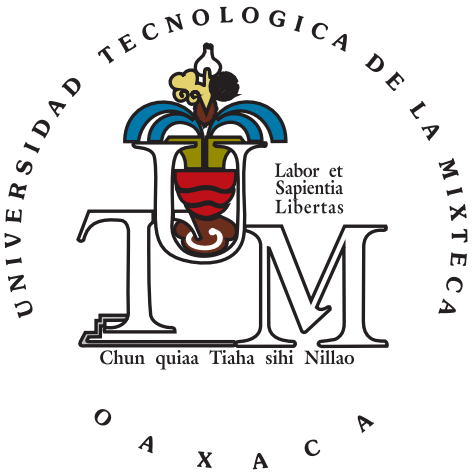

UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

Desarrollo de una interfaz para la verificación de diagramas de tareas en la especificación del software

TESIS PARA OBTENER EL TÍTULO DE:
INGENIERO EN COMPUTACIÓN

PRESENTA:

Hermenegildo Fernández Santos

DIRECTOR DE TESIS:

Dr. CARLOS ALBERTO FERNÁNDEZ Y FERNÁNDEZ

Huajuapan de Leoń, Oaxaca, Septiembre 2013

Dedicatoria

A Mis Padres

...por su confianza infinita, sus consejos y todas las oportunidades que me han brindado para ser alguien en la vida.

A Mis Hermanos

...por su confianza, apoyo y fraternidad en momentos los momentos importantes de mi vida.

A Mi Profesora y Amiga María Fernanda Puentes Rodríguez

...por su amistad y motivación a lo largo de este viaje.

Agradecimientos

Agradezco a todos mis profesores de la Universidad Tecnológica de la Mixteca por sus conocimientos, experiencias y ejemplos de profesionalidad que me servirán para desempeñar orgullosamente mi carrera.

A mi asesor Dr. Carlos Alberto Fernández y Fernández por su confianza al permitirme realizar el proyecto, su dedicación y enorme paciencia.

De igual manera le doy las gracias a mis sinodales M.C. Mario Alberto Moreno Rocha, M.C. David Martínez Torres, Dr. Moisés Homero Sánchez López y al M.A.E. Rodolfo Maximiliano Valdés Dorado por su tiempo, dedicación y su valiosa contribución al revisar este trabajo.

A mis amigos con quienes he compartido el proyecto; sus experiencias y enseñanzas me han sido de gran ayuda.

Y a todos aquellos que hicieron posible la elaboración de este trabajo.

Muchas Gracias.

Resumen

La construcción y verificación de modelos debería ser un requerimiento obligatorio para garantizar la consistencia de los sistemas en el proceso de desarrollo de software, sin embargo, en muchos casos el tiempo y la falta de experiencia impiden llevar a cabo esta actividad. Aún con estas limitantes el proceso de verificación de propiedades requiere además una base libre de ambigüedades y actualmente la notación estándar del Lenguaje de Modelado Unificado (UML) no cumple este requisito.

Una de las alternativas a las ambigüedades e inconvenientes UML es el Método Discovery. Este método se define como una aproximación al análisis y diseño de sistemas, el cual adopta un perfil UML restringido, centrándose en un minimalismo y consistencia [32]. En el Método Discovery, el flujo de actividades se representa utilizando Diagramas de Tareas.

Los diagramas de tareas pueden traducirse fácilmente a una especificación algebraica abstracta, en la cual la verificación de propiedades resulta sencilla gracias a una herramienta desarrollada por [9], la desventaja de esta herramienta es que el proceso de construcción de expresiones y análisis de resultados es complicado debido a que funciona a nivel consola.

Ante la situación anterior, surge la necesidades de modificar el escenario de construcción de expresiones de verificación de interfaz de comandos a un concepto visual (interfaz), transparente y que se integre al proceso de desarrollo de software.

Para detallar el desarrollo de este proyecto, el presente documento se encuentra organizado de la siguiente manera: el capítulo 1 expone de forma más precisa el problema a resolver, la solución y los objetivos.

El capítulo 2 se enfoca a estudiar la base teórica del Método Discovery, aspectos de la teoría de compiladores, el Proceso Unificado y el desarrollo de interfaces de usuario.

Los capítulos 3,4 y 5 tratan sobre la construcción específica de la interfaz de verificación de diagramas de tareas, además del proceso de evaluación al cual se somete la herramienta desarrollada desde los enfoques de usuario y sistema.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Introducción

Ha pasado tiempo desde que el concepto de ingeniería de software fue propuesto y el desarrollo de software se estableció como disciplina; a lo largo de este periodo las metodologías, herramientas y técnicas de desarrollo de software han evolucionado. La evolución de estos aspectos ha permitido hoy en día centrarse no solo en los requerimientos del cliente, si no también en aspectos como seguridad, escalabilidad y estabilidad.

Hoy en día aún con la evolución de las metodologías, una de las herramientas/lenguaje que continúa siendo útil es el Lenguaje de Modela Unificado (UML). El éxito de UML es el uso de la característica gráfica en sus componentes, lo que permite estructurar sistemas de manera visual y fácil de comprender. sin embargo,, diversas investigaciones han encontrado inconsistencias en el lenguaje; ante esto han surgido otras alternativas consistentes como es el Método Discovery.

El Método Discovery es una aproximación al análisis y diseño de sistemas, el cual adopta un perfil UML; centrándose en la consistencia. Lo más interesante del método es que todo flujo de actividades tiene una representación gráfica y un soporte en un álgebra estrictamente definida.

Para comprender un poco más el tema, analicemos algunos conceptos de desarrollo, UML y verificación de modelos; centrándonos después en el motivo y problema que despertó el interés por este tema de investigación.

1.1.1. Desarrollo de Software

En la actualidad el desarrollo de software es una de las actividades más relevantes en el mundo de los negocios, y es en este contexto donde el concepto se ha transformado a una actividad de diseño colaborativa compleja que involucra diversos requerimientos, tareas y etapas iterativas [36]. Pero gracias a este cambio y a las exigencias del cliente, el desarrollo de software ha evolucionado en términos de estrategias y metodologías, mejorando la calidad, robustez y seguridad de los sistemas **[3].**

En los últimos años, una de las principales preocupaciones de los clientes es el fallo de los sistemas; esta preocupación se debe a que en muchos casos el funcionamiento de un sistema se puede traducir en ganancias, pérdidas, prevención y accidentes en actividades críticas como: control de vuelos, bolsa de valores, balística, evaluaciones médicas, entre otras aún más simples. Es por ello que los desarrolladores de software deben elegir cuidadosamente las metodologías; herramientas, y enfoques que permitan desarrollar sistemas seguros, fáciles de utilizar, funcionales y que satisfagan los requerimientos **[7]**.

Desde 1960, los modelos del ciclo de vida del software se han orientado a proporcionar esquemas conceptuales que permita planear, organizar, contratar, coordinar, presupuestar y encaminar las actividades relacionadas al desarrollo. Los modelos más conocidos son: el Modelo Clásico (Modelo en cascada), Refinamiento Paso a Paso, Desarrollo y Lanzamiento incremental **[28]**; estos modelos usualmente incluyen algunas versiones o subconjuntos de las siguientes actividades:

- Planeación
- Análisis de especificación y requerimientos.
- Especificación funcional o Prototipo.
- Partición y selección.
- Diseño de arquitectura y especificación de configuración.
- Especificación detallada de diseño.
- Implementación y depuración.
- Integración de software y pruebas.
- Documentación y liberación del sistema.
- Distribución e instalación.
- Entrenamiento y uso
- Mantenimiento del software.

El modelo clásico del ciclo de vida del software o Modelo en cascada es uno de los esquemas más utilizados para definir la estructura, planeación y gestionar grandes proyectos de desarrollo de software en entornos organizacionales complejos. Sus 6 etapas ofrecen un panorama de los requerimientos, estrategias y problemas que pueden presentarse a lo largo del desarrollo de un proyecto: estudio de feasibilidad, análisis y especificación de requerimientos, diseño, codificación, integración , distribución y mantenimiento**[28]**. Alternativamente a los modelos clásicos existen nuevas tendencias que centran su atención en el desarrollo final, en las configuraciones asociadas y en el proceso de desarrollo: Desarrollo de productos, Prototipo rápido y Desarrollo en conjunto.

Sin importar si el desarrollador utiliza un modelo clásico o avanzado existen 3 fases que resumen el procesos de desarrollo de un software: especificación de requerimientos, diseño preliminar e implementación; cada una de estas fases describe interconexiones iterativas de tareas

no lineales que producen un sistema funcional. Sin embargo, debido a la dependencia entre fases una mala especificación de requerimientos y diseño puede generar un sistema inestable o ambiguo que fallara a medida que se modifique y evolucione [1].

Para evitar que los sistemas fallen los ingenieros de software han desarrollado diversos métodos y herramientas que detectan errores, verifican propiedades de diseño en hardware y software, modelan escenarios críticos y permiten conocer como responderán los sistemas ante algunas situaciones no planeadas. Estas nuevas actividades requieren de mayor experiencia en los involucrados, mayor número de verificaciones a nivel diseño y mayor tiempo de desarrollo. Sin embargo, la presión por terminar el software en tiempo evita en muchos casos que se lleven a cabo, además aun cuando los sistemas se terminen en forma, las posibilidades de fallo no desaparecen.

Si analizamos el proceso de desarrollo de software, las etapas que determina el rumbo de los proyectos son la especificación de requerimientos y diseño. En estas etapas un mal diseño, una interpretación ambigua, un modelado erróneo, una especificación funcional dependiente incorrecta y otras características pueden generar un producto de software inconsistente [9].

Para evitar inconsistencias una de las opciones es analizar los artefactos que traducen los requerimientos del cliente a las especificaciones de diseño. Uno de los estándares para capturar aspectos de diseño en el software es el Lenguaje UML. Mediante UML es posible modelar, presentar y establecer requerimientos y estructuras para el desarrollo de sistemas de software previo al proceso intensivo de escribir código fuente [25].

1.1.2. Lenguaje de Modelado Unificado (UML)

El Lenguaje de Modelado Unificado es un lenguaje de propósito general de modelado visual que se utiliza para especificar, visualizar, construir y documentar los artefactos de un sistema de software; captura las decisiones y el entendimiento acerca de cómo deberían construirse los sistemas y se utiliza para comprender, diseñar, explorar, configurar, mantener y controlar la información acerca de los sistemas. UML incluye conceptos semánticos, notaciones y directrices con manejo de elementos dinámicos, ambientales y organizacionales. Su objetivo principal es el apoyo del modelado visual, el cual es la herramienta más útil del desarrollador de software [25].

Para recrear escenarios del sistema, UML cuenta con diversos tipos de diagramas, los cuales muestran diferentes aspectos de las entidades representadas. Los diagramas más comunes son: diagrama de clases, diagrama de componentes, diagrama de objetos, diagrama de estructura compuesta, diagrama de despliegue, diagrama de paquetes, diagrama de actividades, diagrama de casos de uso y diagrama de estados [25].

UML es ideal para ilustrar aspectos claves de diseño y ayudar a generar un mapa correspondiente a la implementación de un sistema; a lo largo de los años se ha convertido en una herramienta indispensable en el proceso de desarrollo de software, sin embargo, ha sido cuestionado por problemas de inconsistencia, semántica y ambigüedad. Estos problemas se presentan

fuertemente y podrían involucrar especificaciones contradictorias en los sistemas. Los problemas UML han sido estudiados con especial interés por [1] y [13] , quienes reportan una lista de problemas semánticos (significado) y de ambigüedad en la dinámica UML.

De manera particular, el problema de la semántica en UML se debe principalmente al hecho de estar descrita por un lenguaje semi-formal, con restricciones de objetos (OCL) y una semántica dinámica expresada en un inglés informal. Con respecto a la ambigüedad e inconsistencia estas se deben a la existencia de múltiples conceptos y tareas que muchas veces se contradicen en los diferentes diagramas que representan [13].

La imprecisión semántica de UML traduce en algunos casos interpretaciones ambiguas en los requerimientos de los sistemas, causando errores en etapas tempranas y resultados no deseados en el desarrollo final . Para [15] a menos que las notaciones gráficas estén soportadas por una semántica precisa, la utilidad de los diagramas es casi nula. Aun así UML ha demostrado que con diagramas visuales el proceso de desarrollo se agiliza enormemente, pero al presentar imprecisión no puede ser considerada como una herramienta fiable [1].

Aun cuando la semántica de UML no es realmente formal, muchas investigaciones como **[22]** coinciden que se pueden combinar las representaciones visuales con la precisión de un lenguaje de especificación formal. La formalización de las notaciones gráficas ayuda a identificar y eliminar las ambigüedades de los modelos, lo cual en el futuro facilita el mantenimiento del sistema. Bajo este concepto una alternativa de UML es el Método Discovery.

1.1.3. Método Discovery

El Método Discovery es una metodología Orientada a Objetos propuesta formalmente en 1998 por [32] y es considerada por el autor como un método centrado la mayor parte en el proceso técnico del desarrollo de sistemas. Este método de manera global es una aproximación al análisis y diseño de sistemas, el cual adopta un perfil UML restringido, centrándose en un minimalismo y consistencia.

El Método Discovery ha sido utilizado como notación semántica basada en UML, pero cambiando algunos modelos cuando se considera necesario; propone una sintaxis reducida y precisa para especificar el software [33] y ha sido probado en un gran número de proyectos industriales en la Universidad de Sheffield [9], lo cual hace que sea una opción apropiada trabajar con él.

En el Método Discovery existen 4 principios dominantes; Dirección, Selectividad, Transformación y Compromiso, los cuales describen la secuencia de diseño, selección, análisis de modelos y comunicación con los clientes. Adicionalmente el método está organizado en 4 fases: Modelado de negocios, Modelado de Objetos, Modelado del sistema y Modelado del software.

En la fase de Negocios, la idea principal es obtener el contexto del sistema, capturar sus requerimientos y analizarlos con el objetivo de buscar una decisión acerca del desarrollo del producto (contrato y cotización). El Modelado de Objetos es la siguiente fase y su objetivo es

identificar los objetos y unidades modulares de diseño. Para la tercera fase; Modelado del sistema, es necesario analizar las dependencias, clases e identificar los subsistemas naturales. La última fase es el modelado de software, donde se trabaja en la traducción del diseño hacia código en algún lenguaje de programación [9].

En similitud con UML, el método plantea una nueva forma de representar el flujo de actividades en la fase de diseño mediante los diagramas de tareas. Una tarea es definida como algo que "tiene el sentido específico de una actividad llevada a cabo por las partes interesadas con un fin económico" [34].

Los diagramas de flujo de tareas representan la secuencia de actividades que describen el comportamiento del sistema; su flujo y estructura pueden mapearse en términos de un álgebra de tareas abstracta, la cual se define utilizando axiomas y una sintaxis específica, esta álgebra a su vez puede traducirse en una semántica notacional, la cual consiste en un conjunto de trazas de eventos que representan tareas atómicas [9].

Como resultado, Cualquier modelo de tareas desarrollado en el Método Discovery puede convertirse a expresiones del álgebra de tareas, con su correspondiente notación semántica sin ambigüedad. Después de la conversión algebraica los diseños pueden comprobar automáticamente la equivalencia y las propiedades lógicas temporales. Dicha comprobación facilita la validación temprana del diseño, estableciendo características de consistencia y completitud.

1.1.4. Verificación de Modelos

El Método Discovery establece los componentes de diagrama y álgebra de tareas para modelar los sistemas. Un modelo representa el puente entre los procesos de análisis y diseño, describe el entorno, comportamiento y estructura de un sistema. Los modelos en el proceso de desarrollo de software se utilizan en la etapa de análisis para desarrollar o especificar un sistema.

La verificación de modelos *(Model checking*) es la técnica de verificación que consiste en probar que un sistema dado , por ejemplo un sistema de computadora, protocolo de red o diseño de hardware cumple o no una especificación dada. La comprobación de estas especificaciones permite evitar situaciones, estados o condiciones no deseadas en un modelo o flujo de eventos **[18]**.

El proceso de comprobación de estados en Model Checking puede dividirse en 3 pasos [14]:

- **Modelar:** El sistema M que va a ser analizado mediante algún lenguaje de Modelado
- **Especificar:** Establecer las propiedades f que deben satisfacerse utilizando alguna variante de la lógica temporal.
- **Verificar:** comprobar si el sistema diseñado es un modelo de la propiedad.

Una de las herramientas actuales que permite realizar el proceso de verificación de estados aplicada al Método Discovery es definida en el capítulo 8 de la semántica abstracta de tareas y actividades del Método Discovery [9]. El autor describe la herramienta como un compilador que

recibe un modelo de tareas algebraico y a través de especificaciones en Lógica Lineal Temporal (LTL) o Lógica Computacional en Árbol (CTL) realiza la verificación.

Las expresiones en lógica lineal temporal verifican propiedades de caminos en arboles computacionales; particularmente LTL se centra en dos propiedades: Accesibilidad y Seguridad [40]. En Model Checking cuando estas propiedades no se cumplen se devuelve un contra-ejemplo.

En el caso de la lógica computacional en árbol, no existe una propiedad específica debido a que el modelo se analiza como una ramificación de posibilidades. CTL utiliza los cuantificadores sobre caminos *All (A) y Exist (A)* para comprobar que ciertas propiedades sean verdaderas en todas las rutas o en algunos caminos [42]. En Model checking los resultados de estas expresiones son un conjunto de ramificaciones que cumplen una especificación dada.

La herramienta que describe el autor está desarrollada en el lenguaje Haskell y es completamente funcional, sin embargo, funciona a nivel consola y muestra los resultados en modo texto. Esta situación nos hace pensar directamente en sus desventajas, pero también en sus mejoras, evoluciones y nuevas adaptaciones.

Analizando la forma en que la herramienta representa los resultados, las comprobaciones con LTL no presentan dificultad de interpretación, sin embargo, en el caso de CTL la comprobación se hace sobre todos los caminos, por ello es innegable que los resultados de la verificación sean ramificaciones de eventos y caminos que conducen a la existencia verdadera de una propiedad; imaginemos por un momento que el sistema es completo a una ramificación de 200 actividades ; sería imposible o altamente complejo que el usuario pueda entender, interpreta o seguir resultados de más de 4 páginas.

1.2. Planteamiento del Problema

La validación de los diseños/modelos en el proceso de desarrollo de software debería ser un requerimiento para garantizar sistemas consistentes; sin embargo, no siempre es posible realizarla. Considerando la importancia de la consistencia en los sistemas, la nueva propuesta del Método Discovery y los avances que se tienen respecto a sus herramientas; surge la necesidad de dar continuidad al proyecto de verificación de especificaciones del Método Discovery.

Analizando las deficiencias que se tienen respecto a la herramienta [9], es necesario generar una estructura/interfaz visual que integre y enlace las funcionalidades del compilador y simplifique el proceso de verificación. Recordemos que la etapa en la que se encuentra el proyecto es una interfaz consola (modo texto) aislada, por ello es necesario modificar el escenario de construcción de expresiones de interfaz de comandos a un concepto visual (interfaz), transparente y que se integre al proceso de desarrollo de software.

Es importante mencionar que la construcción de expresiones LTL y CTL aplicadas en modelos es una actividad compleja, por ello se requiere además definir en el escenario visual subherra-

mientas que faciliten la construcción sintáctica de las expresiones, su ejecución y la visualización de resultados. Con el desarrollo de una interfaz para realizar el proceso de verificación de expresiones en el Método Discovery, el usuario tendrá a la mano una herramienta visual-estructurada que le permitirá verificar los modelos visualmente, corregirlos y/o mejorarlos antes de codificarlos.

Con la creación de estos componentes se hará más accesible el desarrollo de software, se difunde la utilización de nuevas metodologías y con la ayuda de las tecnologías adecuadas representará un gran aporte a la fase de análisis y diseño del software.

1.3. Entornos de Desarrollo

Con respecto a la inquietud por integrar la verificación de modelos como parte del proceso de desarrollo, una de las estrategias que debe utilizarse es la incorporación de la herramienta a los entornos de desarrollo. Los entornos de desarrollo líderes en el mercado son : Eclipse IDE, Visual Studio y Netbeans IDE[26]. Aun cuando cada uno ofrece diferentes ventajas, Eclipse ha ganado popularidad debido a la libertad de integración que ofrece y a la capacidad de incorporar diversas herramientas de desarrollo para cualquier lenguaje de programación, mediante la implementación de complementos (plug-ins).

La arquitectura de plug-ins de Eclipse permite integrar diversos lenguajes sobre un mismo entorno e introducir otras aplicaciones que pueden resultar útiles durante el proceso de desarrollo de sistemas como: herramientas UML, editores visuales de interfaces, ayuda en línea de librerías, entre otras. Eclipse IDE es uno de los entornos más populares según estadísticas de la fundación Eclipse realizada en el 2009[26] y gracias a todas sus ventajas lo utilizaremos para desarrollar nuestro proyecto de investigación.

La fundación Eclipse es una organización sin fines de lucro, apoyada por los miembros de las sociedades que albergan los proyectos de Eclipse, y al ser uno de los más utilizados los escenarios son ideales para incorporar y distribuir plug-ins que permitan agilizar y formalizar el desarrollo de software [10].

1.4. Objetivos

1.4.1. Objetivo General

El objetivo general sobre el cual se guiará esta investigación es el desarrollo e implementación de interfaces que permitan la estructuración, verificación y ejecución de consultas en lógicas temporales (LTL y CTL) aplicadas a la verificación de Diagramas de Tareas en la especificación del software. También se pretende representar los resultados de las verificaciones de una manera simple, fácil de analizar y entendible para el usuario; en su conjunto las interfaces se incorporarán en un plug-in en el entorno de desarrollo Eclipse para el uso de los desarrolladores de software.

1.4.2. Objetivos Específicos

- Analizar las implementaciones existentes respecto a la construcción de consultas LTL y CTL.
- Analizar las implementaciones existentes respecto a la representación visual de los resultados obtenidos al utilizar la comprobación de modelos usando lógicas temporales (LTL y CTL).
- Desarrollar un método propio para modelar la construcción de las consultas (sintaxis, estructura) y la visualización de los resultados.
- Desarrollar análisis sintácticos de la construcción de la consulta
- Diseño e implementación de una interfaz que permita la construcción de las consultas LTL y CTL.
- Diseño e implementación de una interfaz de visualización de resultados de las consultas de manera gráfica.
- Incorporar las herramientas desarrolladas en una estructura plug-in para el entorno de desarrollo eclipse
- Realizar la comprobación del plug-in utilizando diferentes casos de prueba.
- Analizar los resultados obtenidos y el impacto que estos generan en la fase de especificación del software.
- Plantear las aportaciones de la herramienta al área de ingeniería de software, así como señalar la eficiencia que puede lograrse al utilizarse la herramienta en la comprobación del diseño del software en la vida real.

Capítulo 2

Marco Teórico

2.1. Introducción

El desarrollo de sistemas de software es una actividad colaborativa en la cual cada uno de los involucrados obtiene información, realiza tareas, soluciona problemas y aporta un punto de vista específico para el sistema. Participar en un proyecto de software requiere habilidades específicas para realizar algunas de las siguientes tareas: analizar problemáticas de usuarios, reunir especificaciones, verificar requerimientos, diseñar funcionalidades, codificar módulos, documentar sistemas y realizar el mantenimiento.

En el proceso de desarrollo de software la interrelación y dependencia en las tareas demanda en los involucrados tener conocimiento del estado del sistema en todo momento. Desconocer el estado del sistema puede provocar un rumbo no deseado en el desarrollo; diversos autores recomiendan especial cuidado en la transición entre la etapa de análisis y diseño; esto debido que las etapas definen *el qué y el como* del proyecto [20].

Considerando un sistema de software como un conjunto de elementos abstractos relacionados, el autor [31] define que la etapa más importante de un sistema es el diseño de la interfaz. La interfaz de un sistema es el elemento que permite comunicar al usuario con el sistema y dependiendo del tipo puede simplificar, complicar o facilitar la realización de las tareas en diferentes niveles [19]. Antes de conceptualizar el sistema como un arreglo de componentes gráficos es necesario aclarar que debe existir una capa de funcionalidad programable que de soporte al sistema, es decir, en sus componentes individuales básicos.

Una vez analizados estos dos enfoques, en el actual tema de investigación el objetivo principal es el desarrollo de una interfaz/entorno de verificación de diagramas de tareas (*Task Flow Diagrams Verification Interface*, TFI por simplicidad), sin embargo antes de analizar la capa visual es necesario resumir las funcionalidades programables y requerimientos del sistema:

- El sistema debe centralizarse en los modelos y expresiones como objetos primarios.
- El sistema debe tener la capacidad de verificar sintácticamente cada una de las expresiones que el usuario introduce.

- El sistema debe tener la capacidad de recibir un conjunto de expresiones en lógicas temporales, ejecutarlas y generar salidas útiles.
- El sistema debe tener la capacidad de representar los resultados de manera simple y fácil de analizar y comprender por el usuario.
- El sistema debe ser configurable por el usuario.
- El sistema debe ser escalable y tener la capacidad de comunicarse con otros sistemas.

Como podemos darnos cuenta los requerimientos especifican un proyecto de verificación sintáctica, que ejecuta expresiones, sea configurable y que además represente resultados en un modelo visual comprensible para los usuarios. Para construir estas características se requieren las bases teóricas de los siguientes temas: **Método Discovery, Diagramas de Tares, Lógicas Temporales, Teoría de Compiladores, Interfaces de Usuario y Entornos de desarrollo.**

2.2. Método Discovery

A lo largo de los años, el desarrollo de software ha evolucionado paralelamente con sus metodologías, notaciones, lenguajes de codificación y herramientas. Estas situaciones permiten hoy en día que los sistemas sean más fáciles de construir y analizar desde diversas perspectivas. Una de las notaciones estándares en los procesos de desarrollo es el lenguaje de Modelado Unificado (UML).

Desde la perspectiva del autor [29] UML es una de las herramientas más emocionantes en el desarrollo de sistemas, el cual se define como un lenguaje de modelado visual de propósito general utilizado para especificar, visualizar, construir y documentar los artefactos de software de un sistema [25]. El lenguaje UML ha sido adoptado como estándar de la representación orientada a objetos principalmente por su facilidad de compresión, uso y amplia perspectiva.

En UML un sistema de software debe reflejar la perspectiva del cliente y del equipo de desarrollo involucrado; para conjuntar los diferentes puntos de vista, las herramientas que se utilizan consisten en una serie de diagramas que describen puntos de vista, estados y estrategias de diseño. Los diagramas UML por definición son considerados modelos, debido a que describen el flujo, la composición, la arquitectura y comunicación de un sistema [36].

De los diagramas UML los principales son: diagrama de clases, casos de uso, estados, secuencias, colaboración y componentes. Autores como [1, 13, 22] han estudiado características de estos diagramas y han concluido que presentan inconsistencias y ambigüedades. Algunas inconsistencias particulares estudiadas por el autor [24] son las siguientes:

- **Inconsistencia entre clases de diferentes niveles:** la representación de diferentes niveles de abstracción no se basa en un formalismo. Simplemente se usan diferentes diagramas en los que algunas clases coinciden (se permite inconsistencias de dependencia en dos diagramas similares).
- **Inconsistencia entre clase y diagrama de secuencia:** se permite en el diagrama de secuencia llamadas a métodos que el diagrama de clases prohíbe (llamadas a objetos con

métodos de ámbito prohibidos).

- **Inconsistencia de cardinalidad:** no se comprueba que se respete la cardinalidad expresada en el diagrama de clases cuando se generan instancias en las secuencias.
- **Inconsistencia entre estado y secuencia:** se permiten secuencias incompletas con respecto al estado de los sistemas.

Las inconsistencias anteriores corresponden a un caso de estudio particular; otro de los problemas es que en la especificación actual no se restringen los formatos gráficos y probablemente no exista una sintaxis gráfica para indicar relaciones ilegales. También UML varia de significado dependiendo del usuario, cada usuario puede interpretar los diagramas de diferentes maneras y es que no existe una notación formal que rija todos los niveles; la ocurrencia de estas situaciones y otros problemas origina constantemente inconsistencia, ambigüedad, inexactitud y reconocimiento cognitivo erróneo **[9, 13, 22]** .

Ante las situaciones inconsistentes diversos métodos, lenguajes y herramientas han sido creados para la validación de los componentes UML entre ellos se encuentran: lenguaje Z, Allow y OCL [9], sin embargo estas herramientas son independientes de UML y cuando UML libera una nueva versión estas no se integran. Alternativo a los lenguajes de corrección existen otros métodos independientes que plantean nuevos paradigmas para el proceso de desarrollo, uno de ellos es el Método Discovery.

El Método Discovery es un método orientado a objetos utilizado principalmente en el modelado de negocios, su notación simple y similar a UML permite que sea fácil de entender y utilizar, además dedica especial interés a especificar el propósito de cada elemento y de la notación en particular. El Método Discovery fue propuesto formalmente en 1998 por [32, 33][35] en la Universidad de Sheffield y desde su versión 1 específica que existen 4 principios fundamentales : dirección, selectividad, transformación y compromiso. De manera similar a un proceso de desarrollo de software el Método Discovery se organiza en 4 fases: Modelado de negocios, Modelado de Objetos, Modelado de Sistema y Modelado de Software; la fase en la que nos enfocaremos será el Modelado de negocios, debido a que es la parte donde se exploran los objetivos y se plantean las especificaciones de los sistemas.

2.2.1. Modelado de Negocios

El Modelado de negocios es la fase inicial del Método Discovery y en esta fase el objetivo central es explorar y representar los requerimientos del cliente en un modelo estructurado contextualizando el ambiente donde funcionará. Las actividades principales en esta fase es identificar las tareas correspondientes al modelo y plantear un sistema mejorado en términos de las tareas de negocios, cuidando los objetivos del cliente [9].

En similitud con el ciclo de vida del software, la fase de Modelado permite identificar los requerimientos del sistema, las tareas; su relación con el usuario y el proceso de planeación que debe seguirse. La necesidad de obtener el conocimiento del contexto del sistema, sus tareas, dependencias e involucrados permite encontrar una decisión acerca del ámbito y costo del pro-

yecto. En el modelado de negocios la unidad básica de trabajo es la *tarea* , su concepto se define a continuación:

Tarea: Se define en el Método Discovery como "una actividad de propósito específico realizada por los involucrados con un fin de negocios"; esta unidad eventualmente conduce a los dos tipos de diagramas de tareas: El diagrama de estructuras de tareas y el diagrama del flujo de tareas [9].

2.2.2. Diagrama de Estructura de Tareas

Después de que los desarrolladores han recopilado información referente a las tareas, sus dependencias y los actores involucrados; el siguiente paso consiste en representar gráficamente estos elementos a través de los diagramas de estructura de tareas. La notación utilizada por los diagramas de tareas en el Método Discovery es simple, los elementos son tomados del estándar UML pero son presentados de forma más concisa y consistente [9].

Dentro de la notación del Método Discovery existen 3 componentes principales: actor, tareas y objetos. Una tarea se representa por medio de una elipse sin importar la magnitud del proceso de negocios; los actores son representados como un personaje-dibujo y los objetos participantes como rectángulos (ver figura 2.1). Las relaciones entre los actores u objetos con las tareas puede ser de dos tipos: participación y propiedad.

Una relación de participación implica que el actor u objeto se encuentra involucrado con la tarea, la forma de representar esta relación es por medio de una línea simple. Una relación de propiedad implica que además de existir una relación de participación el actor u objeto es responsable de la tarea, en este caso la representación es una línea simple con un pequeño círculo relleno al final de la relación (Ver figura 2.1).

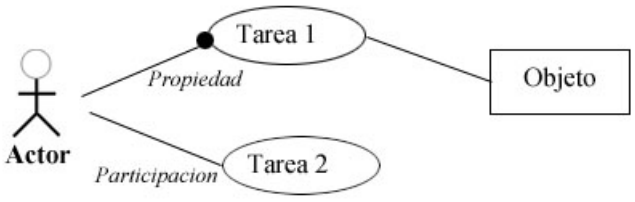

Figura 2.1: Elementos básicos de un diagrama de estructuras de tareas[9]

Los diagramas de estructura de tareas también consideran las operaciones de agregación y generalización, ambas utilizadas bajo el mismo significado que UML. Una operación de agregación se representa con un rombo indicando que una tarea mayor se subdivide en dos subtareas más pequeñas; mientras que la operación de generalización se representa por medio de un triángulo

e indica que una tarea abstracta general generaliza una colección de tareas específicas concretas (ver figura2.2).

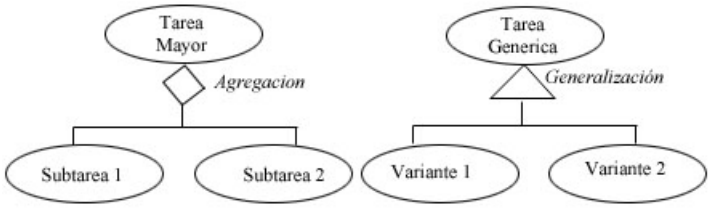

Figura 2.2: Relaciones estructurales en un diagrama de estructura de tareas[9]

2.2.3. Diagrama de Flujo de Tareas

Cuando los actores, objetos y tareas han sido identificados, resulta de gran utilidad analizar el flujo de trabajo del negocio para identificar secuencias, dependencias, excepciones y condiciones. En el Método Discovery la forma de representar el flujo de trabajo es a través de los diagramas de Flujo de Tareas. Los diagramas de flujo de tareas están basados en los diagramas de actividades de UML y representa el orden en que las tareas se realizan en el negocio así como su dependencia lógica con otras tareas [9].

En un diagrama de flujo de tareas, las tareas se conectan por flechas que indican la dirección del flujo de ejecución. En la dirección del flujo pueden existir las siguientes operaciones: secuencia, selección, excepción y paralelismo. La selección representa una división de flujos alternativos y se representación gráfica es un rombo, mientras que una excepción representa la selección entre un flujo normal o un salto hacia una excepción, su representación se basa en un triángulo.

La realización de tareas en paralelo en el proceso de negocios es común y necesaria debido a que muchas veces los procesos son independientes y consecuentemente concurrentes. Para indicar una transición de paralelismo se utilizan dos símbolos: bifurcación y unión. Una bifurcación es una transición de una tarea fuente hacia múltiples tareas objetivo y la unión se conceptualiza como una transición de múltiples tareas fuentes hacia una tarea objetivo (Ver figura 2.3).

De la figura 2.3 cuando dos tareas se ejecutan en paralelo, cada uno de los flujos puede terminar de manera independiente, sin embargo las tareas subsecuentes a la unión solo se ejecutarán cuando los dos flujos se hayan sincronizado.

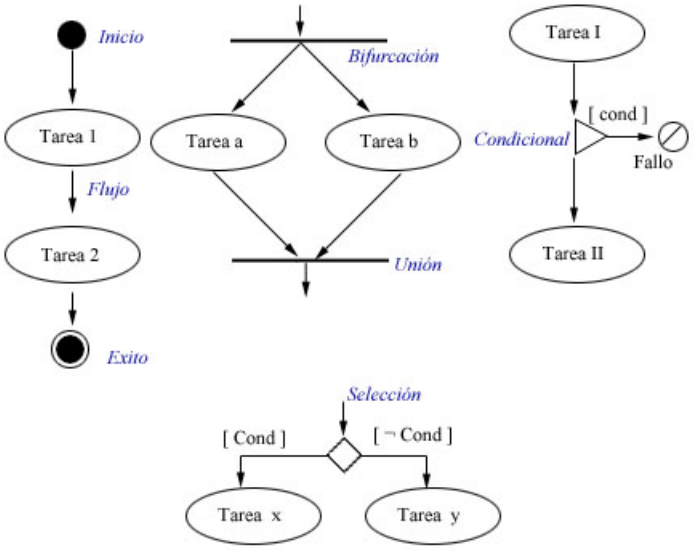

Figura 2.3: Elementos de un diagrama de flujo de tareas[9]

2.2.4. Álgebra de Tareas

Una forma de establecer formalidad en los diagramas de flujo tareas es a través del Álgebra de Tareas. El Álgebra de Tareas es una representación sintáctica que traduce los diagramas de flujo de tareas en un modelo algebraico. Para cada componente del diagrama de tareas existe una estructura sintáctica abstracta equivalente, es decir, para las operaciones de secuencia, selección, paralelismo y repetición[9].

1. **Secuencia de Tareas:** Permite expresar flujos secuenciales de tareas; para ello cada una de las actividades debe separarse por **';'** y se ejecutarán de izquierda a derecha (ver figura 2.4).

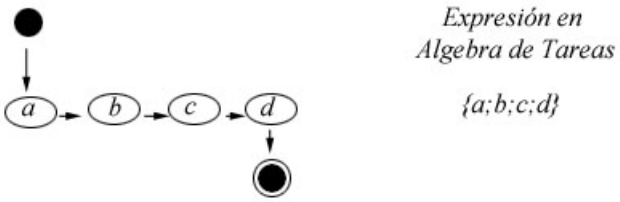

Figura 2.4: Traducción de una instancia de secuencia en álgebra de tareas.

2. **Selección:** Describe una bifurcación de alternativas del flujo de ejecución; cada una de las tareas se representan separadas por el símbolo **'+'** y las condiciones de la selección no se incluyen; en el caso de excepciones el símbolo de fallo se traduce directamente como

elementoφ (ver figura2.5) .

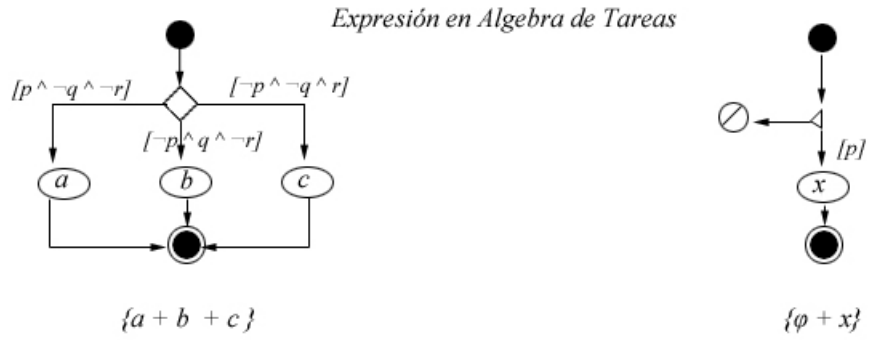

Figura 2.5: Traducción de una instancia de selección y excepción en el álgebra de tareas.

Es importante mencionar que en los diagramas de tareas existen 2 situaciones importantes: fallo y éxito. Las cuales se representan a través de los símbolos σ para expresar el éxito y φ para indicar fallo en ejecución del flujo.

3. **Repetición:** Esta operación puede representarse mediante dos estructuras similares a los lenguajes de programación: ciclo hasta y ciclo mientras. La repetición ***"hasta"*** se define como un ciclo de ejecución de una tarea *a* hasta que una condición p se cumpla; en el álgebra de tareas la representación es la siguiente: μx. (a ; ε +x) (ver figura2.6). En el caso de la repetición ***"mientras"*** la ejecución de la tarea *a* se mantienen mientras una condición p continúe siendo válida, su traducción directa es: μx. (ε +x; a).

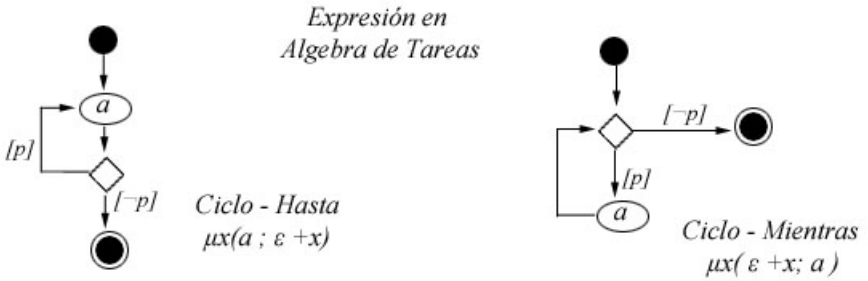

Figura 2.6: Traducción de una instancia de repetición en el álgebra de tareas

En la figura 2.6 el símbolo ε en el álgebra de tareas representa una actividad vacía.

4. **Composición Paralela:** Muchas veces resulta útil expresar actividades que se realizan de manera simultanea, para ello cada uno deben separarse a través del símbolo **'||'**, por ejemplo la figura siguiente muestra la ejecución de dos flujos paralelos agrupados (ver figura

2.7).

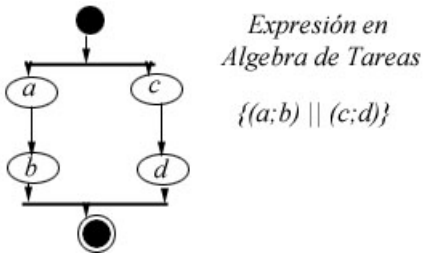

Figura 2.7: Traducción de una instancia de paralelismo en el álgebra de tareas.

2.2.5. Representación sintáctica abstracta en el modelo de flujo de tareas

En el apartado anterior se mencionaba que la unidad básica de trabajo en el modelo de negocios del Método Discovery es la tarea, la cual puede ser de tipo simple o compuesto; una tarea compuesta es aquella que agrupa subtareas en una nueva instancia global. En el modelo de flujo de tareas se pueden combinar tareas simples y compuestas utilizando los operadores del modelo de flujo de tareas. Las estructuras sintácticas básicas para el modelo de flujo de trabajo son las siguientes [9]:

Composición Secuencial: Define la ejecución cronológica de tareas expresando una ejecución de izquierda a derecha; el operador que representa esta estructura es el *';'*.

Selección: Representa una selección de alternativas dentro del flujo de un sistema; su sintaxis utiliza el símbolo ***'+'***.

Composición Paralela: Define la ejecución simultanea de elementos en una expresión, el símbolo que se utiliza es ***'||'***.

Repetición: Permite la reiteración de una expresión en el flujo a través de dos estructuras hasta y mientras, el símbolo que se utiliza es ***'μ x'***.

Encapsulación: Se utiliza para agrupar un conjunto de tareas en una instancia de tarea , para ello se utiliza los símbolos ***'{ }'***.

Dentro de estas estructuras se permite el uso de paréntesis para agrupar subconjuntos de tareas, de esta manera es mucho más sencillo expresar las sentencias. La sintaxis para el modelo de flujo de tareas es la que controla la definición del álgebra de tareas y puede ser considerada como un álgebra universal; su forma BFN se muestra en la figura 2.8 y sobre la cual nos enfocaremos en capítulos siguientes.

Activity ::= ε	*Actividad vacia*
| σ	*Exito*
| Φ	*Fallo*
| Task	*Tarea unica*
| Activity ; Activity	*Secuencia de actividades*
| Activity + Activity	*Seleccion de actividades*
| Activity || Activity	*Actividades paralelas*
| μx.(Activity ; ε + x)	*Ciclo de actividades: hasta*
| μx.(ε + Activity ; x)	*Ciclo de actividades: Mientras*
Task::= Simple	*Tarea simple*
| {Activity}	*Encapsulación de Actividades*

Figura 2.8: Representación sintáctica BNF para el modelo del flujo de tareas[9]

2.2.6. Verificación de Modelos

La verificación de Modelos es una técnica de verificación que explora todos los posibles estados de un sistema utilizando fuerza bruta, de manera similar a como se explorarían todos los caminos en una jugada de ajedrez. La implementación de esta técnica es un sistema que explora todos los escenarios posibles de manera sistemática y es través de esta forma que puede comprobarse si un modelo satisface realmente una propiedad específica [2].

Examinar todas las posibilidades de estados en un sistema representa un reto para estas herramientas debido a que los recursos en procesamiento y memoria son limitados. Típicamente la aproximación de verificación de modelos debe verificar si una propiedad específica se satisface en un modelo y en caso contrario debe mostrar un contra-ejemplo que exprese la violación de la propiedad [2].

Al ser la comprobación una búsqueda exhaustiva es necesario modelar el sistema como una estructura de Kripke la cual es similar a una máquina de estados finitos. *Una estructura de Kripke es una triple M= (S,R,P) donde S es el conjunto finito de estados,* $R \subseteq S \times S$ *es el total de relaciones de transición y P:* $S \rightarrow 2^{AP}$ *es la función de etiquetado (AP es el conjunto finito de proposiciones atómicas). Una función de Kripke implica que un estado* $s \in S$ *en un momento específico puede saltar hacia cualquier estado* $s' \in S$ *a través de la función P (s) en un conjunto de proposiciones válidas [18].*

En el Método Discovery la estructura de kripke que describe los flujos del sistema son las trazas, las cuales representan el proceso de ejecución de un sistema y se obtienen procesando el álgebra de tareas por medio de un compilador. Una traza se define como una cadena, donde cada símbolo representa una acción atómica ejecutada por un proceso y en conjunto describen todos los posibles caminos a través de los cuales el sistema se ejecuta [9].

Para modelar la especificación a verificar en la estructura de kripke se utilizan fórmulas en lógica temporal de tipo estado o camino; la fórmula puede contener operadores y cuantificadores que especifiquen comportamientos dinámicos del sistema. Dentro de las lógicas temporales más comunes se encuentran: Lógica lineal temporal (LTL) y Lógica computacional en árbol (CTL) [8][18] .

Una fórmula de estado temporal por inducción se define de la siguiente manera [18]:

- Si $f \in$ AP, entonces f es una fórmula de estado.
- Si f y g son fórmulas de estado, entonces $\neg f$, $f \bigwedge g$, $f \bigvee g$ y $f \rightarrow g$ son fórmulas de estado.
- Si f es una fórmula de camino, entonces Af y Ef son fórmulas de estado; donde A representa All: En todos los caminos y E representa Exist: Existe al menos 1 camino.

Una fórmula de camino por inducción se define de la siguiente manera[18]:

- Si f es una fórmula de estado , entonces f es una fórmula de camino.
- Si f y g son fórmulas de camino , entonces $\neg f$, $f \bigwedge g$, $f \bigvee g$ y $f \rightarrow g$, Xf , Ff , Gf y fU g son fórmulas de camino.

Dónde:

- **X**: Representa “En el siguiente estado del camino”;
- **F**: Representa “Eventualmente en un estado a lo largo del camino”,
- **G**: Representa “Globalmente en todos los estados a lo largo del camino ”
- **U**: Representa “Hasta”.

En base a estas consideraciones podemos definir una fórmula CTL y LTL de la siguiente manera:

Computation Tree Logic (CTL): La lógica computacional en árbol es un conjunto de fórmulas de estado temporal, en la cual se construye un árbol de transición designando a un estado como raíz y muestra todas las posibles computaciones desde ese estado. En esta lógica se permite utilizar operadores temporales y de cuantificación sobre un conjunto de caminos: All y Exist; los cuales son utilizados para evaluar directamente una especificación sobre el árbol de transiciones [18]. Al ser una fórmula de estado temporal que trabaja sobre un conjunto de caminos los resultados de su evaluación devuelven un árbol de transiciones que alcanzan cierto estado en común. La comprobación de especificaciones utilizando este tipo de lógica permite saber si por ejemplo una planta nuclear explotaría y cuales serían los caminos que conducirían a ese estado [5, 8].

Una expresión CTL está formada típicamente por una combinación de operadores lógicos, operadores de camino y operadores modales temporales. Los operadores lógicos son usualmente: $\neg$, $\bigwedge$,$\bigvee$ y $\rightarrow$. La sintaxis de las expresiones se muestran a continuación [9]:

1. Operadores Lógicos:
   a) **Not** p
   b) **And** p q
   c) **Or** p q

d) **Impl** p q

Donde p y q son proposiciones.

2. Operadores de Camino: Son cuantificadores All (A) y Exist (E) utilizados en combinación con operadores temporales modales: Next (X), Always or Globally (G), Finally (F) y Until (U).

   a) Next: **X** p - *p se mantiene en el siguiente estado.*

   b) Globally: **G** p - *p se mantiene globalmente.*

   c) Finally: **F** p - *p se mantiene en algún estado futuro.*

   d) Until: **U** p q - *q se mantiene en el estado actual o p es verdadero entonces q.*

Es importante aclarar que los operadores de camino A y E no pueden utilizarse sin los operadores modales X,G,F y U.

De acuerdo a [9] una expresión CTL puede enunciarse como: Pr <task> y para la verificación de específicaciones se requieren expresiones de la forma: check <task-algebra -expression> <CTL-expression>. La evaluación de este tipo de expresiones además de devolver el árbol de caminos, subraya los nodos que son considerados válidos para la especificación dada.

LTL (Linear Temporal Logic): La lógica lineal temporal es un subconjunto de CTL que contiene fórmulas de la forma Ef , donde f es una fórmula de camino con proposiciones atómicas como subformulas de estado. Al ser una fórmula de verificación de existencia de 1 camino los resultados de su evaluación son directamente: falso o verdadero; en el caso de un resultado falso LTL devuelve un contra-ejemplo que hace invalida la especificación [18]. De manera formal una fórmula LTL de camino por inducción se define de la siguiente manera [18]:

- Si f es una proposición atómica, entonces f es una fórmula de camino.
- Si f y g son fórmulas de camino, entonces $\neg f, f \bigwedge g$, $f \bigvee g$ y $f \rightarrow g$, Xf , Ff , G f y fU g son fórmulas de camino.

LTL es una lógica temporal que se construye agregando operadores a un predicado, estos operadores pueden ser utilizados para referirse a estados futuros sin cuantificación sobre los caminos. Los operadores lógicos que se utilizan son heredados del cálculo preposicional tales como: $\neg$, $\bigwedge$,$\bigvee$ y $\rightarrow$. La sintaxis de las expresiones se muestran a continuación [9]:

1. Expresiones Lógicas

   a) **Not** p

   b) **And** p q

   c) **Or** p q

   d) **Impl** p q

2. Expresiones Modales Unarias

a) Next: **X** p - *p se mantiene en el siguiente estado.*

b) Globally: **G** p - *p se mantiene globalmente.*

c) Finally: **F** p - *p se mantiene en algún estado futuro.*

3. Expresiones Modales Binarias

a) Until: **U** p q - *q se mantiene en el estado actual o p es verdadero entonces q.*

b) Weak-Until: **W** p q - *q se mantiene en el estado actual o p es verdadero entonces q, o p es verdadero para todos los estados.*

c) Release: **R** p q - *q se mantienen en todos los estados o hasta que p sea verdadero.*

De acuerdo a [9] una expresión LTL puede enunciarse como: Pr <task> y para la verificación de especificaciones se requieren expresiones de la forma: check <task-algebra -expression> <LTL-expression>.

2.2.6.1. Verificación de especificaciones en el Método Discovery

Hasta este punto se ha definido el Método Discovery, los diagramas de tareas, álgebra de tareas y las especificaciones de las lógicas temporales; pero aun es necesario ilustrar como se aplican el concepto de verificación de modelos al Método Discovery. La verificación de especificaciones en un modelo es una de las causas por las que se eligió el tema de investigación.

Una de las herramientas desarrolladas para el Método Discovery es un compilador que utiliza la técnica de verificación **Model Checking** para verificar propiedades en un modelo algebraico resultante de un diagrama de flujo de tareas. El estado de esta herramienta es en modo consola, la cual recibe como entradas el modelo y la expresión en alguna de las lógicas LTL y CTL. A continuación se analiza el proceso de funcionamiento utilizando esta herramienta.

El punto de partida para la verificación de especificaciones en un modelo en el Método Discovery es la construcción del diagrama de flujo de tareas y la traducción al álgebra de tareas. Para comenzar imaginemos el escenario donde un cliente tiene un problema x y para solucionarlo acude con una empresa la cual estudiará el problema y le ofrecerá una solución.

La empresa ha solucionado diversos problemas a diferentes clientes y en su experiencia plantea que la búsqueda de una solución para todo problema x se basa en los siguientes flujos:

1. Comprender el problema.
2. Revisar si existen soluciones de otras personas para el problema.
3. Seleccionar soluciones y realizar una similitud de variables con el nuevo problema.
4. Proponer una solución o bien observar nuevamente el problema.
5. Aplicar la solución
6. Al aplicar la solución esta puede fallar. En caso del éxito el siguiente paso es la evaluación de resultados.
7. Finalmente se documenta y entrega la solución al cliente.

8. El problema finaliza.

Bajo el escenario anterior los siguientes puntos especifican la verificación del modelo del problema x.

1. Diagrama de tareas

Si traducimos el flujo del problema x a un diagrama de tareas (flujos), el resultado es el mostrado en la figura 2.9.

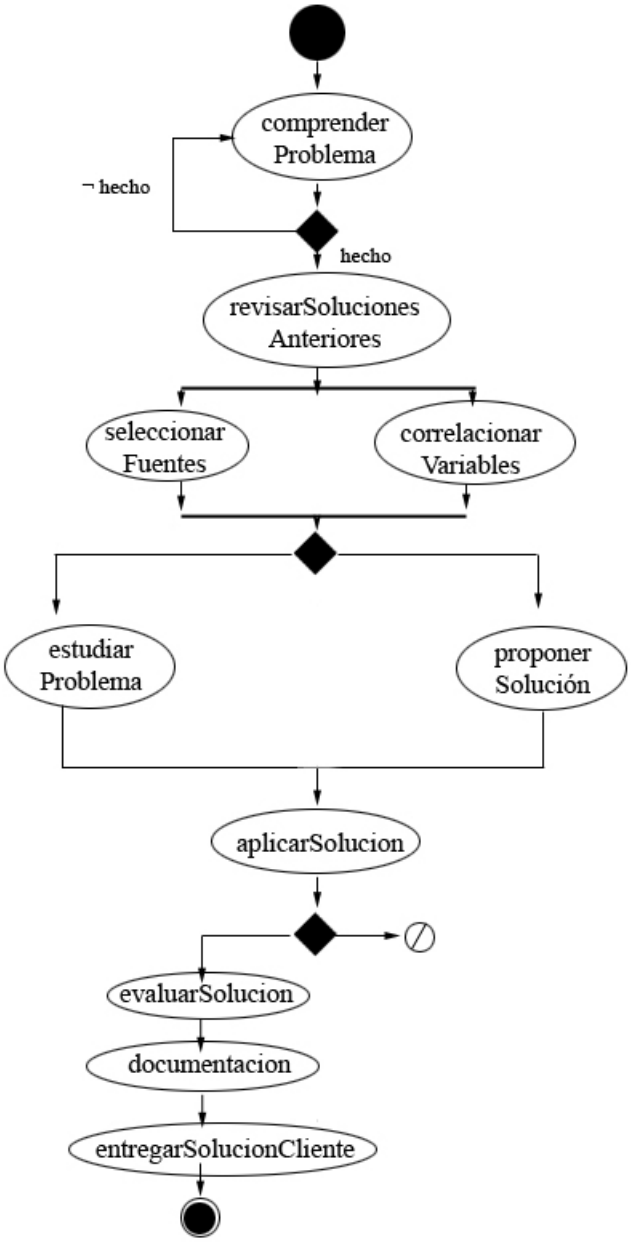

Figura 2.9: Proceso de solución de un problema creativo

2. Álgebra de tareas

Basados en las especificaciones del álgebra de tareas, el diagrama de tareas se traduce en la siguiente expresión algebraica:

{ μ.x (comprenderProblema ; ε +x); revisarSolucionesAnteriores; {seleccionarFuentes || correlacionarVariables}; {estudiarProblema+ proponerSolucion}; aplicarSolucion; {{evaluarSolucion; documentacion; entregarSolucionCliente}+φ}}

3. **Estructura de Kripke:** Anteriormente se mencionaba que para la comprobación de especificaciones se requiere modelar el sistema como una estructura de kripke y en el Método Discovery las trazas representan esta estructura. Para obtenerlas utilizaremos un traductor que ha sido elaborado por [9] correspondiente al Método Discovery relacionado con nuestro tema de investigación.

fromList *[[comprenderProblema, !, comprenderProblema, revisarSolucionesAnteriores, correlacionarVariables, seleccionarFuentes, !, estudiarProblema, aplicarSolucion, !, evaluarSolucion, documentacion, entregarSolucionCliente], [comprenderProblema, !, comprenderProblema, revisarSolucionesAnteriores, correlacionarVariables, seleccionarFuentes, !, estudiarProblema, aplicarSolucion, !, Phi], [comprenderProblema, !, comprenderProblema, revisarSolucionesAnteriores, correlacionarVariables, seleccionarFuentes, !, proponerSolucion, aplicarSolucion, !, evaluarSolucion, documentacion, entregarSolucionCliente], [comprenderProblema, !, comprenderProblema, revisarSolucionesAnteriores, correlacionarVariables, seleccionarFuentes, !, proponerSolucion, aplicarSolucion, !, Phi], [comprenderProblema, !, comprenderProblema, revisarSolucionesAnteriores, seleccionarFuentes, correlacionarVariables, !, estudiarProblema, aplicarSolucion, !, evaluarSolucion, documentacion, entregarSolucionCliente], [comprenderProblema, !, comprenderProblema, revisarSolucionesAnteriores, seleccionarFuentes, correlacionarVariables, !, estudiarProblema, aplicarSolucion, !, Phi], [comprenderProblema, !, comprenderProblema, revisarSolucionesAnteriores, seleccionarFuentes, correlacionarVariables, !, proponerSolucion, aplicarSolucion, !, evaluarSolucion, documentacion, entregarSolucionCliente], [comprenderProblema, !, comprenderProblema, revisarSolucionesAnteriores, seleccionarFuentes, correlacionarVariables, !, proponerSolucion, aplicarSolucion, !, Phi], [comprenderProblema, !, revisarSolucionesAnteriores, correlacionarVariables, seleccionarFuentes, !, estudiarProblema, aplicarSolucion, !, evaluarSolucion, documentacion, entregarSolucionCliente], [comprenderProblema, !, revisarSolucionesAnteriores, correlacionarVariables, seleccionarFuentes, !, estudiarProblema, aplicarSolucion, !, Phi], [comprenderProblema, !, revisarSolucionesAnteriores, correlacionarVariables, seleccionarFuentes, !, proponerSolucion, aplicarSolucion, !, evaluarSolucion, documentacion, entregarSolucionCliente], [comprenderProblema, !, revisarSolucionesAnteriores, correlacionarVariables, seleccionarFuentes, !, proponerSolucion, aplicarSolucion, !, Phi], [comprenderProblema, !, revisarSolucionesAnteriores, seleccionarFuentes, correlacionarVariables, !, estudiarProblema, aplicarSolucion, !, evaluarSolucion, documentacion, entregarSolucionCliente], [comprenderProblema, !, revisarSolucionesAnteriores, seleccionarFuentes, correlacionarVariables, !, estudiarProblema, aplicarSolucion, !, Phi], [comprenderProblema, !, revisarSolucionesAnteriores, seleccionarFuentes, correlacionarVariables, !, proponerSolucion, aplicarSolucion, !, evaluarSolucion, documentacion, entregarSolucionCliente], [comprenderProblema, !, revisarSolucionesAnteriores, seleccionarFuentes, correlacionarVariables,*

!, proponerSolucion, aplicarSolucion, !, Phi]]

4. **Lógica temporal LTL y CTL:** La verificación de propiedades es particular de cada sistema, por ejemplo para este caso podríamos revisar si el estado ***"entregarSolucionCliente"*** se mantiene en algún estado futuro. Esta condición puede expresarse con el operador modal Finally, y las expresiones respectivas son las siguientes:

   LTL: F ("entregarSolucionCliente")

   CTL: AF ("entregarSolucionCliente")

5. **Verificación:** En base a la vista de diagramas de tareas, podemos anticiparnos a estos dos resultados, en el primer caso la verificación devolverá *falso* debido a que la ejecución de la tarea ***entregarSolucionCliente*** no siempre se mantiene. En el caso de la verificación CTL devolverá un árbol con todos los posibles flujos que puedan conducir a la ejecución de esta tarea. A continuación se muestra la expresión de especificación y su resultado final.

   MyLTL: check "{ μ.x (comprenderProblema ; ε +x); revisarSolucionesAnteriores; {seleccionarFuentes || correlacionarVariables}; {estudiarProblema+ proponerSolucion}; aplicarSolucion; {{evaluarSolucion; documentacion; entregarSolucionCliente}+φ}}" F ("entregarSolucionCliente")

   MyCTL: check "{ μ.x (comprenderProblema ; ε +x); revisarSolucionesAnteriores; {seleccionarFuentes || correlacionarVariables}; {estudiarProblema+ proponerSolucion}; aplicarSolucion; {{evaluarSolucion; documentacion; entregarSolucionCliente}+φ}}" AF ("entregarSolucionCliente")

6. **Resultados:**

MyLTL: (False, [comprenderProblema, !, comprenderProblema, revisarSolucionesAnteriores, correlacionarvariables, seleccionarFuentes, !, estudiarProblema, aplicarSolucion, !, Phi])

MyCTL: ([[0], [0,1], [0,1,1], [0,1,1,1], [0,1,1,1,1], [0,1,1,1,1,1], [0,1,1,1,1,1,1], [0,1,1,1,1,1,1,1], [0,1,1,1,1,1,1,1,1], [0,1,1,1,1,1,1,1,1,1], [0,1,1,1,1,1,2], [0,1,1,1,1,1,2,1], [0,1,1,1,1,1,2,1,1], [0,1,1,1,1,1,2,1,1,1], [0,1,1,1,1,1,2,1,1,1,1], [0,1,1,1,2], [0,1,1,1,2,1], [0,1,1,1,2,1,1], [0,1,1,1,2,1,1,1], [0,1,1,1,2,1,1,1,1], [0,1,1,1,2,1,1,1,1,1], [0,1,1,1,2,1,1,1,1,1,1], [0,1,1,1,2,1,2], [0,1,1,1,2,1,2,1], [0,1,1,1,2,1,2,1,1], [0,1,1,1,2,1,2,1,1,1], [0,1,1,1,2,1,2,1,1,1,1], [0,1,2], [0,1,2,1], [0,1,2,1,1], [0,1,2,1,1,1], [0,1,2,1,1,1,1], [0,1,2,1,1,1,1,1], [0,1,2,1,1,1,1,1,1], [0,1,2,1,1,1,1,1,1,1], [0,1,2,1,1,2], [0,1,2,1,1,2,1], [0,1,2,1,1,2,1,1], [0,1,2,1,1,2,1,1,1], [0,1,2,1,1,2,1,1,1,1], [0,1,2,2], [0,1,2,2,1], [0,1,2,2,1,1], [0,1,2,2,1,1,1], [0,1,2,2,1,1,1,1], [0,1,2,2,1,1,1,1,1], [0,1,2,2,1,1,1,1,1,1], [0,1,2,2,1,2], [0,1,2,2,1,2,1], [0,1,2,2,1,2,1,1], [0,1,2,2,1,2,1,1,1], [0,1,2,2,1,2,1,1,1,1]], Node ([0], null) [Node ([0,1], comprenderProblema) [Node ([0,1,2], revisarSolucionesAnteriores) [Node ([0,1,2,2], seleccionarFuentes) [Node ([0,1,2,2,1], correlacionarvariables) [Node ([0,1,2,2,1,2], proponerSolucion) [Node ([0,1,2,2,1,2,1], aplicarSolucion) [Node ([0,1,2,2,1,2,1,2], Phi) [Empty], Node ([0,1,2,2,1,2,1,1], evaluarSolucion) [Node ([0,1,2,2,1,2,1,1,1], documentacion) [Node ([0,1,2,2,1,2,1,1,1,1], entregarSolucionCliente) [Empty]]]]], Node ([0,1,2,2,1,1], estudiarProblema) [Node ([0,1,2,2,1,1,1], aplicarSolucion) [Node ([0,1,2,2,1,1,1,2], Phi) [Empty], Node ([0,1,2,2,1,1,1,1], evaluarSolucion) [Node ([0,1,2,2,1,1,1,1,1], documentacion)

[Node ([0,1,2,2,1,1,1,1,1,1], entregarSolucionCliente) [Empty]]]]]]], Node ([0,1,2,1], correlacionarvariables) [Node ([0,1,2,1,1], seleccionarFuentes) [Node ([0,1,2,1,1,2], proponerSolucion) [Node ([0,1,2,1,1,2,1], aplicarSolucion) [Node ([0,1,2,1,1,2,1,2], Phi) [Empty], Node ([0,1,2,1,1,2,1,1], evaluarSolucion) [Node ([0,1,2,1,1,2,1,1,1], documentacion) [Node ([0,1,2,1,1,2,1,1,1,1], entregarSolucionCliente) [Empty]]]]], Node ([0,1,2,1,1,1], estudiarProblema) [Node ([0,1,2,1,1,1,1], aplicarSolucion) [Node ([0,1,2,1,1,1,1,2], Phi) [Empty], Node ([0,1,2,1,1,1,1,1], evaluarSolucion) [Node ([0,1,2,1,1,1,1,1,1], documentacion) [Node ([0,1,2,1,1,1,1,1,1,1], entregarSolucionCliente) [Empty]]]]]]]], Node ([0,1,1], comprenderProblema) [Node ([0,1,1,1], revisarSolucionesAnteriores) [Node ([0,1,1,1,2], seleccionarFuentes) [Node ([0,1,1,1,2,1], correlacionarvariables) [Node ([0,1,1,1,2,1,2], proponerSolucion) [Node ([0,1,1,1,2,1,2,1], aplicarSolucion) [Node ([0,1,1,1,2,1,2,1,2], Phi) [Empty], Node ([0,1,1,1,2,1,2,1,1], evaluarSolucion) [Node ([0,1,1,1,2,1,2,1,1,1], documentacion) [Node ([0,1,1,1,2,1,2,1,1,1,1], entregarSolucionCliente) [Empty]]]]], Node ([0,1,1,1,2,1,1], estudiarProblema) [Node ([0,1,1,1,2,1,1,1], aplicarSolucion) [Node ([0,1,1,1,2,1,1,1,2], Phi) [Empty], Node ([0,1,1,1,2,1,1,1,1], evaluarSolucion) [Node ([0,1,1,1,2,1,1,1,1,1], documentacion) [Node ([0,1,1,1,2,1,1,1,1,1,1], entregarSolucionCliente) [Empty]]]]]]], Node ([0,1,1,1,1], correlacionarvariables) [Node ([0,1,1,1,1,1], seleccionarFuentes) [Node ([0,1,1,1,1,1,2], proponerSolucion) [Node ([0,1,1,1,1,1,2,1], aplicarSolucion) [Node ([0,1,1,1,1,1,2,1,2], Phi) [Empty], Node ([0,1,1,1,1,1,2,1,1], evaluarSolucion) [Node ([0,1,1,1,1,1,2,1,1,1], documentacion) [Node ([0,1,1,1,1,1,2,1,1,1,1], entregarSolucionCliente) [Empty]]]]], Node ([0,1,1,1,1,1,1], estudiarProblema) [Node ([0,1,1,1,1,1,1,1], aplicarSolucion) [Node ([0,1,1,1,1,1,1,1,2], Phi) [Empty], Node ([0,1,1,1,1,1,1,1,1], evaluarSolucion) [Node ([0,1,1,1,1,1,1,1,1,1], documentacion) [Node ([0,1,1,1,1,1,1,1,1,1,1], entregarSolucionCliente) [Empty]]]]]]]]]])

Ahora que tenemos un esquema ilustrativo del proceso de comprobación de especificaciones en el Método Discovery a través de las lógicas LTL y CTL, es más fácil comprender el problema de análisis al cual se enfrentaría un usuario, por ello nuestros objetivos próximos se orientan al desarrollo de las interfaces de verificación de tareas que simplifiquen esta actividad. Para continuar analicemos el enfoque de un compilador, el cual será útil para la construcción del verificador sintáctico del álgebra de tareas y expresiones lógicas.

2.3. Compiladores

De acuerdo con la definición de [39] *"un compilador es un programa que lee un programa escrito en un lenguaje fuente y lo traduce a un programa equivalente en otro lenguaje , denominado objeto*" (ver figura 2.10).

La definición parece en realidad bastante sencilla, sin embargo en la práctica; la construcción de compiladores es una de las actividades más complejas debido a la gran cantidad de fases y requerimientos que deben considerarse. Escribir un programa de esta naturaleza o incluso comprenderlo no es una tarea fácil; [6] menciona que un compilador puede varía entre 10, 000 a 1,000,000 líneas de código. No obstante, los compiladores se utilizan en casi todas las áreas de la computación, especialmente en el desarrollo de programas de interfaces e intérpretes de comandos [6].

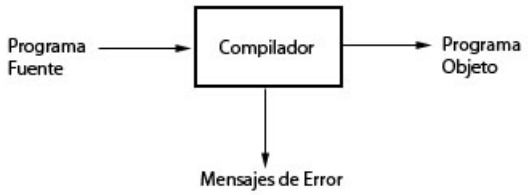

Figura 2.10: Esquema general de un compilador

Hoy en día aun cuando la mayoría de compiladores se escriben en lenguajes de alto nivel, las técnicas básicas de diseño e implementación iniciales siguen siendo las mismas; por ello el conocimiento y estudio de estos principios teóricos es importante para el presente proyecto de investigación.

2.3.1. Tipos de Compilación

Los sistemas de compilación desde una perspectiva simplificada pueden clasificarse de acuerdo con los diferentes tipos de código y las diversas formas de funcionamiento. Según el autor [37] los sistemas de compilación se clasifican de la siguiente manera:

- **Ensamblador:** Se considera la forma de representación intermedia más cercana al código objeto final (código máquina), utiliza abreviaturas mnemónicas como nombres simbólicos para representar operaciones máquina y direcciones del programa. Los ensambladores traducen programas escritos en lenguaje ensamblador a código máquina.
- **Compilador:** Los compiladores traducen programas escritos en lenguaje de alto nivel a un código intermedio o a un código máquina.
- **Intérprete:** Considerados un caso particular, debido a que analizan y ejecutan directamente cada proposición del programa de entrada. Particularmente este tipo de compilador se considera independiente de la máquina debido a que no generan código objeto .
- **Pre procesador:** Es un programa específico encargado de recopilar algunos recursos requeridos para la generación de código objeto: sustitución de macros, la inclusión de archivos o la extensión del lenguaje.

En este punto podemos notar que el sistema que se desea desarrollar para la verificación de diagramas de flujo tareas se orienta sobre el tipo de compilación **"compilador"**, el cual debe recibir como entrada un modelo de alto nivel (especificación algebraica) y transformarlo a un conjunto de trazas/caminos (código intermedio). Para comprender lo que sucede en la parte intermedia analicemos las fases de un compilador.

2.3.2. Fases de un Compilador

La compilación es un proceso que consiste en analizar y sintetizar la estructura de un programa, determinar el significado del código y traducirlo en código máquina equivalente. El proceso de compilación de un programa fuente generalmente suele dividirse en dos etapas: análisis y síntesis. En el análisis se dividen los elementos componentes de la entrada fuente y se construye una representación intermedia del programa; en la siguiente fase se construye el programa objeto final a partir de la representación intermedia.

Clasificar a los compiladores en dos fases no es suficiente para explicar el proceso de compilación, y es que en realidad es aún más complejo, [39],[38],[37] y [27] concuerdan que las tareas o fases principales de un compilador son las siguientes (ver figura2.11) :

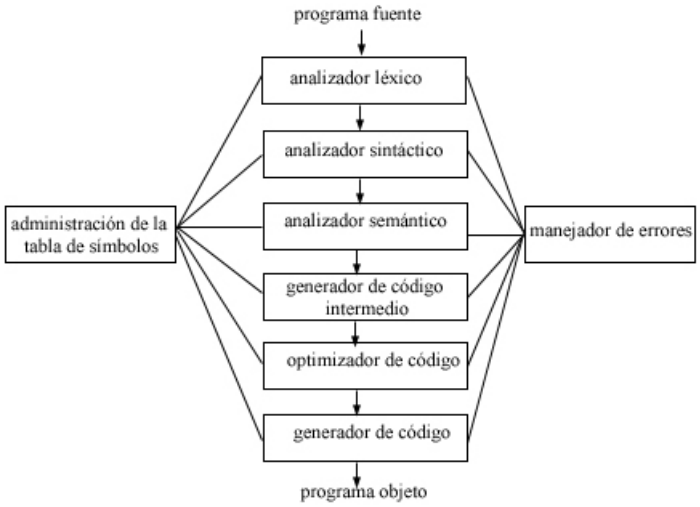

Figura 2.11: Fases de un compilador [39]

Análisis Léxico: también denominado análisis lineal, scaner o de exploración, en esta fase se obtiene el contenido del programa fuente y se agrupa en componentes léxicos (símbolos) para pasarlos a los analizadores sintácticos y semánticos.

Análisis Sintáctico: también denominado análisis jerárquico o parser. La función de esta fase, es tomar la cadena producida por el explorador, y utilizando un algoritmo analizador o reconocedor (parser), determinar si es o no una cadena válida, es decir, sintácticamente correcta de acuerdo a la gramática del lenguaje utilizado. Por lo general, la verificación sintáctica se facilita representando las frases gramaticales del programa fuente mediante un árbol de análisis sintáctico como el que se muestra en la figura 2.12

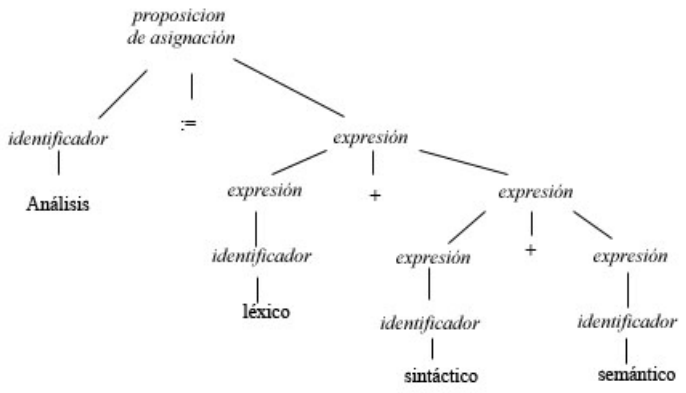

Figura 2.12: Árbol de análisis sintáctico para Análisis:= léxico + sintáctico +semántico

Análisis Semántico: en esta fase se revisa el programa fuente para tratar de encontrar errores de significado (semántica) y se reúne la información sobre los tipos para la fase posterior de generación de código. Es común que en esta fase se utilice la estructura jerárquica determinada por la fase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

Generación de Código Intermedio: traduce el programa de entrada a una representación intermedia explicita para una máquina abstracta. Esta representación intermedia más tarde se traducirá en el código objeto final, por lo cual debe tener dos propiedades importantes: fácil de producir y fácil de traducir al programa objeto.

Optimizador de código: su función es mejorar el código intermedio, de manera que pueda generarse un código de máquina más rápido de ejecutar.

Generación de Código: es la fase final del compilador y su función es generar un código objeto comúnmente en código máquina relocalizable o ensamblador.

Analizando el proceso de compilación presentado en la figura 2.11 podemos notar que a lo largo de la ejecución las etapas administran una estructura de datos en particular: La tabla de símbolos. La tabla de símbolos es una estructura de datos que contiene información acerca de los identificadores y procedimientos utilizados en el programa fuente. La estructura permite encontrar rápidamente información acerca de la memoria asignada a un identificador, su tipo, su ámbito y en el caso de procedimientos el número, tipo de argumentos, método de envió y el tipo que devuelve [39].

A lo largo de cada una de las etapas del compilador, los errores pueden presentarse y un compilador que se detiene por un error no es útil. Para ello es necesario que cada fase de compilación administre los errores de manera que no interrumpan el proceso de compilación.

Un aspecto clave en el proceso de compilación es el algoritmo de reconocimiento debido a que es la estructura que determina si las entradas corresponden al ámbito del lenguaje del

compilador. Para entender el ámbito del lenguaje de un compilador es necesario estudiar algunos aspectos formales descritos a continuación.

2.3.3. Aspectos Formales: Lenguajes Formales

2.3.3.1. Forma de Backus-Naur (BNF)

Antes de realizar la compilación de un programa es necesario conocer la definición del lenguaje en el que ha sido escrito y cuando hablamos de definición nos referimos a los aspectos de sintaxis y semántica. Podemos especificar la definición de un lenguaje en particular de manera formal utilizando la forma de Backus-Naur (BNF). La forma de Backus-Naur es un metalenguaje y fue creada para definir la estructura sintáctica de ALGOL60 (ver cuadro 2.1)[37].

símbolo	*significado*
$\rightarrow$	"se define como" fin de definición
|	or, alternativa
[X]	una o ninguna ocurrencia de x
{X}	numero arbitrario de ocurrencias de x (0,1,2,..)
{X | Y}	selección: x,y

Cuadro 2.1: Notación BNF

2.3.3.2. Gramáticas

Una de las formas más consistentes de definir el lenguaje de un compilador se denomina gramática. Una gramática G consiste en un conjunto finito no vacío de reglas o producciones en la cual se específica la sintaxis de un lenguaje [38]. Si retrocedemos a la forma BNF anterior podemos notar que las reglas de la notación definen la sintaxis del lenguaje particular.

De manera formal podemos expresar el concepto de gramática de la siguiente manera [38]:

Una gramática G se define como una 4-tupla G (T,N,P,S) donde:

- **T** es el conjunto de símbolos terminales.
- **N** es el conjunto de símbolos no terminales.
- **P** es el conjunto de producciones.
- **S** $\in$ N es el símbolo inicial

En este ámbito el lenguaje formal L que genera una gramática está definido por **L (G) = L (N,T,P,S)**

2.3.3.3. Clasificación de las Gramáticas

De acuerdo a su complejidad, las gramáticas se clasifican en 4 tipos: sin restricciones, dependientes del contexto, independientes del contexto y regulares[38]. Para cada una su especificación es la siguiente:

1. **Sin restricciones:** como su nombre lo indica no existen restricciones para la definición de sus reglas.
2. **Dependientes/Sensibles al contexto:** una gramática sensible al contexto contiene únicamente producciones de la forma: $\alpha \rightarrow \beta$, donde $|\alpha| \leq |\beta|$, donde $|\alpha|$ denota el tamaño de α.
3. **Independiente/Libre del contexto:** una gramática independiente del contexto contiene unicamente producciones de la forma: $\alpha \rightarrow \beta$, donde $|\alpha| \leq |\beta|$ y $\alpha \in$ N.
4. **Regular:** una gramática regular contiene únicamente producciones de la forma: $\alpha \rightarrow \beta$ donde $|\alpha| \leq |\beta|$, $\alpha \in$ N y β tiene la forma aB o a, donde a $\in$ T y B $\in$ N.

De la lista anterior la forma más común y deseable en un lenguaje son las gramáticas libres de contexto. Las gramáticas libres del contexto son muy importantes en la teoría de los lenguajes de programación, ya que los lenguajes que definen poseen una estructura muy sencilla (ver figura 2.13). Sin embargo, los lenguajes de programación típicos consisten en propiedades que no pueden expresarse con gramáticas independientes del contexto, debido a que deben cumplirse propiedades de tipos compatibles y tener declaraciones previas antes de utilizarse[37].

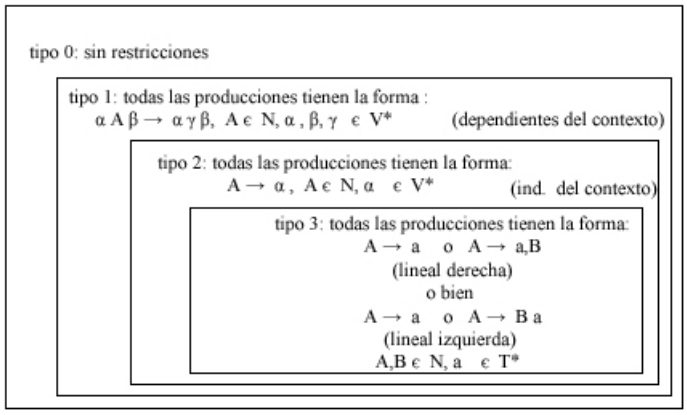

Figura 2.13: Jerarquía de Chomsky[37]

Otra de las clasificaciones comunes para las gramáticas está determinada por las reglas de producción. Existen dos tipos básicos: Lineales izquierda y Lineales Derecha.

- Una gramática G es lineal izquierda si cada producción P tiene la forma: A$\rightarrow$ Ba o A $\rightarrow$ a ; donde A y B pertenecen a N y a esta en T*.
- Una gramática G es lineal derecha o regular si cada producción P tiene la forma: A$\rightarrow$ aB o A $\rightarrow$ a ; donde A y B pertenecen a N y a esta en T*.

Ahora que hemos estudiado el concepto de linealidad y especificaciones gramaticales es importante indicar que siempre que se construyan gramáticas en particular libre de contexto deben evitarse dos características: ambigüedad y recursividad [39].

Una gramática independiente del contexto es no ambigua si y sólo si hay una sola derivación por la derecha (o por la izquierda) y por ende generan un árbol de análisis sintáctico único para

cada frase que pueda derivarse con las producciones de la gramática; en caso contrario la gramática se le conoce como ambigua [37].

Con respecto a la recursividad, una gramática G libre de contexto con al menos un no terminal recursivo X (derecho o izquierdo) se conoce como gramática recursiva izquierda o derecha si todos los símbolos no terminales (con excepción del inicial) son recursivos. En este ámbito existen dos tipos de recursividad: inmediata e indirecta [6].

- La recursividad inmediata izquierda se genera si la gramática G tiene un no terminal A tal que exista una producción de la forma: A$\rightarrow$Aα.
- La recursividad indirecta se origina en producciones subsecuentes que re-invocan reglas que generan recursividad directa, por ejemplo:
  - S$\rightarrow$ Aa | b
  - A$\rightarrow$ Ac | Sd | ϵ, donde la producción A$\rightarrow$ Sd es recursiva por S$\rightarrow$Aa.

La recursión es una característica no deseable en las especificaciones y es que muchos algoritmos de análisis sintáctico no funcionan con gramáticas recursivas izquierdas, debido a que se presentan riesgos de lazos infinitos. más adelante se definirán algunas técnicas para eliminar recursividad en gramáticas.

2.3.3.4. Árboles de Análisis sintáctico

Uno de los problemas más comunes en la comprobación de símbolos pertenecientes a un lenguaje es el análisis sintáctico. El Análisis sintáctico consiste en encontrar de qué manera se puede derivar una secuencia de símbolos a partir del inicial mediante las producciones gramaticales [37].

Por ejemplo, considerando una gramática G (T,N,P,S0) que acepta expresiones aritméticas tales como: x+y-x*y su definición seria la siguiente:
T={x,y,+,-,*,/, (,)}
N={EXPR, TERM, FACTOR}
P={EXPR $\rightarrow$ TERM | EXPR + TERM | EXPR - TERM
TERM $\rightarrow$FACTOR | TERM * FACTOR | TERM / FACTOR
FACTOR $\rightarrow$ X| Y | (EXPR)
}
S0= {EXPR}

Los proceso de derivación izquierda y derecha se realizan de la siguiente manera :
Derivación Izquierda
EXPR
$\rightarrow$EXPR - TERM
$\rightarrow$EXPR + TERM - TERM
$\rightarrow$TERM + TERM - TERM

→FACTOR + TERM -TERM
→**x**+ TERM -TERM
→**x** + FACTOR - TERM
→**x** + **y** - TERM
→**x + y -** TERM * FACTOR
→**x + y -** FACTOR * FACTOR
→**x + y - x** *FACTOR
→**x + y - x * y**

Derivación Derecha
EXPR
→EXPR - TERM
→EXPR - TERM * FACTOR
→EXPR - TERM * **y**
→EXPR - FACTOR * **y**
→EXPR - **x** * **y**
→EXPR + TERM - **x** * **y**
→EXPR + FACTOR - **x** * **y**
→EXPR + **y** - **x** * **y**
→TERM + **y** - **x** * **y**
→FACTOR + **y** - **x** * **y**
→**x + y - x * y**

Basados en el proceso de derivación en cualquiera de los sentidos, el árbol de análisis sintáctico para la expresión x+y-x*y se ilustra en la figura 2.14.

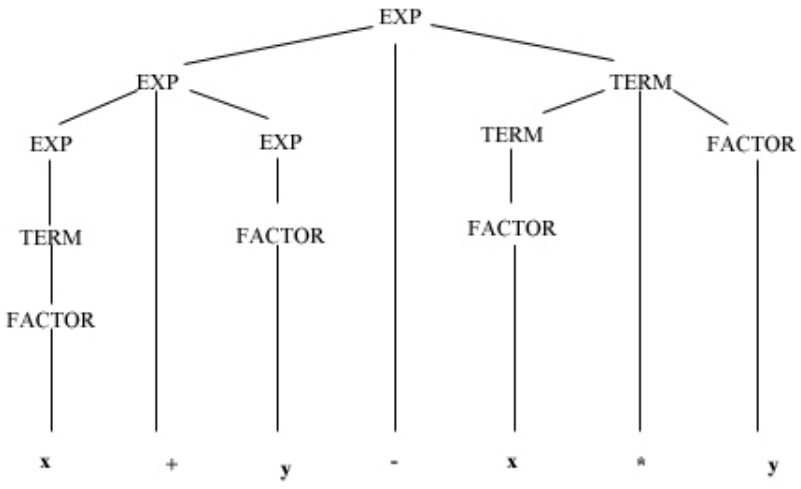

Figura 2.14: Árbol de análisis sintáctico para la expresión x+y-x*y

Los árboles sintácticos son muy útiles para determinar si una gramática G es ambigua o no,

esto puede verificarse construyendo los árboles correspondientes para una frase de un lenguaje dado a partir de ambas derivaciones. Si las derivaciones permiten construir dos árboles sintácticos diferentes se dice que la gramática G es ambigua. La ambigüedad en las gramáticas es un problema que puede generar múltiples interpretaciones de una frase y con ello diferentes resultados en el tratamiento de las expresiones, lo cual es problemático con respecto a los lenguajes de programación *[39].*

2.3.3.5. Técnicas de análisis sintáctico

El análisis sintáctico es el proceso a través del cual se determina el árbol sintáctico correspondiente a una frase dada, de acuerdo a las gramáticas definidas del lenguaje de programación específico. Existen dos métodos que permiten realizar este análisis: *Análisis sintáctico descendente y ascendente [37].*

El **análisis sintáctico descendente** puede definirse como la construcción del árbol partiendo de la raíz hacia las hojas utilizando una derivación que genere un no terminal derecha o izquierda consecuente. Este tipo de análisis crece de arriba hacia abajo a medida que se lee la frase de entrada de izquierda a derecha.

El **análisis sintáctico ascendente** parte de las hojas del árbol sintáctico y asciende a la raíz continuando su derivación hacia el no terminal superior candidato. La construcción en este caso asciende hacia la raíz por ello se denomina ascendente.

En la construcción del árbol sintáctico utilizando análisis descendente y ascendente existe un problema particular: no siempre es posible conocer la frase de entrada por lo que un pre-análisis no siempre estará disponible. El problema de la elección del estado próximo a partir de un símbolo inicial puede bloquear el proceso de derivación e impedir el procesamiento de la cadena de entrada. A este problema se le conoce como bloqueo mutuo o abrazo mortal. Sin embargo este problema se soluciona planteando un conjunto probable de símbolos siguientes a partir de un inicial que no genere un abrazo mortal.

2.3.4. Técnicas para eliminar recursividad y Ambigüedad en gramáticas

La recursividad en gramáticas es una propiedad no deseable, para eliminar la recursión por la izquierda inmediata simple [6] propone el siguiente procedimiento:

Una Gramática Libre de Contexto es recursiva por la izquierda si tiene un no terminal A tal que existe una producción de la forma:
A$\rightarrow$ A α

Algoritmo para eliminar recursividad izquierda:

1. Se agrupan todas las producciones de A en la forma:
   A→Aα_1|Aα_2|...|Aα_m|β_1|β_2|...|β_n
   donde ninguna β_i comienza con A.
2. Se sustituyen las producciones de A por:
   A→β_1A'| β_2A'|...|β_nA'
   A'→α_1A' | α_{12}A' | ... | α_mA' | ϵ

El procedimiento de eliminación de recursividad inmediata simple por la izquierda elimina la recursión inmediata por la izquierda de las producciones A y A', pero no elimina la recursión por la izquierda que incluya derivaciones de dos o más pasos (indirecta). Por ejemplo, considere la gramática:

- S→ Aa | b
- A→ Ac | Sd | ϵ

El no terminal S es recursivo por la izquierda, porque S → Aa → Sda, pero no es recursivo inmediato por la izquierda. El algoritmo de eliminación inmediata funciona si la gramática no tiene ciclos o producciones ϵ . Los ciclos, al igual que las producciones ϵ, se pueden eliminar sistemáticamente bajo el siguiente algoritmo[39].

1. Para eliminar la recursividad indirecta se debe encontrar el elemento conflictivo y sustituirlo por su definición.
   Dada la siguiente gramática con recursividad indirecta G:
   S→ Aa | b
   A→ Ac | Sd | ϵ
   De la gramática anterior, sustituimos:
   S→ Aa | b
   A→ Ac | Aad |b d| ϵ
2. Se elimina la recursividad indirecta de las producciones utilizando el algoritmo inmediato:
   S→ Aa | b
   A→ b d A' | A'
   A'→ c A' | a d A' | ϵ

En el caso de la ambigüedad en las gramáticas existen 3 formas de influir en las reglas de producción para solucionar el problema [23], estas técnicas son las siguientes :

1. **Refactorización izquierda:** Si para una misma producción existen dos alternativas con el mismo inicio, se toma la parte en común y se reestructura para evitar que exista un problema de decisión en el proceso de análisis sintáctico:
   R: L+k | L+M ; → es equivalente a R: L+ (K|M)
   .....
   a: b | c;

b: L + K ;

c: L+M;

Es equivalente a:

R: L + (b|c);

b: K;

c: M;

. . . . . .

2. **Predicados sintácticos:** Los predicados sintácticos plantean que antes de decidir sobre dos reglas de producción es conveniente comprobar si la entrada se ajusta a las siguientes reglas. Si es así se debe tomar el camino indicado de otra forma se debe elegir la siguiente alternativa. Un ejemplo de esta especificación es la siguiente:

   a: (LK) => b | c

   b: LK;

   c: LM;

3. **Retroceso o Backtracking:** Esta técnica se específica al momento de ejecución del compilador y se basa en el hecho de que si una elección en las reglas de producción es incorrecta la elección debe retroceder y probar la siguiente alternativa.

2.3.5. Análisis Léxico

Al inicio de un proceso de compilación, el código fuente de un programa carece de significado y puede conceptualizarse únicamente como un flujo de caracteres. Para descifrar su significado y reglas de estructura es necesario aplicar un proceso de análisis que permita reconocer símbolos en el flujo de caracteres y presentarlos en una representación más útil para el análisis sintáctico (ver figura 2.15). El análisis que realiza esta tarea se denomina analizador léxico y sus tareas específicas son las siguientes [39]:

- Eliminar espacios y comentarios.
- Reconocer identificadores y palabras claves.
- Reconocer constantes y numerales.
- Generar un listado para el compilador.

Los analizadores léxicos deben reconocer componentes léxicos en un flujo de caracteres y estos componentes son elementos de un lenguaje regular, es decir, pueden ser generados por una gramática regular . El proceso de reconocimiento de componentes lee una frase de izquierda a derecha y la acepta como elemento del lenguaje si se puede reducir al símbolo inicial de la gramática. Por ejemplo, considere la gramática siguiente [37]:

T= {a,b}

N={A,B,C}

P= {A$\rightarrow$Aa | Ba , B $\rightarrow$ Cb , C $\rightarrow$ Ca | a }

S={A}

Genera el lenguaje :

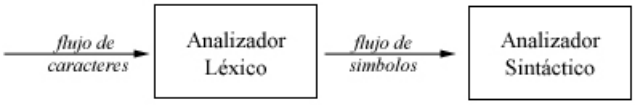

Figura 2.15: Proceso simplificado del análisis léxico

L (G) = {a^m b a^n | m,n >=1 }

El proceso de reconocimiento de la frase: aaabaa, la cual debería ser válida según la especificación del lenguaje es el siguiente:

1. Reducción: **a** *a a b a a*
2. Reducción: **C a** *a b a a*
3. Reducción: **C a** *b a a*
4. Reducción: **C b** *a a*
5. Reducción: **B a** *a*
6. Reducción: **A a**
7. Reducción Final: **A**

El resultado final es el símbolo inicial, por lo que la frase es aceptada por el lenguaje.

El proceso de análisis léxico o de reconocimiento puede entenderse como la transformación de un flujo de caracteres en un flujo de símbolos reducido; es decir, un conjunto filtrado para eliminar elementos del texto que solo sirvan para hacer legible el programa. Dentro de los elementos que hacen legible un programa están: espacios, tabuladores, comentarios y saltos de línea (separadores) [6].

Dentro de un programa fuente es fácil identificar elementos como números, identificadores y palabras reservadas, en las cuales el reconocimiento se basa en autómatas finitos. Pero la cuestión real es saber distinguir entre u identificador y una palabra reservada, para ello es necesario comparar las entradas con una tabla de palabras reservadas del lenguaje, comúnmente esta estructura es la tabla de símbolos.

2.3.6. Análisis Sintáctico

En el proceso de compilación el primer análisis consiste en obtener los componentes léxicos del programa de entrada y como segunda etapa se determina si son aceptados o no por el lenguaje de programación especificado; esta tarea se realiza por el analizador sintáctico. Existen dos métodos para realizar la tarea del analizador sintáctico: Análisis sintáctico descendente y ascendente, los cuales se basan en gramáticas LL y LR ambas independientes del contexto [37].

2.3.6.1. Gramáticas LL

En el proceso de construcción del árbol sintáctico a medida que se evoluciona en su construcción es posible elegir una derivación incorrecta y ello implica que la gramática presenta imprecisión. Debido a que pueden existir diferentes alternativas de decisión en cada una de las etapas del proceso de derivación, es necesario aplicar la teoría de agrupación de conjuntos siguientes [39]. Planteando la siguiente regla podemos aclarar este concepto:

Para cada producción de la forma: A $\rightarrow \sigma_1$ / σ_2 / ... /σ_n siempre debe ser posible elegir la alternativa correcta para la generación de un árbol de análisis sintáctico, para lograrlo es necesario conocer la siguiente información:

1. El conjunto de todos los símbolos terminales que pueden aparecer al principio de una frase que puede derivarse de una secuencia arbitraria de símbolos.
2. El conjunto de todos los símbolos terminales que pueden aparecer después de uno no terminal.

es decir,, debe ser posible conocer el conjunto PRIMERO y SIGUIENTE de la derivación para una gramática G (N, T, P, S). El conjunto de terminales que pueden aparecer al principio de cualquier frase derivable de una secuencia arbitraria de símbolos $\alpha \in (N \bigcup T)^*$ como:

PRIMERO (α) = { t | t $\in T_\varepsilon$ ^$\alpha \rightarrow^*$ tα'} ..... donde $T_\varepsilon = T \bigcup \{\varepsilon\}$.

Sea X un símbolo no terminal. Entonces, SIGUIENTE (X) es el conjunto de todos los símbolos terminales que pueden aparecer inmediatamente a la derecha de X:

SIGUIENTE (X) = {t | t $\in T_\varepsilon$ ^ S$\rightarrow^*$ α X t β}

Con ayuda del concepto anterior podemos definir una gramática independiente del contexto LL (1) si para cada producción de la forma:

1. A $\rightarrow \sigma_1$ / σ_2 / ... /σ_n se requiere que PRIMERO (σ_i) $\bigcap$ PRIMERO (σ_j) = $\emptyset$ $\forall$ i $\neq$ j
2. Si puede derivarse la cadena vacía ε de un símbolo no terminal X, se requiere que: PRIMERO (X) $\bigcap$ PRIMERO (X) = $\emptyset$

La gramática LL (1) significa que la entrada se lee desde la izquierda y que además utiliza una derivación por la izquierda con una anticipación de 1 carácter en cualquier paso del proceso de análisis sintáctico es decir, un símbolo de pre-análisis. Con ayuda de las premisas anteriores se puede decir que para una cadena de entrada dada existe solo una producción posible en un cierto estado de la derivación.

2.3.6.2. Análisis descendente recursivo

El análisis descendente se puede considerar como una forma de encontrar una derivación por la izquierda para una entrada dada, de esta forma se construye un árbol sintáctico desde la parte superior hasta las hojas. Así, al analizar una entrada, un analizador sintáctico descendente comienza por el axioma de la gramática y mientras no haya hojas no terminales se seleccionará una de las producciones pertenecientes a estas hojas para generar los hijos de la hoja no terminal, de acuerdo con el lado derecho de la producción seleccionada. Al final, una entrada es correcta si la secuencia de hojas terminales generadas coincide con la secuencia de símbolos de entrada [37].

Este método de análisis descendente recursivo se rige por las condiciones de una gramática LL (1), las cuales garantizan que no se presentaran bloqueos mutuos y por ello no es necesario retroceder para buscar un error de selección. Aun cuando LL (1) garantiza el no-bloqueo, también puede realizarse un análisis descendente en gramáticas que no sean LL (1) sin embargo, por cada paso, se corre el riesgo de elegir la producción incorrecta y en el momento en que no exista correspondencia con el símbolo terminal es necesario retroceder en la producción y seleccionar la producción alternativa que puede o no garantizar la nueva elección.

La forma más común de implantar un analizador sintáctico descendente recursivo es asociando un procedimiento a cada no terminal de la gramática. Por ejemplo, suponga que G (T,N,P,S) es una gramática LL (1), donde N={N_0,N_1,N_2,...N_m}, S=N_0 . Los no terminales corresponden a las categorías sintácticas que deberían reconocerse. Al usar una gramática LL (1), el símbolo de búsqueda por anticipación determina exactamente cuál es el procedimiento llamado para cada no terminal y, de esta manera, no se requieren retrocesos. El procedimiento general se describe de la siguiente manera:

PROGRAMA Analizador:
..**PROCEDIMIENTO** Error (..);
..**PROCEDIMIENTO N0;**.....;
..**PROCEDIMIENTO N1;**.....;
. . .
..**PROCEDIMIENTO Nm;**.....;

INICIO
Leer_Símbolo;
N0;
FIN.

2.3.6.3. Gramáticas LR (k)

Las gramáticas LR (k) independientes del contexto por definición, se utilizan para realizar análisis sintácticos de forma ascendente; y en general se basan en reglas que especifican la lectura de la entrada de izquierda a derecha y por cada paso de derivación utilizan una derivación

derecha. Al igual que las gramáticas LL (1) se basan en el pre-análisis, pero en este caso utilizan un pre-análisis de k elementos de anticipación [37].

Dentro de las gramáticas LR (k) el concepto base es el mando, el cual específica que una subcadena de una cadena se denomina mando si puede reducirse utilizando el lado izquierdo de una producción apropiada, siempre y cuando la reducción corresponda a un paso en la reducción por la izquierda de la cadena al símbolo inicial de la gramática. De manera formal , sea G (N,T,P,S) una gramática independiente del contexto , suponiendo que:

S$\rightarrow$* α X t $\rightarrow$ α β t : es una derivación por la derecha (donde t $\in$ T*). Entonces, β puede definirse como un mando de α β t.

De esta manera se puede aclarar que una subcadena β de una cadena α β t es un mando si:

$\alpha\beta$ t $\leftarrow$ α X t : es una reducción por la izquierda.

El proceso para aplicar un análisis sintáctico ascendente consiste en leer el carácter más a la izquierda de la entrada y se desplaza a una pila, a continuación se verifica si cumple la propiedad de mando, si la cumple se reduce el mando a un no terminal (es decir, el mando se reemplaza por el lado izquierdo de una producción apropiada).A lo largo de este procedimiento podemos notar que por cada reducción de pila el árbol sintáctico crece una hoja arriba , también se concluye que para aplicar este procedimiento se requieren dos operaciones: desplazamiento y reducción en conjunto con la estructura de datos pila.

2.3.7. Análisis semántico

En apartados anteriores hemos mencionado dos tipos de analizadores: léxico y sintáctico, sin embargo para cumplir con la especificación de un compilador, es importante verificar el significado de los programas, es decir,, realizar un análisis semántico. El análisis semántico verifica que los componentes pertenezcan al contexto del lenguaje de programación especificado y que todas las reglas dependientes del contexto sean consideradas [39].

En esta fase de análisis se utiliza la información del análisis sintáctico en combinación con las reglas semánticas del lenguaje para generar una representación intermedia del código fuente que se va a compilar. Esta representación interna es un código intermedio que se envía al generador de código. El código intermedio es una estructura de código cuya complejidad esta entre el código fuente de alto nivel y el código máquina; por ello pueden considerarse como una interfaz entre el generador de código y las fases previas del compilador.

2.3.8. Manejo de Errores

En el diseño de un compilador uno de los aspectos de especial interés es el manejo de errores, el cual es un elemento que debe estar considerado desde la conceptualización del sistema. Todo

desarrollador y usuario en particular le interesa que se le indique si es incorrecta la forma sintáctica en que implementa un programa y la forma de solucionarlo [6, 39]. De hecho en la teoría de compiladores es indispensable lo siguiente:

- Un compilador debe ser capaz de detectar errores en la entrada.
- Es importante que el compilador logre recuperarse de los errores sin perder demasiada información
- El compilador debe producir mensajes de error que permita al programador encontrar y corregir fácilmente los elementos incorrectos de su programa.

Dentro de los tipos de errores más comunes en el desarrollo de programas podemos encontrar errores invisibles y visibles. Los errores invisibles son errores que no pueden detectarse por el compilador, son resultado de decisiones erróneas durante el proceso de especificación o de la mala formulación de algoritmos. Por el contrario, los errores visibles son los que detecta el compilador y se presentan debido a errores de ortografía y por omitir requisitos formales del lenguaje de programación.

Hasta este punto se han considerado alternativas y técnicas para elaborar el compilador que dará soporte a nuestra interfaz de verificación de diagramas de flujo tareas; la cual debe operar bajo estas tres características. Ahora es necesario comenzar a conceptualizar la forma en que el usuario se comunicará con el sistema, es decir, el componente interfaz.

2.4. Desarrollo de Interfaces de Usuario

El proceso de desarrollo de software es un actividad compleja, intelectual y creativa que depende de las personas que toman decisiones. Muchas organizaciones han desarrollado sus propias metodologías que se ajustan a las magnitudes de los desarrollos, los requerimientos del cliente y a proyectos que dependen del tiempo.

El objetivo principal de un desarrollo de software es satisfacer las necesidades de los clientes ayudando a mejorar la calidad de un producto/servicio, mejorando la productividad y facilitando el desarrollo de las tareas en la empresa. Sin embargo antes de centrarnos en la perspectiva de los clientes debemos involucrar a los verdaderos usuarios del sistema que en muchos casos no son quienes adquieren el producto de software [19].

Después de que se reúnen los requerimientos definidos por los involucrados, un problema común en la etapa de diseño es el entendimiento de los requerimientos, por ejemplo: los ingenieros de software y diseñadores no siempre entienden el mismo concepto del sistema deseado, pero el punto crítico de muchos equipos de trabajo es el hecho de que en ocasiones el papel de los diseñadores es adoptado por los ingenieros de software lo cual podría disminuir algunas propiedades del sistema [36].

Es importante que siempre que se diseñe un nuevo sistema todos deben conceptualizarlo de la misma manera. Un componente global que requiere este principio es la conceptualización de la

interfaz gráfica, la cual es el componente que muchas veces determina la aceptación del cliente. Siempre que diseñe una interfaz gráfica el autor [31] sugiere analizar las siguientes preguntas: ¿Que sucede cuando se presentan actividades de diseño de interfaces?, ¿Cómo deben desarrollarse y enfocarse estas tareas?, ¿Qué métricas deben seguir los diseñadores o desarrolladores de software para realizar el diseño?

2.4.1. Importancia de las Interfaces de Usuario

La interfaz de un sistema es el medio a través del cual el usuario puede comunicarse con una computadora, dispositivo o sistema; es parte de lo que el usuario ve, toca e interactúa. Comúnmente la palabra interfaz se asocia con interfaz gráfica de usuario (GUI), la cual es el punto de conexión que utiliza un conjunto de imágenes y objetos gráficos para representar la información y las acciones disponibles de un sistema [41][36].

Los usuarios interactúan comúnmente con un sistema de una computadora con el objetivo de llevar a cabo una tarea específica fundamental o crítica de su trabajo. Dependiendo de la experiencia de usuario con la interfaz el sistema puede tener éxito o fracasar en la ayuda del desarrollo de la tarea. Cuando un sistema pobre en diseño fracasa reduce la productividad del usuario, genera tiempos de aprendizaje inaceptables y errores constantes; todos estos errores conducen a la frustración y al rechazo del sistema [19].

Las interfaces gráficas de usuario han traído una gran cantidad de beneficios a usuarios y organizaciones. Estudios recientes muestran que usuarios de interfaces gráficas cometen menor cantidad de errores al interactuar con los sistemas y tareas, sienten menor frustración, sufren menor fatiga y son más capaces de aprender por si mismos nuevas funcionalidades de los sistemas [31]. Desde el punto de vista de los desarrolladores de software, las interfaces gráficas son más difíciles de diseñar que aquellas basadas en línea de comandos. La interacción con GUI en realidad es más compleja debido a que se basa en diferentes principios de manipulación directa y acceso múltiple a ventanas, iconos, menús y dispositivos. Una interfaz basada en línea de comandos o solo texto normalmente solo permite al usuario accesos secuenciales, lo que genera un proceso secuencial que puede ser diseñado fácilmente.

En las GUI los diseñadores presentan al usuario diversas acciones y objetos donde el usuario decide qué acciones hacer y en qué orden. De hecho, podemos imaginar que existen diversas alternativas para el desarrollo de tareas y es el usuario quien decide cual es la forma más sencilla o compleja de realizarla. Muchos sistemas que incorporan diferentes grupos de usuarios trabajando simultáneamente permiten incrementar la productividad, generan ambientes colaborativos que permiten tener un enfoque más sencillo de un problema complejo desde diversos puntos. Es importante mencionar que el diseño e implementación de una exitosa interfaz de usuario es importante, aun así no solo se dirige hacia diseñadores, incluye además usuarios, grupos y organizaciones [19].

2.4.2. Diseño de Interfaces de Usuario

Un diseño adecuado de interfaz permite armonizar la relación entre los usuarios y las tareas, incrementando las habilidades y la capacidad de aprendizaje. Una buena interfaz por definición es fácil de aprender y fácil de utilizar, además de motivar al usuario a experimentar y probar sus funcionalidades sin caer en la frustración. Una interfaz que cumple las características anteriores ofrece una buena imagen que convence a clientes, administradores y a los propios usuarios.

2.4.2.1. Clases de Interfaces de Usuarios

La interacción entre los usuarios y un sistema puede llevarse a cabo de diferentes maneras, una de ellas se da a través de los diálogos (comunicación con el sistema). Existe diferentes clases de diálogos y cada uno de ellos presenta ventajas y desventajas dependiendo de la situación en la que se utilicen [19].

1. **Lenguaje de comandos:** Son aquellos en el que los usuarios teclean instrucciones a la computadora en un lenguaje formalmente definido. La ventaja de esta aproximación es la flexibilidad que ofrece, permitiendo a los usuarios crear sus propios comandos; sin embargo este tipo de diálogo requiere entrenamiento y capacidad de memorización. Un ejemplo de este tipo de lenguaje seria el comando unix:
   'mv file1 file2'
2. **Lenguaje Natural:** Son aquellos en los cuales el lenguaje de comandos de usuario es importante, corresponde a un subconjunto bien definido de algún lenguaje natural, por ejemplo el Inglés. Las interfaces en lenguaje natural son fáciles de aprender, aunque requieren un nivel de escritura considerable en la parte del usuario. Un ejemplo de este tipo de lenguaje sería la siguiente entrada correspondiente al sistema AI Corp de la Universidad de Cambridge:
   'Which women work in New York City'
3. **Sistemas de Menú:** Permiten al usuario la ejecución de acciones seleccionando opciones de un menú de alternativas desplegables. Los sistemas de menú son populares debido a que reducen el tiempo de aprendizaje, el número de acciones para realizar una tarea y principalmente por ayudar a la toma de decisiones. En la actualidad la mayoría de entornos de desarrollo ofrecen funcionalidades que soportan el diseño de interfaces basadas en menús.
4. **Formularios de Relleno:** Los diálogos basados en esta estructura son aquellos en el cual los usuarios introducen datos rellenando los campos solicitados por los formularios en pantalla. Este tipo de diálogos simplifican considerablemente la entrada de datos y requieren solo un poco de entrenamiento.
5. **Manipulación Directa de Interfaces:** Son aquellas en las que el usuario manipula e interactúa directamente con elementos de la interfaz a través de pulsaciones en botones y movimientos del ratón; el elemento principal en este tipo de diálogos es el icono; un icono es un símbolo gráfico o pictograma que se utiliza en lugar de palabras. La manipulación directa de interfaces utiliza un sistema de ventanas o entornos en los cuales la pantalla del usuario se divide en áreas rectangulares que se superponen, cada una de estas administra

una función específica.

La manipulación directa de interfaces representa conceptos de tareas visuales, fáciles de aprender y utilizar; fomentando la experimentación con las características del sistema y generalmente ofrecen un alto grado de satisfacción al usuario. Aún con todos los beneficios anteriores, el diseño y programación de interfaces de usuario es una tarea difícil, sin embargo la mayoría de estándares de diseño de interfaces se orientan actualmente sobre la manipulación directa.

Un aspecto importante que determina un buen diseño de interfaz es la apropiada selección de la clase de interfaz de usuario que mejor se adapte a las necesidades de las personas y es que en realidad una decisión sobre una clase particular de interfaz requiere una gran cantidad de aspectos (decisiones) que deben considerarse, por ejemplo: la información que debe aparecer en la interfaz, la cantidad, el orden, los mensajes de error y su presentación en pantalla. Aun cuando esta tarea resulta complicada existen ciertas guías de diseño que facilitan el diseño de interfaces de usuario y se orientan hacia el concepto de un buen diseño [31].

Hasta este punto se puede dar una idea de que los diálogos de manipulación directa y sistemas de menú son los que mejor se adaptan a nuestras necesidades y restricciones. Para diseñar estos componentes estudiemos los principios que deben utilizarse en el proceso.

2.4.2.2. Principios de un buen diseño

Siempre que se diseñe un componente dialogo, el autor [4] recomienda considerar los siguientes criterios:

1. La interfaz/dialogo debe ser **consistente**. Esta propiedad asegura que las expectativas que el usuario acumula a través del uso de una parte del sistema no son frustradas por los cambios en las convenciones utilizadas en otra parte, por ejemplo nombres de un componente en dos vistas diferentes.
2. Los diálogos deben permitir accesos a través de las diferentes partes del sistema.
3. Los diálogos deben ofrecer **retroalimentación** al usuario.
4. Las secuencias de diálogos deben organizarse en grupos lógicos.
5. Los sistemas deben ofrecer un manejo de errores simple.
6. Los sistemas deben permitir deshacer acciones.
7. Los sistemas deben ofrecer a los usuarios la experiencia de control total del sistema.
8. Los opciones de los sistemas deben ser fáciles de recordar para el usuario.
9. Los diálogos deben presentar **naturalidad**. Un diálogo natural es aquel que no altera la aproximación del usuario a la tarea en el proceso de interacción con el sistema. En este caso el orden del diálogo es importante; por ejemplo el orden de una entrada de usuario debería orientarse al flujo normal de trabajo del usuario sin importar si esta es demasiado complicada para el programador.

2.4.2.3. Diseño Gráfico de Interfaz de Usuario

Existe una gran diferencia entre las interfaces gráficas y aquellas que funcionan bajo líneas de comando; diversos estudios muestran que el uso de interfaces gráficas permite a los usuarios cometer menor cantidad de errores, frustración, fatiga e incrementa la disponibilidad para aprender nuevos componentes del sistema [19, 31]. Para lograr estos objetivos se requieren dos requerimientos: *naturalidad e intuición*; Ambos pueden obtenerse con la creación adecuada de metáforas, comunicación entre las diversas capas de la interfaz y el diseño gráfico.

Uno de los objetivos principales de una interfaz gráfica de Usuarios (GUI) es crear una ilusión de objetos que pueden ser manipulados; para ello se requiere una ventana, iconos, menús, cuadros de diálogo, botones y barras de desplazamiento. Sin embargo existe una gran diferencia entre lo que aparentemente se ve y lo que el usuario realmente ve. Desde el punto de vista del usuario, una interfaz gráfica posee 3 elementos principales que contribuyen a su aspecto y sensación:

1. **La Apariencia Visual:** Hace referencia a la posición de los elementos, disposición de menús y barras de desplazamiento, así como el diseño de los iconos utilizados para representar aplicaciones y funcionalidades.
2. **Comportamiento:** Es la manera en que responde a ciertas acciones iniciadas por el usuario, por ejemplo; clic y enfoque.
3. **Metáfora:** Comúnmente utilizado para describir la analogía utilizada en el diseño y la implementación de una interfaz gráfica. El uso de este elemento ayuda a visualizar las consecuencias de una acción [19].

2.4.2.4. Diseño de Iconos

El icono es uno de los elementos más comunes en las interfaces y es en la actualidad el punto familiar de las interfaces para los usuarios. Los iconos pueden agruparse en iconos de datos e iconos de función; los primeros se refieren a objetos que responden a acciones, por ejemplo: documentos, carpetas y archivos; en los cuales las acciones que pueden aplicarse son: copiar, pegar, cortar, etc. En el caso de los iconos de función representan objetos que realizan acciones, por ejemplo calculadoras[19].

Comúnmente existen 4 categorías de iconos, estas son las siguientes:

1. **Semejanza:** Representan el referente a través de una imagen análoga directa.
2. **Ejemplar:** Representan la existencia de una clase de objeto.
3. **Simbólicos:** Representan propiedades del referente a un nivel de abstracción superior.
4. **Arbitrarios:** Este tipo de iconos no tiene una semejanza con el referente.

La importancia real de los iconos es el hecho de expresar directamente el significado de su acción; es decir, las tareas que pueden realizar. Para lograr una buena especificación de las acciones que puede realizar un icono, es recomendable involucrar al usuario en el proceso de diseño; de esta manera el usuario se familiariza con el significado de las metáforas y con el futuro entorno del sistema .

La Organización Internacional para la Estandarización (*ISO: the International Organization Standardization*) enumera dos tipos de iconos: interactivos y No-interactivos. Los iconos interactivos son representaciones de objetos, apuntadores, controles y herramientas a través de los cuales el usuario interactúa con la aplicación. Los iconos no-interactivos son comúnmente indicadores de estado.

2.4.3. Usabilidad de un sistema

Es fácil comprender que los sistemas deberían diseñarse para ser fáciles de utilizar; y con frecuencia es un factor clave del mercado de los sistemas. Cuando un sistema se etiqueta como fácil de usar implica que: ¿cualquiera lo puede utilizar; no importando sus habilidades y circunstancias? ¿Ha sido probado para que cumpla la facilidad de uso? ¿Puede aprenderse desde el primer momento de uso? ¿El usuario disfrutara usando el sistema? Cada una de estas interrogantes debe contextualizarse antes de enunciar una definición coherente del término *'usabilidad'*.

De acuerdo al autor [4], *"la usabilidad de un producto o sistema es el grado en el cual usuarios específicos pueden realizar tareas u objetivos específicos dentro de un ambiente particular; eficientemente, efectivamente, cómodamente y en una forma aceptable"*. [19] por su parte específica que el término usabilidad hace referencia a los siguientes principios:

Usabilidad: Es la efectividad, eficiencia y satisfacción con la cual usuarios específicos pueden lograr determinados objetivos específicos en entornos particulares.

Efectividad: Es la precisión y completitud con la cual los usuarios logran objetivos específicos.

Eficiencia: Es la precisión y completitud de los objetivos logrados en relación a los recursos invertidos.

Satisfacción: Es la comodidad y aceptación en el uso de un sistema.

2.4.3.1. Metodologías de Desarrollo

Muchos proyectos de desarrollo fallan al intentar cumplir sus objetivos; en la industria se estima un porcentaje del 60 % en fallos. Esta cifra según diversos autores se origina principalmente por la falta de atención en cuestiones de diseño [31]. Cuidar de estos aspectos al inicio de un desarrollo puede garantizar un menor costo total debido a que en etapas posteriores las correcciones y mantenimiento en los sistemas será menor.

En el ciclo de uso de un producto los usuarios tienden a alcanzar una etapa de uso limitado debido a que solo utilizan ciertas funcionalidades del sistema; sin embargo ello implica que existen otras que no entienden o no pueden utilizar, si esto sucede y la etapa de dominio total no ocurre existe un claro problema de usabilidad. Este problema implica que existe un gasto para la empresa y un esfuerzo innecesario para los desarrolladores de software; que puede traducirse en insatisfacción para los usuarios.

La especificación de usabilidad se interesa en identificar para una situación dada los aspectos que construirían una experiencia de usabilidad positiva para el usuario. La evaluación de usa-

bilidad se enfoca en probar un prototipo o sistema para averiguar si es usable de acuerdo a la especificación [19] (Ver figura 2.16).

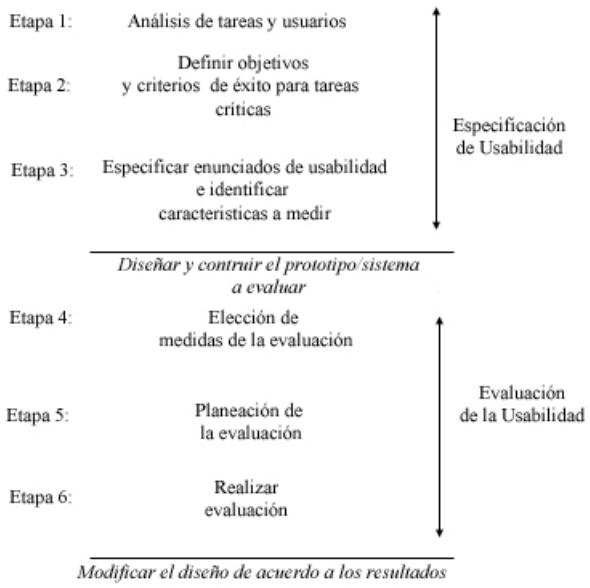

Figura 2.16: Principales etapas en la específicación de usabilidad y evaluación.

Aun cuando la especificación de usabilidad ha sido previamente estudiada en las empresas, muchos consideran que solo es aplicable al desarrollo de sistemas y es innecesaria; sin embargo es importante aclarar que la especificación y evaluación de la usabilidad de un sistema también puede hacer lo siguiente:

1. Ayudar a los clientes a seleccionar productos comparando efectividad, eficiencia y satisfacción en el mismo contexto.
2. Ayudar a los equipos de diseño a reconocer la magnitud de los problemas de diseño relacionados con la usabilidad.
3. Ayudar a los proveedores y clientes a específicar requerimientos de sistemas a medida.

2.4.3.2. Especificación de Usabilidad

Dentro del proceso de especificación y evaluación de la usabilidad la primera etapa es (ver figura 2.16) la especificación de los objetivos. Es importante que antes de evaluar un sistema o producto como tal es necesario comprender el entendimiento de los usuarios, identificar las tareas y las relaciones. La importancia de la especificación de usabilidad exige que forme parte de la etapa de diseño del desarrollo de software y los objetivos de usabilidad deben incorporarse a la específicación de requerimientos en el proceso de desarrollo.

1. **Etapa 1.** *Análisis de Tareas y Usuario*
   Se considera un objetivo clave y consiste en identificar a los usuarios, sus características, las tareas que realizan, circunstancias y los criterios de éxito. Por ejemplo los usuarios de los sistemas de oficina pueden dedicar cierto tiempo a aprender el uso de tecnologías siempre que el trabajo lo requiera, también pueden ser necesarias habilidades con los diversos dispositivos de entrada; lo más importante es conocer como realizan sus tareas, las fuentes de ayuda y la forma en que solucionan problemas del sistema [19].
2. **Etapa 2.** *Establecimiento de los objetivos de usabilidad y criterios de éxito*
   Cada objetivo de usabilidad debe estar asociado a un criterio de éxito, es decir, el objetivo debe ser alcanzable y medible, por ejemplo: 85 % de los usuarios disfrutaron usando el sistema. Una consideración importante es la evaluación de las tareas más importantes, sin embargo no es claro que criterio tomar para seleccionar una tarea importante, para ello se establecen los siguientes lineamientos:

   a) *Ventaja competitiva:* Los sistemas pueden diferenciase de los competidores mediante funcionalidades claves, es una buena decisión evaluar estas tareas.

   b) *Requerimientos de mercado:* Para algunos sistemas , usabilidad es un pre-requisito debido a que otros sistemas cumplen esta propiedad.

   c) *Tareas primarias:* Con frecuencia la impresión que reciben los usuarios de un sistema puede ser un factor clave. Si los usuarios expresan una buena experiencia con la primera tarea que el sistema les permite realizar, la confianza en el resto del sistema puede mantenerse.

   d) *Tareas de mayor frecuencia:* Una vez que los sistemas se encuentran en uso, la tarea que se realice frecuentemente se transforma en crítica, por ello es necesario anteponer eficiencia como factor clave del sistema. Si la tarea se realiza 100 veces la diferencia de duración puede afectar en el rendimiento del usuario.

3. **Etapa 3.** *Establecer enunciados de usabilidad e identificar medidas. Un enunciado de usabilidad debería incluir lo siguiente:*

   a) Título, propósito y una breve descripción del sistema

   b) Descripción de los usuarios objetivo y sus características (conocimientos, habilidades, motivación)

   c) Breve descripción de las tareas compartidas y sus características (frecuencia, tiempo)

   d) Descripción de los materiales utilizados (hardware, software y otras herramientas)

   e) Una descripción del entorno (físico, psicológico, aprendizaje y condiciones de trabajo)

   f) Una especificación de los objetivos de usabilidad para un contexto particular.

   Cada uno de los objetivos de usabilidad deben ir acompañados del criterio de éxito en un contexto dado, también es necesario incluir el criterio que se medirá, por ejemplo la satisfacción, efectividad, tiempo en realizar la tarea, etc. De manera específica en cada una de las tareas podemos medir:

- Número de comandos (pasos) utilizados
- Número de acciones requeridas por el usuario para completar una tarea.
- Tiempo en realizar una tarea.
- Tiempo en aprender una funcionalidad.
- Conocimiento del sistema.
- Aptitudes y opiniones.
- Disponibilidad de materiales de ayuda.
- Errores de entendimiento o tipográficos
- Tiempo de recuperación de errores

4. **Etapa 4.** *Seleccionando las técnicas para medir la usabilidad.*
Aun cuando se han especificado los aspectos que desean medirse, puede resultar complicado llevar acabo esas mediciones debido a que en el proceso pueden ocurrir eventos que alteren los resultados de la evaluación, por ejemplo:

   a) *Problemas con el usuario participante:* El rendimiento del usuario podría disminuir por el hecho de estar siendo medido, catalogado por sus aptitudes y entendimiento con cada una de las preguntas que se le hacen.

   b) *Problemas con la persona que está llevando a cabo la evaluación:* Algunos métodos requieren habilidades específicas, si las personas que se encargan de hacer las medidas carecen de ellas los resultados se verán afectados.

   c) *Problemas de interpretación de resultados:* Muchas veces el criterio personal podría interpretar negativamente los comentarios de un usuario afectando directamente la evaluación.

   Es necesario ser cuidadosos al aplicar alguna de las técnicas de medida, en caso contrario se recomienda que el evaluador sea un experto de HCI o un psicólogo. Algunas técnicas que pueden utilizarse para medir aspectos de usabilidad son las siguientes [19]:

   a) *El primer grupo incluye aquellas que requieren conocimientos de psicología para llevar acabo el análisis.* O bien podrían requerir un experto en diseño de interfaces de usuario ello implica que los diseñadores estarían presentes aun en esta parte del desarrollo de software.

   b) *El segundo grupo incluye técnicas en las cuales se requiere un diseñador pero con asesoramiento de un experto.* En el uso de cuestionarios el diseñador podría involucrarse sin embargo se requiere que el experto revise el cuestionario de criterios del diseñador para elegir los mejores resultados.

   c) *El tercer grupo incluye aquellas técnicas que pueden utilizar a un diseñador sin la supervisión de un experto.* La evaluación cooperativa por ejemplo es uno de estos casos, en la cual se evalúan sistemas sin ayuda de expertos en HCI.

a) Técnicas Grupo 1

Evaluación Analítica: Las técnicas de evaluación analítica se utilizan con frecuencia en

etapas tempranas del diseño; requieren conocimientos de psicología y están basadas en descripciones que predicen el rendimiento del usuario. La interfaz se presenta en un lenguaje de especificación formal o semi-formal, en el cual se analizan y predicen comportamientos de las tareas individuales. Las tareas se describen en términos de operaciones cognitivas y físicas, como se ilustra en la figura 2.17 siguiente [19]:

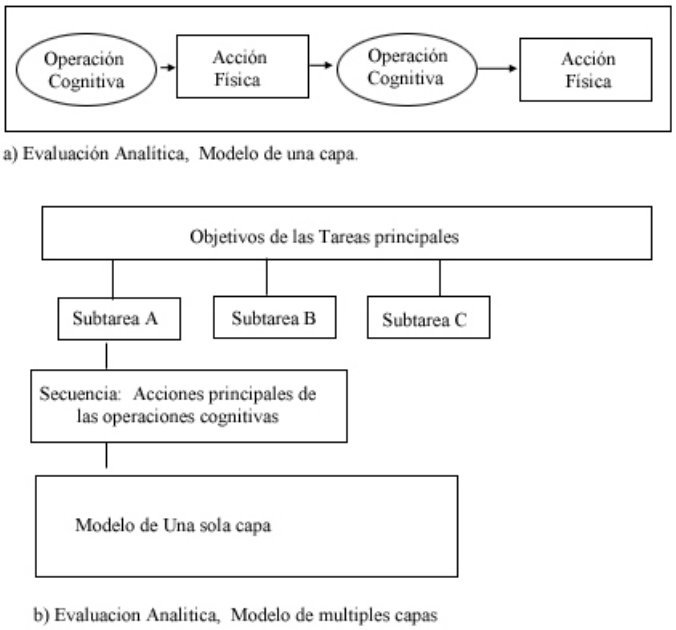

Figura 2.17: Evaluación Analítica: 1 Capa y Multicapa [19]

De la figura anterior el modelo de una capa es utilizado para describir la interacción del usuario en el nivel de teclas principales, mientras tanto el modelo multicapa se utiliza para describir la interacción en los niveles de tareas.

La principal ventaja de una evaluación analítica es que los diseños pueden evaluarse sin necesidad de desarrollar prototipos o pruebas de usuario. La desventaja es el hecho de que los modelos utilizados implican que los usuarios son expertos y libre de errores de interacción, además este tipo de evaluaciones es costosa y requiere una dedicación especial del diseñador, debido a que debe involucrarse en la configuración de la evaluación y en la interpretación de los resultados.

Evaluación Experta: Este método de evaluación requiere expertos en HCI y personal con experiencia en factores humanos; en estas pruebas se le da al experto una tarea relacionada con la interfaz de usuario y el experto plantea su punto de vista. La evaluación del experto se basa en principios de naturalidad, flexibilidad, apoyo, no-redundancia y consistencia o algún otro criterio que permita evaluar el diseño y reflejar los resultados en un reporte; de esta manera el diseñador puede apoyarse para replantear el

diseño de las interfaces.

Evaluación Experimental: Se utiliza cuando se desea manipular un número de factores asociados con el diseño de la interfaz y observar los efectos que genera en el rendimiento. Los experimentos que se planteen deben planearse con ayuda de un psicólogo o un experto en factores humanos. Antes de realizar los experimentos se debe especificar las variables a cambiar y los aspectos a medir, también es importante plantear una hipótesis y una selección estadística de usuarios; el diseñador debe involucrarse estableciendo los factores a medir y decidiendo según los resultados.

b) Técnicas Grupo 2

Observación en un laboratorio de usabilidad: En este tipo de pruebas se permite la grabación de los usuarios utilizando el sistema, lo que permite analizar el comportamiento y los problemas del usuario al finalizar alguna tarea. Aun cuando estas pruebas se realizan en un espacio dedicado, requieren un experto de HCI que en conjunto con los diseñadores discutirán las grabaciones del usuario utilizando el sistema.

Observación en el espacio de trabajo: En este tipo de pruebas la observación se realiza en el espacio de trabajo del usuario, el evaluador observa las tareas que realiza el usuario, los problemas a los que se enfrenta y la forma en que los soluciona; es común que los evaluadores pregunten a los usuarios las tareas que están realizando de manera que pueda grabarse en audio la interacción entre los involucrados. Es importante indicar que cuando un evaluador observa debe ser cuidadoso de no afectar el comportamiento del usuario, evitando influir en sus decisiones y en la forma en que realiza las tareas.

Estudios, Entrevistas y Cuestionarios: El propósito de un método de estudio es reunir las opiniones de los usuarios a través de entrevistas o cuestionarios. Las entrevistas pueden ser altamente estructuradas y el evaluador tendrá una lista de preguntas que el usuario deberá responder; pero también el evaluador puede ser flexible, es decir, en base a una pregunta específica puede explorar aún más realizando otra pregunta improvisada o intuitiva.

Los cuestionarios se utilizan para los estudios de evaluación y requieren un especial cuidado al construirse debido a que deben orientarse a un propósito específico y requiere que sean revisados por un experto. Los tipos de preguntas varían dependiendo del objetivo, es decir, se puede solicitar abiertamente una respuesta, restringirla a una lista de respuestas establecidas, solicitar un porcentaje de satisfacción o simplemente cerrarla a un *si o no*.

Grupo Focal: En un grupo focal los usuarios discuten abiertamente sobre un producto/sistema en particular en presencia de un moderador (investigador, diseñador), como técnica de evaluación puede utilizarse después de que los usuarios han probado un producto o sistema. Antes de llevar a cabo un grupo focal se recomienda que el evaluador prepare una lista de temas a discutir y reúna de 4-6 usuarios.

En un grupo focal los usuarios pueden expresarse naturalmente sobre el tema e interactuar con otros usuarios, de esta manera los resultados definen la forma en que piensan y las cosas importantes en un sistema; se recomienda que el evaluador no participe en la discusión debido a que podría alterar los resultados. Recordemos que el objetivo de esta técnica en este capítulo se orienta a la evaluación de las interfaces, por lo cual los temas que deben tratarse son: tareas, orden de realización, dificultad y experiencias en la interfaces.

c) Técnicas Grupo3

Lista comprobación de Características: Este tipo de evaluación permite determinar aquellas características que son utilizadas por los usuarios; consta de dos o más columnas, la primera indica la característica y las consecuentes preguntas al usuario marcando sí o no ha utilizado esa funcionalidad. A pesar de ser una técnica de evaluación muy simple permite obtener una gran cantidad de información efectiva y el análisis de resultados puede realizarse fácilmente por el diseñador.

Diario de incidencias: Permite al usuario crear un control de los incidentes que ocurren al utilizar el sistema. Un diario de incidencias consta de un formato estructurado que contiene una serie de preguntas relacionados a la ocurrencia de problemas: ocurrencia, origen, forma de solucionarlo, el estado del problema, tiempo invertido, entre otras.

Esta técnica es útil cuando el diseñador desea obtener información de eventos específicos, sin embargo puede ser estresante para el usuario completarlo; por ello se requiere un entrenamiento previo antes de implementar esta técnica de evoluciona.

Evaluación Cooperativa: Esta técnica se prepara especialmente para los diseñadores y es utilizado para obtener información acerca de problemas experimentados por los usuarios cuando se trabaja con un prototipo; la característica más importante es la evaluación conjunta entre el usuario y el diseñador. La forma de llevar acabo esta evaluación es por turnos, el diseñador pregunta acerca del entendimiento del sistema y el usuario responde; la información y los detalles de cada pregunta y situación se registran en un manual , de esta manera cuando se requiere iniciar el construir el sistema el diseñador consulta el manual y puede obtener información acerca de las tareas que deben incluirse paso a paso.

5. **Etapa 5.** *Planeación de la evaluación de la usabilidad.*
   Una evaluación de la usabilidad de un sistema requiere una planeación cuidadosa; los aspectos que deben considerarse son los siguientes: las técnicas de evaluación que se utilizarán, selección de los usuarios, las tareas a realizar , espacio físico, horarios, personal requerido, materiales extras (equipo de grabación) y el papel que desempeñará cada uno de los involucrados.

6. **Etapa 6.** *Realización de la Evaluación.*
   Gracias a la planeación de las fases anteriores se conoce con certeza los involucrados, se cuenta con los usuarios y se conoce la bitácora de las pruebas. Al final de la evaluación el evaluador debe preparar un reporte subrayando los problemas específicos a los cuales

se enfrentaron los usuarios. Este reporte se utiliza como base para que los diseñadores replanteen los diseños y se obtenga una mejor interfaz de usuario para el lanzamiento del producto.

Las técnicas de evaluación anteriores resultan de gran utilidad para medir la usabilidad de un sistema, en nuestro caso las que mejor se orientan al proyecto son las técnicas del grupo 2 en las cuales se requiere la presencia de un experto; estas técnicas se aplicarán en la etapa de la evaluación del proyecto descrita en capítulos posteriores.

2.5. El proceso Unificado de Desarrollo de Software

El proceso Unificado es un metodología/proceso de desarrollo de software que traduce los requisitos de un cliente/usuario en un sistema de software. Sin embargo, el Proceso Unificado es más que un simple proceso; es un marco de trabajo que puede especializarse para una gran variedad de sistemas de software en diferentes áreas, organizaciones, niveles de aptitud y diferentes tamaños de proyectos [16].

La especificación del Proceso Unificado está basada en componentes que se interconectan a través de interfaces bien definidas. Para preparar estos componentes se utiliza el Lenguaje de Modelado Unificado (UML); esta integración define a la metodología como un proceso dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental.

Al hablar de un proceso dirigido por casos de uso nos referimos a que el proceso se enfoca en los escenarios de interacción típica entre un usuario y un sistema de cómputo [12]. Desde el punto de vista de [16] la interacción es un caso de uso. Un caso de uso es un fragmento de funcionalidad del sistema que proporciona al usuario un resultado importante. Un caso de uso representan requisitos funcionales.

Lo más interesante de los casos de uso es que no solo son herramientas que reflejan los requisitos del cliente, sino que también guían el diseño, la implementación y las pruebas. Basados en estos modelos, los desarrolladores al igual que otros involucrados revisan constantemente cada uno de los sucesivos modelos para que cumplan las especificaciones de los casos de uso. Aunque los casos de uso representen el hilo del flujo de desarrollo de sistemas, estos no se desarrollan aisladamente; en conjunto se diseñan especificaciones de arquitectura y otros modelos particulares.

Con respecto a la arquitectura, es el componente que incluye aspectos estáticos y dinámicos más significativos del sistema. La arquitectura es una vista del diseño completo con las características más importantes por ejemplo plataforma de desarrollo, los bloques de construcción, consideraciones de implementación, sistemas heredados y requisitos no funcionales. Como podemos notar, estos aspectos dependen en parte de una valoración de los involucrados que se adquiere por la experiencia.

En términos de ambientes comerciales, un producto de software por lo general dura meses o años. Ante estas situaciones es práctico dividir el trabajo en partes más pequeñas, cada una de estas partes puede corresponder a una iteración que resulta en un incremento del proyecto. Las iteraciones hacen referencia a pasos en el flujo del proyecto y los incrementos al crecimiento del producto [16]. Para garantizar iteraciones en tiempo es recomendable llevar un control y ejecutarse de forma planificada.

2.5.1. Ciclo de vida del Proceso Unificado

El proceso unificado se repite a lo largo de una serie de ciclos que constituyen la vida de un sistema. Cada ciclo concluye con una versión del producto. Formalmente cada ciclo debe constar de 4 fases principales: inicio, elaboración, construcción y transición.

Al finalizar un ciclo particular de un proyecto es necesario liberar una nueva versión del sistema, y cada versión debe estar preparada para su entrega. La finalización de un ciclo termina con un hito en el cual es necesario analizar y determinar la disponibilidad de recursos, artefactos y avances en los modelos desarrollados con el objetivo de tomar decisiones respecto al siguiente ciclo del proyecto. Cuando el proyecto termine el producto terminado debe haber construido las siguientes especificaciones: requisitos, casos de uso, especificaciones no funcionales y los casos de prueba [16].

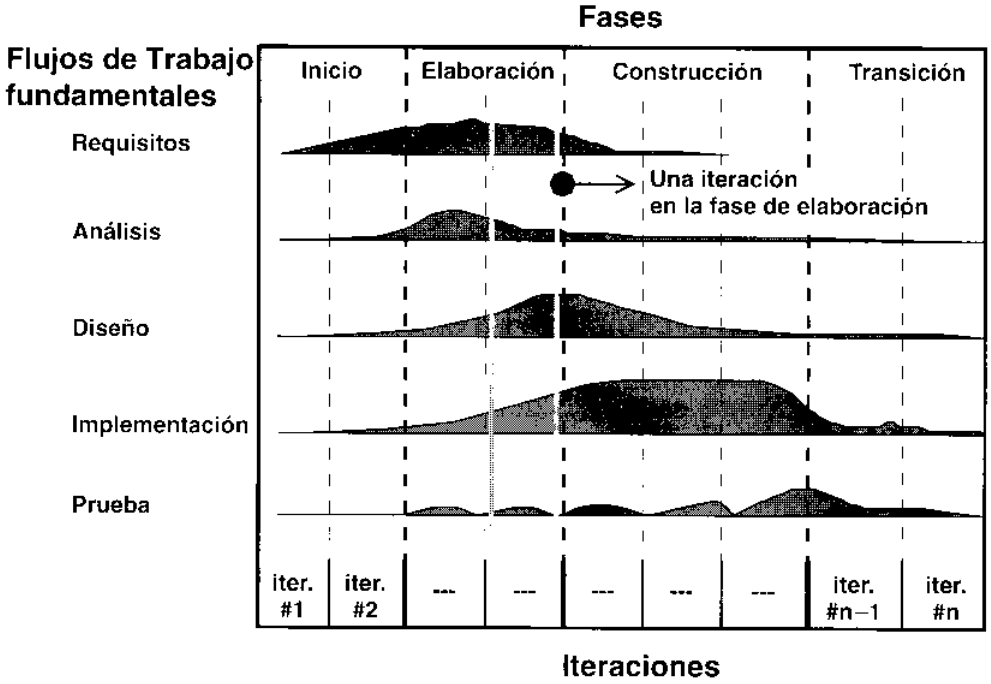

Figura 2.18: Flujos de trabajo fundamentales en un proyecto: requisitos, análisis, diseño, implementación y prueba.

Para comprender el ámbito de los ciclos de un proyecto, analicemos la especificación de flujos de trabajo del Proceso Unificado (ver figura 2.18)

Los flujos de trabajo en el proceso unificado representan las fases a través de las cuales un proyecto de desarrollo de software debe estar planificado y desarrollado. De la figura 2.18 podemos visualizar que sobre cada flujo de trabajo se generan n iteraciones a través de las 4 fases de inicio, elaboración, construcción y transición.

- **Fase inicio:** en esta fase se desarrolla una descripción del producto final a partir de una buena idea y se presenta el análisis de negocio para el producto.
- **Fase de elaboración:** en fase se especifican en detalle la mayoría de los casos de uso del producto y se diseña la arquitectura del sistema. Al finalizar esta fase los líderes de proyectos deben planificar las actividades y estimar recursos para terminar el ciclo.
- **Fase de construcción:** en esta fase se crea el producto planeado para el ciclo.
- **Fase de transición:** cubre el periodo durante el cual el producto se convierte en una versión beta.

En relación a los flujos de trabajo fundamentales, podemos apreciar que estos son similares a otras metodologías como se mencionaba en el capítulo 1. Algunas de las justificaciones referentes a la elección de esta metodología es el hecho de estar basada en UML, la cual es muy simple de utilizar, modelar y los artefactos que libera en cada una de sus etapas son de gran valor por que permiten apreciar los sistemas desde diferentes perspectivas. En capítulos siguientes desarrollaremos el proyecto interfaz/entorno de verificación de diagramas de tareas (TFI) guiado por esta metodología.

Capítulo 3

Análisis y Diseño del Sistema

3.1. Introducción

En el capítulo anterior se establecieron las bases teóricas del Método Discovery y se describió brevemente la forma en que se gestionan los proyectos de software a lo largo de sus diferentes etapas. Las etapas en los proceso de desarrollo de software varían de acuerdo a la metodología, sin embargo, coinciden en el concepto inicial, es decir, la captura de requisitos.

El propósito de los requisitos es guiar el desarrollo y establecer un acuerdo sobre lo que debe y no debe hacer el sistema. Existen diferentes puntos de partida, sin embargo, la mayoría son escalables y evolucionan iterativamente. La escalabilidad es un factor dinámico e incremental común en los desarrollos y una de las metodologías que mejor apoya esta noción es el Proceso Unificado [16].

El Proceso Unificado como se describió en el capítulo 2 constituye una proceso/herramienta que establece flujos de trabajo gestionados iterativamente. Cada iteración libera versiones que se integran con los diagramas UML permitiendo construir sistemas que evolucionan en el tiempo.

En este capítulo se detalla cada uno de los flujos (requerimientos, análisis, diseño) y decisiones que guían el desarrollo del producto de software: interfaz de verificación de diagramas de tareas (*Task Flow Diagrams Verification Interface, TFI*) basados en el ciclo de vida del Proceso Unificado.

3.2. Requisitos

Cada proyecto de software es diferente y puede comenzar a solucionarse de diversas maneras. El Proceso Unificado plantea como posibles puntos de partida un modelo de negocios, un modelo de objetos, una especificación detallada o una noción simple del sistema [16].

En el caso particular de este proyecto se partió de una noción simple con el objetivo de refinar, estudiar y especificar detalladamente cada uno de los aspectos del sistema. Tomando como

punto de partida esta noción la descripción textual del contexto del sistema es la siguiente:

Con el objetivo de integrar las diferentes fases del Método Discovery es importante desarrollar un sistema en el cual se permita la construcción de diagramas de tareas, traducción a modelos algebraicos y la verificación de propiedades por medio de lógicas temporales. La construcción de las expresiones para la verificación de propiedades debe guiarse por un proceso visual simple que de soporte al usuario en todo momento, además la ejecución de las consultas debe apoyarse en las herramientas existentes; siendo todo el procedimiento transparente al usuario guiado por elementos gráficos, ventanas, diálogos y otros elementos GUI. El producto de software que debe entregarse al cliente es un componente (plug-in) instalable en el entorno de desarrollo Eclipse.

Observación: De acuerdo a las justificaciones y objetivos expuestos en el capítulo 1, el sistema se centra en el proceso de verificación de propiedades por medio de lógicas temporales y debe diseñarse como complemento (plug-in) para integrarse al entorno de desarrollo Eclipse.

Analizando la descripción anterior se identificaron algunos aspectos del sistema a desarrollar:

- **Participantes:** usuario del sistema y un sistema externo.
- **Funcionalidades principales:** verificación de propiedades en un diagrama de tareas, la cual se realiza en los siguientes pasos detallados en el capítulo anterior.
  - Construir el diagrama de tareas
  - Traducir el diagrama de tareas al álgebra de tareas
  - Generar estructura de kripke.
  - Especificar propiedades a verificar mediante LTL y CTL.
  - Realizar verificación.
  - Analizar resultados
  - Representar y visualizar resultados.

En el análisis de requisitos se evidenció una limitante: la dependencia de las herramientas existentes. Estas herramientas hacen referencia a los compiladores desarrollados para el Método Discovery descritos en el capítulo 8 de [9]. Pensando en que estas herramientas deberían utilizarse en el proyecto se realizó un análisis de sus características, funcionamiento y reglas de ejecución. Los resultados de estas pruebas se resumen a continuación:

1. **Ambiente - Interacción:** Consola del sistema en modo texto.
2. **Requerimientos de Usuario (perfil):**
   - Amplio conocimiento de diagramas de tareas, modelos algebraicos y comprensión de flujos en un proyecto.
   - Conocimiento de operadores temporales, sintaxis e interpretación de resultados.
   - Conocimiento de actividades en términos de trazas (caminos) de ejecución.
3. **Tareas no contempladas por la herramienta:**

- Análisis de resultados.
- Representación visual de resultados.
- Resumen de flujos de ejecución.
- Simplicidad de ejecución.
- Manual de ayuda.

Las características de las herramientas existentes mostró que en el desarrollo del proyecto era necesario tener especial cuidado en los siguientes aspectos: nivel de abstracción presentado a los usuarios, representación de resultados y disponibilidad de ayuda.

3.2.1. Encontrando actores y casos de uso

Una vez comprendida la visión general del sistema y sus posibles limitantes, se identificaron a los actores y se describieron las funcionalidades esperadas, esto por medio de los casos de uso.

Particularmente *Task Flow Diagrams Verification Interface,* ***TFI*** se conceptualizó como una herramienta que permite construir expresiones en lógica temporal para la verificación de diagramas de tareas, analizar sus resultados y compararlos con el modelo algebraico base de los diagramas. Dentro de la herramienta el usuario podrá:

1. Importar/establecer modelos algebraicos que describen diagramas de tareas particulares.
2. Construir expresiones en lógica temporal (LTL/CTL) sintácticamente correctas dentro de una área de edición de manera guiada.
3. Visualizar si las expresiones construidas presentan errores de sintaxis y en caso de inconsistencias conocer la fuente del error.
4. Consultar la sintaxis, uso y operaciones permitidas en Lógica Temporal y Computacional en Árbol aplicadas al modelo algebraico base.
5. Enlazar y configurar gráficamente las librerías necesarias para la ejecución de un proyecto.
6. Visualizar los resultados de verificación agrupados en categorías.
7. Modificar modelos algebraicos según las necesidades detectadas, reflejando los cambios en las expresiones.
8. Utilizar las acciones básicas de los editores: copiar, cortar, pegar, borrar, comentar, deshacer, imprimir, etc.
9. Establecer perspectivas (vistas) del proyecto.

De los requisitos anteriores, se seleccionaron y priorizaron los casos de uso, los cuales se muestran en el cuadro 3.1 y en la figura 3.1.

Casos de Uso: TFI

Primarios:

1. Importar modelo de tareas (álgebra).
2. Crear modelo de tareas (álgebra).
3. Verificar sintaxis del modelo de tareas.
4. Crear archivo de consulta.
5. Crear expresión lógica de consulta.
6. Verificar sintaxis de consultas.
7. Consultar documentación aplicada.
8. Comentar código.
9. Definir dependencias de ejecución.
10. Ejecutar expresiones.
11. Visualizar resultados.
12. Abrir archivo.
13. Editar archivo.
14. Guardar archivo.

Secundarios:

1. Asociar perspectiva de proyecto.
2. Copiar, Pegar , Cortar y Borrar expresión
3. Imprimir archivo

Opcionales:

1. Acercar vista a resultados
2. Alejar vista a resultados
3. Deshacer últimos cambios
4. Cambiar color de fuente
5. Cambiar tamaño de fuente

Cuadro 3.1: Casos de uso priorizados del sistema

Con la priorización de casos de uso se identificaron 2 actores del sistema: 1 usuario y el recurso externo (compilador) que ejecuta las expresiones especificadas.

3.2.2. Detallar casos de uso

Una vez definido el esquema de casos de uso, el siguiente paso consistió en describir los sucesos en detalle, las condiciones de activación, flujos alternos y las interacciones que se generan en el sistema. A continuación se detallan los principales casos de uso identificados:

. . . . . .

CASO DE USO 1 (CU_1): **Importar modelo de tareas (álgebra)**

Descripción breve: El caso de uso permite al actor principal del sistema añadir un modelo del álgebra de tareas previamente desarrollado a un proyecto para realizar el proceso de verificación de propiedades.

Precondición: El usuario ha construido un modelo algebraico del diagrama de tareas y decide incorporarlo al proyecto para verificar propiedades específicas.

Flujo de sucesos-Camino básico

Acción del actor	Respuesta del sistema
1.- El usuario invoca el caso de uso para importar su modelo.	
	2.- El sistema muestra un asistente en el cual se solicitan: 1)La ubicación del modelo (archivo) a importar 2) El proyecto al cual se desea importar el modelo y 3)Nombre del nuevo modelo (extensión tfa)
3.- El usuario ingresa los datos solicitados y da clic en el botón finalizar	
	4.- El sistema abre automáticamente el modelo y verifica su estructura (parser rápido) , mostrando información en la vista problemas en caso de existir errores.
	5.- La instancia del caso de uso finaliza.

Cuadro 3.2: Flujo normal de eventos del CU_1

Caminos alternativos: En el paso 1, 3 el usuario puede cancelar el procedimiento.

Postcondicion: La instancia del caso de uso finaliza cuando el modelo seleccionado por el usuario se añade correctamente al proyecto o se cancela el procedimiento.

.....

CASO DE USO 2 (CU_2). **Crear modelo de tareas (álgebra)**

Descripción breve: El caso de uso permite al actor principal del sistema construir un modelo del álgebra de tareas (archivo en blanco) para realizar el proceso de verificación de propiedades.

Precondición: El usuario ha diseñado el diagrama de tareas y decide incorporarlo al proyecto en forma de modelo algebraico para verificar propiedades específicas.

Flujo de sucesos-Camino básico

Acción del actor	Respuesta del sistema
1.- El usuario invoca el caso de uso para crear su modelo algebraico.	
	2.- El sistema muestra un asistente solicitando: 1)El nombre del modelo (archivo con extensión tfa) y 2) El proyecto al cual pertenecerá el modelo.
3.- El usuario ingresa los datos solicitados y da clic en el botón finalizar	
	4.- El sistema abre automáticamente el archivo modelo (vacío) para que el usuario escriba sus especificaciones (modelo).
	5.- La instancia del caso de uso finaliza.

Cuadro 3.3: Flujo normal de eventos del CU_2

Caminos alternativos: En el paso 1, 3 el usuario puede cancelar el procedimiento.

Postcondicion: La instancia del caso de uso finaliza cuando el archivo del modelo se crea satisfactoriamente y es abierto por el sistema o bien cuando el usuario decide cancelar el procedimiento.

.....

*CASO DE USO 3 (CU_3).***Crear archivo de consulta**

Descripción breve: El caso de uso permite al actor principal del sistema construir un archivo de consulta en el cual se especifican las propiedades a verificar en un modelo del álgebra de tareas.

Precondición: El usuario ha incorporado el modelo algebraico que describe el diagrama de tareas; también ha verificado que la sintaxis sea correcta. A continuación decide crear un contenedor de consultas donde se especificarán las expresiones en lógica temporal.

Flujo de sucesos-Camino básico

Acción del actor	Respuesta del sistema
1.- Con el archivo del modelo abierto el usuario invoca el caso de uso para crear el archivo de consulta .	
	2.-El sistema verifica la sintaxis del modelo y en caso de ser incorrecta informa al usuario a través de un cuadro de dialogo , muestra la vista de problemas y el proceso se detiene. En caso contrario el flujo continua en el paso 4.
3.- El usuario corrige los problemas e invoca nuevamente al caso de uso	
	4.- El sistema detecta que no existen errores de sintaxis y genera internamente un submodelo de tareas.
	5.- El sistema crea y presentar el contenedor de consultas al usuario.
	5.- La instancia del caso de uso finaliza.

Cuadro 3.4: Flujo normal de eventos del CU_3

Caminos alternativos: En el paso 2 existe una restricción que debe cumplirse si se desea continuar con el flujo, si no se desea continuar el caso de uso puede ser cancelado por el usuario.

En el paso 4 si el sistema por problemas internos no puede construir el submodelo, el caso de uso es abortado y se informa al usuario.

Postcondicion: La instancia del caso de uso finaliza cuando el archivo de consulta es creado y mostrado satisfactoriamente al usuario o bien es cancelado por problemas internos o decisión del usuario.

.....

*CASO DE USO 4 (CU_4).***Crear expresión lógica de consulta**

Descripción breve: El caso de uso permite al actor principal del sistema construir gráficamente paso a paso diversas expresiones de consulta en lógica LTL y CTL.

Precondición: El usuario ha creado el contenedor de consultas y decide comenzar a construir las expresiones en lógicas temporales (LTL/CTL) para verificar propiedades en el modelo.

Flujo de sucesos-Camino básico

Acción del actor	Respuesta del sistema
1.- Con el archivo del modelo abierto el usuario invoca el caso de uso para construir expresiones (consultas).	
	2.-El sistema ofrece indicaciones al usuario sobre la forma de comenzar una expresión y una lista de operaciones y tareas.
3.- El usuario selecciona una opción especifica de las disponibles .	
	4.- El sistema muestra las posibilidades de la operación aplicadas al modelo algebraico del proyecto.
5.- El usuario elige una expresión y da clic en el botón “ok”.	
	6.- El sistema añade la expresión automáticamente a la posición del cursor en el contenedor.
	7.- A medida que el usuario construye sus consultas el sistema detecta en caso de existir errores de sintaxis y le informa al usuario por medio de la vista de problemas.
	8.- La instancia del caso de uso finaliza.

Cuadro 3.5: Flujo normal de eventos del CU_4

Caminos alternativos: En el paso 2,5, 7 el usuario puede optar por abortar el caso de uso.

Postcondicion: La instancia del caso de uso finaliza cuando el usuario ha construido su expresión (es) de consulta o bien es cancelado.

. . . . . .

*CASO DE USO 5 (CU_5).***Definir dependencias de ejecución**

Descripción breve: El caso de uso permite al actor principal del sistema definir la ubicación de los archivos externos necesarios para la ejecución de la verificación de propiedades en el álgebra de tareas.

Precondición: El usuario ha construido las expresiones que verificaran propiedades en el modelo algebraico y decide configurar los compiladores para la ejecución de las consultas.

Flujo de sucesos-Camino básico

Acción del actor	Respuesta del sistema
1.- Con el archivo de consulta abierto el usuario invoca el caso de uso.	
	2.- El sistema presenta un cuadro de dialogo donde solicita seleccionar el archivo de ejecución LTL y CTL para la ejecución de las consultas construidas.
3.- El usuario explora los archivos hasta encontrar los compiladores necesarios para la ejecución y a continuación da clic en el botón “ok”	
	4.- El sistema crea/actualiza la configuración del archivo LtlCtl.ini del espacio de trabajo de Eclipse.
	5.- La instancia del caso de uso finaliza.

Cuadro 3.6: Flujo normal de eventos del CU_5

Caminos alternativos: En el paso 2,3 el usuario puede optar por abortar el caso de uso.

Postcondicion: La instancia del caso de uso finaliza cuando el usuario proporciona la ruta de los archivos (compiladores) requeridos para la ejecución del sistema o bien es cancelado.

.....

*CASO DE USO 6 (CU_6).***Ejecutar expresiones.**

Descripción breve: El caso de uso permite al actor principal del sistema la ejecución de la verificación de propiedades en el álgebra de tareas.

Precondición: El usuario ha construido las expresiones que verificaran propiedades en el modelo algebraico, configurado las dependencias de ejecución y revisado que no existan errores de sintaxis, a continuación decide ejecutar las consultas.

Flujo de sucesos-Camino básico

Acción del actor	Respuesta del sistema
1.- Con el archivo de consulta abierto el usuario invoca el caso de uso para la ejecución de las expresiones de consulta.	
	2.- El sistema verifica que no existan errores de sintaxis en las expresiones construidas, ademas de que las dependencias de ejecución existan. En caso de no cumplirse las condiciones se muestran los errores de sintaxis en la vista problemas y una advertencia para las dependencias a través de un cuadro de dialogo. En caso que no existan errores el flujo continua en el paso 4.
3.- El usuario corrige los errores e invoca nuevamente al caso de uso.	
	4.- Cuando el sistema detecta que no existen errores enlaza las dependencias y ejecuta las expresiones en el sistema externo (compiladores).
	5.- El sistema muestra los resultados de ejecución en modo texto en la consola y en forma gráfica en un apartado visual del entorno.
	6.- La instancia del caso de uso finaliza.

Cuadro 3.7: Flujo normal de eventos del CU_6

Caminos alternativos: En el paso 4 en caso de generarse errores al intentar ejecutar las expresiones en los compiladores asociados el caso de uso se cancela e informa al usuario.

Postcondicion: La instancia del caso de uso finaliza cuando el sistema presenta los resultados al usuario o bien el caso de uso se cancela debido a errores o decisión del usuario.

.....

CASO DE USO 7 (CU_7). **Visualizar resultados:**

Descripción breve: El caso de uso permite al actor principal del sistema visualizar los resultados de ejecución de dos formas distintas: modo consola y en modo gráfico.

Precondición: El usuario ha ejecutado las expresiones de consulta de manera satisfactoria (CU_6). A continuación decide analizar detenidamente la información resultante.

Flujo de sucesos-Camino básico

Acción del actor	Respuesta del sistema
1.- El usuario invoca el caso de uso para visualizar los resultados de ejecución.	
	2.- El sistema muestra las diferentes vistas del entorno.
3.- El usuario selecciona la vista relacionada con la ejecución de expresiones LTL/CTL.	
	4.- El sistema muestra el apartado de resultados de ejecución categorizados por expresión.
5.- El usuario se desplaza por las categorías analizando la información y toma una decisión.	
	6.- La instancia del caso de uso finaliza.

Cuadro 3.8: Flujo normal de eventos del CU_7

Caminos alternativos: En el paso 3 el usuario puede optar por cancelar el caso de uso.

Postcondicion: La instancia del caso de uso finaliza cuando el sistema presenta los resultados al usuario o bien el caso de uso es cancelado.

3.2.3. Estructurar el modelo de casos de uso.

Con ayuda de la descripción a detalle de los casos uso utilizando la notación UML se construyó el diagrama de casos de uso:

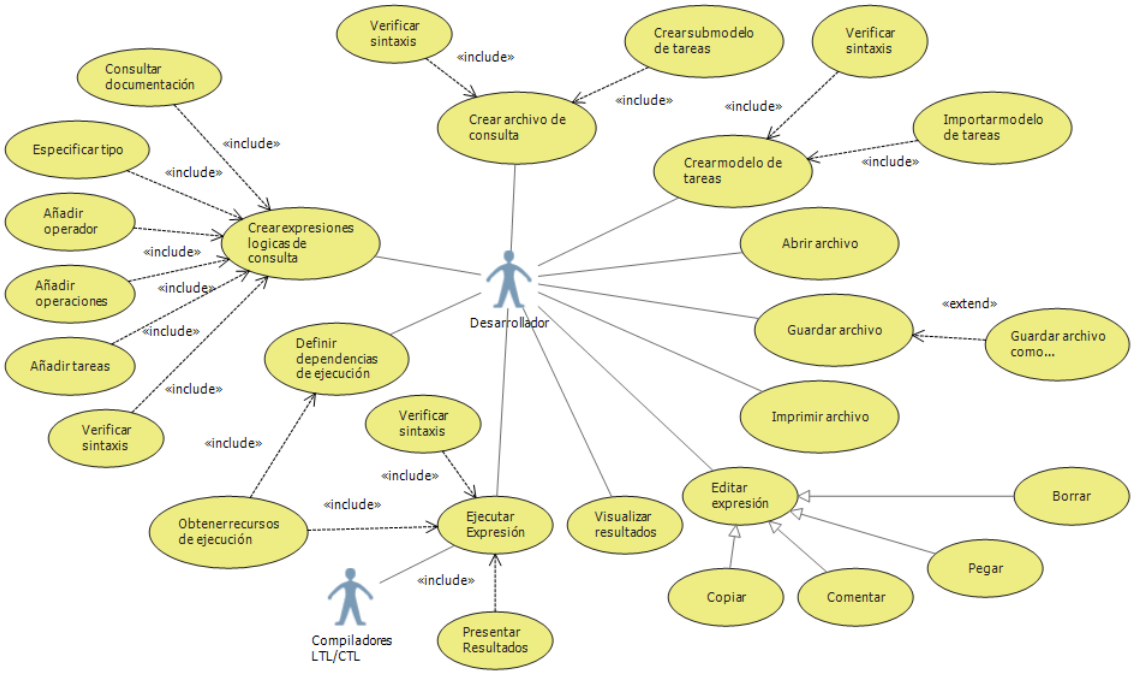

Figura 3.1: Diagrama de casos de uso de la herramienta

En este diagrama se puede observar que se añaden casos de uso que complementan el desarrollo de otros (inclusión); además de que los dos actores del sistema interactúan en la ejecución de las expresiones compartiendo información, por ello el formato que se utilice en la comunicación debe ser estándar en ambos sentidos.

3.3. Análisis

En el Proceso Unificado este flujo se realiza con el objetivo de refinar y estructurar de una mejor manera los requisitos establecidos en el flujo anterior. El lenguaje que se utiliza se basa en un modelo de objetos conceptual denominado modelo de análisis [16].

El modelo de análisis describe los requisitos del sistema utilizando el lenguaje del desarrollador y confiere una vista interna del sistema. Este enfoque ayuda a comprender como debería estructurarse, diseñarse e implementarse el sistema.

Del modelo de análisis se retomaron para el proyecto TFI 3 artefactos útiles: *clases de análisis, paquetes de análisis y realización de casos de uso.*

3.3.1. Artefacto 1: Clases de análisis

Las clases de análisis representan una abstracción de una o varias clases y/o subsistemas del diseño del sistema. Esta abstracción se centra en el tratamiento de requisitos funcionales y como

tal define atributos conceptuales de alto nivel [16].

Tomando como punto de partida el diagrama general de casos de uso y la descripción a detalle, se seleccionaron las clases de análisis identificando sustantivos, frases nominales y categorías del sistema [17]. Más tarde estas se categorizaron en los 3 tipos de clases: entity, boundary y control.

Lista de categorías y clases

Contenedor:

- Modelo (entity): Administra la información referente a modelos algebraicos de diagramas de tareas.
- Consultor (entity): Gestiona la información referente a las expresiones lógicas temporales (consultas).
- Librería (entity): Administrar los recursos de ejecución o bien los submodelos de consulta.
- Editor (boundary): Se utiliza para presentar los elementos gráficos del entorno.
- BarraHerramientas (boundary): Se utiliza para agrupar componentes que cumplen una función dentro del editor.

Cosas del Contenedor:

- Expresión (entity): Administra unidades de consultas (expresiones) construidas por el usuario.
- Dialogo (boundary): Componente que muestra el estado del sistema.
- Vista (boundary): Elemento que presenta los resultados de ejecución de las consultas generadas por el usuario.
- Icono (boundary): Elemento que representa diversas acciones individuales en el editor.

Transacciones

- Ejecución (control): Coordina la ejecución de las expresiones en lógicas temporales construidas por el usuario.
- Configura (control): Controla las librerías y dependencias de ejecución.
- AnalizadorS (control): Gestiona la verificación de sintaxis en modelos y consultores.
- Procesador (control): Administra el proceso de análisis de resultados.

Referencias/Manuales

- Helper (entity): Utilizado para administrar la ayuda en el sistema.

Otros

- Parser (control): Gestiona el proceso de adquisición de datos del modelo/consultor hacia editor.
- GestorArchivos (control): Administra las operaciones básicas que pueden realizarse sobre los archivos: crear e importar.
- Dibujo (control): Realiza el proceso de representación de resultados.

3.3.2. Artefacto 2: realización de casos de uso-análisis

Una realización de casos de uso-análisis es una colaboración dentro del modelo de análisis que describe cómo se lleva a cabo y se ejecuta un caso de uso determinado en términos de las clases del análisis y de sus objetos del análisis en interacción [16].

Este artefacto modela clases individuales basándose en los casos de uso principales y las clases identificadas. En nuestro sistema fue mucho más representativo expresar las realizaciones de casos de uso principales de manera grupal. El diagrama de clases de análisis que represento nuestro producto de software fue el siguiente:

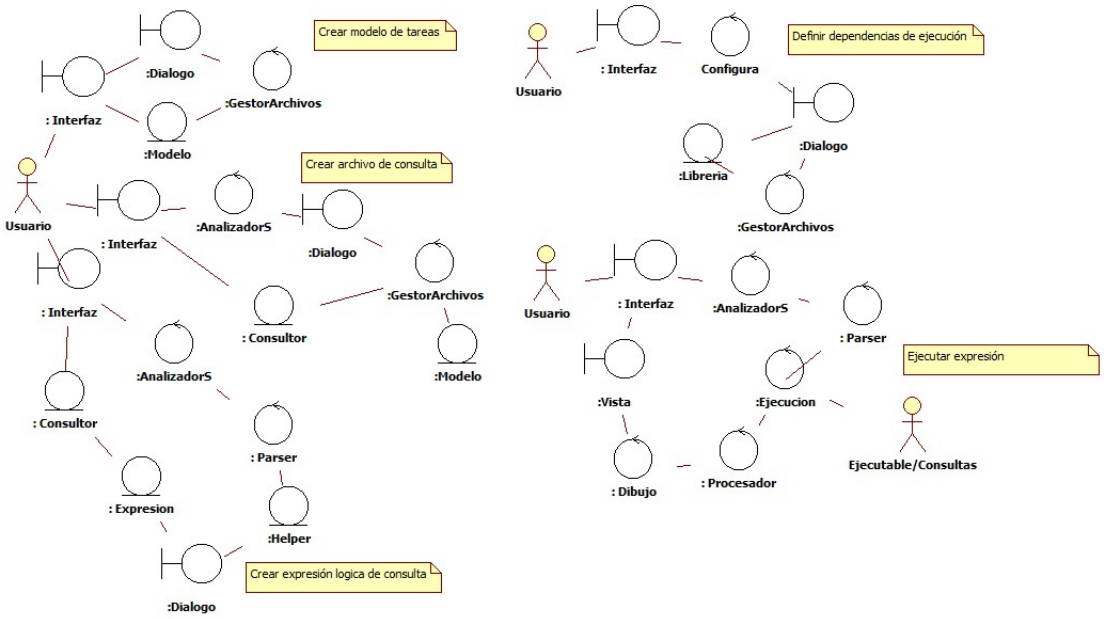

Figura 3.2: Diagrama de clases de los casos de uso-análisis del proyecto

Continuando con el proceso, para identificar las responsabilidades e interacciones sobre los objetos se diseñaron los diagramas de colaboración reutilizando la notación anterior. En este nuevo diagrama se añadieron algunas interacciones que no se consideraron en las clases de análisis.

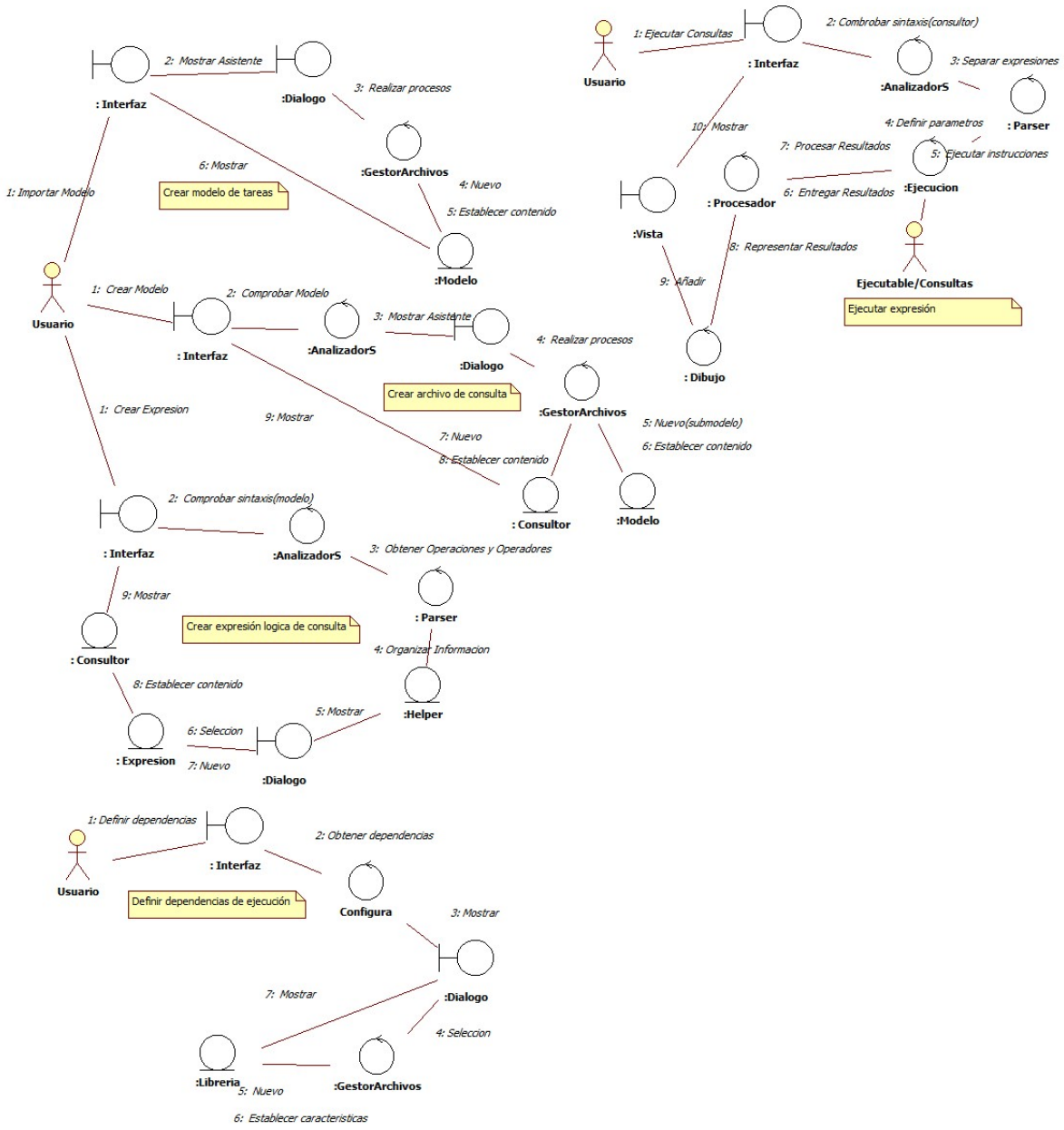

Figura 3.3: Diagrama de colaboración de la realización de casos de uso generales

3.3.2.1. Requisitos Adicionales

En el proceso de realización de las clases de análisis se determinó que los siguientes requisitos impactan en el sistema:

- Desarrollo específico para el IDE Eclipse (especificación del cliente).
- Esquema de componentes: plug-in.

- El sistema debe representar los resultados de verificación de tareas en forma de árbol, debido a que es la estructura que permite entender los resultados fácilmente.
- La ayuda que ofrezca el sistema debe cumplir propiedades de disponibilidad y visibilidad.
- La documentación de expresiones debe ejemplificar operaciones considerando el modelo base del proyecto para facilitar el entendimiento y la retención en el usuario.

3.3.3. Artefacto 3: Paquete de Análisis

Los paquetes de análisis proporcionan un medio para organizar los artefactos del modelo de análisis en piezas manejables. Un paquete de análisis puede agrupar las clases de análisis y la realización de casos de uso; esta agrupación permite separar y entender el sistema desde diferentes enfoques [16].

Para la creación del paquete de análisis del proyecto, se asignaron el mayor número de casos de uso a un grupo (paquete) específico. Esta agrupación de casos de uso se realizó considerando los siguientes criterios:

- Los casos de uso de un paquete deben referenciar a un determinado proceso de negocio.
- Un paquete hace referencia a un determinado actor por las actividades que involucra.
- Los paquetes del análisis deben crearse basándose en los requisitos funcionales y en el dominio del problema.
- Los paquetes del análisis por lo general se convierten en subsistemas del sistema.

La primera aproximación a la creación de paquetes de uso resultó en el esquema de la figura 3.4, el cual a medida que el desarrollo evolucionó, generó una estructura más refinada.

3.3.4. Análisis de Clases

Hasta este punto se han identificado clases de análisis y responsabilidades, así como actores y casos de uso; para continuar fue necesario generar un esquema de cómo deberían estructurarse las clases y sus atributos. Para identificar atributos se consideraron las siguientes observaciones:

- El nombre de un atributo debe ser un nombre.
- El tipo de los atributos debería ser conceptual en el análisis y, si es posible no debe restringirse por el entorno de implementación.
- Al decidir el tipo, se debe intentar reutilizar tipos existentes.
- Los atributos de las clases de interfaz suelen representar propiedades de una interfaz de comunicación.

Partiendo de las clases de análisis detectadas y del diagrama de clases de los casos de uso, los atributos de cada clase resultantes se muestran en la figura 3.5.

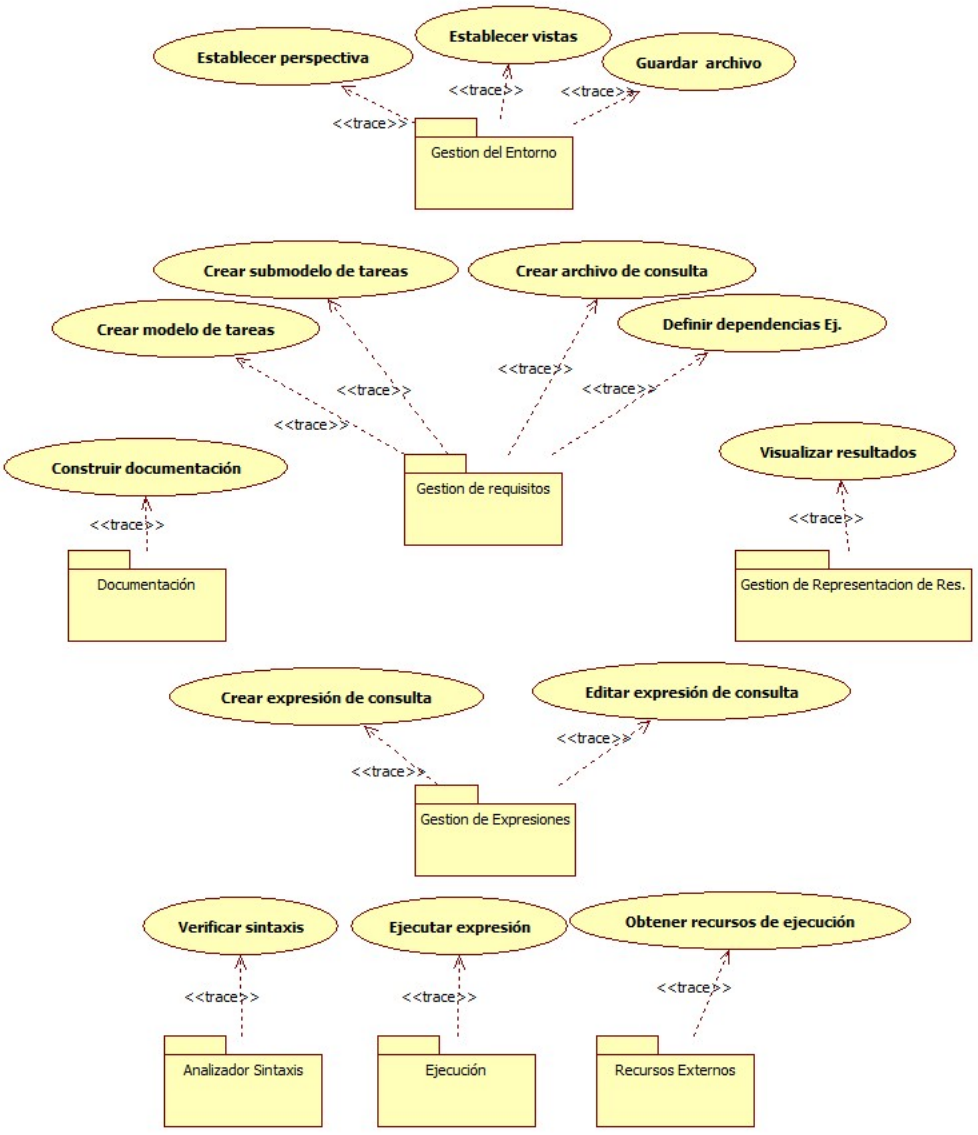

Figura 3.4: *Task Flow Diagrams Verification Interface:* identificación de paquetes de análisis a partir de los casos de uso

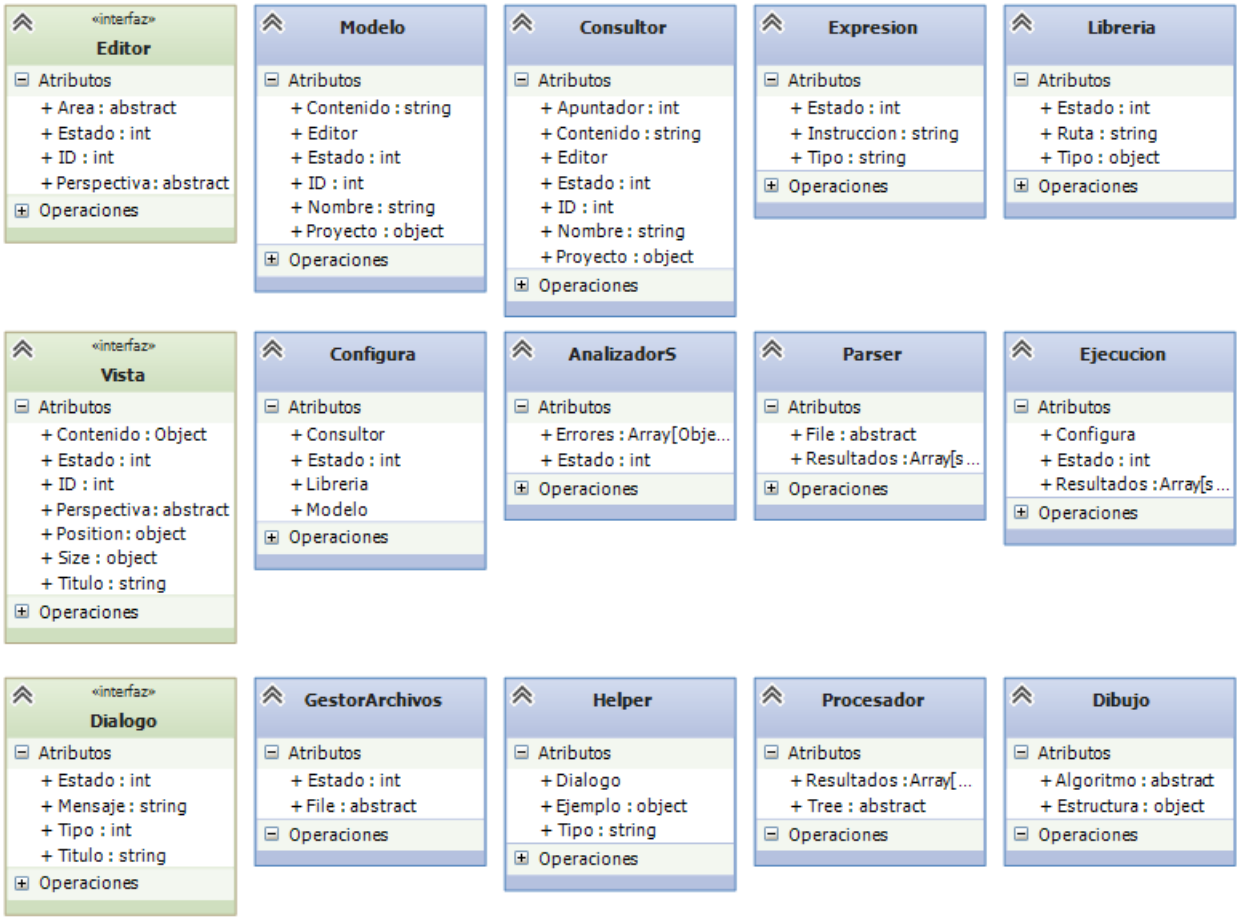

Figura 3.5: Clases de análisis con sus respectivos atributos.

3.4. Diseño

El Proceso Unificado establece esta etapa con el objetivo de adquirir una comprensión en profundidad de los aspectos relacionados con los requisitos no funcionales y restricciones relacionadas con los lenguajes de programación, componentes (sistemas externos), sistemas operativos y tecnologías de distribución, ocurrencia e interfaz de usuario [16].

Para especificar los componentes de diseño se utilizaron dos artefactos: Clases de Diseño e Interfaz. En ambos se retomó el esquema de clase de diseño/paquetes y se estableció el modelo de interacción de componentes.

3.4.1. Clases de Diseño

Una clase de diseño es una abstracción de la implantación del sistema en términos de operaciones, parámetros, atributos y tipos de clases enfocados al lenguaje de programación candidato.

El lenguaje de programación utilizado para describir las clases de diseño en el proyecto es Java, el cual constituye la base del conjunto de herramientas del entorno Eclipse IDE. Considerado que el lenguaje es Orientado a Objetos, las clases de diseño se expresan en términos de polimorfismo, abstracción y composición. El esquema resultante se muestra en la figura 3.6.

El esquema de clases de diseño de la figura 3.6 mostró que los componentes interactuaban en diferentes niveles del sistema. sin embargo, aún no quedaba claro cómo deberían agruparse/establecerse estos niveles, por lo cual considerando la figura 3.6 se establecieron 4 capas del sistema:

1. Capa de Interacción y Control: Agrupación de funcionalidades en subsistemas específicos de la problemática a resolver.
2. Capa intermedia y de software del sistema: Constituye los cimientos del sistema sobre el cual se apoyan los desarrollos, por ejemplo: sistemas operativos, sistemas de gestión de base de datos, etc.

3.4.2. Artefacto Interfaz

Las interfaces se utilizan para especificar las operaciones que proporcionan las clases y los subsistemas del diseño. En términos generales constituyen una forma de separar la especificación de la funcionalidad en función de implementaciones y métodos. La mayoría de estas interfaces se convierten en requisitos e instrumentos de sincronización entre los subsistemas [16].

El proceso unificado establece que la forma de identificar interfaces es basándose en la accesibilidad desde fuera de los subsistemas. Para identificar las interfaces iniciales del proyecto se identificaron los subsistemas generales, los cuales son los siguientes:

1. Capa de interacción
    - **Subsistema**: Gestión del entorno
2. Capa de control

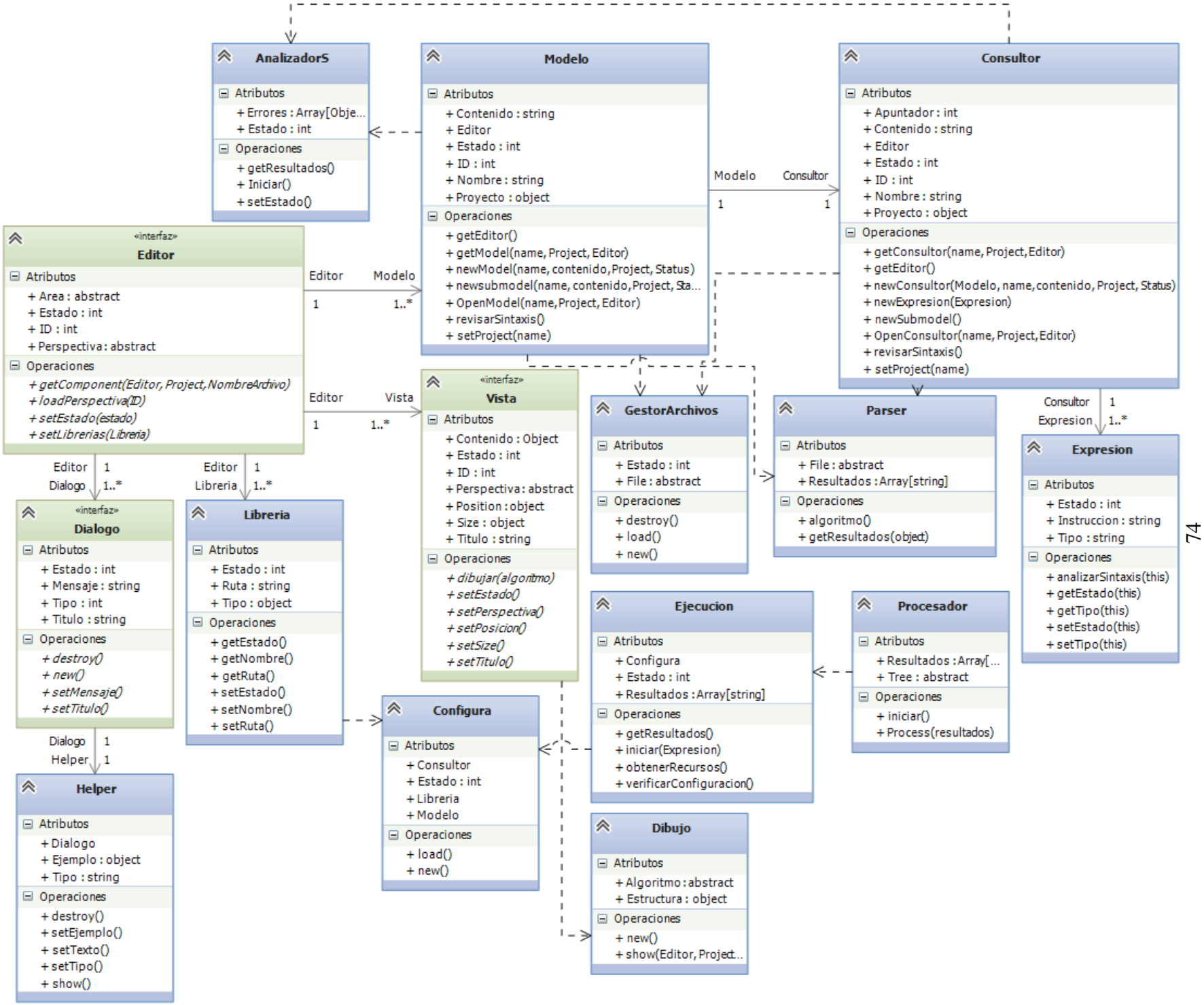

Figura 3.6: Clases de diseño, atributos y relaciones

- **Subsistema**: Gestión de requisitos
- **Subsistema:** Documentación
- **Subsistema:** Gestión de Representación Res.
- **Subsistema:** Gestión de Expresiones

3. Capa intermedia
   - **Subsistema:** Analizador Sintáctico
   - **Subsistema:** Ejecución
4. Capa de Software del sistema
   - **Subsistema:** Recursos Externos

El esquema de subsistemas y dependencias se muestra a continuación:

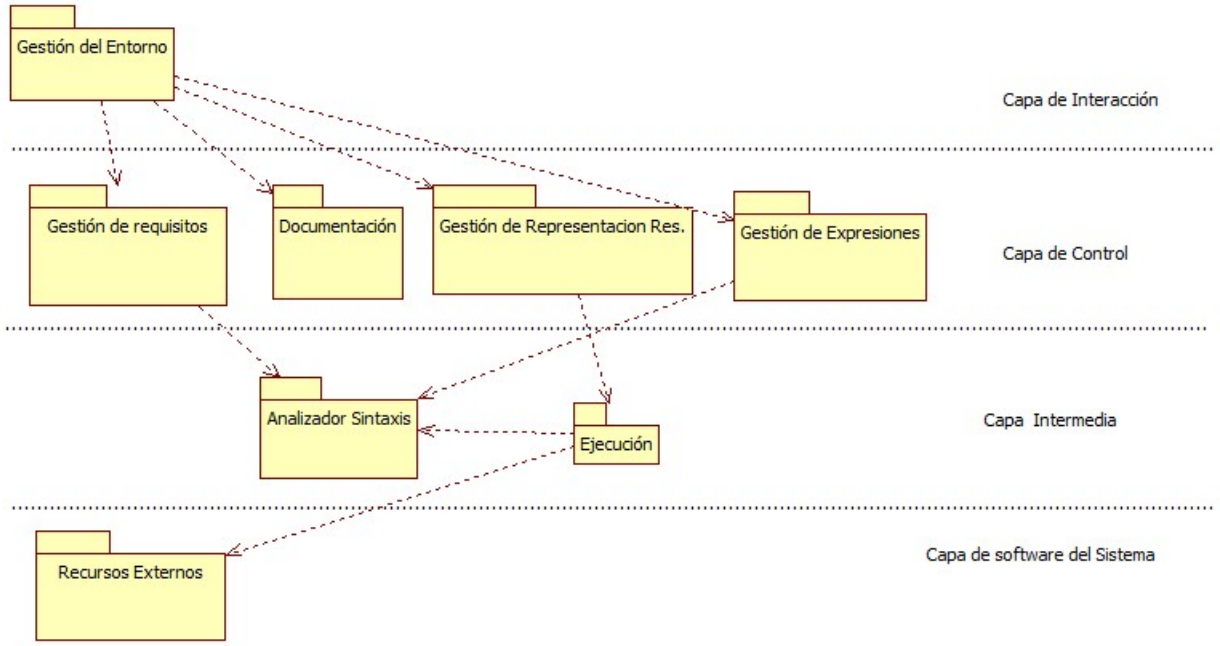

Figura 3.7: Dependencias y capas de los subsistemas del proyecto

Del esquema se observa que las dependencias incidentes sobre paquetes definen las posibles interfaces del sistema; por ello las opciones candidatas que se establecieron fueron las siguientes: gestión de requisitos, representación de documentación, representación de resultados, construcción de expresiones, analizador sintáctico, ejecución y recursos externos.

Teniendo como base el artefacto interfaz, casos de uso, paquetes y las clases de diseño la implementación del sistema se describe en el siguiente capítulo.

Capítulo 4

Implementación

4.1. Introducción

La etapa de implementación en el Proceso Unificado describe como los elementos del modelo de diseño, las clases y paquetes se transforman en código fuente, scripts, binarios, ejecutables y objetos similares.

Nuestro propósito en este capítulo es la implementación del sistema *Task Flow Diagrams Verification Interface* mediante un enfoque incremental, construyendo clases y subsistemas resultantes de la etapa de diseño. A continuación se describen los artefactos utilizados, su aplicación en el proyecto y el desarrollo de la fase de implementación.

4.2. Definición de los Artefactos

En esta etapa del proyecto los artefactos desempeñaron un papel determinante para guiar la construcción del sistema. De la especificación del Proceso Unificado existen 3 artefactos utilizados para el desarrollo del proyecto: Modelo de Implementación, Componentes y Plan de Integración de Construcciones.

4.2.1. Modelo de Implementación

Una forma de organizar y estructurar los componentes de un sistema es el modelo de implementación. A través de este modelo se representa un sistema de nivel superior en términos de componentes pequeños y manejables [16]; estructuralmente el modelo de implementación resultante para el sistema TFI se muestra en la figura 4.1.

Como podemos observar el modelo de implementación (figura 4.1) establece 7 subsistemas que además se comportan como componentes. La integración de los subsistemas definen las funcionalidades del proyecto (TFI).

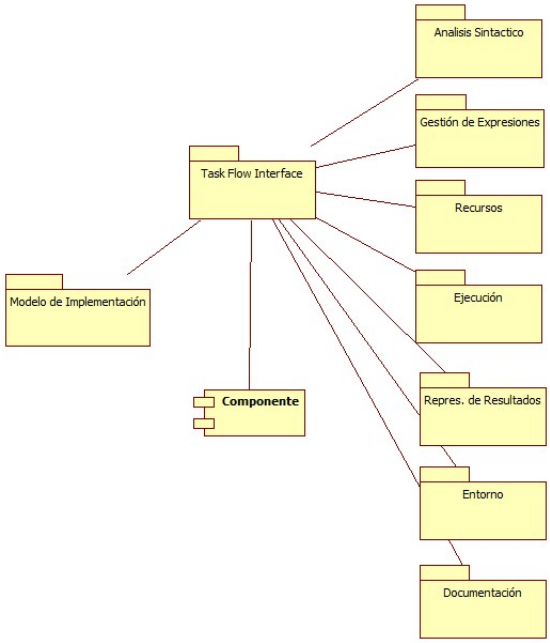

Figura 4.1: Modelo de implementación del sistema TFI. En este caso la cardinalidad de los subsistemas no se añade debido a que se utilizara 1-1

4.2.2. Artefacto Componente

Un componente es el empaquetamiento físico de los elementos de un modelo, el cual se define mediante un estereotipo [16]. Los estereotipos estándares más comunes son: <<ejecutable>, <<file>> y <<library>. En el proyecto actual se utilizarán los siguientes:

1. <<*Ejecutable*>>: Es un programa que puede ser ejecutado en un nodo. En el proyecto son los recursos externos, es decir los compiladores (*exe*) que ejecutan las expresiones que verifican propiedades en los modelos. Los compiladores *exe* son los componentes que describe el autor [9] en su capítulo 8.
2. <<*file*>>: Es un componente que contiene código fuente ó datos. En el proyecto corresponde a todos los archivos de código java (*.java).
3. <<*library*>>: Es una librería estática o dinámica. Particularmente los archivos de configuración que genere el sistema (*.ini) son ejemplos de estos.
4. <<*documento*>>: Es un documento general, se puede hacer referencia en el proyecto a los archivos del modelo algebraico (tfa) y consultor (tfq) que construye el sistema en fase de producción.

Para describir el ámbito de los estereotipos, podemos resumir que el objetivo de la implementación es expresar las funcionalidades del sistema en términos de código fuente (*.java). Las cuales deberán construir archivos algebraicos modelo (tfa) y consultores (tfq). Además de permitir la ejecución de expresiones, reuniendo los recursos externos (*.exe y *.ini) y traduciendo las especificaciones en resultados visuales.

Con ayuda del modelo de clases del capítulo anterior y los estereotipos seleccionados, la representación de la relación entre los componentes del proyecto se resume en la figura 4.2.

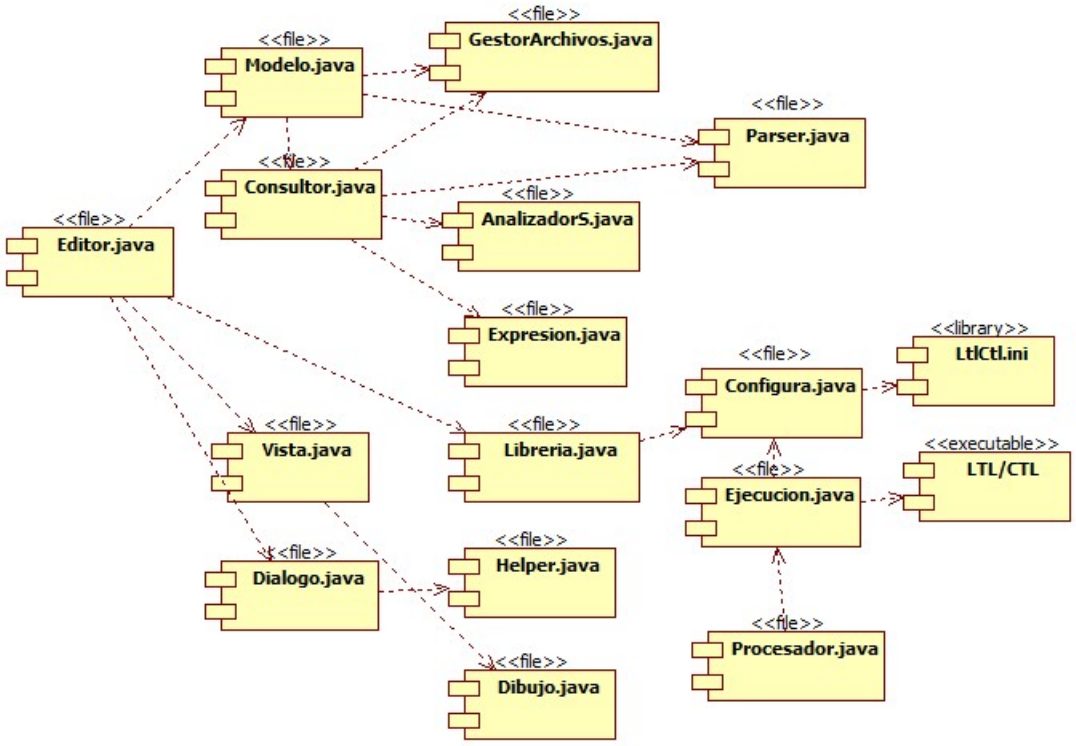

Figura 4.2: Diagrama general de componentes del sistema.

La figura 4.2, es la guía de implementación que expresa lo que debemos construir en términos de archivos de código fuente, configuración y librerías; también expresan el alcance del proyecto y las reglas de desarrollo. En términos generales los componentes serán nuestros entregables (fuente y scripts) del proyecto.

4.2.3. Artefacto Plan de Integración de Construcciones

Al inicio del capítulo se especificó que el desarrollo del sistema se realizará bajo un enfoque incremental, es decir a través iteraciones. Para definir las iteraciones necesarias se utilizó el modelo de implementación, por lo que se planearon 7 versiones del proyecto (7 subsistemas). La descripción general de cada versión se describe a continuación:

- **Liberación 1:** Implementación de la verificación sintáctica en tiempo real de los modelos (álgebra de tareas) y expresiones de consulta.
- **Liberación 2:** Implementación que incluye la funcionalidad de crear expresiones guiadas (LTL y CTL) basadas en un modelo previamente definido.

- **Liberación 3:** Implementación que se encarga de administrar los recursos de configuración internos y externos del proyecto.
- **Liberación 4:** Implementación que libera la funcionalidad de enlazar los recursos y ejecutar las expresiones de consulta.
- **Liberación 5:** Implementación que representa los resultados resultantes de la ejecución de expresiones.
- **Liberación 6:** Implementación referente a la validación de excepciones e integración de los mensajes (informativos, error) en los procesos de propiedades del álgebra de tareas.
- **Liberación 7:** Implementación referente al desarrollo de la interfaz de usuario, integración de las funcionalidades y liberación del proyecto.

De manera detallada las características a implementarse en cada una de las etapas son las siguientes:

1. **Construcción 1: Modelo, verificador y análisis sintáctico**
   - *a*) Crear modelo algebraico (*.tfa)
   - *b*) Importar modelo algebraico (*.tfa)
   - *c*) Crear archivo de consulta (*.tfq)
     1) Crear biblioteca de tareas (libreria_*.tfq).
   - *d*) Verificador sintáctico de modelos (álgebra)
   - *e*) Verificador sintáctico de expresiones (LTL y CTL)
2. **Construcción 2: Construcción de expresiones LTL y CTL.**
   - *a*) Añadir tareas.
   - *b*) Añadir condiciones true, false.
   - *c*) Agrupar expresiones.
   - *d*) Operadores LTL: Not, And , Or, Implication, Next, Globally, Finally, Until, Weak-Until, release.
   - *e*) Operadores CTL: Not, And, Or, Implication, ANext, AFinally, AGlobally, AUntil, ENext, EFinally, EGlobally, ExistUntil.
3. **Construcción 3: Configuración de recursos.**
   - *a*) Crear archivo de configuración: LtlCtl.ini
4. **Construcción 4: Ejecución de expresiones.**
   - *a*) Enlazar recursos.
   - *b*) Ejecutar Expresiones (Proceso).
5. **Construcción 5: Representación de resultados.**
   - *a*) Parser de resultados.
   - *b*) Transformación de resultados al modelo visual.

c) Representación de resultados (componentes visuales Eclipse).

6. **Construcción 6: Excepciones y especificación de mensajes.**

   a) Manejo de mensajes (alertas y diálogos en Eclipse)

   b) Manejo de condiciones en funciones.

   c) Manejo de Excepciones en los procesos.

7. **Construcción 7: Interfaz de usuario y liberación del Proyecto**

   a) Prototipado de la Interfaz.

   b) Integración de componentes en el entorno.

   c) Empaquetado del plug-in.

   d) Liberación del proyecto.

4.3. Flujo de Trabajo

De acuerdo a la especificación del Proceso Unificado [16] cada subsistema que se implemente debe evaluarse a medida que el proyecto avanza (figura 4.3); por ello en nuestro proyecto a medida que se liberaba una funcionalidad también se evaluaba la función y si esta cumplía el requerimiento se pasaba a la implementación siguiente.

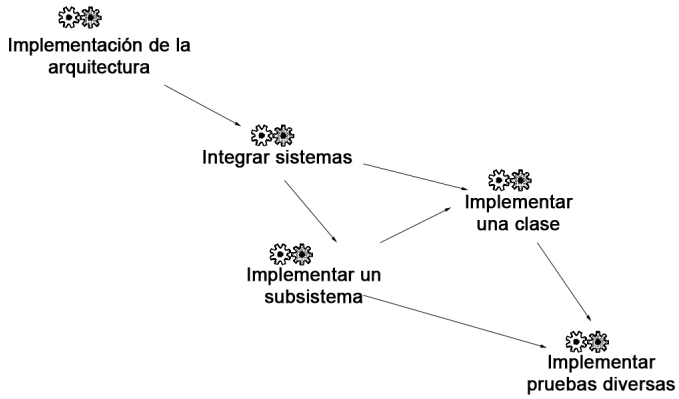

Figura 4.3: Flujo convencional de trabajo en la etapa de implementación

Es importante mencionar que sin importar la etapa de construcción en la cual nos encontráramos; la codificación, estandarización e integración se realizó utilizando los componentes y superclases de la especificación Plug-in Development Environment (PDE).

4.4. Restricciones de Clases y Componentes

Antes de comenzar a implementar la primera etapa del sistema fue necesario establecer una lista de restricciones a cumplirse en la invocación de las funcionalidades. La mayoría de restricciones establecidadas describian situaciones a las cuales debe responder nuestro sistema:

- **Restricción de área:** Toda acción referente al proceso de construcción y ejecución de expresiones lógicas (LTL y CTL) debe provenir de un componente de tipo AbstractTextEditor. El cual es un objeto (editor) de texto que se instancia en el área de edición de Eclipse.
- **Restricción de tipo de archivo:** Toda acción se ejecutará siempre y cuando el archivo en primer plano sea de tipo tfa (modelo) o tfq (expresiones).
- **Restricción de sintaxis:** El proceso de ejecución de expresiones solo podrá realizarse si los archivos de modelo, librería de tareas y expresiones no contienen errores de sintaxis. Además la configuración de los recursos externos debe estar configurada adecuadamente.
- **Restricción de acciones:** Los archivos modelo de primer plano (activos) solo podrá instanciarse la funcionalidad de Nuevo Archivo de Consulta. En el caso de los consultores se instanciarán todas las tareas excepto la anterior.
- **Restricción de ámbito:** Si los archivos en primer plano son de tipo desconocido para el proyecto, las funcionalidades no se mostrarán.
- **Restricciones de ejecución:** Para la ejecución de expresiones de consulta, la funcionalidad deberá invocarse desde un archivo consultor y los recursos que se utilicen deben estar definidos previamente en el archivo de configuración LtlCtl.ini

4.5. Implementación del Sistema Task Flow Diagrams Verification Interface (TFI)

Task Flow Diagrams Verification Interface es un proyecto centrado en los procesos de construcción de expresiones lógicas (LTL y CTL), ejecución y representación de resultados. En el capítulo 2 se describió el flujo convencional de un proceso de verificación de especificaciones de un modelo algebraico en el Método Discovery, en el cual la primera etapa consistía en la construcción del modelo y a continuación la construcción de expresiones. Aun cuando el proyecto TFI no tomará como punto de partida la construcción del modelo era importante dar soporte a esta actividad.

TFI se implementó en función de las características del entorno, utilizando los recursos disponibles y extendiendo sus capacidades como lo haría cualquier plug-in desarrollado en Eclipse. Al finalizar la etapa de implementación el artefacto final fue un sistema empaquetado disponible como plug-in que utiliza las metáforas de perspectivas, vistas, consolas, errores y otros componentes del entorno.

A continuación se describe el desarrollo del proyecto basado en las etapas previamente des-

critas.

4.5.1. Implementación 1: Construcción del Modelo, Consultor y Analizadores Sintácticos

Objetivos: El objetivo de esta etapa consistió en implementar las funcionalidades que permiten las siguientes tareas: construir archivos modelo (tfa), archivos consultores (tfq) y verificadores sintácticos para ambas especificaciones.

Construir un archivo en el entorno de desarrollo de plug-ins (PDE) se realiza utilizando puntos de extensión. Los puntos de extensión son funcionalidades ya implementadas en el entorno Eclipse, que mediante modificación de código extienden sus características. En nuestro proyecto los puntos de extensión utilizados fueron newWizards e importWizards disponibles en el paquete org.eclipse.ui.

Una vez construidos los puntos de extensión fue necesario modificar el código fuente para dar soporte a la creación e importación de archivos en formato tfa (task flow algebra). Cuando se crean puntos de extensión basados en newWizards e importWizards es necesario asignar una categoría, en nuestro caso la categoría asociada fue **Model Checking**, lo que significa que cuando se cree/importe un nuevo archivo tfa a un proyecto específico la opción estará disponible en la categoría mencionada. El resultado de la funcionalidad se puede visualizar en la figura 4.4.

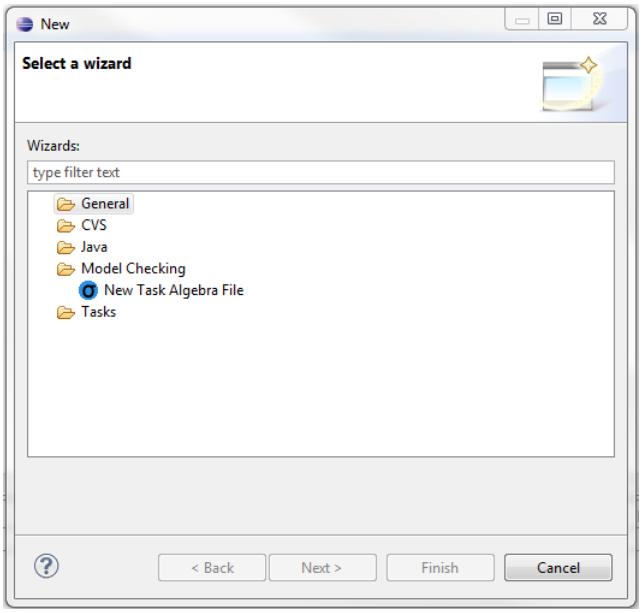

Figura 4.4: Funcionalidad crear nuevo archivo modelo tfa, integrado al entorno Eclipse

Continuando con la implementación, en el caso de la creación de los archivos de consulta tfq (*task flow query*) no se utilizó el esquema de extensiones debido a que la invocación depende de las restricciones de clase y componentes definidas anteriormente. En su lugar para implementar

esta funcionalidad se utilizó el flujo de eventos detallados en el caso de uso *crear archivo de consulta* del capítulo 3, el cual se resume en los siguientes pasos:

1. **Invocación de la funcionalidad:** La invocación de la funcionalidad se realiza desde un archivo modelo algebraico abierto.
2. **Extracción:** Del archivo modelo se extraen las tareas definidas.
3. **Creación de archivos intermedia:** Se crea un archivo librería de tareas con el nombre "**library**_*"nombre_del_archivo_modelo*"**.tfq**", en el cual se incluyen las tareas obtenidas y un encabezado de paquete, un ejemplo de esta estructura es la siguiente:

   package org.task.algebra.library_Model.tfq {

   Task A1

   Task A2

   Task A3...

   Task An

   }

4. **Creación de Consultor:** Se crea el archivo de consulta con el nombre *nombre_del_archivo_modelo*"**.tfq**". Añadiendo la relación de dependencia al archivo biblioteca anterior mediante la instrucción *import.* (figura 4.5).

*import org.task.algebra.library_nombre_del_modelo.tfq.**
*/** Use LTL: , Or CTL: for start a new Query**/*
LTL:

Figura 4.5: Estructura de la cabecera de un archivo de consulta.

El motivo por el cual se creó un archivo librería intermedio se debe a que es más rápido obtener las tareas de un archivo simplificado que del modelo nuevamente, además este concepto forma parte de la lógica de estructura de clases en java. En el caso del archivo consultor fue necesario además establecer dos excepciones de estado:

- Cuando la funcionalidad se invoque, si el archivo consultor ya existe este no debe modificarse para evitar pérdidas de información.
- El archivo librería de tareas debe actualizarse siempre que el modelo cambie o bien siempre que se invoque la funcionalidad.

Una vista de los resultados de implementación de esta funcionalidad se muestran en la figura 4.6:

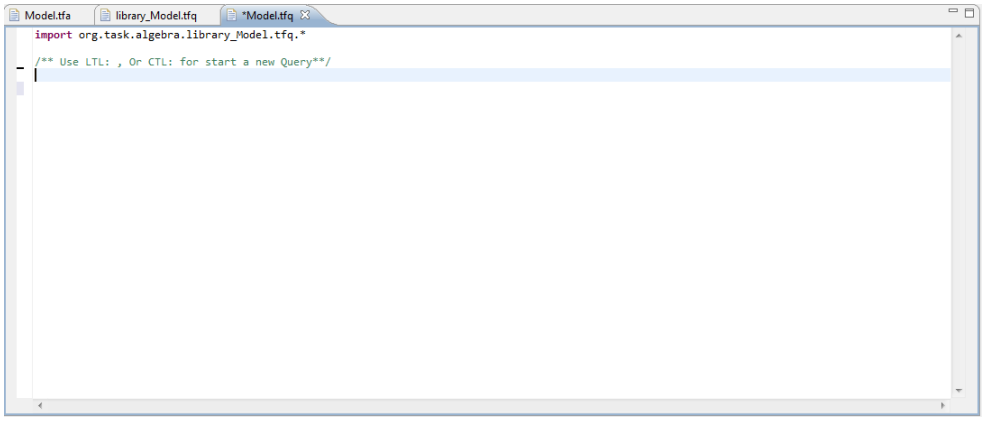

Figura 4.6: Implementación de la funcionalidad nuevo archivo de consulta

La siguiente funcionalidad a implementar fue la verificación sintáctica de los modelos y consultores. La verificación sintáctica en tiempo real es una de las características más conocidas en el entorno Eclipse, sin embargo para construirla se requiere desarrollar un compilador. Desarrollar un compilador desde cero implica un proceso que requiere tiempo y esfuerzo, pero aprovechando las características del entorno esta tarea se simplificó.

La característica que permitió simplificar la codificación del compilador fue el esquema basado en plug-ins; en Eclipse muchas funcionalidades pueden heredar de otros plug-ins e integrarse como si se tratara de uno solo. En el caso de la construcción de compiladores el plug-in seleccionado fue Xtext.

Xtext permite construir compiladores escalables y robustos de manera sencilla, encapsulando algunas etapas del proceso. Xtext por definición es una solución simple para crear lenguajes especializados que aceleran el desarrollo, mantienen la calidad, reducen costos y complejidad en la construcción de verificadores sintácticos [11]. Xtext permite transformar gramáticas EBNF en editores de texto de lenguajes especializados y al estar basados en especificaciones de gramáticas EBNF su construcción resulta sencilla.

Retomando el proceso de construcción del verificador de sintaxis, el primer requisito para implementar nuestro verificador sintáctico en Xtext fue contar con una gramática del lenguaje de dominio del álgebra de tareas y de las expresiones de consulta LTL y CTL. En el primer caso la gramática del álgebra de tareas esta específicada en el capítulo 2 de este documento.

La gramática presentada en el capítulo 2 es una gramática de tipo BNF, esta se muestra a continuación (figura 4.7):

```
Activity  ::=  Epsilon                    -- empty activity
         | Sigma                         -- σ succeed
         | Phi                           -- ϕ fail
         | Task                          -- a single task
         | Activity ; Activity           -- a sequence of activity
         | Activity + Activity           -- a selection of activity
         | Activity || Activity          -- parallel activity
         | Mu.x(Activity ; Epsilon + x)  -- until-loop activity
         | Mu.x(Epsilon + Activity ; x)  -- while-loop activity

Task::= Simple                           -- a simple task
         | { Activity }                  -- encapsulated activity
```

Figura 4.7: Gramática BNF del álgebra de tareas

Al analizar la especificación anterior se detecto que la gramática presentaba recursividad. La recursividad es una característica que como tal no puede traducirse a Xtext, por ello fue necesario simplificar la gramática utilizando las técnicas de eliminación recursiva directa y refactorización descritas en la sección 2.3.4 .

Aplicando estas técnicas los resultados fueron los siguientes (figura 4.8):

```
Unit::=  Epsilon
         | Sigma
         | Phi
         | Task

Task::=  {Activity}
            | ( Activity )
            | TSIMPLE

Ciclos::=   ( 'Epsilon' '+' Activity ';') | Until

Until::= Activity ';' 'Epsilon' '+'

Activity::=  ( Unit
               | Mu.x '(' Ciclos   x ')'
             )  Activity_prima?


Activity_prima::=    ( '+' | ';' | '||') Activity
```

Figura 4.8: Gramática EBNF simplificada del algebra de tareas.

La gramática obtenida se tradujo a la específicación Xtext, su implementación se muestra en la figura 4.9. Una vez finalizada la implementación del verificador sintáctico del modelo, la siguiente construcción fue la implementación del verificador sintáctico de expresiones.

grammar org.plug.algebratareas.Algebratareas with org.eclipse.xtext.common.Terminals
generate algebratareas "http://www.plug.org/algebratareas/Algebratareas"

Model: (elements+=Activity)*;

/*Gramatica G (T,NT,S0,P) */
terminal TSIMPLE: ('a'..'z') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

Unit: 'Epsilon'|'Sigma'|'Phi'|Task;
Task: '{' Activity '}' | ' (' Activity ')' | TSIMPLE;
Ciclos: ('Epsilon' '+' Activity ';') | Until ;
Until: Activity ';' 'Epsilon' '+';
Activity: (Unit | 'Mu.' 'x' ' (' Ciclos 'x' ')') Activity_prima?;
Activity_prima: ('+'|';'|'||') Activity;

Figura 4.9: Implementación del verificador sintáctico del álgebra de tareas en modelos para archivos tfa

Como sabemos las expresiones lógicas utilizadas en el proyecto son específicaciones en Lógica Lineal temporal y Computacional en Árbol; ambos tipos permiten verificar propiedades sobre un modelo particular y cada una de ellas utiliza diversas operaciones, las cuales fueron descritas en el capítulo 2.

La verificación sintáctica de expresiones LTL y CTL de consulta, al igual que el modelo, requiere partir de una especificación gramatical, la cual se tomó del anexo [9]. En esta especificación se define que toda expresión LTL y CTL se estructura bajo el esquema presentado en la figura 4.10.

Lógica Lineal Temporal (1)
LTL_exp : *true* | *false* | Simple_expr
Simple_exp: *simpleTask* | Logic_expr
Logic_expr: *not* LTL_exp | LTL_exp *and* LTL_exp | LTL_expr *or* LTL_expr | LTL_expr *imp* LTL_expr | Unary_temp_expr
Unary_temp_expr: *X* LTL_expr | *G* LTL_expr | *F* LTL_expr | Bin_temp_expr
Bin_temp_expr: LTL_expr *U* LTL_expr | LTL_expr *W* LTL_expr | LTL_expr *R* LTL_expr | *(* LTL_expr *)*
. . . . . .
Lógica Computacional en Árbol (2)
CTL_exp : *true* | *false* | Simple_expr
Simple_exp: *simpleTask* | Logic_expr
Logic_expr: *not* CTL_exp | CTL *and* CTL_exp | CTL_expr *or* CTL_expr | CTL_expr *imp* CTL_expr | A_temp_expr
A_temp_expr: *AX* CTL_expr | *AG* CTL_expr | *AF* CTL_expr | *A* CTL_expr *U* CTL_expr | E_temp_expr
E_temp_expr: *EX* CTL_expr | *EG* CTL_expr | *EF* CTL_expr |*E* CTL_expr *U* CTL_expr| *(* CTL_expr *)*

Figura 4.10: Gramática de las operaciones LTL y CTL basada en el compilador del autor [9]

Aplicando nuevamente las técnicas de eliminación de recursividad directa e indirecta y refactorización, la gramática se simplificó e implementó en Xtext resultando de la siguiente manera (figura 4.11):

grammar org.plug.Logicastemporales with org.eclipse.xtext.common.Terminals
generate logicastemporales "http://www.plug.org/Logicastemporales"

Domainmodel: (elements += AbstractElement)* ;

terminal ID: '^'? ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

AbstractElement: PackageDeclaration |Import ;
PackageName: ID ('.' ID)* ;
PackageDeclaration: 'package' name = PackageName '{' (elements += DefTasks)* '}' ;

DefTasks: 'Task' name = (ID |'Epsilon'|'Sigma'|'Phi'|'true'|'false');

ImportPackage: PackageName '.*'? ;
Import: 'import' importedNamespace = ImportPackage (elements += DefQuery)*;
Unit: {Unit} (type=[DefTasks]);

Simple_ltl: 'not'|'X'|'G'|'F';
Complex_ltl: 'and'|'or'|'imp'|'U'|'W'|'R';
LTL_exp: (Unit|' (' a0=LTL_exp ')'|Simple_ltl a1=LTL_exp) (a2=LTL_exp_prim)?;
LTL_exp_prim: (Complex_ltl) LTL_exp;

Simple_ctl: 'not'|'AX'|'AG'|'AF'|'EX'|'EG'|'EF';
Complex_ctl: 'and'|'or'|'imp';
Special_ctl: 'A'|'E';
CTL_exp: (Unit|' (' b0=CTL_exp ')'| Simple_ctl b1=CTL_exp| Special_ctl b2=CTL_exp 'U' b3=CTL_exp) (b4=CTL_exp_prim)?;
CTL_exp_prim: (Complex_ctl) CTL_exp;

DefQuery: 'LTL:' LTL_exp | 'CTL:' CTL_exp;

Figura 4.11: Simplificación de la gramática de las operaciones LTL y CTL

En la especificación anterior podemos notar que se añadieron dos indicaciones importantes. La primera es que todo archivo de consulta debe tener una cabecera import a un archivo librería de tareas (*import importedNamespace = ImportPackage (elements += DefQuery)**). La segunda es que toda expresión de consulta deberá iniciar explícitamente con el tipo de expresión, es decir: **LTL o CTL**. Los resultados de implementación de los verificadores sintácticos se muestran en la figura 4.12.

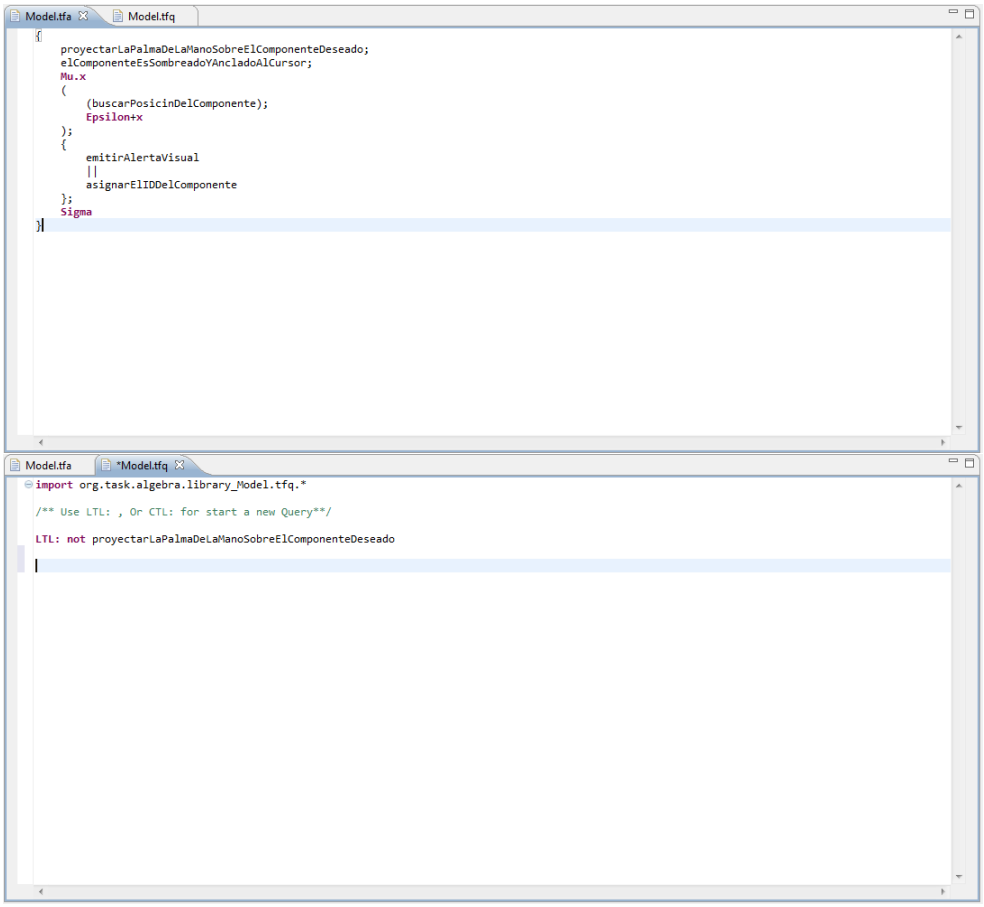

Figura 4.12: Integración de los verificadores sintácticos de archivos tfa y tfq

La implementación de verificadores sintácticos en Xtext tiene una cualidad adicional heredada del entorno Eclipse. Esta cualidad es que cuando un archivo de modelo o expresiones se construye manualmente puede utilizarse el auto completado. La cualidad de auto completado en archivos tfq, tfa se invoca utilizando la combinación de teclas Ctrl+Space.

4.5.2. Implementación 2: Proceso de Construcción de Expresiones Guiadas LTL y CTL

Objetivos: El objetivo en esta etapa consistió en implementar las funcionalidades que permitieran al usuario construir expresiones LTL y CTL de forma guiada.

Hasta este punto de la implementación un usuario cualquiera podría construir expresiones LTL y CTL de forma manual, sin embargo el proceso requería consultar en todo momento el modelo para conocer que tareas pueden incluirse en las expresiones y tener conocimiento de los operadores permitidos en la lógica LTL y CTL. La tarea de construir expresiones lógicas aplicadas a un modelo resultaba aún una tarea complicada, por ello era necesario crear procedimientos visuales guiados que facilitarán esta actividad.

Los elementos principales utilizados para realizar esta implementación fueron los cuadros de dialogo y asistentes, estos componentes del entorno están disponibles en la clase dialogs del paquete org.eclipse.jface. De manera específica el flujo de eventos al que responde la implementación de esta etapa es el siguiente:

1. **Invocación de la funcionalidad:** La invocación de la funcionalidad se realiza desde un archivo consultor abierto.
2. Dependiendo del tipo de Lógica y Operador se realizan los siguientes pasos:

   a) **Obtener tareas:** Del archivo librería se extraen las tareas, este archivo contiene las tareas ya separadas por lo que la actividad resulta sencilla.

   b) **Construir Variantes:** De acuerdo al operador se construyen las variantes aplicadas a las tareas del modelo

   c) **Presentación:** Se presenta al usuario un cuadro de dialogo donde puede seleccionar una variante específica del operador invocado.

3. De acuerdo a la elección del usuario se concatena en el documento la expresión o bien se aplica el operador a la selección actual.

Con ayuda del flujo anterior se construyó la funcionalidad para los operadores LTL y CTL, añadir tareas específica del modelo, específicas lógicas (falso y verdadero)y agrupar expresiones. Es importante mencionar que en el caso de la construcción de las variantes de operadores con dos argumentos (tareas) se utilizó el concepto de combinación, para presentar todas las variantes que podían generarse con las tareas en el operador.

El resultado de esta implementación se puede visualizar en la figura 4.13, el cual permite añadir alguna variante del operador Next (LTL):

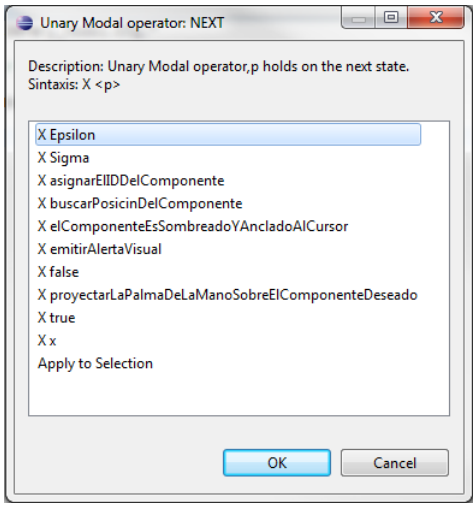

Figura 4.13: Ejemplo visual de la implementación de las variantes del operador Next aplicado al modelo de tareas

4.5.3. Implementación 3: Administración de Recursos de Configuración Externa

Objetivos: El objetivo en esta etapa fue implementar la funcionalidad encargada de administrar los archivos de configuración del plug-in en general.

En el proceso de verificación de propiedades de un modelo en álgebra de tareas, las expresiones deben ejecutarse en un compilador (exe) que devolverá los resultados de la verificación. En este punto reafirmamos que el compilador como se mencionó en el capítulo 3 se considera un recurso externo. Los recursos externos en el proyecto se administran por medio de un archivo de configuración denominado "**LtlCtl.ini**".

El archivo LtlCtl.ini contiene las rutas de los compiladores (ejecutables) de expresiones, en este caso existen : LTL y CTL. La implementación de esta funcionalidad se realizó utilizando la clase dialogs GUI del paquete org.eclipse.jface e IFile del paquete org.eclipse.core.resources, java.io.File . Esta implementación al igual que las anteriores se basa en las restricciones de clase y componentes de la sección 4.4.

En la implementación de esta funcionalidad el comportamiento general establece que únicamente se crearía un archivo por entorno, y cuando el archivo ya existiera se referenciaría y actualizaría. La vista de esta configuración implementada se muestra en la figura 4.14.

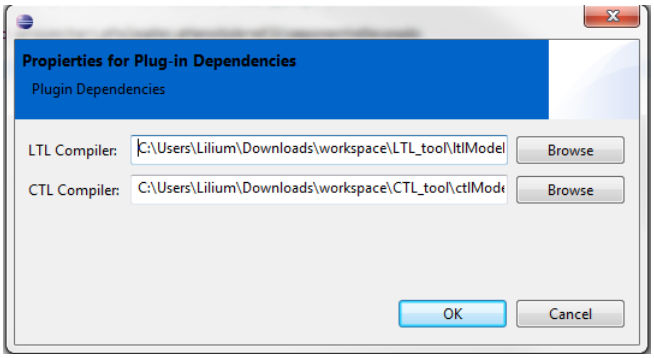

Figura 4.14: Implementación de la funcionalidad de administración de recursos externos

4.5.4. Implementación 4: Proceso de Ejecución de Expresiones de Consulta

Objetivos: El objetivo de esta etapa fue implementar una de las funcionalidades más importantes del sistema, la cual se encarga de ejecutar las expresiones de consulta especificadas en los archivos tfq.

Un proceso de ejecución típico consta de ciertas entradas que son procesadas por un programa y devuelve una salida en un formato específico. En el proyecto Task Flow Interface las entradas de ejecución son aquellas especificaciones en lógicas temporales construidas por el usuario, las

cuales deben ejecutarse en el recurso externo denominado compilador y mostrar los resultados en un contenedor del entorno Eclipse, en particular en la consola de resultados.

Recordemos que al igual que las etapas anteriores existen ciertas restricciones que deben considerarse en el momento de construir el sistema, en particular nos interesamos en las siguientes:

- La invocación de la funcionalidad solo puede realizarse desde un archivo de consulta libre de errores.
- La ejecución de las expresiones se realizará si el recurso modelo existe en el proyecto y además no contiene errores de sintaxis en la definición.
- Los recursos externos deben estar especificados en la configuración de recursos LtlCtl.ini.

Consideradas estas condiciones y con ayuda del caso de uso, la codificación se implementó bajo el siguiente flujo de eventos:

1. **Invocar** la funcionalidad.
2. **Verificar** que el origen de ejecución sea un archivo de consultas.
3. **Verificar** que exista el modelo y la librería de tareas.
4. **Verificar** que todos los componentes asociados estén libre de errores de sintaxis.
5. **Verificar** que los recursos a utilizarse estén correctamente definidos en el archivo de configuración.
6. **Construir** la instrucción de ejecución, **enlazar** los recursos externos y **ejecutar** las expresiones.
7. **Mostrar** los resultados en la consola de Eclipse.

En el punto 6 la construcción de la instrucción se refiere a definir la entrada del compilador (LTL y CTL) bajo el formato especificado en el capítulo 2 y 3, el formato es el siguiente:

>> *"algebra_del_modelo" "expresión_lógica"*

En términos de código las clases principales utilizadas en el proceso son: Process del paquete java.io para ejecución de archivos externos desde el entorno Eclipse e IConsole del paquete org.eclipse.ui.console para mostrar los resultados en la consola de Eclipse. La funcionalidad implementada en ejecución se muestra en la figura 4.15 teniendo como modelo de entrada **"{ { solicitarForma + Phi } ; irALibreros ; {revisarInformacion || seleccionarCandidatos } ; rellenarForma ; { solicitarLibros || entregarForma } ; { otorgarLibros || Phi } } "** y como expresión de consulta LTL **"not otorgarLibros"**:

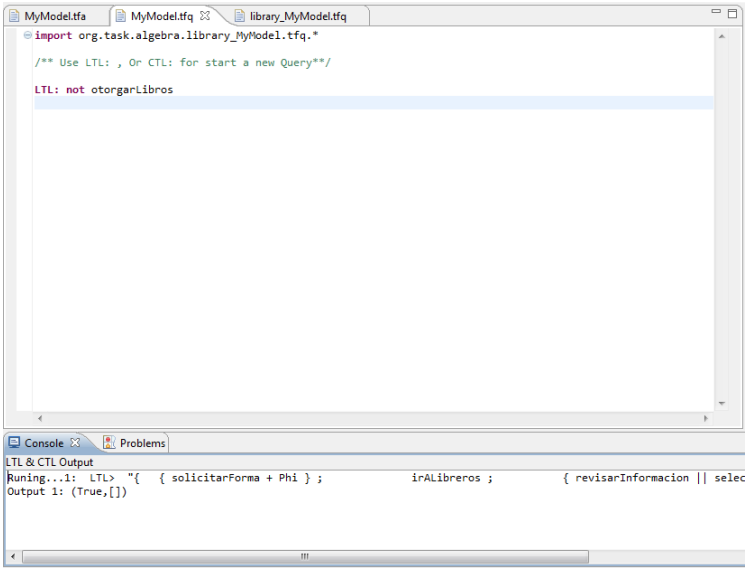

Figura 4.15: Integración de la funcionalidad ejecución de expresiones del plug-in en el entorno Eclipse.

4.5.5. Implementación 5: Representación de Resultados de Ejecución.

Objetivos: El objetivo en esta etapa de implementación fue representar los resultados de ejecución anteriores en un esquema de árbol de manera gráfica.

Los resultados de verificación de propiedades en un modelo del álgebra de tareas varían dependiendo de tres factores principales: el tipo de expresión (LTL/CTL), la complejidad del modelo y los operadores utilizados en la expresión.

En el caso más simple una expresión de tipo lineal (LTL) como resultado de ejecución devolverá verdadero o bien falso con el contra-ejemplo asociado. Analizar un resultado de este tipo es relativamente rápido y simple. Sin embargo cuando se utilizan expresiones CTL el análisis se vuelve complejo, debido que por definición la búsqueda se realiza sobre todos los caminos y los resultados pueden variar de una simple línea hasta múltiples páginas dependiendo de los flujos en el modelo y los operadores.

Para comprender un poco la dificultad de análisis de expresiones CTL, se analizó el ejemplo particular siguiente:

- **Modelo:** { { solicitarForma + Phi } ; irALibreros ; { revisarInformacion || seleccionarCandidatos } ; rellenarForma ; { solicitarLibros || entregarForma } ; { otorgarLibros || Phi } }
- **Expresión:** CTL: not otorgarLibros
- **Resultados de ejecución:**

([[0], [0,1], [0,1,1],[0,1,1,1], [0,1,1,1,1], [0,1,1,1,1,1], [0,1,1,1,1,1,1], [0,1,1,1,1,1,1,1],

[0,1,1,1,1,1,1,1,1,1], [0,1,1,1,1,1,1,2], [0,1,1,1,1,1,1,2,1], [0,1,1,1,1,1,1,2,1,1], [0,1,1,1,2], [0,1,1,2,1], [0,1,1,2,1,1], [0,1,1,2,1,1,1], [0,1,1,2,1,1,1,1], [0,1,1,2,1,1,1,1,1], [0,1,1,2,1,1,2], [0,1,1,2,1,1,2,1], [0,1,1,2,1,1,2,1,1], [0,2]],Node ([0],null) [Node ([0,2],Phi) [Empty],Node ([0,1], solicitarForma) [Node ([0,1,1],irALibreros) [Node ([0,1,1,2],seleccionarCandidatos) [Node ([0,1,1,2,1],revisarInformacion) [Node ([0,1,1,2,1,1],rellenarForma) [Node ([0,1,1,2,1,1,2],solicitarLibros) [Node ([0,1,1,2,1,1,2,1], entregarForma) [Node ([0,1,1,2,1,1,2,1,1],Phi) [Empty]]],Node ([0,1,1,2,1,1,1],entregarForma) [Node ([0,1,1,2,1,1,1,1],solicitarLibros) [Node ([0,1,1,2,1,1,1,1,1],Phi) [Empty]]]]]],Node ([0,1,1,1],revisarInformacion) [Node ([0,1,1,1,1],seleccionarCandidatos) [Node ([0,1,1,1,1,1],rellenarForma) [Node ([0,1,1,1,1,1,2],solicitarLibros) [Node ([0,1,1,1,1,1,2,1],entregarForma) [Node ([0,1,1,1,1,1,2,1,1],Phi) [Empty]]],Node ([0,1,1,1,1,1,1],entregarForma) [Node ([0,1,1,1,1,1,1,1] ,solicitarLibros) [Node ([0,1,1,1,1,1,1,1,1],Phi) [Empty]]]]]]]]]

Del resultado de ejecución anterior podemos darnos cuenta que para entenderlo el usuario necesita traducir mentalmente la forma textual a un modelo gráfico. Una ventaja de estos resultados textuales devueltos por el compilador es el hecho de utilizar la metáfora de árbol de jerarquías y es que además el principio de la verificación CTL es un flujo de caminos. Traducir un árbol de jerarquías a un modelo visual es un proceso simple que requiere un algoritmo particular.

Para obtener el algoritmo que se utilizaría en el proceso, se siguió el siguiente análisis:

1. Partiendo de los resultados de ejecución de una expresión CTL en particular

   ([[0], [0,1], [0,1,1],[0,1,1,1], [0,1,1,1,1], [0,1,1,1,1,1], [0,1,1,1,1,1,1], [0,1,1,1,1,1,1,1], [0,1,1,1,1,1,1,1,1], [0,1,1,1,1,1,2], [0,1,1,1,1,1,2,1], [0,1,1,1,1,1,2,1,1], [0,1,1,2], [0,1,1,2,1], [0,1,1,2,1,1], [0,1,1,2,1,1,1], [0,1,1,2,1,1,1,1], [0,1,1,2,1,1,1,1,1], [0,1,1,2,1,1,2], [0,1,1,2,1,1,2,1], [0,1,1,2,1,1,2,1,1], [0,2]],Node ([0],null) [Node ([0,2],Phi) [Empty],Node ([0,1], solicitarForma) [Node ([0,1,1],irALibreros) [Node ([0,1,1,2],seleccionarCandidatos) [Node ([0,1,1,2,1],revisarInformacion) [Node ([0,1,1,2,1,1],rellenarForma) [Node ([0,1,1,2,1,1,2],solicitarLibros) [Node ([0,1,1,2,1,1,2,1], entregarForma) [Node ([0,1,1,2,1,1,2,1,1],Phi) [Empty]]],Node ([0,1,1,2,1,1,1],entregarForma) [Node ([0,1,1,2,1,1,1,1],solicitarLibros) [Node ([0,1,1,2,1,1,1,1,1],Phi) [Empty]]]]]],Node ([0,1,1,1],revisarInformacion) [Node ([0,1,1,1,1],seleccionarCandidatos) [Node ([0,1,1,1,1,1],rellenarForma) [Node ([0,1,1,1,1,1,2],solicitarLibros) [Node ([0,1,1,1,1,1,2,1],entregarForma) [Node ([0,1,1,1,1,1,2,1,1],Phi) [Empty]]],Node ([0,1,1,1,1,1,1],entregarForma) [Node ([0,1,1,1,1,1,1,1] ,solicitarLibros) [Node ([0,1,1,1,1,1,1,1,1],Phi) [Empty]]]]]]]]]

2. Podemos notar que los resultados se componen de dos partes:

   a) La primera parte (negritas) expresa los caminos mediante una estructura jerárquica.

   b) La segunda parte expresa información particular de cada nodo: identificador, nombre del nodo.

3. Basado en este análisis un algoritmo para traducir los resultados de ejecución en un modelo visual es el siguiente:

a) Separar la parte del etiquetado de la parte de caminos en la expresión.

b) Inicializar una estructura de tipo árbol n-ario de la clase JTree (A) en java.

c) Para cada componente (y) en el flujo de caminos:

1) Obtener su identificador del flujo y su etiqueta descriptiva (nombre del nodo).
2) Obtener el padre (x) del nodo y en caso de existir.
3) Añadirlo el nodo (x) al árbol con la relación de parentesco definida (x padre de y). Donde previamente se ha añadido x.

d) Al finalizar de construir el árbol, iterar el componente por niveles, dibujar el nivel mediante los componentes drawOval , drawLine y drawText de la clase GC del paquete org.eclipse.swt.graphics.

Ademas de la lógica de implementación anterior se requirió un análisis para la parte dibujado del árbol en el contenedor (vista), la cual se resume a continuación:

Si observamos cuidadosamente los resultados devueltos por la ejecución podemos notar dos características importantes: La primera es que el árbol tiene un número máximo de niveles y un número máximo de nodos de nivel en general. En el momento de dibujar los nodos, el algoritmo para dibujar en pantalla responde al siguiente procedimiento:

1. El ancho del área de dibujo se define como la amplitud máxima de nivel (máximo número de nodos de nivel en el árbol, **Amplitud**) multiplicada por un ancho de nodo individual (**wi**) . A esto se le suma un espacio entre nodos (**vspace**), más la amplitud máxima de nivel multiplicada por el espacio individual. En decir:

   a) *Width= (Amplitud*wi)+ (vspace* Amplitud)+vspace;*

2. El alto del área de dibujo se define como la profundidad máxima del árbol (número de niveles, **Profundidad**) multiplicada por un alto de nodo individual (**hi**). A esto se le suma un espacio entre niveles (**hspace)**, más la profundidad multiplicada por el espacio individual entre niveles. Es decir:

   a) *Height= (Profundidad*hi)+ (hspace*Profundidad)+hspace;*

3. En el proceso de iteración en cada nivel del árbol, las consideraciones para dibujar son las siguientes:

   a) El nodo inicial (root) se coloca proporcionalmente en la mitad del ancho del contenedor.

   b) Para el resto del árbol, el algoritmo de dibujo por niveles es el siguiente:

   1) Hallar el número de nodos a pintar.
   2) Hallar la diferencia de nodos respecto al número máximo (**Dif**).
   3) Para garantizar un árbol simétrico con respecto a las dimensiones de la vista, la posición inicial *x* será igual al punto medio de la distancia horizontal menos el balance de la mitad de nodos totales en el caso de que el número total de nodos sea impar. En caso contrario el valor en *x* será igual a un espacio inicial vspace

mas el balance de la diferencia de nodos por su espacio horizontal y ancho de nodo, matemáticamente se traduce de la siguiente manera:

- *Xi= Xinitial- ((Totali/2) *wi) - ((Totali/2) *vspace);*
- De otra forma: *Xi=vspace+ ((Dif/2)*wi)+ (vspace* (Dif/2));*

4) Con respecto a la posición inicial *y*, esta será igual a un espacio inicial entre niveles (**hspace**) más el número de nivel del árbol (**NivelPaint**) multiplicado por un alto de nodo definido global (**hi**). A esto se le suma el espacio entre niveles multiplicado por el número de nivel actual, es decir:

- *Yi=hspace+ (NivelPaint*hi)+ (hspace*NivelPaint);*

5) Con las posiciones iniciales, el primer nodo de nivel tendrá las coordenadas iniciales x,y y para los consecuentes el incremento en x se define como: el ancho de nodo individual (**wi**) más un espacio entre nodos (**vspace**), es decir:

- *Xi+=wi+vspace;*

Con ayuda de las especificaciones anteriores se construyó la funcionalidad de representación de resultados en el entorno. Una consideración importante es que para cada expresión a verificar existe un apartado en la vista de resultados. La funcionalidad en ejecución se muestra en la figura4.16 teniendo como modelo de entrada **"{ { solicitarForma + Phi } ; irALibreros ; { revisarInformacion || seleccionarCandidatos } ; rellenarForma ; { solicitarLibros || entregarForma } ; { otorgarLibros || Phi } }"** y como expresión de consulta CTL **"not otorgarLibros"**:

4.5.6. Implementación 6: Validación de excepciones e integración de los mensajes orientados al usuario

Objetivos: El objetivo en esta etapa de implementación consistió en realizar una validación general de las excepciones que podrían generarse en el sistema y presentar al usuario una retroalimentación para poder solucionarlas.

A lo largo de las diferentes etapas de implementación nos hemos dado cuenta que el proceso de verificación de propiedades requiere que se cumplan ciertas restricciones y en cada fase del proceso el sistema pasa por diversos estados. Cuando las restricciones no se cumplen, es necesario informar al usuario; el elemento seleccionado para realizar esta acción fue el cuadro de dialogo de la clase dialog. A continuación se enlistan una serie de excepciones que pueden generarse en el sistema y el mensaje que se presenta al usuario:

Tarea: Nuevo consultor
Excepciones:

- La funcionalidad se invoca con errores de sintaxis en el modelo
  - **Mensajes:**"Unable create LTL,CTL query ", "Sintax Errors in Task Algebra File"

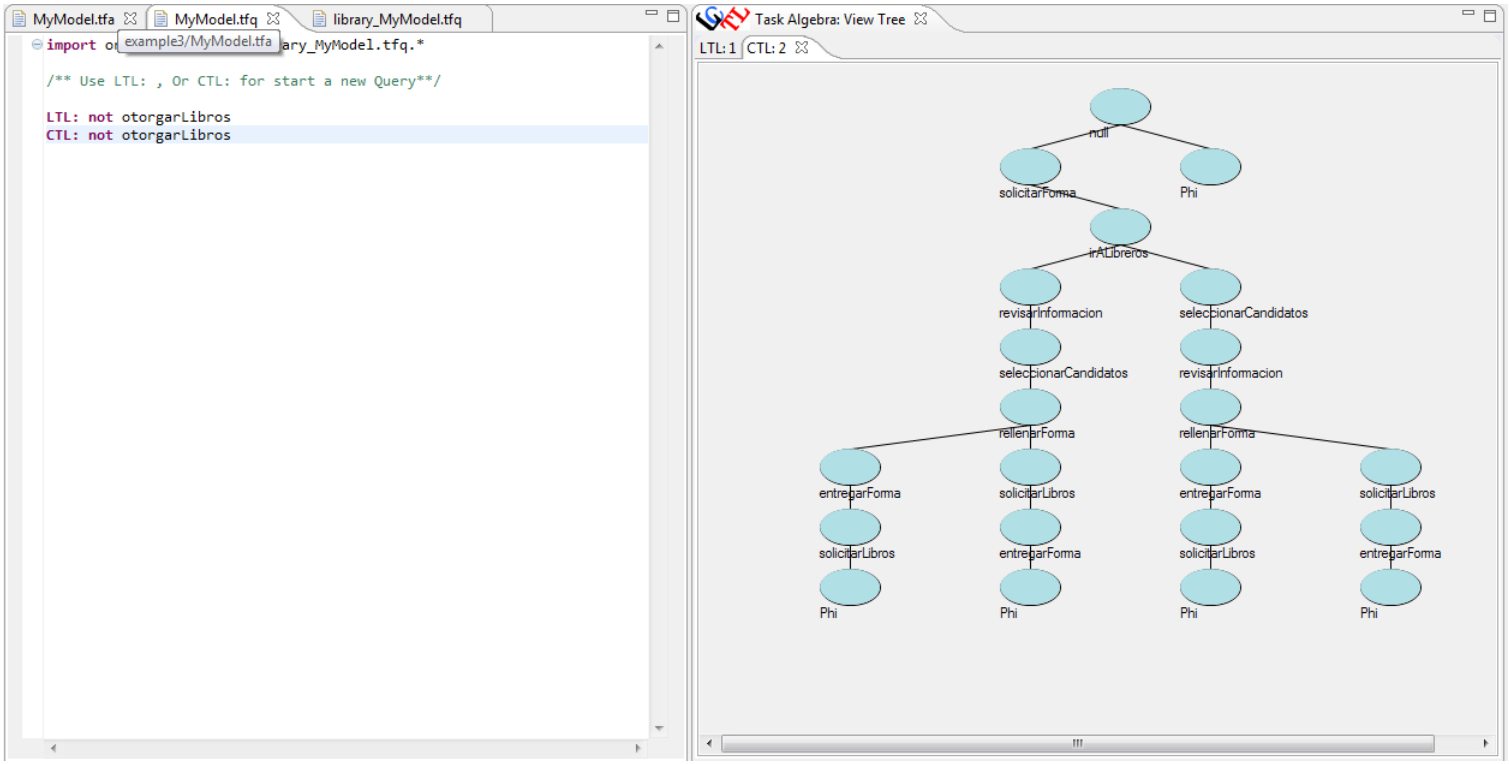

Figura 4.16: Integración de la funcionalidad de representación de resultados de ejecución del plug-in en el entorno Eclipse.

Tarea: Ejecución

Excepciones:

- La funcionalidad se invoca con errores de sintaxis en el modelo/archivo librería/consultor
  - **Mensajes:** "Unable to Run the query ", "Sintax Errors in : "+Errores
- Los recursos a utilizarse no se han definido en el archivo de configuración LtlCtl.ini
  - **Mensajes:** "Unable to Run the query", "LTL,CTL path compilers are undefined"
- El modelo de tareas/archivo biblioteca no existe en el proyecto
  - **Mensajes:**"Unable to Run the query: Review compilers and files requiered", "The following files are requiered:"+Errores
- Excepción en tiempo de ejecución de las expresiones.
  - **Mensajes:** "Fatal Error", "Unabled to RUN the query with defined compiler", e.getMessage ()
- Los recursos externos no pueden ejecutarse en el sistema operativo actual.
  - **Consola:** "Compilers can not run in the current OS: "+e.getMessage ()

Cada uno de estos mensajes se codificó e integró en el ámbito, la funcionalidad en ejecución se muestra en la figura 4.17.

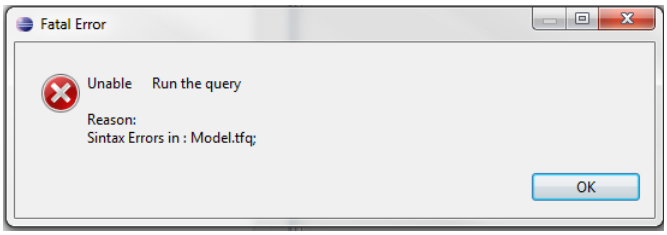

Figura 4.17: Integración del control de excepciones y mensajes de usuario del plug-in en Eclipse.

4.5.7. Implementación 7: Interfaz de Usuario y Liberación

Objetivos: El objetivo en esta etapa final de implementación fue integrar las funcionalidades del sistema en el entorno Eclipse, empaquetar el plug-in y liberar el sistema.

Los proyectos plug-in Eclipse presentan una gran ventaja sobre cualquier proyecto convencional y es que pueden utilizar los elementos ya desarrollados del entorno para realizar sus funcionalidades, es decir por ejemplo: Vistas, Barras de herramientas, Asistentes, Áreas del entorno, Consola de Resultados, Perspectivas, Editores, Errores, Cuadros de dialogo y otros. Utilizar los componentes ya existentes presenta una gran ventaja y es que no se requiere volver a diseñarlos.

De manera particular en esta implementación se utilizaron las barras de herramientas (toolbar), iconos (command) y perspectivas (perspective), los cuales se encuentran disponibles como

puntos de extensión del paquete org.eclipse.ui. La lista de funcionalidades que se integraron en el entorno son las siguientes:

- **New Logic Query (command):** Permite al usuario crear un archivo de consulta basado en un modelo específico del álgebra de tareas.
- **Add Task (command):** Permite al usuario añadir una tarea del modelo al archivo de consulta en la posición del cursor.
- **Group (command):** Permite al usuario agrupar por medio de los símbolos ' (' , ')' un conjunto de expresiones de la selección actual.
- **True (command):** Permite al usuario añadir la condición lógica 'true'.
- **False (command):** Permite al usuario añadir la condición lógica 'false'.
- **LTL Expressions (commmand, group):** Categoría general de las opciones LTL.
    - *Logic Operator:* Not. Muestra las diversas combinaciones del operador *Not* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Logic Operator: And.* Muestra las diversas combinaciones del operador *And* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Logic Operator: Or.* Muestra las diversas combinaciones del operador *Or* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Logic Operator: Implication.* Muestra las diversas combinaciones del operador *Implication* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Unary Operator: Next.* Muestra las diversas combinaciones del operador *Next* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Unary Operator: Globally.* Muestra las diversas combinaciones del operador *Globally* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Unary Operator: Finally.* Muestra las diversas combinaciones del operador *Finally* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Binary Operator: Until.* Muestra las diversas combinaciones del operador *Until* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Binary Operator: Weak-until.* Muestra las diversas combinaciones del operador *Weak-Until* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Binary Operator: Release.* Muestra las diversas combinaciones del operador *Release* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
- **CTL Expressions (command, group):** Categoría general de las opciones CTL.
    - *Logic Operator: Not.* Muestra las diversas combinaciones del operador *Not* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Logic Operator: And.* Muestra las diversas combinaciones del operador *And* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
    - *Logic Operator: Or.* Muestra las diversas combinaciones del operador *Or* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.

- *Logic Operator: Implication.* Muestra las diversas combinaciones del operador *Implication* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
- *All-Unary Operator: ANext.* Muestra las diversas combinaciones del operador *ANext* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
- *All-Unary Operator: AFinally.* Muestra las diversas combinaciones del operador *AFinally* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
- *All-Unary Operator: AGlobally.* Muestra las diversas combinaciones del operador *AGlobally* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
- *All-Unary Operator: AUntil.* Muestra las diversas combinaciones del operador *AUntil* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
- *Exists-Unary Operator: ENext.* Muestra las diversas combinaciones del operador *ENext* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
- *Exists-Unary Operator: EFinally.* Release. Muestra las diversas combinaciones del operador *EFinally* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
- *Exists-Unary Operator: EGlobally.* Muestra las diversas combinaciones del operador *EGlobally* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.
- *Exists-Binary Operator: EUntil.* Muestra las diversas combinaciones del operador *EUntil* aplicadas al modelo de tareas, permitiendo al usuario seleccionar alguna.

- **Config Build Settings (command):** Permite al usuario configurar los recursos de ejecución externos (LTL, CTL)

Para realizar la integración de las funcionalidades en el sistema primero fue necesario prototipar el aspecto de la integración, el cual se muestra en la figura 4.18.

En la figura anterior podemos apreciar que las funcionalidades se colocarán en el área de herramientas denominada **TaskFlowInterface** en conjunto con los otros elementos del sistema. Continuando con el proceso implementación, se definió una perspectiva que agrupara todos los componentes utilizados en el proyecto bajo el nombre de (LC)TL Perspective; la cual incluye el explorador de proyectos, consola de resultados, vista de errores y vista de resultados de ejecución.

En esta implementación al igual que en las anteriores se codificaron las restricciones de clases y componentes que especifican que las barras de herramientas solo estarán disponibles en el ámbito del proyecto, es decir, mientras un archivo de tareas o consulta este abierto, la barra de herramientas también lo estará. Al final del proceso de codificación el aspecto de integración del sistema Task Flow Interface es el mostrado en la figura 4.19.

Al finalizar la implementación de las funcionalidades del proyecto, el proceso de compilado y empaquetado del plug-in se realizó creando los proyectos de características (Feature Project) y de actualización de sitio (Update Site Project); referenciando adecuadamente el plug-in y compilando los archivos individuales. Una vez finalizado el paso anterior el plug-in se alojó en el sitio remoto del proyecto:

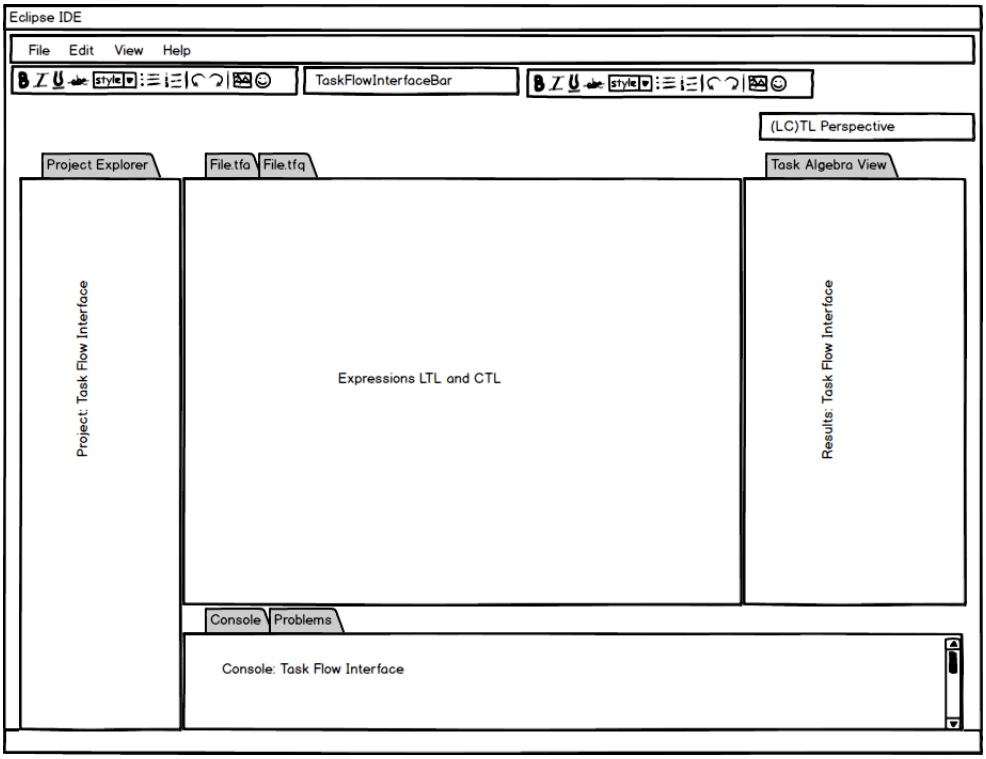

Figura 4.18: Prototipo de integración de funcionalidades en el entorno Eclipse.

http://www.utm.mx/ ~taskmodel/ projects/ modelCheckingEclipse/
org.plug.taskalgebraproject.all_update /

Los compiladores LTL y CTL en la siguiente dirección:

http://www.utm.mx/ ~taskmodel/ projects/ modelCheckingEclipse/ Compiladores/ .

Con esta etapa de implementación finaliza la fase del proyecto, en el siguiente capítulo se muestra la evaluación del proyecto con usuarios reales.

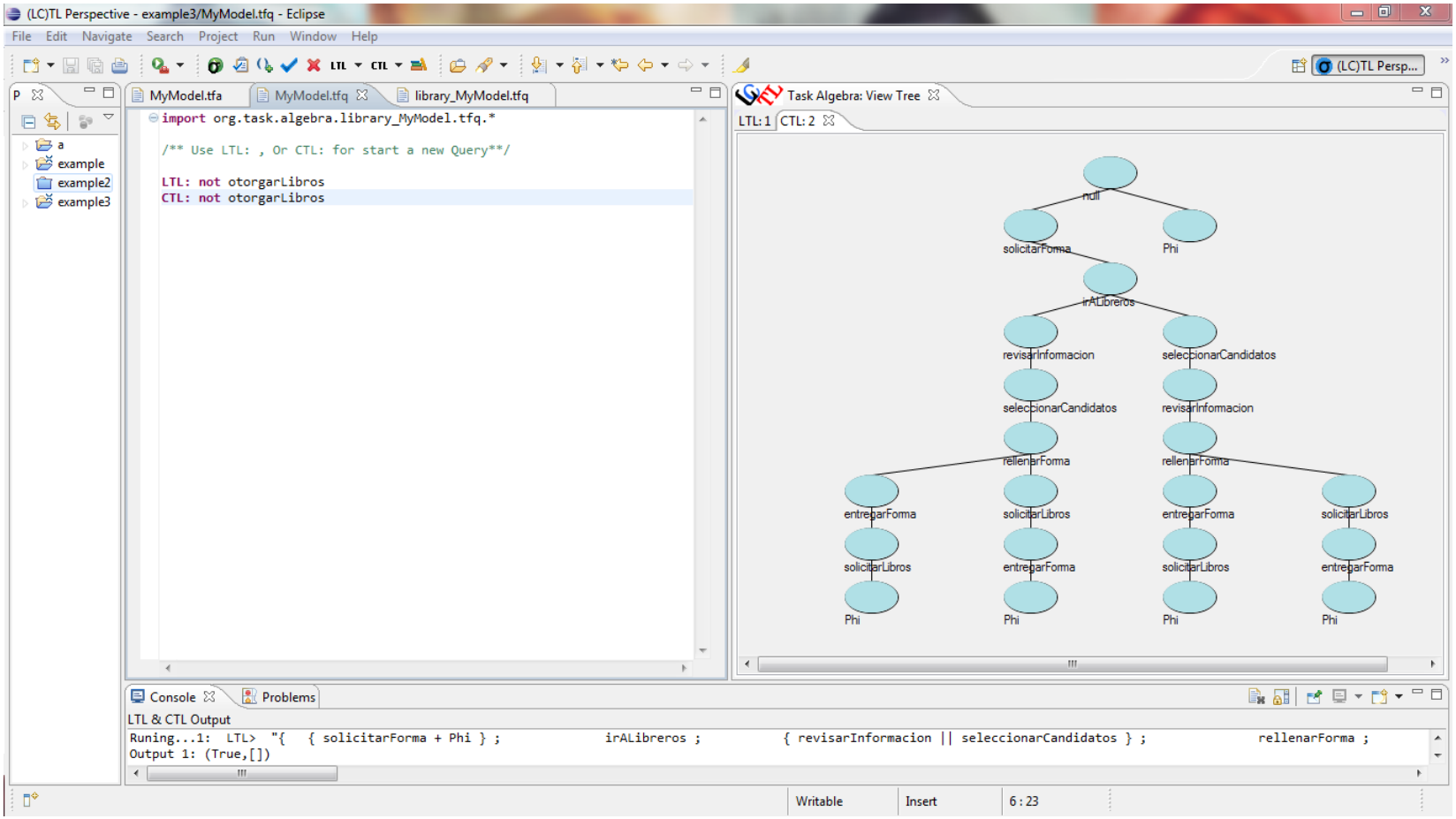

Figura 4.19: Integración de los componentes del proyecto en el entorno Eclipse

Capítulo 5

Evaluación

5.1. Introducción

Las pruebas en el Proceso Unificado consisten en verificar que los requerimientos del cliente/usuario se cumplan y funcionen de acuerdo a las especificaciones. En las pruebas los componentes y/o subsistemas deben evaluarse en términos de funcionalidad e integración [16].

Nuestro objetivo en este capítulo es combinar el enfoque de pruebas del Proceso Unificado con el enfoque de evaluación de usabilidad; es decir evaluar el sistema desde dos perspectivas: Sistema y Usuario. A continuación se detalla el desarrollo de cada una de las evaluaciones.

5.2. Pruebas: Enfoque de Sistema

Las pruebas en el Proceso Unificado tienen como objetivo detectar defectos y errores en software para mejorarlo antes de que sea liberado a terceros. Para lograr estos objetivos la especificación cuenta con diversos artefactos como son: modelo de pruebas, casos de prueba, procedimientos de pruebas y componentes de prueba. Particularmente en el proyecto actual el artefacto seleccionado fue el modelo de pruebas debido a que define *el que y el como* a través de sus casos de pruebas, procedimientos y componentes (ver figura 5.1).

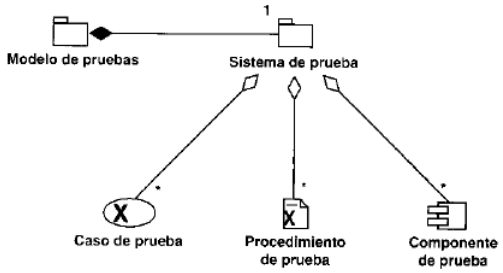

Figura 5.1: Modelo de pruebas del Proceso Unificado

5.2.1. Artefacto: Modelo de Pruebas

El modelo de pruebas consiste en la especificación de los casos de prueba, procedimientos individuales y componentes. Considerando las diversas etapas de construcción del proyecto Task Flow Interface, los caso de prueba que se seleccionaron son los subsistemas liberados en cada una de ellas:

1. **Caso 1:** Importar modelo de tareas
2. **Caso 2:** Creación de consultor
3. **Caso 3:** Construir expresiones
4. **Caso 4:** Configurar recursos externos
5. **Caso 5:** Ejecución de consultas.
6. **Caso 6:** Visualización de resultados.

5.2.2. Artefacto: Casos y Procedimientos de Prueba

Los casos de prueba mencionados anteriormente además de corresponder a los principales subsistemas liberados en las etapas de construcción corresponden al esquema general del proceso de verificación de especificaciones de los diagramas de tareas, es decir, toda etapa de verificación de especificaciones está guiada por los siguientes pasos:

1. Construir el modelo
2. Definir expresiones de consulta
3. Ejecutar expresiones
4. Visualizar y analizar resultados

Una vez establecida esta similitud es necesario establecer un escenario que contextualice el proceso en la vida real. Por ejemplo analicemos la actividad de reservación de libros en una biblioteca a través de un sistema.

Reservar libros a domicilio a través de un sistema en un caso de uso particular requiere de la realización de las siguientes tareas individuales:

1. Consultar libros en el sistema
2. Añadir libros a reservaciones
3. Acceder con una cuenta de usuario
4. Confirmar Reservación
5. Finalizar reservación.

Imaginemos ahora que el sistema desarrollado responde al flujo anterior, sin embargo no se tiene la certeza de que se cumplan ciertos estados en ejecución. Para ello los desarrolladores de software desean analizar los flujos, establecer un modelo y verificar especificaciones. Los actores

involucrados han modelado el diagrama de tareas de la actividad y su álgebra de tareas asociada, las cuales se definen a continuación (Ver figura 5.2 y figura 5.3):

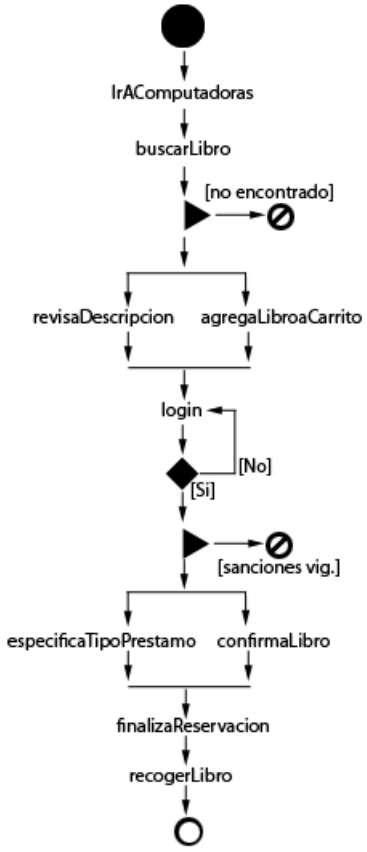

Figura 5.2: Diagrama de tareas de la actividad Reservar Libros

```
{
   irAComputadoras; buscarLibro; (
       Phi + (
                (revisaDescripcion || agregaLibroaCarrito) ; Mu.x (login; Epsilon+x); (
               Phi + (
                (especificaTipoPrestamo || confirmaLibro); finalizaReservacion; recogerLibro
              )
             )
             )
    )
}
```

Figura 5.3: Especificación algebraica de la actividad Reservar Libros

Una vez que los actores involucrados en el proceso han definido el modelo, el siguiente paso consiste en verificar las propiedades/tareas/actividades. Para ello se utiliza la herramienta TFI, en la cual la verificación está determinada por los siguientes casos y procedimientos de prueba.

5.2.2.1. Caso de Prueba: Importar Modelo de tareas

En este caso de prueba se realiza la actividad de importar la especificación en álgebra de tareas (modelo) construida por los actores en el escenario inicial.

Entradas:

- Los desarrolladores de software han modelado el sistema de reservación de libros a domicilio mediante un diagrama de tareas y el álgebra asociada.
- Existe un archivo que contiene la especificación en álgebra de tareas.
- Existe un proyecto sobre el cual se desea importar el modelo.

Resultados:

- Se crea una copia del archivo de entrada con el mismo nombre y extensión tfa.
- El archivo tfa se añade al proyecto especificado por el usuario.
- Se asocia al proyecto una perspectiva Xtext Nature.

Condiciones:

- El contenido del archivo de entrada debe estar basado en la gramática del álgebra de tareas del Método Discovery.
- El sistema TFI se ha instalado adecuadamente en el entorno.

Procedimiento de prueba:

1. Con el proyecto candidato abierto haga clic en la opción Import del Menú File; o bien de clic derecho en el proyecto y a continuación seleccione Import
2. De la lista de categorías Import haga clic en Model Checking y a continuación seleccione Import Task Agebra File
3. En la siguiente ventana seleccione el archivo que contiene el modelo que desea importar y a continuación especifique el nombre que tendrá el modelo en el proyecto, puede omitir este paso si desea tomar la sugerencia del asistente.
4. Para finalizar haga clic en Finish.

El segundo paso para continuar con la verificación consiste en construir el archivo consultor en el cual se definirán las especificaciones.

5.2.2.2. Caso de Prueba: Creación de consultor

Este caso de prueba consiste en la creación del archivo consultor (tfq), en el cual se definen las expresiones Ltl y Ctl.

Entradas:

- Existe un modelo de tareas (tfa) disponible en el proyecto.

Resultados:

- Se crea un archivo con el mismo nombre del modelo pero con extensión tfq.
- Se crea un archivo librería tfq que enlaza el modelo con el consultor por medio de las instrucción package e Import.

Condiciones:

- El modelo de tareas debe estar libre de errores de sintaxis.

Procedimiento de prueba:

1. Con el modelo de tareas abierto, haga clic en la opción New Logic Query de la barra de herramientas estándar.
2. El archivo de consulta se crea satisfactoriamente si las condiciones del caso se cumplen satisfactoriamente.

Continuando con el proceso, el siguiente paso que deben realizar los desarrollados es la construcción de expresiones lógicas aplicadas al modelo inicial, este procedimiento se realiza utilizando el siguiente caso de prueba.

5.2.2.3. Caso de Prueba: Construir Expresiones

Este caso de prueba consiste en definir diversas expresiones en lógica lineal temporal y computacional en árbol basadas en un modelo algebraico especifico. El proceso de construcción se realiza utilizando asistentes guía.

Entradas:

- Existe un modelo de tareas (tfa) disponible en el proyecto.
- Existe una librería de tareas (tifo) disponible en el proyecto.
- Existe un consultor (tfq) disponible en el proyecto.

Resultados:

- Se definen expresiones lógicas en el consultor tfq relacionadas con el modelo de tareas requisito.
- Se verifica sintácticamente si las expresiones definidas presentan errores de sintaxis.
- Se crea un archivo con el mismo nombre del modelo pero con extensión tfq.
- Se crea un archivo librería tfq que enlaza el modelo con el consultor por medio de las instrucciones package e Import.

Condiciones:

- Las expresiones Ltl se inician con la especificación: “LTL:” o bien Ctl con “CTL: ”

Procedimiento de prueba:

1. Con el consultor abierto en una nueva línea inicie la invocación de las expresiones con “LTL:”/ “CTL:”, dependiendo del tipo de expresión que desea definir.
2. Para continuar su expresión haga clic en cualquiera de las opciones: LTL Expressions, CTL Expressions, Add Task , Group, True o False; localizadas en la barra de herramientas estándar.

3. Verifique que las expresiones construidas se encuentren libres de errores de sintaxis.

Una vez que las expresiones han sido construidas por los desarrolladores el siguiente paso es la configuración de los recursos externos. El caso de prueba siguiente describe el escenario.

5.2.2.4. Caso de Prueba: Configurar Recursos Externos

Este caso de prueba consiste en configurar y definir la ubicación de los compiladores Ltl y Ctl utilizados en los la verificación de expresiones de los diagramas de tareas.

Entradas:

- Existe un consultor (tfq) disponible en el proyecto.
- Existen los compiladores Ltl y Ctl

Resultados:

- Se crea un archivo de configuración en el espacio de trabajo del entorno Eclipse con el nombre: LtlCtl.ini

Condiciones:

- El usuario es responsable de la integridad, funcionamiento de sus archivos especificados como compiladores de los proyectos.

Procedimiento de prueba:

1. Con el consultor abierto, haga clic en la opción *Config Build Settings* de la barra de herramientas estándar.
2. En el cuadro de dialogo configurar recursos examine la ubicación hasta encontrar los archivos ejecutables LTL / CTL.
3. Cuando termine de especificar los recursos haga clic en la opción Guardar.

El paso final del proceso de verificación de expresiones consiste en la ejecución, para ello los desarrolladores ejecutan sus expresiones utilizando el sistema TFI.

5.2.2.5. Caso de Prueba: Ejecución de Consultas

Este caso de prueba consiste en la ejecución de las expresiones definidas en un archivo consultor.

Entradas:

- Existe un consultor (tfq) disponible en el proyecto en el cual se encuentran diversas expresiones de consulta Ltl y Ctl.
- Existe el archivo de configuración LtlCtl.ini.

Resultados:

- Las expresiones se ejecutan a través de los compiladores y los resultados se muestran en la consola del entorno.

Condiciones:

- El archivo consultor no presenta errores de sintaxis.

- El archivo de configuración ini presenta una definición explicita de los compiladores recurso a utilizarse en la ejecución.
- Los compiladores existen y son funcionales en el sistema operativo actual.

Procedimiento de prueba:

1. Con el consultor abierto, haga clic opción Run Logic Expression de la barra de herramientas estándar.
2. Se mostrarán a continuación los resultados de ejecución en la consola del entorno. Si la consola no se encuentra visible, esta se activara automáticamente.

Aún con este caso de prueba el proceso de análisis de resultados requiere un caso de prueba mas denominado visualización de resultados.

5.2.2.6. Caso de Prueba: Visualización de Resultados

Este caso de prueba corresponde a la segunda parte de la ejecución de las expresiones. La acción que debe realizarse consiste en representar los resultados de ejecución de forma gráfica utilizando el esquema de árbol.

Entradas:

- Existe un consultor (tfq) disponible en el proyecto en el cual se encuentran diversas expresiones de consulta Ltl y Ctl.
- Existe el archivo de configuración LtlCtl.ini.

Resultados:

- Las expresiones se ejecutan a través de los compiladores y los resultados se muestran en la consola del entorno.
- Los resultados de ejecución individuales se muestran de manera gráfica utilizando el esquema de árbol.

Condiciones:

- El archivo consultor no presenta errores de sintaxis.
- El archivo de configuración ini presenta una definición explicita de los compilares recurso a utilizarse en la ejecución.
- Los compiladores existen y son funcionales en el sistema operativo actual.
- Las expresiones lógicas son no nulas.

Procedimiento de prueba:

1. Con el consultor abierto, haga clic opción Run Logic Expression de la barra de herramientas estándar.
2. Se mostrarán a continuación los resultados de ejecución en la consola del entorno. Si la consola no se encuentra visible, esta se activará automáticamente.
3. Se mostrarán también los resultados de ejecución gráficamente en la vista Task Algebra: View Tree .

Cuando se finalice el caso de prueba, el escenario de verificación de flujos/tareas/actividades del proceso de reservación de libros en el sistema realizado por los actores quedará concluido. A

continuación, tomando las especificaciones y el escenario simulado, la realización de pruebas del sistema TFI se realizó y mostró los resultados especificados en la siguiente sección.

5.2.3. Ejecución y Planeación de las Pruebas

Antes de realizar las pruebas definidas anteriormente fue necesario reunir y cumplir los siguientes requerimientos iniciales:

1. Sistema TFI codificado e integrado, finalizado y liberado.
2. Sistema TFI instalado en el entorno eclipse como versión de prueba.
3. 1 Desarrollador de software para realizar las pruebas relacionado con el proyecto de diagramas de tareas y modelos.
4. Conjunto de métricas o indicadores a medir.

Con respecto a los indicadores o métricas, los criterios a medirse en cada uno de los casos de prueba son: Funciones incorrectas o ausentes (funcionalidad), validación de procesos, errores en interfaz e integración entre subsistemas. Una descripción breve de estos criterios es la siguiente:

Funcionalidad: La funcionalidad del sistema es correcta siempre y cuando se obtengan las salidas especificadas en el caso de prueba.

Validación-Procesos: El sistema es válido en relación a los procesos si cumple el procedimiento de prueba y se integra adecuadamente con otros casos de pruebas.

Errores-Interfaz: El sistema cumple con no defectos en interfaz si la guía/tip que da al usuario corresponde con la acción que realiza, además si las acciones definidas en el icono se llevan a cabo.

Integración: El sistema cumple esta propiedad si las salidas de un caso de prueba se integran con otro como parte del proceso de verificación en las condiciones que lo requiera.

Con ayuda de las especificaciones anteriores, los casos de prueba se llevaron a cabo por parte del mismo desarrollador del proyecto. Los resultados se reflejan en la tabla 5.1.

Caso de prueba	Funcionalidad	Procesos (caso de prueba)	Interfaz (guía)	Integración
Importar modelo de tareas	0	0	0	0
Creacion de consultor	0	0	0	0
Construir expresiones	0	0	0	0
Configurar recursos externos	0	0	0	0
Ejecución de consultas	2	0	0	0
Visualización de resultados	0	0	1	0

Cuadro 5.1: Tabla de resultados de pruebas en el enfoque sistema

De la tabla 5.1 podemos visualizar la presencia de 2 errores de funcionalidad y 1 error de interfaz. Estos errores son los siguientes:

1. Funcionalidad:

   a) El error se origina cuando la expresión de consulta no se define en la misma línea

donde se comenzó, cuando esto sucede el sistema no realiza la ejecución de la expresión y devuelve un error.

b) El error se origina cuando el usuario incluye comentarios de código (/**/ y //) en el archivo de expresiones, esto generó una excepción en ejecución y el caso de prueba no se concluye.

2. Interfaz

   a) El algoritmo de dibujo no es el adecuado debido a que no balancea simétricamente el árbol de resultados, lo que dificulta el análisis de estos.

Con los resultados obtenidos de estas pruebas el sistema se corrige con una nueva etapa de construcción, permitiendo liberar una segunda versión del proyecto estable.

5.3. Pruebas: Enfoque de Usuario

En el capítulo 2 se especificó la importancia de diseñar y construir funcionalidades basadas en las necesidades reales de los usuarios, estas necesidades incluyen además de funcionalidad el entendimiento del modelo mental del usuario.

Las pruebas con este enfoque evalúan si los sistemas desarrollados realizan las actividades en la misma forma que esperaría el usuario, también si el usuario entiende la iconografía y los flujos de procesos que propone el sistema. Para realizar este tipo de pruebas es necesario definir básicamente 3 aspectos: número de usuario y perfil, escenario de uso y tareas a realizar. Comúnmente estos criterios y otros son definidos en un documento denominado protocolo de experimentación.

El protocolo de experimentación forma parte del anexo A de este proyecto, el cual se basa en la técnica de observación en un laboratorio de usabilidad descrita en el capítulo 2. Los principales factores y detalles se resumen en el siguiente apartado:

5.3.1. Protocolo de Experimentación

El protocolo de experimentación es un guión de especificaciones, criterios y actividades definidas por el constructor del sistema en el cual se especifican los factores claves a observar en un usuario real del sistema. La especificación de estos fue la siguiente:

5.3.1.1. Métricas de la evaluación:

En el proyecto TFI se desean obtener, observar y evaluar 3 aspectos importantes del sistema: efectividad, eficiencia y satisfacción. Una breve definición de la forma de medir estos aspectos es la siguiente [30]:

- **Efectividad:** Se determina de acuerdo al porcentaje total de tareas completadas, porcentaje de tareas realizadas en el primer intento, porcentaje de éxitos y fracasos y el número de veces que los usuarios solicitan ayuda.

- **Eficiencia:** Se mide en base al número de errores cometidos y la severidad de los problemas presentados.
- **Satisfacción:** Es una relación entre los adjetivos positivos y negativos expresados por los usuarios.

5.3.1.2. Requerimientos

Para el desarrollo de las pruebas se requirieron 3 usuarios, 2 observadores y 1 facilitador. Los perfiles definidos para cada uno fueron los siguientes:

- **Usuarios**
  - Desarrolladores de software o estudiantes relacionados con la ingeniería de software.
  - Utilizar/Haber utilizado el entorno Eclipse IDE para el desarrollo de software.
  - Tener conocimientos del Método Discovery, Modelos, Expresiones Lógicas y Álgebra de tareas (sesión de capacitación).
- **Observadores:** Los observadores deben tener habilidades para analizar comportamiento de usuario y tener conocimiento de ingeniería de software.
- **Facilitadores:** Este perfil lo debe llenar personas con habilidades para relacionarse con los usuarios, tener conocimiento del Método Discovery, Modelos, Expresiones Lógicas y Álgebra de tareas.

5.3.1.3. Reclutamiento de Usuarios.

Los usuarios fueron seleccionados por sus características del área de postgrado de la maestría en cómputo aplicado. Los observadores fueron profesores investigadores con conocimientos del álgebra de tareas y bases de usabilidad.

5.3.1.4. Características del Entorno

El entorno no se consideró como factor decisivo en esta evaluación, por lo que se seleccionó un ambiente controlado: Laboratorio de Usabilidad de la Universidad Tecnológica de la Mixteca.

5.3.1.5. Lista de tareas evaluadas:

1. Acceder al entorno de desarrollo Eclipse (No evaluada).
2. Crear un nuevo proyecto General (No evaluada).
3. Activar la perspectiva de trabajo LT (CL) Perspective.
4. Importar el modelo de tareas: Model.tfa, localizado en la Carpeta Proyectos del Escritorio.
5. Estructurar alguna de las siguientes consultas:
   a) CTL, Operador Exist Finally, actividad: recogerLibro
   b) LTL, Operador Not, actividad: buscarLibro

6. Configurar los compiladores LTL y CTL del proyecto, localizados en la carpeta Proyectos del Escritorio.
7. Ejecutar las consultas y describir brevemente los resultados.

5.3.1.6. Técnica de interacción con el usuario

La interacción con el usuario se realizó por medio de Think Aloud. La técnica de Think Aloud[21] consistió en indicar al usuario que explicará en voz alta las acciones y decisiones que estaba realizando a lo largo de las pruebas, de esta manera se puede obtener mayor alimentación y entender los modelos mentales. También se utilizó un cuestionario breve para conocer el entendimiento de la iconografía, grado de satisfacción con el sistema y nuevas mejoras (ver anexo B).

5.3.2. Resultados de Evaluación

Al realizar la evaluación en el entorno especificado, los usuarios realizaron las tareas predefinidas; mientras los observadores analizaban el comportamiento y la realización de las pruebas. En este caso los observadores se mantuvieron alejados sin influir en las decisiones del usuario (ver figura 5.4). Los resultados que a continuación se presentan incluyen los puntos de vista de observadores y facilitador (los resultados detallados se muestran en el anexo C de este documento).

Figura 5.4: Pruebas de usabilidad, enfoque de usuario

En el aspecto de **efectividad** los observadores registraron datos que muestran que el usuario presentaba experiencia media en el manejo de la herramienta TFI y del entorno eclipse, ade-

más aun cuando se realizó una sesión previa de entrenamiento los usuarios no cumplieron el perfil adecuadamente. En experiencia de los usuarios los temas que se abordaron en el proyecto son de carácter particular, difíciles de comprender y asimilar, por ello algunas tareas resultaron complicadas.

1. Porcentaje promedio de tareas completadas: 96 %
2. Porcentaje de tareas completadas en el primer intento: 46 %
3. Solicitud de ayuda por parte del usuario (promedio): 50 %

En el aspecto de **eficiencia**, las funciones del sistema presentaron 2 fallos relacionados con la ejecución de la consulta. Sin embargo un punto a favor es el hecho de que se observó que el modelo mental del usuario coincidía con el flujo de la herramienta, es decir, el sistema realizaba las tareas de acuerdo a las expectativas del usuario.

1. Numero promedio aproximado de errores cometidos por el usuario: 1
2. Errores presentados en la herramienta y severidad: 1 error de severidad 1 y 2 errores de severidad 2 (Normal).
3. Tareas que tomaron el mayor tiempo de la evaluación: La tarea de Ejecutar expresión lógica.

En **satisfacción** los usuarios expresaron que la herramienta cumplía las tareas para las cuales fue desarrollada, pero que era recomendable estudiar la teoría que estaba involucrada, además consideraron que con la práctica podrían mejorarse el uso del sistema.

1. Adjetivos promedio de satisfacción: 6 adjetivos por usuario.
2. Adjetivos promedio de insatisfacción: 2 adjetivos por usuario

Además de estos factores los observadores anexaron observaciones, las cuales en conjunto con los resultados obtenidos del facilitador se resumen en los siguientes puntos:

1. El motivo de que algunos usuarios requirieran ayuda, es el hecho de que cualquier usuario tiene habilidades y características diferentes. Esto se apreció claramente en las pruebas.
2. Esta evaluación reflejó que la herramienta estaba lista pero los usuarios no estaban familiarizados al 100 % con los temas.
3. Es importante mencionar que los usuarios otorgaron a la herramienta una calificación promedio de: 8.6
4. El modelo mental reflejado por los usuarios es el modelo mental de la herramienta, sin embargo podemos notar que el usuario está fuertemente influenciado por el esquema de menús contextuales, lo cual es un factor muy importante para considerarse como mejora.
5. Los usuarios cuando no están familiarizados con la teoría involucrada dudan de sus propias elecciones, prueba de ello fue que el archivo de consulta contenía instrucciones de cómo elaborar las expresiones y solo uno de ellos se tomó el tiempo para leerlo.
6. La iconografía en general fue entendida completamente por los usuarios, sin embargo no identificaban la ubicación de las vistas o barras de herramientas; sin embargo este no es un problema de la herramienta, más bien es de familiarización con el entorno eclipse.
7. La herramienta cumple las funciones para las que fue conceptualizada.

Considerando los resultados de evaluación se seleccionaron las siguientes mejoras para la herra-

mienta:

1. Se integraron atajos contextuales sobre los archivos modelo y consultores.
2. Se integró un menú de ayuda contextual que explica los pasos para construir una expresión de consulta.
3. Se repitió el escenario que generó error de funcionalidad en el sistema y se determinó el motivo del error.

Las mejoras en la herramienta añadieron una nueva etapa de construcción del proyecto, la cual corrigió los errores indicados y liberó una versión final de la herramienta (V3.0), la cual reúne las expectativas del usuario cumpliendo sus requerimientos.

Capítulo 6

Conclusiones

El proyecto actual propone una herramienta (plug-in) Task Flow Interface construida de acuerdo a las especificaciones del Método Discovery. Con la que se pretende que los desarrolladores integren la verificación de especificaciones como parte del proceso de desarrollo de software de una forma rápida y sencilla.

La Metodología de desarrollo utilizada a lo largo del proyecto fué el Proceso Unificado, la cual demostró que sus artefactos continúan siendo útiles y valiosos. El Proceso Unificado además combinado con la programación orientada a objetos facilitan las estrategias de desarrollo.

Con la herramienta, el proceso de verificación de expresiones en un modelo se transforma en un proceso guiado, simple y claramente definido. A lo largo de este proceso se utiliza la filosofía de reutilizar componentes del entorno Eclipse en lugar de crearlos nuevamente.

El motivo de integrar la herramienta a uno de los entornos de desarrollo más utilizados es el hecho de que no es necesario aprender y familiarizarse con un ambiente nuevo. Las funcionalidades que se utilizan son: auto completado, verificación sintáctica, elementos GUI, consolas y administración de proyectos, son conceptos sencillos para los usuarios de eclipse. También con ayuda de este plug-in la comunidad Eclipse se informa del Método Discovery y los diagramas de tareas.

El plug-in desarrollado puede utilizarse como herramienta de apoyo en la enseñanza del Método Discovery, lógicas temporales y verificación de diagramas de tareas. La teoría que involucra el sistema es complicada, sin embargo, con la herramienta se encapsulan diversos aspectos complejos. Para alguien interesado en el tema con la herramienta puede comenzar a recrear verificaciones experimentales.

En el proyecto se comprobó que las expectativas del usuario se cumplen satisfactoriamente, además las evaluaciones se observó que los modelo mentales y las metáforas coinciden con el pensamiento del usuario. Desarrollar sistemas de acuerdo a las necesidades y entendimiento de los usuarios, ayuda a generar software que cumple las expectativas de los usuarios reales.

Recordemos finalmente que el plug-in desarrollado es una opción para verificar especificaciones basadas en los diagramas de tareas, sin embargo no es la única forma. Pero debido a las bases sólidas, experiencia particular y casos de prueba exitosos del Método Discovery en la Universidad de Sheffield, Inglaterra se recomienda utilizarla.

Bibliografía

[1] J.L.F. Aleman and A.T. Alvarez. Can intuition become rigorous? foundations for uml model verification tools. *Software Reliability Engineering*, 2001. 1.1.1, 1.1.2, 2.2

[2] Christel Baler and Joost Pleter Katoen. *Principles of Model Checking*. Massachusetts Institute of Technology, USA, 2008. ISBN 978-0-262-02649-9. 2.2.6

[3] P. Barthelmess and Anderson K. M. A view of software development environments based on activity theory. *Computer Supported Cooperative Work*, 11, 2002. 1.1.1

[4] Paul A. Booth. *An Introduction To Human-Computer Interaction*. Lawrence Erlbaum Associates, Londres, 1995. ISBN 0-86377-123-8. 2.4.2.2, 2.4.3

[5] Pablo Braceló. *Introducción a las Lógicas Temporales*. Chile, 2006. 2.2.6

[6] Kenneth C Louden. *Construccion de compiladores: principios y practica*. Paraninfo, Mexico, 2005. ISBN 970-686-299-4. 2.3, 2.3.3.3, 2.3.4, 2.3.5, 2.3.8

[7] Heitmeyer Constance. Managing complexity in software development with formally based tools. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 108, 2004. 1.1.1

[8] Edsko De Vries. (temporal) logic. *Universidad de Dublín*, 2006. 2.2.6

[9] 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), 1.1.1, 1.1.3, 1.1.4, 1.2, 2.2, 2.2.1, 2.2.2, 2.1, 2.2, 2.2.3, 2.3, 2.2.4, 2.2.5, 2.8, 2.2.6, 2.2.6, 2.2.6, 2.2.6.1, 3.2, 1, 4.5.1, 4.10

[10] Eclipse Foundation. Eclipse foundation: What is eclipse? http://www.eclipse.org/home/newcomers.php, Último acceso Mayo 2011. 1.3

[11] Eclipse Foundation. Xtext. http://www.eclipse.org/Xtext/community.html, Último acceso Mayo 2013. 4.5.1

[12] Martin Fowler and Kendall Scott. *UML Gota a Gota*. Pearson, México, 1999. ISBN 968-444-364-1. 2.5

[13] Lucas Francisco J., Fernando Molina, and Ambrosio Toval. A systematic review of uml model consistency management. *Information and Software Technology*, 51:1631–1645, 2009. 1.1.2, 2.2

[14] María del Mar Gallardo. Métodos para la construcción de software fiable. *Universidad de Málaga*, 2007. 1.1.4

[15] David Harel. On visual formalism. *Comm. of the ACM*, 31, 1998. 1.1.2

[16] Ivar Jacobson, Grady Booch, and James Rumbaugh. *El Proceso Unificado de Desarrollo de Software*. Addison Wesley, México, 2000. ISBN 84-7829-036-2. 2.5, 2.5.1, 3.1, 3.2, 3.3, 3.3.1, 3.3.2, 3.3.3, 3.4, 3.4.2, 4.2.1, 4.2.2, 4.3, 5.1

[17] Craig Larman. *Applying Uml And Patterns- An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, Vancouber. 3.3.1

[18] Peter Lohmann. Pspace-completeness of ltl/ctl* model checking. *Leibniz Universitšat Hannover*, 2007. 1.1.4, 2.2.6, 2.2.6

[19] Linda Macaulay. *Human-Computer Interaction for Software Designers*. International Thomson Publishing, London, 1995. ISBN 1-850-32177-9. (document), 2.1, 2.4, 2.4.1, 2.4.2.1, 2.4.2.3, 3, 2.4.2.4, 2.4.3, 2.4.3.1, 1, 4, 4, 2.17

[20] Karen Najera. Código innovare. *Software Gurú*, 2013. 2.1

[21] Nielsen Norman Group. Nn/g nielsen norman group, think aloud. http://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/, Último acceso Julio 2014. 5.3.1.6

[22] Greg OŽKeefe. *Model Driven Engineering Languages and Systems: Improving the Definition of UML*. Springer Berlin / Heidelberg, Berlin, 2006. ISBN 978-3-540-45772-5. 1.1.2, 2.2

[23] Terrence Parr. *The Definitive ANTLR Reference. Building Domain-Specific Languages(Pragmatic Programmers)*. Pragmatic Bookshelf, 2007. ISBN 0978739256. 2.3.4

[24] Javier Portillo. *Entorno Multidisciplinar para el Desarrollo de Sistemas de Control Distribuido con Requisitos de Tiempo Real*. PhD thesis, Escuela Superior de Ingenieros de Bilbao, Bilbao, ES, May 2004. 2.2

[25] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*, 1999. 1.1.1, 1.1.2, 2.2

[26] Paul Ryan. Eclipse survey results show growth in linux, open source. http://arstechnica.com/open-source/news/2009/05/eclipse-survey-results-show-growth-in-linux-open-source.ars, Último acceso Mayo 2011. 1.3

[27] Jesús Salas. *Sistemas Operativos y Compiladores*. McGraw Hill, Madrid, España, 1988. ISBN 84-7615-205-1. 2.3.2

[28] Walt Scacchi. Process models in software engineering. *Encyclopedia of Software Engineering,2nd Edition*, 2001. 1.1.1

[29] Joseph Schmuller. *Aprendiendo UML en 24 Horas*. Prentice Hall Hispanoamericana S.A., México, 2003. ISBN 968-444463X. 2.2

[30] DeInterfaz Services-Consulting S.L. Metricas de usabilidad. http://www.deinterfaz.com/blog/medir-metricas-de-usabilidad, Último acceso Julio 2014. 5.3.1.1

[31] Ben Shneiderman. *Designing the User Interface- Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 1998. ISBN 0-201-69497-2. 2.1, 2.4, 2.4.1, 2.4.2.1, 2.4.2.3, 2.4.3.1

[32] A.J.H. Simons. Object discovery - a process for developing medium-sized applications. *European Conference on Object-Oriented Programming (ECOOP '98)*, 1998. (document), 1.1.3, 2.2

[33] A.J.H. Simons. Object discovery - a process for developing applications. *Conference on Object Technology (OT '98)*, 1998. 1.1.3, 2.2

[34] A.J.H. Simons. Discovery method. (discovery and invention). *(in preparation - personal communication), University of Sheffield*, 2007. 1.1.3

[35] A.J.H. Simons. The discovery method for object-oriented software engineering. http://staffwww.dcs.shef.ac.uk/people/A.Simons/discovery/, Último acceso Noviembre 2011. 2.2

[36] Ian Sommerville. *Ingeniería del software*. Editorial Pearson Educación, S.A., Madrid España, 2005. ISBN 8478290745. 1.1.1, 2.2, 2.4, 2.4.1

[37] Bernard Teufel, Stephanie Schmidt, and Thomas Teufel. *Compiladores. Conceptos fundamentales*. Addison-Wesley Iberoamericana, Mexico, 1995. ISBN 0-201-65365-6. (document), 2.3.1, 2.3.2, 2.3.3.1, 2.3.3.3, 2.13, 2.3.3.3, 2.3.3.4, 2.3.3.5, 2.3.5, 2.3.6, 2.3.6.2, 2.3.6.3

[38] Jean-Paul Tremblay and Paul G. Sorenson. *The theory and practice of compiler writing*. McGraw Hill, New York, EU, 1985. ISBN 0-07-Y66616-4. 2.3.2, 2.3.3.2, 2.3.3.3

[39] Alfred V. Aho. *Compiladores: principios, tecnicas y herramientas*. Editorial Pearson Educación, S.A., 1990. ISBN 968-444-333-1. (document), 2.3, 2.3.2, 2.11, 2.3.2, 2.3.3.3, 2.3.3.4, 2.3.4, 2.3.5, 2.3.6.1, 2.3.7, 2.3.8

[40] Andrei Voronkov's. *Linear Temporal Logic, LTL.* Capitulo 14, 2009. 1.1.4

[41] Wucius Wong. *Fundamentos del Diseño*. Gustavo Gili S.A., Barcelona, 1995. ISBN 9788425216435. 2.4.1

[42] Lan Zhang, Ullrich Hustadr, and Clare Dixon. Ctl-rp: A computational tree logic resolution prover. *AI Communications*, 0921-7126, IOS Press, 2007. 1.1.4

Apéndice A

Protocolo de Experimentación

El proceso de evaluación con usuarios reales del sistema se realizará considerando el siguiente guión de evaluación el cual describe diversos aspectos a considerarse en el proceso, su estructura es la siguiente:

A.1. Guion General del Proyecto

A.1.1. Nombre del Proyecto

Desarrollo de una interfaz para la verificación de diagramas de tareas en la especificación del software.

A.1.2. Descripción del Proyecto

Task Flow Diagrams Verification Interface(TFI) es un plug-in desarrollado para el entorno Eclipse IDE el cual permite la verificación de propiedades sobre modelos del álgebra de tareas del Método Discovery.

El proyecto tiene como objetivo integrar la verificación de especificaciones al proceso de desarrollo de software como actividad común y cotidiana, permitiendo que desde etapas tempranas de los proyectos se verifique la consistencia de los sistemas de software a implementar.

A.1.3. Dirección del proyecto

http://www.utm.mx/~taskmodel/projects/modelCheckingEclipse/ org.plug.taskalgebraproject.all_update /2.0/ , en su versión 2.0.

A.2. Proceso de Evaluación

A.2.1. Objetivos

- Evaluar el proyecto con usuarios reales en un ambiente controlado.
- Observar cómo el usuario resuelve las tareas asignadas utilizando las funcionalidades del sistema.
- Analizar y obtener información referente a las impresiones, opiniones y experiencia del usuario al utilizar el sistema.
- Analizar la correspondencia del modelo mental proyectado en el sistema y el modelo real del usuario.

A.2.2. Aspectos a registrar en cada evaluación

En la evaluación del proyecto se desea obtener, observar y evaluar 3 aspectos importantes del sistema: efectividad, eficiencia y satisfacción. Estos factores se registrarán y analizarán de manera detallada mediante los siguientes aspectos:

- Porcentaje total de tareas completadas
- Porcentaje de tareas realizadas en el primer intento
- Porcentaje de éxitos y fracasos
- Numero de veces que los usuarios solicitan ayuda.
- Numero de errores cometidos a lo largo de las pruebas
- Severidad de los problemas presentados.
- Relación entre los adjetivos positivos y negativos expresados por los usuarios.
- Entendimiento de la iconografía.
- Inquietudes, dudas, sugerencias y actitud de los usuarios ante el proyecto.

A.2.3. Requerimientos

Número de Usuarios. 3

Duración de la evaluación. 20-25 minutos por usuario.

Perfil de usuario. Los usuarios deben cumplir estrictamente el siguiente perfil:

- Desarrolladores de software o estudiantes relacionados con la ingeniería de software.
- Utilizar/Haber utilizado el entorno Eclipse IDE para el desarrollo de software.
- Tener conocimientos del Método Discovery, Modelos, Expresiones Lógicas y Álgebra de tareas (sesión de capacitación).

Observadores: Los observadores deben tener habilidades para analizar comportamiento de usuario y tener conocimiento de ingeniería de software.

Facilitadores: Este perfil lo deben llenar personas con habilidades para relacionarse con los usuarios, tener conocimiento del Método Discovery, Modelos, Expresiones Lógicas y Álgebra de tareas.

Reclutamiento de Usuarios: Los usuarios han sido seleccionados por sus características del área de postgrado de la maestría en cómputo aplicado (2) y medios interactivos (1). Los observadores son profesores investigadores con conocimientos del álgebra de tareas y bases de usabilidad.

Lugar de Evaluación y Características del Entorno: El entorno no se consideró como factor decisivo en esta evaluación, por lo que se seleccionó un ambiente controlado: Laboratorio de Usabilidad de la Universidad Tecnológica de la Mixteca.

Lugar y hora de inicio de las evaluaciones: miércoles 19 de junio de 2013, 10:00 am.

A.2.4. Dinámica Previa a la Evaluación

1. Obtener el consentimiento de participación de los usuarios en las pruebas (ver forma de consentimiento del anexo B).
2. Informar a los usuarios sobre el lugar, ubicación y hora de la evaluación.
3. Preparar el material del proyecto: Versión final del plug-in, Software Eclipse IDE Indigo y Computadora.
4. Imprimir en caso de ser necesario la lista de tareas del usuario y el cuestionario de evaluación (ver anexo B).
5. Reunir recursos extras: cámara fotográfica, video y grabadora.

A.2.5. Proceso de Evaluación: Desarrollo

1. Bienvenida (responsable del proyecto, facilitador)
2. Explicación del motivo de las pruebas, del proyecto y de lo que se realizara en las pruebas (facilitador)
3. Presentación del Sistema, escenario y de la lista de tareas a realizar (facilitador).
4. Realización de las pruebas (usuario)
5. Fin de las pruebas (facilitador)
6. Sesión breve de preguntas (facilitador y usuarios)

En el desarrollo del proyecto los observadores deberán realizar anotaciones sin influir en la realización de las pruebas de los criterios especificados en el formato de métricas del anexo B.

A.2.5.1. Escenario de evaluación

El usuario experto en desarrollo de software preocupado por la seguridad y correcta definición de las especificaciones de los proyectos, ha decidido aplicar la técnica de verificación de

modelos; para ello hace uso de la herramienta TFI.

En esta ocasión el sistema sobre el cual debe verificar propiedades es el sistema de reservación de libros en biblioteca de una universidad. El diagrama de flujo de tareas del sistema se muestra en la figura A.1.

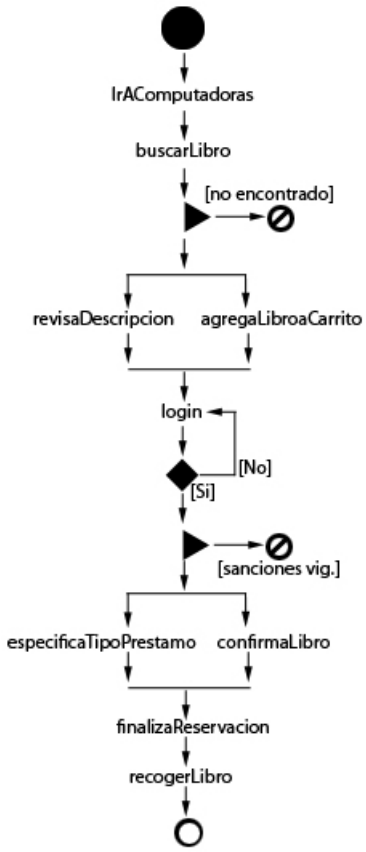

Figura A.1: Diagrama de flujo de tareas del sistema de reservación de libros en una biblioteca

A.2.5.2. Lista de tareas

1. Acceder al entorno de desarrollo Eclipse (No evaluar).
2. Crear un nuevo proyecto General (No evaluar).
3. Activar la perspectiva de trabajo LC(TL) Perspective.
4. Importar el modelo de tareas: Model.tfa, localizado en la Carpeta Proyectos del Escritorio.
5. Estructurar las siguientes consultas:
   a) CTL, Operador Exist Finally, actividad: recogerLibro
   b) LTL, Operador Not, actividad: buscarLibro
6. Configurar los compiladores LTL y CTL del proyecto, localizados en la carpeta Proyectos del Escritorio.

7. Ejecutar las consultas y describir brevemente los resultados.

A.2.5.3. Técnica de interacción con el usuario

La interacción con el usuario se realizará por medio de *Think Aloud*. La técnica *Think Aloud* consiste en indicar al usuario que explicará en voz alta las acciones y decisiones que estaba realizando a lo largo de las pruebas, de esta manera se puedo obtener mayor alimentación y entender los modelos mentales. También se utilizará un cuestionario breve para conocer el entendimiento de la iconografía, grado de satisfacción con el sistema y nuevas mejoras.

Apéndice B

Materiales Extras de Evaluación

B.1. Formato de Consentimiento de Usuario

B.1.1. Evaluación del Proyecto

Desarrollo de una interfaz para la verificación de diagramas de tareas en la especificación del software.

B.1.2. Director de Tesis

Dr. Carlos Alberto Fernández y Fernández

B.1.3. Descripción

Usted ha sido invitado a participar en la evaluación del proyecto *Task Flow Diagrams Verification Interface* (TFI). TFI es un plug-in desarrollado para el entorno Eclipse IDE el cual permite la verificación de propiedades sobre modelos del álgebra de tareas del Método Discovery.

El proyecto tiene como objetivo integrar la verificación de especificaciones al proceso de desarrollo de software como actividad común y cotidiana, permitiendo que desde etapas tempranas se verifique la consistencia de los sistemas a implementar.

A lo largo de la evaluación usted realizara diversas actividades utilizando el sistema y se le realizarán una serie de preguntas relacionadas con el proyecto. Las evidencias de esta evaluación se registraran por medio de video, audio y fotografías; estas últimas se publicarán en el reporte de evaluación.

B.1.4. Lugar y Hora

Miércoles 19 de junio del 2013, 10:00-11:20 am

B.1.5. Tiempo Requerido

La evaluación tomará aproximadamente 20-25 minutos por participante.

B.1.6. Riesgos y Beneficios

El riesgo que se tiene en la evaluación puede ser un desfase no planeado de 5 minutos en la hora especificada. Respecto a los beneficios no podemos garantizar ni prometer que va a recibir beneficios en esta evaluación. Usted tiene la libertad de decidir participar o no en esta evaluación.

B.1.7. Pago

No Aplica.

B.1.8. Derechos del Interesado

Si usted ha leído este documento y ha decidido participar en esta evaluación, entendemos por favor que su participación es voluntaria. Usted tiene el derecho de retirar su consentimiento o suspender su participación en cualquier momento sin sanción o pérdida de los beneficios citados anteriormente. Sus datos personales no serán publicados en el estudio, todo se mantendrá anónimo.

B.1.9. Información de Contacto

Si usted tiene alguna pregunta, duda o queja sobre esta evaluación. Puede enviar sus inquietudes a la dirección de correo desarrollador del proyecto: Hermenegildo Fernández Santos, hfernandez801@gmail.com o bien al director de tesis: Dr. Carlos Alberto Fernández y Fernández, caffmx@gmail.com.

Contacto Independiente: Si usted no está satisfecho con la forma en que se llevó a cabo este estudio, o si usted tiene alguna inquietud, queja o pregunta general acerca de la evaluación o sus derechos como participante por favor póngase en contacto con el jefe de área del Laboratorio de Usabilidad, profesor Mario A. Moreno Rocha de esta universidad. Tels. 953 5320399, 5320214 y 5324560.

B.1.10. Autorización

Doy mi consentimiento para ser fotografiado durante este estudio:
Escriba sus Iniciales: ___Si ___No

Doy mi consentimiento para ser grabado (voz) durante este estudio:
Escriba sus Iniciales: ___Si ___No

Doy mi consentimiento para ser grabado (video) durante este estudio:
Escriba sus Iniciales: ___Si ___No

FIRMA _____ FECHA (Enterado) _____

Protocolo-Fecha de Aprobación: 18 de Junio de 2013

Protocolo-Fecha de expiración: 18 de Julio de 2013

B.2. Cuestionario de Satisfacción de Usuario

1. ¿Se enfrentó a algún problema al realizar las tareas? En caso afirmativo describir el problema, la tarea y la situación (retroalimentación).
2. Para la construcción de las expresiones que método eligió: ¿Construcción manual o Asistente? ¿Por qué?
3. En una escala del 0-10 y considerando la utilidad, eficacia y eficiencia, ¿qué puntuación le otorgaría al proyecto después de haber realizado la evaluación?
4. Mencione algunas recomendaciones para la mejora del proyecto.
5. En su opinión, ¿considera que es necesario integrar alguna funcionalidad extra en la herramienta?
6. Explicar la funcionalidad de cada icono de la siguiente figura:

Figura B.1: Iconografía utilizada en el proyecto.

B.3. Métricas de Evaluación

Los observadores registrarán hallazgos importantes presentados en el proceso a través del siguiente formato de métricas.

Métrica	*Usuario1*	*Usuario2*	*Usuario3*
Efectividad			
% total de tareas completadas			
% de tareas completadas en el primer intento			
% proporción de éxitos sobre fracasos			
Numero de veces en que los usuarios solicitan ayuda por no saber que hacer			
Eficiencia			
Numero aproximado de errores cometidos por el usuarios			
Severidad de los problemas presentados(escala del 1[sin problema]-4 [grave])			
Satisfacción			
Proporción de adjetivos positivos/negativos que el usuario expresa			
Numero de veces que el usuario expresa satisfacción o insatisfacción			
Calificación otorgada al proyecto por el usuario			
Anotaciones generales del observador			

Cuadro B.1: Métricas de evaluación a utilizarse en el proyecto.

Apéndice C

Resultados de Evaluación de Usuarios

Las conclusiones del proceso de evaluación del sistema por parte del usuario presentadas en el capítulo 5 resumen diversos aspectos importantes; los cuales resultaron de un proceso de observación, retroalimentación y análisis entre los involucrados del proyecto y los participantes.

Los objetivos principales por los cuales se realizó una evaluación con usuarios fueron: determinar en qué grado el sistema cumplía aspectos de satisfacción, efectividad, eficiencia y satisfacción de usuario. A continuación se presentan con mayor detalle los eventos más importantes del proceso.

C.1. Evidencias de Evaluación

En el protocolo de experimentación del anexo A y en el capítulo 5 se evidencio que los roles y los participantes utilizados fueron los siguientes:

- Usuarios (3). Estudiantes con experiencia en el desarrollo de software y con conocimientos en los principales temas requeridos para el proceso.
- Observadores (2). Profesores de la Universidad Tecnológica de la Mixteca: Dr. Carlos Alberto Fernández y Fernández y M.C. Mario Alberto Moreno Rocha.
- Facilitador (1). El proceso de evaluación fue dirigido por el responsable del proyecto: Hermenegildo Fernández Santos.

Las evidencias de la evaluación del proyecto pueden apreciarse en las figuras C.1, C.2 y C.3.

Figura C.1: Proceso de evaluación, perspectiva de usuarios y observadores.

Figura C.2: Proceso de evaluación, perspectiva de usuarios y observadores.

Figura C.3: Proceso de evaluación, perspectiva de usuarios y observadores.

Los resultados reportados por los observadores a través del documento de métricas de evaluación (ver anexo B) se describen en las tablas C.2 y C.1 .

Métrica	*Usuario1*	*Usuario2*	*Usuario3*
Efectividad			
% total de tareas completadas	100 %	100 %	100 %
% de tareas completadas en el primer intento	20 %	20 %	40 %
% proporción de éxitos sobre fracasos	1/0	1/0	1/0
Numero de veces en que los usuarios solicitan ayuda por no saber que hacer	5	3	2
Eficiencia			
Numero aproximado de errores cometidos por el usuarios	1	1	1
Severidad de los problemas presentados(escala del 1[sin problema]-4 [grave])	1	1	1
Satisfacción			
Proporción de adjetivos positivos/negativos que el usuario expresa	5/5	7/3	7/0
Numero de veces que el usuario expresa satisfacción o insatisfacción	S=2, I=2	S=2, I=1	S=2, I=1
Calificación otorgada al proyecto por el usuario	8	8	10
Anotaciones generales del observador		La tarea de configurar bibliotecas falló en su primer intento debido a problemas de los recursos externos	El usuario difícilmente observo las instrucciones presentadas al momento de construir las expresiones.

Cuadro C.1: Resultados detallados indicados por el observador 1

Métrica	*Usuario1*	*Usuario2*	*Usuario3*
Efectividad			
% total de tareas completadas	100 %	80 %	100 %
% de tareas completadas en el primer intento	20 %	40 %	40 %
% proporción de éxitos sobre fracasos	1/0	1/0	1/0
Numero de veces en que los usuarios solicitan ayuda por no saber que hacer	El usuario necesitó ayuda en cada paso de la evaluación	El usuario necesitó poca ayuda	El usuario no requirió de ayuda
Eficiencia			
Numero aproximado de errores cometidos por el usuarios	0	2	2
Severidad de los problemas presentados(escala del 1[sin problema]-4 [grave])	1	2(construir expresiones)	2(construir expresiones)
Satisfacción			
Proporción de adjetivos positivos/negativos que el usuario expresa	9/6	4/0	3/1
Numero de veces que el usuario expresa satisfacción o insatisfacción	S=1	S=1	S=1
Calificación otorgada al proyecto por el usuario	8	8	10
Anotaciones generales del observador	El facilitador ayuda al usuario, implica que el usuario no domina la base teórica del sistema.	Los asistentes utilizados en la construcción de expresiones realmente simplifican la tarea al usuario.	El usuario no presento ningún problema durante las pruebas

Cuadro C.2: Resultados detallados indicados por el observador 2

Apéndice D

Manual de Usuario de la Herramienta

D.1. Introducción

Task Flow Diagrams Verification Interface es un plug-in/sistema desarrollado para el Eclipse-IDE; a través del cual es posible realizar la verificación de especificaciones de los diagramas de tareas del Método Discovery.

El objetivo del proyecto es integrar la verificación de especificaciones al proceso de desarrollo de software, esto para construir sistemas consistentes con las especificaciones del cliente. La decisión de integración se basa en el concepto de familiarización-uso que se genera entre los usuarios y Eclipse.

El plug-in implementado ha sido desarrollado para usuarios con conocimientos de la especificación del Método Discovery, álgebra de tareas, lógicas temporales y verificación de modelos.

D.2. Objetivos

Desde los inicios del proyecto, los objetivos generales son los siguientes:

- Ofrecer una interfaz a través de la cual sea posible la estructuración y verificación de consultas en lógicas temporales (LTL y CTL) aplicadas a los diagramas de flujo de tareas.
- Permitir a los usuarios involucrados realizar el proceso de ejecución de verificaciones de especificaciones en una forma simple y transparente.
- Simplificar el proceso de interpretación de resultados de la verificación de especificaciones en los modelos del álgebra de tareas.
- Mejorar las herramientas de verificación en consola hacia un concepto gráfico, simple, rápido, eficiente y abstracto.
- Integrar la tarea de verificación de modelos al diseño software previo al proceso de codificación.
- Desarrollar herramientas que posicionen al Método Discovery como complemento/alternativa

del desarrollo de software.

D.3. Instalación

Actualmente el proyecto ha sido liberado bajo la versión 3.0. Antes de iniciar la instalación del plug-in asegúrese de tener instalados los siguientes complementos:

1. JDK 7 *update* 21 o superior.
2. Eclipse Helios o superior.

A continuación instale la base *Xtext* mediante los siguientes pasos:

D.3.1. Instalación de Xtext

1. Inicie la versión de Eclipse de su preferencia (recomendable la versión Helios).
2. Haga clic en la opción *Help* > *Install New Software*.
3. En la ventana de *Install New Software* añada un nuevo repositorio para *Xtext* con la siguiente dirección: *http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/*
4. De la lista de versiones seleccione Xtext-2.4.2.
5. Una vez finalizado el proceso reinicie el entorno Eclipse y continué con la instalación del plug-in.

D.3.2. Instalación del plug-in TFI

Para la instalación del plug-in siga los siguientes pasos:

1. En el entorno Eclipse haga clic en la opción *Help*>*Install New Software*.
2. En la venta de *Install New Software* añada un nuevo repositorio con el nombre **Task Algebra Plug-in**, con la siguiente dirección: *http://www.utm.mx/~taskmodel/projects/ modelCheckingEclipse/org.plug.taskalgebraproject.all_update/Version_Final/*
3. A continuación haga clic en siguiente, acepte los términos de licencia y haga clic en finalizar.
4. Eclipse automáticamente comenzara la instalación del plug-in, así como de las dependencias.
5. Cuando el proceso finalice, reinicie el entorno.

Una vez instalado el plug-in, es recomendable descargar los ejecutables de los compiladores LTL y CTL disponibles como archivo comprimido en la siguiente dirección: *http://www.utm.mx/~taskmodel/projects/ modelCheckingEclipse/Compiladores/*

D.4. Componentes del Sistema

El primer paso antes de comenzar a explorar los aspectos las partes más importantes del plug-in es la activación de la perspectiva *(LC)TL Perspective* . Al activarse, el entorno eclipse mostrará

los componentes del sistema con vista similar a la figura D.1.

De la figura D.1 los elementos más representados son los siguientes:

1. Explorador de Proyectos
2. Área de edición
3. Consola de resultados
4. Vista de Resultados gráfica
5. Barra de herramientas LTL/CTL

D.5. Accediendo a las funciones

Las funcionalides del plug-in/sistema se relacionan directamente con el proceso de verificación de especificaciones en modelos. A continuación se presentan diversas secciones en las cuales cada uno de los apartados permite realizar una etapa de la verificación.

D.5.1. Creación del proyecto.

Para comenzar el proceso de verificación de especificaciones, comience por crear un proyecto. Este proceso se realiza en los siguientes pasos:

1. Haga clic en el menú *Archivo*> *Nuevo Proyecto*
2. De las categorías elija el tipo *General* y de clic en *Siguiente*.
3. A continuación asigne un nombre al proyecto y haga clic en *Finalizar*.

D.5.2. Cambiar Perspectiva del Proyecto

Para establecer la perspectiva relacionada al proceso de verificación de especificaciones en el entorno, siga los siguientes pasos:

1. Haga clic en el menú *window* > *Open Perspective* > *Otra*
2. En las opciones disponibles seleccione *(LC)TL Perspective*.
3. Automáticamente se activan las vistas del proyecto, las cuales se muestran en la figura D.1.

D.5.3. Álgebra de Tareas del Proyecto

Dentro de este apartado tendrá dos opciones: crear el álgebra de tareas de forma manual o bien importar un modelo existente. Para cada una de las opciones siga los siguientes pasos:

D.5.3.1. Crear Álgebra de Tareas

1. Haga clic derecho en un proyecto particular y a continuación elija la opción *Nuevo* > *Otro*. También puede activar este paso en el menú archivo.

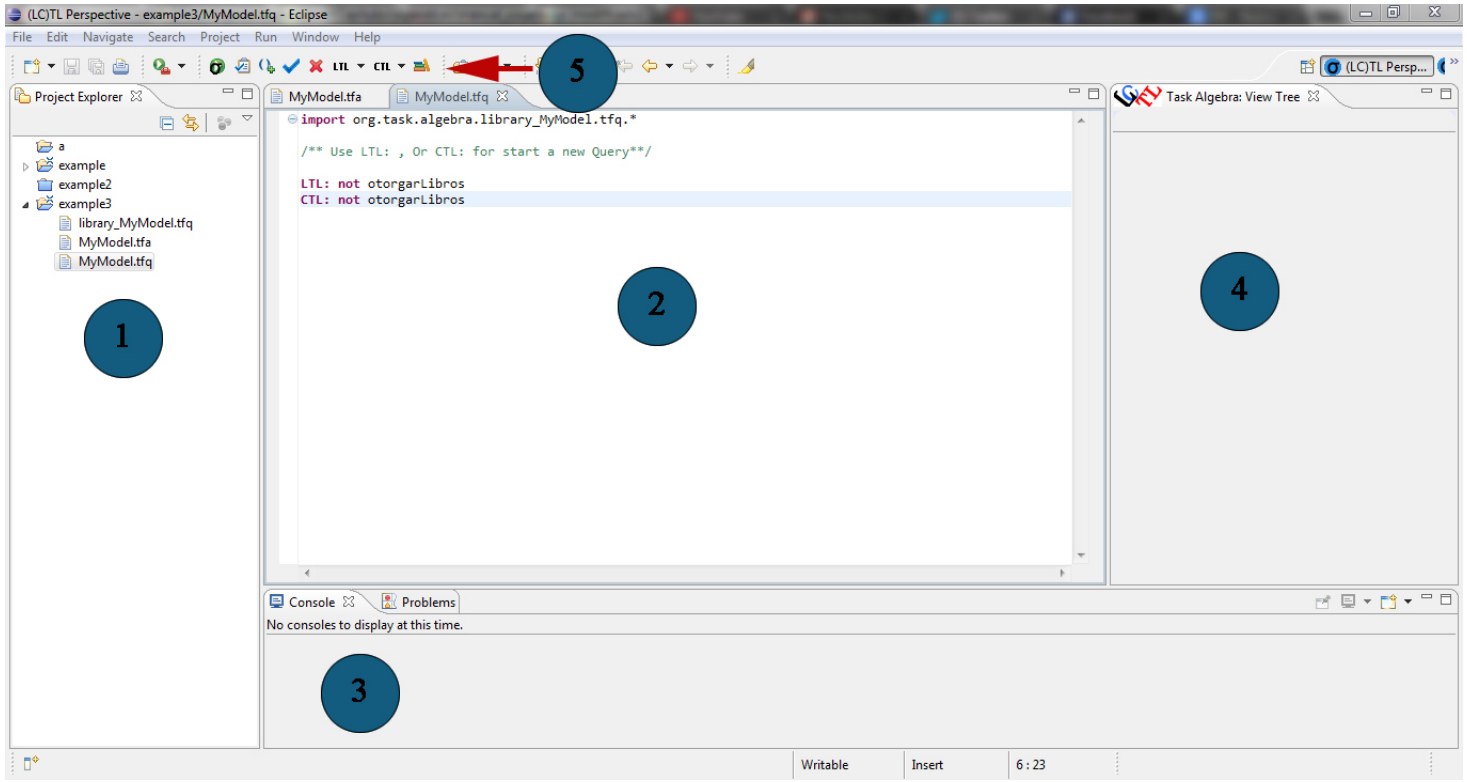

Figura D.1: Vistas y componentes principales del proyecto, TFI.

2. En la ventana de selección de categorías, haga clic en *Model Checking* > *New Task Algebra File* (ver figura D.2).

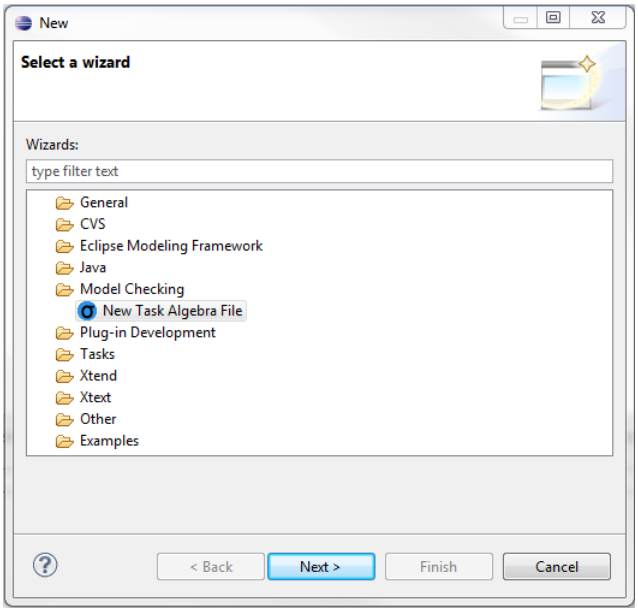

Figura D.2: Nuevo modelo de álgebra de tareas

3. A continuación asigne un nombre al archivo de álgebra con extensión tfa o bien utilice el sugerido por el asistente.
4. Haga clic en finalizar.
5. Si es la primera vez que crea un modelo en el proyecto, Eclipse solicitará su confirmación para asociar el proyecto a un tipo *Xtext Nature* (ver figura D.3).

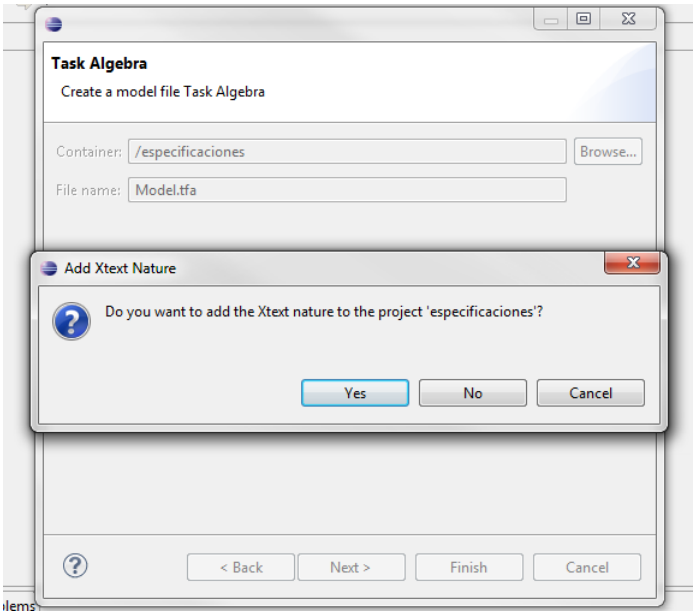

Figura D.3: Asociación de tipo *Xtext nature* al proyecto.

6. Al finalizar los pasos anteriores el modelo tfa se abrirá automáticamente para que usted pueda comenzar con la construcción.

Recuerde que al crear un modelo de tareas, las especificaciones del contenido deben estar guiadas por el álgebra de tareas descrita en el capítulo 4. En caso de que el álgebra de tareas no se especifique adecuadamente se mostrar errores de sintaxis y una breve descripción para solucionarlos (ver figura D.4)

D.5.3.2. Importar Algebra de Tareas

1. Haga clic derecho en un proyecto particular y a continuación elija la opción *Importar*. También puede activar este paso en el menú archivo.
2. En la ventana de selección de categorías, haga clic en *Model Checking* > *Import Task Algebra File* .
3. A continuación seleccione el archivo a importar y especifique un nuevo nombre para el nuevo archivo con extensión tfa.
4. Haga clic en finalizar
5. Si es la primera vez que crea/importa un modelo en el proyecto, Eclipse solicitará su confirmación para asociar el proyecto a un tipo *Xtext Nature* (ver figura D.3).
6. Al finalizar los pasos anteriores el modelo se abrirá automáticamente para que pueda revisar su estructura y en caso de que existan errores de sintaxis estos serán detectados por el sistema (ver figura D.4).

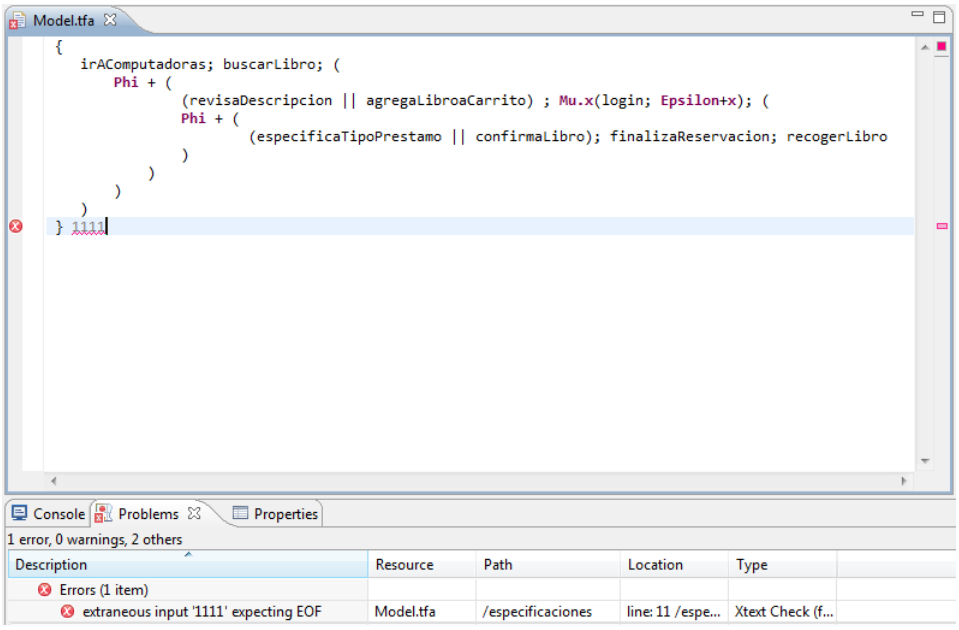

Figura D.4: Detección de errores de sintaxis en el modelo tfa.

D.5.3.3. Crear Especificaciones/Propiedades a Verificar en el Modelo.

Una vez que el modelo algebraico esta correcto (no presenta errores de sintaxis), usted puede comenzar a realizar verificaciones, para ello siga los siguientes pasos:

1. Con el archivo del álgebra abierto (tfa) haga clic en la opción *New Logic Query* de la barra de herramientas LTL/CTL. También puede activar esta opción seleccionado Model Checking del menú contextual del archivo del álgebra de tareas (ver figura D.5).

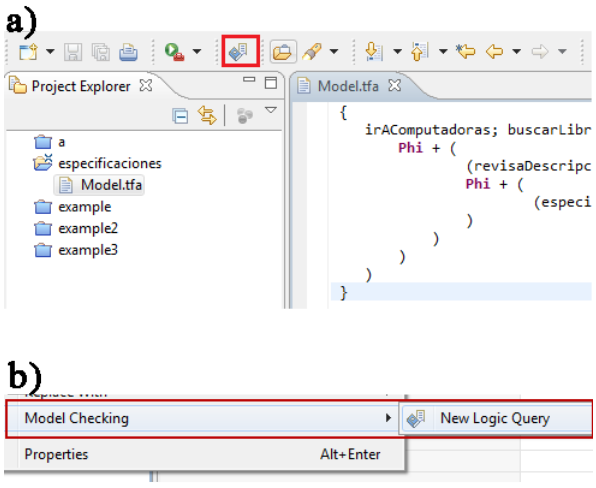

Figura D.5: Opción New Logic query, dos posibles atajos.

2. Si el modelo del álgebra está libre de errores, el plug-in creará y abrirá automáticamente un archivo consultor. Este archivo tendrá el mismo nombre del modelo y una extensión tfq (ver figura D.6).

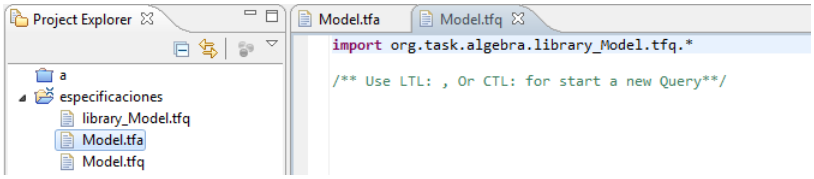

Figura D.6: Creación del archivo de consulta, tfq.

En este paso podemos notar que en la barra de herramientas LTL/CTL de Eclipse se han añadido otras opciones (ver figura D.7). La descripción de cada una de estas se muestra en la tabla D.1 .

Figura D.7: Nuevas opciones añadidas a la barra de herramientas LTL/CTL de Eclipse.

Opción	Descripción
	Permite añadir a la posición actual del cursor tareas especificadas en el modelo algebraico tfa.
	Añade un par de paréntesis agrupando la selección actual en la posición del cursor.
	Añade la tarea simple true.
	Añade la tarea simple false.
LTL	Muestra los operadores LTL combinados con las tareas del modelo algebraico para que el usuario pueda seleccionar alguno de ellos.
CTL	Muestra los operadores CTL combinados con las tareas del modelo algebraico para que el usuario pueda seleccionar alguno de ellos.

Cuadro D.1: Descripción de las funcionalidades disponibles para la construcción de expresiones LTL/CTL.

Al momento de especificar expresiones en el archivo consultor es importante que considere las siguientes indicaciones:

1. En un consultor se admiten dos tipos de instrucciones globales: LTL y CTL. Para iniciar especificar una consulta utilice la sintaxis siguiente: <*LTL:/CTL:*> <*expresión lógica*> , por ejemplo:
   LTL: not a
   CTL: not b
2. En un archivo de consulta puede especificar tantas especificaciones necesite, unicamente escriba cada especificación en una línea diferente con su inicio respectivo LTL: o bien CTL:
3. Es posible construir especificaciones LTL/CTL de forma manual o automática a través de las funciones descritas en la tabla D.1.

A medida que construya las especificaciones, Eclipse le indicara si existen errores de sintaxis en la consulta mediante la vista de problemas, además recuerde que si desea auto-completar líneas, utilice la combinación de teclas *Ctrl+Space* disponible en el entorno.

También puede modificar en cualquier momento el modelo algebraico y de manera automática todas las tareas y operaciones se actualizarán automáticamente.

D.5.3.4. Configuración de Bibliotecas

Las bibliotecas son una parte importante del proyecto debido a que permiten ejecutar las expresiones LTL/CTL especificadas en los archivos consultores. Antes de configurarlas asegúrese de haber descargado los archivos especificados en el proceso de instalación, y a continuación realice los siguientes pasos:

1. Haga clic en la opción *Config Build Settings* de la barra de herramientas LTL/CTL (ver figura D.8).

Figura D.8: *Config Build Settings*.

2. En el cuadro de dialogo especifique los archivos compiladores descargados en el proceso de instalación (ver figura D.9).

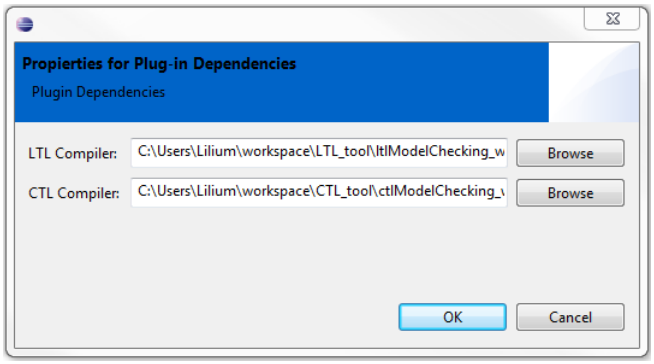

Figura D.9: Configuración de las bibliotecas del proceso de verificación de especificaciones.

3. Una vez configurados los archivos que utilizará haga clic en OK.

D.5.3.5. Ejecución de las Expresiones de Verificación

Una vez finalizada la especificación de expresiones de verificación (archivo de consulta) y cumpliendo los requisitos de sintaxis. Puede realizar el proceso de ejecución mediante los siguientes pasos:

1. Con el archivo de expresiones (tfq) abierto haga clic en la opción *Run Logic Expression* de la barra de herramientas LTL/CTL. También puede activar esta opción seleccionado Model Checking del menú contextual del archivo consultor (ver figura D.10).

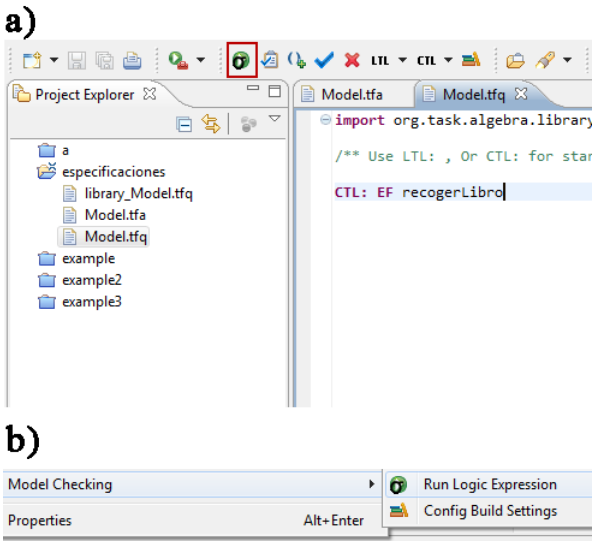

Figura D.10: Opción *Run Logic Expression*, dos posibles atajos.

2. Automáticamente el sistema ejecutará las expresiones especificadas por el usuario, y al finalizar la ejecución se presentaran los resultados modo texto en la consola y de forma gráfica en el apartado visual (ver figura D.11).

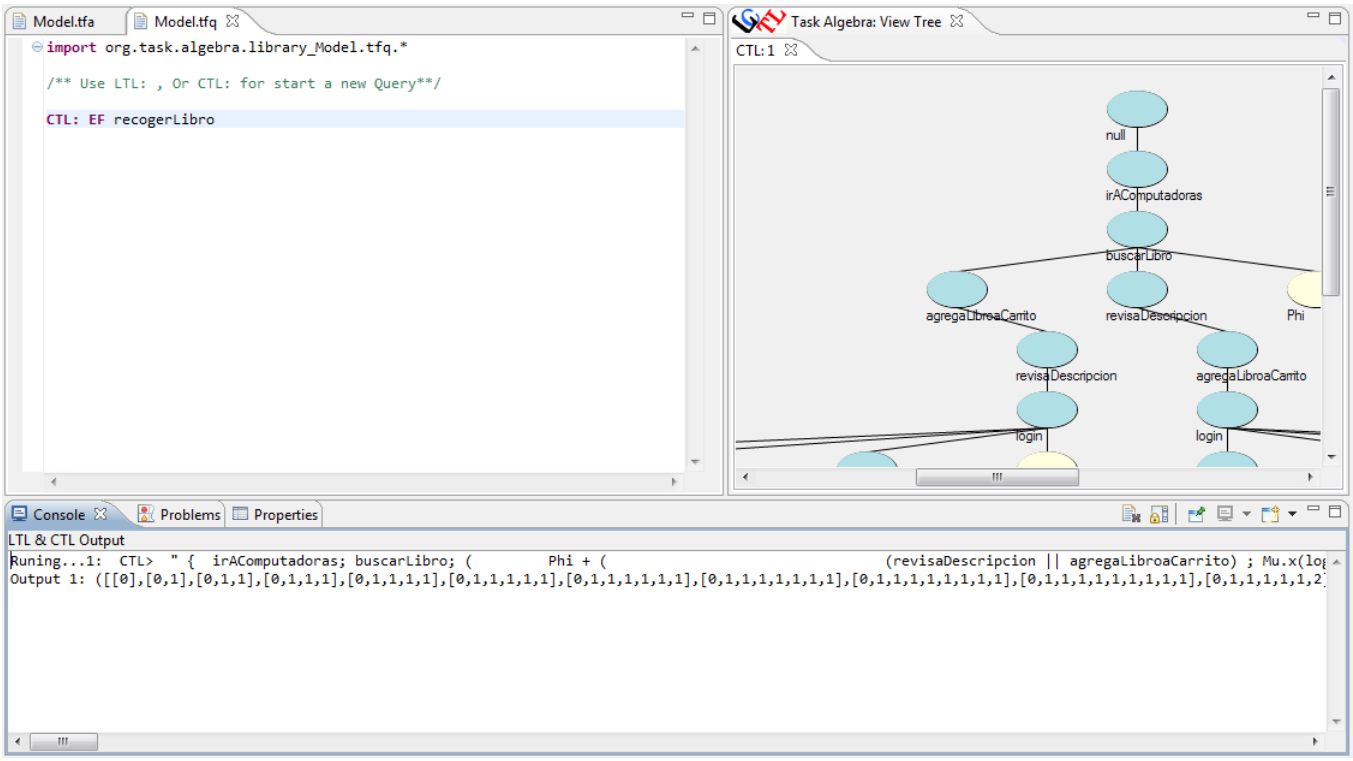

Figura D.11: Ejecución y representación de resultados resultante del proceso de verificación de especificaciones.

3. La interpretación de estos resultados requiere tener conocimientos de verificación de modelos y lógicas temporales.

D.6. Mensajes de Error Comunes

Cuando algún requerimiento de la funcionalidad no se cumple al realizar la invocación, el plug-in responde con un mensaje informativo indicando los motivos que lo originaron y una breve descripción para solucionarlo (ver figura D.12).

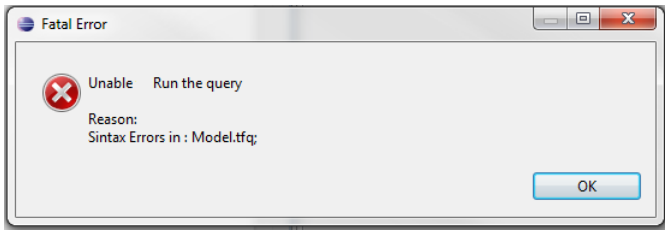

Figura D.12: Control de excepciones y mensajes de usuario del plug-in en Eclipse.

Los diversos mensajes considerados en el sistema puede originarse dentro de los siguientes ámbitos:

Tarea: Nuevo consultor
Excepciones:

- La funcionalidad se invoca con errores de sintaxis en el modelo
  - **Mensajes:**"Unable create LTL,CTL query ", "Sintax Errors in Task Algebra File"

  Tarea: Ejecución
  Excepciones:
- La funcionalidad se invoca con errores de sintaxis en el modelo/archivo librería/consultor
  - **Mensajes:** "Unable to Run the query ", "Sintax Errors in : "+Errores
- Los recursos a utilizarse no se han definido en el archivo de configuración LtlCtl.ini
  - **Mensajes:** "Unable to Run the query", "LTL,CTL path compilers are undefined"
- El modelo de tareas/archivo biblioteca no existe en el proyecto
  - **Mensajes:**"Unable to Run the query: Review compilers and files requiered", "The following files are requiered:"+Errores
- Excepción en tiempo de ejecución de las expresiones.
  - **Mensajes:** "Fatal Error", "Unabled to RUN the query with defined compiler", e.getMessage()
- Los recursos externos no pueden ejecutarse en el sistema operativo actual.
  - **Consola:** "Compilers can not run in the current OS: "+e.getMessage()

D.7. Dudas y Solución a Problemas Frecuentes

D.7.1. No se encuentran las opciones de la barra de herramientas LTL/CTL.

La barra de herramientas LTL/CTL del plug-in no siempre está disponible, esta se activa cuando un archivo de tipo tfa, tfq se encuentran abiertos y además no todas las opciones (iconos) de la barra están disponibles en ambos archivos, los casos son los siguientes:

- Cuando usted este construyendo el modelo del álgebra de tareas, la única opción disponible será New Logic Query.
- Si el archivo en primer plano es el consultor (tfq) se mostrarán todas las opciones excepto la anterior.
- En el caso de que el archivo en primer plano no sea tfq ni tfa, la barra de herramientas LTL/CTL no se mostrará.

D.7.2. No se muestra la vista de Resultados después de la ejecución de las consultas.

Muchas veces cuando las expresiones de consulta contienen errores o bien los archivos compiladores no han sido especificados adecuadamente el proceso de ejecución no podrá realizarse

y el sistema informará de estos errores.

Si por error ha cerrado la vista de resultados, puede abrirla a través de la opción *show view* del menú *window* de Eclipse con el nombre de *Task Algebra View Tree*.

D.7.3. ¿Puedo tener más de dos modelos algebraicos y consultores en el mismo proyecto?

Si, el proyecto puede tener n modelos los cuales se asocian a un consultor diferente y dependiendo del consultor ejecutado se muestran únicamente los resultados asociados.

D.7.4. ¿Qué sucede si las bibliotecas del proyecto no se configuran adecuadamente?

Este caso al igual que otros impedirán la ejecución de las consultas, sin embargo Eclipse le notificará en caso de que la configuración sea incorrecta, no esté presente o existan algunas cuestiones que impidan la ejecución de ciertas tareas.