructurado para EMF y GMF.

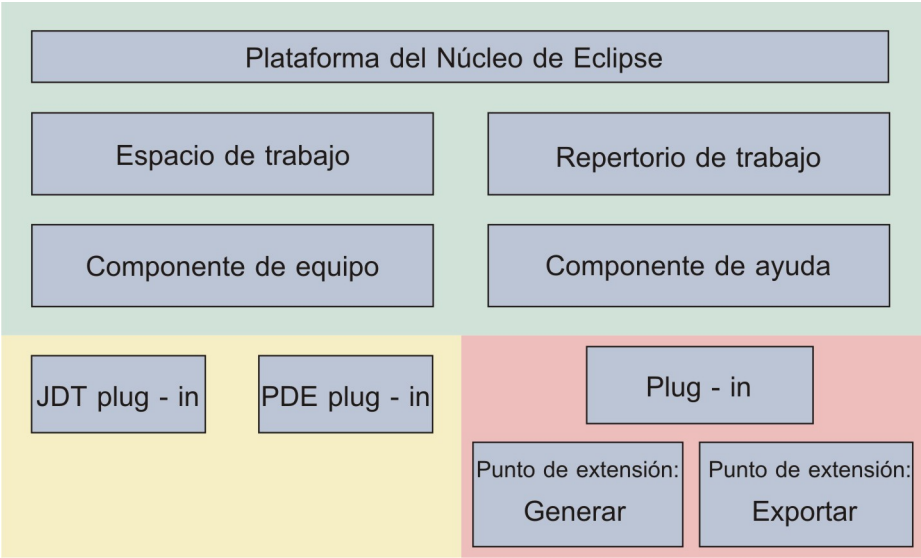


Figura 3.5: Arquitectura de Eclipse, plug-ins característicos y vista del plug-in dentro de la arquitectura.

Sin embargo, como se mencionó en la sección 3.3.2 se requieren de puntos de extensión para tener un comportamiento específico. En este caso, el acceso a los archivos que contienen la información referente al diagrama de flujos de tareas descritos en la sección 3.3.1.

Un punto de extensión básicamente es un elemento de la plataforma que puede ser usado y personalizado por cualquier plug-in. Ejemplos de puntos de extensión son las vistas y *wizards*. Hay puntos de extensión que no son visibles pero desempeñan alguna función que puede ser útil para todos los plug-ins. En nuestro caso se utilizarán dos puntos de extensión, uno para acceder al diagrama y el otro para exportar, nombrándose "Generar álgebra" y "Exportar a XMI" respectivamente. Por lo tanto será necesaria la implementación de dos clases adaptadoras.

Para que estas clases adaptadoras puedan llevar a cabo su función, es necesaria la comunicación con el espacio de trabajo actual, para ello es indispensable el uso de una clase especial, en dicha clase se importarán los paquetes necesarios para realizar la comunicación de las clases adaptadoras y el espacio de trabajo activo. La sección 3.7 detalla estas y otras clases.

3.4. Diagrama de flujos de tareas y su naturaleza no estructurada

Como se mencionó en la sección 3.2, el plug-in mantiene su simplicidad en cuanto al diseño de los diagramas de flujos de tareas, sin embargo, surge una problemática dada esta misma simplicidad, la causa es que se mantiene al elemento *Choice*, descrito en la sección 2.7.5 y con representación gráfica mostrada en la imagen de la figura 3.3, como un componente a partir del cual se pueden definir tres estructuras para la sintaxis del álgebra de tareas, dichas estructuras son:

- Selección binaria (figura 2.13).

- Ciclo *Until* (figura 2.16) y.
- Ciclo *While* (figura 2.15).

La generación de la respectiva álgebra no pudo realizarse en una pasada, ya que para llevar a cabo el proceso completo destinado a obtener esto es necesario realizar varias pasadas. Esta forma de generar el álgebra de tareas se debe a que se necesita analizar la ramificación completa de un diagrama de flujo de tareas para así detectar qué estructura de la sintaxis del álgebra de tareas representa.

Una explicación es que se entra en conflicto con el teorema de la estructura propuesto en 1966 por Bömh y Jacopini, que se ratificó con los trabajos de Charlan D. Mills [9]. Un bosquejo del algoritmo diseñado y parcialmente implementado para traducir en una sola pasada se muestra en el apéndice A.

Otra opción (la cual se analizó), fue crear una barra de estructuras completas, emulando la funcionalidad implementada en la herramienta DFD, es decir, en dicha barra se colocarían componentes estructurales con una representación formal en el álgebra de tareas, tales como, los ciclos *While* y *Until*, así como las estructuras *Or* y *Exception*.

El comportamiento de las estructuras que maneja la herramienta DFD se muestra en la figura 3.6. Al colocar este tipo de barra de estructuras, se tendría una implementación reducida, ya que las posibilidades de crear un diagrama erróneo serían mínimas, sin embargo, este diseño se enfrenta a dos obstáculos, el primero es que se rompe con el esquema de componentes tradicionales del diagrama de flujo de tareas del Método *Discovery*, es decir, construir estructuras definidas en la sintaxis del álgebra de tareas a partir de componentes básicos individuales. El segundo, es la falta de soporte para llevar a cabo una implementación de este tipo, ya que los *framework* GMF y EMF de Eclipse no poseen herramientas para llevar a cabo un proyecto con tales características.

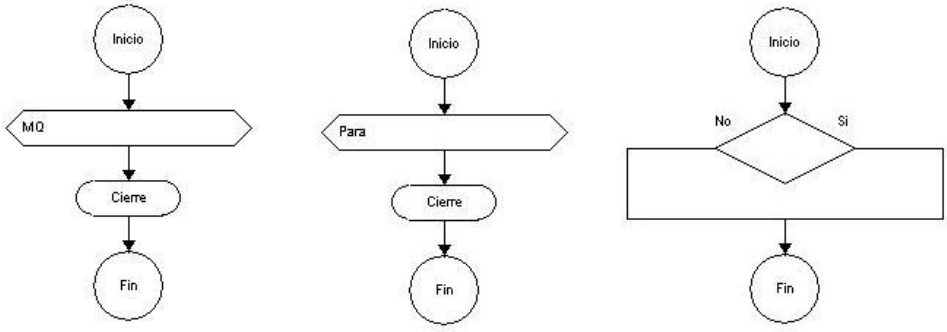


Figura 3.6: Estructuras utilizadas en la herramienta DFD.

Por lo anterior, se optó por la vía de reducción de expresiones, es decir, pasar una cantidad indeterminada de veces por las estructuras abstractas 3.3 y 3.2 del diagrama, para tratar de formar estructuras completas con los componentes hasta llegar a tener un diagrama elemental formado por un inicio y un componente terminal.

La figura 3.7 muestra las estructuras identificadas para formar el álgebra de tareas, dentro de esta se observan componentes marcados como X_n, estos componentes hacen referencia a reducciones llevadas a cabo con anterioridad. Por ejemplo, cualquiera de las estructuras marcadas como *Or* Terminal Doble, puede representarse como un componente X_n. De forma similar las

estructuras *Or* Terminal Vacío y *Or* Terminal Simple, contienen un elemento marcado como F, dicho elemento puede representar algún símbolo terminal, ya sea *End* o *Failure*.

De esta manera fue como se automatizó la traducción del diagrama de flujos de tareas al álgebra de tareas. En las siguientes secciones se muestra el estudio completo mediante el cual se diseño la herramienta.

3.5. Casos de uso

El proceso de identificación de los casos de uso se llevó a cabo mediante la descripción de qué es lo que se desea que la herramienta realice. A continuación se presenta dicha descripción:

El plug- in permite crear diagramas de flujos de tareas, dichos diagramas correspondan a lo descrito en la sección 2.7.4, se realizará una validación y posteriormente se procederá a generar el álgebra de tareas respectiva, tomando en cuenta para ello la sintaxis abstracta descrita en la sección 2.7.5 y sintetizada en la figura 2.17.

El plug-in deberá:

1. Colocar dentro de la pantalla de edición cualquiera de los componentes representados en la paleta de componentes del diagrama de flujo de tareas, los cuales pueden ser *Start, Task, Fork, Join, Exception, Failure, Choice, End* y *Flow*.
2. En el componente *Task*, agregar una etiqueta la cual representará la tarea a llevar a cabo.
3. En el componente *Choice*, agregar dos etiquetas, las cuales representarán las condiciones de salida.
4. En un componente *Exception*, agregar una etiqueta la cual representará la condición hacia un flujo excepcional.
5. Establecer un flujo entre dos o más componentes mediante el componente *Flow*.
6. Crear, guardar y abrir un archivo con la información del diagrama.
7. Manipular los diversos componentes para acomodar el diagrama en pantalla.
8. Usar el porta papeles para copiar, cortar y pegar los componentes del diagrama.
9. Generar imágenes a partir del diagrama.
10. Deshacer los últimos cambios hechos sobre el diagrama.
11. Rehacer los últimos cambios en el diagrama.
12. Validar el diagrama.
13. Generar el álgebra de tareas correspondiente al diagrama.
14. Exportar a un formato XMI.

Una vez que se analizó la descripción de las funcionalidades del plug-in, se determinó la existencia de sólo un usuario. Los casos de uso identificados se clasifican en dos categorías “manual” y “automatizado” ambos se muestran en la tabla 3.6. Los casos de uso “automatizado” hacen referencia a características que se cumplen automáticamente al utilizar las herramientas de creación de plug-ins, tal como se menciona en la sección 3.3.

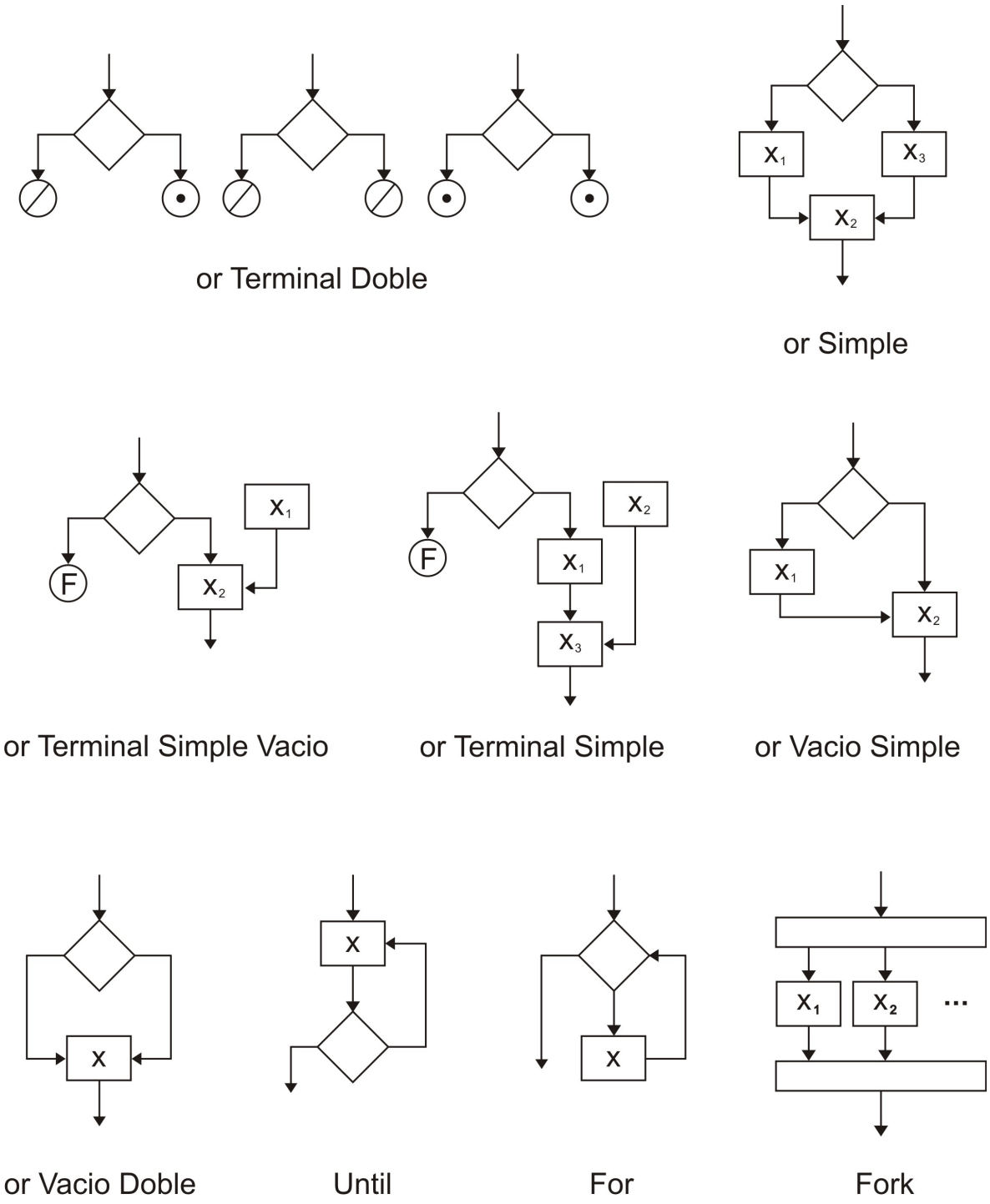


Figura 3.7: Estructuras identificadas para reducción de diagramas de flujo de tareas.

Categoría	Caso de uso
Manual	Validar diagrama.
	Reportar errores.
	Generar álgebra
	Mostrar álgebra.
	Exportar a XMI
Automatizado	Trazar componente
	crear Start
	crear Join
	crear Failure
	crear Fork
	crear End
	crear Choice
	crear Exception
	crear Task
	agregar Tarea
	modificar Tarea
	modificar Choice
	modificar Exception
	agregar Condición
	agregar Referencia
	trazar Flujo
	modificar Flujo.
	editar Componente
	borrar
	copiar
	mover
	cortar
	pegar
	abrir archivo
	crear archivo
	guardar archivo
	guardar archivo como

Cuadro 3.6: Casos de uso identificados en la descripción de las necesidades para el plug-in

Los correspondientes a la categoría “manual” se describen de la sección 3.5.1 a la sección 3.5.5. Para ello, se realiza una descripción del flujo de eventos por los que pasa, es decir, cómo comienzan, terminan e interactúan con los actores. Para la descripción de los casos de uso se utiliza la plantilla propuesta por OpenUP.

3.5.1. Caso de uso validar diagrama

Plug-in
Use-Case: Validar diagrama

1. Brief Description

   El presente caso de uso permite verificar si la estructura de un modelo de flujo de tareas es correcta.

2. Actor Brief Descriptions

   El actor involucrado en el caso de uso es el desarrollador el cual pretende comprobar su modelo de flujo de tareas.

3. Preconditions

   3.1 El proyecto debe estar activo.

   3.2 Debe existir un diagrama ya modelado.

4. Basic Flow of Events

Acción del actor	Respuesta del sistema
El caso de uso inicia cuando el usuario elige la opción de generar álgebra.	El sistema responde llamando a otra funcionalidad para generar el álgebra de un diagrama válido, en el caso de que hubiese algún error se ejecuta el flujo alterno 5.1. Se relaciona con el caso de uso: Generar código en álgebra de tareas

5. Alternative Flows

   El sistema envía los errores a otra funcionalidad que los desplegará posteriormente.

6. Subflows

   Se inicia reportar errores.

7. Key Scenarios

   Se crea información sobre errores referente a la validación.

8. Post-conditions

   La validación termina con éxito

9. Special Requirements

   Ninguno.

Cuadro 3.7: Caso de uso validar diagrama bajo el formato de OpenUp.

Para el caso de uso Validar diagrama, se tiene contemplada la validación del diagrama mostrado en el área de trabajo. Si en este proceso de validación se detectan errores, estos deberán ser mostrados al usuario. De esta manera, el usuario podrá corregirlos y volver a iniciar el proceso de validación. Sino se detectan problemas de validación, se procederá a realizar el proceso respectivo para la generación del álgebra de tareas.

3.5.2. Caso de uso reportar errores

Plug-in
Use-Case: Reportar errores.

1. Brief Description

   El caso de uso permite dar a conocer errores al usuario referentes a problemas encontrados en la generación del álgebra correspondiente a un diagrama..

2. Actor Brief Descriptions

   El actor involucrado en el caso de uso es el desarrollador el cual recibirá información sobre errores de modelado.

3. Preconditions

   3.1 El proyecto debe estar activo.

   3.2 Debe existir un diagrama ya modelado con errores.

   3.3 Los errores del diagrama deben estar agrupados.

4. Basic Flow of Events

Acción del actor	Respuesta del sistema
El caso de uso inicia cuando el usuario elige la opción de generar álgebra y se producen errores en el proceso de traducción.	El sistema responde mostrando toda la información concerniente a los errores encontrados y que impide generar una traducción literal y correcta de dicho diagrama al álgebra de tareas.

5. Alternative Flows

   El sistema muestra los errores en el apartado asignado para ello.

6. Subflows

   Ninguno.

7. Key Scenarios

   Aparecen errores referentes a la validación.

8. Post-conditions

   Los errores se muestran con éxito

9. Special Requirements

   Ninguno.

Cuadro 3.8: Caso de uso reportar errores presentado bajo el formato de OpenUp.

Para este caso de uso, el usuario obtendrá información relativa a los errores encontrados en el proceso de validación del diagrama, así como la problemática que se pueda presentar al momento de generar el álgebra de tareas. El caso de uso inicia cuando se detectan problemas en la parte de validación del diagrama, esto implica utilizar el área asignada para mostrar información de este tipo, en ella el usuario podrá observar qué elementos y flujos provocaron los errores.

3.5.3. Caso de uso generar álgebra

Plug-in
Use-Case: Generar álgebra

1. Brief Description

   El presente caso de uso permite al usuario generar el código en el álgebra de tareas equivalente al modelo de diagramas de flujo.

2. Actor Brief Descriptions

   El actor involucrado en el caso de uso es el desarrollador el cual quiere generar el código en el álgebra de tareas.

3. Preconditions

   3.1 El proyecto tiene que estar activo.

   3.2 Se debe tener al menos un diagrama.

   3.3 El diagrama tiene que ser válido.

4. Basic Flow of Events

Acción del actor	Respuesta del sistema
El caso de uso inicia cuando el diagrama termina de validarse y este resulta sin errores .	El sistema responde generando el álgebra del diagrama, en el caso de que hubiese algún error se ejecuta el flujo alterno 5.1. Se relaciona con: ▪ Desplegar álgebra ▪ Reportar errores.

5. Alternative Flows

   El sistema envia los errores a otra funcionalidad que los desplegará posteriormente.

6. Subflows

   Se inicia reportar errores.

7. Key Scenarios

   7.1 El código en el álgebra de tareas es generado.

   7.1 El código no se genera.

8. Post-conditions

   8.1 El código es generado con éxito.

   8.2 El código en el álgebra de tareas es almacenado en un archivo.

   8.3 Finalización sin éxito.

   8.4 No se generan archivos de sintaxis abstracta.

9. Special Requirements

   Ninguno.

Cuadro 3.9: Caso de uso generar álgebra presentado bajo el formato de OpenUp.

Este caso de uso se inicia, si el diagrama a traducir no presenta ningún tipo de error. De ser así, se aplican las reducciones mostradas en la figura 3.7, posteriormente se inicia el caso de uso mostrar álgebras.

3.5.4. Caso de uso mostrar álgebra

Plug-in
Use-Case: Mostrar álgebra

1. Brief Description

   El presente caso de uso muestra al usuario el álgebra literal correspondiente a un diagrama de flujo de tareas válido.

2. Actor Brief Descriptions

   El actor involucrado en el caso de uso es el desarrollador el cual pretende recibir al álgebra abstracta de un diagrama de flujo de tareas.

3. Preconditions

   3.1 El proyecto debe estar activo.

   3.2 Debe existir un diagrama ya modelado.

   3.3 El diagrama correspondiente debe ser válido.

4. Basic Flow of Events

Acción del actor	Respuesta del sistema
El caso de uso inicia cuando el usuario elige la opción de generar álgebra y no se detecta ningún error.	El sistema responde llamando a los mecanismos correspondientes al IDE para mostrar el álgebra almacenada previamente en un archivo. Se relaciona con el caso de uso: Generar álgebra.

5. Alternative Flows

   El sistema muestra el álgebra del diagrama actual contenida en un archivo.

6. Subflows

   Ninguno.

7. Key Scenarios

   Se despliega el álgebra correspondiente.

8. Post-conditions

   El álgebra de tareas se visualiza con éxito.

9. Special Requirements

   Ninguno.

Cuadro 3.10: Caso de uso desplegar álgebra presentado bajo el formato de OpenUp.

El caso de uso mostrar álgebra se inicia una vez que se cuenta con el álgebra de tareas correspondiente, el usuario observará cómo esta álgebra es mostrada al abrirse automáticamente un archivo ocupando el primer plano del área de trabajo.

3.5.5. Caso de uso exportar a XMI

Plug-in
Use-Case: exportar a XMI

1. Brief Description

   El caso de uso permite crear y guardar en un archivo con formato XMI la información de un diagrama de flujo de tareas.

2. Actor Brief Descriptions

   El actor involucrado en el caso de uso es el desarrollador, el cual pretende obtener un archivo XMI del modelo de flujo de tareas.

3. Preconditions

   3.1 El proyecto debe estar activo.

   3.2 Debe existir un diagrama ya modelado.

4. Basic Flow of Events

Acción del actor	Respuesta del sistema
El caso de uso inicia cuando el usuario elige la opción de exportar a XMI.	El sistema responde extrayendo información referente al diagrama de flujo de tareas actual y construyendo un diagrama de actividades UML, después pide información al usuario referente a la ruta de destino para el archivo recién creado

5. Alternative Flows

   El sistema muestra un mensaje de error referente al problema producido por este proceso.

6. Subflows

   Ninguno.

7. Key Scenarios

   Aparecen errores referentes a la exportación del diagrama.

8. Post-conditions

   El archivo XMI, se guarda con éxito en la ruta seleccionada por el usuario.

9. Special Requirements

   Ninguno.

Cuadro 3.11: Caso de uso exportar a XMI presentado bajo el formato de OpenUp.

Para este caso de uso, no es necesario que se cuente con un diagrama válido o ya traducido. Inicia con la petición de exportar al formato XMI, posteriormente el usuario seleccionará un nombre válido así como una ruta, en caso de que el nombre no sea correcto, el sistema asignará uno, el caso de uso finaliza cuando el archivo XMI se guarda en la ruta asignada.

3.6. Estructuración del modelo de casos de uso

Una vez detallados los casos de uso de la categoría manual e identificados los de la categoría automatizado, se crea el diagrama de casos de uso, el cual se muestra en la figura 3.8, en éste se observa cómo es que se definen las relaciones entre cada uno de ellos; cabe mencionar, que los casos de uso en la categoría manual, darán paso a las funcionalidades más importantes del plug-in.

Se optó por no desarrollar los casos de uso de la categoría automatizada, ya que no puede inferir en su proceso, sino más bien, el usuario el cual tenga experiencia en el desarrollo de software utilizando el IDE Eclipse, tendrá claramente como se llevan a cabo esas tareas.

3.7. Identificación inicial de las clases

Una vez que se crearon los diferentes casos de uso que conforman al sistema, se procedió a la identificación inicial de clases, para ello se hizo uso de la estrategia de frases nominales, que consiste en identificar sustantivos o pronombres en las descripciones textuales de la especificación del problema, los cuales pueden convertirse en clases o atributos.

Por ahora se pondrá atención a las clases que surgen de los casos de uso pertenecientes a la categoría "manual", ya que estas serán desarrolladas íntegramente y no se verán afectadas por el comportamiento de las diferentes herramientas involucradas en el proyecto. Además es de suma importancia aclarar que estas clases iniciales corresponden a la parte de control descrita por el patrón Modelo-Vista-Controlador. Hasta este punto y de acuerdo con lo descrito en la sección 3.3, la información del modelo visual sera extraída de archivos en formato XML mediante las clases adaptadoras GenerarAlgebra y ExportarXMI, además de una clase especial mencionada en la misma sección la cual llamaremos Editor, es decir por ahora es prescindible el editor gráfico para los diagramas de flujos de tareas. Más adelante en la sección 4, se detallará el proceso específico para ello.

A partir de la categoría "manual", se obtuvieron las siguientes frases nominales.

- Traducir.
- Exportar.
- Validar.
- Reportar.

A partir de esta información, y conocedores del tipo y la forma en que se obtendrá la información de entrada, podemos definir estas clases como la correspondientes la etapa de control del patrón Modelo-Vista-Controlador, por tal motivo, se infiere que las clases que diseñaremos para llevar a cabo la actividad esperada de esta fase, consiste en generar el álgebra de tareas correspondiente así como la creación de errores en caso de que existan.

Por tal razón, agregamos la frase nominal archivos, dicha frase hace referencia al manejo de archivos de texto, de igual forma tomando en consideración lo descrito en la sección 3.4, nuestra frase nominal Traducir se descompondrá en Reducir y Parsear, siendo esta última frase la que implica un proceso de validación, por lo tanto la frase nominal Validar deja de tomarse en consideración, de tal manera que ahora se tiene la siguiente lista de frases nominales.

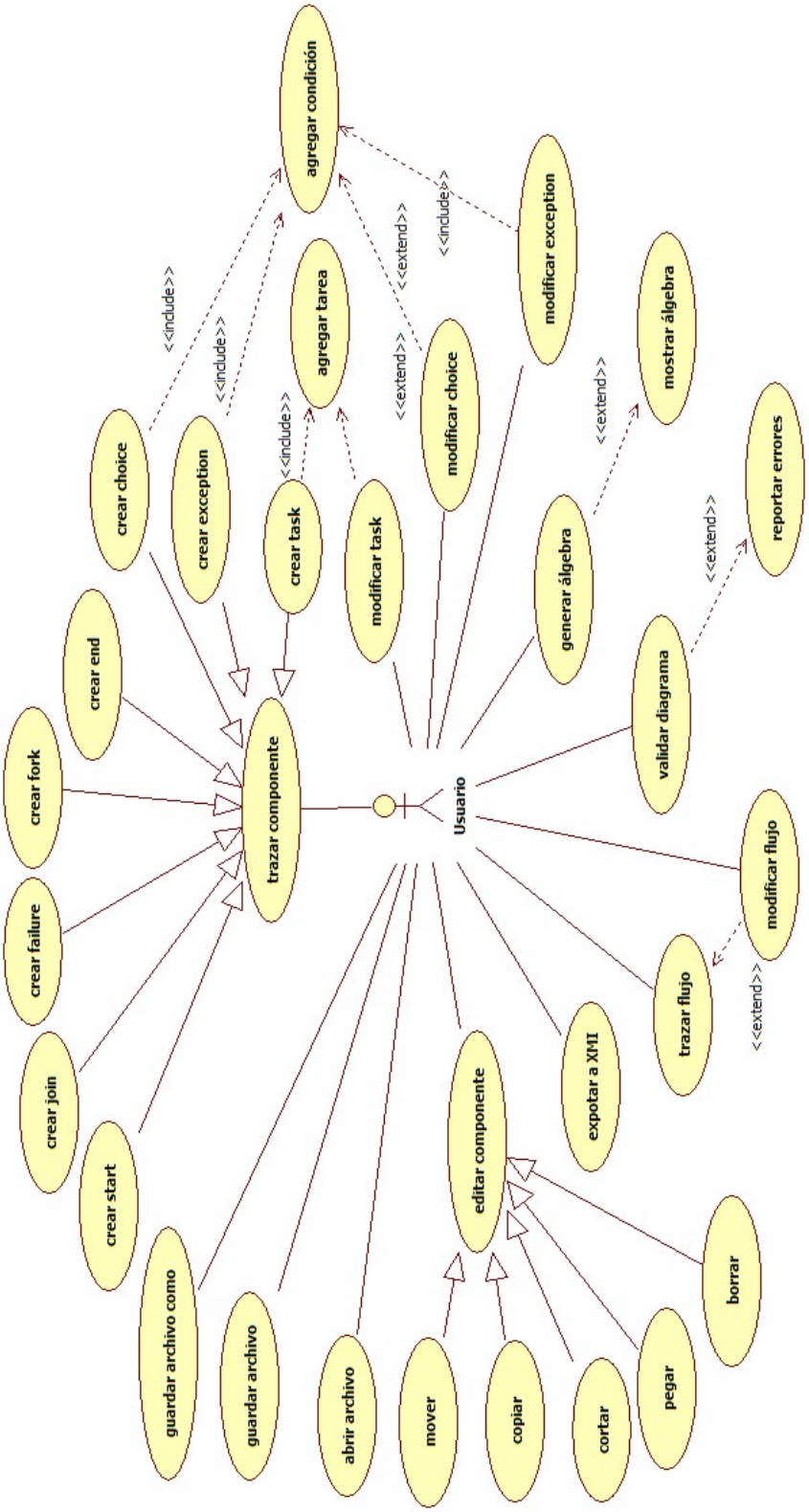


Figura 3.8: Diagrama de casos de uso del plug-in.

- Archivos.
- Parsear.
- Reducir
- Exportar.
- Reportar.

A continuación se muestra una lista con las primeras clases identificadas a partir de la técnica de frase nominales, así como también las clases adaptadoras y la clase especial. Nuevamente se modifican las frases nominales, ya que ahora se sabe de la necesidad de una clase específica para comunicarnos con el IDE, en este caso Editor. Podemos prescindir de la frase Reportar como error y para separar las operaciones entre la lectura de archivos y su manipulación en forma de cadena, agregamos la frase nominal Cadenas, en la figura3.9 se pueden apreciar dichas clases.

- GenerarAlgebra.
- ExportarXMI.
- Editor.
- Archivos.
- Cadenas.
- Parsear.
- Reducir.
- Exportar.

Más adelante se identifican características de cada una de las clases como son: estereotipos, atributos y comportamiento.

Entre los estereotipos, se definen cuatro los cuales utilizaremos para reconocer su comportamiento. Los tipos son:

***Boundary*:** Aquellas clases que se utilizan para modelar la interacción entre el sistema y sus actores.

***Control*:** Aquellas clases que generalmente representan coordinación, transiciones y control de objetos.

***Entity*:** Clases utilizadas para modelar información que sea duradera en el sistema, tal como el almacenamiento de información.

***Implementation*:** Clases que heredan de otras y por medio de este mecanismo nos permiten acceder a partes específicas del IDE.

3.8. Análisis de los atributos

Un atributo especifica la propiedad de una clase y contiene su propia información. Hasta este momento se han podido identificar ciertos atributos, sin embargo conforme se profundice en el uso de las herramientas necesarias para la construcción del plug-in, se agregarán los que sean necesarios.

Figura 3.9: Diagrama inicial de clases identificadas en los casos de uso.

A cada clase se le generarán atributos propios para realizar su función, por tal motivo el diagrama de la figura 3.9 será modificado.

3.8.1. Atributos para la clase GenerarAlgebra

Esta clase se define por si sola, ya que proviene de un punto de extensión, es decir responderá a un evento dado en el IDE. Se identifica un único atributo, el cual será weditor del tipo Editor, ya que es de quién se requiere extraer información. Se marca con el estereotipo **"*implementation*"** debido a que implementa la clase abstracta *IWorkbenchWindowActionDelegate*, de esta manera se obtiene la clase de la figura 3.10.

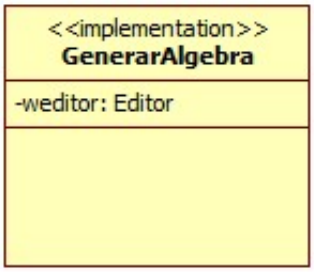


Figura 3.10: Clase GenerarAlgebra con un atributo, identificada con el estereotipo *implementation*.

3.8.2. Atributos para la clase ExportarXMI

Esta clase al igual que GenerarAlgebra se define por si sola, ya que proviene de un punto de extensión, es decir también responderá a un evento dado en el IDE. Se identifica un único atributo, el cual será *weditor*, ya que es de quién se requiere extraer información. Se marca con

el estereotipo **"*implementation*"** debido a que implementa la clase abstracta *IWorkbenchWindowActionDelegate*, de esta manera se obtiene la clase de la figura 3.11.

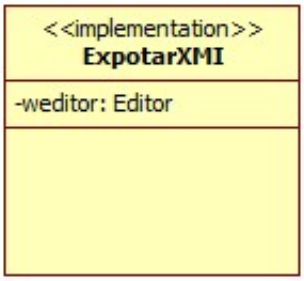


Figura 3.11: Clase ExportarXMI con un atributo identificada con el estereotipo *implementation*.

3.8.3. Atributos para la clase Editor

La clase Editor es una clase especial ya que a través de ella se accederán a recursos del IDE tales como espacio de trabajo, archivos e inclusive a información del sistema operativo (esto utilizando las capacidades propias del IDE), por tal motivo las demás clases cuyo origen sea un punto de extensión harán uso de esta misma. Se identifican los siguientes atributos *window* tipo *IWorkbenchWindow*, *workspace* tipo *IWorkspace*, editor tipo *IEditorPart*, *input* tipo *IFileEditorInput*, parse tipo Parsear, XMI tipo Exportar, *project* tipo *IProject*, monitor tipo *IProgressMonitor*, *status* tipo *Status*, *file* y algebra tipo *IFile*, lector tipo Archivos. Se asigna el estereotipo **"control"** de esta manera se obtiene la clase de la figura 3.12.

Figura 3.12: Clase Editor con atributos identificada con el estereotipo control.

3.8.4. Atributos para la clase Archivos

Esta clase servirá como manejador de archivos. No se identifican atributos necesarios, se le asigna el estereotipo **"*entity*"** de esta manera se obtiene la clase de la figura 3.13.

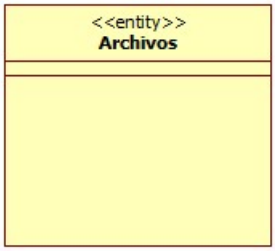


Figura 3.13: Clase Archivos sin atributos identificada con el estereotipo *entity*.

3.8.5. Atributos para la clase Cadenas

La clase cadenas estará enfocada a manejar los datos obtenidos mediante la lectura de los archivos abstractos tfe y tfd realizados por la clase archivos, por tal motivo no se identifican atributos propios, se marca con el estereotipo **"*entity*"** de esta manera se obtiene la clase de la figura 3.14.

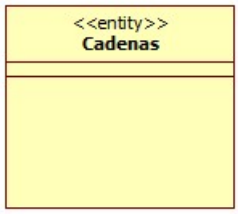


Figura 3.14: Clase Cadenas sin atributos identificada con el estereotipo *entity*.

3.8.6. Atributos para la clase Parsear

Esta clase cumplirá la función de validar el diagrama, por tal motivo está ligada a la clase reducir. Es fundamental dar a conocer que, debido al mismo análisis, se tendrá que utilizar una cantidad considerable de estructuras dinámicas capaces de almacenar datos temporalmente, dichas estructuras se muestran a continuación: componentes, errores, pila, pilaForks, pilaJoins, arrayForks, arrayJoins, arrayJoinsDestino, arrayExceptions y arrayChoices todas del tipo ArrayList<String>; y flujos del tipo ArrayList<Integer>. Así mismo, dicha clase se marca con el estereotipo **"control"**, la figura 3.15 muestra la clase modificada.

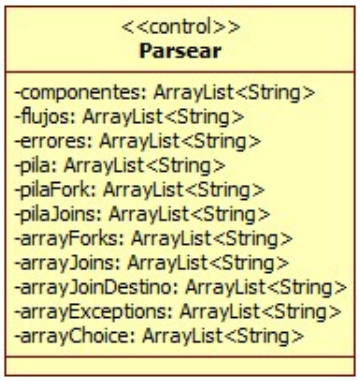


Figura 3.15: Clase Parsear con los atributos identificados, marcada con el estereotipo control.

3.8.7. Atributos para la clase Reducir

Para iniciar con la generación del álgebra, es necesario tener la información abstracta del diagrama en turno, por tal motivo entre los atributos tenemos las dos estructuras dinámicas, identificadas como flujos y componentes del tipo ArrayList<*Integer*> y ArrayList<*String*> respectivamente. Es necesario el campo algebraDelDiagrama de tipo *String*, dado que se requiere un campo para almacenar los errores de semántica para su posterior despliegue dentro de la lista de errores del tipo ArrayList<*String*>.

Para pasar de las estructuras dinámicas de datos utilizadas para el proceso de generación del álgebra a la información real con los datos residentes en el diagrama de flujo de tareas, se agregan los siguientes atributos: frAuxiliar de tipo ArrayList<*String*>, además de las constantes *SIGMA, EPSILON, PHI* y *MU* todas de tipo *String*, las cuales contendrán el valor *unicode* de la letra griega que representen. Por último, el atributo continuarReduciendo de tipo *boolean*. Esta clase se marca con el estereotipo **"control**" en la figura 3.16 se presenta el estado actual de la clase.

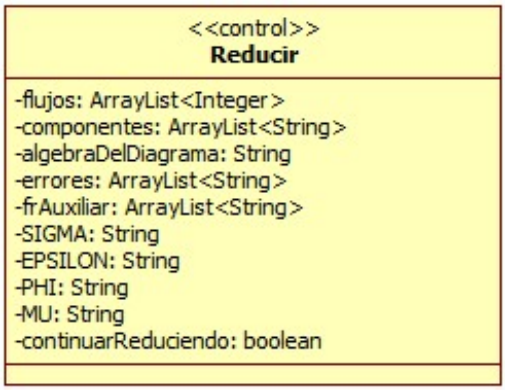


Figura 3.16: Clase Reducir con los atributos identificados marcada con el estereotipo control.

3.8.8. Atributos para la clase Exportar

Debido a que esta clase es independiente del proceso llevado a cabo para generar el álgebra de tareas respectiva, se requieren tres campos los cuales corresponden a la información de las estructuras dinámicas, siendo estos campos, contenidoTFD, contenidoTFE y contenidoXMI todos de tipo ArrayList<*String*>, además de fc de tipo *JFileChooser*. Finalmente se marca con el estereotipo **"*entity*"**, con la finalidad de preservar información y obtenerla de los archivos en formato XML. La imagen de la figura 3.17 muestra esta clase.

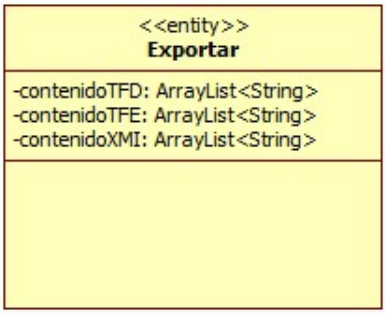


Figura 3.17: Clase Exportar con los atributos identificados marcada con el estereotipo *entity*.

3.9. Análisis del comportamiento de las clases

Una responsabilidad es un orden que se le solicita a un objeto para que realice alguna acción sobre uno o varios de sus atributos, la cual puede ser: ejecutar rutina o proporcionar el conocimiento. Al analizar las clases se debe identificar sus responsabilidades y atributos. Hasta este punto se tienen contempladas las clases con sus atributos pero sin su debido comportamiento.

Las responsabilidades de las clases se documentan dentro de la clase como parte de su descripción, en este caso se trató de darle un nombre acorde a la acción que se requiere cumplir. El diagrama de la figura 3.9 se modificó para agregar dicho comportamiento. A continuación se describen las clases y sus respectivos comportamientos agregando los métodos necesarios para hacerlo funcional y acorde a nuestras necesidades.

3.9.1. Comportamiento para la clase GenerarAlgebra

Esta clase desarrolla el comportamiento derivado de un punto de extensión el cual responde a la acción de un clic sobre el botón "generar álgebra". Es necesario que implemente 4 métodos de la clase abstracta *IWorkbenchWindowActionDelegate*, sin embargo sólo un método es utilizado para el desarrollo del plug-in.

init: Inicializa sobre este punto de extensión el proceso asignado a esta clase.

dispose: Se encarga de notificar al recolector de basura que su proceso ha terminado. Parámetro de entrada *window* de tipo *IWorkbenchWindow*.

selectionChanged: Método mediante el cual se detectan cambios y se responden de acuerdo a la configuración hecha en el archivo plug-in.XML. Parámetros de entrada *action* y *selection* de tipos *IAction* y *ISelection* respectivamente.

run: Sobre este método es que se realizan las operaciones necesarias para generar el álgebra de tareas, es decir instanciar y coordinar a las clases necesarias para llevar a cabo este proceso.

La clase modificada se muestra en la figura 3.18.

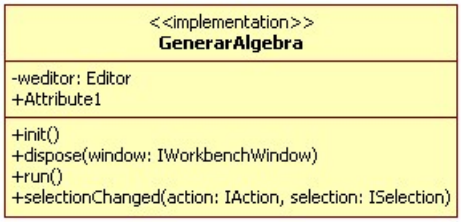


Figura 3.18: Clase GenerarAlgebra con sus atributos y métodos identificados.

3.9.2. Comportamiento para la clase ExpotarXMI

Esta clase desarrolla el comportamiento derivado en un punto de extensión el cual responde a la acción de un clic sobre el botón "exportar a XMI". Es necesario implementar 4 métodos de la clase abstracta *IWorkbenchWindowActionDelegate*, sin embargo sólo un método se utiliza para el desarrollo del plug-in.

init: Inicializa sobre este punto de extensión el proceso asignado a esta clase.

dispose: Se encarga de notificar al recolector de basura que su proceso ha terminado. Parámetro de entrada *window* de tipo *IWorkbenchWindow*.

selectionChanged: Método mediante el cual se detectan cambios y se responden de acuerdo a la configuración hecha en el archivo plug-in.XML. Parámetros de entrada *action* y *selection* de tipos *IAction* y *ISelection* respectivamente.

run: Sobre este método es que se realizan las operaciones necesarias para llevar el diagrama de flujo de tareas a un formato XMI. Es decir instanciar y coordinar a las clases necesarias para llevar a cabo este proceso.

La clase modificada se muestra en la figura 3.19.

3.9.3. Comportamiento para la clase Editor

Lo especial de esta clase, es que servirá a otras clases para tener acceso a recursos del IDE, por tal motivo se requiere de ciertos métodos capaces de obtener referencias e información necesaria a través del uso de clases contenidas en los paquetes org.eclipse.core y org.eclipse.ui. A continuación se muestran los métodos que componen esta clase:

crearTFA: Este método creará el correspondiente archivo TFA, el cual contendrá el álgebra de tareas del diagrama de flujos de tareas analizado. Para ello requerirá de dos parámetros de

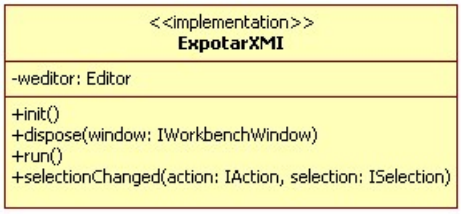


Figura 3.19: Clase GenerarAlgebra con sus atributos y métodos identificados.

entrada álgebra de tipo *InputStream* y nombreArchivo de tipo *String*. Devolverá un tipo *boolean* dependiendo del resultado del proceso de la creación y apertura del archivo.

verificarArchivo: Dados los archivos TFE y TFD se comenzará con el proceso de parseo y posterior reducción para obtener el álgebra de tareas. Este método no requiere de parámetros de entrada ni de salida, sin embargo instanciará diversos objetos para llevar a cabo el proceso encomendado ya que a partir de la existencia de los archivos que componen a un diagrama de flujos de tareas se iniciará el mecanismo para la obtención de dicha álgebra o en su defecto los errores encontrados.

getFile: Método que no requiere de parámetros de entrada pero que devuelve un tipo *IFile* con el archivo que se encuentre en primer plano en un determinado espacio de trabajo.

StringToStream: Método que, a partir de un *String* devuelve un *InputStream*.

IsExtTrue: Este método requiere como parámetro de entrada un *String*, el cual contendrá una extensión para comprobar que el archivo corresponda con esa extensión. Devuelve un *boolean* dependiendo del resultado de verificar el archivo contra el *String* que contiene la extensión.

Enlazar: Método que a partir de un *String* (el cual contiene una extensión), devuelve un *String* del contenido del archivo dado por la extensión.

getNameFile: En un *String* devuelve el nombre del archivo residente en el espacio de trabajo indicado por los parámetros de entrada inp y ext de tipo IFile y String respectivamente.

DefineNameFromFile: Obtiene a partir de los parámetros de entrada *IFile* y *String* el archivo y extensión respectiva, devuelve en un *String* el nombre de un archivo con la extensión dada.

cleanIR: Este método en particular se utiliza para marcar información a ser eliminada del área de errores, no requiere de parámetros de entrada ni de salida pero lleva a cabo un proceso especial al instanciar objetos de esa área en específico.

deleteAuditMarkers: Método encargado de borrar la información de errores relativa a un proyecto específico, para ello requiere de un parámetro de tipo IProject que haga referencia al proyecto actual. Devuelve un valor *boolean* dependiendo del resultado del proceso.

Guardar: Guarda los cambios realizados a los archivos activos pertenecientes a un proyecto. No requiere de parámetros de entrada ni de salida.

reportarProblemas: A partir de los parámetros msg de tipo *String*, archivo de tipo *IFile*, viola-

ción de tipo int y esError de tipo *String* creará mensajes de error o advertencia dependiendo del valor del parámetro esError.

La clase modificada se muestra en la figura 3.20.

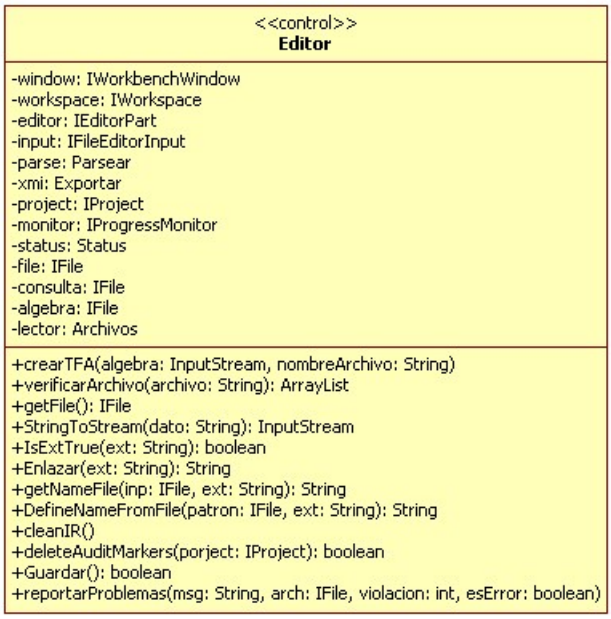


Figura 3.20: Clase Editor con sus atributos y métodos identificados.

3.9.4. Comportamiento para la clase Archivos

Para la clase Archivos no se tiene identificado ningún atributo; su estereotipo *entity* indica la interacción directa con archivos XML de los diagramas del plug-in, de esta manera se identifican dos métodos necesarios para su funcionamiento: leer y guardar. Para leer se requiere de una variable de entrada tipo *IFile,* la cual deberá contener la información referente a los datos a ser leídos en el disco, este método devolverá un *String*. Por otro lado, para el método guardar se necesita una variable de salida tipo *IFile,* la cual contiene la referencia y los datos del archivo a guardar, y una variable que contiene la información de tipo *String*. Finalmente, la variable monitor de tipo *IProgressMonitor* para saber el estado en que se encuentra la operación, no es necesario devolver ningún valor. La clase modificada se muestra en la figura 3.21.

3.9.5. Comportamiento para la clase Cadenas

El comportamiento de la clase Cadenas es simple, pero sumamente importante, ya que el objetivo principal es pasar de un *String* a un *InputStream* y viceversa, por tal motivo, esta clase implementa dos métodos.

stringToStream: Este método recibe un parámetro de tipo *String* para posteriormente devolver un *InputStream* .

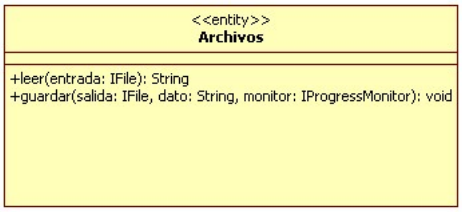


Figura 3.21: Clase Archivos con los atributos y métodos identificados.

StreamToString: A partir de un parámetro de tipo *InputStream* se procesa y se devuelve un *String*.

La clase modificada se muestra en la figura 3.21.

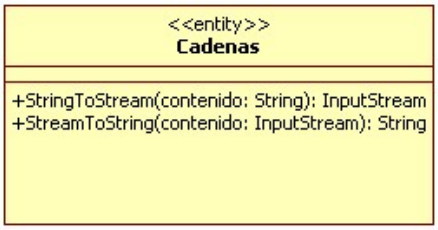


Figura 3.22: Clase Cadenas con los atributos y métodos identificados.

3.9.6. Comportamiento para la clase Parsear

El comportamiento de la clase Parsear tiene un grado de complejidad mayor que las clases anteriores, debido a ello sólo se muestran los métodos principales, los métodos restantes se pueden consultar en el anexo C.

procesamientoCorrecto: Método que devuelve un valor lógico verdadero si no se detectaron errores.

getAlgebra: Método que devuelve el valor del álgebra de tareas del diagrama correspondiente.

getComponentes: Devuelve la lista abstracta componentes.

getAnchoFlujos: Devuelve el valor de tipo entero correspondiente al ancho de la estructura abstracta flujos.

getAnchoComponentes: Devuelve el valor de tipo entero correspondiente al ancho de la estructura abstracta componentes.

getFlujos: Devuelve la lista abstracta flujos.

procesaTfd: Analiza el archivo correspondiente a la representación visual del diagrama de flujos de tareas.

procesaTfe: Analiza el archivo correspondiente a la representación lógica del diagrama de flujos de tareas.

valConexion: Valida que el número de conexiones (tanto de entradas como de salidas de un determinado componente sea el correcto). Recibe como parámetros de entrada el id del componente, así como el número de entradas y salidas permitidas. Este método utiliza la tabla de estados 3.12 para validar los flujos que se le presenten.

	Start	*Task*	*Fork*	*Join*	*Exception*	*Failure*	*Choice*	*End*
Start	0	1	1	0	1	0	1	0
Task	0	1	1	1	1	1	1	1
Fork	0	1	1	0	1	0	1	0
Join	0	1	1	1	1	1	1	1
Exception	0	1	1	1	0	1	1	1
Failure	0	0	0	0	0	0	0	0
Choice	0	1	1	1	1	1	1	1
End	0	0	0	0	0	0	0	0

Cuadro 3.12: Tabla de estados utilizada por el método valConexion para la verificación de flujos.

Los métodos públicos de esta clase son procesamientoCorrecto, getAlgebra, getComponentes, getFlujos, getAnchoComponentes, getAnchoFlujos y ParsearTareas. Una vista parcial de la clase modificada se muestra en la figura 3.21, la vista completa de la clase se puede consultar en el apéndice B.

3.9.7. Comportamiento para la clase Reducir

El comportamiento de la clase Reducir resulta fundamental para el proceso de traducción de los diagramas de flujos de tareas a su respectiva álgebra de tareas, ya que, como se mencionó en la sección 3.4 se buscará formar expresiones completas del álgebra de tareas hasta llegar (si la semántica es correcta) a un flujo básico de inicio y un componente terminal. Para ello se describen los métodos principales, los cuales cumplirán diversas funciones para cada una de las formas estructurales del álgebra de tareas. Los métodos restantes se presentan en el anexo C.

inicializar: Método encargado de tener una copia de las estructuras abstractas correspondientes al diagrama de flujo de tareas para su posterior utilización dentro de esta misma, no devuelve ningún tipo de valor y recibe las estructuras a copiar.

algebraCorrecta: Devuelve un valor lógico verdadero si la estructura final corresponde a un inicio y un componente terminal.

generarAlgebra: Método que llama a todos los demás métodos encargados de buscar la posible formación de estructuras del álgebra de tareas.

getAlgebra: Devuelve el álgebra de tareas correspondiente a un diagrama de flujo de tareas.

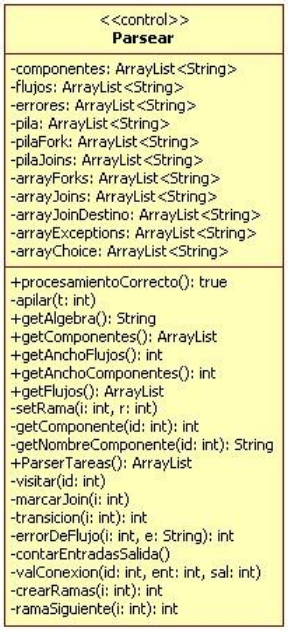


Figura 3.23: Clase Parsear con los atributos y métodos identificados.

getErrores: Método que devuelve los errores correspondientes al diagrama analizado.

reducirTareasLineales: Dadas dos tareas lineales las combina para obtener una tarea compuesta.

crearFORK: Previa identificación de tareas paralelas, este método las agrupa en un sólo componente para crear una secuencia lineal.

crearUNTIL: A partir de dos valores de tipo entero representando índices, modifica la estructura abstracta para crear un *Until* y eliminar las tareas que lo componen.

reducirOr: Método que busca sobre los componentes *Choice* la posible formación de alguna estructura *Or*.

reducirFor: Método que evalúa cada uno de los componentes *Choice* para verificar si alguno puede crear un *For*.

Los únicos métodos públicos son inicializar, algebraCorrecta, generarAlgebra, getAlgebra y getErrores. Una vista parcial de la clase modificada se muestra en la figura 3.21, la clase completa se puede consultar en el apéndice B.

3.9.8. Comportamiento para la clase Expotar

Para el comportamiento de la clase Exportar se identifican los siguientes métodos necesarios para su funcionamiento:

cargarXMI: Con los parámetros archivoTDF y archivoTFE ambos de tipo *String* sin devolver ningún valor. En este método se extraerá toda la información necesaria para llevar a cabo el proceso de exportación de datos contenido en los archivos TFE y TFD.

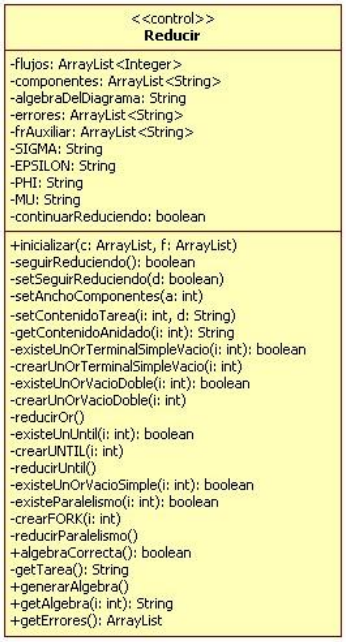


Figura 3.24: Clase Reducir con los atributos y métodos identificados.

guardar: Encargado de crear un archivo en formato XMI para almacenar los datos del diagrama en este formato; requiere de un parámetro que contenga los datos de creación del archivo.

procesaTFD: A partir del parámetro entrada de tipo *String*, se extraerá la información referente al archivo TFD y lo almacenará en la estructura dinámica contenidoTFD. Este parámetro no devolverá ningún valor.

procesaTFE: A partir del parámetro entrada de tipo *String*, se extraerá la información referente al archivo TFE y lo almacenará en la estructura dinámica contenidoTFE. Este parámetro no devolverá ningún valor.

crearXMI: Este método no tiene parámetros de entrada ni de salida, sin embargo es el núcleo de esta clase, ya que a partir de la información previamente obtenida se aplica el algoritmo de exportación para generar un nuevo archivo compatible con el estándar XMI.

crearNombreArchivo: A partir de información previamente almacenada y con el parámetro nombre de tipo *String* el cual representa la ruta absoluta del archivo a crear, este método devuelve en un *String* un nombre válido para el archivo destino que almacenará información en el estándar XMI, cabe destacar que si el nombre está mal formado, devolverá el valor “diagrama.XMI”.

crearRutaArchivo: Este método extrae y devuelve únicamente la ruta del archivo a crear, requiere de dos parámetros: ruta, que contiene la ruta absoluta del nuevo archivo y nombre, ambos de tipo String.

obtenerRutaNombre: La finalidad de este método es obtener la información por parte del usuario del nombre y ruta del archivo nuevo donde se almacenará la información en formato

XMI, devuelve en un *String* la ruta y el nombre ya validados. No posee parámetros de entrada.

La clase modificada se muestra en la figura 3.25.

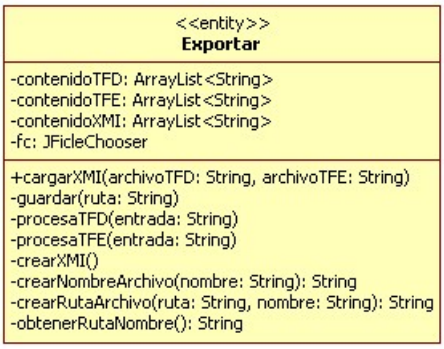


Figura 3.25: Clase Exportar con sus atributos y métodos identificados.

3.10. Identificación de asociaciones y agregaciones

Se analizó la arquitectura de los plug-ins de Eclipse mostrada en la sección 3.3.4, así como también el comportamiento de las clases definidas en la sección 3.9, las cuales corresponden a los puntos de extensión y la clase especial utilizada para la comunicación con el propio IDE.

Además para identificar una asociación se tomó en cuenta lo siguiente [19]:

- Un objeto A es parte física o lógica de un objeto B.
- Un objeto A está física o lógicamente contenido en un objeto B .
- Un objeto A está registrado en un objeto B.

Las agregaciones son un caso particular de asociación y debe utilizarse cuando los objetos representan.

- Conceptos que contienen físicamente a otros.
- Conceptos que están compuestos uno de otro.
- Conceptos que forman una colección conceptual de objetos.

La agregación maneja otra variedad conocida como composición, en ésta, el componente puede pertenecer a un todo y se espera que las partes vivan y mueran con el todo. La figura 3.26 muestra el diagrama de clases final, en ella se pueden apreciar las asociaciones, agregaciones y composiciones identificadas.

3.11. Diseño general del sistema

La arquitectura del plug-in mostrada en la figura 3.27 responde a una representación general basada en el modelo vista controlador, más adelante en la sección 4 se detallará el proceso llevado

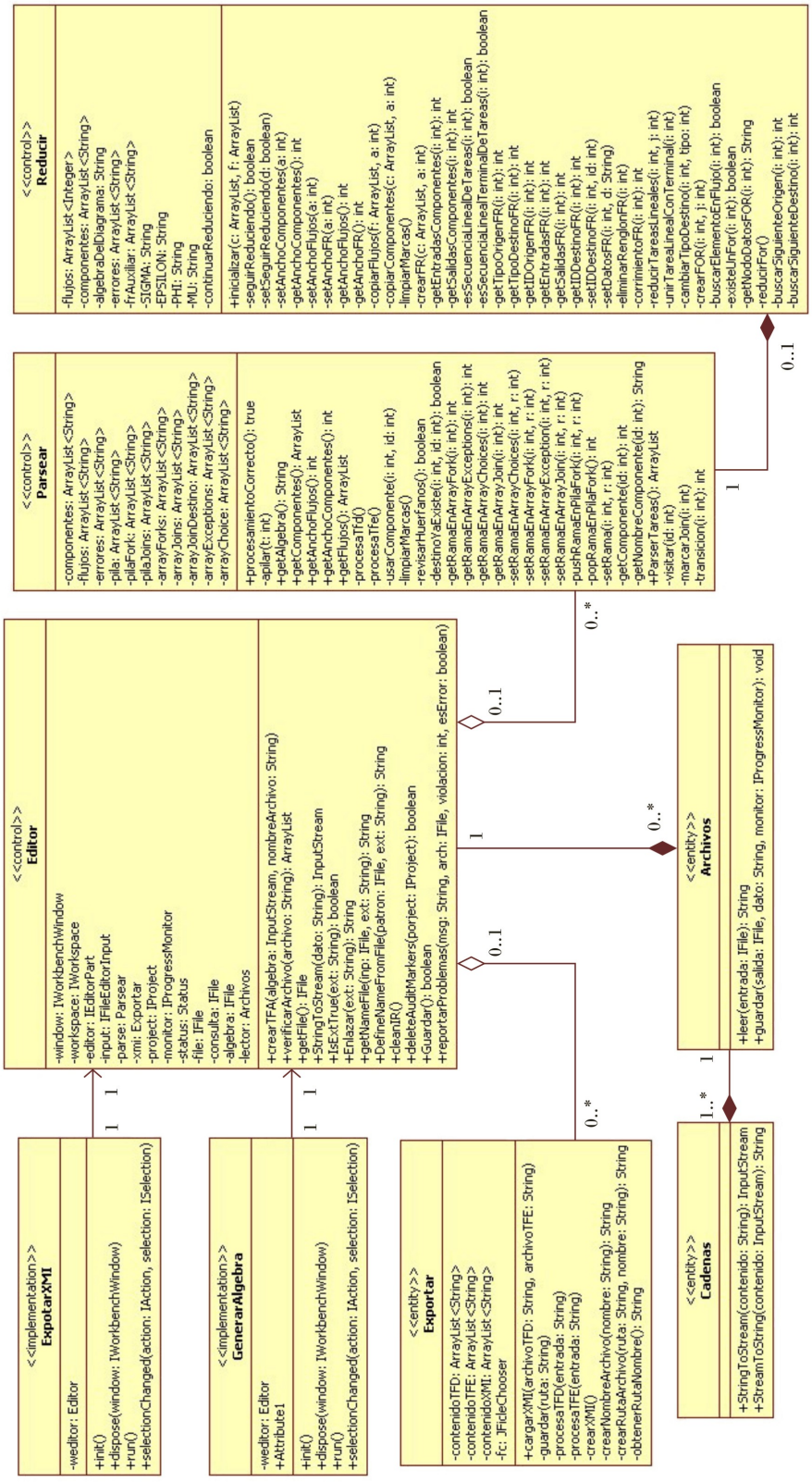


Figura 3.26: Diagrama de clases final para el desarrollo del controlador del plug-in

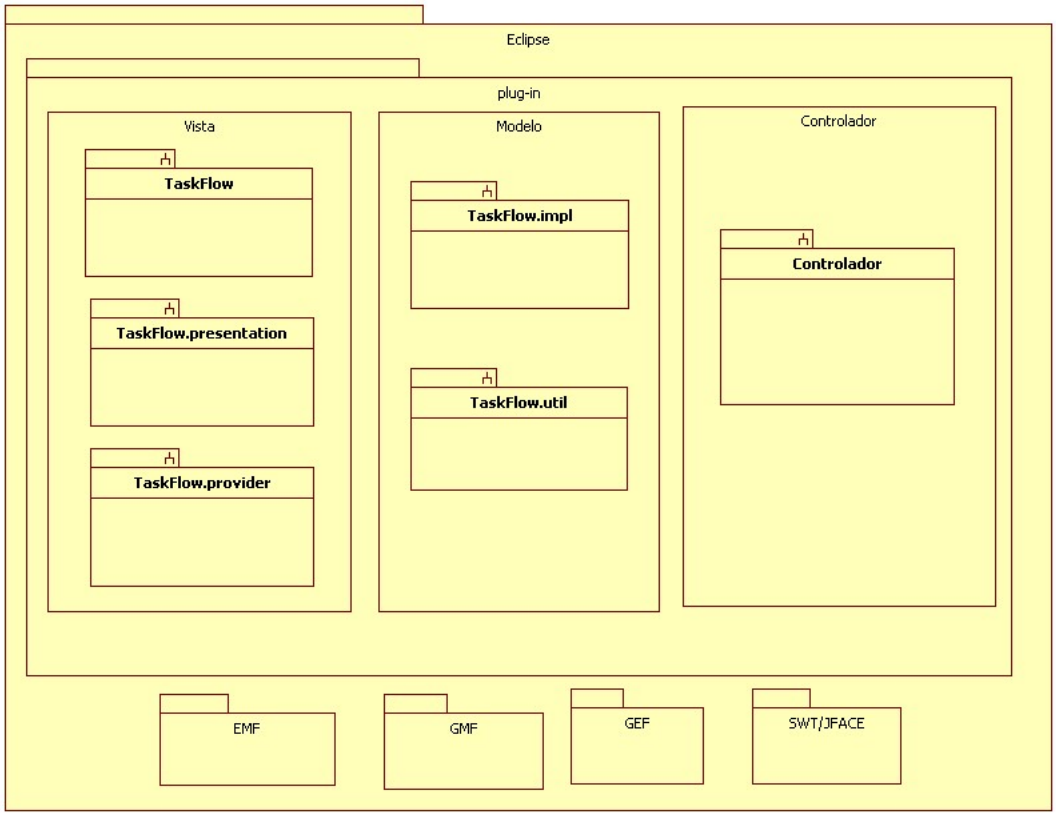


Figura 3.27: Arquitectura incial del plug-in basada en el modelo vista controlador.

a cabo para la implementación del modelo, por ahora se muestran las funcionalidades de cada uno de los tres subsistemas en que se divide el plug-in.

Cada uno de estos subsistemas realiza su función de manera independiente, comunicándose por medio de información almacenada en formato XML, específicamente los archivos correspondientes a los diagramas; más los subsistemas mínimos necesarios para la arquitectura tradicional de un plug-in.

3.11.1. Vista

3.11.1.1. Subsistema Taskflow

El susbsistema *Taskflow* se encarga del manejo lógico de las figuras que compondrán los diferentes diagramas, dando soporte en el área de trabajo de Eclipse para la edición de los diagramas. Es decir, funcionará como contenedor para los demás elementos visuales del diagrama. A continuación se detalla su funcionamiento.

1. Crear dentro de Eclipse un área de trabajo destino para la visualización de los diagramas.
2. Integrar dentro del espacio de trabajo directorios con la finalidad de almacenar permanentemente los diagramas.

3. Brindar soporte para las funcionalidades nativas del IDE, tales como: editar, guardar y guardar como.

3.11.1.2. Subsistema TaskFlow.presentation

El subsistema *Taskflow.presentation* se encarga de llevar a cabo la implementación visual de los componentes del diagrama de flujo de tareas. Es decir, la creación visual de los diagramas en un área de trabajo destinada a contener elementos gráficos. Resumiendo lo anterior tenemos:

1. Creación de un área de trabajo la cual tiene la capacidad de contener componentes visuales así como las funcionalidades necesarias para ello tales como: eliminar, arrastrar, mover, cambiar de tamaño, etc.
2. Creación de una paleta de componentes, la cual contendrá la representación visual de los componentes admitidos por los diagramas de flujo de tareas, como lo son, *Start, Task, Fork, Join, Exception, Failure, Choice, End* y *flujos*.

3.11.1.3. Subsistema TaskFlow.provider

El subsistema *Taskflow.provider* cumple la función de interpretar el modelo lógico en el cual se basa la representación visual del plug-in, más adelante en la sección 4 se detalla esta función específica del modelo vista controlador. De manera general, cumple con las siguientes funciones:

1. Interpretación de los datos contenidos en el modelo lógico para los diagramas de flujos de tareas.
2. Contiene los módulos necesarios para que los subsistemas *TaskFlow* y *TaskFlow.presentation* puedan realizar su función.

3.11.2. Modelo.

3.11.2.1. Subsistema TaskFlow.impl

El subsistema genera (a partir de un diagrama de flujo de tareas) archivos en formato XML, de esta manera se tiene la persistencia del modelo visual con datos capaces de ser interpretados de manera abstracta. A continuación se detallan sus funciones:

1. Creación de información referente a un diagrama de flujos de tareas de cada uno de los elementos que integran dicho diagrama.
2. Creación de archivos en el espacio de trabajo para el almacenamiento de la información creada en un diagrama de flujo de tareas.
3. Almacenamiento de la información en formato XML de los archivos creados dentro del espacio de trabajo.

Cabe resaltar que el proceso de abstracción de la información de un diagrama a un archivo XML se realiza de forma inmediata, no así los cambios realizados de forma manual.

3.11.2.2. Subsistema TaskFlow.util

El subsistema provee las funcionalidades correspondientes a la validación de un archivo XML, ya que no se tiene la posibilidad de crear diagramas a partir de archivos XML. Existen ciertos parámetros que construye esta clase, de esta forma se asegura que cualquier diagrama creado con el plug-in, no tenga problemas para su posible edición desde el entorno visual. En resumen, tenemos que:

1. Genera ciertos parámetros únicos necesarios para construir un archivo XML válido utilizado por el subsistema *TaskFlow*.impl.
2. Asegura la correcta estructura de los archivos XML.

3.11.3. Controlador

3.11.3.1. Subsistema Controlador

El subsistema controlador, es el encargado de llevar a cabo, entre otras funciones, la generación del álgebra correspondiente. Para ello, debe tener acceso a los archivos XML del diagrama. Debemos recordar que los archivos XML, corresponden a la representación abstracta la cual nos brindará la información necesaria para llevar a cabo este proceso.

Es también en este subsistema donde se tiene la funcionalidad de exportar a un formato XMI, para utilizar los diagramas en otras herramientas que implementen la importación de este tipo de archivos.

Si bien estos archivos una vez modificados no pueden ser importados por el plug-in, si se otorga la posibilidad de un acercamiento a la portabilidad de los diagramas. Las funcionalidades que implementa son las siguientes:

1. Obtención de datos abstractos de los diagramas.
2. Validación de los diagramas.
3. Información de errores.
4. Generación del álgebra de tareas.
5. Exportación a formato XMI.

.

En el siguiente Capítulo se profundizará sobre los detalles de la implementación del proyecto, tales como configuración y personalización del editor visual, así como el desarrollo de las clases diseñadas previamente.

Capítulo 4

Implementación del plug-in

En este capítulo se describe el proceso llevado a cabo para finalizar la configuración del prototipo diseñado y parcialmente implementado en la sección 3.3, así mismo, se abordará con detalle la lógica que da origen a la implementación de las clases pertenecientes al control expuestas en el capítulo anterior.

4.1. Proceso de desarrollo mediante GMF

Mediante EMF se diseñó el modelo estructurado de la figura 3.4, posteriormente se utilizó para crear el editor visual mostrado en la figura 3.1.

El proceso de desarrollo del editor de diagramas de flujos de tareas se realiza con GMF mediante la definición de diversos modelos los cuales serán descritos más adelante, como punto de inicio se debe tener un modelo estructurado diseñado y con soporte del *framework* EMF.

Los modelos de GMF están descritos por el metamodelo Ecore. Es a partir de éste que se construyen los diversos modelos, mediante un conjunto de herramientas y asistentes para su edición. A continuación se describen brevemente cada uno de ellos:

- Modelo de dominio (.ecore): También conocido como modelo semántico. Es el punto de partida para la construcción del editor de diagramas, ya que es un metamodelo Ecore que describe la capa del modelado de la aplicación y actúa como metamodelo del modelo que el usuario podrá editar gráficamente.
- Modelo de definición gráfica (.gmfgraph): Contiene la información relativa a los elementos gráficos que aparecerán en el editor. En él se definen los elementos a representar, en este caso, los diferentes componentes con los que se podrán diseñar los diagramas de flujos de tareas.
- Modelo de herramientas (.gmftool): Define la barra de herramientas del editor gráfico, en ella se tienen los componentes de los diagramas que le permitirá al usuario añadir elementos, es decir agrupa los componentes mostrados en la figura 3.3 más el componente flujo.
- Modelo de mapping (.gmfmap): Relaciona el modelo de dominio con el modelo gráfico y el modelo de herramientas. Es el modelo más importante del desarrollo del proyecto, ya

que en él convergen las definiciones de los demás modelos y da lugar al modelo generador de código de GMF. Además de almacenar las relaciones entre los elementos del modelo y los elementos de la vista, permite definir una serie de atributos para personalizar el editor de diagramas a generar.

- Modelo generador (.gmfgen): Una vez que se tienen definidos los modelos anteriores, GMF construye un modelo generador a partir del modelo de mapeo. Esto es, un modelo que extiende al modelo de *mapping* para especificar los parámetros de la generación automática de código.

La relación establecida entre los diferentes modelos de GMF se puede observar en la figura 4.1.

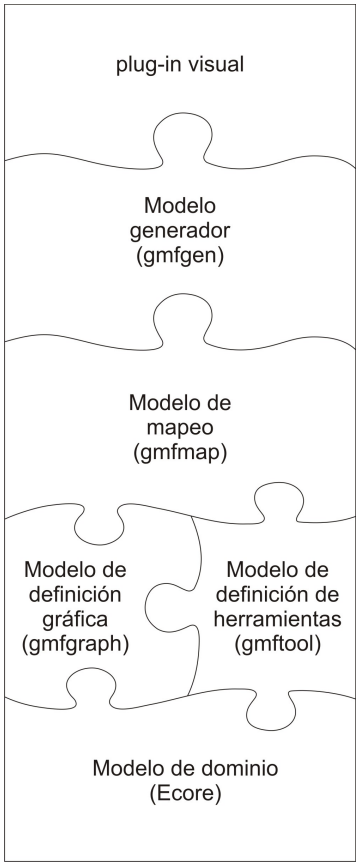


Figura 4.1: Relación de los modelos GMF para la construcción del plug-in

Además de estos modelos básicos hay que resaltar que un metamodelo Ecore no ofrece todo el soporte que exige el *framework* GEF. Específicamente aspectos tales como la posición y el tamaño de los elementos en el editor, además de otras características las cuales quedan fuera del alcance de la definición semántica.

Para suplir esta carencia GMF introduce un modelo para describir las características de los diagramas, llamado modelo de notación. Esta información se usa para modelar y representar los elementos visuales en los diagramas.

Este modelo de notación es el que genera el archivo Ecore el cual almacena el modelo semántico, creando a partir de este último los diferentes modelos necesarios para visualizar la

representación gráfica de los diagramas de flujos de tareas.

El desarrollo mediante el *framework* GMF requiere del uso de diferentes herramientas ofrecidas por el mismo *framework*, esto para la construcción de una serie de modelos que describen las características del editor de diagramas.

Los pasos para el desarrollo del editor de diagramas y en general para cualquier plug-in basado en GMF son:

1. Crear el modelo de dominio mediante un metamodelo Ecore, descrito en la sección 3.3.3.
2. Crear un modelo de definición gráfica, el cual define los aspectos visuales del editor de diagramas generado.
3. Crear el modelo de las herramientas, comprende una serie de elementos relacionados con barras de herramientas, menús y otros componentes que servirán para llevar a cabo la interacción con el usuario.
4. Crear un modelo de mapeo, que es la unión del modelo ya definido con la ayuda de EMF (figura 3.4), el cual representa en un archivo .ecore. La finalidad de este modelo es visualizar las formas y las relaciones que tendrá el aspecto visual y las herramientas disponibles..

La creación de los modelos del dominio (ecore), de definición gráfica y de herramientas se pueden construir en forma paralela. Una vez que se generan estos tres modelos es posible crear la definición del mapeo.

A partir de la definición del mapping se crea la definición del generador del editor. Por último, se construye el plug-in con el editor gráfico, tomando como base la información del generador del modelo. La secuencia descrita se muestra en la figura 4.2 . Básicamente el desarrollo de una aplicación mediante el *framework* GMF no se lleva a cabo de forma estrictamente secuencial, sino mas bien queda abierta y sujeta a una gran cantidad de retroalimentaciones y modificaciones posibles entre los mismos modelos.

4.2. Modelo de definición gráfica

El primer paso fue crear un modelo gráfico a partir del modelo Ecore que se generó con EMF. Para ello se seleccionó la opción: *File -> New -> Other -> Simple Graphical Definition Model*.

Esta acción inicia un asistente, donde se debe seleccionar aquellos elementos del metamodelo Ecore que tendrán una representación gráfica para los diagramas de flujos de tareas, es de resaltar que previamente se debe seleccionar como elemento raíz a *ShapeDiagram*. Para el proyecto del plug-in que se está desarrollando no se requiere mostrar la totalidad de los elementos que se han definido, por lo que únicamente se seleccionaron aquellos elementos que se enlistan en el cuadro 3.3 más el componente flujo.

Por ejemplo, no es necesario que la clase "*ShapeDiagram*", así como la clase "*Shape*" tengan representación visual dentro del editor de diagramas de flujos de tareas, dada la relación de herencia que tienen hacia los demás componentes del modelo estructurado, es decir, sólo los elementos que hereden de la clase *Shape* contarán con representación en el editor gráfico.

Dada la simplicidad del modelo estructurado y dado que es necesario marcar los atributos que tendrán representación visual en el editor, se marcarán todos los atributos de las siguientes clase:

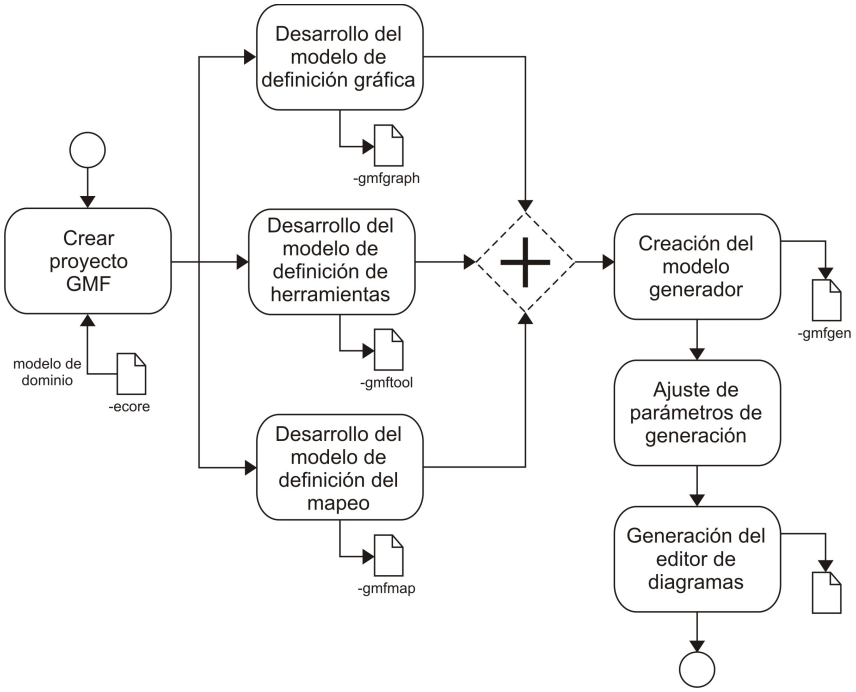


Figura 4.2: Proceso de trabajo llevado a cabo por GMF [3].

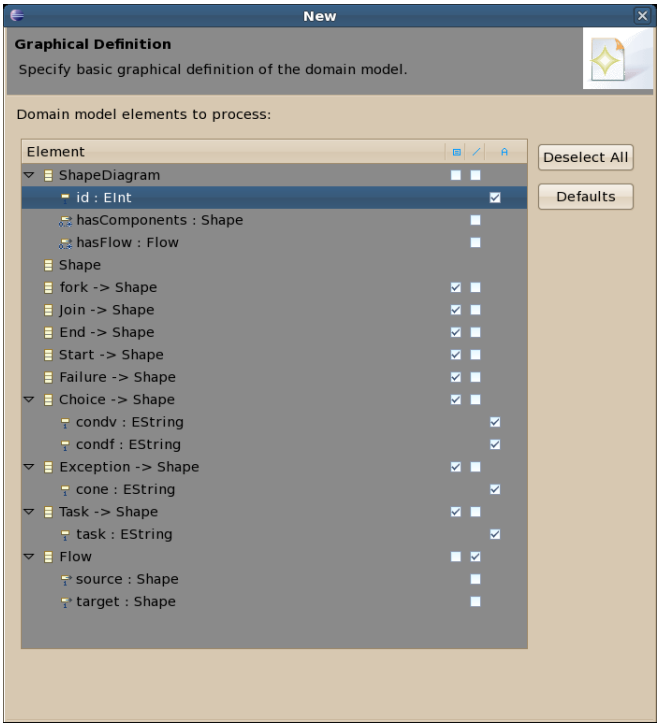


Figura 4.3: Asistente Modelo de Definición Gráfica

- *ShapeDiagram*: id.

- *Choice*: condicionVerdadera y condicionfalsa.
- *Exception*: condicion.
- *Task: task.*

El modelo gráfico generado tendrá la extensión .gmfgraph. Es necesario aplicar cambios a este modelo gráfico, para obtener el comportamiento adecuado y esperado de tal manera que la representación gráfica sea acorde a las necesidades del proyecto, así, la representación visual de los diagramas de flujos de tareas será la correcta.

Entre los cambios a las diferentes propiedades que se realizarán, se agregarán elementos visuales, ya sea circulares, elípticos (*Ellipse*) y rectangulares (*Rectangle*), además de ciertas propiedades de estos objetos como colores de fondo (*Background*) y de contorno (*Foreground*), así como se modifica de la relación de tamaño, distribuidos en tres categorías específicas: tamaño esperado (*preferredSize*), tamaño máximo (*MaximunSize*), tamaño mínimo (*MinimunSize*) y el tamaño predeterminado (*Size*), estas y otras características se modifican para cada uno de los componentes visuales.

4.2.1. Componente Task

Una vez que terminada esta fase del modelo, la representación gráfica para todas las figuras a desplegar en el plug-in corresponden a rectángulos, por ello es imprescindible la modificación de esta característica.

Como primer componente a configurar se tiene al elemento *Task*, su representación gráfica resulta simple, sin embargo requiere de una configuración especial ya que este elemento contiene información de suma importancia para la generación del álgebra de tareas, es decir, la etiqueta asociada a este componente es en realidad una tarea que será enviada al compilador del Método *Discovery*. La configuración por defecto se muestra en la figura 4.4. A continuación se enlistan los cambios realizados a la configuración del componente:

- *Ellipse: Task*
- *Foreground*: Negro.
- *Background*: Blanco.
- *PreferredSize*: 70 por 30.
- *Label: TaskFigure.*
- *XYLayoutData*: posición 0,0 largo y alto 12 por 12.

Para la configuración adicional de la etiqueta del componente *Task*, se requiere agregar un descriptor hijo de tipo ***ChildAccess*,** llamado *getFigureTaskFigure*; de igual forma la configuración *NODE* en su propiedad *Resize Constraint* se establece a "*none*". Para la misma configuración de la etiqueta *Label Task*, se asigna el valor *Center*. La figura 4.5 muestra la configuración correcta.

4.2.2. Componente Choice

La representación gráfica del componente *Choice* posee las características de una figura irregular, por tanto su configuración se basa en un polígono. La figura 4.6 muestra la configuración por defecto para este componente, los cambios llevados a cabo se muestran a continuación:

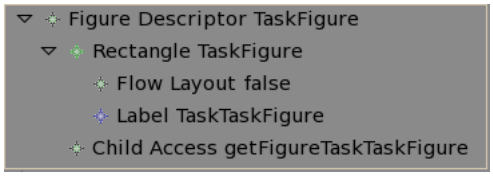


Figura 4.4: Estructura del componente Task antes de su configuración.

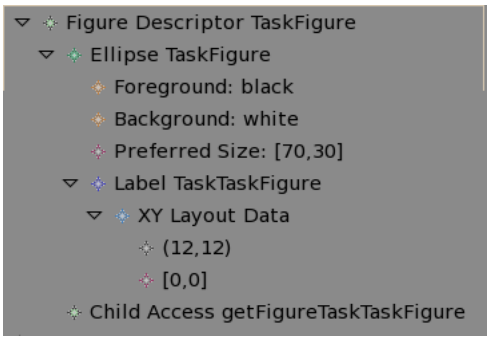


Figura 4.5: Estructura del componente Task después de su configuración.

- *Scalable: Polygon.*
- *FlowLayout*: Falso.
- *Foreground*: Negro.
- *Background*: Blanco.
- Puntos: Pares ordenados para definir el polígono 10,0; 20,10; 10,20; 0,10; 10,0.

El comportamiento dado por la configuración *NODE* no debe permitir cambio en su tamaño, por tal motivo la propiedad *Resize Constraint* se establece a "*none*".

Para las etiquetas que contendrán la información dada por el modelo estructurado y nombradas ChoiceConv yChoiceConF respectivamente, se les asigna en su configuración *Label* el valor *WEST* para la propiedad *Resize Constraint.*

La configuración final del componente se muestra en la figura 4.7.

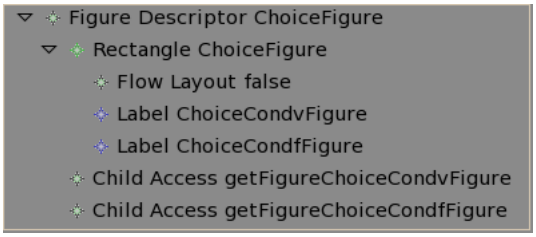


Figura 4.6: Estructura del componente Choice antes de su configuración.

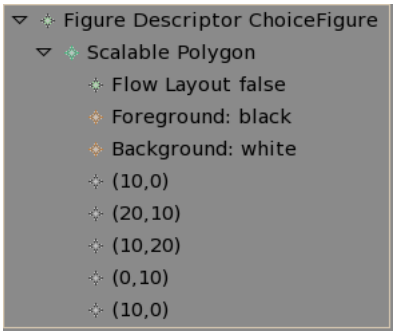


Figura 4.7: Estructura del componente Choice después de su configuración.

4.2.3. Componente Flow

El componente *Flow* es el más simple de configurar, ya que únicamente requiere añadir el color y el acabado final para que obtenga la apariencia de una flecha. La configuración por defecto se muestra en la figura 4.8. Los cambios realizados se muestran a continuación:

- *PolylineConection: FlowFigure*.
- *Foreground*: Negro.

Se añade un subcomponente el cual representará el final de la conexión:

- *PolylineDecoration*: Flecha.
- *Background*: Negro.

Dado que el componente *Flow* no es un nodo sino líneas con punta triangular que semejan a una flecha, este carece de la configuración *NODE*.

La figura 4.9 muestra la configuración final de elemento *Flow*.

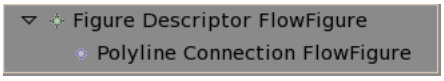


Figura 4.8: Estructura del componente Flow antes de su configuración.

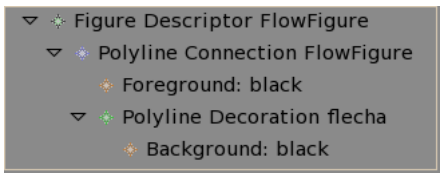


Figura 4.9: Estructura del componente Flow después de su configuración.

La configuración que define gráficamente a los componentes restantes se puede consultar en el Anexo D.

4.3. Modelo de definición de herramientas

Una vez que se ha definido el modelo gráfico, y según el mapa de trabajo descrito en la figura 4.2, el siguiente paso es definir el modelo de herramientas.

Para la construcción de este modelo, básicamente se indicará qué elementos se deben tener disponibles en la barra de herramientas del plug-in, los elementos que ahí aparezcan serán los componentes con los cuales se diseñarán los diagramas de flujos de tareas. Se necesita que aparezcan los elementos *Start, Task, Fork, Join, Exception, Failure, Choice, End* y *Flow*, quedando descartados los componentes *Shape* y *ShapeDiagram*.

Para su creación se seleccionó la opción *File->New->Other->Simple Tooling Definition Model*. De forma similar a la creación del modelo de definición gráfica, se inicia el asistente mostrado en la imagen de la figura 4.10, es en este asistente donde se deben seleccionar los componentes antes listados.

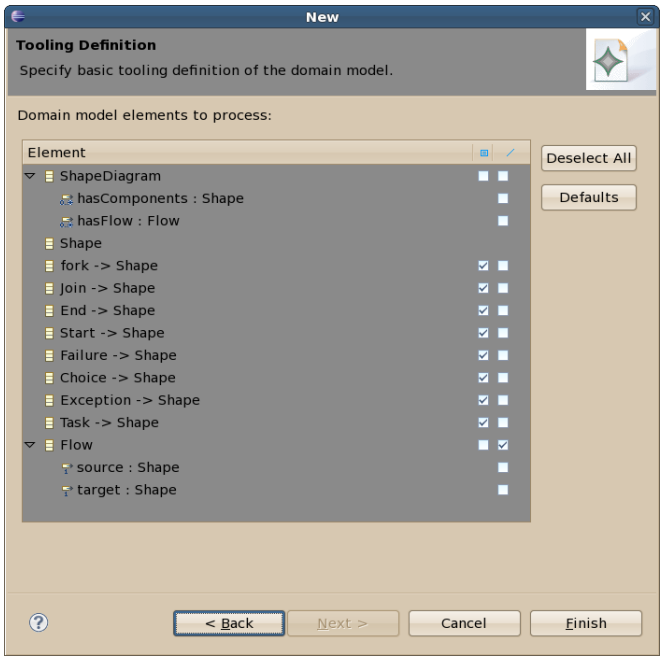


Figura 4.10: Asistente para la creación del Modelo de definición de herramientas.

El modelo generado tendrá la extensión ".gmftool" y contendrá a los elementos que hayan sido seleccionados. La figura 4.11 muestra el aspecto final del modelo de definición de herramientas.

4.4. Modelo de definición de mapeo

El siguiente paso, según el mapa de trabajo con GMF, es generar el modelo de mapeo. Este modelo tiene la función de enlazar el metamodelo Ecore mostrado en la figura 3.4, el modelo de definición gráfico y el modelo de definición de herramientas.

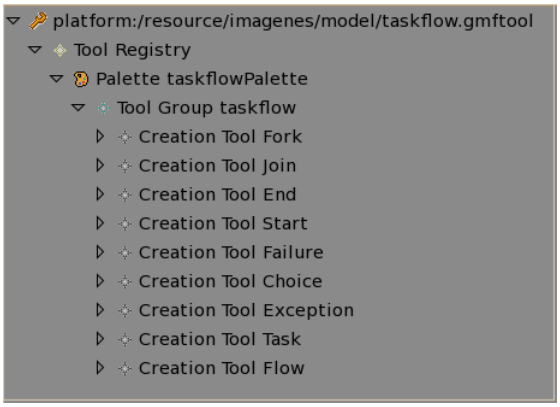


Figura 4.11: Aspecto final del Modelo de definición de herramientas.

Para su generación se selecciona la opción *File->New->Other->Guide Mapping Model Creation*.

El primer paso para generar el modelo de definición de mapeo, es seleccionar las rutas de los distintos modelos que este asistente requiere. Cabe resaltar que para el modelo Ecore, se debe también seleccionar el componente raíz, en este caso *ShapeDiagram*.

Una vez finalizado el proceso, se seleccionan que elementos del metamodelo serán mostrados como componentes visuales o cuales cumplirán las funcionalidad de enlazar. Ésta configuración se observa en la figura 4.12. Este mismo asistente brinda el respectivo direccionamiento al componente *Flow*, lográndose mediante la modificación de su configuración primaria, las propiedades *Source Feature y Target Feature* se les asigna respectivamente los valores *EReference source* y *EReference target*.

Una vez finalizado el asistente, se tiene la definición de mapeo mostrada en la figura 4.13.

En el modelo de mapeo se tuvieron que realizar diversas configuraciones personalizadas para las diferentes etiquetas a mostrar, así como para el componente *Flow*. Es importante destacar que la única restricción OCL que se logró compatibilizar con el motor OCL de Eclipse se aplica al componente *Flow*, dicha restricción tiene la finalidad de evitar diseñar diagramas con elementos que apunten así mismos, ya que este tipo de flujos no tienen una representación en el álgebra de tareas.

Las modificaciones realizadas a los Nodos del modelo de mapeo son las siguientes:

-Choice*:** Se agregó una nueva etiqueta de tipo ***Feature Label Maping, dado que el modelo posee por defecto una sola etiqueta. Sin embargo, el componente *Choice* consta de dos etiquetas marcadas como condv y condf para sus valores verdadero y falso respectivamente (figura 3.4). Una vez que el componente *Choice* cuenta con dos etiquetas, ambas tienen que asociarse a un componente de la clase, en este caso la propiedad ***Features to display*** y ***Features to edit*** se asocian a ***Choice.condv:Estring***, la propiedad ***Diagram Label*** se asocia con ***Diagram Label ChoiceCondv***. El mismo procedimiento se lleva a cabo para la etiqueta ***Choice.condf:Estring***.

-Exception*:** De la misma forma que las etiquetas para el componente *Choice*, la etiqueta correspondiente al componente *Exception* se debe asignar al atributo ***cone, tal como se aprecia en el modelo estructurado de la figura 3.4. De esta forma las propiedades del modelo de

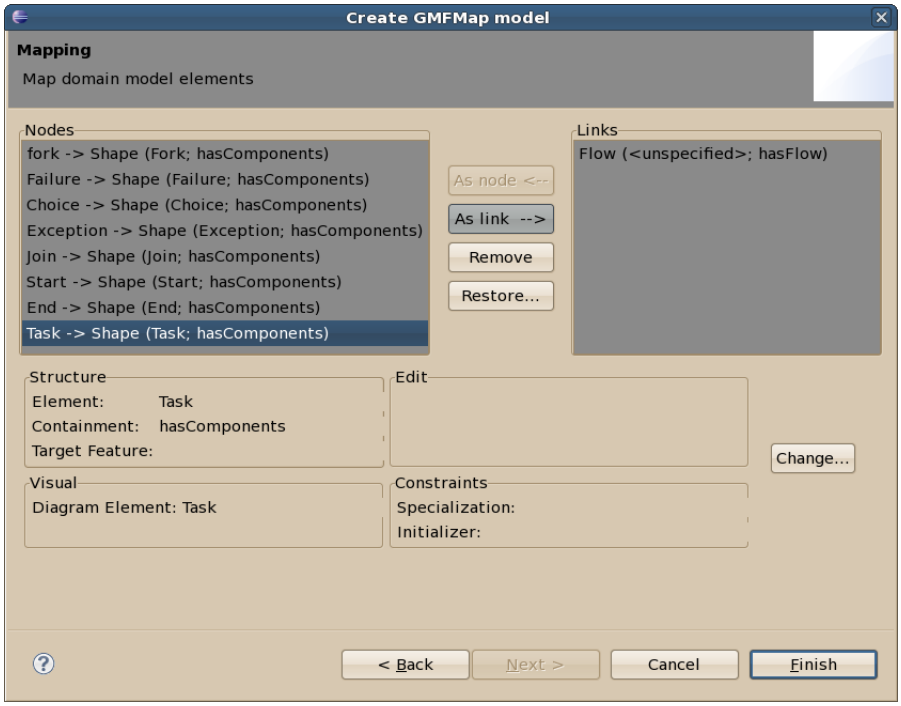


Figura 4.12: Selección de enlaces y clases en el Modelo de Mapeo.

mapeo **"*Features to display*"** y "***Features to edit***" se asocian a ***Exception.cone:Estring***, la propiedad **"*Diagram Label*"** se asocia con ***Diagram Label ExceptionCone***.

-*Task*: Al igual que el componente Choice y Exception la etiqueta task se debe asignar a un atributo de la clase Task, para ello las propiedades del modelo de mapeo **"*Features to display*"** y "***Features to edit***" se asocian a ***Task.task:Estring***, la propiedad **"*Diagram Label*"** se asocia con ***Diagram Label TaskTask***.

Después de configurar los Nodos para asociar las etiquetas visuales a los atributos de sus respectivas clases, se debe configurar la clase que representará a los enlaces, es decir, la clase encargada de implementar los flujos en los diagramas de flujos de tareas. Para ello se realizaron modificaciones sobre diversas propiedades del elemento ***Link Mapping.***

- *SourceFeature*: Se asignó el atributo *Flow.source.Shape* de la clase *Flow* del modelo estructurado.
- *TargetFeature*: Se asignó el atributo *Flow.target.Shape* de la clase *Flow* del modelo estructurado.
- *DiagramLink*: Se asignó el atributo *Connection Flow*.

Así mismo es necesario agregar un componente del tipo **"*Link Constraints*"** y en este último un componente del tipo **"*Constraint*"**, de esta manera se logra construir sobre este nuevo elemento la restricción OCL. Dicha regla sirve para evitar que los componentes puedan apuntarse a sí mismos. La regla OCL a aplicar es la siguiente:

- ***self <> oppositeEnd***

La figura 4.14 muestra la configuración final del modelo de definición de mapeo.

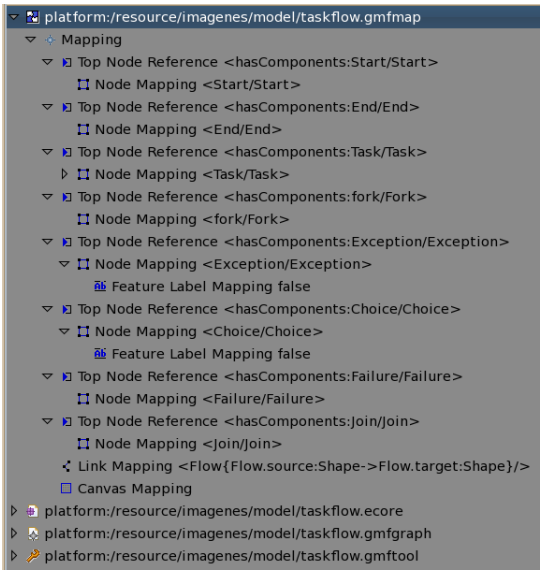


Figura 4.13: Modelo de Mapeo antes de la configuración para el proyecto.

4.5. Modelo generador

El último modelo a construir es el Modelo generador, para ello, basta con pulsar el botón derecho del cursor sobre el modelo de mapeo recién creado y seleccionar la opción "*Create generator model*". Como resultado se obtiene un archivo con la extensión **"gmfgen"**.

La figura 4.15 muestra el modelo generador.

Una vez definidos todos los modelos necesarios, para generar el plug-in simplemente se pulsa el botón derecho del cursor sobre el archivo **".gmfgen"** recién creado y se escoge la opción **"*Generate diagram code*"**. Esta acción genera cuatro proyectos: **.diagram**, **.edit**, **.editor** y un **.tests**, además dentro del espacio de trabajo se crean los siguientes paquetes:

- taskflow
- taskflow.impl
- taskflow.presentation
- taskflow.provider
- taskflow.util

Para agregar las clases que realizarán la lógica de la aplicación, es decir la parte de control del patrón Modelo-Vista-Controlador, se agrega el paquete controlador. En dicho paquete se agregarán las clases que dependerán de los puntos de extensión así como las demás clases necesarias para tener un plug-in funcional.

4.6. Puntos de extensión

Los puntos de extensión se crearon mediante la modificación del archivo plugin.xml, esto en el apartado **"*extensions*"**, para ello fue necesario abrir ese archivo con el editor de Eclipse y

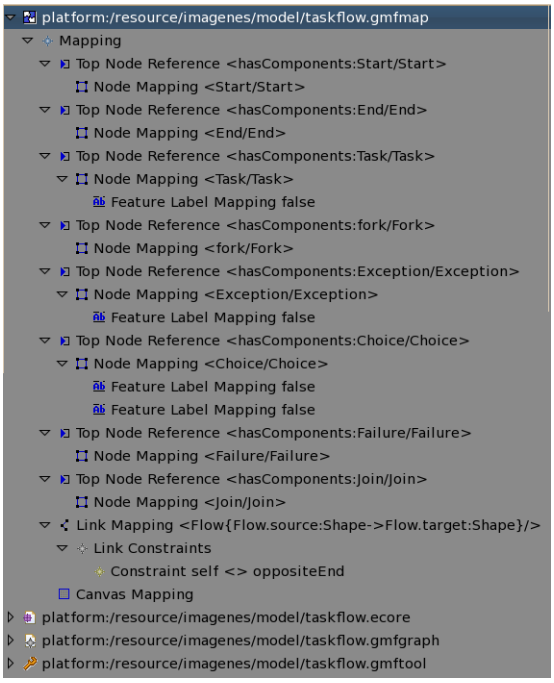


Figura 4.14: Modelo de Mapeo final configurado para el proyecto.

dirigirse al apartado extensions.

El siguiente paso en el proceso fue dar clic en el botón ***Add***, y seleccionar el tipo de punto de extensión, para el proyecto se indicó el tipo **"*org.eclipse.ui.actionSets*"**; esta acción crea un punto de extensión el cual únicamente tendrá asignada una etiqueta.

A este punto de extensión se le agregó un elemento de tipo ***actions***, de esta forma se obtuvo un nuevo elemento llamado **"Menús y botones"** del tipo actionSet; sobre este mismo se creó otro elemento, en esta ocasión del tipo **"*Menu*"**, la etiqueta de este menú fue nombrada *Discovery Method*.

Finalmente, se crearon dos nuevos componentes de tipo **"*action*"**, asignándoles los nombres Generar álgebra y Exportar a XMI; es sobre estos dos últimos elementos que se realiza la asociación de los puntos de extensión a sus respectivas clases; para ello, se da clic sobre el elemento Generar álgebra en específico sobre el apartado ***class***, se inicia un asistente, en el cual se introducirá el nombre de dicha clase, así como el paquete que la contendrá. Este proceso se repite para el elemento Exportar a XMI.

La figura 4.16 muestra una vista parcial del editor del archivo plugin.xml con el elemento generar álgebra configurado.

Es hasta esta fase del proyecto que el uso de los *framework* GMF y EMF termina, el siguiente paso es crear la lógica de la aplicación, es decir la parte del controlador, para ello es necesario implementar las clases que llevarán a cabo la traducción para generar el álgebra de tareas y el proceso de exportación al formato XMI.

Es importante resaltar que el desarrollo del plug-in manifestó una serie de problemas, ello debido a que al llevar el proyecto a una versión nueva (3.5 Galileo a la versión 3.6 Helios) los diferentes modelos marcados por GMF debían regenerarse, ya que los métodos no tenían un

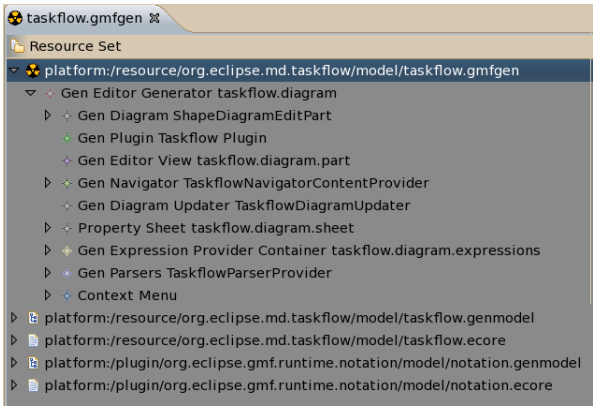


Figura 4.15: Vista parcial del Modelo generador para el proyecto.

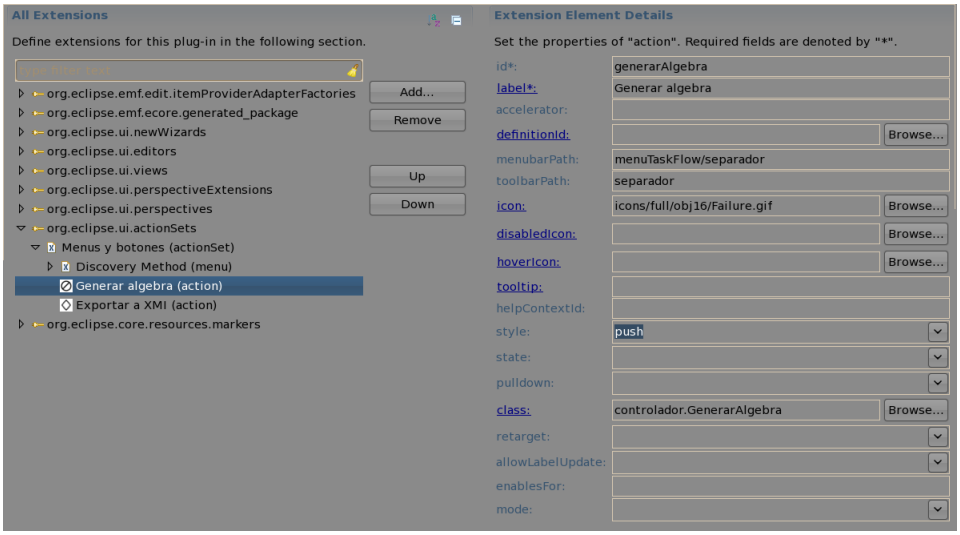


Figura 4.16: Vista parcial de la configuración del archivo plugin.xml.

correcto funcionamiento.

4.7. Implementación de las clases Controlador

La implementación para esta parte del proyecto hace referencia al flujo de trabajo donde se traduce el resultado de la fase de diseño de clases en código fuente, el cual será el encargado de abstraer la información de los diagramas de flujos de tareas a una serie de estructuras dinámicas temporales, mismas que serán procesadas para obtener el álgebra de tareas de un diagrama específico.

Basado en la arquitectura del diseño se logra la implementación final del plug-in por medio de código fuente desarrollado en lenguaje Java, una vez finalizada esta fase, se tendrá un sistema ejecutable y funcional.

Como parte final del capítulo de implementación, se describirá la lógica que implementan las clases que conforman al plug-in.

4.7.1. Implementación de la clase Parsear

Con la generación del código fuente se logra obtener una estructura estática de la herramienta, la cual incluye tanto atributos como comportamientos. Para iniciar con la descripción de la implementación de esta clase, se tiene que especificar que posterior a su proceso hace un llamado al atributo de tipo Reducir.

El paso siguiente para que la clase sea funcional, es la implementación del cuerpo de las funciones. La implementación de los diversos métodos se basó en el análisis de la estructura de los diagramas de flujos de tareas, siendo los fundamentales procesaTFD, procesaTFE, transicion y parsearTareas, las demás operaciones de la clase llevan a cabo funciones secundarias, siendo coordinadas por los métodos mencionados anteriormente.

A lo largo de esta sección se describe el funcionamiento lógico de los métodos principales.

4.7.1.1. Método procesaTFD

El proceso a llevar a cabo por este método, es abstraer la información visual de un diagrama de flujos de tareas a una estructura dinámica temporal. Esta acción se inicia al recibir el contenido del archivo tfd en un *string*.

La información que nos interesa obtener de los archivos tfd, es la referente a los elementos que tienen asociada una etiqueta, es decir, se debe obtener la información contenida en las etiquetas de los elementos *Choice, Exception* y principalmente *Task*. La imagen de la figura 4.17 muestra el diagrama de flujo utilizado para la implementación del método.

4.7.1.2. Método procesaTFE

Este método implementa una acción similar a procesaTFD. Toma como entrada el *String* que contiene los datos del archivo tfe, en este archivo se especifican los componentes y los flujos que forman el diagrama. Se asigna la información respectiva a las estructuras dinámicas “flujos” y “componentes”. Cabe resaltar que la información obtenida por procesaTFD se guarda de manera definitiva en las estructuras dinámicas, de esta manera, los componentes ya identificados se

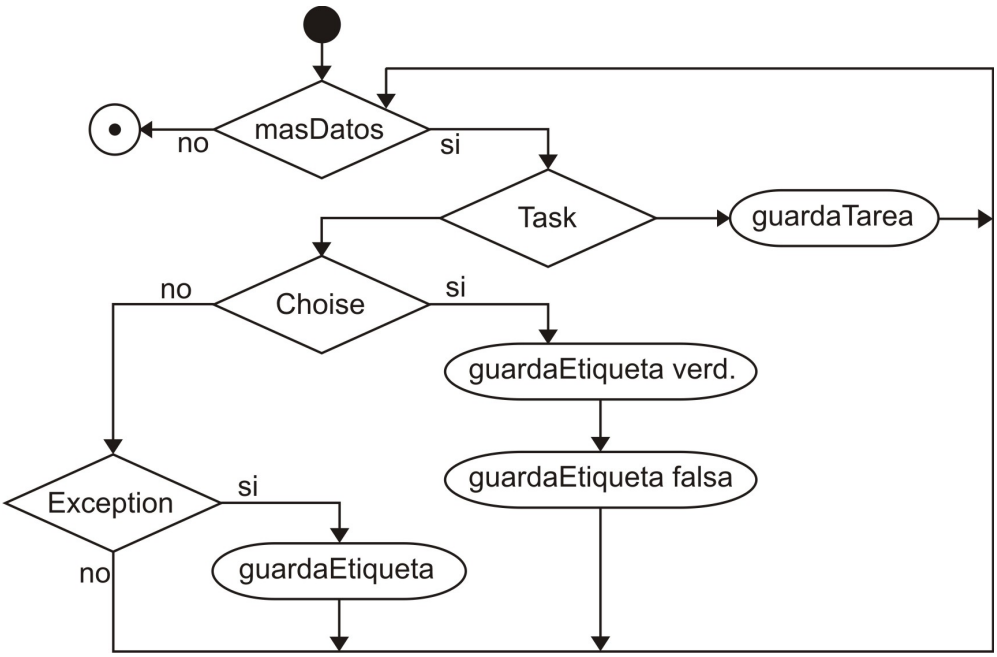


Figura 4.17: Diagrama de flujo del método procesaTFD.

asocian con la información dada por el usuario. En la figura 4.18 se observa el diagrama de flujo respecto al método procesaTFE.

4.7.1.3. Método transición

El método transición es el encargado de coordinar métodos secundarios para validar las conexiones del diagrama. Como primer acción determina la existencia de un único componente *Start*, y a partir de este se inicia el proceso de validación. Toma un renglón dentro de la estructura dinámica **flujos** y realiza un análisis sobre el componente origen, entre las acciones llevadas a cabo en el análisis tenemos:

- Validación de flujos, es decir que un componente origen apunte sólo a componentes permitidos establecidos en el cuadro 3.12.
- Asignación de rama, información utilizada para corroborar la pertenencia a un nivel de anidamiento dado por una estructura *Fork-Join*.
- Validación de rama.
- Apilar componentes en diversas estructuras para la validación de los mismos.
- Marcar componentes ya sea como visitados o libres según sea el caso.
- Almacenamiento de los errores provocados por flujo erróneos.
- Almacenamiento de los errores debido a etiquetas sin contenido.

En la figura 4.19 se observa el diagrama de flujo respectivo al método transición.

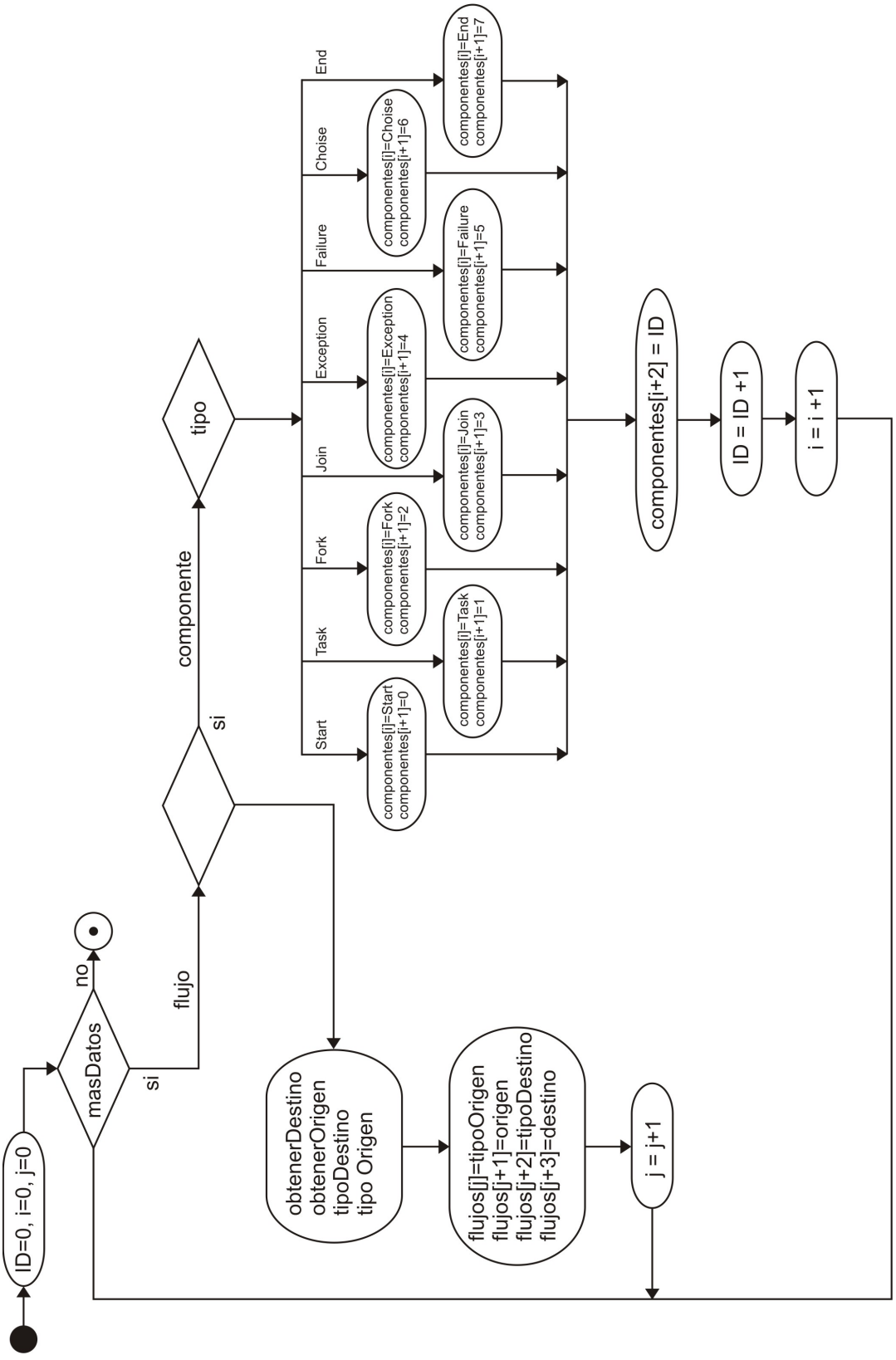


Figura 4.18: Diagrama de flujo utilizado para desarrollar el método procesaTFE.

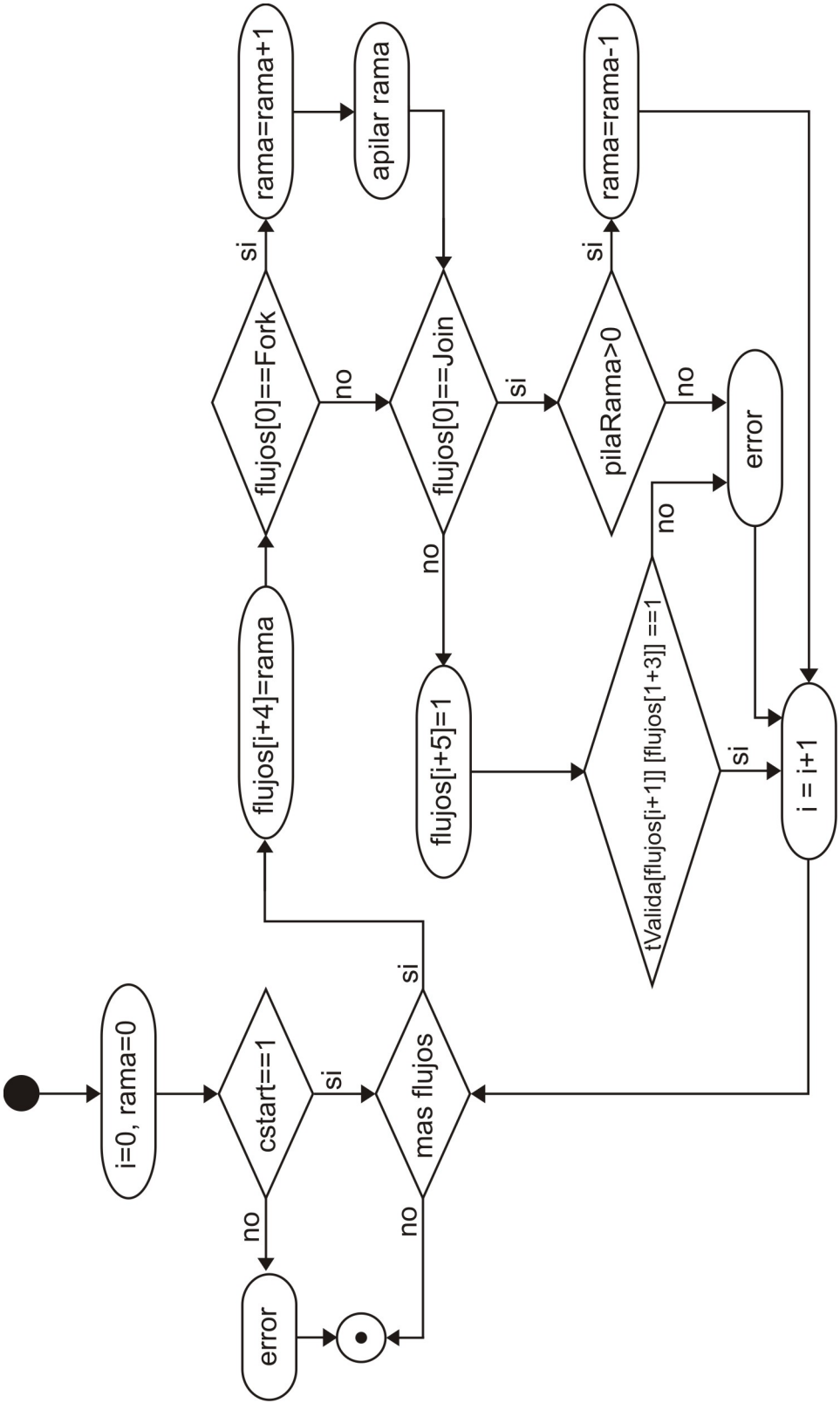


Figura 4.19: Diagrama de flujo correspondiente al método transición.

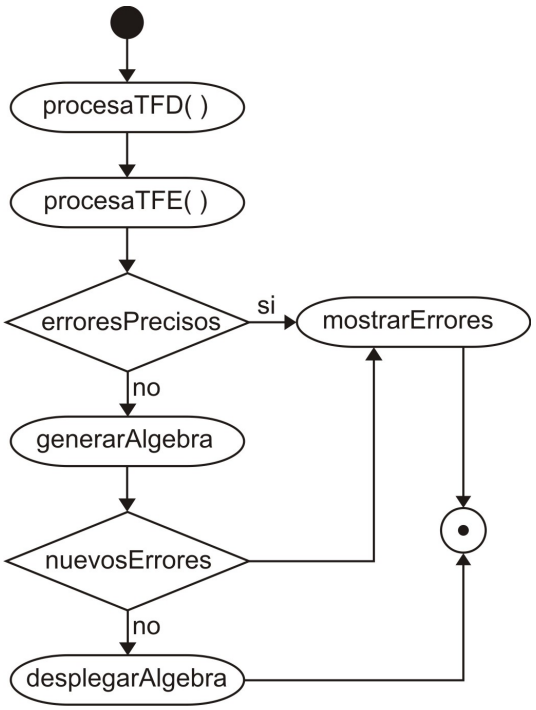


Figura 4.20: Diagrama de flujo usado para implementar el método parsearTareas.

4.7.1.4. Método parsearTareas

Este método es el encargado de coordinar a los métodos principales descritos anteriormente. Para ello se obtiene la información abstracta de los archivos que conforman al diagrama de flujo de tareas, es decir invoca a los métodos procesaTFE y procesaTFD. Posteriormente, se verifican los errores detectados en el método transición y si no existieran errores, se invoca al método generarAlgebra.

Una vez que el método generarAlgebra termina de ejecutarse, se verifica la existencia de los errores derivados en el proceso llevado a cabo para reducir expresiones, los errores que se hayan detectado se muestran en el área de problemas. Si no se presentan errores, se lleva a cabo un proceso de sustitución para mostrar en el álgebra de tareas la información referente a la etiqueta *task* del componente *Task* y no información abstracta utilizada para facilitar la manipulación de los diagramas. Finalmente la expresión del álgebra de tareas se devuelve para que se cree el archivo tfa y éste sea mostrado al usuario.

El proceso lógico de este método se muestra en la figura 4.20.

4.7.2. Implementación de la clase Reducir

Como se mencionó en la sección 4.7.1.4 la funcionalidad que implementa esta clase es llamada por el método parsearTareas. Sin embargo, los métodos públicos que ofrece se enfocan a dar información respeto a los datos procesados.

La lógica llevada a cabo consiste en buscar dentro de la estructura dinámica de flujos los patrones mostrados en la figura 3.7, el proceso inicia reduciendo tareas lineales, es decir, agrupando

componentes *Task*, estos componentes agrupados reciben el nombre de "tareas compuestas".

Una vez que se han reducido todas las tareas lineales existentes, se reducen las tareas que apunten a un elemento terminal, el cual puede ser un componente *End* o *Failure*. Esta agrupación genera un componente temporal terminal marcado como final.

El proceso siguiente es reducir las tareas compuestas. A partir de un par de tareas compuestas, se fusionan y se crea una nueva tarea compuesta.

Hasta este punto del proceso de reducción del diagrama de flujos de tareas actual ya tendría alguna de las estructuras identificadas en la figura 3.7, por lo tanto la próxima reducción tomará como punto de comparación al elemento *Choice*.

Es sobre el componente *Choice* que se realizarán las reducciones para formar las estructuras *For, Until* y *Choice*. Finalmente se crea el elemento *Fork-Join* a partir del análisis del componente *Fork*.

Cada vez que se lleva a cabo una reducción, se asigna un valor positivo a la bandera seguirReduciendo, de esta forma, cuando la bandera toma el valor falso el proceso termina. A continuación se detallan los métodos fundamentales implementados para realizar el proceso descrito.

4.7.2.1. Método reducirTareasLineales

Este método se enfoca en la detección y posterior unión de tareas lineales. Una tarea lineal se define como la sucesión de componentes *Task*, la fusión de un par de tareas lineas da como origen a una tarea compuesta. Para llevar a cabo este proceso, primero se debe determinar el número de entradas y salidas de los componentes *Task*, si para ambos es uno, entonces se procede a la fusión.

El procedimiento para fusionar las tareas se explica suponiendo la existencia de dos tareas, A con destino B, y consiste en:

1. Obtener el id destino de la tarea B.
2. Obtener la información de la localidad marca de B.
3. Sustituir el destino de la tarea A por el de B.
4. Fusionar el valor de la localidad marca de A con el de B.
5. Eliminar a B de la estructura flujos.

4.7.2.2. Método reducirTareasCompuesta

El método reducirTareasCompuestas lleva a cabo un proceso similar al método para tareas lineales, excepto que éste opera sobre tareas compuestas. Una tarea compuesta se define como la sucesión de componentes reducidos, la fusión de un par de tareas compuestas da como origen a otra tarea compuesta. Para llevar a cabo este proceso, primero se debe determinar el número de entradas y salidas de las tareas compuestas, si para ambos es uno, entonces se procede a la fusión.

El procedimiento para fusionar las tareas compuestas se explica suponiendo la existencia de dos elementos, A con destino B, y consiste en:

1. Obtener el id destino del elemento B.
2. Obtener la información de la localidad marca de B.

3. Modificar el tipo de componente en las estructuras dinámicas de flujos y componentes.
4. Sustituir el destino de la tarea A por el de B.
5. Fusionar el valor de la localidad marca de A con el de B.
6. Eliminar a B de la estructura flujos.

La eliminación del componente B es relativo, ya que ahora quedará contenido en A.

4.7.2.3. Método reducirTareaLinealConTerminal

Este método consiste en generar un elemento terminal etiquetado como final. Definimos al elemento terminal como la estructura compuesta por una secuencia de tareas lineales o tareas compuestas seguido de un componente *End*, o un componente *Failure* o un elemento terminal. Para obtenerlo, es necesario identificar una secuencia de tareas lineal o compuesta que apunta a un elemento terminal, una vez hecho esto, se lleva a cabo el siguiente procedimiento.

Suponiendo los elementos A y B como tareas compuestas y C como elemento terminal destino:

1. Identificar el tipo de componente C.
2. Si no se trata de un componente *End* o *Failure*, se obtiene el valor de la localidad marca de la estructura flujos.
3. Se obtiene el valor de la localidad marca de B.
4. Se sustituye destino de A por el valor de C.
5. Se fusiona la localidad marca de A con la B y con C, sólo si éste no fuera un componente *End* o *Failure*.
6. Se marca a B como elemento terminal.
7. Se elimina a C.

4.7.2.4. Método reducirFor

Basa su búsqueda únicamente en los componentes *Choice*, en cada recorrido donde se encuentre este elemento, será analizada la posible formación de una estructura *For*.

Cuando se detecta una estructura de este tipo, y suponiendo que se tiene la estructura de la figura 4.21, donde el elemento *Choice* apunta hacia A para formar la estructura *For* y hacia B cuando la condición no se cumple, tenemos que:

1. Se identifica el flujo que da salida al *For*, es decir el flujo que seguiría cuando la condición de iteración ya no se cumpla, en este caso B.
2. Se obtiene el valor de la localidad marca del componente al que apunta *Choice* cuando la condición se cumple, componente A.
3. A la localidad marca del *Choice* se le asigna la información obtenida en el paso 2 bajo el formato *Mu*(*Epsilon*+*datos*;x).
4. Se elimina el componente A.

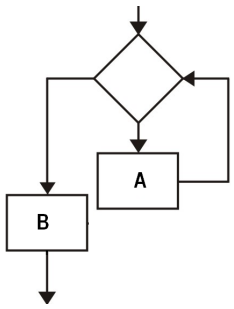


Figura 4.21: Estructura FOR identificada para ser reducida.

4.7.2.5. Método reducirUntil

De la misma forma en que el método reducirFor analiza los componentes *Choice*. Este método busca la formación de estructuras *Until*, con la diferencia de que el componente interno debe ser a su vez origen y destino, además el componente interno debe poseer dos flujos de entrada y uno de salida.

El procedimiento para reducir una estructura *Until* es el siguiente, suponiendo que se tiene la figura 4.22, donde el elemento A apunta a un *Choice*, éste último apunta al elemento A y cuando la condición no se cumple, el elemento *Choice* apunta al elemento B, tenemos que:

1. Se identifica el destino del componente *Choice* cuando la condición no se cumple, es decir cuando el flujo apunta a B.
2. Se sustituye el destino de A, por el destino obtenido anteriormente, es decir, el flujo de A tendrá como destino a B.
3. El campo marca del componente A se modifica para tener el formato *Mu*(*datos*;*Epsilon*+x).
4. El componente *Choice* es eliminado.

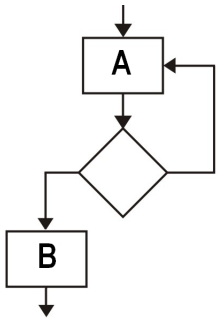


Figura 4.22: Estructura UNTIL identificada para ser reducida.

4.7.2.6. Método reducirParalelismo

La identificación de la estructura *Fork-Join* resulta hasta cierto punto sencilla. Para su identificación se necesita que por cada componente *Fork*, exista una tarea lineal o compuesta seguido

de un componente *Join*. Si hubiese una tarea terminal no es necesario que esta apunte a un componente *Join*. Para su reducción se lleva a cabo el siguiente procedimiento:

Suponiendo que se tiene la estructura de la figura 4.23, donde el componente *Fork* apunta a las tareas compuestas A_1, A_2 , ..., A_n donde todas las tareas desde A_1hasta A_n apuntan a un único componente *Join* y éste último apunta a un componente B, tenemos qué.

1. El contenido de la localidad marca del componente *Fork* se fusiona con el elemento A_1mediante el formato A_1|| A_2|| ... A_n.
2. Se elimina el componente A_1.
3. Esto se repite hasta A_n.
4. Se sustituye el destino del componente *Fork* por el destino del componente *Join*, en este caso B.
5. Se elimina el componente *Join*.

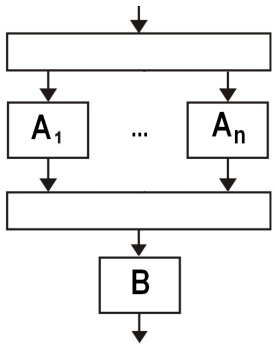


Figura 4.23: Estructura *Fork-Join* identificada para ser reducida.

4.7.2.7. Método getErrores

El método getErrores se activa únicamente cuando el diagrama no se puede reducir totalmente. Cuando esto sucede, se lleva a cabo un recorrido por el diagrama apilando en una estructura dinámica temporal los componentes que aún se encuentran en la estructura Flujos. Esta información se utiliza para sustituir los valores abstractos (ID) por la información proveniente del diagrama. Finalmente se envía como valor de retorno a la instancia que lo invocó.

4.7.3. Método reducirOr

El método reducirOr es el de mayor complejidad, ya que, como se muestra en la figura 3.7, existen 6 diferentes formas de reducirla. A continuación se detallará el proceso de reducción para cada una de ellas.

4.7.3.1. Reducción de un orSimple

Esta reducción se considera como el mejor de los casos, ya que este tipo de estructura posee contenido en ambos destinos del componente *Choice*, por lo tanto no es necesario llevar a cabo

ninguna evaluación. Para llevar a cabo este proceso suponemos que se tiene la estructura de la figura 4.24:

Los destinos del componente *Choice* se dirigen a un elemento A y B respectivamente, y cada uno de estos componentes tiene como destino común al elemento C, luego:

1. Se identifica el destino de A ó B, en este caso el elemento C.
2. Se obtiene el valor de la localidad marca de A y de B.
3. Se crea el contenido de la localidad marca del componente *Choice* bajo el formato (A+B).
4. Se sustituye en el componente *Choice* el destino B por C.
5. Se elimina el destino A y B.
6. Se modifica el tipo de componente de *Choice* a *Or*.

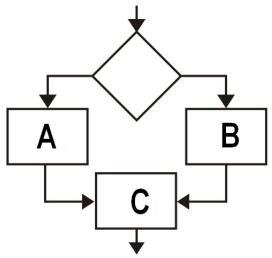


Figura 4.24: Estructura Or simple identificada para ser reducida.

4.7.3.2. Reducción de un orTerminalDoble

Este tipo de reducción es un caso especial de *Or*, ya que se convertirá en un elemento terminal, esto, debido a que ambos destinos del componente *Choice* tienen como destino alguna combinación de los elementos A y B del tipo *End, Failure* o un componente terminal. El proceso de reducción de la estructura (véase la figura 4.25) se detalla a continuación.

1. Se identifica el tipo de componente destino del elemento *Choice*, si es un componente terminal se almacena el valor de la localidad marca, en caso contrario se almacena el tipo.
2. Se identifica el tipo de componente del siguiente destino del elemento *Choice*, si es un componente terminal se almacena el valor de la localidad marca, en caso contrario se almacena el tipo.
3. Si uno o ambos tipos de destino corresponden a un componente terminal esta información se almacena en la localidad marca del elemento *Choice*, bajo el formato (A+B).
4. Si ambos componentes son elementos del tipo *End* o *Failure* o alguna combinación de ellos, se almacena el tipo del componente en la localidad marca del elemento *Choice* bajo el formato (A+B).
5. Se modifica el tipo de componente de *Choice* a Terminal.
6. Se eliminan los componentes A y B.

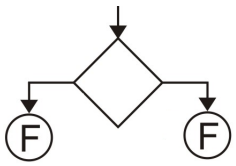


Figura 4.25: Estructura Or terminal doble identificada para ser reducida.

4.7.3.3. Reducción de un orTerminalSimple

Este tipo de reducción es un caso especial del *Or*, ya que como se observa en la figura 4.26 uno de los destinos del componente *Choice* es un componente terminal y el otro destino, por el contrario, es un elemento no terminal el cual pertenece a la estructura *Or*, es decir el único flujo que apunta a dicho componente no terminal, es el flujo destino del elemento *Choice*. Para el proceso de reducción, definimos un componente *Choice* que tiene como uno de sus destinos a un elemento terminal F, el otro destino se dirige a un elemento no terminal A, el cual a su vez tiene como destino un componente B. Entonces:

1. Se identifica el componente terminal F, si es un elemento *End* o *Failure*, se guarda su tipo de componente, en caso contrario se obtiene el valor de la localidad marca de la estructura dinámica de Flujos.
2. Se obtiene el valor de la localidad marca del componente no terminal A.
3. Se obtiene el destino del componente A, en este caso el elemento B.
4. Se fusiona el valor obtenido de los componentes F y A bajo el formato (F+A).
5. Se sustituye el destino A del componente *Choice* por B.
6. Se elimina el componente F.
7. Se elimina el componente A.
8. Se modifica el tipo de componente *Choice* por *Or*.

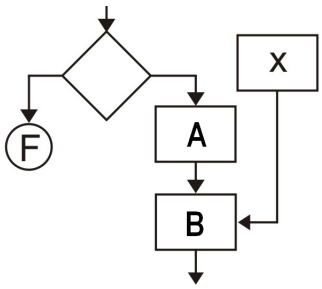


Figura 4.26: Estructura Or terminal simple identificada para ser reducida.

4.7.3.4. Reducción de un orTerminalSimpleVacio

Este tipo de reducción es similar a la llevada a cabo en orTerminalSimple. La variación que presenta es que ningún componente no terminal pertenece a la estructura *Or*, tal como se muestra

en la figura 4.27. Para el proceso de reducción, se define un componente *Choice* que tiene como destino a un elemento terminal F, el otro destino se dirige a un elemento no terminal A el cual no pertenece a la estructura *Or*. Es decir, el elemento A también es destino de otro componente. Por lo tanto tenemos que:

1. Se identifica el componente terminal F, si es un elemento *End* o *Failure*, se guarda su tipo de componente, en caso contrario se obtiene el valor de la localidad marca de la estructura dinámica Flujos.
2. Se fusiona el valor obtenido del componente F y el símbolo *Epsilon* bajo el formato (F+*Epsilon*).
3. Se elimina el componente F.
4. Se modifica el tipo de componente *Choice* por *Or*.

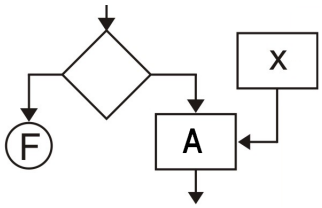


Figura 4.27: Estructura Or terminal simple vacio identificada para ser reducida.

4.7.3.5. Reducción de un orVacioSimple

La reducción de este componente es un tanto simple, similar al proceso llevado a cabo en orTerminalSimple, con la excepción de que el componente terminal no existe. Ese destino se dirige a un componente que está fuera de la estructura *Or*, tal como se muestra en la figura 4.28. El proceso de reducción se inicia suponiendo que el componente *Choice* tiene como destino al elemento A, y al elemento B, el cual también es destino del elemento A. Dadas estas suposiciones tenemos que:

1. Se obtiene el valor de la localidad marca del componente A en la estructura dinámica flujos.
2. Se sustituye el valor de la localidad marca del componente *Choice* por el contenido de A bajo el formato (A+*Epsilon*).
3. Se elimina el componente A.
4. El tipo de componente del *Choice* es modificado a *Or*.

4.7.3.6. Reducción de un orVacioDoble

Este método implementa la reducción más simple para la estructura *Or* y corresponde al que se muestra en la figura 4.29. El componente *Choice* tiene ambos destinos dirigidos a un único componente A, por lo tanto se realiza el siguiente proceso:

1. Se sustituye el valor de la localidad marca del componente *Choice* por la cadena (*Epsilon*+*Epsilon*).

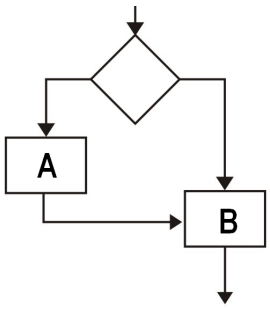


Figura 4.28: Estructura Or vacio simple identificada para ser reducida.

2. Se elimina un flujo que apunta hacia A.
3. El tipo de componente *Choice* se modifica por un *Or*.

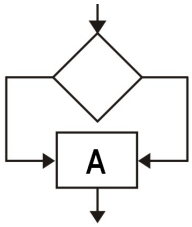


Figura 4.29: Estructura Or vacio doble identificada para ser reducida.

4.7.4. Implementación de la clase Exportar

Cómo última parte de la implementación del plug-in tenemos a la clase Exportar. Para describir su funcionamiento es importante recalcar que el proceso de exportación al formato XMI requiere de la intervención directa del usuario, esto al indicar la ruta y nombre del archivo el cual contendrá la información del diagrama de flujos de tareas, además, este proceso de exportación no implica la validación del diagrama, ya que el diagrama de flujos de tareas que se encuentre activo será exportado sin revisión alguna.

Los métodos principales que implementan la lógica de la aplicación son crearNombre y crearXMI, ambos se describen a continuación.

4.7.4.1. Método crearNombre

Este método tiene la función de validar el nombre asignado por el usuario, si este nombre no es correcto entonces se devuelve como nombre por defecto “diagrama.xmi”. El proceso de validación se muestra a continuación:

1. Se recibe la cadena y se valida que tenga longitud mayor o igual a 1.
2. Se valida que tenga una extensión.
3. En caso de no tenerla se le asigna la extensión *.xmi

4. Por el contrario, si el nombre del archivo no es mayor a 1 se asigna por defecto el nombre "diagrama.xmi".

4.7.4.2. Método crearXMI

Finalmente se describe la lógica del método crearXMI. Cabe resaltar que el diagrama de flujos de tareas exportado se convierte a un diagrama de actividades de UML correspondiente a XMI versión 1.1 y UML versión 1.3.

El proceso de creación del diagrama se basa en la sustitución de algunas de las etiquetadas del diagrama de flujos de tareas por su equivalente en UML 1.3, esta equivalencia se muestra en el cuadro 4.1.

Modelo TFE	Modelo XMI
hasComponents	UML:ActionState
hasFlow	UML:Transition
type	sin equivalencia
task	name
condf, condv, cone	nombre en flujo
source://@hasComponents.X	source=UMLPseudoState.X
target://@hasComponents.X	source=UMLPseudoState.X

Cuadro 4.1: Tabla de equivalencias entre el modelo TFE y el modelo XMI

Los componentes *Choice* y *Exception* poseen la misma equivalencia en el diagrama de actividades de UML. La secuencia para realiza la conversión es la siguiente:

1. Definir el encabezado del archivo XMI.
2. Sustituir el componente *Task* por el elemento *UMLActionState.X*, donde X corresponde al ID de la tarea.
3. Sustituir el valor *task* del componente *Task* por la etiqueta *name*.
4. Sustituir el componente *Start* por el elemento *UMLPseudoState.X*, donde X corresponde al ID del componente.
5. Crear la etiqueta *name* del componente inicial y asignar el valor Initial1.
6. Sustituir el componente *End* o *Failure* por el elemento *UMLFinalState.X*, donde X corresponde al ID del componente.
7. Crear la etiqueta *name* del componente final y asignar el valor *FinalState1*.
8. Para cualquier otro elemento sustituirlo por el valor *UML:PseudoState.X*, donde X es el ID del componente.
9. Crear la etiqueta *name* del componente y asignar el valor *DesicionX*, *SynchronizationX* según corresponda.

De esta manera es como se obtiene finalmente el plug-in.

En el siguiente capítulo se presenta el uso del plug-in aplicado a un ejemplo concreto, mediante el cual se mostrarán sus diversas funcionalidades.

Capítulo 5

Caso de estudio

Se analizará el comportamiento de las diferentes estructuras reconocidas por el plug-in, así como su traducción al álgebra de tareas. Las estructuras expuestas en las sección 5.1 se conjugan en el caso de estudio de la sección 5.2, el cual corresponde al diseño real de una aplicación y la inclusión del Método Discovery, en específico los diagramas de flujo de tareas y su correspondiente álgebra de tareas.

5.1. Estructuras de reducción

Como se mencionó en la sección 3.4, el plug-in genera el álgebra de tareas al llevar a cabo una reducción de componentes a estructuras temporales; cada una de estas estructuras se pusieron a prueba, para ello se creó el diagrama correspondiente y se obtuvo el álgebra asociada al diagrama.

El plug-in genera el álgebra de tareas asociado a un diagrama de flujo de tareas, sin embargo, para comprobar la salida, es necesario utilizar el compilador del álgebra de tareas.

Dicho compilador transforma la expresión del álgebra tareas y, si la expresión es correcta, genera las trazas correspondientes para la expresión.

El resultado es un sistema de tareas dado como un conjunto de trazas, en el que una sola traza es una cadena simbólica que denota una ruta de posible de ejecución a través del sistema y el conjunto de trazas denota todas las rutas de ejecución posibles [16].

A continuación se presentan las estructuras temporales implementadas en el plug-in.

5.1.1. Estructura lineal

La estructura lineal de tareas (mostrada en la figura 5.1) es la composición básica para reducir diagramas, su correspondiente álgebra de tareas se muestra en el cuadro 5.1.

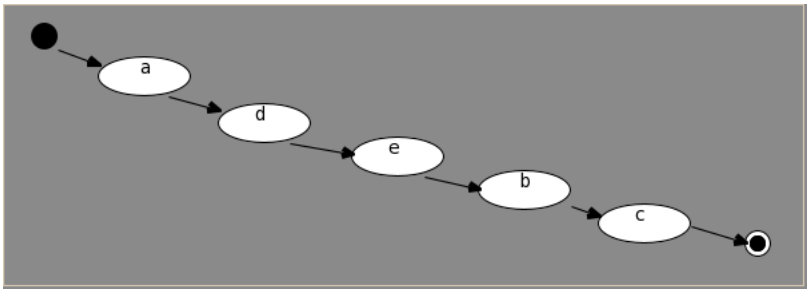


Figura 5.1: Diagrama de flujos de tareas mostrando una estructura lineal.

{a;d;e;b;c;Sigma} fromList [[a,d,e,b,c]]

Cuadro 5.1: Álgebra de tareas para una estructura lineal y las trazas derivadas del álgebra.

5.1.2. Estructura Fork

Una forma compleja de la estructura Fork se muestra en la imagen 5.2, ya que contiene un elemento terminal el cual no posee un flujo que vaya hacia el componente *Join*. El álgebra de tareas de este diagrama y las trazas originadas derivadas del álgebra de tareas se presentan en el cuadro 5.2.

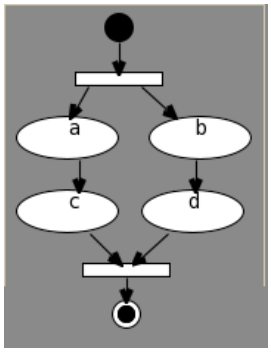


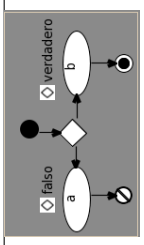
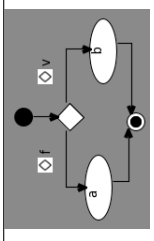
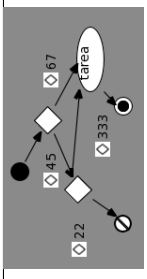
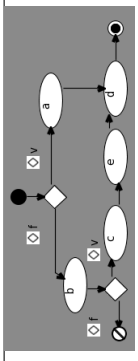
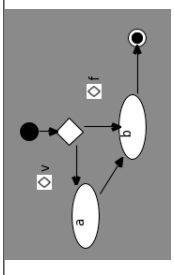
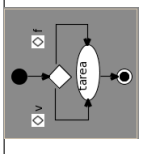
Figura 5.2: Diagrama de flujos de tareas mostrando una estructura Fork/Join.

{((a;c)||(b;d));Sigma} fromList [[a,b,c,d],[a,b,d,c],[a,c,b,d],[b,a,c,d],[b,a,d,c],[b,d,a,c]]

Cuadro 5.2: Álgebra de tareas para una estructura Fork/Join y las trazas derivadas del álgebra.

5.1.3. Estructura or

La estructura or presenta una serie de variantes, debido a que puede contener componentes terminales o vacíos, el cuadro 5.3 muestra las posibles formas de construir la estructura or, así como su correspondiente álgebra de tareas.

Diagrama de flujo de tareas	Álgebra de tareas	Trazas
 Estructura Or terminal doble.	{(a;Phi)+(b;Sigma)}	fromList [[!,a,Phi],[!,b]]
 Estructura Or simple.	{((b)+(a));Sigma}	fromList [[!,a],[!,b]]
 Estructura Or terminal simple vacio.	{(((Phi+Epsilon))+Epsilon);tarea;Sigma}	fromList [[!,tarea],[!,Phi]]
 Estructura Or terminal simple.	{((a)+(b;(Phi+(c;e))));d;Sigma}	fromList [[!,a,d],[!,b,!,c,e,d],[!,b,!,Phi]]
 Estructura Or vacio simple.	{((a)+Epsilon);b;Sigma}	[[!,a,b],[!,b]]
 Estructura Or vacio doble.	{(Epsilon+Epsilon);tarea;Sigma}	fromList [[tarea]]

Cuadro 5.3: Posibles formas de construir la estructura or.

5.1.4. Estructura Until

La forma básica de la estructura *Until* se presenta en la figura 5.3, de igual forma en el cuadro 5.4 se muestra el álgebra de tareas asociado a dicha estructura y sus correspondientes trazas originadas por el álgebra de tareas.

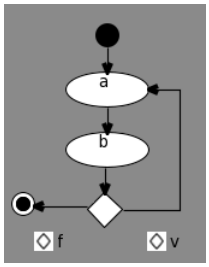


Figura 5.3: Diagrama de flujos de tareas mostrando una estructura Until.

```
{Mu.x((a;b);Epsilon+x);Sigma}

fromList [[a,b,!],[a,b,!,a,b]]
```

Cuadro 5.4: Álgebra de tareas para una estructura Until y las trazas derivadas del álgebra.

5.1.5. Estructura For

La figura 5.4, muestra un diagrama de flujo de tareas con la estructura *For*. El álgebra de tareas de este diagrama y las trazas originadas derivadas del álgebra de tareas se presentan en el cuadro 5.5.

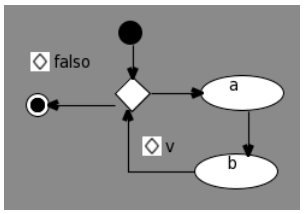


Figura 5.4: Diagrama de flujos de tareas mostrando una estructura For.

```
{Mu.x(Epsilon+(a;b);x);Sigma}

fromList [[!],[!,a,b,!],[!,a,b,!,a,b]]
```

Cuadro 5.5: Álgebra de tareas para una estructura For y las trazas derivadas del álgebra.

5.2. Definición del caso de estudio

A continuación se presenta una parte del análisis del proyecto *"Integrated Development Environment Gesture for modeling workflow diagrams"* (IDEG-MWD) propuesto en el Congreso Internacional de Investigación e Innovación en Ingeniería de Software 2012 [14].

El trabajo aborda la propuesta para el diseño e implementación de un prototipo de IDE para el diseño de diagramas de flujo de tareas. A diferencia del plug-in de Eclipse, la forma de interacturar que soportará IDEG-MWD será dada por la capacidad de tomar como medio de entrada los movimientos que el usuario realice con las manos. Esto con la finalidad de generar una nueva línea de interacción aprovechando las tecnologías que permitan añadir una especie de comunicación natural mediante la identificación y establecimiento de ciertos patrones descritos por movimientos.

Para desarrollar el caso de estudio se tomarán un par de casos de uso del proyecto IDEG-MWD. Basado en estos mismos casos de uso se llevará a cabo el diseño asociado y definido por los diagramas de flujos de tareas. Posterior al diseño de los diagramas se obtendrá su respectiva álgebra de tareas.

De esta forma quedará ejemplificado el funcionamiento del plug-in. Para conocer los detalles necesarios para la creación de los diagramas se puede consultar el anexo F.

Los casos de uso que se toman en cuenta son trazar componente y exportar a formato XMI, el diagrama de casos de uso se muestra en la figura 5.5.

5.3. Caso de uso trazar componente

El caso de uso trazar componente se detalló con la finalidad de mostrarlo en un contexto más apegado a las tecnologías a utilizar. Se tiene identificado a un único actor, el usuario. El caso se uso se presenta bajo el formato de la plantilla propuesta en OpenUP.

El cuadro 5.6 muestra la secuencia necesaria para trazar un componente, para ello es imprescindible tener un diagrama activo. Previamente se debe tener contemplado qué componente se desea colocar en el área de trabajo, ya que para hacerlo es necesario colocar la mano para que la proyección se haga sobre el componente deseado.

Se debe esperar algunos segundos para que el sistema responda, esta respuesta es visual, ya que el componente quedará sombreado y anclado al cursor.

Finalmente cuando se requiera colocar el cursor en la posición indicada, se deberá mantener dicha posición por algunos segundos, cuando este proceso se cumpla, se mostrará una respuesta visual.

Este proceso implica la realización de dos tareas de forma simultanea: la advertencia visual y la asignación del ID al componente.

5.3.1. Diagrama de flujo de tareas para trazar componente

Para la construcción del respectivo diagrama de flujos de tareas se toma como punto de vista el del usuario. La figura 5.6 muestra el diagrama donde se describe qué proceso se lleva a cabo

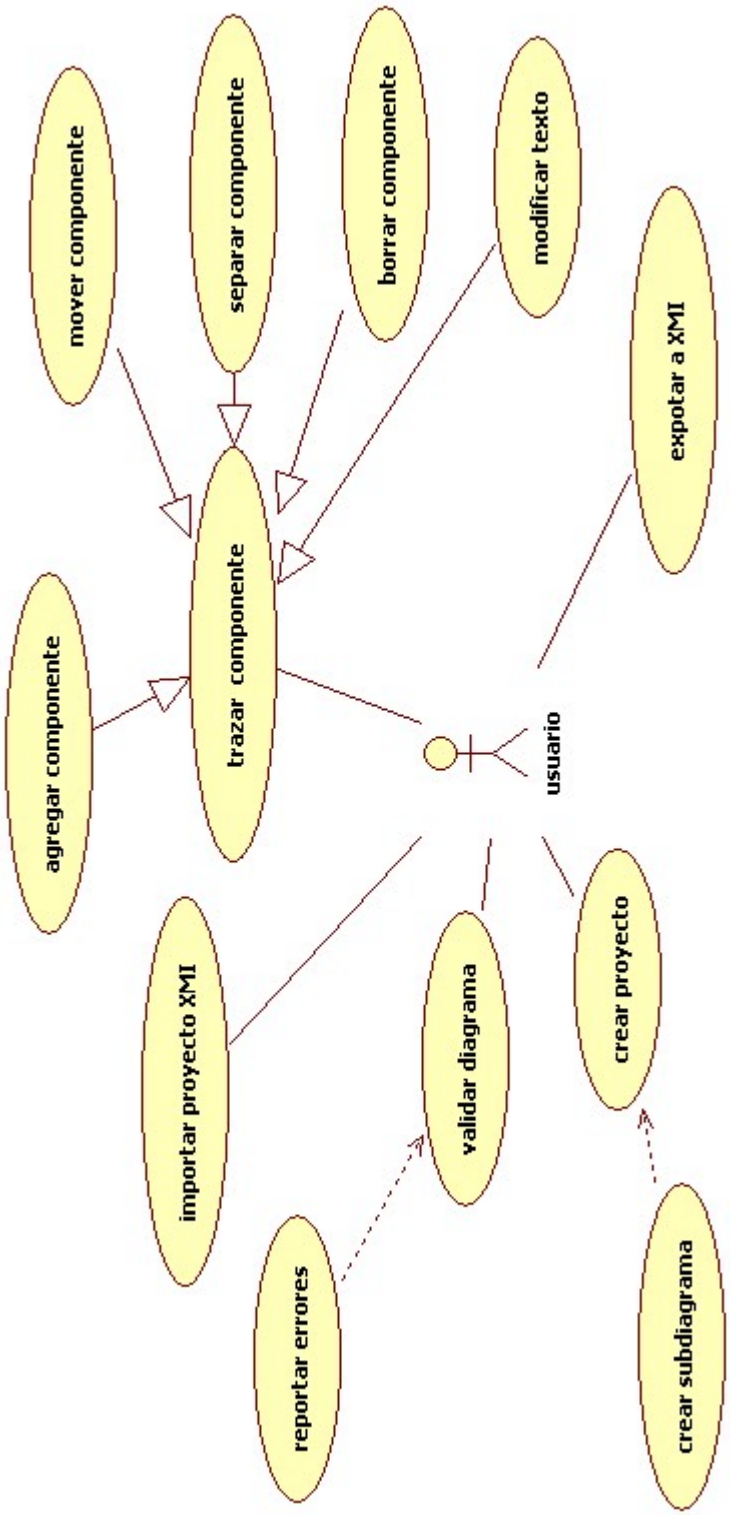


Figura 5.5: Diagrama de casos de uso para el IDEG-MWD.

Plug-in
Use-Case: trazar componente.

1. Brief Description
   El caso de uso tienen como propósito colocar un componente de clase en el diagrama de flujo de tareas. El usuario selecciona el componente visual mostrado en la barra de herramientas y lo coloca dentro del área de trabajo de un diagrama activo.
2. Actor Brief Descriptions
   El actor involucrado en el caso de uso es el desarrollador, el cual pretende colocar un componente dentro de un diagrama flujo de tareas.
3. Preconditions
   1. Debe existir un diagrama activo.
4. Basic Flow of Events

Acción del actor	Respuesta del sistema
1.- El caso de uso inicia cuando el usuario determina que componente va a seleccionar. El usuario proyecta la palma de su mano sobre el componente seleccionado.	2.- El sistema responde sombreando el componente apuntado por el usuario.
3.- El usuario desplaza por el área de trabajo el componente seleccionado.	4.- El sistema responde cambiando la posición del componente dado por el movimiento de la mano del usuario.
5.- Cuando el usuario decide colocar el componente en algún lugar específico, este deberá colocar la mano proyectando la posición deseada.	6.- Al detectar que un componente permanece un tiempo en la misma posición, el sistema fija ese componente en esa área. 7.- Se genera una ID único para identificar el componente, de igual forma se muestra una advertencia visual para denotar que el componente se ha fijado.

5. Alternative Flows
   Ninguno.
6. Subflows
   Ninguno.
7. Key Scenarios
   El diagrama se le agrega el componente seleccionado por el usuario.
8. Post-conditions
   El componente es mostrado visualmente en el diagrama, ademas de tener un ID único.
9. Special Requirements
   Ninguno.

Cuadro 5.6: Caso de uso trazar componente presentado bajo el formato de OpenUp.

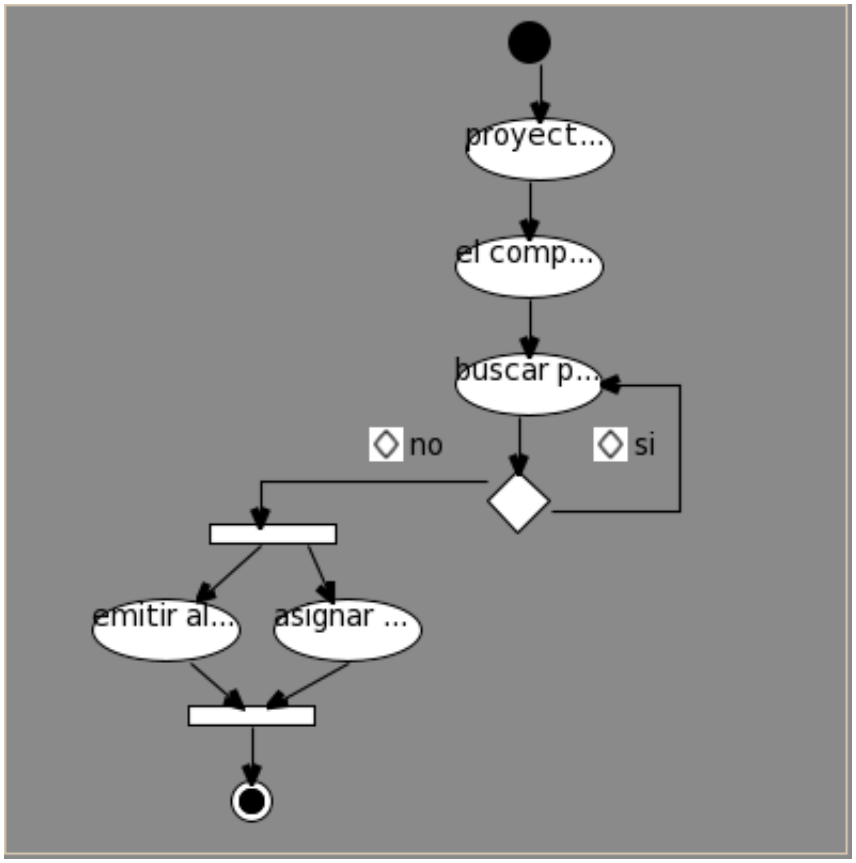


Figura 5.6: Diagrama de flujos de tareas diseñado sobre el plug-in.

para que un componente se coloque dentro de un diagrama de flujos de tareas. La figura 5.7 muestra el diagrama de flujo de tareas en el área de trabajo del plug-in.

El diagrama de flujo de tareas se compone de 5 tareas, donde una de ellas conforma el ciclo de repetición *Until*, dos más se realizan de forma paralela. Las tareas que se crearon en el diagrama llamado trazarComponenteIDEG.tfd son las siguiente:

1. Proyectar la palma de la mano sobre el componente deseado.
2. El componente se sombrea y ancla al cursor.
3. Buscar posición del componente
4. Emitir alerta visual.
5. Asignar el ID del componente

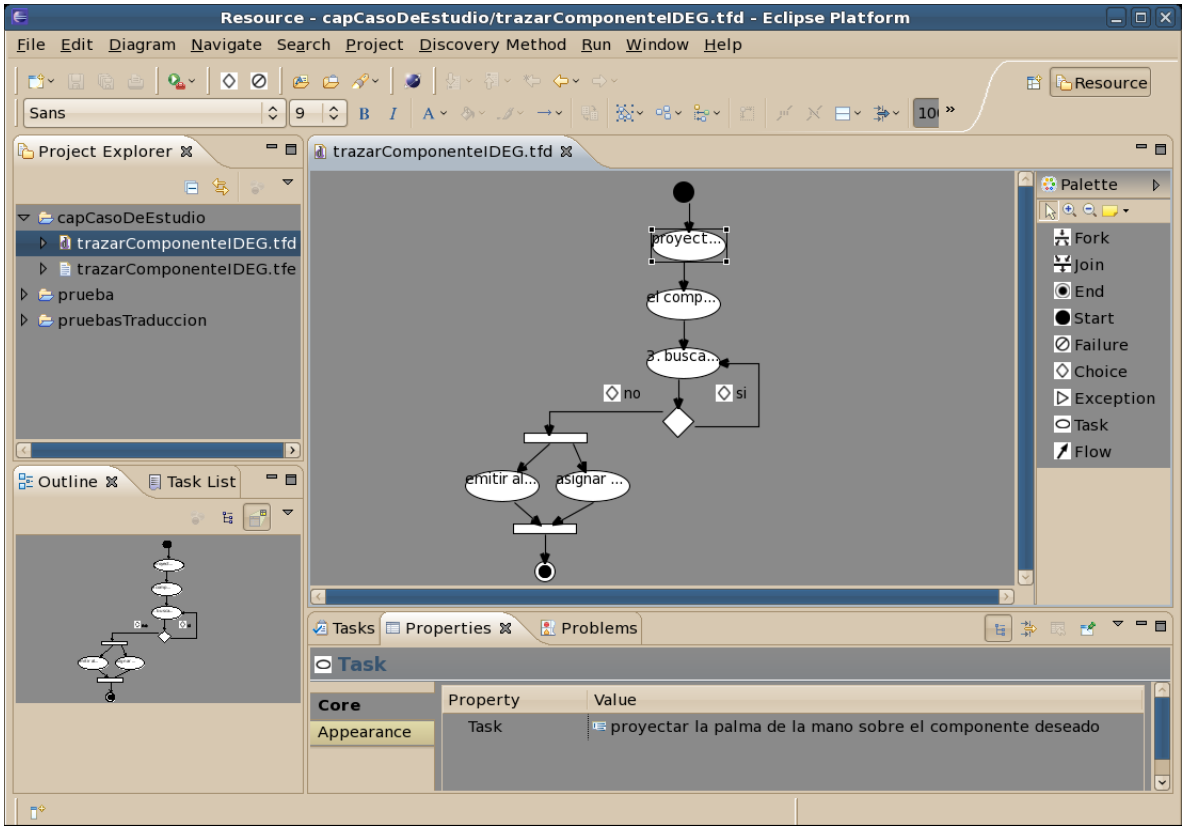


Figura 5.7: Diagrama de flujos de tareas diseñado sobre el plug-in.

El álgebra de tareas respectivo a la traducción del diagrama de flujo de tareas trazar componente, se despliega en un archivo denominado trazarComponente.tfa. Para una mejor comprensión del álgebra de tareas generado, se muestra el cuadro 5.7, donde el contenido ha sido sangrado de acuerdo al estilo Allman.

En la figura 5.8 se puede observar la salida real del archivo tfa. El álgebra de tareas no requiere ser sangrado bajo ningún estilo, ya que se pretende utilizar otra herramienta la cual enviará dicha álgebra al compilador, sin embargo, para facilitar su lectura el plug-in la sangra utilizando el estilo Allman. Es necesario recalcar que al prescindir de la herramienta y utilizar únicamente el compilador, es necesario no utilizar sangrado.

Cuando se obtiene el álgebra de tareas correspondiente al diagrama de flujos de tareas, se procesa por el compilador y como resultado se obtienen las trazas del cuadro 5.8.

5.4. Caso de uso exportar a XMI

El caso de uso exportar a XMI también ha sido detallado y expandido, esto con la misma finalidad de mostrarlo en un contexto más apegado a las tecnologías a utilizar, el único actor es el usuario. La presentación del caso de uso se hace mediante el formato de la plantilla propuesta en OpenUP.

```
{
   proyectarLaPalmaDeLaManoSobreElComponenteDeseado;
   elComponenteEsSombreadoYAncladoAlCursor;
   Mu.x
   (
      (
         buscarPosiciónDelComponente
      );
      Epsilon
      +
      x
   );
   (
      emitirAlertaVisual
      ||
      asignarElIDDelComponente
   );
   Sigma
}
```

Cuadro 5.7: Álgebra de tareas del diagrama de la figura 5.7 sangrado con el estilo Allman.

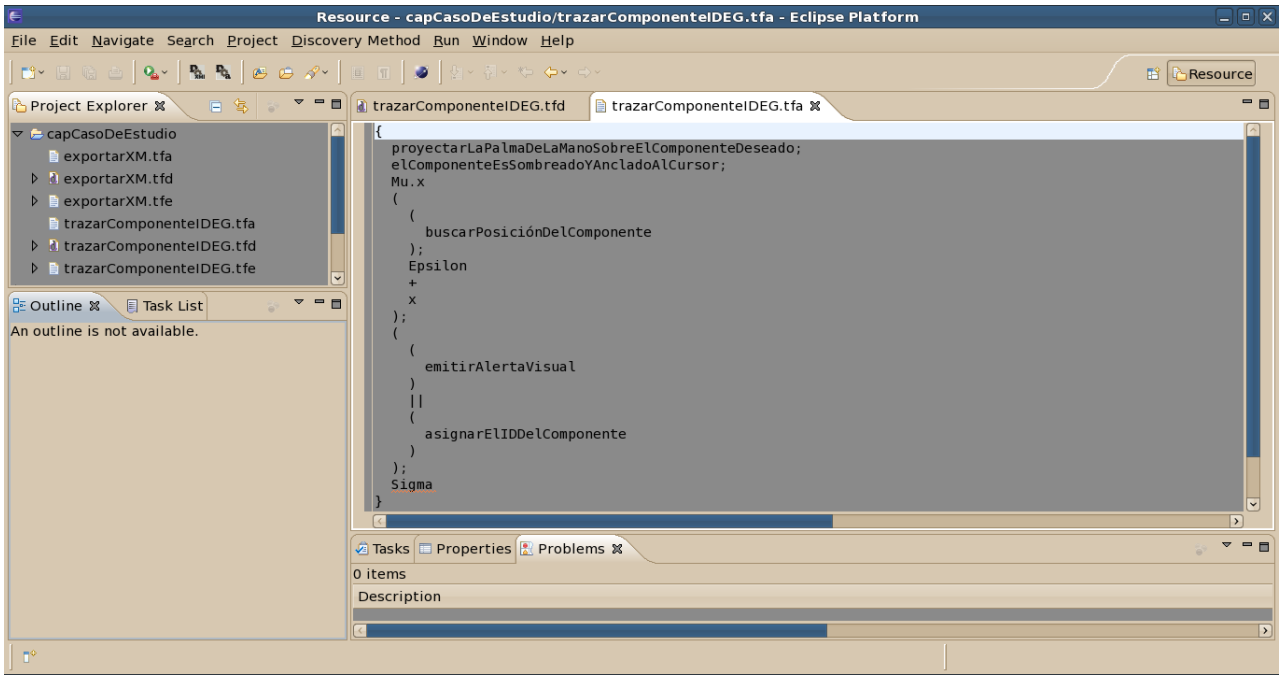


Figura 5.8: Álgebra de tareas generada por el plug-in.

fromList
[[proyectarLaPalmaDeLaManoSobreElComponenteDeseado,elComponenteEsSombreadoYAncladoAlCursor,
buscarPosiciónDelComponente,!,asignarElIDDelComponente,emitirAlertaVisual],
[proyectarLaPalmaDeLaManoSobreElComponenteDeseado,elComponenteEsSombreadoYAncladoAlCursor,
buscarPosiciónDelComponente,!,buscarPosiciónDelComponente,asignarElIDDelComponente,
emitirAlertaVisual],[proyectarLaPalmaDeLaManoSobreElComponenteDeseado,
elComponenteEsSombreadoYAncladoAlCursor,buscarPosiciónDelComponente,!,
buscarPosiciónDelComponente,emitirAlertaVisual,asignarElIDDelComponente],
[proyectarLaPalmaDeLaManoSobreElComponenteDeseado,elComponenteEsSombreadoYAncladoAlCursor,
buscarPosiciónDelComponente,!,emitirAlertaVisual,asignarElIDDelComponente]]

Cuadro 5.8: Trazas correspondientes al diagrama de flujo de tareas trazar componente.

En el cuadro 5.9 se muestra la secuencia necesaria para que el usuario pueda exportar un diagrama de flujo de tareas.

Para ello como primer punto es necesario tener un diagrama activo. Posteriormente se debe asignar un nombre al archivo que contendrá la información del diagrama de flujo de tareas. El sistema automáticamente cambia el formato del nombre de archivo para asignar la cadena .xmi como extensión del archivo.

Por último, la información del diagrama se guarda en el archivo recién creado, a la vez se muestra una alerta visual.

5.4.1. Diagrama de flujo de tareas para exportar a XMI

El punto de vista del usuario en este caso identificado como el único actor, es el que se toma en cuenta para la construcción del diagrama de flujo de tareas. La figura 5.9 muestra el diagrama de flujo de tareas por separado y la figura 5.10 muestra el mismo diagrama dentro del área de trabajo del IDE el cual describe el proceso llevado a cabo para que un diagrama de flujo de tareas sea exportado a formato XMI.

El diagrama inicia verificando si existe un diagrama activo. En caso de que no exista, el proceso finaliza. Si por el contrario el proceso continúa se tienen 5 tareas, 3 de las cuales son: el proceso para mostrar el cuadro de diálogo para el nombre del archivo, dar el formato correcto a la cadenas (si el nombre dado no es correcto se repite el proceso anterior) y la creación del archivo para almacenar el diagrama. Estas forman parte de una estructura de tipo *Until*. Dos más se realizan de forma paralela.

Las tareas que se crearon en el diagrama llamado expotarXMI.tfd son las siguientes:

1. Mostrar cuadro de diálogo para capturar el nombre del archivo.
2. Dar formato al nombre del archivo.
3. Crear el archivo xmi.
4. Guardar información.
5. Emitir alerta visual.

Plug-in
Use-Case: exportar a XMI.

1. Brief Description
   El caso de uso tienen como propósito crear un diagrama de actividades de UML 1.3 a partir del diagrama de flujo de tareas.
2. Actor Brief Descriptions
   El actor involucrado en el caso de uso es el desarrollador, él requiere tener el diagrama en un formato compatible, para ello debe saber que nombre asignarle.
3. Preconditions
   1. Debe existir un diagrama activo.

4. Basic Flow of Events

Acción del actor	Respuesta del sistema
1.- El caso de uso inicia cuando el usuario determina que el diagrama debe ser exportado al formato XMI, y este selecciona la opción de exportar a XMI.	2.- El sistema responde evaluando si hay diagrama activo, en caso contrario emite una advertencia y termina el proceso. 3.- El sistema muestra el cuadro de diálogo para asignar un nombre.
4.- El usuario asigna un nombre y ruta para el archivo.	5.- El sistema da el formato correcto al nombre del archivo. 6.- El sistema verifica que el nombre sea válido, en caso contrario se ejecuta el flujo 3 7.- El sistema crea el archivo XMI, en caso de no poder hacerlo se inicia el paso 3. 8.- De la misma forma el sistema guarda la información en el archivo XMI, y muestra un mensaje de final del procedimiento..

1. Alternative Flows
   Ninguno.
2. Subflows
   Ninguno.
3. Key Scenarios
   El diagrama es exportado a un formato XMI y almacenado.
4. Post-conditions
   El diagrama se encuentra en la ubicación dada por el usuario.
5. Special Requirements
   Ninguno.

Cuadro 5.9: Caso de uso exportar a XMI presentado bajo el formato de OpenUp.

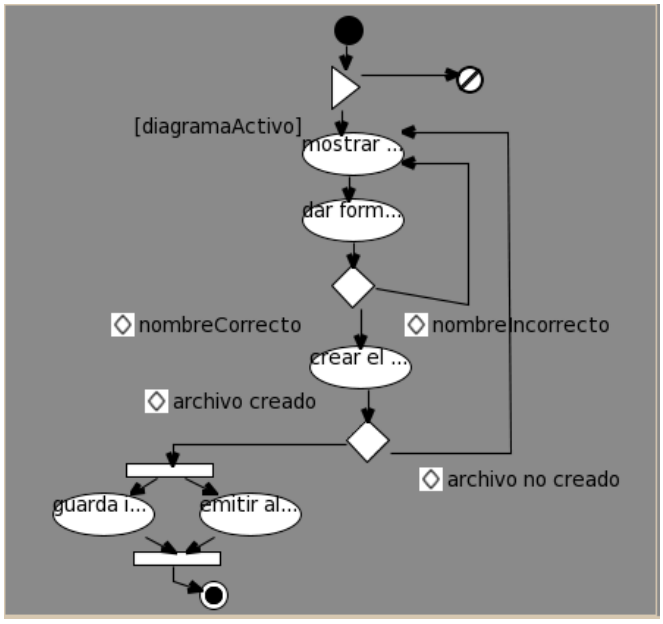


Figura 5.9: Diagrama de flujos de tareas diseñado sobre el plug-in.

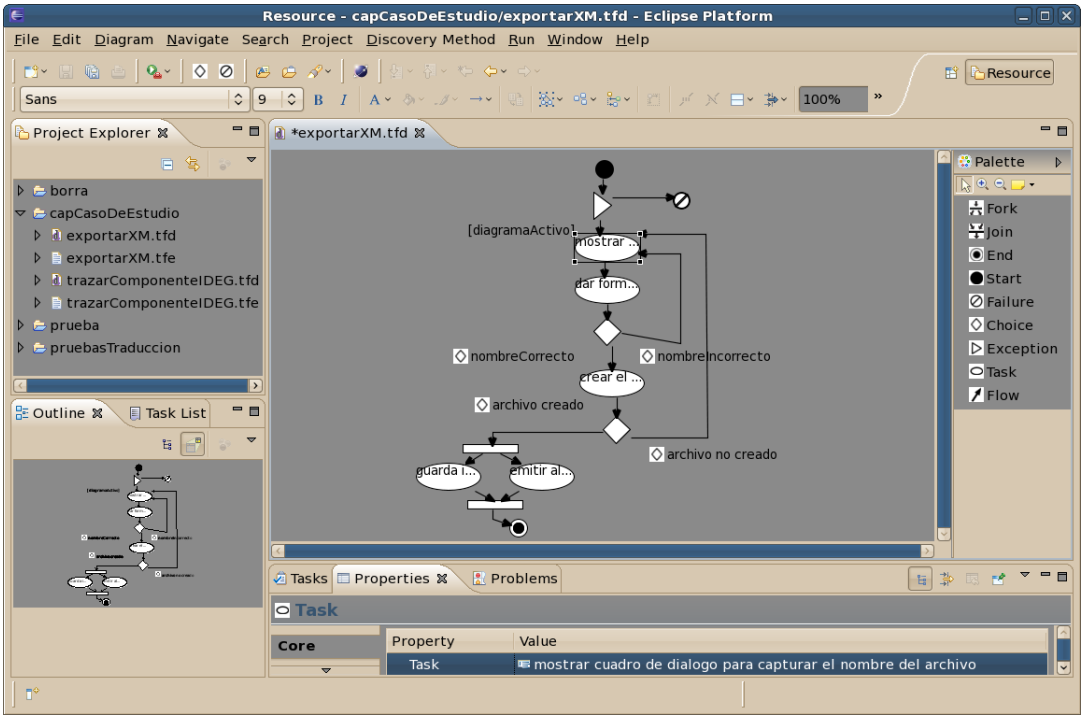


Figura 5.10: Diagrama de flujos de tareas diseñado sobre el plug-in.

El álgebra de tareas respectivo a la traducción del diagrama de flujo de tareas exportarXMI.tfd, se despliega en un archivo denominado exportarXMI.tfa.

Para una mejor comprensión del álgebra de tareas generado, se muestra el cuadro 5.10, donde el contenido ha sido sangrado de acuerdo al estilo Allman.

```
{
 (
  (
   Mu.x
   (
    (
     Mu.x
     (
      (
       mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo;
       darFormatoAlNombreDelArchivo
      );
      Epsilon
      +
      x
     );
     crearElArchivoXmi
    );
    Epsilon
    +
    x
   );
   (
    (
     guardaInformacion
    )
    ||
    (
     emitirAlertaVisual
    )
   );
   Sigma
  )
  +
  Phi
 )
}
```

Cuadro 5.10: Álgebra de tareas del diagrama de la figura 5.10 sangrado con el estilo Allman.

En la figura 5.11 se puede observar la salida real del archivo tfa, en ella el álgebra de tareas es sangrado bajo el estilo Allman.

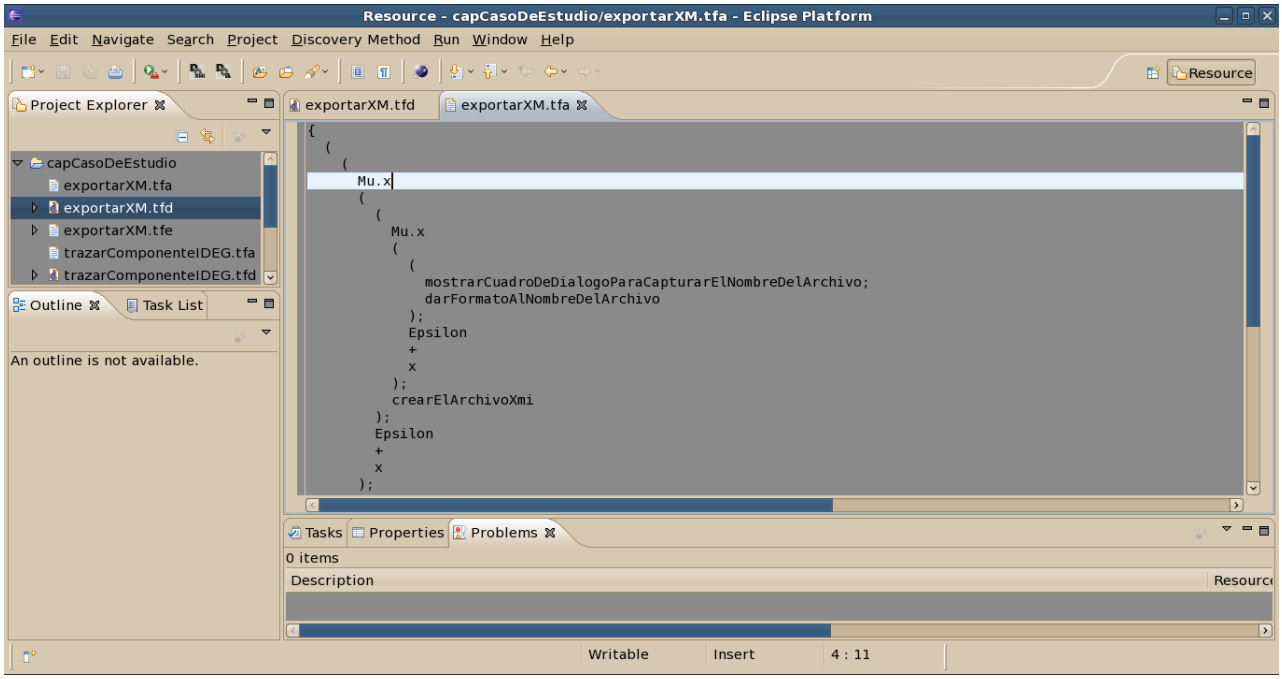


Figura 5.11: Álgebra de tareas generada por el plug-in.

Finalmente con el álgebra de tareas obtenida correspondiente al diagrama de flujos de tareas, se procede a utilizar el compilador del álgebra de tareas. Como resultado se obtienen las trazas del cuadro 5.11.

fromList [[!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,!,crearElArchivoXmi,!,emitirAlertaVisual,
guardaInformacion],[!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,!,crearElArchivoXmi,!,guardaInformacion,emitirAlertaVisual],
[!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,!,crearElArchivoXmi,!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
!,crearElArchivoXmi,emitirAlertaVisual,guardaInformacion],[!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
!,crearElArchivoXmi,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,!,crearElArchivoXmi,guardaInformacion,emitirAlertaVisual],
[!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
!,crearElArchivoXmi,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,crearElArchivoXmi,emitirAlertaVisual,guardaInformacion],[!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
!,crearElArchivoXmi,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,crearElArchivoXmi,guardaInformacion,emitirAlertaVisual],
[!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
crearElArchivoXmi,!,emitirAlertaVisual,guardaInformacion],[!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,!,

mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
crearElArchivoXmi,!,guardaInformacion,emitirAlertaVisual],[!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
crearElArchivoXmi,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,!,crearElArchivoXmi,emitirAlertaVisual,guardaInformacion],[!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
crearElArchivoXmi,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,!,crearElArchivoXmi,guardaInformacion,emitirAlertaVisual],[!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
crearElArchivoXmi,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,crearElArchivoXmi,emitirAlertaVisual,guardaInformacion],[!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,!,
mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,darFormatoAlNombreDelArchivo,
crearElArchivoXmi,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,!,mostrarCuadroDeDialogoParaCapturarElNombreDelArchivo,
darFormatoAlNombreDelArchivo,crearElArchivoXmi,guardaInformacion,emitirAlertaVisual],[!,Phi]]

Cuadro 5.11: Trazas correspondientes al diagrama de flujo de tareas exportar a XMI.

.

En el capítulo final, se dan a conocer las conclusiones a las que se llegó al desarrollar el plug-in, así como los trabajos futuros a realizar para el proyecto.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

A lo largo del documento de tesis se mostró como se cumplieron de forma satisfactoria los objetivos planteados al inicio de la investigación, entre los principales objetivos tenemos la implementación del editor visual únicamente por medio del diseño de los diferentes modelos del *framework* GMF.

La principal ventaja, es que con ello se facilitará la posibilidad de crear nuevas versiones, en donde se agreguen funcionalidades propias de nuevas especificaciones del álgebra de tareas, ya que el modelo principal Ecore se convirtió en punto de partida para todo el desarrollo, bastaría con modificar este modelo y utilizar las herramientas que GMF provee para realizar las actualizaciones correspondientes.

La razón por la cual se concibió el plug-in fue ayudar en el proceso de automatización para la generación del álgebra de tareas, por medio del plug-in es que se le permite al desarrollador obtenerla de manera eficiente y rápida, lo que se pudo comprobar con las pruebas realizadas en el caso de estudio de la sección 5.3 y 5.4 donde se observa como únicamente basta crear los respectivos diagramas de flujos de tareas para poder obtener el álgebra de tareas correspondiente. El proceso completo para obtener el álgebra de tareas anteriormente requería una cantidad considerable de tiempo, además se estaba propenso a errores de traducción.

Eventualmente se unirá una serie de herramientas enfocadas a cubrir las etapas que conforman al Método Discovery, cumpliendo de esta manera uno de los objetivos planteados al inicio del trabajo.

Si bien es cierto que al haber desarrollado el plug-in basado en el IDE Eclipse, se tuvo poca flexibilidad para la implementación del editor visual, ésta falta de flexibilidad se ve compensada al momento de finalizar el plug-in, ya que se pone de manifiesto que al fundamentar el proyecto en un modelo estructurado, se puede disminuir el tiempo de desarrollo.

Es gratificante observar que el plug-in mantendrá un segmento de usuarios a los cuales ayudará a una mejor comprensión de la relación que existe entre el diagrama de flujo de tareas y el álgebra de tareas resultante, permitiéndoles centrar su atención a los resultados y no desviar sus esfuerzos al realizar el proceso manual para la obtención del álgebra de tareas.

Finalmente, al hacer una revisión del proceso técnico llevado a cabo para realizar el trabajo,

podemos darnos cuenta de que algunas funcionalidades no pudieron utilizarse rápidamente, ya que la documentación final no era liberada al mismo tiempo que las herramientas, esto provocaba un ligero retraso. La etapa de implementación fue relativamente sencilla, ya que las herramientas utilizadas ofrecen la transparencia necesaria (sin llegar a ser un obstáculo) para facilitar este proceso.

6.2. Trabajo futuro

La herramienta desarrollada es un trabajo el cual podrá estar en constante evolución, ya que se implementarán algunas otras funcionalidades de la especificación actual del álgebra de tareas, las cuales se acotaron para adecuarse al desarrollo en Eclipse.

En primera instancia, se tiene que para futuras versiones se implementará una funcionalidad más al componente *Choice*. Actualmente a partir del *Choice* se pueden construir tres estructuras temporales *For*, *Until* y *Or,* a estas se añadirá la estructura de selección múltiple, la cual tendrá un comportamiento similar a la sentencia *switch* del lenguaje C. También, se agregará la funcionalidad que permita el uso de subtareas.

De esta forma se finalizaría la implementación para satisfacer la especificación del álgebra de tareas. Además de los puntos anteriores, se deberá atender a las actualizaciones que surjan, algunas de ellas pueden consultarse en [15].

Otro aspecto a considerar es la falta de herramientas del Método Discovery, hasta este punto se encuentra en su fase final de desarrollo otro plug-in, ambos se complementarán y ayudarán a mejorar el proceso de automatización en el IDE Eclipse.

Es importante resaltar que esta herramienta tiene la posibilidad de poder adaptarse al diagrama de actividades, esto por la relación que guarda con el respectivo diagrama de flujo de tareas tal como se mencionó en la sección 2.6.

El presente trabajo servirá como fundamento para la creación de una nueva herramienta basada en el metamodelo del álgebra de tareas. Sin embargo, para este nuevo proyecto se deja de lado el concepto tradicional de IDE ya que éste se construirá mediante HTML 5, de esta manera ofrecerá un entorno multiplataforma.

El nuevo proyecto reutilizará el trabajo llevado a cabo en las clases pertenecientes a controlador del patrón de desarrollo Modelo-Vista-Controlador.

Referencias

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, y Pedro Flores-Suarez. *Compiladores: principios, técnicas y herramientas*. Addison-Wesley Iberoamericana, México, 1990. ISBN 968-444-333-1. 2.8

[2] Sayde Alcantara y Carlos Alberto Fernández-y Fernández. *Compilador y máquina virtual para un lenguaje de sociedades de agentes reactivos*. Huajuapan de León, Oaxaca, Febrero 2006. 2.8.1

[3] Chris Aniszczyk. Get started with model-driven development the eclipse way, Último acceso Febrero 6 2013. URL `http://www.ibm.com/developerworks/library/os-ecl-gmf/gmfoverview.gif`. (document), 4.2

[4] Pablo Barceló. Introducción a la verificación formal, Último acceso Noviembre 30 2010. URL `http://www.infor.uva.es/~jvalvarez/docencia/tema8.pdf`. 1.2

[5] Marchal Benot. *XML con ejemplos.* Pearson education, México, 2001. ISBN 970-26-0163-0. 3.2

[6] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 75 Arlington Street, Suite 300 Boston, 2007. ISBN 0-201-89551-X. 2.1

[7] Grady Booch. Uml in action. *Communications of the ACM*, (10), 1995. 1.3, 2.2, 2.3, 2.3.1

[8] Frank Budinsky, David Steinberg, Ed. Merks, Raymond Ellersick, y Timothy J. Grose. *Eclipse Modeling Framework: A Developer's Guide*. Addison Wesley, Boston, Massachusetts, firts edition, 2003. ISBN 0-13-142542-0. 1.1, 2.1, 2.7.6

[9] Corrado Böhm y Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9:366 –371, 1966. 3.4

[10] Michael Jesse Chonoles. *UML 2 for dummies*. Wiley Publishing, Inc., 909 Third Avenue New York, NY 10022, 2003. ISBN 0-7645-2614-6. 2.3.1, 2.5, 2.6

[11] Eric Clayberg y Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Publisher: Addison Wesley Professional, U.S.A, 2006. ISBN -13: 978-0-321-42672-7. 2.7.6

[12] Aurent Debrauwer. *UML 2 Iniciación, ejemplos y ejercicios resueltos.* Addison-Wesley, 75 Arlington Street, Suite 300 Boston, 2007. ISBN 0-201-89551-X. 1.3, 2.1, 2.3.1, 2.4, 2.4, 2.4, 2.6

[13] Fundación Eclipse. 2.7.5

[14] Carlos Alberto Fernández y Fernández y Jose Angel Quintanar-Morales. Integrated development environment gesture for modeling workflow diagrams. *Congreso Internacional*

de Investigacion e Innovacion en Ingenieria de Software (Conisoft 2012), abs/1205.0751, 2012. 5.2

[15] Carlos Alberto Fernández-y Fernández. Towards a new metamodel for the task flow model of the discovery method. *Congreso Internacional de Investigación e Innovación en Ingeniería de Software*, 2012. 2.7.5, 6.2

[16] Carlos Alberto Fernández-y Fernández. *The Abstract Semantics of Tasks and Activity in the Discovery Method*. PhD thesis, The University of Sheffield, Sheffield, UK, February 2010. (document), 1.2, 1.4, 1.5, 2.6.4, 2.7.1, 2.7.3, 2.7.4, 2.11, 2.17, 5.1

[17] Eclipse foundation. Introduction to openup, Último acceso Julio 3 2012. URL `http://epf.eclipse.org/wikis/openup/`. 2.9.1, 2.9.2

[18] Eclipse foundation. Openup basic lifecycle, Último acceso Julio 3 2012. URL `http://epf.eclipse.org/wikis/openupsp/openup_basic/guidances/supportingmaterials/resources/openup-basic_lifecycle.jpg`. (document), 2.20

[19] Martin Fowler. *UML Distilled*. Addison Wesley longman Inc., Massachusetts E.U.A, 2001. ISBN 0-201-32563-2. 1.3, 1.4, 2.1, 2.2, 2.3.1, 2.6, 3.10

[20] Iam Graham y Anthony J H Simons. 30 things that go wrong in object modelling with uml 1.3. *Kluwer Academic Publishers*, 1999. 1.3, 2.6.4

[21] D. Abowd Gregory. *Formal aspects of human-computer-interaction*. PhD thesis, Oxford England, 1991. 2.6.4

[22] Object Management Group. Uml 2.0 ocl final adopted specification, Último acceso Agosto 14 2012. URL `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14`. 2.6.4

[23] UML Group. Uml 2.0, Último acceso Septiembre 2 2009. URL `http://www.omg.org/news/pressroom.htm`. (document), 2.1, 2.2, 2.1, 2.4, 2.2, 2.5

[24] Eriksson Hans-Erik y Penker Magnus. *Bussines Modeling with UML*. John Wiley and Sons, Inc., USA, 2000. ISBN 0-471-29551-5. 2.6.3, 3.2

[25] Kenneth C. Louden. *Construcción de compiladores. principios y prácticas*. Ediciones Paraninfo, 2004. ISBN 9706862994. 2.8, 2.8.1

[26] L. Martino y E. Bertino. *Sistemas de bases de datos orientadas a objetos*. Editorial Díaz de Santos, S.A., Madrid España, 2005. ISBN 0201653567. 1.3

[27] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, y Philippe Vanderheyden. *Eclipse Developmentnt using the Graphical Editing Framework and the Eclipse Modeling Framework*. REDBooks, U.S.A, 2004. 2.7.6

[28] Salvador David Musat. *Tutorial de introducción a EMF y GMF*. Madrid España, 2011. 2.7.6

[29] T. Parkes y Anthony J H Simons. Discovery method tool, Último acceso Agosto 13 2009. URL `http://www.dcs.shef.ac.uk/~ajhs/discovery/tool/`. 1.4.2

[30] María Isabel Pita Andreu. *Técnicas de especificación formal de sistemas orientados a objetos basados en lógica de reescritura*. Universidad Complutense de Madrid, Madrid

España, 2002. 1.2

[31] Roger S. Pressman. *Ingeniería de Software, un enfoque práctico.* McGraw Hill, Madrid, España, fifth edition, 2002. ISBN 0-07-709677-0. 1.2, 2.1

[32] N. Quirón. Introducción a uml 2.0, la evolución de la programación hacia la ejecución y validación automática de modelos, Último acceso Septiembre 8 2010. URL `http://www.epidataconsulting.com/tikiwiki/tiki-read_article.php?articleId=15`. 1.5, 2.4, 2.4

[33] Joshep Schmuller. *UML en 24 hrs*. Prencice Hall, Madrid España, 2009. 1.3, 2.1, 2.4, 2.4, 2.5, 2.6.4

[34] Anthony J H Simons. *Object Discovery: A process for developing applications*. Oxford : BCS, 1998. 1.4, 1.5, 2.7.1

[35] Anthony J H Simons. Discovery history, Último acceso Junio 23 2012. URL `http://staffwww.dcs.shef.ac.uk/people/A.Simons/discovery/`. 2.7.1

[36] Anthony J H. Simons y Carlos Alberto Fernández-y Fernández. Using alloy to model-check visual design notations. *Sixth Mexican International Conference on computer Science*, 2005. 2.6.4

[37] Ian Sommerville. *Ingeniería del software*. Pearson educación, Madrid, España, seventh edition, 2005. ISBN 84-7829-074-5. 1.2, 1.4

[38] Richard Stallman. La comunidad del software libre 20 años después: Un gran pero incompleto éxito. ¿ahora qué? *www.gnu.org*, 2004. 1.5

[39] B. Teufel, S. Schmid, y T. Teufel. *Compiladores conceptos fundamentales*. AaddisonWesley Iberoamericana, México, 1995. ISBN 0-201-65365-6. 2.8.1

[40] Dolores R. Wallace, Laura M. Ippolito, y Barbara Cuthill. Reference information for the software verification and validation process. 1996. 1.2, 1.5

Apéndice A

Algoritmo para una sola pasada

```
variables pilas: simbolos=0s, RETORNOS, F_R, E_R, componenteAcual.
enteros: D, S.
INICIO
D=buscarDestino(S)
case (D.tipo)
    FAILURED:  TASK:
            si D.marca=0 AND D.entradas=1 // cuando se pasa por primera vez y sólo hay un flujo
                que apunte a esa tarea              guardar(simbolos,D)
               D.marca=1
            endSi
                si D.marca=0 AND D.entradas>1 // cuando se pasa por primera vez y sólo hay un
                    flujo que apunte a esa tarea                        guardar(simbolos,-2)
                    guardar(simbolos,D)
                    guardar(simbolos,-2)
                   D.marca=D.marca+1
                    case componenteActual.tope:
                        CHOICE:
                            D=pop(RETORNOS)
                        endChoice
                        FORK:
                            D=pop(F_R)
                        endFork
                        EXCEPTION:
                            D=pop(E_R)
                        endException
                    endCase
                   D.marca=1
                finSi
                    si D.marca<>0
                    S.tipo=choice AND si esPadre(D,S)
                        hacer(UNTIL)
                        case componenteActual.tope:
                            CHOICE:
                                D=pop(RETORNOS)
                            endChoice
                            FORK:
                                D=pop(F_R)
                            endFork
                            EXCEPTION:
                                D=pop(E_R)
                            endException
                        endCase
                        D.marca=2
                        sino
                            si D.marca<D.entradas
```

```
                                D.marca=D.marca+1
                                    si D.entradas=D.marca
                                        guardar(simbolos,-3)
                                        guardar(simbolos,D)
                                        guardar(simbolos,-3)
                                        sino
                                            guardar(simbolos,-2)
                                            guardar(simbolos,D)
                                            guardar(simbolos,-2)
                                            case componenteActual.tope:
                                                CHOICE:
                                                    D=pop(RETORNOS)
                                                endChoice
                                                FORK:
                                                    D=pop(F_R)
                                                endFork
                                                EXCEPTION:
                                                    D=pop(E_R)
                                                endException
                                            endCase
                                        finSi
                                    finsi
                                endSi
                            endSi
endTask
CHOICE:
    push(componenteActual,CHOICE,D)
        case D.marca
            0:
                push(RETORNOS,D)
                guardar(simbolos,-1)
                D.marca=1
            end0
            1:
                si D.id = ultimoChoiceUsado
                    hacer(FOR)
                endSi
                D.marca=2
                D=pop(RETORNOS)
            end1
            default:
                D=hallarComponenteNuevo(D)
            endCase
endChoice
FORK:
    push(componenteActual,FORK,D)
    guardar(simbolos,-4)
    case D.marca
        0:
            push(R_F,D)
            D.marca=1
        end0
        1:
            D=hallarComponenteNuevo(D)
        end1
    endCase
endFork
JOIN:
    si D.marca<D.entradas
        D.marca=D.marca+1
            si D.marca=D.entradas
                guardar(simbolos,-5)
                guardar(simbolos,D)
                guardar(simbolos,-5)
                pop(F_R)
            siNo
```

```
                    guardar(simbolos,-4)
                    guardar(simbolos,D)
                    guardar(simbolos,-4)
                    D=pop(F_R)
                    push(F_R,D)
                endsi
            endsi
    endJoin
    EXCEPTION:
        push(componenteActual,EXCEPTION,D)
        case D.marca
            0:
                push(E_R,D)
                guardar(simbolos,-6)
                D.marca=1
            end0
            1:
                D.marca=2
                guardar(simbolos,-7)
            end1:
            2:
                D=hallarComponenteNuevo(D)
            end2
        endCase
    endEXCEPTION
    END:
        si D.marca<D.entradas
            D.marca=D.marca+1
                si D.marca=D.entradas
                    guardar(simbolos,-9)
                    guardar(simbolos,D)
                    guardar(simbolos,-9)
                    si RETORNOS <> null || F_R <> null || E_R <> null
                        case componenteActual.tope:
                            CHOICE:
                                D=pop(RETORNOS)
                            endChoice
                            FORK:
                                D=pop(F_R)
                            endFork
                            EXCEPTION:
                                D=pop(E_R)
                            endException
                        endCase
                    endSi
                        siNo                    guardar(simbolos,-8)
                            guardar(simbolos,-D)
                            guardar(simbolos,-8)
                            case componenteActual.tope:
                                CHOICE:
                                    D=pop(RETORNOS)
                                endChoice
                                FORK:
                                    D=pop(F_R)
                                endFork
                                EXCEPTION:
                                    D=pop(E_R)
                                endException
                            endCase
                        endsi
                    endsi
                endEnd
FAILURED:
    si D.marca<D.entradas
        D.marca=D.marca+1
            si D.marca=D.entradas
```

```
                    guardar(simbolos,-11)
                    guardar(simbolos,D)
                    guardar(simbolos,-11)
                    si RETORNOS <> null || F_R <> null || E_R <> null
                        case componenteActual.tope:
                            CHOICE:
                                D=pop(RETORNOS)
                            endChoice
                            FORK:
                                D=pop(F_R)
                            endFork
                            EXCEPTION:
                                D=pop(E_R)
                            endException
                    endCase
                endSi
            siNo
                guardar(simbolos,-10)
                guardar(simbolos,-D)
                guardar(simbolos,-10)
                case componenteActual.tope:
                    CHOICE:
                        D=pop(RETORNOS)
                    endChoice
                    FORK:
                        D=pop(F_R)
                    endFork
                    EXCEPTION:
                        D=pop(E_R)
                    endException
                endCase
            endsi
        endsi
    endFailured
endCase
S=D
END
```

Apéndice B

Clase Parsear y Reducir

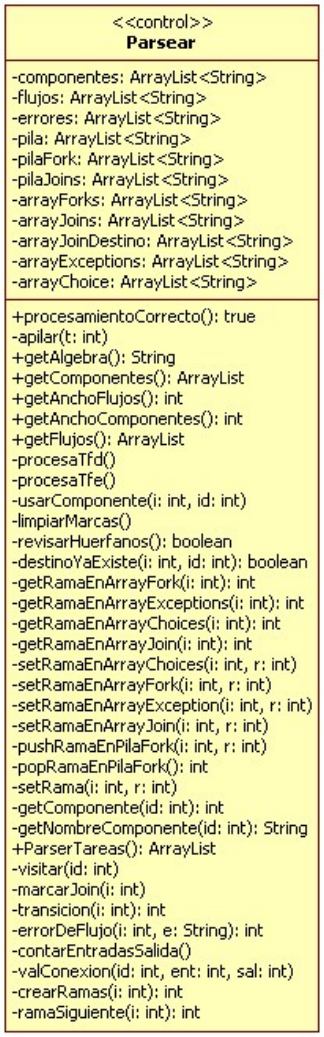


Figura B.1: Vista completa de la clase Parsear.

Figura B.2: Vista completa de la clase Reducir.

.

Apéndice C

Comportamiento de las clases Parsear y Reducir

C.1. Comportamiento de la clase Parsear

procesamientoCorrecto: Método que devuelve un valor lógico verdadero si no se detectaron errores.

apilar: Inserta un componente en la pila dado el parámetro de entrada correspondiente a un entero.

getAlgebra: Método que devuelve el valor del álgebra de tareas del diagrama correspondiente.

getComponentes: Devuelve la lista abstracta componentes.

getAnchoFlujos: Devuelve el valor de tipo entero correspondiente al ancho de la estructura abstracta flujos.

getAnchoComponentes: Devuelve el valor de tipo entero correspondiente al ancho de la estructura abstracta componentes.

getFlujos: Devuelve la lista abstracta flujos.

procesaTfd: Analiza el archivo correspondiente a la representación visual del diagrama de flujos de tareas.

procesaTfe: Analiza el archivo correspondiente a la representación lógica del diagrama de flujos de tareas.

usarComponente: A partir de dos valores de entrada de tipo entero elimina de la lista arraycomponente los componentes a los que hagan referencia los parámetros de entrada.

limpiarMarcas: Eliminar las marcas hechas en la estructura abstracta flujos.

revisarHuerfanos: Devuelve un valor lógico verdadero si es que se detecta que todos y cada uno de los componentes fueron usados.

destinoYaExiste: A partir de dos parámetros de entrada de tipo entero, verifica que un destino no haya sido usado, devuelve un valor verdadero si esta condición se cumple.

getRamaEnArrayFork: Devuelve un valor entero referente a una marca hecha sobre un com-

ponente *fork*, recibe como parámetro de entrada el valor índice de tipo entero.

getRamaEnArrayExceptions: Devuelve un valor entero referente a una marca hecha sobre un componente *exception*, recibe como parámetro de entrada el valor índice de tipo entero.

getRamaEnArrayChoices: Devuelve un valor entero referente a una marca hecha sobre un componente *choice*, recibe como parámetro de entrada el valor índice de tipo entero.

getRamaEnArrayJoin: Devuelve un valor entero referente a una marca hecha sobre un componente *join*, recibe como parámetro de entrada el valor índice de tipo entero.

setRamaEnArrayChoices: A partir de un índice y un valor que representa una rama, establece dicho valor sobre el componente *choice* identificado con el índice.

setRamaEnArrayFork: A partir de un índice y un valor que representa una rama, establece dicho valor sobre el componente *fork* identificado con el índice.

setRamaEnArrayException: A partir de un índice y un valor que representa una rama, establece dicho valor sobre el componente *exception* identificado con el índice.

setRamaEnArrayJoin: A partir de un índice y un valor que representa una rama, establece dicho valor sobre el componente *join* identificado con el índice.

pushRamaEnPilaFork: Dados dos parámetros, id y valor de rama respectivamente, apila sobre la lista pila *fork* dichos valores.

popRamaEnPilaFork: Devuelve el último valor correspondiente a la lista pila *fork*.

setRama: Dados los parámetros índice y rama, establece esos valores en la estructura abstracta flujos.

getComponente: Devuelve el valor del parámetro ID del componente.

getNombreComponente: Devuelve (según el valor del parámetro índice) el nombre de tipo *String* del componente.

ParserTareas: Método público el cual realiza el proceso de parsear y posteriormente llama crear un objeto del tipo reducir, devuelve en una lista los errores resultantes del proceso.

visitar: Establece el valor de un flujo como visitado, dicho flujo se representa por el parámetro de entrada de tipo entero índice.

marcarJoin: Establece el valor de un componente *join* como visitado, dicho componente se representa por el parámetro de entrada de tipo entero índice.

transicion: Devuelve el valor del componente destino a partir del id de un componente origen.

errorDeFlujo: Devuelve un error detectado en un flujo, esto por medio de dos parámetros de entrada, elemento de tipo *string* e índice de tipo entero.

contarEntradasSalida: Cuenta las entradas y salidas de todos y cada uno de los componentes del diagrama de flujos de tareas.

valConexion: Valida que el número de conexiones tanto de entradas como de salidas de un determinado componente sea el correcto. Recibe como parámetros de entrada de tipo enteros el id del componente así como su número de entradas y salidas permitidos. Además, este método utiliza la tabla de estados 3.12 para validar los flujos que se le presenten.

crearRamas: A partir de un origen crea un valor de rama, esto para saber donde inicia y termina el anidamiento de la estructura *fork-join*, devuelve un valor incrementado para el caso de un nuevo nivel de anidación.

ramaSiguiente: Devuelve el valor de rama en un componente dado el parámetro de entrada índice.

C.2. Comportamiento de la clase Reducir

inicializar: Método encargado de tener una copia de las estructuras abstractas correspondientes al diagrama de flujo de tareas para su posterior utilización dentro de esta misma. No devuelve ningún tipo de valor y recibe las estructuras a copiar.

seguirReduciendo: Devuelve el valor booleano correspondiente al atributo continuarReduciendo, esto para buscar estructuras del álgebra de tareas. No recibe ningún parámetro.

setSeguirReduciendo: Modifica por medio de un parámetro recibido del tipo booleano el valor de la variable continuarReduciendo. No devuelve ningún valor.

setAnchoComponentes: Establece el valor a partir de un parámetro de entrada de tipo entero el ancho de la estructura abstracta componentes, no devuelve ningún valor.

getAnchoComponentes: Devuelve el valor correspondiente al ancho de las estructura abstracta componentes, no recibe ningún valor de entrada.

setAnchoFlujos: Establece un valor a partir de un parámetro de entrada de tipo entero, el ancho de la estructura abstracta flujos, no devuelve ningún valor.

setAnchoFR: Establece el valor del ancho para fr a partir de un valor del tipo entero, no devuelve ningún tipo de valor.

getAnchoFlujos: Devuelve el valor del ancho de la estructura abstracta flujos mediante un valor del tipo entero, no recibe parámetros.

getAnchoFR: Devuelve el valor del ancho de la estructura abstracta FR mediante un valor del tipo entero, no recibe parámetros.

copiarFlujos: Realiza un copia de la estructura abstracta flujos a partir de la lista original y el ancho de la misma estructura con los parámetros de tipo ArrayList y entero.

copiarComponentes: Realiza una copia de la estructura abstracta componentes a partir de la lista original con el parámetro de tipo entero.

limpiarMarcas: Limpia las marcas establecidas en la estructura abstracta original, no recibe ni devuelve ningún parámetro.

crearFR: Crea una lista dinámica del tipo ArrayList a partir de un parámetro del mismo tipo.

getEntradasComponentes: A partir de un índice de tipo entero devuelve el número de entradas de ese índice en la estructura abstracta componentes

getSalidasComponentes: A partir de un índice de tipo entero devuelve el número de salidas de ese índice en la estructura abstracta componentes

esSecuenciaLinealDeTareas: Por medio de dos parámetros de tipo entero los cuales represen-

tan un origen y un destino en los componentes del diagrama, devuelve un valor lógico en caso de que se trate de una secuencia lineal de tareas, es decir, que sólo haya un flujo de entrada y salida para ambos componentes.

esSecuenciaLinealTerminalDeTareas: Por medio de un parámetro de tipo entero, devuelve un valor lógico en caso de que se trate de una secuencia donde el destino sea un componente terminal.

getTipoOrigenFR: A partir de un parámetro de tipo entero que representa el índice dentro de estructura abstracta fr, devuelve el valor correspondiente al tipo de origen.

getTipoDestinoFR: A partir de un parámetro de tipo entero que representa el índice dentro de la estructura abstracta fr, devuelve el valor correspondiente al tipo de destino.

getIDOrigenFR: A partir de un parámetro de tipo entero que representa el índice dentro de la estructura abstracta fr, devuelve el valor correspondiente al ID de origen.

getEntradasFR: Devuelve según el índice recibido la cantidad de entradas en la estructura abstracta fr.

getSalidasFR: Devuelve según el índice recibido la cantidad de salidas en la estructura abstracta fr.

getIDDestinoFR: A partir de un parámetro de tipo entero que representa el índice dentro de estructura abstracta fr, devuelve el valor correspondiente al ID de destino.

setIDDestinoFR: Método para cambiar un ID destino en la estructura abstracta fr, recibe dos parámetros, el índice y el valor nuevo para el ID, ambos de tipo entero.

setDatosFR: Método para cambiar el valor de los datos en la estructura abstracta fr, recibe dos parámetros, el índice de tipo entero y el valor nuevo de tipo *String* para los datos.

eliminarRenglonFR: A partir del valor índice de tipo entero elimina un renglón de la estructura abstracta fr.

corrimientoFR: Efectúa el corrimiento de un valor numérico a su correspondiente índice dentro de la estructura abstracta fr.

reducirTareasLineales: Dadas dos tareas lineas las reduce para obtener una sola tarea lineal.

unirTareaLinealConTerminal: A partir de un índice dado de tipo entero, une una tarea lineal con un componente terminal.

cambiarTipoDestino: A partir de un valor índice de tipo entero modifica el tipo correspondiente a ese índice.

crearFOR: A partir de dos valores de tipo entero representando índices, modifica la estructura abstracta para crear un *for* y eliminar las tareas que lo componen.

buscarElementoEnFlujo: A partir de un parámetro de entrada de tipo entero verifica si este existe en la estructura abstracta, devuelve un valor lógico según el resultado del proceso.

existeUnFor: Mediante el ID de un componente *choice*, verifica si puede formarse una estructura *for*, devuelve un valor lógico según el resultado del proceso.

getNodoDatosFOR: Devuelve los datos contenidos en una estructura *for*.

reducirFor: Método que evalúa cada uno de los componentes *choice* para verificar si alguno puede crear un *for.*

buscarSiguienteOrigen: Devuelve el siguiente valor correspondiente a un id origen.

buscarSiguienteDestino: Devuelve el siguiente valor correspondiente a un id destino

existeUnOrSimple: A partir del índice para un componente *choice* se busca que se pueda formar un or simple.

crearORSimple: Crear un or simple previamente identificado.

existeUnOrTerminalDoble: A partir del índice para un componente *choice* se busca que se pueda formar un or con terminal doble.

crearORDeTerminalDoble: A partir del índice crea un or con terminal doble previamente identificado.

existeUnOrTerminalSimple: A partir del índice para un componente *choice* se busca que se pueda formar un or con terminal simple.

crearORTerminalSimple: A partir del índice crea un or con terminal simple previamente identificado.

setContenidoTarea: Devuelve en un *String* el contenido de una tarea dado un índice.

getContenidoAnidado: Devuelve en un *String* el contenido anidado de una estructura dado un índice.

existeUnOrTerminalSimpleVacio: A partir del índice para un componente *choice* se busca que se pueda formar un or con terminal simple vacío.

crearUnOrTerminalSimpleVacio: Crear un or de terminal simple vacío previamente identificado sobre un paŕametro entero que representa un índice

existeUnOrVacioDoble: A partir del índice para un componente *choice* se busca que se pueda formar un or vacío doble.

crearUnOrVacioDoble: Crear un or vacío doble previamente identificado sobre un paŕametro entero que representa un índice

reducirOr: Método que busca sobre los componentes *choice* la posible formación de alguna estructura or.

existeUnUntil: Mediante el ID de un componente *choice*, verifica si puede formarse una estructura *until*, devuelve un valor lógico según el resultado del proceso.

crearUNTIL: A partir de dos valores de tipo entero representando índices, modifica la estructura abstracta para crear un *until* y eliminar las tareas que lo componen.

reducirUntil: Método que busca sobre los componentes *choice* la posible formación de la estructura *until*.

existeUnOrVacioSimple: A partir del índice para un componente choice busca que se pueda formar un or vacío simple.

existeParalelismo: A partir del índice para un componente *fork* se busca la existencia de tareas paralelas.

crearFORK: Previa identificación de tareas paralelas, este método las agrupa en un sólo componente para crear una secuencia lineal.

reducirParalelismo: Método que busca sobre los componentes *fork y join* la posible formación de estructuras paralelas.

algebraCorrecta: Devuelve un valor lógico verdadero si la estructura final corresponde a un inicio y un componente terminal.

getTarea: Método que a partir de un índice obtiene el valor dado por el usuario sobre la tarea correspondiente a un diagrama.

generarAlgebra: Método que llama a todos los demás métodos encargados de buscar la posible formación de estructuras del álgebra de tareas.

getAlgebra: Devuelve el álgebra de tareas correspondiente a un diagrama de flujo de tareas.

getErrores: Método que devuelve los errores correspondiente al diagrama analizado.

Apéndice D

Modelo de definición gráfica de GMF

Configuración de las componentes presentada en la sección 4.2 llevada a cabo en el modelo de definición gráfica de GMF.

D.1. Componentes Fork y Join

La configuración inicial del elemento *Fork* se muestra en la imagen D.1, en donde se observa su estructura sin personalización, así como la del componente *Join*.

Entre los cambios a llevar a cabo tenemos los siguientes:

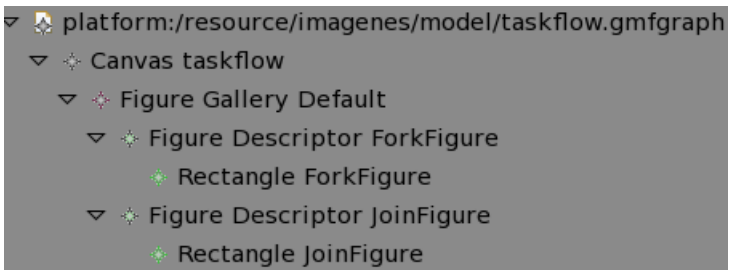


Figura D.1: Estructura del componente *Fork* y *Join* antes de su configuración.

- *Foreground*: Negro.
- *Background*: Blanco.
- *MaximunSize*: 60 por 10.
- *MinimunSize*: 60 por 10.
- *PreferredSize*: 60 por 10.
- *Size*: 60 por 10.

Este componente al igual que el componente *Join* comparten la misma configuración, ya que tienen la misma representación visual, sin embargo, el significado semántico es totalmente diferente.

Figura D.2: Estructura de los componentes *Fork* y *Join* después de su configuración.

Ambos componentes requieren de una configuración de comportamiento, en este caso para responder al momento de modificar su tamaño en el área de trabajo del editor de diagramas, para ello bastará con modificar la configuración del *NODE*. A la propiedad *Resize Constraint* se le asigna el valor *EAST_WEST*.

Con la configuración realizada, los componentes *Fork* y *Join* tendrán una representación similar dentro del diagrama de flujos de tareas.

D.2. Componente End

Como se vió en la sección 3.3.1, el componente *End* debe tener una representación visual similar a la mostrada en la imagen de la figura 3.3. Sin embargo, la visualización por defecto que asigna GMF es un rectángulo, esta se muestra en la figura D.3, por tal motivo se debe configurar para que tenga una representación similar a la esperada, los cambios realizados sobre *XYLayout* fueron:

- *Foreground*: Negro.
- *Background*: Blanco.
- *MaximunSize*: 20 por 20.
- *MinimunSize*: 20 por 20.
- *PreferredSize*: 20 por 20.
- *Size*: 20 por 20.

Además, para este componente en específico se requiere de un círculo en su interior, por ello se inserta otro componente del tipo *Ellipse EndFigureCenter*, es decir, tendrá una imagen anidada.

Su configuración se muestra a continuación:

- *XYLayoutData*: Posición 4, 4. Largo 12 y alto 12.
- *Background*: Negro.

El comportamiento dado por la configuración *NODE* no debe permitir cambios de tamaño, por tal motivo la propiedad *Resize Constraint* se establece a "*none*".

La figura D.4 muestra el componente *End* después de su configuración.

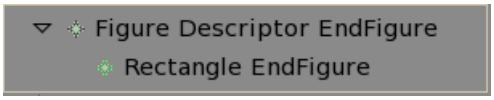


Figura D.3: Estructura del componente *End* antes de su configuración.

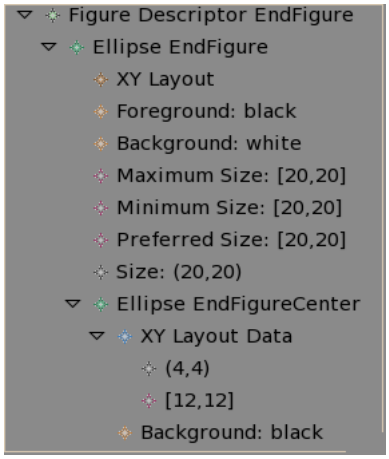


Figura D.4: Estructura del componente *End* después de su configuración.

D.3. Componente Start

El elemento *Start* fue el más simple de configurar, ya que únicamente requiere de un círculo compuesto de un fondo y borde negro. La configuración por defecto se muestra en la figura D.5 y la configuración personalizada se observa en la figura D.6. Los cambios realizados se describen a continuación:

- *Ellipse: StartFigure*.
- *Foreground*: Negro.
- *Background*: Negro.
- *MaximunSize*: 20 por 20.
- *MinimunSize*: 20 por 20.
- *PreferredSize*: 20 por 20.
- *Size*: 20 por 20.

El comportamiento dado por la configuración *NODE* al igual que *End*, debe ser inmutable, por tal motivo la propiedad *Resize Constraint* se establece a "*none*".

Figura D.5: Estructura del componente *Start* antes de su configuración.

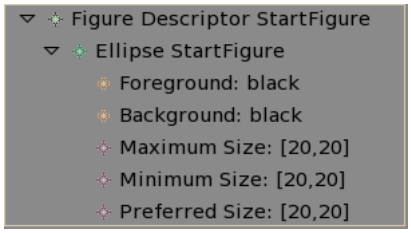


Figura D.6: Estructura del componente *Start* después de su configuración.

D.4. Componente Failure

Failure al igual que el componente *End* requieren de un subcomponente en su interior, la diferencia radica en que mientras para *End* es un círculo, para *Failure* se necesita de una forma irregular, es decir la figura en su interior es de un rectángulo rotado aproximadamente 60º, tal como se observa en la figura 3.3. La configuración por defecto se muestra en la figura D.7. Para ello se realizaron las siguientes modificaciones a la configuración del componente:

- *Ellipse FailureFigure*.
- *XY: Layout*.
- *Foreground*: Negro.
- *Background*: Blanco.
- *MaximunSize*: 20 por 20.
- *MinimunSize*: 18 por 18.
- *PreferredSize*: 19 por 19.

Así mismo la configuración para el componente interno es la siguiente.

- *Polygon*: cruzfigure.
- *XYLayoutData*: posición 0, 0. Largo y alto 20 por 20.
- *Background*: Negro.
- Puntos: pares ordenados para definir el polígono 15,2; 16,3; 3,16; 2,15; 15,2.

El comportamiento dado por la configuración *NODE* debe ser inmutable, por tal motivo la propiedad *Resize Constraint* se establece a "*none*".

El componente ya correctamente configurado se presenta en la figura D.8.

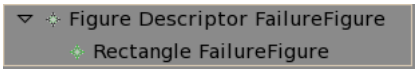


Figura D.7: Estructura del componente *Failure* antes de su configuración.

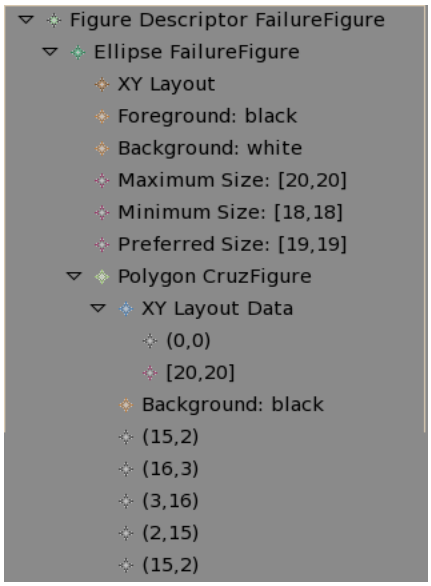


Figura D.8: Estructura del componente *Failure* después de su configuración.

D.5. Componente Exception

Al igual que el componente *Choice* se requiere de un polígono como base para generar correctamente el componente *Exception*, la figura D.9 muestra la configuración por defecto, los cambios hechos a la configuración original son los siguientes:

- *Scalable: Polygon*.
- *Stack: Layout*.
- *Foreground*: negro.
- *Background*: blanco.
- Puntos: pares ordenados para definir al polígono 0,0; 10,10; 0,20; 0,0.

Al igual que los componentes anteriores la configuración *NODE* en su propiedad *Resize Constraint* se establece a "*none*"; de la misma manera que las etiquetasChoiceConv yChoiceConF del componente *Choice*. Se le asigna en su configuración *Label* el valor *WEST* para la propiedad *Resize Constraint*.

La imagen de la figura D.10 muestra la configuración final para el componente.

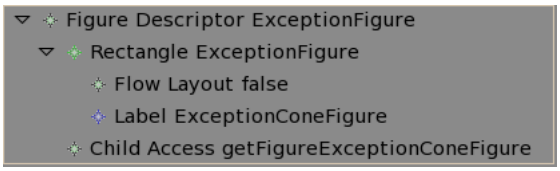


Figura D.9: Estructura del componente *Choice* antes de su configuración.

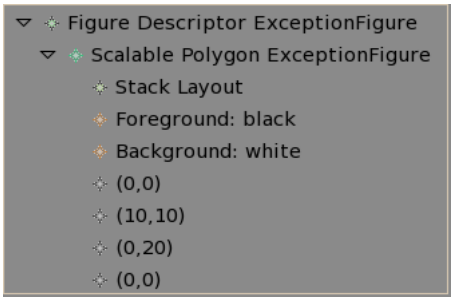


Figura D.10: Estructura del componente *Choice* después de su configuración.

Apéndice E

Manual de instalación

Este manual contiene la información necesaria para realizar la instalación en la plataforma *Windows* del IDE Eclipse con el plug-in para el Modelado de Diagramas de Flujo de Tareas, así como los requerimientos de *hardware* necesarios para su correcto funcionamiento.

E.1. Software requerido

Para que el IDE Eclipse pueda funcionar es necesario contar con máquina virtual de Java. Para instalar la máquina virtual de Java, es necesario descargarla del sitio web de Oracle `java.com/es/download/index.jsp`, en esa página seleccionaremos el paquete adecuado a nuestra plataforma.

El instalador ejecutándose se muestra en la figura E.1

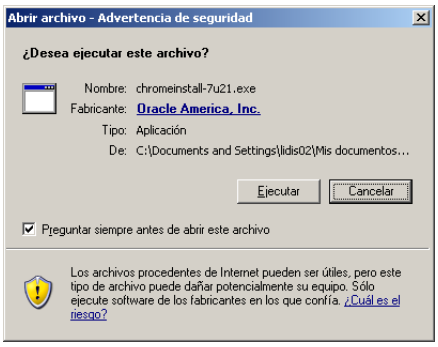


Figura E.1: Inicio de la instalación.

El proceso de instalación de la máquina virtual de Java finaliza cuando se muestra en la pantalla un mensaje de instalación exitosa.

Si al momento de iniciar Eclipse se muestra el mensaje de la figura E.2, será necesario modificar la variable de entorno *Path*. A continuación se describe el procedimiento a seguir:

1. Damos clic derecho sobre “Equipo” o “Mi PC” y seleccionamos propiedades.
2. Damos clic sobre la opción “opciones avanzadas” o configuración avanzada del sistema.

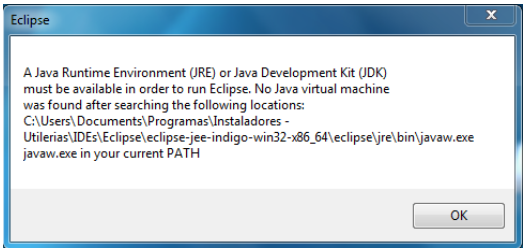


Figura E.2: Mensaje mostrado cuando no se encuentra la ruta a la máquina virtual de Java.

3. Damos clic sobre la opción “Variables de entorno” y la ventana de la figura E.3.
4. En el apartado Variables del sistema se localiza la variable Path y se da clic en la opción editar.
5. En las opciones de configuración para Path se agrega la ruta (sin modificar el contenido actual) hasta la carpeta “bin”. Por ejemplo: ...;C:\Program Files\Java\jdk1.7\bin;...

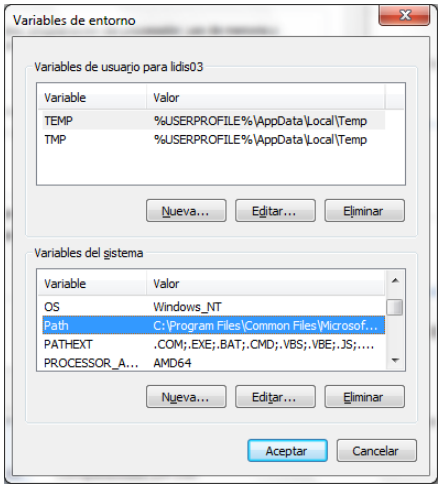


Figura E.3: Variables de entorno del sistema *Windows*.

E.2. Instalación de Eclipse

Para instalar el IDE Eclipse con el plug-in para el Modelado de Diagramas de Flujo de Tareas será necesario descargarlo de la siguiente ruta `http://www.utm.mx/~taskmodel/projects/tskmdlEclipse/eclipse.rar`, descomprimirlo y dar clic en el archivo eclipse.exe.

Apéndice F

Manual de usuario

El presente manual contiene a información necesaria para utilizar de forma adecuada el plug-in para el Modelado de Diagramas de Flujo de Tareas (pMDFT)

El manual le guiará paso a paso a través de las funcionalidades que ofrece el plug-in y le ofrecerá un ejemplo gráfico.

Cabe resaltar que este documento está dirigido a lectores que hayan instalado y configurado correctamente la máquina virtual de Java, así como la distribución del IDE con pMDFT, por lo que se omiten características técnicas fuera del uso del plug-in.

F.1. Información general

El IDE con pMDFT es un software que ofrece a los usuarios la posibilidad de modelar sistemas de negocios con la creación de diagramas de flujo de tareas, además de mostrar el significado de esos diagramas dado por el álgebra de tareas.

El usuario podrá interactuar de manera directa con los componentes del diagrama de flujo de tareas que se ubican en la barra de herramientas, de esta manera podrá formar el diagrama que desee. Además de obtener el álgebra de tareas con la opción "generar álgebra de tareas" también podrá llevar el diagrama de flujo de tareas a un formato XMI estándar.

El software está dirigido a usuarios que deseen realizar el proceso de verificación formal de modelos bajo el Método *Discovery*.

F.2. Interfaz del sistema

La interfaz del pMDFT se divide en 6 partes fundamentales:

1. Explorador de proyectos.
2. Área de diseño.
3. Barra de componentes.
4. Área de notificaciones.
5. Botones de acción.

6. Menú de acción.

Cada parte dentro del sistema consta de una funcionalidad diferente. Más adelante se describe por separado cada una de estas funcionalidades. En la figura F.1 se muestra la distribución de las partes de la interfaz.

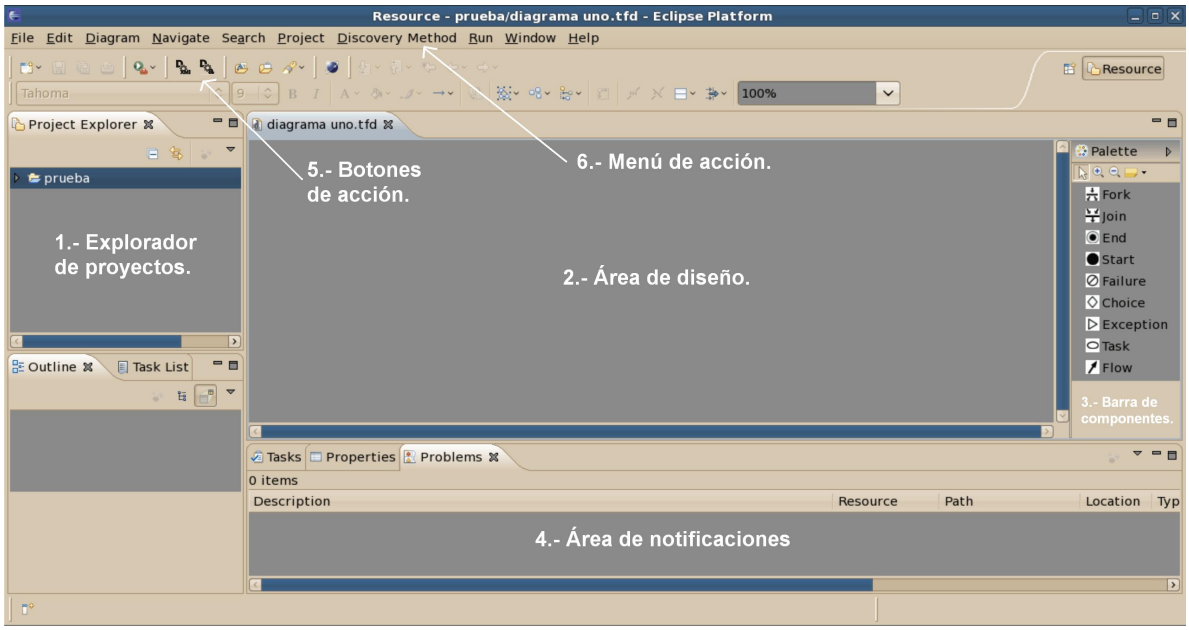


Figura F.1: Distribución de las partes de la interfaz de usuario.

F.2.1. Explorador de proyectos

En este apartado del pMDFT se muestran los proyectos del espacio de trabajo (ya sea en uso o cerrados). Un proyecto es un conjunto de carpetas o archivos (no es obligatorio que contenga ambos, incluso puede tener cualquier cantidad de cada uno de ellos) que tienen alguna funcionalidad relacionada. Esta área se utiliza para navegar por los proyectos en uso y tener acceso a los archivos que contienen.

F.2.2. Área de diseño

Es el área principal del plug-in, ya que en ella se colocarán los componentes seleccionados de la barra de componentes hasta formar un diagrama de flujo de tareas

F.2.3. Barra de componentes

Esta barra contiene los 8 diferentes componentes para la construcción de diagramas de flujo de tareas. En ella se muestra el componente visual y el nombre dado por el Método *Discovery*.

F.2.4. Área de notificaciones

Área encargada de mostrar los errores que se presenten al momento de generar el álgebra de tareas. En esta área también se mostrarán las propiedades de los componentes que integran el diagrama.

F.2.5. Botones de acción

Los botones de acción llevan a cabo una tarea específica. El botón “generar álgebra” (figura F.2.a) inicia el proceso para obtener el álgebra de tareas del diagrama activo en el área de diseño. El botón “exportar a XMI” (figura F.2.b) inicia el proceso para exportar el diagrama activo al formato XMI.

Figura F.2: Botones de acción generar álgebra y exportar a XMI.

F.2.6. Menú de acción

Contiene las opciones “generar álgebra” y “exportar a formato XMI”, ambas tienen el mismo funcionamiento que los botones de acción respectivos.

F.3. Funciones del sistema

El sistema pMDFT brinda el soporte para diseñar diagramas de flujo de tareas así como las funcionalidades para generar el álgebra de tareas y exportar a formato XMI. A continuación se detallará el proceso para diseñar un diagrama y obtener el álgebra de tareas correspondiente.

F.3.1. Crear un proyecto

Para esto damos clic en el menú File->New->Project, dentro de las opciones mostradas en el cuadro de diálogo *New Project* (figura F.3), se seleccioná la opción *General - Project*.

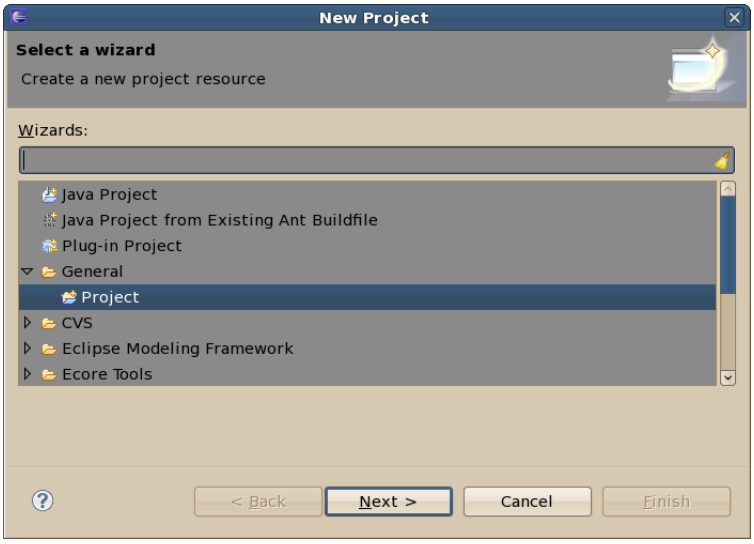


Figura F.3: Cuadro de dialogo crear proyecto.

Finalmente se da clic en el botón *next*, en la pantalla que se despliega (figura F.4), se debe dar un nombre válido al proyecto. Según la configuración del manejo de versiones de Eclipse, este puede responder conectando el proyecto recién creado al manejador, en este caso se deja al usuario la decisión de utilizar esa funcionalidad de Eclipse.

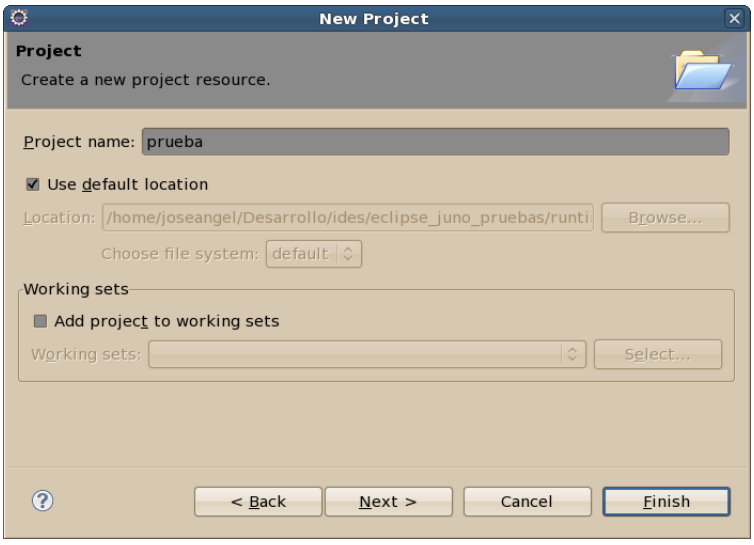


Figura F.4: Cuadro de dialogo para asignar nombre al nuevo proyecto .

El aspecto final del IDE con el proyecto creado se muestra en la figura F.5.

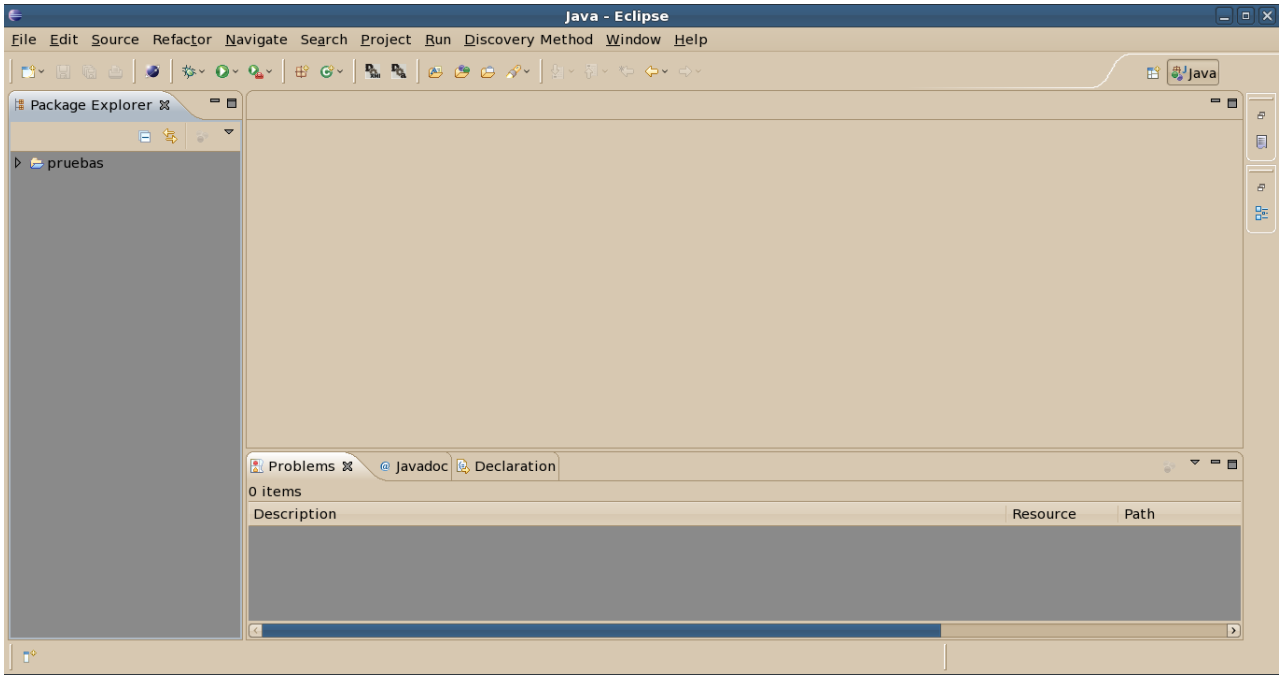


Figura F.5: Vista del IDE con el proyecto creado.

Sobre este proyecto se mostrará como crear un diagrama de flujo de tareas.

F.3.2. Crear un diagrama

Para crear un nuevo diagrama de flujo de tareas, es necesario dar clic derecho sobre el proyecto en el cual se guardará dicho diagrama (en este caso se toma el proyecto creado en la sección anterior). Sobre el menú contextual que se despliega se debe seleccionar la siguiente ruta *New-Other*, está acción iniciará el cuadro de dialogo *New* mostrado en la figura F.6.

En el cuadro de dialogo que se abre se navega hasta la lista "*Examples*", la primera opción desplegada será "*Taskflow Diagram*", en caso contrario, se deberá seleccionar esa opción.

La pantalla mostrada en la figura F.7, nos brinda la posibilidad de guardar el diagrama de flujo de tareas a crear en el proyecto prueba – si se requiere guardar el diagrama en algún otro proyecto existente esta es la pantalla indicada para ello –, el nombre que se le asigne debe contener la extensión tfd, en caso contrario no se activa la opción siguiente.

Es de suma importancia dar clic en la opción *Next*, ya que si este paso se omite, el nombre del archivo ecore (tfe) no será el mismo que el asignado al archivo visual (tfd). Por último al dar clic en el botón *Finish* se abre el área de trabajo para el diseño de diagramas de flujo de tareas.

La figura F.8 muestra el área de trabajo con el proyecto prueba recién creado.

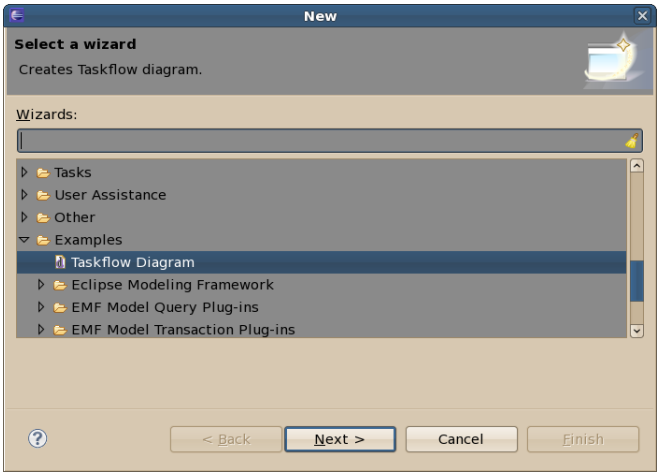


Figura F.6: Cuadro de dialogo crear proyecto.

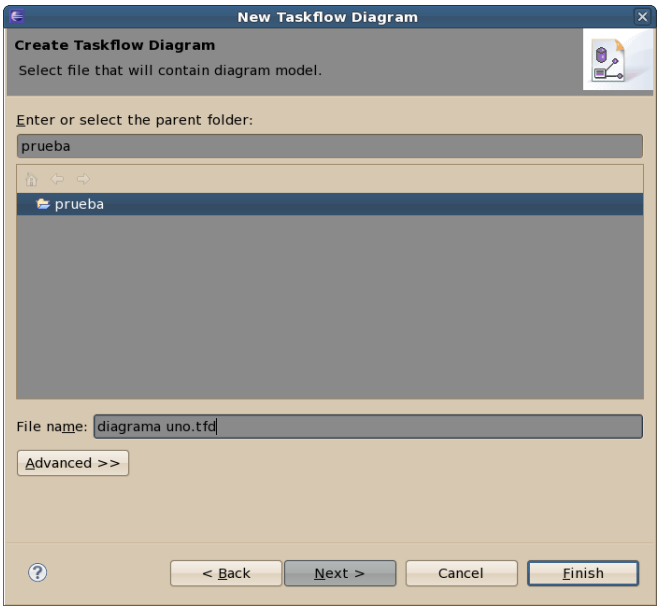


Figura F.7: Cuadro de dialogo para asignación de proyecto y nombre.

Es en esta área de trabajo donde se diseñarán los diagramas de flujo de tareas. Para ello se requiere dar clic sobre algún componente de los disponibles en la barra de componentes, al hacerlo el apuntador será sombreado por un rectángulo. Para colocar el componente en el diagrama se deberá dar clic izquierdo en el área de diseño preferida.

Para unir dos componentes se debe llevar a cabo el siguiente proceso:

- Después de hacer clic sobre el componente *Flow,* el cursor se situará sobre el componente origen y se dará clic con el botón izquierdo sin soltar, se buscará otro componente, al soltar el botón izquierdo sobre un componente diferente al origen este será considerado como el destino del flujo.

Este proceso se debe repetir hasta obtener el diagrama deseado.

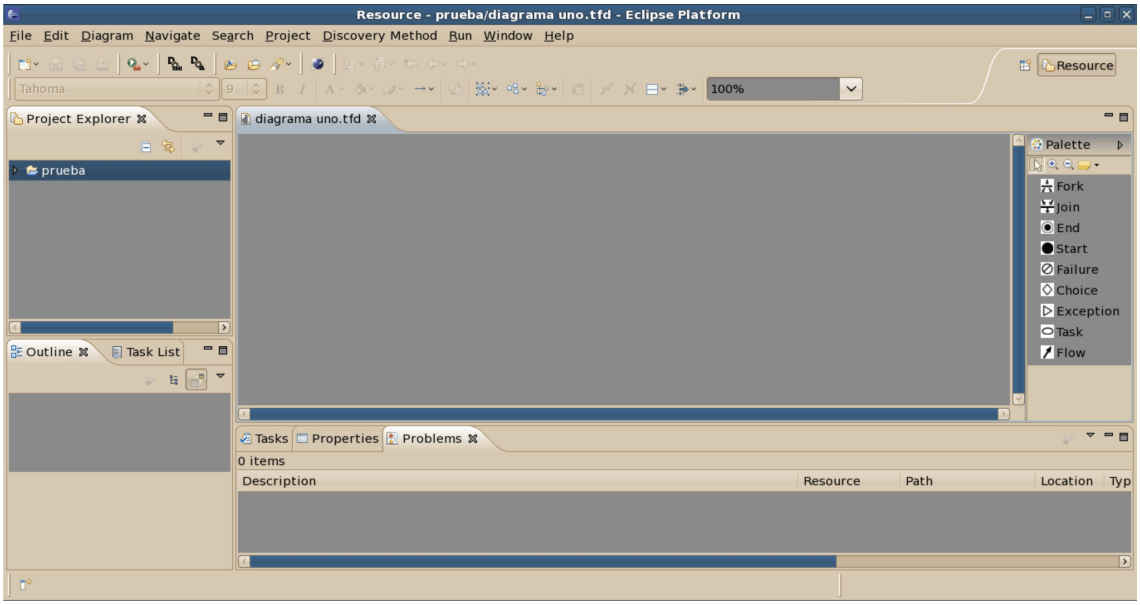


Figura F.8: Aspecto del área de trabajo previo al diseño del diagrama de flujo de tareas.

F.3.3. Generar el álgebra de tareas

Una vez que se tiene creado un diagrama de flujo de tareas (como el mostrado en la figura F.9), bastará con dar clic en el botón generar álgebra, es importante señalar que el diagrama sobre el que se desea generar el álgebra debe estar abierto y en primer plano, si no se encuentra en primer plano, no iniciará con el proceso de reducción.

Si el usuario no guardó los cambios realizados se mostrará el mensaje de advertencia de la figura F.10, tendrá la opción de guardar los cambios para que estos se reflejen en el proceso de generación del álgebra, en caso de no aceptar el álgebra corresponderá al último cambio guardado.

En caso de que el diagrama sea correcto se abrirá y ocupara el primer plano un archivo con extensión tfa (figuraF.11) siendo este el que contiene el álgebra asociada al diagrama.

F.4. Exportar a formato XMI

Para exportar un diagrama de flujo de tareas es necesario tener un diagrama abierto y en primer plano, posteriormente dar clic en el botón “Exportar a XMI”, con ello se abrirá el cuadro de diálogo de la figura F.12.

En el cuadro de diálogo “Guardar” se introducirá una ruta y un nombre para el archivo XMI que se creará.

Finalmente, cuando el archivo se guarda de manera exitosa se desplegará un cuadro emergente mostrando la leyenda “El archivo se exportó con éxito”. En caso contrario el cuadro emergente muestra el mensaje “Se produjo un error, el archivo no pudo ser exportado”.

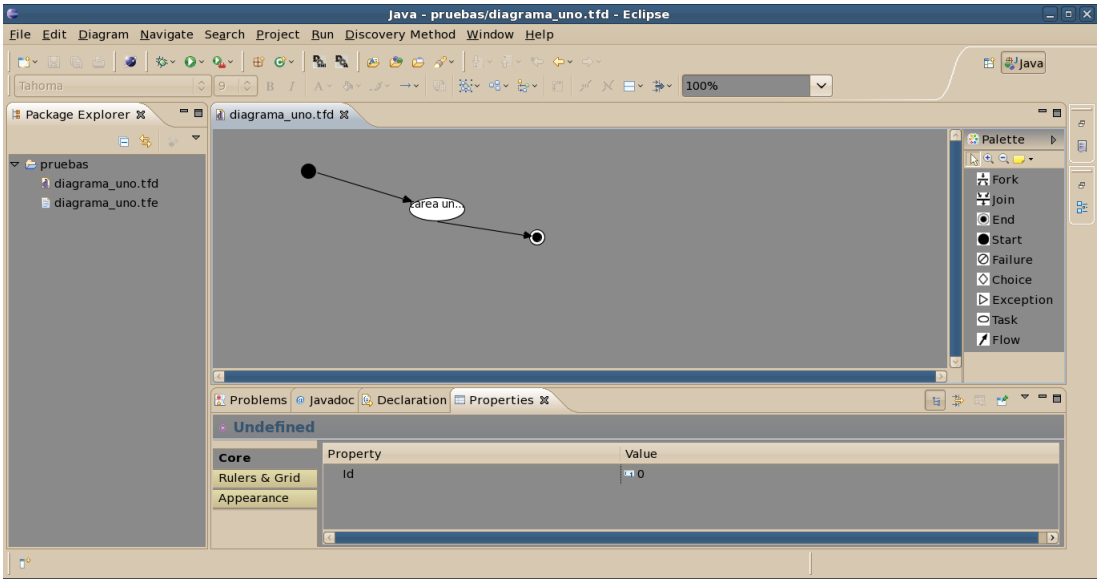


Figura F.9: Diagrama de flujo de tareas ejemplo.

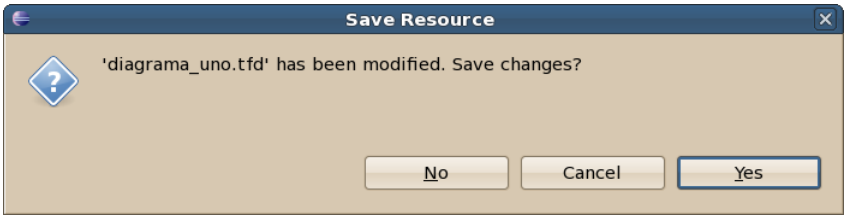


Figura F.10: Cuadro de dialogo para guardar cambios antes de generar el álgebra de tareas.

Figura F.12: Cuadro de dialogo “Guardar”, para exportar a formato XMI.

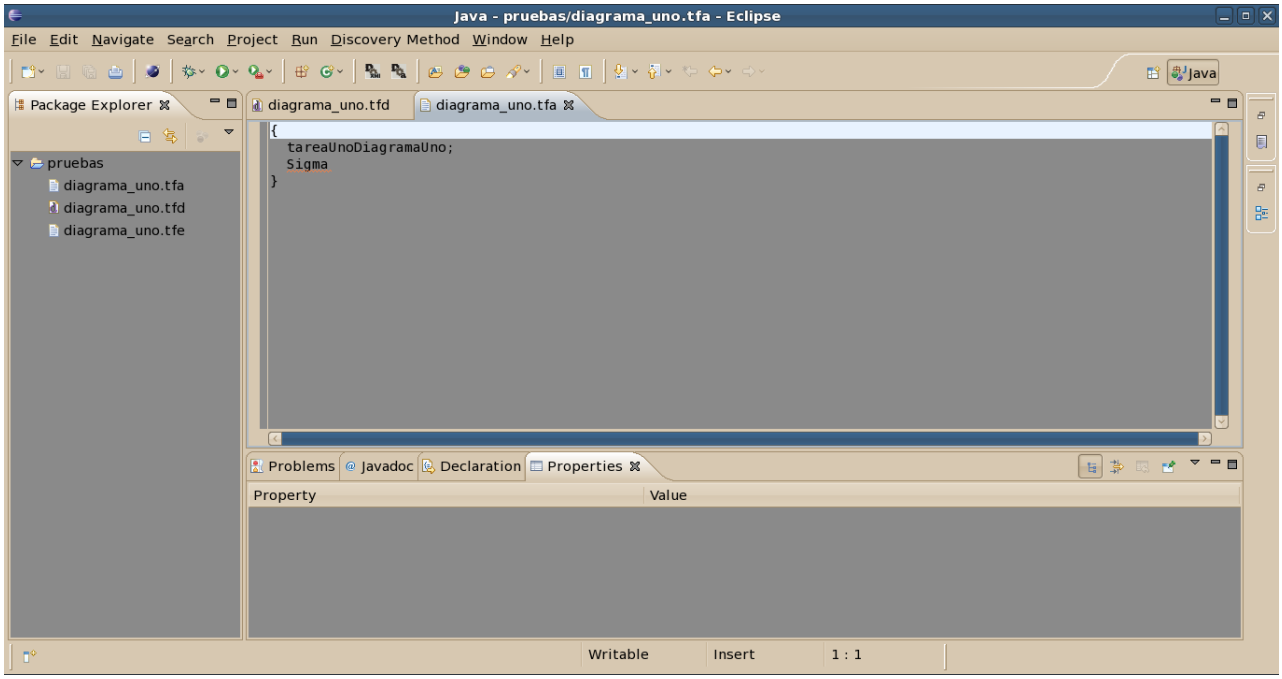


Figura F.11: Álgebra de tareas correspondiente al diagrama de flujo de tareas de ejemplo.

F.5. Problemas al generar el álgebra de tareas

En caso de que el diagrama de flujo de tareas presente errores, el área de notificaciones (vista denominada *Problems)* será la zona donde se mostrará la información al usuario. Cuando esto suceda, se activará el cuadro de información de la figura F.13.

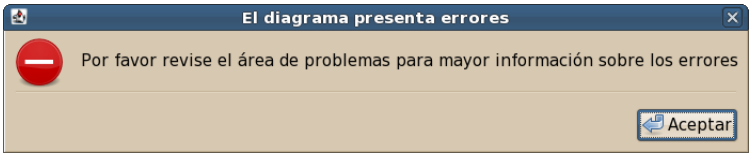


Figura F.13: Cuadro de información para advertir de la existencia de errores.

Los errores que pueden generarse al momento de diseñar diagramas son mostrados a continuación.

- Componente huérfano: Componente que alguno de sus flujos no está conectado a otro componente exceptuando *End* y *Failure*, o bien ningún componente se conecta a él, para este último caso se exceptúa *Start*. La figura F.14 muestra un ejemplo de la vista de problemas con este error.
- Componente *Task* vacío: El componente *Task* no tiene asignada ninguna tarea. Figura F.15.
- Fuera de ámbito: El flujo de un componente sale o entra a un ámbito *Fork/Join* que no le corresponde. Figura F.16.

- Error lógico: Cuando no se reconoce una estructura y por lo tanto el diagrama no puede ser reducido se muestran los flujos hasta la última reducción posible. Es de resaltar que debido a ello algunos componentes toman el nombre de “temporal:ID”, donde ID es un identificador numérico único para cada componente.

En caso de que alguno de estos errores se presenten, la vista de problemas mostrará información referente a los componentes involucrados.

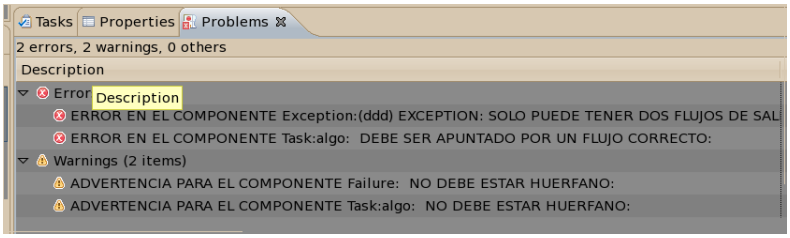


Figura F.14: Vista problemas cuando un componente presenta errores en sus flujos.

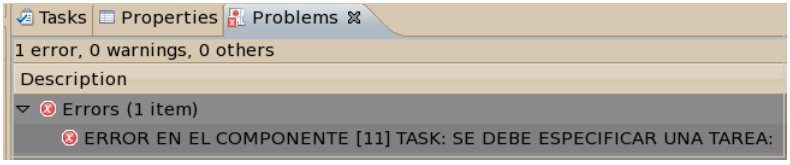


Figura F.15: Vista problemas cuando un componente Task se encuentra vacío.

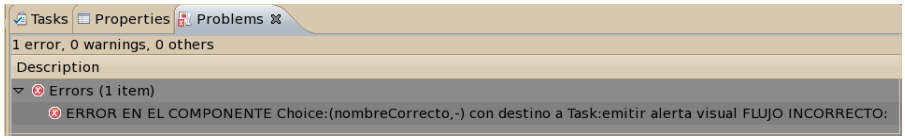


Figura F.16: Vista problemas cuando se presenta un error de ámbito.