DESARROLLO DE INFRAESTRUCTURA PARA FACILITAR EL ANÁLISIS Y LA VERIFICACIÓN DE DIAGRAMAS DE ACTIVIDAD IDENTIFICANDO LOS CAMINOS FUNCIONALES MODELADOS

TESIS

PARA OBTENER EL GRADO DE

MAESTRO EN SISTEMAS INTELIGENTES MULTIMEDIA

PRESENTA

ING. GUSTAVO ESPEJEL SALAZAR
ASESOR: MTRO. ERICK MANUEL LUGO ÁLVAREZ

QUERÉTARO, QUERÉTARO, NOVIEMBRE 2019.

DESARROLLO DE INFRAESTRUCTURA PARA FACILITAR EL ANÁLISIS Y LA VERIFICACIÓN DE DIAGRAMAS DE ACTIVIDAD IDENTIFICANDO LOS CAMINOS FUNCIONALES MODELADOS

CARTA DE LIBERACIÓN DEL ASESOR

DIEHL Controls

Querétaro, Querétaro. 19 de Agosto del 2019.

Mtro. Geovany González Carlos
Coordinador Académico de Posgrado
CIATEQ, A.C.

Los abajo firmantes, miembros del Comité Tutorial del Ingeniero Gustavo Espejel Salazar, una vez revisado su Proyecto Terminal de tesis/tesina, titulado "DESARROLLO DE INFRAESTRUCTURA PARA FACILITAR EL ANÁLISIS Y LA VERIFICACIÓN DE DIAGRAMAS DE ACTIVIDAD IDENTIFICANDO LOS CAMINOS FUNCIONALES MODELADOS" **autorizo** que el citado trabajo sea presentado por el alumno para su revisión, con el fin de alcanzar el grado de **Maestría**.

Sin otro particular por el momento, agradezco la atención prestada.

Maestro Erick Manuel Lugo Álvarez
Asesor Académico y en Planta

CARTA DE LIBERACIÓN DEL REVISOR

Querétaro, Querétaro, 05 de noviembre del 2019.

Dra. María Guadalupe Navarro Rojero
Directora de Posgrado
CIATEQ, A.C.

Por medio de la presente me dirijo a usted en calidad de Revisor del proyecto terminal del (la) alumno (a) **Gustavo Espejel Salazar**, cuyo título es:

"DESARROLLO DE INFRAESTRUCTURA PARA FACILITAR EL ANÁLISIS Y LA VERIFICACIÓN DE DIAGRAMAS DE ACTIVIDAD IDENTIFICANDO LOS CAMINOS FUNCIONALES MODELADOS"

Después de haberlo leído, corregido e intercambiado información con el (la) alumno(a), y realizado los cambios que le fueron sugeridos, puede ser autorizada su impresión, a fin de que se inicien los trámites correspondientes para su defensa.

Sin otro particular por el momento, y en espera de que mis sugerencias sean tomadas en cuenta en beneficio del estudiante y la Institución, agradezco la atención prestada.

Atentamente,

Firma 

Dr. Ismael Solís Moreno

RESUMEN

"Evolution is an essential property of real-world software" (1).

Los sistemas se mantienen en continua evolución, buscando satisfacer las necesidades del contexto en el que interactúan se vuelven más complejos y costosos. Por el contrario, dentro de las necesidades de un mercado demandante, las compañías buscan reducir los costos para mantener un precio competitivo. Es por esta razón que herramientas para automatizar procesos toman importancia. Al reducir los tiempos de desarrollo, es posible mantener la evolución de los sistemas y seguir con un precio competitivo. Herramientas actuales de automatización de procesos de verificación de software utilizan una semántica definida por el mismo creador de esta. En todos los proyectos, se terminan utilizando alternativas de diseño buscando adaptarse a esta semántica que no fue diseñada para un proyecto en específico. Esta carencia de adaptabilidad semántica produce que los ingenieros terminen trabajando para la herramienta y no viceversa. Es necesaria una herramienta capaz de manejar una semántica adaptable a las necesidades de cualquier proyecto; de esta manera, podemos garantizar que la herramienta trabajará para el proyecto y reducirá los costos de desarrollo. El concepto de análisis semántico lo encontramos dentro del mundo de la teoría de lenguajes y las Gramáticas Libres de Contexto (GLC). Definiendo la gramática adecuada, un diagrama de actividad se traduce a un árbol sintáctico donde cada rama representa un elemento semántico a procesar y cada hoja todos lo lexemas involucrados en el concepto. La plataforma se probó con el proyecto de TrueCourse, el sistema de gestión de vuelo más actual de General Electric (GE). La plataforma redujo en un 27% el tiempo de desarrollo, logrando adaptarse a las necesidades de cada componente y a la metodología de desarrollo basado en etapas de maduración tecnológica de la NASA (TRL por sus siglas en inglés). En proyectos donde las necesidades son cambiantes debido a su proceso de maduración, herramientas con la capacidad de adaptarse a las necesidades son importantes. Tecnologías sin esta capacidad no tienen un impacto significativo en proyectos complejos.

Palabras clave: Semántico, Adaptarse, Verificación, Evolución.

ABSTRACT

"Evolution is an essential property of real-world software" (1).

The systems are continuously evolving. Looking to satisfy the needs of their context, they get an increment on complexity and cost in every iteration. On the contrary, in the aggressive market needs, the companies need to reduce the cost to be competitive in every iteration. One of the solutions to this situation are the process automation tools. These kinds of tools allow to handle the software evolution and keep the cost competitive to the market. However, current automation tools for the software verification process are designed with a fabric defined semantic which happen to be generic and not 100% useful for a specific project. Consequently, engineers start working for the tool instead of having the tool working for the engineers. A tool with adaptable semantics feature is needed to be able to satisfy the needs of each project and get a maximum cost reduction. The concept of semantic analysis is found in the Languages Theory and Context Free Grammars worlds. Defining a proper grammar for UML activity diagrams with extra data we can get a syntax tree where each branch represents a semantic element to process; and each leaf will contain the lexemes involved in the concept. The framework was tested with the TrueCourse project, the newest General Electric's Flight Management System. The development time of some features was reduced in a 27% proving to be able to be adapted to the needs of each component, and to the development methodology based on the Technology Readiness Level (TRL).

In a project with a maturity process based on evolution, the needs are changing every day. Consequently, the adaptability capacity of the tools is important. Technologies without this capacity will not have a significant impact in complex project because an investment should be done to adapt the project to the tools.

Keywords: Semantic, Adapt, Verification, Evolution.

ÍNDICE DE CONTENIDO

CARTA DE LIBERACIÓN DEL ASESOR I
CARTA DE LIBERACIÓN DEL REVISOR II
RESUMEN III
ABSTRACT IV
ÍNDICE DE CONTENIDO V
ÍNDICE DE TABLAS VII
ÍNDICE DE FIGURAS VIII
GLOSARIO IX
1. INTRODUCCIÓN 1
1.1 ANTECEDENTES 1
1.1.1 EVOLUCIÓN DEL SISTEMA DE GESTIÓN DE VUELO 3
1.1.2 TRUECOURSE 4
1.1.3 REQUERIMIENTOS BASADOS EN ORACIONES 5
1.1.4 REQUERIMIENTOS BASADOS EN MODELOS 7
1.2 DEFINICIÓN DEL PROBLEMA 9
1.3 JUSTIFICACIÓN DEL PROBLEMA 9
1.4 OBJETIVOS 11
1.4.1 OBJETIVO GENERAL 12
1.4.2 OBJETIVOS ESPECÍFICOS 12
1.5 HIPÓTESIS 12
2 MARCO TEÓRICO 13
2.1 VERIFICACIÓN DE SOFTWARE 13
2.1.1 PROPÓSITO DE VERIFICAR EL SOFTWARE 13
2.1.2 CASOS DE PRUEBA 14
2.1.3 REQUERIMIENTOS 15
2.2 ASSERT 17
2.3 UML 19
2.4 DIAGRAMAS DE COMPORTAMIENTO 20
2.4.1 DIAGRAMAS DE ACTIVIDADES 22
2.4.2 MÉTODO DE CAJA GRIS 22
2.4.3 ANÁLISIS DE LOS ELEMENTOS SEMÁNTICOS 25
2.5 MÉTODO PROPUESTO – MÉTODO DE CAJA NEGRA 27

2.5.1 Generación automática de casos de prueba 27
2.5.2 Algoritmo de búsqueda a profundidad 27
3 PROCEDIMIENTO 29
3.1 Analizador léxico 29
3.2 Plataforma para semántica adaptable 30
3.3 Analizador sintáctico basado en un archivo de gramáticas 31
3.3.1 Semántica implementada 31
3.4 Generador de casos de prueba 34
3.4.1 Estructuras del árbol sintáctico 34
4 RESULTADOS 37
4.1 Diferencias con el método de Linzhana y Jiesong 37
4.2 Tiempos de ciclo 38
4.3 Cobertura funcional 40
CONCLUSIONES 42
APORTACIÓN DE LA TESIS 44
RECOMENDACIONES 45
REFERENCIAS BIBLIOGRÁFICAS 47
ANEXO A - ARQUITECTURA DE TCG_UML
ANEXO B - ALGORITMO
ANEXO C - EXPRESIONES REGULARES CONFIGURADAS
ANEXO D - GRAMÁTICA LIBRE DE CONTEXTO DE TCG_UML

ÍNDICE DE TABLAS

Tabla 1.1 – Leyes de Lehman (1) 2
Tabla 1.2 Porcentajes de Tiempo de TrueCourse – Semanas fiscales de la 24 a la 40. ... 12
Tabla 2.1. Características de un buen caso de prueba (29) 14
Tabla 2.2. Tipos de Requerimientos 15
Tabla 2.3. Características de un buen requerimiento (30) 16
Tabla 2.4. Métodos para generar casos de prueba con diagramas de comportamiento 21
Tabla 3.1. Pasos para Definición de Conceptos Semánticos 32
Tabla 3.2. Casos de Prueba 36
Tabla 4.1. Tabla comparativa entre método de caja gris y método de caja negra propuesto. 38
Tabla 4.2 – Cuadro Comparativo de los Tiempos de Ciclo. 39
Tabla 4.3 – Tabla Comparativa con ASSERT 40

ÍNDICE DE FIGURAS

Figura 1-1. TrueCourse FMS (11) 5
Figura 1-2 Desplazamiento a la derecha de la curva de oferta 10
Figura 2-1. Componentes de ATG de ASSERT (14) 18
Figura 2-2. Tipos de Diagramas UML 20
Figura 2-3. Ejemplo de Diagrama de Actividad con PlantUML 24
Figura 2-4. Elementos de un analizador Sintáctico (46) 25
Figura 2-5. Árbol Sintáctico y Diagrama de Actividad 26
Figura 3-1. Arquitectura del Analizador Léxico Configurable 30
Figura 3-2. Elementos del Análisis Léxico y Sintáctico. 31
Figura 3-3. "if-else" Árbol Sintáctico con Elementos Semánticos 33
Figura 3-4. "if-else" Diagrama de Actividad en PlantUML 34
Figura 3-5. Arquitectura del Árbol Sintáctico 35

GLOSARIO

A

Agile

Conjunto de metodologías para el desarrollo de proyectos que precisan de una especial rapidez y flexibilidad en su proceso. En muchas ocasiones son proyectos relacionados con el desarrollo de software. 9

ASSERT

Analysis of Semantic Specifications and Efficient generation of Requirements-based Tests 7, 19, 20, 21, 22, 40, 42, 44, 47

ATG

Generador de pruebas automáticas de ASSERT 20, 21

D

DO-178C

Consideraciones de software en sistemas de aviacion y certificación de equipo ...6, 7, 43

E

ENIAC

Integrador y calculador numérico electrónico 1

F

FMS

Sistema gestionador de vuelo 3, 4, 5, 6, 7, 21

G

GE

General Electric 4, 7, 42

GLC

Gramática Libre de Contexto 47

O

Open Source

Modelo de desarrollo de software basado en la colaboración abierta. Se enfoca más en beneficios prácticos que en cuestiones éticas. 9

P

PlantUML

Herramienta Open Source que permite generar diagramas UML a partir de texto plano 9, 10, 12, 13, 14, 27, 28, 31, 36, 37, 46, 47, 48

S

SADL

Lenguaje de diseño de semántica de aplicación 20, 47

SCADE

Ambiente integrado de diseño para aplicaciones críticas con manejo de requerimientos de tiempo, diseño basado en modelos, simulación, verifcación, generación de código certificable. 8, 9

SIMULINK

Entorno de programación visual para diseñar y simular sistemas basado en modelos. 8

T

TCG_UML

Infraestructura de semántica adaptable para generación de casos de prueba automática a partir de diagramas de actividad UML 32, 36, 40, 42, 43, 44, 45, 47, 48, 49, 50, 56, 58, 59

TRL

Nueve etapas de maduración tecnológica de la NASA ..... III, V, 13

TrueCourse

Sistema de Gestion De Vuelo de General Electric ..... III, V, 4, 5, 6, 7, 13, 43, 45

U

UML

Lenguaje de Modelado Unificado... V, 16, 17, 22, 23, 24, 25, 27, 28, 30, 32, 34, 40, 46, 47, 48, 49
Lenguaje Unificado para Modelado 8, 9

V

VOR
Radiofaro omnidireccional de alta frecuencia 3

1. INTRODUCCIÓN

Antes del descubrimiento del transistor en el siglo XX. Los sistemas principalmente mecánicos basándose en la máquina de vapor de James Watt (2). Eventualmente, durante el siglo XX, con la creación del transistor, sistemas electrónicos comenzaron a elaborarse. Es así como en 1946 cuando la ENIAC (Electronic Numerical Integrator and Calculator) es creada (3). A partir de este momento, la era de las comunicaciones dio comienzo (3). Sin embargo, en un principio el reto era el hardware (4). El costo de procesamiento y almacenamiento de datos era alto (4). Las compañías se enfocaban en la fabricación de hardware donde el software que corría era dedicado y su desarrollo era considerado un oficio de prueba y error (4). Gracias a las capacidades de procesamiento y almacenamiento del hardware moderno, la evolución de los sistemas dio un giro drástico donde el software, en lugar del hardware, se convirtió en el elemento principal del costo (4).

1.1 ANTECEDENTES

El desarrollo de software es un proceso donde la ingeniería de requerimientos es indispensable en el desarrollo de software (5). En teoría, asumiendo que se conocen todos los requisitos, es posible implementar un sistema que los satisfaga completamente (6); sin embargo, en la práctica el software interactúa con un ambiente complejo del mundo real, por lo que es imposible anticiparse a todas las posibles necesidades aun con metodologías de desarrollo agile (7). Además, es importante considerar que el ambiente de ejecución se ve alterado desde el momento en que se instala el software; al cambiar el ambiente, cambian las necesidades, cambia el software y este a su vez vuelve a alterar su mismo ambiente resultando en un proceso iterativo (8). Lehman describe el desarrollo de software como un proceso evolutivo sujeto a las leyes mostradas en la Tabla 0.1. Las cuales describen un cambio continúo donde se incrementa la complejidad en cada iteración (1).

Ley	Descripción
Cambio continuo	Un programa debe estar en constante cambio para adaptarse a las nuevas necesidades de sus usuarios. De lo contrario, se volverá menos satisfactorio.
Incremento de la complejidad	A medida que un programa evoluciona, la complejidad y deterioro de su estructura se incrementaran, a menos que se trabaje para reducirla o mantenerla.
Auto regulación	El proceso de la evolución del software es autorregulatorio, apegado a la distribución del producto y al proceso de los elementos producidos.
Conservación de la estabilidad organizacional	La cantidad de trabajo necesario en cada entrega es similar.
Conservación de la familiaridad	La cantidad de nuevo contenido en cada entrega de un software tiende a permanecer constante o decrece sobre el tiempo.
Crecimiento continuo	La funcionalidad de un sistema de software se incrementará con el tiempo buscando satisfacer a las necesidades cambiantes de los usuarios.
Deterioro de la calidad	La calidad de un sistema será percibida como disminuida a menos que su diseño sea cuidadosamente mantenido y adaptado a las nuevas reglas operacionales.
Sistema de retroalimentación	La evolución exitosa de un software demanda el reconocimiento de que el desarrollo de software es un proceso que requiere retroalimentación de múltiples ciclos, múltiples agentes y múltiples niveles.

Tabla 0.1 – Leyes de Lehman (1)

Como Michael Godfrey menciona en su artículo "On the Evolution of Lehman's Laws" Lehman identificó este proceso evolutivo como una tarea complicada y compleja (1). Sin embargo, Michael solo reconoce como vigentes a las leyes relacionadas al cambio continuo, al incremento de la complejidad, al crecimiento continuo, al deterioro de la calidad y al sistema de retroalimentación (1). La evolución de software se define entonces como un cambio continuo donde se incrementa la complejidad y funcionalidad con el riesgo de deteriorar la calidad si una retroalimentación y un manejo adecuado de los cambios basados en las necesidades del cliente y de su ambiente de ejecución no son llevados adecuadamente. Las leyes relacionadas a la autorregulación, la conservación de la estabilidad organizacional y la conservación de la familiaridad se encuentran en duda actualmente. (1)

1.1.1 Evolución del sistema de gestión de vuelo

El sistema de gestión de vuelo (FMS por sus siglas en inglés) es un ejemplo de cómo la complejidad de un sistema crece a medida que el sistema evolucionó a un nivel donde es capaz de facilitar a los pilotos operar aviones comerciales de una manera segura y eficiente. (9)

Sam Miller en su artículo "Contribution of Flight Systems to Performance-Based Navigation" proporciona un breve resumen de la evolución del sistema (9). En un inicio, la navegación del avión se lograba por medio de instrumentos que se enfocaban en monitorear la altura y ruta hacia el destino deseado (9). Además, los procedimientos se basaban en referencias visuales en tierra como lo son los faros de luz (9). El sistema era suficiente para volar un avión en condiciones favorables de clima; sin embargo, el negocio de la aviación evolucionó y surgieron nuevas necesidades de vuelo que generaron incertidumbres relacionadas a las reglas visuales de vuelo (9). El sistema dio su primer salto evolutivo y cambió de utilizar referencias visuales en tierra en la forma de faros de luz a utilizar sistemas de navegación en el avión que seguían ondas de radio generadas por faros de radio en tierra, iniciando la navegación por instrumentación (9). Los sistemas de radio navegación siguieron evolucionando desde los faros de radio hasta los sistemas de navegación de rango omni-direccional de alta frecuencia (VOR por sus siglas en inglés) (9). Sin embargo, la complejidad del sistema creció a tal grado que la cantidad de información de vuelo que los pilotos tenían que recolectar, interpretar e integrar de manera manual era tan compleja que existían grandes riesgos

de tener errores operacionales (9). Es por esta razón que en 1970s se crea la primera computadora de vuelo que integraba funciones que iban más allá de la navegación y las operaciones de eficiencia (9). Esto redujo el riesgo de errores de navegación en vuelo; pero, aun se tenía la necesidad de tener a un integrante de la tripulación enfocado a revisar la ejecución del plan de vuelo y navegación del avión, tres pilotos en total (9). En 1981, se determinó que volar con dos pilotos no era menos seguro que volar con tres aun cuando era evidente que la carga de actividades se incrementaba al volar con dos pilotos únicamente (9). Esto representó una oportunidad de reducción de costos para Boeing; solo había que garantizar que dos pilotos eran capaces de ejecutar todas las acciones necesarias para realizar un vuelo seguro (9). Es por estas razones de negocio que el FMS evolucionó nuevamente buscando reducir la carga de los pilotos (9). La computadora de vuelo progresó hasta convertirse en el corazón del plan de vuelo y funciones de navegación en vuelo y en tierra (9). Actualmente, el FMS se encuentra en una continua evolución, integrando mejoras de control, o participando con otros sistemas a bordo utilizados en los métodos de control de vuelo. (9)

Actualmente, un FMS tiene que ejecutar múltiples tareas complejas sin comprometer la seguridad (10). Sin, embargo, las compañías se enfrentan al reto de agregar nuevas funcionalidades que actualmente resultan costosas y poder seguir otorgando un precio competitivo en el mercado (10). Por necesidades de negocio el FMS tiene que dar un nuevo salto evolutivo. La primera compañía que lo logré se podrá situar como líder en el mercado.

1.1.2 TrueCourse

Basándose en más de 12,000 aviones instalados con los FMS de General Electric, GE Aviation esta llevado el FMS al siguiente nivel (11). TrueCourse está diseñado para soportar las necesidades actuales y futuras de aviones comerciales y militares, con una nueva plataforma y arquitectura que permite desarrollar de forma modular cada funcionalidad (10). Cada modificación solo afecta a los módulos relacionados, eliminando la necesidad de certificar nuevamente el sistema completo reduciendo tiempo y costo de desarrollo, implementación y pruebas de funcionalidades nuevas (11). Gracias a esto, TrueCourse es el FMS más avanzado en el mercado, fue desarrollado para satisfacer los estándares de espacio aéreo actuales y futuros; es por

esta razón, que sus beneficios no solo se limita a la reducción del costo de actualizaciones, sino también a tener caminos de vuelo optimizados con trayectorias 4D permitiendo un menor ruido de intrusión, menos emisiones y el mejor consumo de combustible posible (11). Buscando maximizar la seguridad y minimizar la carga de trabajo de los pilotos, TrueCourse es capaz de integrarse con todos los sistemas del avión; por ejemplo, es capaz de leer datos de performance de ambos motores y utilizarlos para calcular un control de vuelo optimizado y un perfil de eficiencia específico del avión, ningún otro FMS en el mercado puede ofrecer este nivel de adaptabilidad y eficiencia (11). Muchas funcionalidades base del FMS son agnósticas al avión, es por esta razón que muchos componentes de software se pueden implementar fácilmente a través de diferentes programas utilizando artefactos de certificación reutilizables; en otras palabras, TrueCourse permite actualizaciones rápidas a costos bajos (11).

Figura 0-1. TrueCourse FMS (11)

1.1.3 Requerimientos basados en oraciones

Un sistema como lo es el FMS, está regido por diferentes estándares que garantizan el un sistema eficiente y con la calidad necesaria como para transportar vidas humanas por los aires (12). Hablando específicamente del software embebido, el DO-178C lo

clasifica como un software de nivel B (12). Esto significa que un error en el sistema no concluye en un evento catastrófico donde se pongan en riesgo vidas humanas (13).

Actualmente, debido a su complejidad y nivel de impacto en el vuelo, se requieren de al menos 50,000 requerimientos para definir la funcionalidad de un FMS. Considerando que el diseño de cada requerimiento aún sigue siendo un proceso manual resulta bastante costoso el manejar la complejidad para lograr que cada uno de los requerimientos sea único, completo, claro, consistente, certificable y traceable (5). Esto representa un área de oportunidad para bajar aún más el costo del desarrollo de los artefactos de certificación de TrueCourse (14).

La evolución del software y el cambio continuo traen como consecuencia la necesidad de manejar el incremento de la complejidad de alguna forma para lograr un desarrollo eficiente (1). En una metodología de desarrollo donde el comportamiento del sistema se describe a base de palabras, resulta costoso el realizar un cambio continuo a medida que la complejidad se incrementa (5). Como consecuencia los requerimientos se vuelven más largos, más complejos y difíciles de entender (5). Debido a esto, se han desarrollado metodologías que buscan solucionar o manejar adecuadamente el problema de la complejidad (7).

Líderes en la industria de la aviación están conscientes de la necesidad de encontrar una forma de manejar la complejidad de los requerimientos que nos permitan tener la calidad necesaria para certificarse de acuerdo con el DO-178C bajando los costos de producción para seguir siendo competitivos en el mercado (14). Se han realizado esfuerzos fallidos para lograr este objetivo enfocados en requerimientos basados en oraciones como lo es ASSERT (Análisis de eSpecificaciones Semánticas y Generación eficiente de Pruebas basadas en Requerimientos) (14). GE dedicó un equipo a investigar y desarrollar un nuevo conjunto de herramientas para abordar los desafíos con el diseño, desarrollo y verificación de estos productos como el FMS en software (14). Los objetivos eran el desarrollar tecnología, procesos y herramientas que den como resultado un desarrollo de software y sistema más eficiente, medido por el costo y el tiempo de ciclo, y permitir nuevas capacidades como la autonomía y el internet industrial (14). Sin embargo, la herramienta resultó no estar lista para adaptarse a las necesidades de un proyecto como lo es TrueCourse (14).

1.1.4 Requerimientos basados en modelos

El desarrollo de software basado en modelos nos permite manejar la complejidad de los sistemas y su evolución de forma eficiente por medio de modelos (15). Sin embargo, para aprovechar todo su potencial, es necesario llegar a una estandarización gráfica que nos permita desarrollar herramientas de generación de código y generación pruebas de forma automática (16).

Claudia Pons menciona en su artículo "El proceso de desarrollo de software basado en modelos" cómo esta filosofía comenzó a tomar fuerza a finales de los años 70 (17). Sin embargo, el concepto de ingeniería de software basada en modelos fue introducido por Tom DeMarco en su libro "Structured Analysis and System Specification" (18). DeMarco hace evidente la importancia de crear un modelo del sistema antes de hacer un desarrollo de software (18). Este modelo de sistema se usa para: definir las necesidades del usuario, como un medio de comunicación y negociación, como un documento de referencia durante la corrección de errores, y como un documento de referencia durante la evolución del software (5).

"Se ha observado que la construcción de modelos es una técnica muy efectiva para detectar y resolver discrepancias entre los divergentes puntos de vista de los usuarios acerca de sus requerimientos, brindando así bases firmes para las siguientes etapas del proceso de desarrollo." (17)

Aunque, los modelos en SCADE y SIMULINK son plataformas de modelado gráfico que constan de herramientas de análisis y verificación para desarrollo de software; estas plataformas carecen de una estandarización formal de su lenguaje gráfico. Al no existir una estandarización, un modelo creado en una plataforma no puede reutilizase en otras plataformas. Un modelo creado en SCADE no puede ser reutilizado por SIMULINK y viceversa. A causa de esto, surge el Lenguaje Unificado para Modelado (UML por sus siglas en inglés) (19). Actualmente, el desarrollo de software busca utilizar esta filosofía a través de distintas herramientas cuya diferencia radica en los tipos de modelos utilizados y su representación gráfica. (17)

Considerando, que los requerimientos describen la funcionalidad del software, un diagrama de actividad UML es la opción adecuada para modelar la funcionalidad de un sistema (20). UML define varios subconjuntos de diagramas que puede modelar: los

diagramas estructurales, los de interacción y los diagramas de comportamiento. Los diagramas de actividad son un subconjunto de los diagramas de comportamiento que se utilizan para describir la funcionalidad del software (21).

Las herramientas existentes de análisis, generación de código y verificación de modelos UML obedecen a la semántica definida por el mismo fabricante, semántica que es incompatible entre las diferentes plataformas desarrolladas. En otras palabras, para lograr reutilizar modelos a través de diferentes proyectos es necesario estandarizar la semántica (17). El Grupo de Manejo de Objetos (OMG por sus siglas en inglés) realizó un excelente trabajo estandarizando la sintaxis de UML; sin embargo, aún carece de una semántica estandarizada. (17)

Dentro de la industria de la aviación, se sigue la metodología Agile que permite lograr una evolución del software de manera controlada (22). Sin embargo, a medida que el software se vuelve más complejo, el costo de modificar o añadir un requerimiento se incrementa en cada iteración (18). Los lenguajes de modelado gráfico como SCADE que cuentan con herramientas de generación de código y verificación automáticas, nos permiten reducir el costo de cada cambio (23). Sin embargo, el costo de la herramienta podría ser considerado alto en algunos proyectos; el utilizarla significaría incrementar el costo del producto en el mercado, reduciendo así su nivel de competitividad (24). Una solución al problema del costo son las herramientas "Open Source" como lo es PlantUML para el diseño de modelos (25). Sin embargo, PlantUML es considerada una herramienta de dibujo y no de modelado, debido a que carece de los beneficios que una plataforma de análisis y verificación proporciona (17). Una herramienta es considerada de bajo costo cuando su uso no impacta el costo del producto en el mercado; de esta forma, la competitividad del producto en el mercado se mantiene estable. Para que los proyectos de desarrollo de software logren reducir el costo de desarrollo y al mismo tiempo puedan manejar el incremento en la complejidad y el cambio continuo eficientemente, es necesario generar una plataforma que proporcione tanto herramientas de análisis y verificación, como la estandarización de la semántica de modelos representados por diagramas de actividad en PlantUML.

1.2 DEFINICIÓN DEL PROBLEMA

PlantUML se menciona en su página web como una herramienta gratuita que permite la creación de diagramas de actividad a partir de texto de forma rápida (25). Estos diagramas son una excelente opción para representar comportamiento de modelos, ya que sigue una notación gráfica estandarizada y fácil de interpretar; pero, no ofrece herramientas de análisis y verificación (25). En conclusión, para lograr reducir el tiempo de verificación, es necesario el desarrollo de una plataforma que cumpla con las siguientes capacidades:

- Análisis de los diagramas de actividad utilizando la sintaxis de PlantUML que los describe.
- Identificación de cada uno de los caminos funcionales basado en las variables de entrada y salidas de forma automática.
- Generación de los casos de prueba, buscando una cobertura funcional completa basado en el diagrama de actividad analizado.

De esta forma PlantUML en conjunto con la plataforma creada, formarían una herramienta gratuita completa de desarrollo de software basado en modelos.

1.3 JUSTIFICACIÓN DEL PROBLEMA

Andrea Schrage en su presentación "Mercado Competitivo", describe a la curva de coste marginal como una curva que sigue a la ley de los rendimientos marginales decrecientes (26). Esta ley establece que conforme aumenta la producción, más nos cuesta producir una cantidad adicional, es decir, mayor es el costo marginal, dando un comportamiento creciente a su curva. "En un mercado competitivo, la curva de oferta de una empresa coincide con su curva de coste marginal" (26)

El costo marginal, por lo general es absorbido por la compañía; ya que, en un mercado competitivo, la empresa pierde toda su demanda si aumenta su precio por encima del precio de mercado (27). La empresa se vuelve precio-aceptante y absorbe el costo de producir una unidad más. De esta forma reduce las perdidas en las utilidades de la compañía; sin embargo, aún existe una perdida (27).

Considerando que la curva de costo marginal coincide con la curva de oferta (27). Un desplazamiento a la derecha de la curva de la oferta, como se muestra en la Figura 0-2, provocaría el mismo efecto en la curva de costo marginal ya que se lograrían

producir más unidades al mismo costo. Por consecuencia, la empresa disminuye el costo marginal aceptado por producir la misma cantidad de unidades e incrementa sus ganancias. (26)

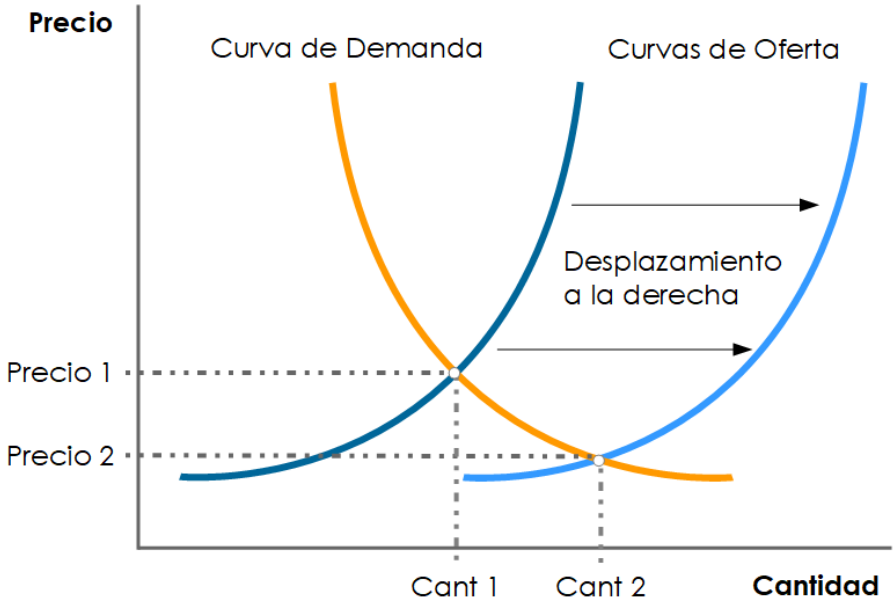

Figura 0-2 Desplazamiento a la derecha de la curva de oferta

Básicamente, un desplazamiento a la derecha de la curva de oferta y de costo marginal significa que se puede producir más a un mismo costo o incluso a un costo más bajo (26). Considerando esto, es posible identificar cuatro factores principales que permiten producir más al mismo costo, o incluso a un costo más bajo (26):

- Se da una mejora tecnológica.
- Disminuye el precio de producción.
- Disminuyen los impuestos sobre las ventas.
- Aumenta el número de empresas en el mercado.

Las variables de impuestos sobre las ventas y el número de empresas en el mercado están fuera del alcance de esta investigación. Sin embargo, el objetivo principal del proyecto impacta directamente en la tecnología y el costo de producción.

La relación entre el costo del software y el tiempo de desarrollo es directamente proporcional y se da por las horas de ingeniería invertidas. A mayor tiempo de desarrollo, mayor es el costo del software. El costo es inversamente proporcional a las

ganancias de la compañía. Con un costo elevado, la empresa debe absorber un costo marginal proporcionalmente alto disminuyendo sus ganancias. Con un costo menor, la empresa absorbe menos costo marginal incrementando sus ganancias (26).

Cabe mencionar, que el alcance de esta investigación no es el de reducir el costo marginal de una compañía, ni tampoco el de incrementar su competitividad en el mercado desplazando la curva de la oferta a la derecha. Lo que se busca es mejorar la tecnología de desarrollo. El aumento en las ganancias de una compañía es un efecto secundario esperado.

La plataforma de análisis y verificación para diagramas de actividad en PlantUML se traduce directamente como mejora tecnológica donde el costo de desarrollo se reduce. Esta mejora tecnológica es necesaria para lograr reducir el costo marginal que se incrementa en cada iteración durante el ciclo de evolución de un proyecto de software.

1.4 OBJETIVOS

Para poder ser competitivo en el mercado es necesario tener costos bajos conservando, y de ser posible mejorando, la calidad del producto; una forma de lograrlo es reduciendo los tiempos de desarrollo de un producto. La filosofía de desarrollo de software basado en modelos ayuda a reducir el tiempo de diseño e implementación al mejorar la comunicación entre los participantes; sin embargo, cuando no se cuentan con herramientas de análisis y verificación, el tiempo necesario para la verificación se puede incrementar de manera significativa. Lejos de reducir el tiempo de desarrollo, lo mantiene igual en el mejor de los casos, o lo incrementa en el peor de los casos.

La Tabla *0.2* muestra el porcentaje de tiempo invertido en cada una de las actividades de desarrollo del software de TrueCourse en su respectivo nivel de TRL (28). Estos datos se tomaron de una muestra de tiempos de ejecución que van de la semana fiscal 20 a la 40. Cómo podemos ver, la creación de casos de prueba representa un 23% del tiempo total de desarrollo.

Actividad	Porcentaje de Tiempo
Requerimientos de Alto Nivel (TRL5)	36%
Casos de Prueba (TRL4)	23%
Requerimientos de Bajo Nivel (TRL2)	1%
Código (TRL4)	12%
Procedimientos de Prueba (TRL4)	27%

Tabla 0.2 Porcentajes de tiempo de TrueCourse – semanas fiscales de la 24 a la 40.

1.4.1 Objetivo general

El objetivo general de este trabajo es el de reducir el tiempo de desarrollo de software basado en modelos mediante la automatización de procesos relacionados a las actividades de verificación de diagramas de actividad en PlantUML, para así lograr reducir el tiempo de desarrollo y tiempo de evolución sin comprometer la calidad del software desarrollado.

1.4.2 Objetivos específicos

Este proyecto busca reducir el tiempo de desarrollo, eliminando el tiempo necesario para el análisis del modelo, la generación de descripciones de prueba, y la generación de casos de prueba. Reduciendo, de acuerdo con la *Tabla 0.2*, en un 23% el tiempo que se asigna a estas actividades dentro del ciclo de desarrollo.

Además, a través de la plataforma de verificación se busca generar casos de prueba que permitan una cobertura funcional del diagrama de actividad analizado igual o superior al 80%.

Finalmente, se busca que la plataforma implementada tenga la capacidad de adaptarse a los elementos semánticos necesarios para cada proyecto en específico.

1.5 HIPÓTESIS

Al proporcionar herramientas de análisis de diagramas de actividad PlantUML, de identificación de caminos funcionales, y generación de casos de prueba que permitan una cobertura funcional superior al 80%; es posible reducir el tiempo de desarrollo de software, en proyectos donde los diagramas de actividad en PlantUML son utilizados para modelar los requerimientos del software, en un 23%.

2 MARCO TEÓRICO

En un sistema donde la mayor parte de la funcionalidad esta manejada por el software (4), es necesario entender los conceptos básicos que involucran el desarrollo y la verificación del software para así poder generar una plataforma que satisfaga las necesidades de cada proyecto. Esto debido a que es necesario garantizar que el software implementado es mantenible, confiable y eficiente; además de adecuado para las capacidades y experiencias de los usuarios (4).

2.1 VERIFICACIÓN DE SOFTWARE

No es posible garantizar que un software está completamente libre de errores después de ser sometido a las pruebas (29). No se puede probar la respuesta de un programa a cada posible valor de entrada; no se puede probar cada camino que el código puede tomar; no se puede encontrar cada error de diseño; no se pueden probar programas correctamente usando lógica (29). Consecuentemente, si no se puede probar un software por completo; no se puede verificar que trabaja sin errores. (29)

Si no se puede garantizar que un programa trabaja correctamente, es necesario utilizar estrategias que permitan seleccionar los casos de prueba que verifiquen los casos críticos del programa y reducir la probabilidad de encontrar un error (14).

2.1.1 Propósito de verificar el software

A mayor criticidad sea un software, más actividades de verificación son requeridas para tener la confianza necesaria de que se han identificado los errores críticos y corregidos. Es por esta razón, que la mayoría de los directores de programa asignan la mitad del presupuesto a estas actividades. (30)

Si bien el propósito básico de la verificación es el de tener la confianza necesaria de que el software funciona correctamente, es complicado definir el concepto de "confianza necesaria". La perspectiva de seguridad nos traduce este concepto como una actividad esencial que debe confirmar que el software trabaja de la forma en que se definió que debe trabajar. Básicamente, su propósito es el de verificar que el software realiza las funciones en que se describe su comportamiento en los requerimientos. (30) Consecuentemente, el concepto de verificación se define como la comprobación de que el software cumple con los requisitos funcionales y no funcionales de su especificación. (31) Es importante observar que el concepto no se

aterriza a garantizar que el software está libre de errores, se acota a comprobar que cumple con su especificación.

De acuerdo con Drake, existen dos formas de comprobar que el software es correcto, la inspección del software y las pruebas de software (31).

2.1.2 Casos de prueba

Shanmuga en su artículo "Test Case Generation From UML Models – A SURVEY" define a un *caso de prueba* como la serie de pasos secuenciales necesarios para ejecutar una prueba. Estos pasos están gobernados por entradas predefinidas que producen una salida esperada. Es posible generar los casos de prueba a partir de requerimientos, modelos de diseño o del código implementado para dar soporte a la verificación del software. (32)

Como Kaner menciona *"A test that reveals a problem is a success. A test that did not reveal a problem was a waste of time."* (29); un caso de prueba que revela un problema es un éxito, de lo contrario, es una pérdida de tiempo. Es por esta razón que Kaner define las características para un buen caso de prueba mostradas en la Tabla 2.1.

Criterio	Razonamiento
Probabilidad de encontrar un error	Se prueba para encontrar errores. Se trabaja desde la perspectiva de hacer fallar al programa.
No es redundante	Si dos casos de prueba persiguen el mismo error, no hay razón para correr ambos. Siempre y cuando los dos tengan la misma probabilidad de encontrar el error.
Es el mejor de su categoría	Si dos casos de prueba persiguen el mismo error, se debe correr únicamente el que tenga mayor probabilidad de encontrarlo.
No es simple pero tampoco complejo	Es posible combinar dos casos de prueba relativamente sencillos, siempre y cuando no se incremente de forma considerable la complejidad del caso de prueba resultante.

Tabla 2.1. Características de un buen caso de prueba (29)

El primer paso para generar casos de prueba de buena calidad que cumplan con las características de la Tabla 2.1, es el tener requerimientos y especificaciones de buena calidad que cumplan con las características de la Tabla 2.3 (33).

2.1.3 Requerimientos

Los requerimientos pueden dividirse en tres categorías, requerimientos de seguridad, requerimientos funcionales y no funcionales u otros como se muestra en la Tabla 2.2. Sin embargo, si consideramos que los requerimientos de seguridad son un subconjunto de los requerimientos funcionales, podemos encontrar únicamente dos categorías: funcionales y no funcionales. Cómo podemos ver en la Tabla 2.2, esto representan un 95% de los requerimientos totales de un sistema.

Tipo de Requerimiento	% en un sistema	Descripción
Seguridad	15%	Subconjunto de los requerimientos funcionales que contribuyen o afectan directamente a la seguridad del sistema.
Funcionales	80%	Definen la funcionalidad del sistema para obtener el comportamiento esperado.
Otros	5%	Requerimientos regulatorios y derivados. Los requerimientos derivados son el resultado de decisiones de diseño resultado de la madurez de la arquitectura del sistema.

Tabla 2.2. Tipos de requerimientos

De acuerdo con el estándar IEEE-STD-830-1998 (34): Especificaciones de los requisitos del software; los requerimientos funcionales, al igual que los diagramas de actividad UML, definen el comportamiento esperado del software. Tanto el diagrama de actividad como los requerimientos describen el proceso a seguir desde las entradas hasta las salidas: "los requisitos funcionales deben definir las acciones fundamentales que deben tener lugar en el software, aceptando y procesando las entradas, procesando y generando las salidas (35). ¨

Aunque los requisitos como los diagramas de actividad definen el comportamiento esperado del software, no es posible decir que un requisito es equivalente a un

diagrama de actividad. Debido a que los requerimientos son basados en oraciones que son interpretadas, debe seguir las características mencionadas en la Tabla 2.3 para lograr ser útiles. Todas las características se aplican a ambos, los diagramas de actividad como los requisitos menos una: la atomicidad. El hecho de que un requerimiento deba describir un comportamiento único es lo que lo desbanca de considerarse equivalente a un diagrama de actividad. Un diagrama de actividad define todos los caminos posibles de ejecución, por esta razón es que no puede ser atómico.

Características	Descripción
Atómico	Cada requerimiento debe describir un comportamiento único del sistema.
Completo	Cada requerimiento debe contener toda la información necesaria para definir la funcionalidad deseada del sistema.
Conciso	Debe ser fácil de leer y entender.
Consistente	Un requerimiento no debe contradecir o duplicar otro requerimiento.
Correcto	Cada requerimiento debe ser adecuado al sistema que se define.
Libre de Implementación	Cada requerimiento debe definir lo que es requerido, no como implementarlo
Necesario	Cada requerimiento debe definir un aspecto único del sistema, de tal forma que si es removido se encontraría una deficiencia en el sistema.
Traceable	Cada requerimiento debe tener un identificador único el cual debe ser traceable al diseño de bajo nivel, implementación y pruebas.
Verificable	Debe ser posible confirmar la implementación de cada requerimiento
Viable	Cada requerimiento debe ser implementable.

Tabla 2.3. Características de un buen requerimiento (30)

Regresando a las características de los requerimientos de la Tabla 2.3. El hecho de que todos sean libres de implementación nos permite definir que la verificación de nuestro

modelo descrito en un diagrama de actividad generado a partir de requerimientos libres de detalles de implementación traerá como resultado pruebas de caja negra del comportamiento esperado de nuestra unidad bajo prueba.

2.2 ASSERT

Cuando la captura formal de requerimientos se basa en oraciones, resulta complicado garantizar que cada requerimiento cumple con las características mencionadas en la Tabla 2.3, ya que estadísticamente hablando, el 35% de las fallas son introducidas en esta fase; sin embargo, solo el 1 % es encontrado en esta fase, el resto se encuentran en fases posteriores (14). Considerando que entre más tarde se encuentre el error, es más caro el costo de solucionarlo, la fase de captura de requerimientos

ASSERT (Analysis of Semantic Specifications and Efficient generation of Requirements-based Tests) es la solución de General Electric para encontrar y corregir errores introducidos por los requerimientos en la fase de captura; además de que permite automatizar la generación de casos de prueba (14). ASSERT se basa en ontologías para poder automatizar el razonamiento de conceptos y la relación entre los dominios del sistema y los requerimientos que están siendo analizados y es capaz de generar los casos de prueba aun antes de que el código sea implementado (14). Esto permite que, al momento de hacer la codificación, el desarrollador cuente con las pruebas necesarias para garantizar que la implementación cumple con los requerimientos solicitados (14).

ASSERT introduce un lenguaje para captura de requerimientos, muy similar al idioma inglés; aunque, es un lenguaje que podría considerarse seudo-ingles, es lo bastante formal para seguir los métodos de análisis formales (14). Pero, para poder analizar los requerimientos ASSERT genera un modelo de conceptos utilizando un lenguaje de ontología basado en la teoría de conjuntos y la lógica de primer orden. Esto se hizo de esta forma, ya que se ha hipotetizado que la teoría de conjuntos es lo que los humanos aplican cuando forman modelos mentales; una organización de los elementos ontológicos basada en la misma forma en la que los ingenieros lo harían (14).

Una vez generado el modelo, es posible escribir requerimientos entendibles ya que han sido escritos utilizando el modelo mental del experto y que son compatibles con la literatura del dominio (14). ASSERT utiliza el Lenguaje de Diseño de Semántica de

Aplicación (SADL por sus siglas en inglés) que permite el uso de los modelos de dominios definidos, además de permitir la captura de información adicional del dominio (14). Además, provee herramientas que ayudan a probar, depurar y mantener los modelos de dominios.

Una vez formados los requerimientos y los modelos de dominio, la generación de casos de prueba se realiza mediante la herramienta de Generación de Pruebas Automática (ATG por sus siglas en inglés) (14). La Figura 2-1 muestra los elementos que intervienen y el proceso para generar los casos de prueba.

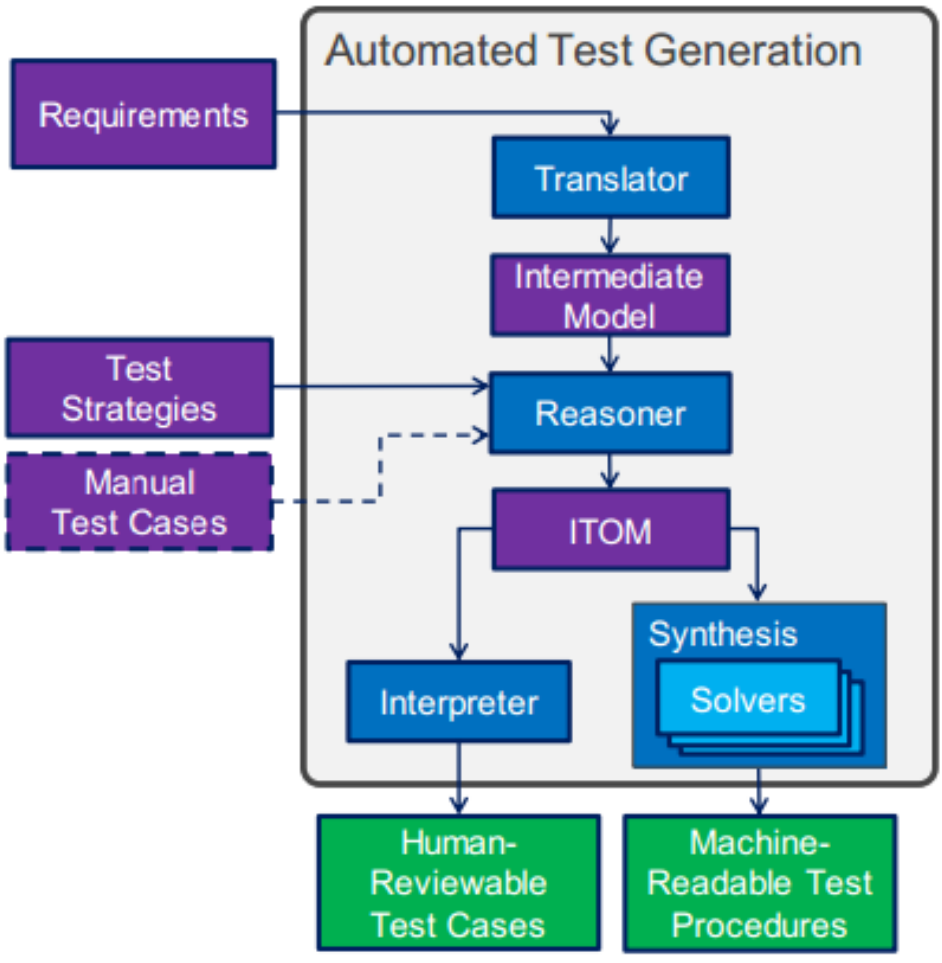

Figura 2-1. Componentes de ATG de ASSERT (14)

Basados en los modelos y en los requerimientos ATG identifica las estrategias a aplicar para escoger los casos de prueba: Cobertura lógica, Eventos, Clases Equivalentes, Listas, Tiempos, Supuestos y Ecuaciones (14).

ASSERT ha sido utilizado en proyectos como lo son el FMS, la red de conmutación de los aviónicos, el sistema de monitoreo de salud de los motores del avión y el controlador del sistema de potencia, todos estos proyectos mostraron reducción de costos y mejoras de sus tiempos de ciclo de desarrollo (14). A pesar de que permite encontrar y corregir errores desde una fase inicial del ciclo de desarrollo, el análisis de los conceptos y requerimientos para corregir los errores encontrados aun es complejo (36).

A pesar de que existen numerosas publicaciones de los beneficios de UML y la generación de casos de prueba a partir de estos diagramar, el equipo de ASSERT abandonó esta posibilidad ya que se ha encontrado que es necesaria información adicional a la contenida en los diagramas para lograr generar casos de prueba (14). ASSERT es capaz de encontrar los errores generados en la fase de captura de requerimientos; sin embargo, el análisis para corregirles sigue siendo complejo y consume tiempo (36).

El lenguaje de modelado simplifica la complejidad de los requerimientos, reduciendo así su tiempo de análisis para detectar y corregir errores (36). Sin embargo, los diagramas UML no tienen la información necesaria para generar los casos de prueba, elementos semánticos adicionales a la notación UML son necesarios para poder generar el análisis y así escoger los casos de prueba (36).

2.3 UML

UML es un lenguaje de modelado que comúnmente se confunde con un modelo. Un modelo en su forma básica se compone por un lenguaje y un proceso para modelar. Sin embargo, UML solo proporciona la notación gráfica del lenguaje. Al carecer del proceso, carece de los pasos a seguir para hacer los diseños. (37)

Considerando que UML es utilizado para mejorar la comunicación entre los grupos de trabajo, más que el proceso, es la notación que define al lenguaje la que sobresale en importancia. "Si usted desea analizar su diseño con alguien, lo que ambos necesitan comprender es el lenguaje de modelado, no el proceso que usted siguió para lograr tal diseño" (37)

En la versión de UML2 podemos encontrar a 13 diagramas concretos y varias categorías de diagramas abstractos, los cuales podemos clasificar como se muestra en la Figura 2-2.

<table>
<tr><td rowspan="11">Diagramas UML</td><td rowspan="6">Diagramas de Estructura</td><td>Diagrama de Clases</td><td></td></tr>
<tr><td>Diagrama de Estructuras Compuestas</td><td></td></tr>
<tr><td>Diagrama de Componente</td><td></td></tr>
<tr><td>Diagrama de Despliegue</td><td></td></tr>
<tr><td>Diagrama de Objeto</td><td></td></tr>
<tr><td>Diagrama de Paquete</td><td></td></tr>
<tr><td rowspan="5">Diagramas de Comportamiento</td><td>Diagrama de Actividad</td><td></td></tr>
<tr><td rowspan="4">Diagramas de Interacción</td><td>Diagrama de Secuencia</td></tr>
<tr><td>Diagrama de Comunicación</td></tr>
<tr><td>Diagrama Resumen de Interacción</td></tr>
<tr><td>Diagrama de Tiempo</td></tr>
<tr><td>Diagrama de Casos de Uso</td><td></td></tr>
<tr><td>Diagrama de Máquina de Estados</td><td></td></tr>
</table>

Figura 2-2. Tipos de diagramas UML

La notación de UML es utilizada para especificar numerosos aspectos de un sistema. Es por esta razón, que se vuelve una fuente importante de información para la generación de casos de prueba. Mohapatra menciona que de explotarse adecuadamente el costo de generar casos de prueba se reduce significativamente. (38) En la sección 2.4 se menciona la forma en que cada uno de los diagramas UML contribuye a generar los casos de prueba. Sin embargo, son los diagramas de comportamiento los que, al igual que los requerimientos, describen la forma en que se comportara el sistema. Es por esta razón que la generación automática de casos de prueba se basa mayormente en los diagramas de comportamiento. (38)

2.4 DIAGRAMAS DE COMPORTAMIENTO

Los diagramas de comportamiento se emplean para visualizar, especificar, construir y documentar los aspectos dinámicos de un sistema. En la Figura 2-2 se muestran todos los tipos de diagramas que se clasifican como diagramas de comportamiento.

La generación de casos de prueba a partir de diagramas de comportamiento UML es un tema retador que ha tomado fuerza entre los investigadores. De acuerdo con

Mohapatra, tradicionalmente se han realizado distintos esfuerzos utilizando técnicas heurísticas como cobertura de declaraciones, cobertura de ramas, cobertura de secuencia de mensajes, etc. (38) Sin embargo, son los diagramas de secuencia y los diagramas de actividad los que contienen la información necesaria para la generación de casos de prueba.

Los diagramas de secuencia describen la forma en la que se intercambiaran los mensajes mientras se realiza la ejecución de un caso de uso. Sin embargo, los diagramas de actividad se enfocan en el flujo de control, así como a la relación entre los objetos basados en su actividad. Mohapatra define una forma de transformar los diagramas de secuencia y actividad en un grafo intermedio. Una vez generado el grafo, es posible generar diferentes secuencias de prueba, que representan diferentes escenarios. De las secuencias de prueba se generar los casos de prueba. En la Tabla 2.4 se explica el método diseñado por Mohapatra. (38)

Autor	Diagramas de comportamiento	Descripción
Trung D. Trong (39)	Clases Secuencia Actividad	Generar casos de prueba que cubran cada borde de un diagrama de actividad. Sin embargo, el método no se define explícitamente
Pilskalns (40)	Clases Secuencia	Del diagrama de secuencias generan un grafo acíclico dirigido de objetos-métodos que es utilizado para generar caminos de prueba y sus condiciones correspondientes.
Ghose (41)	Clases Secuencia	Se genera un grafo de variable asignada utilizando la información de ambos diagramas.
Linzhang (42)	Actividad	Método de caja gris donde los casos de prueba son generados directamente del diagrama de actividad.
Mohapatra (38)	Secuencia Actividad	Los diagramas se convierten una representación intermedia llamada Grafo de Modelo de Flujo, el cuál es una combinación entre el diagrama de secuencia y actividad

Tabla 2.4. Métodos para generar casos de prueba con diagramas de comportamiento

El método de Linzhana basado únicamente en diagramas de actividad explota la ventaja de las pruebas de caja negra al analizar el comportamiento externo, además de ejecutar pruebas de caja blanca analizando el comportamiento interno esperado.

2.4.1 Diagramas de actividades

En UML un diagrama de actividad se usa para mostrar una secuencia de pasos que describe el flujo de trabajo desde el punto de inicio hasta el punto final, detallando muchas de las rutas de decisiones que existen en el progreso de eventos contenidos en la actividad. Es gracias al detalle de las rutas de decisiones que un diagrama de actividad contiene la información necesaria para generar los casos de prueba que ejerciten cada camino funcional basado en los valores que detonan un cambio funcional en las salidas el sistema. Sin embargo, son consideradas pruebas de caja gris, ya que el nivel de detalle siguiere los caminos implementados dentro del código. (43)

2.4.2 Método de caja gris

El método de caja negra utiliza los requerimientos del sistema para seleccionar los casos de prueba que verifican que el sistema se implementó correctamente. Esto se hace sin la necesidad de revisar la forma en que el sistema fue implementado. El método de caja blanca se basa en la implementación (código) para generar los casos de prueba que verifican que el código funciona correctamente. (42)

Las pruebas de caja gris combinan el método de caja blanca y el método de caja negra. Cubre toda la cobertura lógica de la caja blanca, además de encontrar todos los caminos posibles del modelo de diseño que describen el comportamiento esperado de una operación.

Un diagrama de actividad contiene todo el comportamiento esperado de una operación. Si existe un error en la implementación de la operación modelada, debe estar identificada en un camino de ejecución. Al identificarlo, es posible hacer las correcciones necesarias tanto en el modelo como en la implementación. Es en los caminos de ejecución donde se encuentran los casos de prueba. (42)

Para poder obtener los casos de prueba, por lo general, en la mayoría de los esfuerzos como se menciona en la Tabla 2.4 para obtener los caminos de ejecución y los casos de prueba, el diagrama de actividad de la operación modelada se transforma en un grafo que es recorrido de forma transversal para poder tener cobertura.

Por el contrario, Linzhang y Jiesong propusieron un método para obtener la cobertura de los caminos de ejecución directamente del diagrama de actividad. Lo que mencionan es que tanto el grafo como el diagrama de actividad son equivalentes y la conversión no es necesaria. Un camino de ejecución de un diagrama de actividad es un posible riesgo de error en la ejecución del programa que implementa la operación. (42)

Aunque Linzhana y Jiesong mencionan que su método se aplica en pruebas de caja gris; este método puede ser utilizado para pruebas de caja negra únicamente para analizar requerimientos donde es necesario verificar el comportamiento externo esperado. En la sección 2.5 se describe el método propuesto de caja negra partiendo de requerimientos.

Las ventajas de esta propuesta son las siguientes:

- Toma ventaja de pruebas de caja negra para verificar el comportamiento esperado.
- Toma ventaja de pruebas de caja blanca para cubrir la estructura interna del diagrama de actividad de la unidad del sistema en prueba.
- Es útil para encontrar defectos de implementación
- Permite encontrar sobre implementación y sub-implementación
- Los casos de prueba pueden ser generados en paralelo con la implementación de código ya que las pruebas de diseño pueden comenzar una vez que la fase de diseño es terminada.
- Permite a los ingenieros de pruebas de utilizar mejor los recursos.
- Se pueden encontrar errores de diseño al realizar el análisis del modelo
- Los errores de diseño se corrigen en etapas tempranas del desarrollo.
- Previene al ingeniero de pruebas de empaparse con detalles de implementación de forma prematura.
- Es la base para pruebas a partir de modelos.

En conclusión, es posible automatizar la generación de casos de prueba de las especificaciones funcionales de un sistema utilizando la información contenida en un diagrama de actividad. Sin embargo, el tiempo ahorrado en la verificación del software, se puede gastar al tener que diseñar los elementos gráficos de UML; es aquí

donde PlantUML toma relevancia, al ser un archivo de texto su entrada para la generación automática del diagrama, este riesgo se mitiga.

2.4.2.1 Diagramas de Actividad con PlantUML

PlantUML es una herramienta cuyo objetivo es la creación de diagramas UML de manera rápida y eficiente. Los diagramas UML se crean y modifican utilizando texto en lugar de una interfaz gráfica donde se tenga manipulación de los elementos gráficos. De esta forma la creación y modificación de diagramas es fácil y rápida. (44)

Dentro de los diagramas UML que se pueden crear utilizando entradas de texto tenemos los diagramas de actividad. En la Figura 2-3 se muestra la sintaxis utilizada para cada elemento necesario en un diagrama de actividad. Como podemos ver, a través de una sintaxis relativamente sencilla, es posible crear diagramas de actividad completos que faciliten el análisis en la etapa de diseño y que nos permitan la automatización de la generación de casos de prueba.

```
@startuml

start

if (inputOne > Y) then (TRUE)
        if (inputTwo > Z) then (TRUE)
                : B = A;
        else (FALSE)
                :B = D;
        endif
else (FALSE)
        :B = C;
endif

stop

@enduml
```

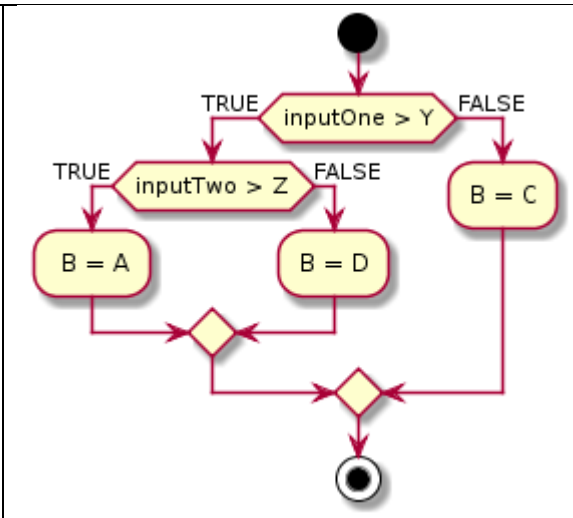

Figura 2-3. Ejemplo de diagrama de actividad con PlantUML

La Figura 2-3 muestra un diagrama de actividad definido utilizando la sintaxis de PlantUML. Como podemos ver, existen dos nodos de decisión que se combinan para generar tres caminos funcionales. El primer nodo de decisión se basa en el valor de la *entrada uno* y el segundo nodo se basa en el valor de la *entrada dos.* Dependiendo del camino funcional, es el valor que la *salida B* toma al salir del flujo.

Además, PlantUML permite la adaptabilidad de UML a las necesidades de cada proyecto. Esto es gracias a que cada palabra reservada es utilizada para definir el flujo

de la ejecución del software, permitiendo que cada proyecto defina los acuerdos necesarios de la información que va en forma de texto dentro de cada uno de los elementos del diagrama. Dentro de este proyecto llamaremos a estos elementos, *elementos semánticos.*

2.4.2.2 Elemento Semánticos

El diccionario de la lengua española define la palabra *semántica* como la disciplina que estudia el significado de las unidades lingüísticas y de sus combinaciones. (45)

Considerando que el texto definido dentro de cada elemento UML mostrado en la Figura 2-3 se basa en acuerdos que surgen de la necesidad de un proyecto de estandarizar la forma en que la información se ordena para dar un significado intencionado, podemos considerar a cada elemento de texto como unidad lingüística o lexema que se combina para satisfacer las necesidades semánticas de un proyecto. Es por esto, que a cada una de estas combinaciones las llamaremos *elemento semántico*. Elementos que se deben de analizar para poder generar los casos de prueba que verifiquen el funcionamiento definido por el diagrama de actividad.

2.4.3 Análisis de los elementos semánticos

Para poder leer el texto mostrado en la Figura 2-3 y generar los casos de prueba es necesario utilizar un analizador sintáctico (parser) con capacidades de análisis semántico como se muestra en la Figura 2-4.

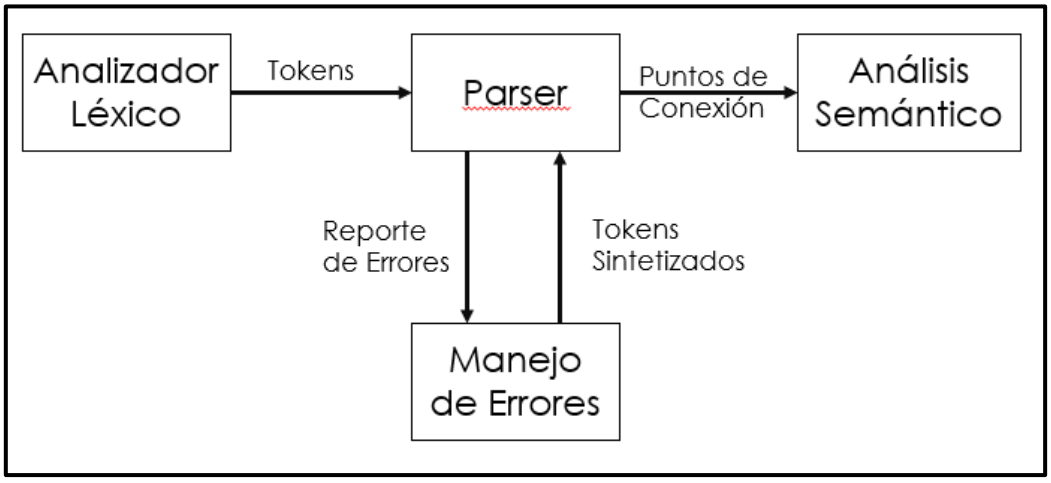

Figura 2-4. Elementos de un analizador sintáctico (46)

El objetivo de un parser es el de determinar las palabras lexemas que son relevantes y al mismo tiempo clasificarlas en tokens e identificar la semántica analizando del orden

en que se encuentran los tokens. Como resultado se obtiene una estructura conocida como árbol sintáctico. (46)

2.4.3.1 Analizador Léxico

Como Inés lo menciona en su presentación *Análisis Léxico* un analizador léxico se encarga principalmente de agrupar los lexemas en tokens. (47) Para esto es necesario definir categorías léxicas para lograr agrupar los lexemas. Cada categoría léxica corresponde a un patrón de símbolos que puede ser expresado mediante expresiones regulares.

El orden de los tokens es revisado en base a una gramática libre de contexto para poder obtener los elementos semánticos.

2.4.3.2 Analizador Sintáctico

El analizador sintáctico revisa el orden entre los tokens para identificar sintaxis específicas de cada elemento semántico. Reordena el archivo de entrada en un árbol de conceptos semánticos conocido como árbol sintáctico. Para este paso es importante tener una gramática libre de contexto donde cada símbolo no terminal defina cada uno de los elementos sintácticos deseados para cada proyecto en específico.

2.4.3.3 Analizador Semántico

En la Figura 2-5 se presenta como es que en un árbol de análisis sintáctico se pueden identificar todos los caminos funcionales definidos por un diagrama de actividad UML.

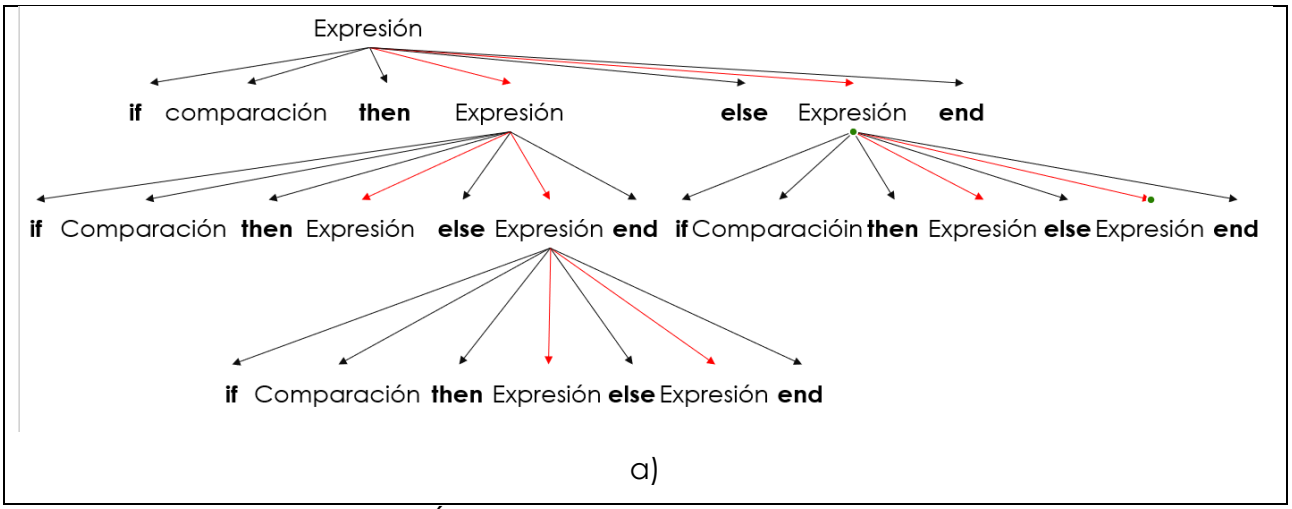

Figura 2-5. Árbol sintáctico y diagrama de actividad

En la Figura 2-5 (a) podemos ver en rojo 5 caminos los cuales son los que se encuentran en la Figura 2-5 (b). Es claro que el árbol sintáctico presenta tokens que parecieran no ser relevantes al análisis de los caminos funcionales de un diagrama de actividad. Sin embargo, en el token de comparación, se encuentra la ecuación booleana a evaluar para escoger el camino a seguir. Es importante mencionar que esta ecuación booleana debe estar definida únicamente con estados de las variables de entrada del sistema bajo prueba. Estos valores de entrada los podemos encontrar en el conjunto de requerimientos utilizados para crear el diagrama de actividad.

2.5 MÉTODO PROPUESTO – MÉTODO DE CAJA NEGRA

El método propuesto busca incorporar las ventajas del desarrollo basado en modelos a las estrategias convencionales de desarrollo donde los requerimientos aún se definen con palabras y oraciones.

Los pasos del método se enfocan a la verificación de un modelo definido mediante un diagrama de actividad. Sin embargo, es necesario generar el diagrama a partir de los requerimientos como lo menciona la sección 2.1.3.

Una vez generado el diagrama, es necesario automatizar el procedimiento de generación de casos de prueba, esto se describe en la sección 2.5.1

2.5.1 Generación automática de casos de prueba

Para la generación de casos de prueba de forma automática se consideran los siguientes pasos:

1.- Diseñar el diagrama de actividad en PlantUML basado en los requerimientos relacionados.

2.- Generar un árbol de análisis sintáctico (grafo acíclico dirigido).

3.- Recorrer el grafo utilizando el algoritmo DFS para analizar cada posible camino dentro del árbol.

2.5.2 Algoritmo de búsqueda a profundidad

La búsqueda a profundidad permite en un grafo acíclico dirigido o un árbol el recorrer todo el árbol rama por rama. (48) Considerando que un árbol sintáctico es equivalente a un diagrama de actividad y siguiendo la lógica del método de caja gris descrito en la sección 2.4.2. De cada rama del árbol podemos obtener la secuencia de un caso

de prueba y generar el caso de prueba para cada camino que se encuentre. Podríamos decir que no es necesario generar el grafo intermedio; sin embargo, como parte del proceso de análisis del texto que utiliza PlantUML para la creación de diagramas de actividad, es necesario generar un árbol de análisis sintáctico que es equivalente al grafo intermedio y al diagrama de actividad. El árbol de análisis sintáctico garantiza tener toda la información para diseñar un caso de prueba por rama logrando una cobertura de cada requerimiento utilizado para la generación del diagrama utilizando pruebas de caja negra.

3 PROCEDIMIENTO

Como se menciona en la sección 2.2 UML por sí solo no contiene toda la información para poder generar casos de prueba; sin embargo, lejos de ser una limitante, es un factor de adaptabilidad a la semántica de los proyectos. Gracias a que información adicional puede ser contenida en los diagramas UML, es posible agregar sintaxis para elementos semánticos adicionales; y mejor aún, se pueden configurar elementos sintácticos para la semántica necesitada y distinta en cada proyecto. De aquí salé la plataforma TCG_UML. Una plataforma capaz de explotar esta bondad de UML y configurar sintaxis para cada elemento semántico necesitado en cada proyecto. Para poder lograrlo, fue necesario aplicar los conceptos definidos en la sección 2.4.3. La Figura 0-1 muestra la arquitectura general de la plataforma implementada.

3.1 ANALIZADOR LÉXICO

Como se menciona en la sección 2.4.3.1, el objetivo principal de un analizador léxico es el de convertir el archivo de entrada en una lista de tokens donde a base de expresiones regulares es posible clasificar los lexemas de entrada en categorías léxicas para su agrupación.

Uno de los objetivos específicos de este proyecto, es la capacidad de adaptarse a las necesidades semánticas del mismo. Para esto se implementó un archivo de entrada que recibe las expresiones regulares que definen el patrón de cada categoría léxica. La plataforma se implementó con la capacidad de clasificar los lexemas de cualquier archivo de entrada basado en los patrones configurados en este archivo de configuración.

Los patrones definidos para el proyecto donde se utilizó la plataforma para generar casos de prueba de forma aleatoria se describen en la Tabla 0.1.

El analizador léxico genérico es parte esencial de la plataforma, ya que permite encontrar errores léxicos dentro de cada lexema antes de poder agruparlos para generar la lista de tokens.

Es importante señalar que no se implementó la ejecución en base a la expresión regular, este algoritmo sigue un orden de complejidad computacional exponencial $O(2^n)$ por el carácter recursivo en su ejecución. Para reducir los tiempos de ejecución de la plataforma, se genera un autómata finito determinístico. Esto permite que cada

análisis de cada cadena se ejecute con un algoritmo O(n). La Figura 3-1 muestra el diagrama de componentes de su implementación.

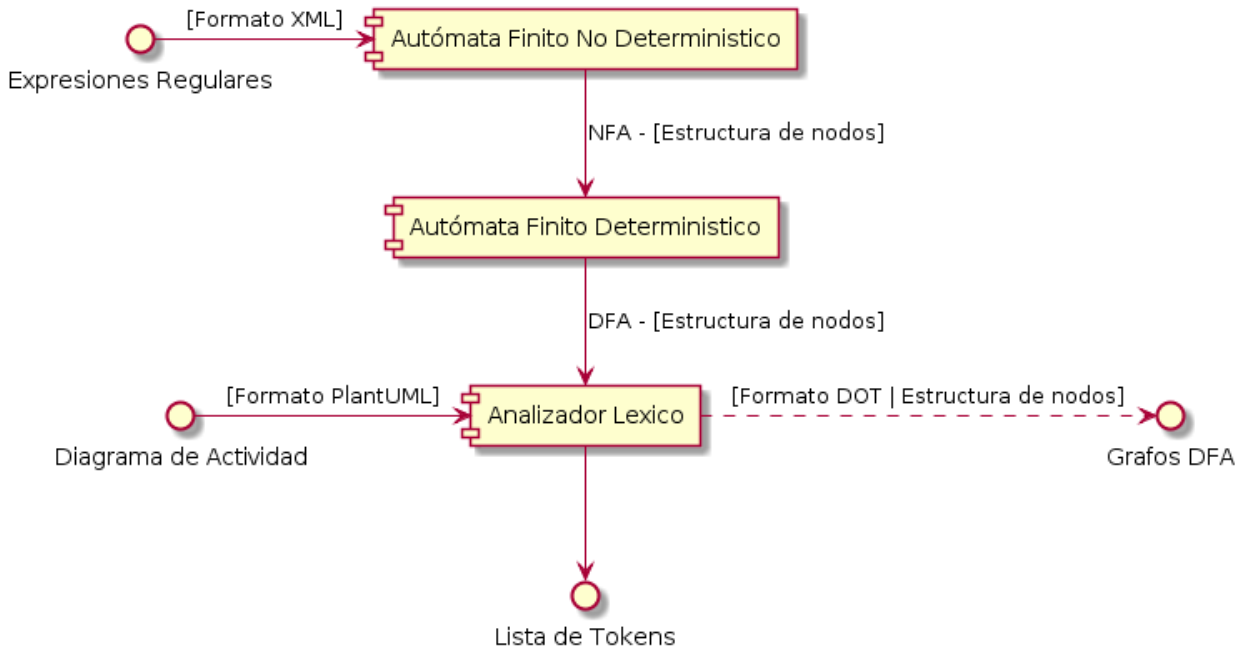

Figura 3-1. Arquitectura del analizador léxico configurable

3.2 PLATAFORMA PARA SEMÁNTICA ADAPTABLE

Se implementaron cuatro procesos principales:

1. Se procesan los archivos de configuración XML para identificar los elementos semánticos a analizar.
2. Análisis léxico del archivo recibido a la entrada basado en los patrones de cada categoría léxica del archivo de configuración y generación de la lista de tokens.
3. Análisis sintáctico de la lista de tokens, identificación y separación de los elementos semánticos para generar el árbol sintáctico.
4. Procesamiento del árbol sintáctico generado para identificar los caminos funcionales y generar los casos de prueba.

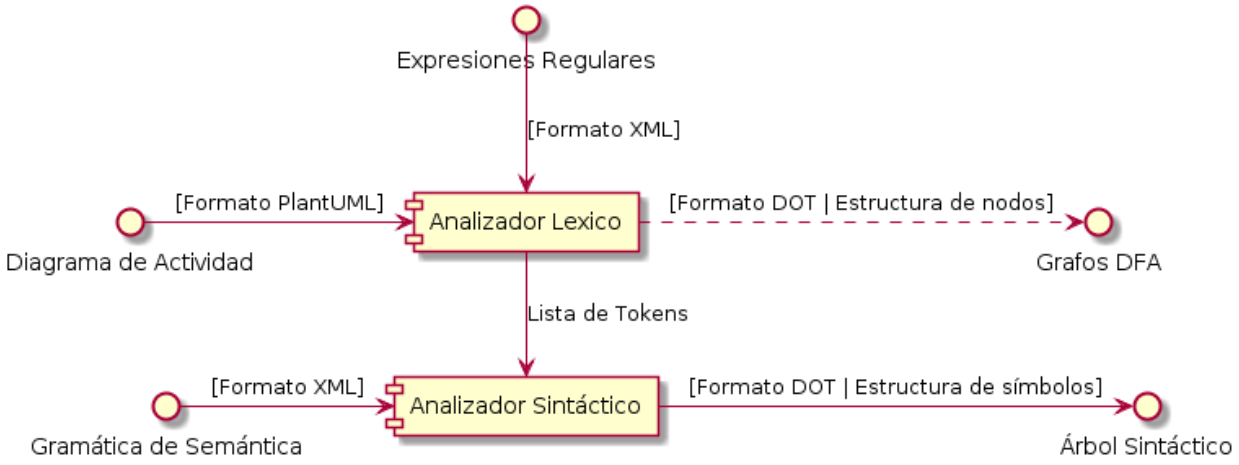

Figura 3-2. Elementos del análisis léxico y sintáctico.

Sin embargo; el paso número uno, el procesamiento de los archivos de configuración es lo que otorga a la plataforma su característica de adaptarse a cualquier proyecto. Para efectos de este proyecto nos enfocamos en la gramática necesaria para generar casos de prueba a partir de un diagrama de actividad en la sintaxis de plantUML.

3.3 ANALIZADOR SINTÁCTICO BASADO EN UN ARCHIVO DE GRAMÁTICAS

La gramática mostrada en la Tabla 0.1 se diseñó buscando separar elementos semánticos importantes expresados en los símbolos no terminales. Debido a que el elemento principal en la definición de caminos funcionales es la oración "if-else" la gramática permite reconocer cada camino y sub-caminos dentro de casa estructura "if" de manera recursiva. Dando como resultado un árbol sintáctico como se muestra en la Figura 3-3.

3.3.1 Semántica implementada

Para poder implementar la gramática mostrada en la Tabla 0.1 fue necesario definir los elementos semánticos que se buscan extraer del archivo de entrada. Para lograr esto es necesario definir los elementos esperados a la salida. La Figura 3-4 muestra el diagrama UML que estos casos de prueba verificarían. El paso 1 de la Tabla 3.1 muestra la salida esperada, dos casos de prueba para dos caminos funcionales. A través de cada uno de los pasos de la tabla, se puede entender cómo se define cada uno de los elementos semánticos.

Paso	Concepto Semántico	Descripción
1.	VERIFY B is set to A When inputOne > Y is TRUE. VERIFY B is set to C When inputOne > Y is FALSE.	Dos casos de prueba necesarios para verificar ambos caminos funcionales del archivo de entrada.
2.	VERIFY OUTPUT_SETTING When inputOne > Y is TRUE. VERIFY OUTPUT_SETTING When inputOne > Y is FALSE.	OUTPUT_SETTING: Elemento semántico que envuelve al concepto de asignación de las variables de salida.
3.	VERIFY OUTPUT_SETTING When CONDITION is TRUE. VERIFY OUTPUT_SETTING When CONDITION is FALSE.	CONDITION: Elemento semántico que representa el concepto de la condición necesaria para la asignación de un valor a la salida.
4.	VERIFY OUTPUT_SETTING When CONDITION is CONDITION_VALUE. VERIFY OUTPUT_SETTING When CONDITION is CONDITION_VALUE.	CONDITION_VALUE: Elemento semántico que envuelve al concepto de valor esperado en la evaluación de la condición.
5.	IF. ELSE.	Dos caminos funcionales identificados que comprenden a diferentes valores de condición para la condición evaluada. IF: Camino funcional principal ELSE: Camino funcional cuando la condición del camino principal no es dada. En este caso, al ser una condición booleana donde solo se pueden obtener dos valores como resultado de la evaluación se considera una estructure equivalente a un *if-else*.
6.	IF_PATH	IF_PAHT: Elemento semántico que representa a la estructura *if-else*.
7.	PATHS	PATHS: Elemento semántico que representa caminos funcionales.

Tabla 3.1. Pasos para definición de conceptos semánticos

La Figura 3-4 y la Figura 3-3 son elementos generados por el mismo archivo de entrada. La Figura 3-4 es generada por PlantUML y la Figura 3-3 es generada por TCG_UML. Como podemos observar, de la Figura 3-4 podemos obtener la cantidad de caminos funcionales existentes, dos en este caso; sin embargo, no es posible identificar información detallada sobre los elementos que forman la condición de selección del camino funcional, o la asignación de alguna variable de salida. Es posible inferir cada uno de los elementos mencionados anteriormente; sin embargo, nada nos separa o identifica dichos elementos. La gramática definida en la Tabla 0.1 nos permite generar un árbol sintáctico, como se muestra en la Figura 3-3, que separa y clasifica cada uno de los elementos necesitados para generar los casos de prueba.

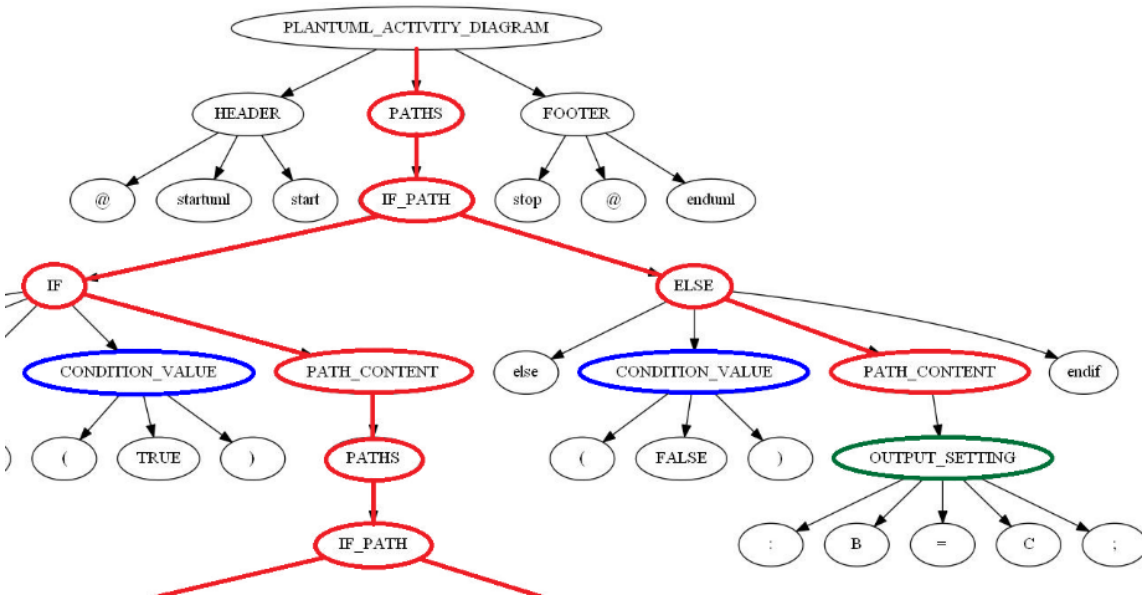

Figura 3-3. "if-else" Árbol Sintáctico con Elementos Semánticos

La Figura 3-3 nos permite identificar no solo la existencia de una estructura condicional "if" con dos caminos funcionales, sino que también, nos permite identificar los siguientes elementos:

- Elementos que conforman a la condición de selección del camino funcional.
- El comparador utilizado en la condición.
- El valor esperado del resultado de la condición.
- El contenido a ejecutar en cada camino funcional identificado.
- La asignación de salida de acuerdo con el camino funcional seleccionado.

De ser requerido por el proyecto, se pueden agregar producciones a la gramática que nos permitan identificar elementos semánticos adicionales.

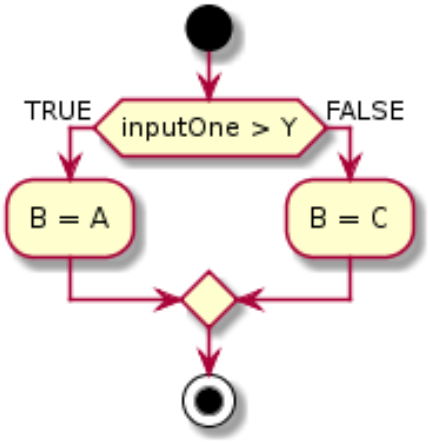

Figura 3-4. "if-else" Diagrama de actividad en PlantUML

3.4 GENERADOR DE CASOS DE PRUEBA

El analizador léxico y sintáctico tienen la capacidad adaptarse a la semántica acordada por cada proyecto; sin embargo, es necesario implementar el procesamiento de cada elemento semántico dentro del árbol. Es importante señalar que la herramienta permite el procesamiento de cualquier archivo de entrada basado en una gramática configurable, sin importar el objetivo final de la semántica implementada. Para fines de este proyecto, la generación de casos de prueba es el objetivo final de la semántica mencionada en la sección 3.3.1. En otras palabras, una vez que el árbol sintáctico ha sido creado, una segunda herramienta que lo procese de acuerdo con los objetivos semánticos específicos es necesaria. El árbol se recorre siguiendo el algoritmo de búsqueda a profundidad mencionado en la sección 2.5.2. Sin embargo, es necesario programar funciones específicas para el procesamiento de cada nodo.

3.4.1 Estructuras del árbol sintáctico

El árbol de análisis sintáctico que se entrega a la herramienta encargada de procesar la semántica sé compone de tres estructuras cuyo diagrama de clases se muestra en la Figura 3-5.

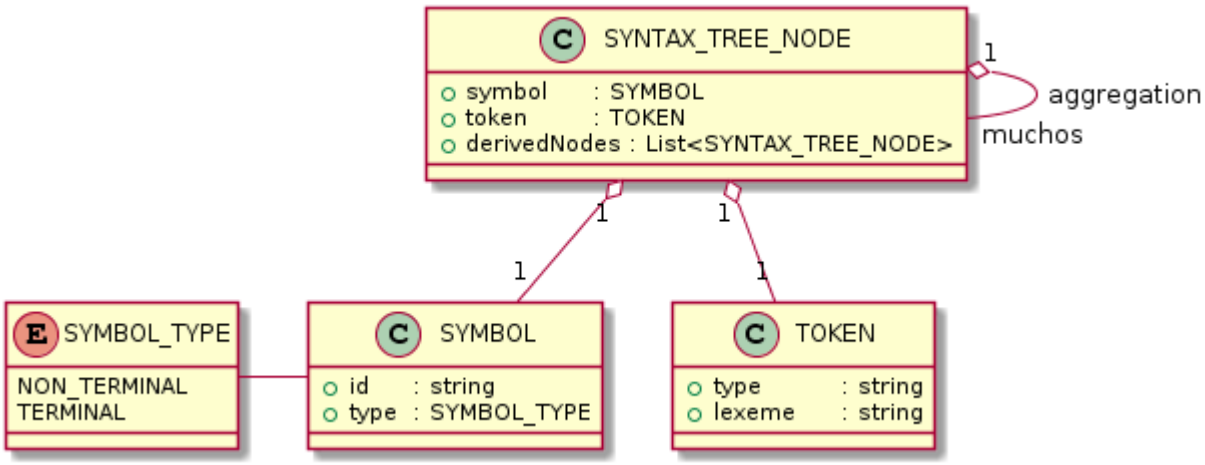

Figura 3-5. Arquitectura del árbol sintáctico

Un nodo del árbol agrega tres elementos.

- Un elemento del tipo SYMBOL extraído de la estructura que representa el símbolo utilizado en la gramática para representar la semántica deseada.
- Un elemento del tipo TOKEN extraído de la lista de tokens como resultado del análisis léxico. Este elemento es importante, ya que contiene toda la información del archivo de entrada en forma de *lexema* y *tipo de lexema*.
- Una lista que contiene los nodos hijos derivados del nodo. Cada nodo hijo es del mismo tipo que el nodo padre *SYNTAX_TREE_NODE*. Es esto lo que permite tener implementar algoritmos recursivos para su procesamiento.

Después de procesar los elementos semánticos del árbol sintáctico mostrado en la Figura 3-3 obtenemos los casos de prueba mostrados en la Tabla 3.2.

Test Case ID	Test Case
TC_1	VERIFY B is set to A WHEN inputTwo > Z is TRUE and inputOne > Y is TRUE.
TC_2	VERIFY B is set to D WHEN inputTwo > Z is not TRUE and inputOne > Y is TRUE.
TC_3	VERIFY B is set to C WHEN inputOne > Y is not TRUE.

Tabla 3.2. Casos de prueba

4 RESULTADOS

Una vez generado el árbol de análisis sintáctico con los elementos semánticos configurados en la gramática, fue posible generar los casos de prueba. En esta sección se presenta una comparativa con el método propuesto por Linzhana y Jiesong desde el punto de vista de generación de casos de prueba a partir de diagramas de actividad UML. Además de que se hace la comparación en cuanto a la reducción de tiempos de ciclo con ASSERT.

4.1 DIFERENCIAS CON EL MÉTODO DE LINZHANA Y JIESONG

Las diferencias se describen en la Tabla 4.1

Característica	**Propuesta de Linzhana y Jiesong**	**TCG_UML**
Toma ventaja de pruebas de caja negra para verificar el comportamiento esperado.	Si	Si
Toma ventaja de pruebas de caja blanca para cubrir la estructura interna del diagrama de actividad de la unidad del sistema en prueba.	Si	No. Toma un set de requerimientos que comparten una misma salida y analiza los caminos generados del diagrama resultante. Sin embargo, nunca considera la implementación
Es útil para encontrar defectos de implementación	Si	Si
Permite encontrar sobre implementación y sub-implementación	Si	Si
Los casos de prueba pueden ser generados en paralelo con la implementación de código ya que las pruebas de diseño pueden comenzar una vez que la fase de diseño es terminada.	Si	Si
Permite a los ingenieros de pruebas	Si	Si

Característica	Propuesta de Linzhana y Jiesong	TCG_UML
de utilizar mejor los recursos		
Se pueden encontrar errores de diseño al realizar el análisis del modelo	Si	Si
Los errores de diseño se corrigen en etapas tempranas del desarrollo	Si	Si
Previene al ingeniero de pruebas de empaparse con detalles de implementación de forma prematura.	Si	Permite la independencia entre implementación y desarrollo de pruebas para industrias como la aviación.
Es la base para pruebas a partir de modelos.	Si	Si. Sin embargo, también considera requerimientos basados en oraciones.
Utiliza un grafo acíclico dirigido	No, ya que el método no se automatizo.	Al automatizar el método, el grafo es equivalente al Se puede automatizar
Grafo Intermedio	No	Árbol de análisis sintáctico generado de leer el diagrama. La diferencia con otros métodos propuestos es que no necesita información de algún otro diagrama.

Tabla 4.1. Tabla comparativa entre método de caja gris y método de caja negra propuesto.

4.2 TIEMPOS DE CICLO

Los resultados se analizaron en el desarrollo de los casos de prueba del área funcional de interfaz de usuario dentro de la funcionalidad de sintonización de radios de navegación. Específicamente la habilitación y des habilitación de elementos gráficos de la pantalla. La funcionalidad analizada y procesada es propiedad de General Electric (GE), por lo tanto, los diagramas generados y casos de prueba no pueden ser entregados como parte de los resultados de este proyecto.

Sintonización de Radios de Navegación - UI	Horas	Porcentaje	ASSERT	TCG_UML
Requerimientos de Alto Nivel (TRL5)	**167.808**	**8%**	**156.1**	**149.34**
Casos de Prueba (TRL4)	**519.984**	**26%**	**0**	**0**
Requerimientos de Bajo Nivel (TRL2)	130.41	7%	130.41	130.41
Código (TRL4)	280.14	14%	280.14	280.14
Procedimientos de Prueba (TRL4)	879.06	44%	879.06	879.06
Horas totales	1977.402	100%	1445.71 **(- 26%)**	1438.95 **(-27%)**

Tabla 4.2 – Cuadro comparativo de los tiempos de ciclo.

En la Tabla 4.2 podemos ver como TCG_UML reduce el tiempo total de desarrollo en un 27%. Es importante mencionar que los resultados son muy similares a los de ASSERT que lo reduce en un 26%. Sin embargo, es posible apreciar el impacto en la reducción de tiempo de ciclo para definir requerimientos de alto nivel. Las herramientas no fueron utilizadas para generar los requerimientos de bajo nivel; sin embargo, en base a los resultados con los requerimientos de alto nivel, es esperado reducir aún más el tiempo de ciclo. TCG_UML mejoró el tiempo de ciclo en los requerimientos de alto nivel, ya que un diagrama de actividad es capaz de agrupar varios requerimientos en un solo diagrama; por consecuencia, al tener menos elementos que analizar y revisar, el tiempo se reduce.

TCG_UML solo afecta la forma en que los requerimientos son documentados y la generación de los casos de prueba. Es necesario generar los requerimientos utilizando los diagramas de actividad que servirán como archivos de entrada para generar los casos de prueba; de otra forma, el generar requerimientos en forma de oraciones y además diagramas de actividad incrementaría el tiempo de desarrollo.

Es importante señalar que los tiempos mostrados en la Tabla 4.2 son generados por un equipo de seis ingenieros de software. Debido a que el uso de diagramas de actividad para reducir el tiempo de desarrollo no es autorizado aún por TrueCourse, la Tabla 4.2 muestra los tiempos de desarrollo generados por dos ingenieros de software. Sin

embargo, la información que nos arroja nos permite confirmar que el costo en tiempo invertido en generar casos de prueba es cercano a cero. Se utilizaron diez horas que no se mencionan en la tabla para revisar los casos de prueba generados y diez horas de entrenamiento en la sintaxis de los elementos semánticos configurados. Sin embargo, una vez que la herramienta este cualificada de acuerdo con el DO-178C, este tiempo de revisión no será necesario.

Elemento	**ASSERT**	**TCG_UML**
Razonamiento de conceptos	Automatizado	Manual
Generación de casos de prueba antes de tener una línea de código	Automatizado	Automatizado
Capacidad de adaptarse a los conceptos del proyecto.	Si	Si
Capacidad de adaptar la sintaxis a los conceptos semánticos del proyecto	**No**	**Si**
Reducción de costos	Si	Si
Detección temprana de errores	**Automática**	**Manual**

Tabla 4.3 – Tabla comparativa con ASSERT

La Tabla 4.3 muestra las principales diferencias obtenidas entre ASSERT y TCG_UML. Es posible observar cómo TCG_UML tiene ventaja sobre ASSERT hablando de la capacidad de adaptarse a las necesidades conceptuales del TrueCourse; sin embargo, TCG_UML aún no cuenta con herramientas de detección de errores de forma automática en la fase de captura de requerimientos.

4.3 COBERTURA FUNCIONAL

El *Proyecto TrueCourse* sigue un concepto de desarrollo evolutivo que comprende nueve niveles. Por la parte de verificación al momento de tomar los datos, se pide un nivel cuatro, el cual pide demostrar el funcionamiento correcto del software. La gramática se diseñó buscando cumplir con el nivel de evolución esperado de los casos de prueba. Es por esta razón que al comparar los casos de prueba generados a mano con los generados con la plataforma de TCG_UML, obtenemos un nivel de cobertura funcional equivalente al 100%. Esto no significa que, al evolucionar el nivel de verificación del Proyecto, perdamos la cobertura funcional al 100%; el factor de

adaptabilidad semántica en la plataforma nos permite evolucionar la gramática junto con el Proyecto a través de los nueve niveles de evolución.

CONCLUSIONES

Como se menciona en la sección 4.2 se logró reducir el tiempo de desarrollo en un valor igual o superior al 23%. La sintaxis de PlantUML permite desarrollar herramientas de análisis con la flexibilidad de definir elementos semánticos adaptables a las necesidades de cada proyecto.

En la Tabla 2.4 se mencionan distintos métodos para generar casos de prueba a partir de un diagrama de actividad. La mayoría de ellos hablan de la generación de un Grafo de Modelo de Flujo intermedio donde se definen los caminos funcionales; sin embargo, el grafo generado no tiene información más allá de la ya contenida en un diagrama de actividad. Los diagramas de UML, como se menciona en la sección 1.1, carecen de una semántica definida. La carencia de la semántica definida le otorga flexibilidad a UML de adaptarse a cualquier proyecto. Pero, trae consigo la desventaja de una automatización complicada, que al final empuja a la estandarización de una semántica. Al combinar el concepto de análisis sintáctico de los compiladores con el concepto de semántica configurable, nos permite adaptarnos a las necesidades de cualquier proyecto logrando una cobertura funcional del 100%. PlantUML ofrece una sintaxis básica con cierto contenido de información, a partir de esta sintaxis general, la definición de elementos semánticos adicionales fue necesario.

Los elementos semánticos arreglados recursivamente en una estructura de un árbol sintáctico permiten la segmentación y definición de lexemas específicos dentro de una sintaxis específica. Los elementos semánticos bien definidos facilitan el procesamiento del árbol sintáctico para generar los casos de prueba. Sin embargo, el factor de adaptabilidad es un arma de dos filos, ya que, de no seleccionar los elementos semánticos adecuadamente, el procesamiento de los elementos semánticos del árbol se vuelve complicado o en su defecto cíclico.

Por el contrario, al manejar el factor de adaptabilidad semántica, que TCG_UML otorga a cualquier archivo de entrada, adecuadamente; es posible automatizar la generación de cualquier elemento de salida. Para efectos de este proyecto, casos de prueba a partir de diagramas de actividad en PlantUML.

TCG_UML resuelve y explota el problema de UML que ASSERT menciona. Debido a que ASSERT utiliza la sintaxis de SADL para Requerimientos (SLR), no es capaz de agregar nuevos elementos semánticos dentro de su sintaxis. Es por esta razón que, en proyectos grandes, es necesario encontrar alternativas a ciertos elementos semánticos que no son considerados dentro de SADL. Sin embargo; aunque TCG_UML simplifica el análisis de los requerimientos para corregir errores, aún carece de las herramientas para detectar y corregir errores en la etapa de captura de requerimientos.

APORTACIÓN DE LA TESIS

Esta tesis toma los esfuerzos realizados por generar casos de prueba de forma automática a partir de diagramas de actividad UML e incorpora el nuevo concepto de Semántica Adaptable a las necesidades del proyecto. El concepto es implementado a través de Gramáticas Libres de Contexto (GLC), analizadores léxico y sintáctico configurables en una plataforma versátil.

RECOMENDACIONES

El alcance del proyecto se limitó a reducción del tiempo de desarrollo eliminando el tiempo requerido para generar casos de prueba de forma manual. Sin embargo, los diagramas de actividad bajo un concepto de desarrollo basado en modelos podrían reducir el tiempo de definición de los requerimientos de un sistema. Se recomienda explorar esta posibilidad en estudios posteriores.

Los requerimientos funcionales de acuerdo con la Tabla 2.2 representan un 80% de todos los requerimientos del sistema. Un diagrama de actividad como se menciona en la sección 2.4.2.1 es adecuado para este tipo de requerimientos. Sin embargo; explotando el factor de adaptabilidad semántica que el TCG_UML otorga a UML es posible cubrir con diagramas de actividad el 20% de requerimientos restantes; potencializando de esta manera el ahorro en costo de desarrollo de un sistema.

La industria de la aviación es una industria de evolución lenta, antes de poder aplicar herramientas que parten de diagramas de UML que sugieren un desarrollo basado en modelos, es necesario hacer un cambio de mentalidad e integrar al desarrollo de los sistemas de aviación el uso de UML para describir requerimientos de un sistema. Sin embargo, TCG_UML puede ser aplicado a cualquier industria y no se limita únicamente a diagramas de actividad que siguen la sintaxis básica de PlantUML como archivos de entrada; por el contrario, se puede definir una semántica para cualquier archivo de entrada siempre que la gramática generada siga las reglas de Noam Chomsky.

Sin embargo, la plataforma necesita del desarrollo de los siguientes elementos para lograr facilitar la interacción con el usuario:

- Desarrollo de una interfaz gráfica adecuada para potencializar todos los beneficios de una semántica adaptable que otorga TCG_UML.
- Análisis de las gramáticas de entrada bajo las reglas de Naom Chomsky para evitar los ciclos infinitos dentro de los algoritmos recursivos implementados.
- Guardar los grafos generados para evitar este tiempo de procesamiento cada vez que se procese un archivo de entrada.

El tema de gramáticas libres de contexto es amplio y tiene distintas áreas de investigación, TCG_UML facilitaría el análisis de gramáticas que se estén estudiando, siempre y cuando sigan las reglas de Chomsky.

Cada proyecto es distinto y tiene sus propias necesidades. UML se vuelve un aliado muy importante para lograr acuerdos; sin embargo, TCG_UML garantiza que los acuerdos se respetan, sobre todo, cuando los grupos de desarrollo del sistema son grandes.

La gramática definida solo considera los caminos funcionales condicionales. Los ciclos no se consideraron ya que en la mayoría de los algoritmos se utilizan para realizar búsquedas o recorridos de listas. Desde el punto de vista funcional, no es necesario definir estos ciclos dentro del diagrama UML. Oraciones como "búsqueda" pueden ser utilizados dentro de un elemento semántico bien definido. Sin embargo, cabe la posibilidad de que algún algoritmo forzosamente requiera del ciclo en su definición y no solo en su implementación. Una línea de investigación sería el definir y procesar los elementos semánticos de los ciclos funcionales; y la forma de expresarlos al generar casos de prueba.

Es necesario desarrollar herramientas para la plataforma de TCG_UML que permitan el análisis de todos los diagramas de actividad UML y detecten errores de diseño en la etapa de captura y modelado de requerimientos, esto, como se menciona en la sección 2.2 reducirá en un 35% los errores introducidos en la etapa de captura de requerimientos y permitirá reducir los tiempos de ciclo de desarrollo significativamente.

Por último, TCG_UML puede evolucionar y generar a partir del diagrama de actividad procedimientos de prueba y código ejecutable. Para efectos de este proyecto se limitó a la generación de casos de prueba; pero los elementos semánticos del árbol de análisis sintáctico pueden ser procesados para generar cualquier clase de salida requerida. Se recomienda buscar la forma de automatizar la generación de todos los artefactos de verificación necesarios. De esta forma se podría alcanzar un ahorro del 40% en el desarrollo de los proyectos en la industria de la aviación.

REFERENCIAS BIBLIOGRÁFICAS

1. **Godfreym, Michael W. y German, Daniel M.** On the evolution of lehman's laws. *JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS.* 2013, pág. 7.

2. **WIKIPEDIA. La enciclopedia libre.** La historia del transistor. [En línea] 14 de Septiembre de 2019. [Citado el: 25 de Octubre de 2019.] https://es.wikipedia.org/wiki/Historia_del_transistor.

3. **Olivo F. D'Inca, Celeste.** Breve Historía de la Computadora. [En línea] Revista de la Universidad de Mendoza. [Citado el: 25 de Octubre de 2019.] http://www.um.edu.ar/ojs-new/index.php/RUM/article/view/110/131.

4. **Barra P. Carlos.** Software e Ingenieria de Software. [En línea] Capitán de Corbeta. [Citado el: 25 de Octubre de 2019.] http://revistamarina.cl/revistas/1998/1/barra.pdf.

5. *La Ingeniería de requerimientos y su importancia en el desarrollo de proyectos de software*. **Arias Chaves, Michael.** 2005, InterSedes, Vol. VI, págs. 1-13.

6. **Ramos Cardozzo, Daniel.** *Desarrollo de Software: Requisitos, Estimaciones y Análisis.* s.l. : Campis Academy, It, 2016.

7. *Metodologías ágiles para el desarrollo de software: eXtreme Programming.* **Letelier, Patricio.** 26, 2006, Técnica Administrativa, Buenos Aires, Vol. 05.

8. *La incertidumbre como herramienta en la ingeniería de software*. **Medinilla, Nelson y Gutiérrez, Inmaculada.** 2006, JISBD.

9. **Miller, Sam.** Contribution of Flight Systems to Performance-Based Navigation. [En línea] [Citado el: 29 de Septiembre de 2019.] https://www.boeing.com/commercial/aeromagazine/articles/qtr_02_09/pdfs/AERO_Q209_article05.pdf.

10. **GE Aviation.** TrueCourse. [En línea] GE Aviation, 2019. [Citado el: 13 de Octubre de 2019.] https://www.geaviation.com/systems/avionics/navigation-guidance/truecourse-flight-management-system-fms.

11. —. TrueCourse Flight Management System. [En línea] GE Aviation, 06 de Julio de 2016. [Citado el: 13 de Octubre de 2019.] https://www.youtube.com/watch?v=GZsbNNiB1L4.

12. **FREE FLIGHT.** Free Flight Systems. [En línea] [Citado el: 29 de Septiembre de 2019.] https://www.freeflightsystems.com/wp-content/uploads/2019/02/2301_Updated.pdf.

13. **Company, Ada Core The GNAT Pro.** *DO-178C: A New Standard for Software Safety Certification.* New York : AdaCore, 2010.

14. **Siu, Kit, y otros.** Flight critical software and systems development using ASSERT. [En línea] IEEE Xplore Digital Library, 09 de Noviembre de 2017. [Citado el: 13 de Octubre de 2019.] https://ieeexplore.ieee.org/document/8102059/authors#authors.

15. **Booch, Grady.** Análisis y Diseño Orientado a Objetos. [En línea] [Citado el: 14 de Octubre de 2019.] https://s3.amazonaws.com/academia.edu.documents/58512537/edoc.site_analisis-y-diseo-orientado-a-objetos-grady-booch.pdf?response-content-disposition=inline%3B%20filename%3DEdoc.site_analisis_y_diseo_orientado_a_o.pdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz.

16. **Picón M., Mercedes y Zulay Maldonado, Carmen.** Generación Automática de Código basada en Modelos UML. [En línea] 28 de Octubre de 2016. [Citado el: 14 de Octubre de 2019.] http://concisa.net.ve/memorias/CoNCISa2016/CoNCISa2016-p134-138.pdf.

17. **Pons, Claudia.** *El proceso de desarrollo de software basado en modelos.* s.l. : Lifia-Universidad Nacional de la Plata, 1999.

18. **DeMarco, Tom.** *Structured Analysis and System Specification.* The University of Michigan : Prentice-Hall, 1979.

19. **Oriente, Joaquín.** UML 2: ¿Cuántos tipos de diagramas existen? [En línea] 9 de Julio de 2014. [Citado el: 05 de Septiembre de 2018.] http://joaquinoriente.com/2014/07/09/uml-2-cuantos-tipos-de-diagramas-existen/.

20. **Ferré Grau, Xavier y Sánchez Segura, María Isabel.** Desarrollo Orientado a Objetos con UML. [En línea] [Citado el: 14 de Octubre de 2019.] http://rafaelmellado.cl/material/com3162/complementario/05.pdf.

21. **Lucidchart.** Tutorial de diagrama de actividades. [En línea] Lucid Software Inc, 2018. [Citado el: 16 de Abril de 2018.] https://www.lucidchart.com/pages/es/diagrama-de-actividades-uml.

22. **Wils, Andrew, y otros.** Agility in the Avionics Software World. [En línea] Springer Link, 2006. [Citado el: 2019 de Octubre de 2019.] https://link.springer.com/chapter/10.1007/11774129_13.

23. **ANSYS.** ANSYS SCADE Suite . *Mission and safety-critical control systems run on software created in SCADE.* [En línea] [Citado el: 14 de Octubre de 2019.] https://www.ansys.com/products/embedded-software/ansys-scade-suite.

24. **Ortiz, Carlos y Arredondo, Eréndira.** Competitividad y factores de éxito en empresas desarrolladoras de software. [En línea] Dialnet, 2014. [Citado el: 14 de Octubre de 2019.] https://dialnet.unirioja.es/servlet/articulo?codigo=5101928.

25. **PlantUML.** PlantUML in a nutshell. [En línea] [Citado el: 14 de Octubre de 2019.] http://plantuml.com/.

26. **Schrage, Andrea.** Microeconomía 1, mercados competitivos. [En línea] Universidad Carlos III de Madrid, 2006. [Citado el: 23 de December de 2017.] http://www.eco.uc3m.es/~aschrage/MicroI_archivos/Tema%202%.

27. **Vazquez, Juan Carlos.** *Costos.* Argentina : Aguilar, 1984.

28. **C. Mankins, John.** Technology Readiness Levels. [En línea] NASA - Office of Space Access and Technology, 6 de Abril de 1995. [Citado el: 27 de Octubre de 2019.] https://aiaa.kavi.com/apps/group_public/download.php/2212/TRLs_MankinsPaper_1995.pdf.

29. **Kaner, Cem, Falk, Jack y Nguyen, Hung Quoc.** *Testing Computer Software.* United States of America : Wiley Computer Publishing, 1999.

30. **Rierson, Leanna.** *Developing Safety - Critical Software. A Practical Guide for Aviation Software and DO-178C Compliance.* s.l. : CRC Press, 2013.

31. **Drake, J. M.** *Verificación y Validación.* s.l. : Ingenieria de Programación, 2009.

32. *Test Case Generation From UML Models - A Survey.* **Sheba, D. P. y Priya, Shanmuga.** 2013, IJETAE Exploring Research and Innovations, Vol. 3, pág. 11.

33. **Serna M., Edgar y Arango I., Fernando.** Revista de Ingeniería. [En línea] SciELO, Julio de 2011. [Citado el: 19 de Octubre de 2019.]

http://www.scielo.org.co/scielo.php?pid=S0121-49932011000300006&script=sci_arttext&tlng=en.

34. **L. Goldman, James, Abraham, George y Song, Ii-Yeol.** Generating Software Requirements Specification (IEEE-Std. 830-1998) document with Use Cases. [En línea] Drexel University, 1998. [Citado el: 19 de Octubre de 2019.] http://www.pages.drexel.edu/~sga72/docs/SRSwithUseCases.pdf.

35. **IEEE.** IEEE-STD-830-1998: Especificaciones de los requisitos del software. s.l. : IEEE, 1998.

36. **B. Carpenter, Paul.** Verification Of Requirements for Safety-Critical Software. [En línea] Aonix, 1999. [Citado el: 19 de Octubre de 2019.] http://www.sigada.org/conf/sigada2001/private/SIGAda2001-CDROM/SIGAda1999-Proceedings/p23-carpenter.pdf.

37. **Fowler, Martin y Scott, Kendall.** *UML Gota a Gota.* México : Pearson Edicación, 1997.

38. *Test Case Generation from Behavioral UML Models.* **Mohapatra, Durga Prasad.** 0975-8887, 2010, International Journal of Computer Applications, Vol. 6, pág. 8.

39. *A Systematic Apporach to Testing Design Models.* **T, Trong Dinh.** Lisbon, Portugal : s.n., 2004. Doctoral Symposium, 7th International Conference on the Unified Modeling Language. págs. 10-15.

40. *Rigorous testing by merging structural and behavioral uml representations.* **O., Pilskalns, y otros.** San Francisco, CA : s.n., 2003. Proceeding of the 6th International Conference on the Unified Modeling Language.

41. *Test adequacy assessment for UML design model testing.* **S, Ghose, y otros.** 2003, Preceeding of the International Symposium on Software Reliability Engineering, págs. 332 - 343.

42. *Generating Test Cases from UML Activity Diagrams based on Gray-Box Method*". **W., Linzhang y Y., Jiesong.** 2004. Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC04).

43. **Álvarez, Jorge Cortés.** SlideShare. [En línea] 17 de Febrero de 2013. [Citado el: 04 de Agosto de 2019.] https://es.slideshare.net/cortesalvarez/diagrama-de-actividades-16587539.

44. **UMLet.** Free UML Tool for Fast UML Diagrams. [En línea] GBU General Public License. [Citado el: 09 de September de 2019.] https://www.umlet.com/.

45. **Real Academia Española.** *Diccionario de la Lengua.* s.l. : Real Academia Española, 2014.

46. **Waite M, William.** *Compiler Construction.* Boulder, Colorado : University of Colorado.

47. **V., Gloria Inés Álvarez.** *Compiladores.* s.l. : Pontifica Universidad Javeriana Cali.

48. **RSI.** Solucion de problemas con algoritmos y estructura de datos. [En línea] [Citado el: 09 de Septiembre de 2018.] http://interactivepython.org/runestone/static/pythoned/Graphs/BusquedaEnProfundidadGeneral.html.

49. **Hopcroft, John E., Motwani, Rajeev y Ullman, Jeffrey D.** *Teoría de Autómatas, lenguajes y computación.* Madrid : Pearson Educación S.A., 2008.

50. **Leal, Vicente.** Problemas de economía - oferta y demanda. [En línea] Aprende Economía, 2009. [Citado el: 25 de Enero de 2018.] https://aprendeconomia.files.wordpress.com/2009/12/oferta-y-demanda-1.pdf.

51. **Thakur, Dinesh.** ECOMPUTER NOTES. [En línea] [Citado el: 16 de June de 2019.] http://ecomputernotes.com/compiler-design/convert-regular-expression-to-dfa.

52. **Nayan, Ruparelia B.** Software development lifecycle models. [En línea] ACM Digital Library, 2010. [Citado el: 14 de Octubre de 2019.] https://dl.acm.org/citation.cfm?id=1764814.

53. **WIKIPEDIA. La enciclopedia libre.** Lenguaje Unificado de Modelado. [En línea] Noviembre de 2017. [Citado el: 05 de Enero de 2018.] https://es.wikipedia.org/wiki/Lenguaje_unificado_de_modelado.

ANEXO A - ARQUITECTURA DE TCG_UML

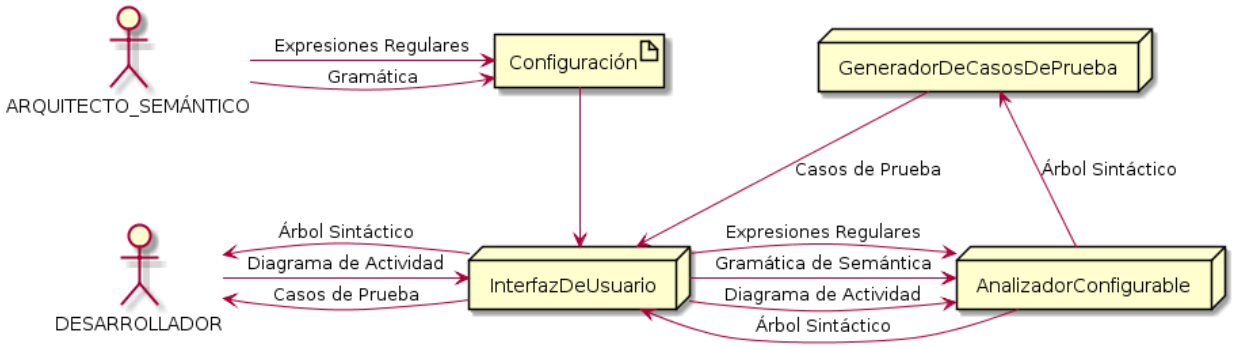

Figura 0-1 TCG_UML – Arquitectura General

ANEXO B - ALGORITMO

1. **Leer las expresiones regulares** que se encuentran en el archivo de configuración. Las expresiones regulares sirven para clasificar cada una de las palabras o lexemas en tokens específicos. La Tabla 0.1 muestra la clasificación de los tokens y las expresiones regulares utilizadas para identificar los lexemas (49).
2. **Crear Autómatas Finitos Determinísticos (DFA) de cada expresión regular** (49). Cada expresión regular se puede procesar utilizando librerías RegEx; sin embargo, la complejidad del algoritmo era del orden exponencial $O(2^n)$. Al utilizar Autómatas Finitos Determinísticos, el nivel de complejidad del bajo al orden de O(n). Esto reduce significativamente el tiempo de ejecución del sistema. El orden en que se registra cada expresión regular en el archivo de configuración es importante, ya que la prioridad de la clasificación del lexema obedece a este orden establecido por la configuración.

3. Utilizar los DFAs para **clasificar cada lexema del texto de entrada**. Es necesario generar una lista de tokens en el orden en que los lexemas aparecen en el texto de entrada (49).
4. **Cargar la Gramática Libre de Contexto definida** en el archivo de configuración. La gramática debe utilizar como elementos terminales los tokens que se

encuentran definidos en el archivo de expresiones regulares. Además, como elementos no terminales, debe contener los elementos semánticos que se buscan extraer de la información contenida en el archivo de entrada.

5. **Crear el árbol sintáctico** utilizando la lista de tokens y la Gramática Libre de Contexto (49). Si la Gramática Libre de Contexto está diseñada de forma correcta, cada rama del árbol sintáctico representara cada elemento semántico deseado y sus hojas representan los lexemas pertenecientes a la semántica de la rama.
6. **Procesar cada elemento semántico** dentro del árbol sintáctico. Es necesario implementar funciones que procesen cada rama del árbol. Estas funciones deben estar orientadas a generar la salida deseada utilizando la información dentro de cada elemento semántico.

ANEXO C - EXPRESIONES REGULARES CONFIGURADAS

Tipo de token	Expresión regular que clasifica al lexema
comment	//([A-Z] | [a-z] | [0-9] | | |#|$|%|=|@|!{|}|,|`|~|&|*|\(|\)|'|?|.|:|;|_|\ ||^|/|+|\[|\]|"|-)*
decimalnumber	[0-9]*.[0-9][0-9]*
integernumber	[0-9][0-9]*
identifier	([a-z] | [A-Z] | _)([a-z] | [A-Z] | [0-9] | _)*
whitespace	( )( )*
newline
tab	
colon	:
semicolon	;
twodots	..
openbraket	\[
closebraket	\]
openparentesis	\(
closeparentesis	\)
pipe	|
inverteddiagonal	\\

tilde	~
lessthan	<
greaterthan	>
equal	==
arrow	->
openquestionmark	?
closequestionmark	¿
arroba	@

Tabla 0.1. Expresiones regulares y su token específico.

ANEXO D - GRAMÁTICA LIBRE DE CONTEXTO DE TCG_UML

Símbolo No Terminal		**Derivación**
PLANTUML_ACTIVITY_DIAGRAM	→	HEADER PATHS FOOTER
HEADER	→	@startuml start
FOOTER	→	stop @enduml
PATHS	→	IF_PATH PATHS IF_PATH
IF_PATH	→	IF ELSE_IF ELSE IF ELSE
IF	→	if CONDITION then (identifier) PATH_CONTENT
ELSE_IF	→	elseif CONDITION then (identifier) PATH_CONTENT
ELSE_IFS	→	ELSE_IF ELSE_IFS ELSE_IF
ELSE	→	else (identifier) PATH_CONTENT endif
EXPRESSION	→	Identifier
CONDITION	→	(identifier COMPARATOR identifier)
COMPARATOR	→	< >
PATH_CONTENT	→	PATHS OUTPUT_SETTING
OUTPUT_SETTING	→	: identifier = identifier ;

Tabla 0.1. Gramática de TCG_UML