# METODOLOGÍA PARA EL CÁLCULO DE COMPLEJIDAD EN PRUEBAS UNITARIAS DE CÓDIGO AUTOGENERADO

# TESIS

PARA OBTENER EL GRADO DE

**MAESTRO EN**

**SISTEMAS INTELIGENTES MULTIMEDIA**

PRESENTA

**ING. ISMAEL MARTÍNEZ GARCÍA**

QUERÉTARO, QUERÉTARO, AGOSTO 2017.

## AGRADECIMIENTOS

*Primeramente, quiero agradecer a Dios por darme la oportunidad y escuchar mis oraciones para que esto terminara bien, por darme la paciencia y abrir mi mente a nuevos conocimientos, por darme salud para cumplir con mis proyectos de vida y guiarme en el camino del bien para seguir cosechando éxitos.*

*Quiero agradecer y dedicar este logro a mi esposa Jessica Reyes, a mi hija Alexa Martínez y a mi hijo Axel Martínez que está por nacer, porque siempre creyeron en mí, Jessica siempre me apoyó incondicionalmente teniéndome paciencia y comprensión, agradezco sus consejos y el valor que muestra para salir adelante, y por supuesto, le agradezco por su amor. A Alexa, que tuvo que sacrificar horas sin la compañía de su padre, aprovecho para disculparme por aquellos momentos de juego negados de mi parte, y porque talvez a su corta edad no entienda porque prefiero estar sentado horas frente a una computadora en lugar de estar conviviendo con ella. Sin embargo, a pesar de todo esto, todos aprendimos a disfrutar al máximo la oportunidad de estar juntos en familia. Lo digo abiertamente, este logro no es sólo mío, este logro es también de ellas.*

*Agradezco a mi abuela María y a mi madre Virginia donde quiera que esté porque me enseñaron a no desfallecer y a no rendirme ante nada, porque ellas son el pilar de lo que soy y la base de mi educación además de que trazaron la vereda por la cual hoy camino, también quiero agradecer a mi padre Rubén por sus consejos oportunos, por supuesto también a mis hermanos Lupita y Juan José por siempre creer en mí.*

*A mis compañeros de estudio y a mis maestros por su apoyo en los cursos, asesorías y consejos.*

*Sinceramente agradezco a mi asesor de Tesis, M.I.E Ramón Reyes, por su esfuerzo y dedicación, gracias por transmitirme su conocimiento y por su orientación y su manera de trabajar. Su paciencia, persistencia y motivación han sido clave para mi formación como investigador.*

# RESÚMEN

Es bien sabido que en el mundo de desarrollo de software existen diferentes tipos de metodologías para calcular la complejidad del mismo software, algunas de estas están basados, por ejemplo, en el número de líneas ejecutables de código, otras en la interacción que existe entre las funciones definidas en el producto.

En general, podríamos decir que todo lo que nos rodea es complejo, y en lo que respecta al software existen varias perspectivas para ésta complejidad, es decir, un software puede ser difícil de diseñar, de implementar o de probar, incluso de todo lo anterior mencionado.

Existen diferentes tipos de pruebas que se le realizan al software, una de ellas es la prueba unitaria, ¿qué tan compleja podría ser?, esto depende de la unidad a probar y del proceso que se utiliza para realizar este tipo de prueba. Cada empresa dedicada al desarrollo de software tiene sus propios procesos y herramientas para realizar estas pruebas, esto es, podríamos tener la misma prueba unitaria por dos procesos diferentes y podrían resultar una más compleja que la otra.

Éste proyecto de investigación analiza las metodologías existentes para calcular la complejidad del software, y con ello propone una nueva metodología que nos proporcione el grado de complejidad que puede llegar a tener una prueba unitaria. Se busca que la metodología propuesta pueda considerar aspectos de las pruebas unitarias que posiblemente ninguna otra metodología toma en cuenta; éstos aspectos incluyen la documentación que hay que generar para la prueba unitaria para un proceso definido, o bien, las técnicas de diseño de casos de prueba así como también las herramientas de ayuda usadas en un dicho proceso.

# ÍNDICE DEL CONTENIDO

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# GLOSARIO

## A

AUP

Actual Usage Pair, 35

## D

DO-178B

Software Considerations in Airborne Systems and Equipment Certification, 2

## E

EASA

European Aviation Safety Agency, 3

## F

FAA

Federal Aviation Administration, 3

## I

ISO

International Organization for Standarization, 12

## L

LOC

## N



## P



## R



## S

# CAPÍTULO 1. INTRODUCCIÓN

## 1.1 ANTECEDENTES

En la industria aeronáutica, la tecnología ha crecido a pasos agigantados en las últimas décadas, tanto en el diseño, como en la manufactura de aparatos mecánicos capaces de elevarse en vuelo, un ejemplo de estos avances tecnológicos son las formas de los aviones totalmente adecuadas para reducir su resistencia al aire, esto, producto de la aplicación de la aerodinámica que se define como una rama de la mecánica de fluidos que estudia el movimiento del aire y otros gases, y su interacción con los cuerpos que se mueven en ellos [1].

Otro avance tecnológico en la aeronáutica es la computación en los aviones, hoy en día, por ejemplo, existe software especializado en pronóstico, mantenimiento y operación de las aeronaves. Éstos avances hacen posible que cada vez sea menos necesario la intervención del piloto en los controles de la cabina; José Antonio Tena Sendra en el artículo “Sobre aviación e Inteligencia artificial” [2] afirma que un estudio realizado por el New York Times determinó que un piloto solo vuela de forma manual su aeronave por siete minutos en promedio, el resto del tiempo que el avión está en el aire, es operado mediante el piloto automático, éste consiste de una serie de controles asistidos por un software que permiten que el avión permanezca por una ruta a cierta altura y velocidad por un periodo de tiempo determinado.

Existen otros ejemplos de software con la funcionalidad específica en la operación de las aeronaves como la del Boeing 787 Dreamliner, éste avión cuenta con un sistema que se encarga de detectar las turbulencias y contrarrestarlas con mecanismos propios del avión, es decir, este innovador software tiene la capacidad de empujar la nave en sentido contrario al que la turbulencia lo está llevando, manteniendo así a la aeronave más estable durante el vuelo. Éste es un claro ejemplo de lo que se puede lograr con software, predecir y actuar ante ciertas circunstancias de la naturaleza en sólo cuestión de milisegundos algo que sencillamente resulta imposible para el ser humano.

También, la posibilidad de aterrizar una aeronave de forma automática es posible gracias a los avances tecnológicos referentes al software; en la actualidad, la mayoría de los aviones comerciales son capaces de realizar lo que se denomina "Autoland", un sistema que automatiza totalmente el aterrizaje de un avión solamente con la supervisión de los pilotos [3], este sistema tiene la capacidad de descender la aeronave incluso cuando las condiciones meteorológicas no son favorables, lo cual haría que efectuar ésta maniobra totalmente por los tripulantes resultara peligroso o más aún imposible.

Estos ejemplos de software son clasificados como críticos en la industria de la aeronáutica ya que una falla o error en uno de estos sistemas podría ocasionar la pérdida de decenas o incluso cientos de vidas humanas. Existen cinco niveles de criticidad dentro de la clasificación del software aeronáutico.

Los niveles de criticidad están en función de la tarea específica que tiene que cumplir el software, es decir, si ésta tarea específica puede poner en mayor o menor medida en peligro la seguridad del avión. Estos cinco niveles van del A al E según el DO-178B documento publicado por la RTCA(Radio Technical Commission for Aeronautics Inc) que define las reglas y estándares para el desarrollo de software dentro de la aviación, además que describe técnicas y métodos apropiados para asegurar la integridad y confiabilidad del mismo software [4].

Los niveles de software son los siguientes [5]:

- Nivel A: Catastrófico
- Nivel B: Peligroso/Severo-Mayor
- Nivel C: Mayor
- Nivel D: Menor
- Nivel E: Sin consecuencias

Estos niveles de software dan lugar a diferentes obligaciones de revisiones y documentación para la certificación de un avión ante las agencias regulatorias como

lo son la Administración Federal de Aviación y la Agencia de Seguridad de Aviación Europea, FAA y EASA por sus siglas en inglés respectivamente, por ejemplo, la documentación para un nivel E es mínima, sin embargo y por obvias razones, para un software de nivel A el DO-178B establece que se debe de cumplir con cada estándar y con cada elemento dentro del mismo documento. Estas agencias regulatorias establecen al DO-178B como el medio aceptado para la certificación del software de aviación. [6]

Por lo anterior, los que desarrollan software para aviación usan estos estándares para el diseño y desarrollo de este tipo de software. Desde su primera versión, el DO-178B define el ciclo de vida del software para la aviación así como el de los procesos, este último incluye el proceso de verificación. [7]

En este proceso de verificación se encuentran los siguientes sub-procesos:

- Revisiones: Proporcionan una evaluación cualitativa de que tan correcto es el software.
- Análisis: Proporcionan una evidencia repetitiva de la exactitud del software.
- Pruebas: Demuestran que el software satisface tanto los requerimientos de bajo nivel como los de alto nivel, además de que proveen un alto grado de confianza de que los errores que pudieran conducir a condiciones de fallo inaceptables han sido eliminados.

EL DO-178B establece que el diseño de las pruebas puede ser tan efectivo como la ejecución de las mismas [8].

## 1.2 DEFINICIÓN DEL PROBLEMA

En el proceso de pruebas, existen las llamadas pruebas unitarias que consisten en ejercitar cada línea de código mediante la ejecución de casos de prueba, éstos son diseñados usando técnicas previamente establecidas en el proceso, y se aplica para cada subconjunto de lógica denominado unidad.

La lógica que gobierna la turbina de una aeronave pude estar dividida entre 650 a 1600 unidades o módulos aproximadamente como se puede apreciar en la siguiente gráfica, cada unidad encapsula cierto tipo de lógica la cual su complejidad es muy variada entre una y otra, ésta puede ir desde una simple asignación hablando en términos de programación a un cálculo muy complejo el cual requiera decenas o incluso cientos de líneas de código.

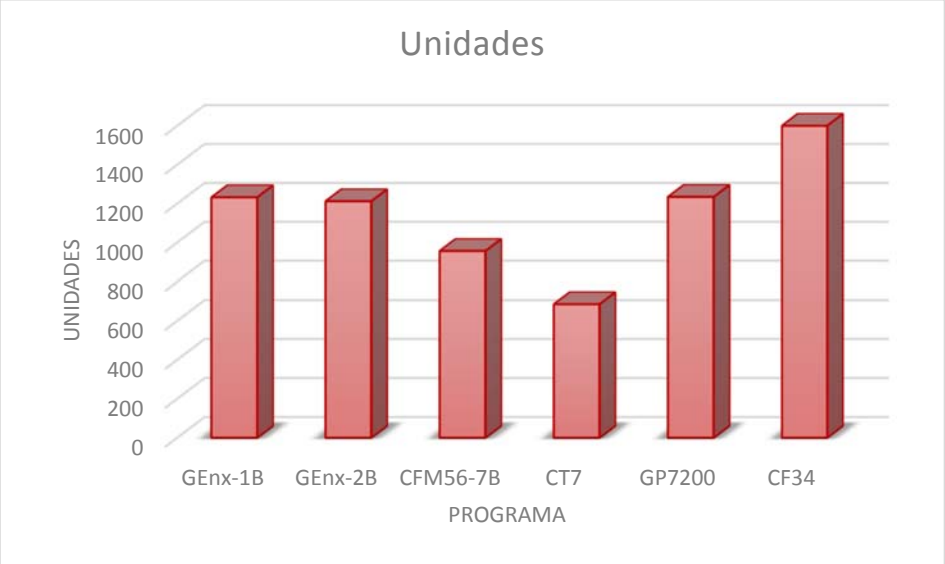


Figura 1Unidades de lógica por programa.

Incluso cuando las turbinas tengan el número de módulos similares, existe la posibilidad de que se ejecuten esfuerzos de pruebas unitarias para sólo un porcentaje del total de unidades de la lógica de la turbina.

Regularmente, la planeación de fechas y presupuestos en el proceso de pruebas unitarias está en función del número de unidades que se van a procesar, ésta planeación es, por supuesto, muy incierta ya que no es lo mismo procesar cien unidades de complejidad “baja” a cien unidades de complejidad “alta”.

Los administradores de proyectos de pruebas unitarias se han dado cuenta que lo antes mencionado no es lo más óptimo para cuestiones de planeación y administración de recursos físicos y humanos dentro del proyecto, entonces, se ha comenzado a registrar tiempos por unidad para llevar un histórico, talvez funcione como referencia al momento de planear, sin embargo existen clientes que cuestionan esta metodología, especialmente los que ejecutarán este proceso por primera vez en sus lógicas.

Lo anterior es por cuestiones de sentido común, ya que la complejidad del software para la turbina A puede ser muy diferente al de la turbina B, provocando que sea difícil que se acepten los planes de entrega del proyecto debido a que han sido estimados con un histórico de otro motor y por consecuente de otro tipo de lógica, dando a lugar a negociaciones de fechas y recursos para llevar a cabo el proceso de pruebas unitarias. La siguiente figura muestra los tiempos de ciclo promedios para la ejecución de este proceso en las principales turbinas.

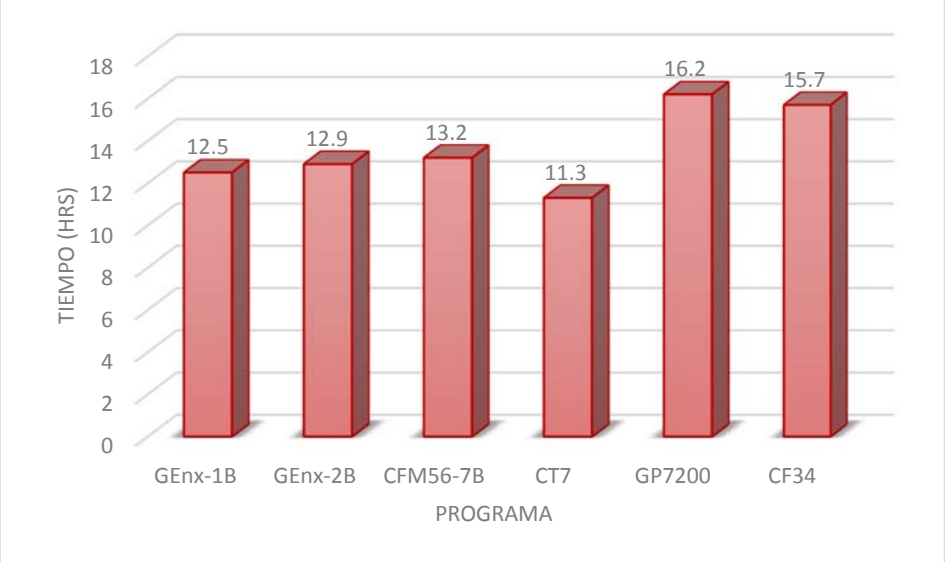

Figura 2 Tiempo de ciclo de trabajo por unidad.

Otra cuestión que se tiene que considerar debido al desconocimiento de la complejidad por unidad es que se debe de realizar un análisis visual para determinar la complejidad de la lógica y así poder determinar la asignación de las mismas para los contribuidores individuales. En la asignación de las unidades a procesar y con el objetivo de hacer más eficiente la ejecución del proceso, se busca que las asignaciones de unidades sean coherentes con la curva de aprendizaje del contribuidor individual al momento de ejecutar el proceso.

Por lo tanto la problemática radica en que no existen herramientas automatizadas que ayuden a estimar la complejidad para determinar la duración de la prueba unitaria, para presentarle al cliente una planeación más precisa; así como proporcionar al líder del proyecto más información para asignar lógicas a los ejecutores de pruebas unitarias de acuerdo a su curva de aprendizaje.

## 1.3 JUSTIFICACIÓN

En los procesos de validación y verificación de software, es difícil contar con un sistema donde su característica principal sea la simplicidad en la planeación y estimación del esfuerzo. Cuando los equipos de verificación son grandes, la administración de las actividades de verificación de software se hace complicada debido a que se desarrollan por varios equipos de trabajo.

La planificación es una actividad de gran importancia en la ingeniería del software, en ésta se establecen objetivos y metas de un proyecto, además de las estrategias y procedimientos para alcanzarlos. Una de las variables a calcular durante la planificación del proyecto es el esfuerzo requerido para el desarrollo del proyecto.

Las estimaciones están asociadas con el esfuerzo, costo y el tiempo de las actividades identificadas del proyecto. El objetivo de la estimación de proyectos es reducir los costos e incrementar los niveles de servicio y de calidad, estas estimaciones se encuentran normalmente asociadas a un valor o un conjunto de valores dentro de un rango probable de resultados, uno de estos valores, por supuesto, podría ser la complejidad de las unidades de software. Existen técnicas de estimación que son una forma de solucionar problemas en donde, en la mayoría de los casos, el problema a resolver es demasiado complejo.

Efectuar de una manera correcta una estimación ayuda a determinar su viabilidad, y de esta manera, los administradores del proyecto se dan cuenta si está destinado al fracaso por no contar con el tiempo, o los recursos necesarios para llevarlo a cabo. En la actualidad son muchos los proyectos que fracasan, e incumplen sus plazos de entrega o bien, las empresas especializadas en la verificación de software se ven obligadas a trabajar a marchas forzadas para cumplir con las fechas aumentando sus costos de operación.

Los resultados de la organización de asesoramiento e investigación enfocada en el desempeño y ejecución de proyectos de desarrollo de software “The Standish Group” [9], publicados en su reporte “2015 Chaos Report” indican que aún hay mucho por hacer para el logro de resultados exitosos en este tipo de proyectos [10]. Esto en su estudio llamado “The Modern Resolution” el cual evalúa el éxito de los proyectos tomando en cuenta si éstos se entregaron a tiempo y con el presupuesto planeado.

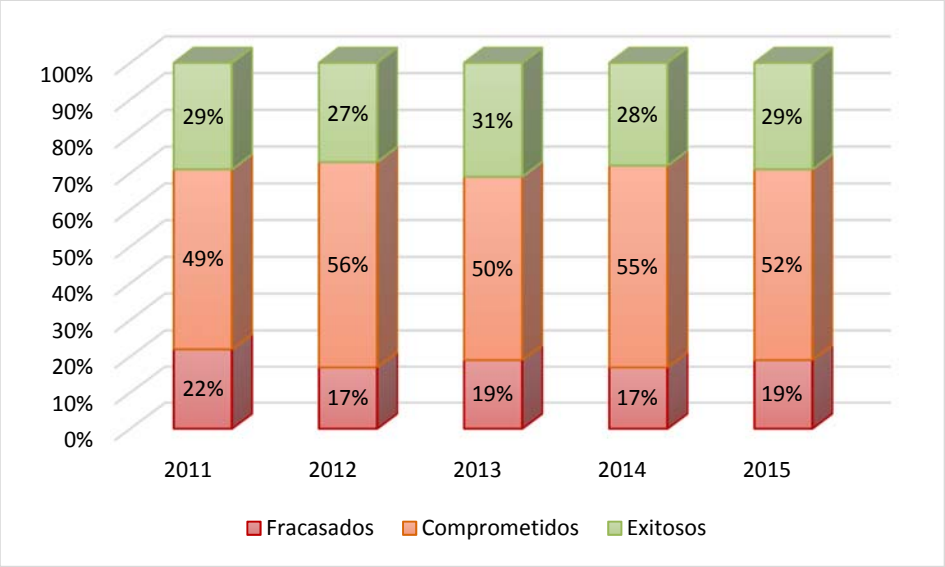


Figura 3 Éxito de proyectos de software según Standish Group.

Conocer de forma oportuna la complejidad de la prueba unitaria, disminuye de manera considerable la incertidumbre en la planeación del proyecto, esto contribuye directamente en la entrega de proyectos a tiempo, además de que se puede hacer un cálculo más real y certero del presupuesto permitiendo a los clientes estar seguros que la planeación se hizo basada a las unidades que se van a procesar y no mediante un histórico.

Además, tener esta métrica permite implementar estrategias de asignación de las unidades para disminuir la frustración del contribuidor individual al momento de estar aprendiendo el proceso, esto es, asignar pruebas unitarias de complejidades bajas e irlas aumentando gradualmente para que su curva de aprendizaje sea más estable, por

supuesto que esto se puede hacer de una manera rápida y eficiente conociendo la complejidad de la prueba ya que no es necesario hacer un análisis visual de las unidades para lograr este último objetivo.

## 1.4 OBJETIVOS

### 1.4.1 Objetivo General

Implementar una metodología que permita calcular la complejidad de una prueba unitaria para lógicas de control de turbinas de aeronaves con código autogenerado en un proceso ya definido, tomando en cuenta las herramientas de ayuda existentes para el proceso y descartando la experiencia del contribuidor individual.

### 1.4.2 Objetivos Específicos

1. Analizar el tipo de lógica a la cual se le realizan las pruebas unitarias.
2. Conocer los estándares para la aplicación del proceso.
3. Analizar las técnicas utilizadas para diseñar los casos de prueba de las pruebas unitarias.
4. Investigar la documentación requerida por cada una de las técnicas aplicadas.
5. Estudiar las consideraciones del diseño de casos se prueba.
6. Investigar las herramientas de ayuda que existen para aplicar las técnicas de diseño de casos y su documentación.
7. Crear una escala o factor que esté en función de las técnicas de diseño de casos.
8. Definir la metodología correcta para aplicar el factor o escala para cada unidad.

## 1.5 HIPÓTESIS

- Es posible desarrollar una metodología para calcular la complejidad de pruebas unitarias, con la cual se estimará de una forma eficiente el tiempo de ejecución de éstas mismas.
- Conocer los tiempos de ejecución de las pruebas unitarias derivados de la complejidad permitirá planear eficazmente las fechas de entrega de los proyectos, y por lo tanto, reducirá el riesgo de no finalizar el proyecto en la fecha estimada.
- Aplicar la metodología para calcular la complejidad de las pruebas unitarias reducirá la diferencia entre los recursos estimados y los reales a no más del 15%.

# CAPÍTULO 2. MARCO TEÓRICO

## 2.1 CICLO DE VIDA DEL SOFTWARE

La ingeniería de software utiliza métodos y herramientas para el desarrollo profesional del mismo, de tal forma que sea fiable y que funcione de una forma eficiente, usando procesos de forma sistemática para idear, implementar y mantenerlo desde que surge la necesidad de hasta que se tiene en producción cumpliendo con el objetivo para el cual fue creado.

Uno de estos procesos sistemáticos es el ciclo de vida del software, el cual la ISO (International Organization for Standarization) define como un marco de referencia que contiene las actividades y las tareas involucradas en el desarrollo, la exploración y el mantenimiento de un producto software, abarcando desde la definición hasta la finalización de su uso. [11]

En el ciclo de vida del software se pueden apreciar tres etapas principales las cuales se explican a continuación:

**Planificación:** Se hace un planeamiento detallado que guíe la administración del proyecto, se estiman tiempo y presupuesto.
**Implementación:** Se definen las actividades que se realizarán para el desarrollo del producto.
**Puesta en Producción:** En esta etapa es donde se presenta al usuario final, funcionando de forma correcta y cumpliendo con los requisitos establecidos en las etapas iniciales.

Antes de estas tres etapas existe una llamada **Inicio**, que es donde surge la idea o la necesidad del software y es donde se definen los objetivos del mismo. Otra más, posterior a las ya mencionadas es la de **Control**, en ésta etapa se hace un análisis de si el producto difiere o no de los requisitos iniciales, aquí es donde se toman acciones correctivas de ser necesarias. En la etapa de control también se genera documentación del producto

como manuales de uso y se proporciona capacitación al usuario final para optimizar el funcionamiento del software. [12]

## 2.2 MODELOS DEL CICLO DE VIDA DEL SOFTWARE

Un modelo de ciclo de vida del software es una representación simplificada del proceso que se lleva a cabo para el desarrollo de éste, es decir, no proporciona información detallada del proceso, sino que se muestran las actividades y secuencias; sin embargo no especifican las personas o roles de quienes realizarán las tareas, así como tampoco se detallan los tiempos de inicio y fin de cada una de las actividades.

Existen varios modelos, sin embargo en este proyecto solo hablaremos del Modelo en Cascada y del Modelo "V", que son de los más comunes en el desarrollo de software en la aeronáutica.

### 2.2.1 Modelo en Cascada

Éste modelo es llamado así debido al paso de una fase en cascada a otra como se observa en la Figura 4, es decir, una fase no puede iniciar hasta que la fase previa finalice, al término de cada fase existen documentos que debieron ser aprobados para poder iniciar con la siguiente. En este modelo se deben planear y programar todas las actividades del proceso antes de empezar a ejecutarlas. [13]

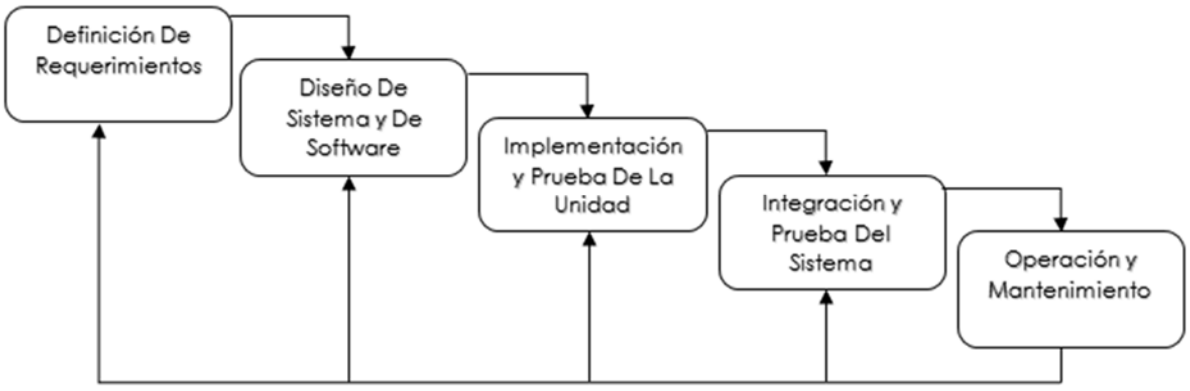


Figura 4 Modelo En Cascada

En las fases del modelo se pueden identificar de cierta forma las principales que se mencionaron con anterioridad. Las etapas del modelo son las siguientes:

1. **Definición de requerimientos:** Aquí se analizan las restricciones y se definen los objetivos del sistema mediante la comunicación con los usuarios finales, posteriormente se definen con detalle para utilizar como especificaciones del sistema.
2. **Diseño de sistema y de software:** Se hace una clasificación de requerimientos de software y de hardware. Aquí se identifican y describen las abstracciones del sistema de software y sus relaciones.
3. **Implementación y prueba de unidad:** El diseño de software de la etapa anterior se implementa en subconjuntos del programa o también llamados unidades del programa. Las pruebas de unidad verifican que cada unidad cumpla con su requerimiento particular.
4. **Integración y Prueba de sistema:** Los programas individuales o unidades se integran para después ser probados como un sistema completo y de esta forma asegurarse que cumple con los requerimientos del software.
5. **Operación y mantenimiento:** Finalmente el software se pone en ejecución y se corrigen errores del mismo que no fueron encontrados en etapas anteriores, también, se hacen actualizaciones al sistema conforme surjan nuevas necesidades y por lo tanto nuevos requerimientos.

Aunque en teoría cada una de éstas etapas tiene que iniciar hasta que la previa termine, en realidad, dichas etapas se traslapan y se retroalimentan una a otra de información, es decir, existe la posibilidad de que en el diseño se identifiquen problemas con los requerimientos, que en la implementación se encuentren errores de diseño, y así sucesivamente, esto explica las líneas de forma ascendente en la Figura 4. [13]

### 2.2.2 Modelo en V

Éste modelo es una variación del modelo en cascada, sin embargo se añaden etapas de retroalimentación haciendo explicito el proceso de verificación en las fases de análisis y diseño como se muestra en la Figura 5.

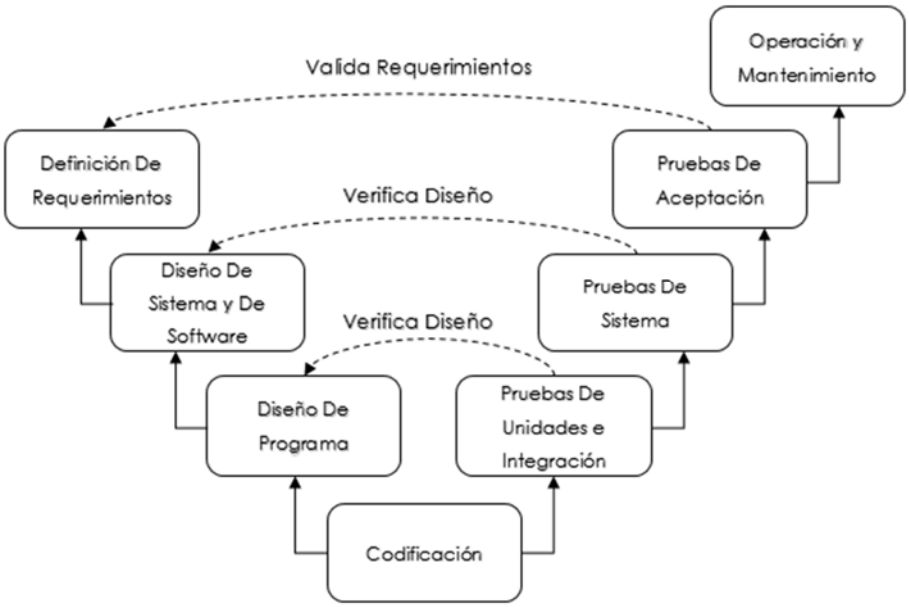


Figura 5 Modelo en V

Los requerimientos inician el proceso como en el de cascada, antes de que la implementación inicie, se crea un plan de pruebas del software (aceptación), éste plan se enfoca en que el producto cumple con la funcionalidad especificada por los requerimientos.

La fase de diseño de sistema y de software define la arquitectura del sistema, donde se prepara un plan de pruebas de sistema el cual verifica que la funcionalidad del software en interacción con el hardware sea la correcta.

En el diseño del programa un plan de pruebas de integración y de unidad es definido, es entonces cuando se comienza a codificar. Una vez que el código es terminado, las

pruebas de unidad, integración y sistema son efectuadas en este orden, es decir, el lado derecho de la V continua su ejecución donde todos los planes de prueba definidos en las etapas anteriores se ponen en marcha. [14]

Existen otros modelos como el incremental, el de espiral o el modelo ágil, pero como se mencionó anteriormente, éstos modelos no son los mejores para software de aeronáutica, por ejemplo, el modelo ágil es más frecuentemente usado para proyectos pequeños donde requieren entregas del producto en periodos cortos de tiempo.

El modelo de cascada es para proyectos largos y con requerimientos bien definidos, mientras que el modelo en V es igualmente para proyectos largos sin embargo los requerimientos pueden o tienden a cambiar un poco además de que el producto requiere de una adecuada verificación en cada una de sus fases.

## 2.3 PRUEBAS DE SOFTWARE

Como se pudo ver en el tema anterior, existen varios tipos de pruebas aplicado en diferentes etapas del desarrollo de software, esto con el objetivo de encontrar errores que se hayan cometido en cada una de las mismas. Los errores son eventualmente corregidos con la finalidad de evitar que aparezcan en campo.

Una generalidad o clases de pruebas aplicadas en el desarrollo de software son las llamadas pruebas de caja negra y pruebas de caja blanca, las cuales se describen a continuación.

**Pruebas de caja negra:** También llamadas pruebas funcionales, este tipo de pruebas ignoran el mecanismo interno del sistema o componente, sólo se enfocan en la salida que proporciona ciertas condiciones de entrada específicas. Quien desarrolla la prueba no tiene acceso o conocimiento de las líneas de código que se generaron para la funcionalidad del sistema, sólo sabe que para la entrada *x* se debe generar la salida *y*.

**Pruebas de caja blanca:** Estas pruebas también son conocidas como pruebas estructurales, éstas toman en cuenta el mecanismo interno del sistema o componente. Quien ejecuta la prueba sabe cómo fue implementado el código y basado en esto diseña los casos de prueba. [15]

La Figura 6 muestra una representación de cada una de éstas dos generalidades.


Figura 6 Pruebas de caja negra (izquierda), Pruebas de caja blanca (derecha)

Una vez entendidas las diferencias principales entre estas clases de pruebas, se podrán discutir los principales tipos de pruebas aplicadas a lo largo del ciclo de vida del software.

### 2.3.1 Pruebas De Unidad

Una unidad de software es la pieza más pequeña que se puede probar, es un conjunto de uno o más módulos de lógica relacionados con un control de datos y procedimientos; estos módulos pertenecen a una sola lógica y cada uno es el objeto único para este tipo de prueba. [16]Comúnmente, el alcance de los módulos o unidades son funciones o clases.

La prueba de unidad verifica la funcionalidad de un módulo de forma aislada [17], se asegura que el código cumpla con el requerimiento que fue especificado en la documentación de la unidad, se hace la verificación mediante la ejecución de la unidad con valores específicos de las entradas haciendo una comparación del comportamiento esperado contra el comportamiento especificado en el requerimiento [16].

En este proceso se utilizan técnicas de diseño de caso de prueba de caja blanca, es decir, cada instrucción que puede ser alcanzada y ejecutada debe ser cubierta por un caso de prueba [16].

### 2.3.2 Pruebas De Integración

Una vez que se han hecho algunas o todas las pruebas unitarias, esto depende del plan de pruebas definido previamente, se inicia con las pruebas de integración. Esta etapa de pruebas asegura que varios módulos de lógica interactúan y pasan datos entre unos y otros de una forma correcta [18], esto se debe, a que los componentes o módulos funcionan correctamente de manera individual como se debió haber demostrado en las pruebas unitarias; no quiere decir que así sea al momento de ser ensamblados o integrados. Por ejemplo, mensajes o datos pueden ser enviados o recibidos entre una unidad a otra de forma incorrecta, o bien, se pueden perder datos al momento de interactuar.

Las pruebas de integración pueden ser implementadas a varios niveles [18], para un producto de software grande y con niveles de confiabilidad altos como en la industria de la aeronáutica, se hacen al más bajo nivel de integración, es decir, integración de módulos de lógica.

Existen diferentes estrategias para hacer la integración del sistema [19], a continuación se describen los dos más comunes:

1. **Basado en descomposición funcional:** Como el nombre lo indica, ésta estrategia se basa en la característica funcional del sistema, es decir, por las acciones o actividades desarrolladas por el módulo. Los cuatro enfoques de esta estrategia son los siguientes.
    a. **Big Bang:** Es el enfoque más fácil de aplicar en las pruebas de integración, todo el sistema se trata como un sub-sistema, es decir, se prueban todos los módulos de una sola vez.

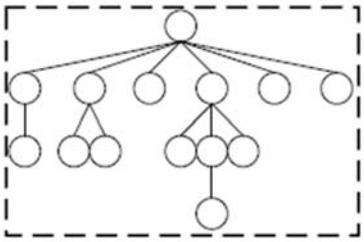

Figura 7 Integración Big Bang [19]

b. **Top-Down:** Ésta estrategia inicia en la raíz del programa y las fases de pruebas se van moviendo hacia abajo a los módulos del más bajo nivel. Se necesitan códigos sustitutos llamados "Stub" en cada fase para simular los módulos de niveles "hijo".

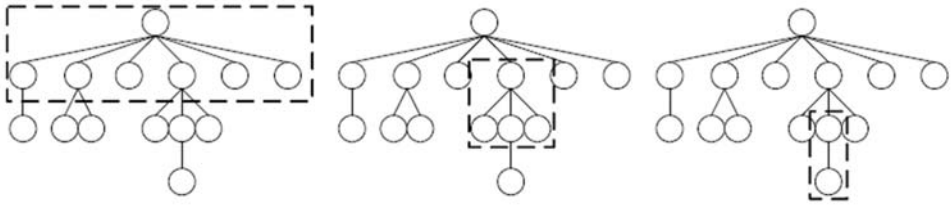

Figura 8 Integración Top-Down [19]

c. **Bottom-Up:** Inicia del nivel más bajo del sistema y se va moviendo hacia la raíz, también se usan códigos substitutos como en la estrategia anterior, sólo que éstos tienen la característica de que deben de llamar el módulo de la fase que se está probando, estos códigos son conocidos como "Drivers".

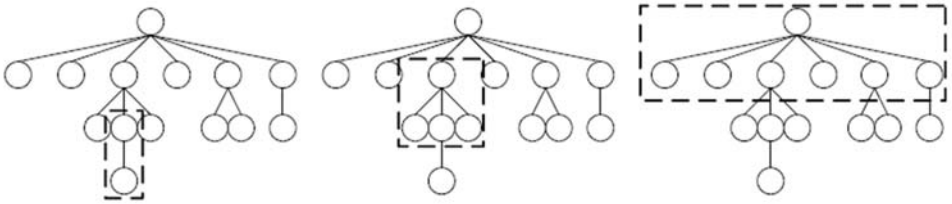

Figura 9 Integración Bottom-Up [19]

d. **Sándwich:** Es una combinación de las dos estrategias anteriores, las sustituciones de código son usadas de acuerdo a la estrategia que se esté usando en la fase, "Stub" si es fase Top-Down o "Driver" si es Bottom-Up. La prueba converge en medio del sistema.

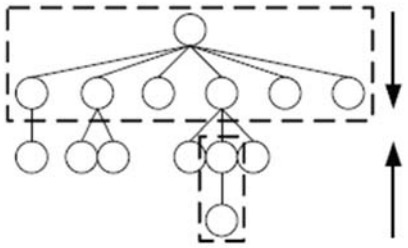

Figura 10 Integración Sándwich [19]

2. **Gráfico De Llamadas:** Ésta estrategia es una mejora de la de descomposición funcional. Aquí se usan gráficos dirigidos en lugar de la descomposición funcional (raíz-hojas), donde los nodos son los módulos y los vértices representan llamadas a funciones o interacciones. Los enfoques son:
   a. **Parejas:** Cada fase de la prueba está restringida a un par de módulos, los pares o parejas se hacen basándose en los vértices [19], es decir, en la interacción entre un módulo y otro.

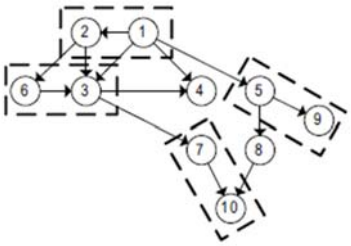


Figura 11 Integración en parejas [19]

   b. **Vecinos:** Los módulos son agrupados como "vecinos", un módulo vecino es el inmediato sucesor o predecesor de otra unidad. En la Figura 11 se puede apreciar como en la parte de la derecha se agrupan los vecinos del nodo 1, en medio los del nodo 2 y a la izquierda los del nodo 3.

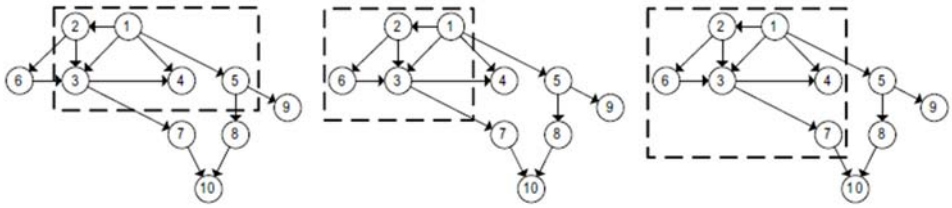


Figura 12 Integración por vecinos [19]

### 2.3.3 Pruebas De Sistema

Hasta este punto, ya se ha asegurado de que cada unidad funciona de forma correcta de manera individual, además de que ya se verificó la interacción entre cada uno de los elementos del software en nuestro producto, ahora es momento de las pruebas de sistema, este tipo de pruebas es realizado sobre un completo e integrado y el objetivo principal de estas pruebas es evaluar que éste cumpla con los requerimientos especificados [20]. Se enfocan en el comportamiento del producto como un todo además de que el código es visto como una caja negra [21].

Realizar pruebas de sistema involucra montar el software en diferentes ambientes para asegurar que funciona, éstos ambientes contienen los normales que son en los cuales el software va a operar una vez que sea entregado al cliente, pero también incluyen otros diferentes como por ejemplo con versiones de sistema operativo diferentes o diferentes plataformas [15].

Los tipos específicos de las pruebas de sistema son:

1. **Pruebas de estrés:** Son realizadas para evaluar un sistema o componente más allá de sus límites o de la especificación de requerimientos [20].
2. **Pruebas de rendimiento:** Evalúan el cumplimiento de un sistema o componente con el rendimiento especificado en los requerimientos [20].
3. **Pruebas de usabilidad:** Se realizan para evaluar hasta qué grado un usuario puede aprender a operar, preparar entradas e interpretar las salidas de un sistema o componente [15].

### 2.3.4 Pruebas De Aceptación

Una vez que las pruebas de sistema se han aprobado, el producto se entrega al cliente, y éste a su vez ejecuta el proceso de aceptación basándose en sus expectativas de funcionalidad del producto. Éste proceso es una prueba formal realizada para determinar si el sistema satisface o no el criterio de aceptación del cliente y para ayudar a éste a determinar si va a aceptar el producto [20].

Las pruebas de aceptación son especificadas por el cliente y puede rechazar la entrega del producto si alguna éstas no cumplió con sus expectativas. Estas pruebas no suelen seguir un procedimiento estricto además de no estar documentadas del todo sino que son más bien basadas en las experiencias de usuario. [21]

La Tabla 1 muestra un resumen de los cuatro tipos de pruebas que se acaban de discutir.

| Tipo De Prueba | Clase | Especificación | Alcance |
|---|---|---|---|
| **Unitaria** | Caja Blanca | Bajo Nivel | Unidad de Lógica (función/Clase) |
| **Integración** | Caja Blanca<br>Caja Negra | Bajo Nivel<br>Alto Nivel | Multiples Unidades |
| **Sistema** | Caja Negra | Análisis de Requerimientos | Sofware Completo En Ambiente Representativo |
| **Aceptación** | Caja Negra | Análisis De Requerimientos | Sofware Completo En Ambiente Del Cliente |

Tabla 1Resumen De Tipos De Pruebas De Software.

## 2.4 GENERADORES AUTOMÁTICOS DE CÓDIGO

En la fase de implementación se producen los módulos de código que seguirán el comportamiento y estructura que se definió en la fase de diseño, usando el lenguaje definido. Cuando se realiza ésta codificación de forma manual, los implementadores se deben apegar a estándares previamente definidos para posteriormente hacerles una validación a los módulos de código y de ésta forma asegurar la integridad del diseño y del cumplimiento de los estándares. Sin embargo, cuando el código es automáticamente generado, la sanidad de las líneas de código es inherente y no necesita revisión de código. [22]

La generación automática de código se refiere al uso de herramientas para su implementación, que de otra forma, los desarrolladores hubieran tenido que generar manualmente line por línea. En esta sección se describirán tres generadores de código usados en la aeronáutica.

### 2.4.1 MathWorks

El generador automático para código embebido Embedded Coder® genera código C y C ++ legible, compacto y rápido para su uso en procesadores embebidos y microprocesadores utilizados en la producción en serie. Este generador permite optimizaciones avanzadas para el control detallado de las funciones, archivos y datos del código.

Las optimizaciones mejoran la eficiencia de la implementación y facilitan la integración con código ya existente, los tipos de datos y los parámetros de calibración utilizados en la producción. Puede incorporar un entorno de desarrollo de terceros en el proceso de creación para producir un ejecutable para la implementación en el sistema embebido. También proporciona informes de trazabilidad, documentación de interfaz de código y verificación automatizada de software para soportar el desarrollo de software DO-178. [23]

La Figura 13 muestra el diagrama con el cual se genera el código de la Figura 14.

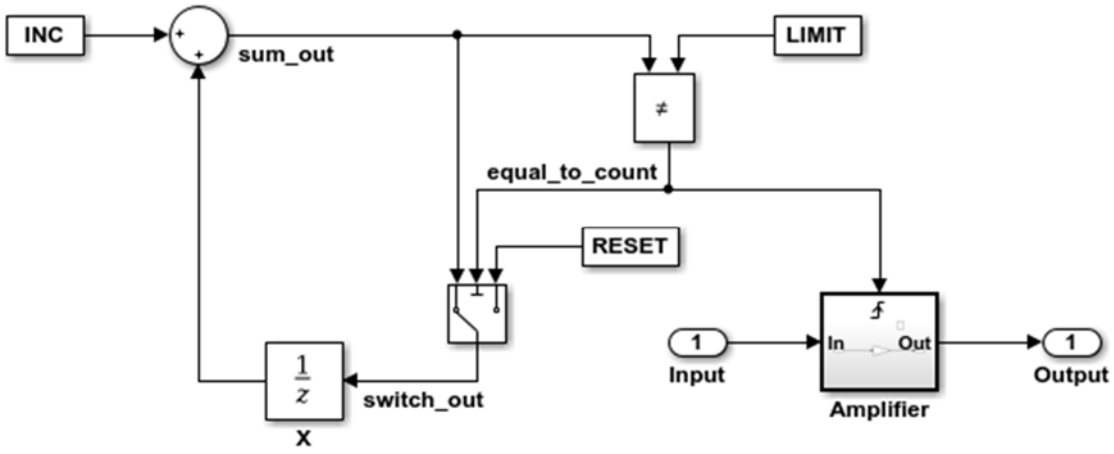


Figura 13 Diagrama en Simulink [23]

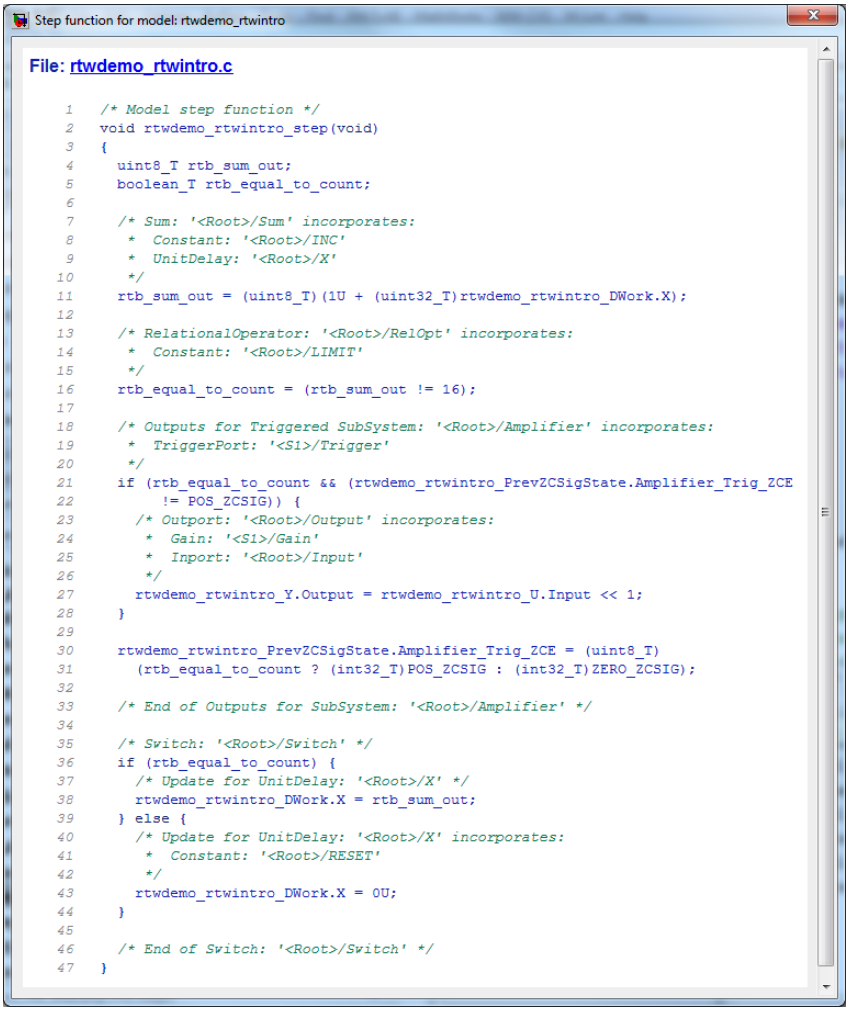


Figura 14 Código generado del diagrama de la Figura 13 [23]

La NASA ha incrementado el uso de éste generador de código, es uno de los más populares de la institución. [24]

### 2.4.2 SCADE

SCADE Suite KCG es un generador de código C de modelos Scade que ha sido calificado como una herramienta de desarrollo para software DO-178B hasta nivel A. Este generador ahorra esfuerzo de verificación en la fase de codificación, como revisiones de código y pruebas de bajo nivel. Esta mejora reduce tiempo y esfuerzo para la certificación y/o modificación del software ya que disminuye el tiempo de verificación

del mismo. SCADE Suite KCG ha aprobado con éxito el procedimiento de calificación en varios programas grandes y se utiliza actualmente en la producción para una serie de programas en Europa, Asia y América.

Las propiedades de un código generado por SCADE son las siguientes:

- Cumple con las restricciones de código embebido tales como bucles limitados estáticos y no recursión.
- Código C y Ada de alta calidad y seguridad: optimizado, personalizable, legible y rastreable.
- No tiene secciones de código muerto.
- Es código portable. [25]

La Figura 15 muestra un ejemplo de Código C generado por SCADE.

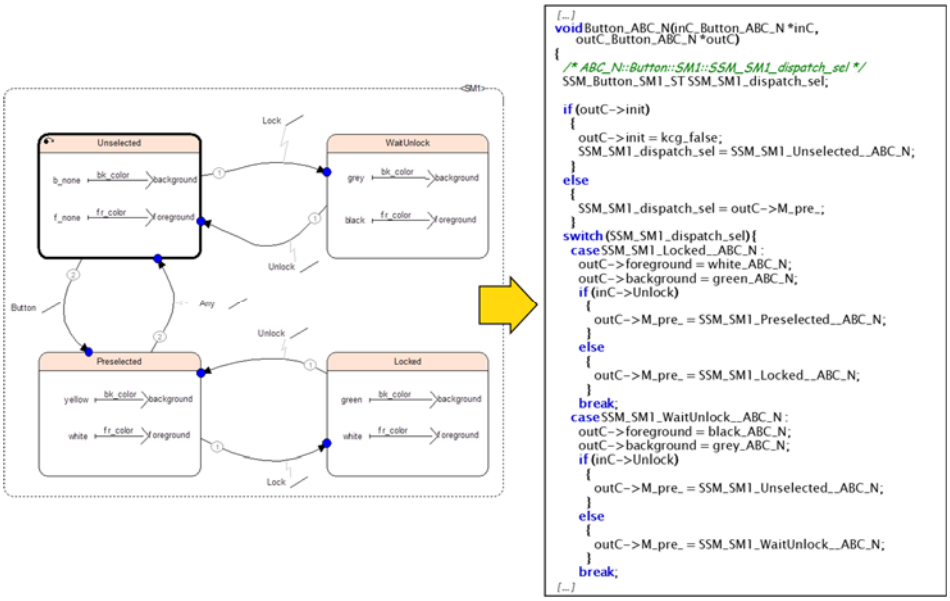


Figura 15 Ejemplo de Código C generado por SCADE. [25]

Los siguientes sistemas son ejemplos en donde se aplica el código generado por SCADE en algunas aeronaves:

- Sistema de referencia de datos aéreos.
- Pilotos automáticos.
- Sistemas de frenado y tren de aterrizaje.
- Presión de la cabina y control climático.
- Sistemas de control del motor.
- Gestión del combustible
- Controles hidráulicos
- Inversores de empuje [25]

### 2.4.3 BEACON

BEACON es un conjunto potente de herramientas de ingeniería de software enfocada en el diseño, implementación, prueba y mantenimiento para sistemas embebidos de alta integridad, incluidos los desarrollados por RTCA / DO-178B. El diseñador gráfico de BEACON y los generadores de código múltiples trabajan juntos para evitar que muchos tipos de errores de software alcancen el código resultante.

BEACON ha sido utilizado por más de una década en la industria aeroespacial comercial, cada cinco segundos un avión despega en alguna parte en el mundo con código generado por BEACON funcionando en los sistemas del control del motor [26].

BEACON proporciona una solución global de ciclo de vida que se basa en el diseño, no en el lenguaje de código o en el procesador integrado, simplificando el desarrollo y facilitando la reutilización.

Para el diseño e implementación de lógica BEACON tiene diferentes tipos de diagramas que comparten un conjunto de información común. El alcance y la visibilidad de cada elemento compartido y tipo de dato son controlados explícitamente por el diseñador. Los tipos de diagramas y la ventana de datos se describen a continuación.

- Diagramas de señal

    - Filtros dinámicos.
    - Bloques matemáticos lineales y no lineales.
    - Tablas multidimensionales.
    - Lógica booleana.
- Diagramas de flujo
    - Declaraciones - pseudocódigo o código real.
    - Decisiones - ramas múltiples y bucles.
    - Tablas de verdad.
- Ventana de datos
    - Visibilidad y alcance local y externo.
    - Calificadores de datos: constante, estático, externo, etc.
    - Escala de punto fijo.
    - Valores mínimos y máximos.
    - Comentarios especificados por el usuario.

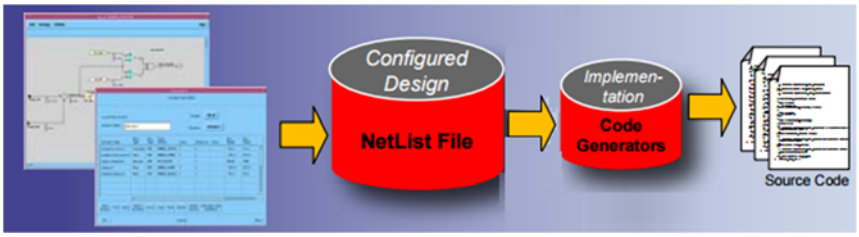


Figura 16 Representación gráfica de la generación de código en BEACON.

BEACON se utiliza para crear una especificación de software clara, rastreable y fácil de revisar utilizando resaltados gráficos, campos de comentarios de código y varios mecanismos de impresión. Este generador de código analiza el diseño antes de generar código para asegurar que es explícito, coherente, seguro y que satisface las prácticas de programación seguras.

Genera software eficiente que cumple con las mejores prácticas y estándares de programación. El código es independiente del compilador y puede ejecutarse en

plataformas host, estaciones en tiempo real o microcontroladores integrados. Los siguientes son los lenguajes soportados. [26]

- ANSI-C
- Ada-83/95
- SPARK-83
- FORTRAN-77

## 2.5 COMPLEJIDAD DEL SOFTWARE

En cualquier sistema, la complejidad es directamente proporcional a los elementos que lo componen, y a su vez, a la de cada uno de estos elementos, ésta es parte esencial de los sistemas de software, en especial de aquellos que su funcionalidad demanda un gran tamaño, en la mayoría si no es que en todos estos casos la complejidad se puede manejar, pero no se puede eliminar.

Grady Booch, afirma que la complejidad del software se desprende de los siguientes factores.

1. **La complejidad del dominio del problema.** Los problemas que se intentan resolver son inherentemente complejos, con una gran cantidad de requisitos, en un sistema grande estos requisitos deben evolucionar para evitar que el sistema sea reemplazado, debido al gran costo financiero que implica el remplazo.
2. **La dificultad de gestionar el proceso de desarrollo.** Los desarrolladores de software enfrentan el reto de dar a los usuarios la impresión de simplicidad, es decir, reducir al mínimo la complejidad en la interacción sistema-usuario. Este reto les obliga a incrementar el tamaño de los sistemas.
3. **La flexibilidad que se puede alcanzar a través del software.** La elaboración de software es una actividad muy laboriosa porque empuja al desarrollador a construir por sí mismo prácticamente todos los bloques fundamentales sobre los que se apoyan los requerimientos de más alto nivel.

4. **Los problemas de caracterizar el comportamiento de sistemas discretos.** Los comportamientos de la mayoría de los objetos se representan por sistemas analógicos en los que, a través de funciones continuas, pequeños cambios en las entradas siempre producen pequeños cambios en las salidas. Por el contrario, puesto que el software se ejecuta en computadoras digitales, se tienen sistemas con un número finito de estados discretos. En sistemas grandes, este número puede crecer a cantidades enormes. Como no existen herramientas matemáticas para modelar el comportamiento completo de los grandes sistemas discretos, se debe aceptar la pérdida de cierto grado de confianza en cuanto a que las salidas sean correctas. [27]

Tomando en cuenta estos factores, se puede decir que la complejidad del software es el grado de dificultad de analizar, diseñar, implementar y mantener el software, este grado se va agregando durante todo el ciclo de vida del software.

Otra forma de ver la complejidad del software, es analizar los recursos que necesita el sistema y qué tan eficiente debe ser, estos son principalmente tiempo y espacio, por ejemplo horas-hombre y memoria en la computadora. Cuando podemos medir cómo y cuándo ésta complejidad se va agregando al proyecto, sabemos en qué etapa del ciclo de vida del software nos tenemos que enfocar para modificar, por ejemplo, el método de codificación o el plan de pruebas , y de esta forma, poder reducir la complejidad del software. [28]

En resumen, la complejidad del software está en función de lo complejo de la situación a resolver, el grado de dificultad del algoritmo que se diseñó para solucionar dicho problema y de lo complicado que pudiera ser la estructura que se utilizó para implementar el algoritmo.

## 2.6 ESTADO DEL ARTE

La medición de la complejidad del Software permite identificar si el código es o no complejo. En el desarrollo del mismo, como en otros aspectos de la vida cotidiana, "no se puede controlar lo que no se puede medir" [29]. Las métricas de la complejidad es lo primero que nos podría llevar a controlar dicha complejidad.

Las métricas en la complejidad del software no son más que medidas cuantitativas de ciertas características del producto, estas métricas pueden medir: [29]

1. Productos (código o documentación).
2. El proceso de desarrollo (aspectos de las actividades del desarrollo).
3. El dominio del problema.
4. Características del ambiente (personas, herramientas).

Por supuesto que para cada uno de los casos mencionados, la métrica será diferente incluso hablando del mismo software, la complejidad ligada a la característica del producto es uno de los más estudiados especialmente aquellas métricas que se enfocan en el código.

Existen factores que influyen en el mantenimiento del software como tamaño del programa, estructuras de los datos, flujos de datos y flujo de control [30], estos factores podrían ser clasificaciones más generales de las métricas, ésta clasificación es dependiendo de lo que las mismas intentan cuantificar.

A continuación se describen los principales tipos de métricas.

### 2.6.1 Métricas Basadas En El Tamaño Del Programa

Pareciera obvio que entre más grande sea el tamaño del software, éste se vuelve más complejo de codificar, probar y mantener.

**Números De Líneas**

Esta metodología LOC o SLOC por sus siglas en inglés mide la complejidad del código en función del número de líneas funcionales o de sentencia dentro del programa, La principal limitante en este método es el uso del tamaño de software como el único factor de complejidad. Es probable que cuanto mayor sea el tamaño de un programa, más defectos podría haber. Las métricas LOC representan algunos aspectos de la complejidad del software, pero es un método aproximado que no proporciona la fidelidad necesaria para una evaluación de fiabilidad del software. [31]

**Métrica De Halstead**

Una forma más precisa para medir el tamaño de un programa fue propuesta por Healdstead, este método mide la complejidad de un programa contando el número de operadores y operandos, digamos que:

$n_1$ = Número de operadores distintos
$n_2$ = Número de operandos distintos
$N_1$ = Número de ocurrencias de los operadores
$N_2$ = Número de ocurrencias de los operandos

Basado en estos componentes, Halstead estableció una fórmula que se compone de vocabulario, longitud y volumen. Se definen como.

Vocabulario: $n = n_1 + n_2$
Longitud: $N = N_1 + N_2 = n_1 \log_2 n_1 + n_2 \log_2 n_2$
Volumen: $V = N \log_2 n = N \log_2 (n_1 + n_2)$

Este método no es del todo correcto ya que la métrica no considera los ciclos y el flujo de información que por supuesto, intensifican la complejidad. Es decir, se puede decir que dos programas con las mismas líneas y el mismo valor de Halstead tienen la misma

complejidad. Sin embargo, uno podría tener códigos secuenciales rectos, mientras que el otro podría tener ciclos muy anidados y comunicación de información complicada. [31]

### 2.6.2 Métricas Basadas En La Estructura Del Programa y El Flujo De Datos

Otro factor relevante en la complejidad del software está dado por la manera en cómo interactúan los datos dentro del programa. Igual que con otros factores, es más complejo un sistema en el que los datos se manejen de una forma más rebuscada.

#### Métrica De Flujo De Información

Cuando se trata de complejidad de software, debemos considerar otro factor: la cantidad de flujo de información entre módulos. El método más utilizado es el de la métrica de *fan-in* y *fan-out*. Ésta métrica de flujo de información está definida como:

$C = (fan\text{-}in * fan\text{-}out)^2$

Donde *C* representa la cantidad de flujo de información en un módulo, *fan-in* es entradas de un módulo y *fan-out* es la cantidad de valores de retorno y variables modificadas. [31]

#### Diagrama De Funciones

Ésta metodología está basada en la anterior, considerando una función como un módulo y un sistema como un todo. La complejidad del flujo de información considera una estructura de datos como un módulo sin pensar en la relación de los diferentes módulos. Si se quiere calcular la complejidad de flujo de información de un programa, lo único que se hace es agregar los módulos sin tener en cuenta la conexión entre ellos.

El flujo de datos se vuelve más complicado debido a las llamadas a funciones. Por lo tanto, es importante analizar el gráfico de llamadas de función para ilustrar la relación

de flujo de datos. El propósito de ésta metología también es calcular la complejidad del flujo de información, así que contamos las veces que nuestra función manda a llamar a otra. Un ejemplo de gráfico de llamadas de función se muestra en la Figura 17 Los dígitos 2 y 1 en la parte derecha significan FunctionA llama a FunctionB dos veces y llama a FunctionC una vez.

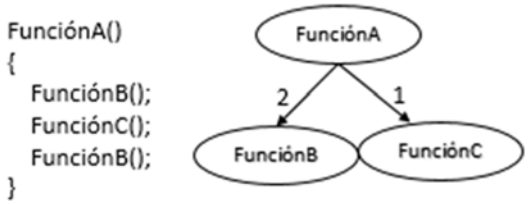


Figura 17 Método para dibujar El Diagrama De Funciones.

Después de identificar el número de llamadas en nuestra función, se considera lo siguiente:

1. Calcular el flujo de información de cada procedimiento sin considerar su posición en el gráfico de llamadas de función según la fórmula $C = (fan\text{-}in * fan\text{-}out)^2$.
2. Desde abajo hasta arriba del árbol de llamada de función, la complejidad de flujo de información de un procedimiento es la complejidad de flujo de información de sí mismo más la complejidad de flujo de información de su sub-procedimiento.
3. La complejidad del procedimiento en la parte superior del gráfico de llamadas de funciones es la complejidad del flujo de información del sistema. [32]

En resumen, y siguiendo el ejemplo de la Figura 17, la complejidad del sistema quedaría de la siguiente forma.

$$C_{Sistema} = (C_{FunciónA})+(2*C_{FunciónB}) + (1*C_{FunciónC})$$

**Intervalo Entre Referencia De Datos**

Un intervalo entre referencia de datos es el número de sentencias que hay en el programa entre dos referencias inmediatas a la misma variable, vea la Figura 18, entonces, la métrica de intervalo entre referencia de datos es contar los rangos entre referencias que sean mayores que un cierto rango [33]. Ésta métrica podría darnos una idea de la complejidad de los datos, es decir, un programa con el 10% de los intervalos mayores a 50 sentencias es más complejo a otro con sólo el 3% de éstos mismos de esas características.

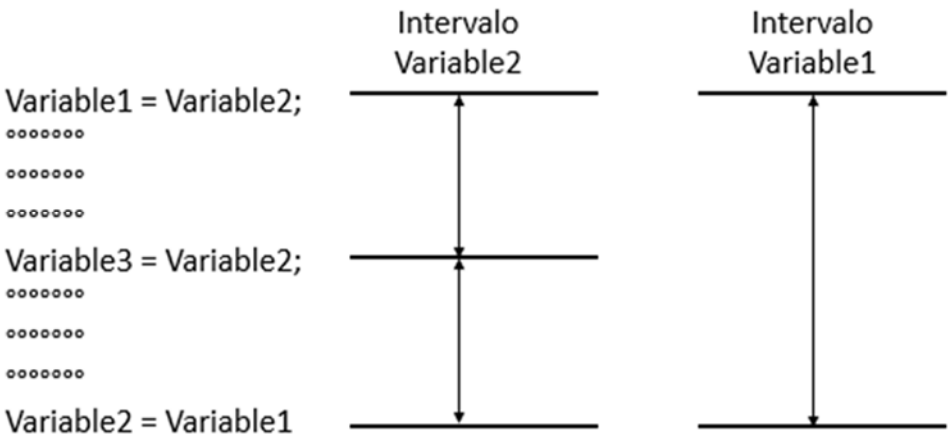


Figura 18 Intervalos entre referencias de datos.

**Par De Uso Segmento-Global**

Ésta métrica mide la cantidad de veces que una función arbitraria de un programa accede a una variable global. Dado una función de programa *p* y una variable global *r* el par de segmento global es (*p,r,)* en otras palabras, la función *p* usa la variable r. El par de uso real (AUP) por sus siglas en inglés, es el número de veces que un módulo o función usa la variable global, el par de uso posible (PUP) por sus siglas en inglés, es el número de veces que un módulo podría acceder a una variable global y finalmente el porcentaje relativo de uso real (RUP) es la relación AUP/PUP, es decir, RUP = AUP/PUP, y ésta es la medición aproximada de cuánto se usan datos globales dentro de una función arbitraria de código [33].

Si tenemos un programa con tres módulos, funciones o segmentos y con cuatro variables, y si el ámbito de las cuatro variables incluye a los tres procedimientos, es decir, las cuatro variables podrían ser accedidas en las tres funciones, tenemos un par de uso potencial PUP=12, si la primera función sólo llama una variable, la segunda cuatro y la tercera sólo una, entonces tenemos un *AUP=3+4+1=8*, finalmente para éste ejemplo resulta un *RUP=8/12=2/3*.

**Medida Q De Chapin**

En ésta metodología, los datos son tratados según su uso dentro de cada módulo, además de ser categorizados de la siguiente manera [34]:

1. Tipo P: Datos de entrada necesarios para que el módulo considerado produzca una salida.
2. Tipo M: Datos que se crean o son modificados dentro del módulo.
3. Tipo C: Datos que se usan para ejercer un papel de control dentro del módulo.
4. Tipo T: Los que pasan a través del módulo sin ser modificados.

Ya que un mismo dato puede usarse de diferentes maneras dentro del módulo, se contará una vez en cada categoría a la que pertenece. En ésta metodología se considera que no todos los tipos de datos descritos contribuyen de igual forma a la complejidad del software. Los tipo C (de control) son los que más complejidad agregan al sistema ya que deciden qué módulos serán invocados. Los datos de tipo M contribuyen menos a la complejidad que los tipo C pero siguen contribuyendo a que su valor esté inicialmente definido o modificado. Los tipo P que suelen ser usados para modificar los M también agregan complejidad, aunque en menor proporción a los anteriores. Finalmente, los datos tipo T casi no contribuyen a la complejidad, ya que sólo se “leen” en el módulo [34].

Dicho lo anterior, se asigna un factor de ponderación diferente a cada uno de los tipos de datos, la Tabla 2 muestra el factor de ponderación apropiado sugerido por Chapin para explicar la complejidad que aporta cada tipo de dato.

| Tipo de Dato | Factor |
|---|---|
| C | 3 |
| M | 2 |
| P | 1 |
| T | 0.5 |

Tabla 2 Factores de Chapin

Los pasos para aplicar la metodología, se pueden resumir de la siguiente forma:

1. Se crea una tabla donde con las variables de cada módulo que usa para comunicarse con otros módulos, junto con sus tipos, en esta tabla se especifica las que tienen fuente, o bien que recibe y que sean de tipos C, P, además las que tienen destino, es decir las que otros reciben de éste módulo y que son de tipos M o T. Por supuesto, una variable puede estar en los dos lados de la tabla.
2. Cada elemento identificado en la tabla anterior, se multiplica por el factor de Chapin según su tipo. Estos productos ponderados se suman a continuación para cada módulo, produciendo una medida intermedia W'.
3. Se calcula el factor de repetición (R), que tiene en cuenta el incremento de complejidad debido a la comunicación de datos entre módulos que se llaman iterativamente dentro de un ciclo. Se calcula de la siguiente manera:
    I. Determinar qué módulos contienen ciclos con condiciones de control que incluyen la llamada de más de un módulo.
    II. Para cada dato de tipo C que haya en cada uno de los módulos encontrados, cuyo valor venga de fuera del cuerpo del ciclo, sumaremos 2 al factor de salida de iteración (E), que ha sido previamente inicializado a 0 para cada módulo. Si el dato de tipo C es creado o modificado en otro segmento que no sea en el que está la condición de control, pero que está

todavía dentro del ámbito de la iteración, sumaremos 1 a E, una vez calculado E, el parámetro R vendrá dado por:

$$R = (\frac{1}{3} * E)^2 + 1$$

4. El índice de complejidad de cada módulo (Q), se calcula a partir de los valores de R y W', es la raíz cuadrada de la suma de los productos ponderados multiplicado por el factor de repetición.

$$Q = \sqrt{W' * R}$$

La complejidad del programa entero se calcula como la media aritmética de las complejidades individuales de los segmentos. [34]

### 2.6.3 Métricas Basadas En La Estructura De Control Del Programa

Definitivamente, la estructura de control que sigue la ejecución de un programa influye en la complejidad del mismo, es lógico saber que entre más decisiones tenga que evaluar la lógica para saber qué camino seguir, ésta se vuelve más difícil de analizar o entender.

**Número Ciclomático**

Ésta metodología propuesta por McCabe mide la complejidad de un programa tomando como referencia su grafo de control, la métrica de éste método está basado en el número ciclomático V(G) que se define para un grafo dado, suponiendo que se tiene un grafo que corresponde al flujo de control de un programa, con **e** arcos, ***n*** nodos y ***c*** componentes conectados (normalmente, c es 1). La complejidad ciclomática de ese programa vendrá dada por la siguiente fórmula [34]:

$$V(G) = e - n + 2 * p$$

El número ciclomático es el número mínimo de caminos necesario para construir cualquier otro camino presente en el grafo mediante combinaciones, entendiendo como camino una sucesión de nodos que puede recorrerse siguiendo arcos (o conexiones entre nodos) presentes en el grafo. [34]

Por ejemplo, en el grafo dirigido de la Figura 19 donde e = 12, n = 10 se tiene una complejidad ciclomática de 4, es decir, *V(G) = 12 – 10 + 2 * 1*.

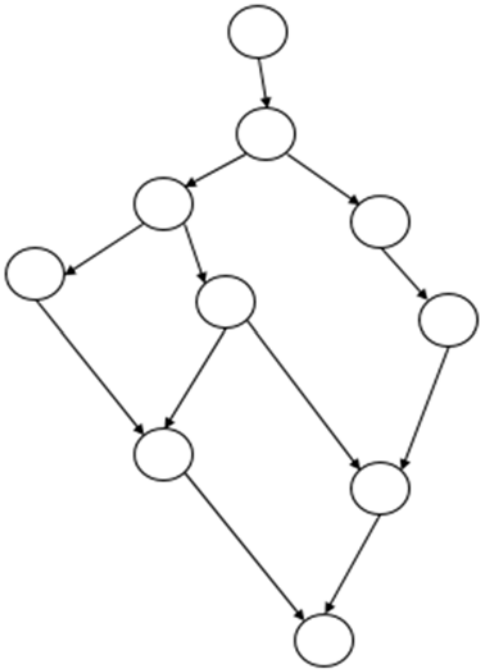

Figura 19 Grafo dirigido.

**Extensión De Myers Al Número Ciclomático**

La extensión de Myers al número Ciclomático consiste en tomar en cuenta si una decisión es simple, es decir, sólo tiene una sola condición, o bien, si es compleja (con más de una condición), ésto se ilustra en la figura Figura 20 (a) y (b) respectivamente, como ambos segmentos de código solo involucran una decisión, tiene el mismo grafo dirigido y por lo tanto, el mismo número ciclomático. [34]

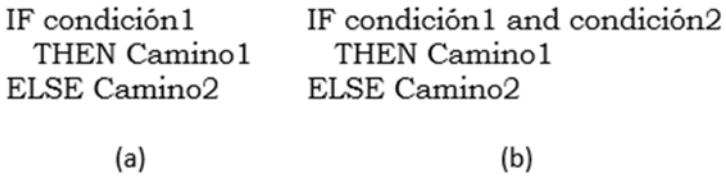


Figura 20 Segmentos de código.

La metodología de Myers consiste en tomar en cuenta las diferencias mencionadas en el párrafo anterior, obteniendo así, la complejidad como un intervalo, y no como un simple número. Para esto, se toma como límite inferior del intervalo el número de decisiones más uno, y como límite superior, el número total de condiciones individuales en cada decisión, también más uno [34]. Tomando en cuenta lo anterior, el intervalo de Myers para Figura 20 (a) quedaría [2,2] y [2,3] para el segmento (b) de la misma figura.

### 2.6.4 Métricas Mixtas

Las metodologías documentadas hasta ahora sólo miden una parte de los aspectos que contribuyen a que un software sea complejo, pero la mayoría de las veces es necesario tomar en cuenta más de una característica para calcular la métrica que proporcione la complejidad de un programa. Para esto, hay que considerar varias propiedades del código como se verá en los métodos siguientes.

**Métrica De Hansen**

Hansen desarrolló una metodología basada en la combinación del número ciclomático de McCabe con una medida del número de operandos, ésta métrica es un par ordenado *(m1,m2),* donde *m1* y *m2* se definen de la siguiente manera:

**m1**: Es el número de sentencias alternativas como por ejemplo *IF* y *CASE* o iterativas tales como DO..WHILE.

**m2**: Es el número de operadores del programa, algunos de ellos son los siguientes:

- Operadores primitivos (+, -, *, AND, SUBSTR, etc.)
- Asignaciones.
- Subrutinas o llamadas a funciones. [34]

Así pues, se caracteriza los fragmentos de código con dos números que dan una idea de la complejidad de su flujo de control (m1) y de la cantidad de información total que contiene (m2).

**Métrica De Obviedo**

Obviedo propone medir la complejidad en función al flujo de datos y al flujo de control mediante la siguiente formula:

$$C = aC_f + bD_f$$

Donde:

$C_f$ = Complejidad del flujo de control.
$D_f$ = Complejidad del flujo de datos.
a = Factor de peso para la complejidad del flujo de control
b = Factor de peso para la complejidad del flujo de datos.

Para calcular la complejidad $C_f$ basta con contar los arcos o conexiones en el grafo dirigido del programa a analizar, sin embargo, para calcular $D_f$, el proceso no es tan sencillo. Se basa en el concepto de “variable localmente expuesta”, estoes, las variables cuyo valor es utilizado en ese segmento de código mediante asignaciones, sentencia de entrada o cualquier otra forma, pero que éste valor haya sido adquirido en otro segmento anterior. Entonces, $D_f$ se calcula a partir del número de posibles "adquisiciones" de valor que han podido tener las variables localmente expuestas de cada uno de los módulos. [33]

### 2.6.5 Otras Métricas

Existen muchas otras metodologías para medir la complejidad de un software además de las ya mencionadas, a continuación se mencionan los objetivos principales para algunas de ellas.

- **Módulo de Coupling**: Mide las relaciones entre módulos.
- **Fortaleza del módulo**: Una medida de qué tan relacionados están los elementos dentro de un módulo.
- **Flujo de información integrado de Rombach**: Una medida de la complejidad intermodular e intramódulo basada en el flujo de información
- **Nudos:** El número de líneas cruzadas (declaraciones GOTO no estructuradas) en un gráfico de flujo de control.
- **Complejidad lógica relativa:** El número de decisiones binarias dividido por el número de declaraciones. [35]

### 2.6.6 Complejidad De Pruebas Unitarias

Si bien existen muchas metodologías para la medición de la complejidad del software, no existe alguna metodología documentada para la medición de la complejidad de pruebas unitarias, incluso no las hay para ningún tipo de prueba de software, al menos no se encontró ninguna en la investigación para la realización de éste proyecto.

Es cierto que muchas empresas desarrolladoras de software o de pruebas para el mismo, usan como base alguna métrica de las mencionadas anteriormente, pero como ya se dio a notar, todas están enfocadas solamente en elementos como el tamaño de la lógica o el flujo de datos, por lo tanto no proporcionan una buena referencia para la complejidad de la prueba y más aún si el proceso de las pruebas unitarias no está automatizado, ya que éstas metodologías no involucran factores relevantes de las pruebas unitarias como son:

- Técnicas para desarrollar casos de prueba

- Documentación de las pruebas unitarias.
- Ambiente de pruebas.
- Ambiente de simulación de la lógica.

Si se usa, por ejemplo, la métrica de la complejidad ciclomática descrita anteriormente, ésta podría ser no muy útil, ya que talvez, por algunas técnicas definidas para el diseño de casos de prueba, se deban crear y ejecutar un número finito de casos de prueba para un módulo con ésta métrica relativamente baja. También, si usamos el método de número de líneas de la lógica, previamente estudiado, podía arrojarnos una complejidad alta para la prueba, sin embargo, la prueba unitaria podría ser relativamente fácil si sólo se tuviera qué diseñar un solo caso de prueba si el proceso y las técnicas de diseño de casos de la misma lo permiten, suponiendo que la lógica a probar es sólo la inicialización de cientos de parámetros.

Debido a lo mencionado en el párrafo anterior, surge la necesidad de generar una metodología que involucre los factores descritos en éste proyecto, además de otros que pudieran ser importantes para un proceso de pruebas ya definido.

# CAPÍTULO 3. PROCEDIMIENTO

El procedimiento para el desarrollo y diseño de este proyecto consta de tres etapas principales, las cuales se describirán a continuación.

## 3.1 ANÁLSIS DEL PROCESO DE PRUEBAS UNITARIAS

Debido a que la metodología propuesta se va a aplicar a un proceso de pruebas unitarias ya definido, es necesario analizar los aspectos de éste mismo. Principalmente se estudiaron elementos tales como ambiente, procedimiento y herramientas, con el objetivo de identificar aquellos factores que pudieran influir en el desarrollo del proyecto.

### 3.1.1 Ambiente Del Proceso

Se hizo un análisis del ambiente en el cual se desarrollan las pruebas unitarias y se encontró que es una mezcla entre Unix y Windows, pero el proceso principalmente es ejecutado en Unix; en ésta plataforma se encuentran los servidores en los cuales están almacenados las unidades a probar y casi todas las herramientas utilizadas en los procedimientos. Así mismo, en Unix también se encuentra el simulador de la lógica de las unidades y las licencias del mismo, además de la herramienta que se encarga de enviar cada caso de prueba al simulador y analizar los resultados obtenidos en la simulación. En Windows, sólo existen herramientas de ayuda para la prueba unitaria.

Los diagramas de las unidades son implementados con el generador de código BEACON que ya estudiamos anteriormente, éstos están compuestos por bloques, los cuales interactúan entre sí para generar el código deseado de acuerdo a la implementación. El código que genera cada uno de los bloques es conocido y documentado como referencia de bajo nivel para la prueba unitaria. Los diagramas son archivos con extensión .net que además de poder ser vistos de una forma gráfica, también puede ser analizado en modo texto, conservando los atributos con el cual fue diseñado.

### 3.1.2 Procedimiento

Existen estándares que rigen el proceso de pruebas unitarias, los procedimientos definidos también deben de estar alineados a éstos estándares. A continuación, se mencionan los pasos que se deben seguir para llevar a cabo el procedimiento.

1. Análisis de la lógica: Se hace un análisis de la lógica para identificar todas las coberturas que se le deben hacer, estas coberturas están definidas y documentadas en los estándares previamente definidos.
2. Diseño de casos de prueba: Una vez que se identificaron las cuberturas aplicables, se crean casos de prueba usando técnicas de diseño basadas en cada una de las coberturas. En éste paso se calculan las salidas de la lógica para cada caso de prueba obteniendo, así lo que en pruebas unitarias son conocidos como valores esperados.
3. Documentación de la prueba: Se documentan todas y cada una de las coberturas aplicables en la unidad mencionando para cada una en qué caso de prueba fue alcanzada.
4. Simulación: Se hace la simulación de la unidad utilizando los casos de prueba previamente diseñados y el código de la lógica real para obtener los valores actuales o bien los valores para cada salida ejecutando la lógica real.
5. Análisis de resultados: Se realiza un análisis comparativo de los valores actuales y los valores esperados para asegurar que sean iguales para cada caso de prueba.
6. Revisión Cruzada: Un ingeniero de pruebas diferente al que realizó la prueba unitaria hace una segunda revisión para asegurar el diseño funcionalidad y calidad de ésta misma. Esto es debido a que éste proceso es aplicado a software de nivel A [5].

En la figura Figura 21 se pueden apreciar los pasos mencionados.

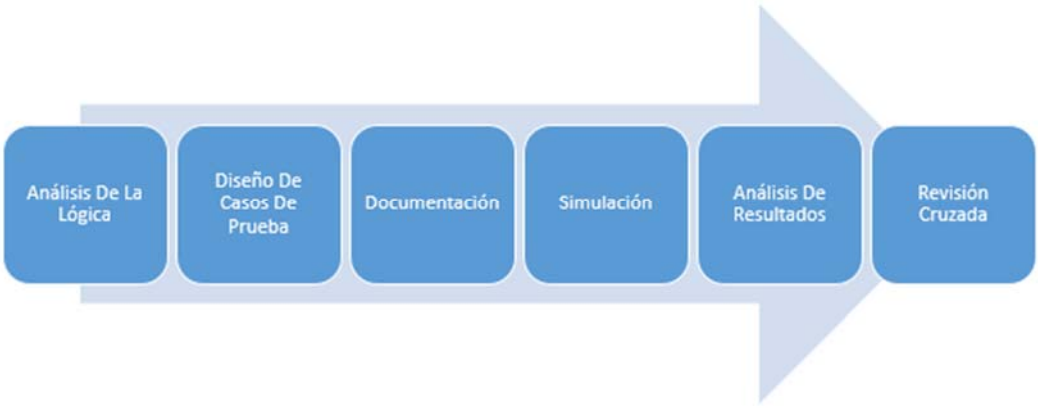


Figura 21 Procedimiento De La Prueba Unitaria

### 3.1.3 Herramientas

Existen herramientas dentro del proceso que ayudan a automatizar o semi-automatizar algún paso o sub-procedimiento, las herramientas encontradas caen en los siguientes tipos.

- Diseño de casos de prueba:
  - Existe una herramientas que sugieren casos de prueba para un bloque de BEACON, ésta herramienta permite al implementador de la prueba tomar ventaja de esta sugerencia.
  - Otra herramienta ayuda al implementador a mantener un orden al momento de diseñar los casos de prueba, de forma automática clasifica las entradas y salidas de la lógica para facilitar su cálculo.
- Documentación:
  - Una herramienta ayuda a la generación de comentarios de forma semi-automática.
  - Existe otra herramienta que facilita la documentación de las coberturas que aplicaron a la unidad probada.

Aunque existen un conjunto de herramientas para facilitar la ejecución del proceso, en realidad muchas de ellas todavía tiene que ser mejoradas o bien existen lógicas que exceden su capacidad, lo cual hace que éstas muchas veces sean de poca ayuda. Sin

embargo, se consideró éste análisis muy importante ya que este tipo de ayudas influyen directamente en la complejidad de la prueba.

## 3.2 PONDERACIÓN DE COMPLEJIDADES

Entre varios expertos en el proceso se hizo un análisis de todos y cada uno de los bloques en el caso de los diagramas de señal y operadores para los diagramas de flujo en BEACON permisibles en las lógicas para darle a cada uno de ellos un peso específico, el cual representaría su complejidad para éste proceso de pruebas unitarias. Se usará la Figura 22 de manera ilustrativa como un diagrama BEACON, ésta imagen muestra un diagrama con lógica de señal; cabe mencionar que BEACON tiene diagramas para lógica de señal y para lógica de flujo.

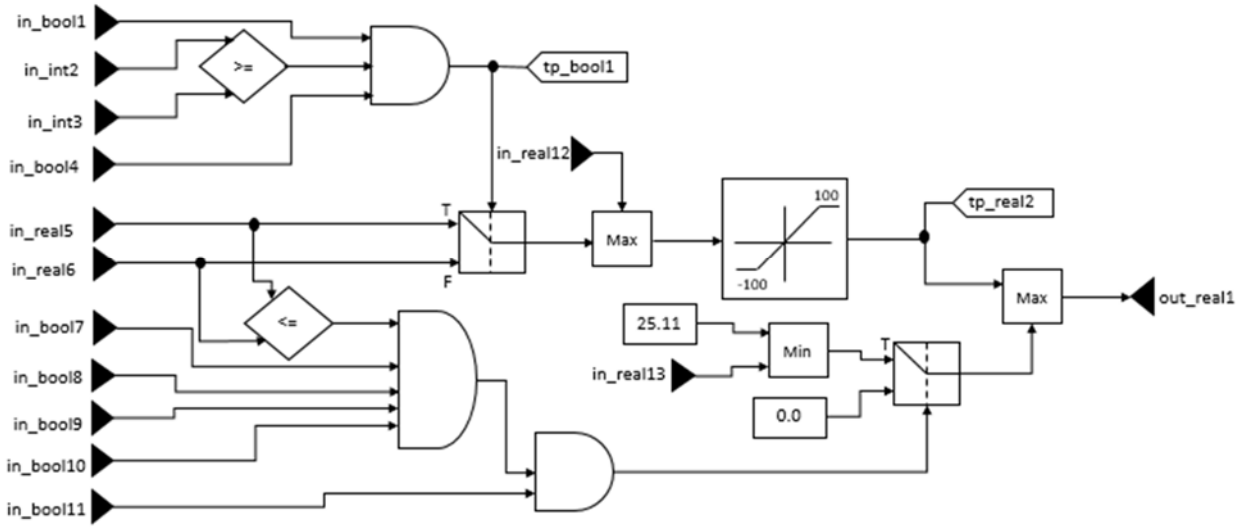


Figura 22 Imagen ilustrativa de un diagrama de BEACON.

Para definir éste peso específico se consideraron los siguientes factores en cada una de las unidades:

**Lógica de flujo o lógica de señal:** Debido a que las herramientas procesan de forma diferente los tipos de lógicas se debe tomar en cuenta ésta diferencia al momento de

calcular la complejidad, incluso hay herramientas que sólo procesan uno u otro tipo de lógica y esto implica menos o más ayuda para el diseño de la prueba unitaria.

Estas ponderaciones se asignaron tomando en cuanta poca experiencia del implementador, es decir, se le dio un peso mayor a un bloque que requiere un análisis más profundo de su lógica interna que a otro que no requiere un análisis exhaustivo de su lógica, en otras palabras es más complejo diseñar tres casos de prueba para el primer bloque que diseñar tres casos de prueba para el segundo.

**Número de entradas a la unidad:** Entre más entradas tenga la unidad más consideraciones se deben tener para el diseño de casos de prueba. La Figura 23 muestra las entradas a la unidad de ejemplo para éste proyecto.

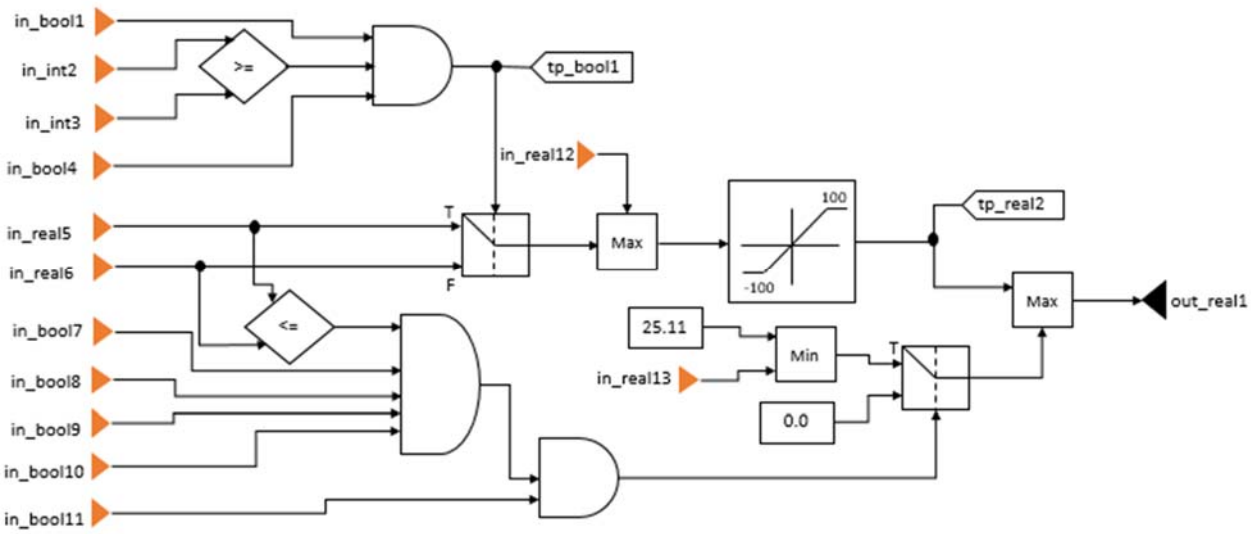


Figura 23 Entradas a la unidad

**Número de salidas de la unidad:** Entre más salidas tenga la unidad es más fácil reflejar directamente en una de ellas el valor estándar de una cobertura aplicada. La Figura 24 resalta la única salida de nuestro ejemplo.

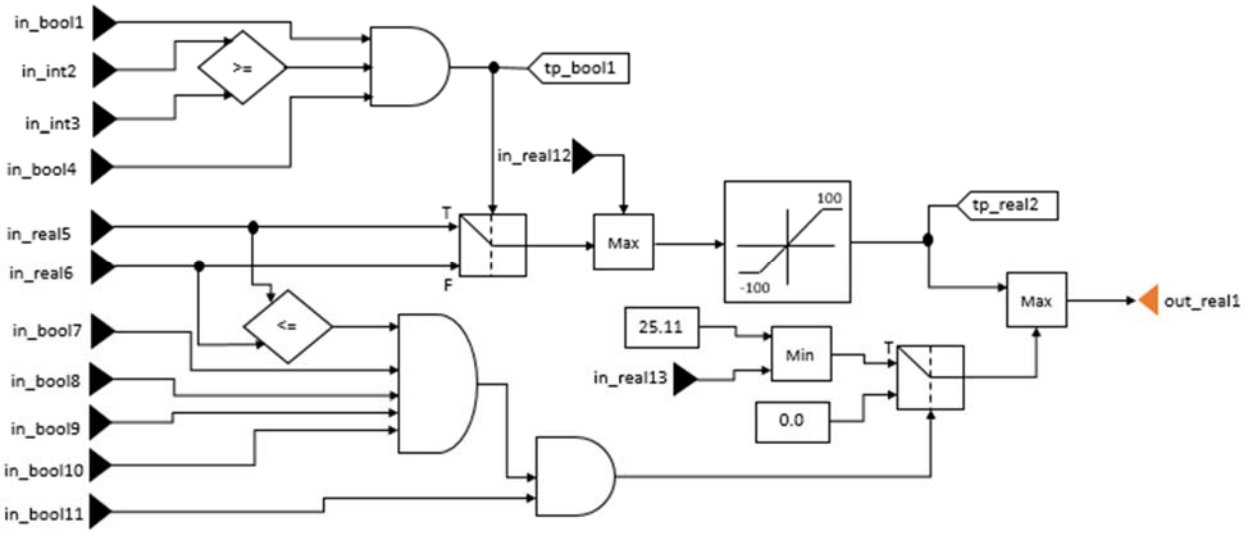


Figura 24 Salidas de la unidad.

**Salidas intermedias (test points) en la unidd:** Este punto está totalmente relacionado con el anterior. Vea la Figura 25 para identificar las salidas intermedias o también llamados puntos de prueba. Para diagramas de flujo éstos puntos de prueba son parámetros globales que se modifican en un punto intermedio de la lógica.

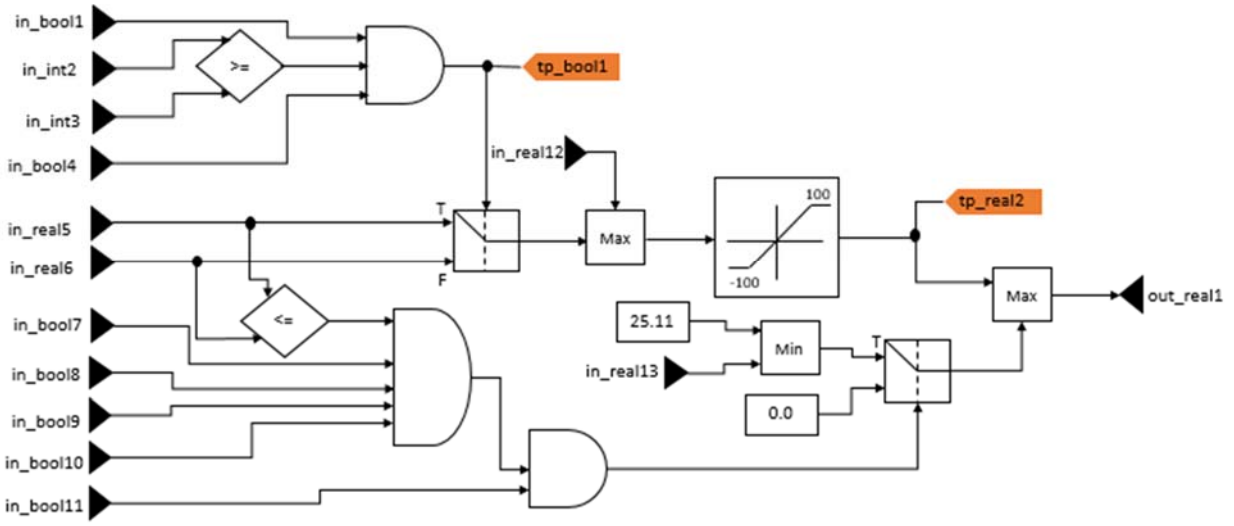


Figura 25 Salidas Intermedias o puntos de prueba

Además de hacer esa ponderación por unidad, también se agregó un peso específico por cada uno de los bloques el cual fue calculado considerando lo siguiente:

**Número de entradas a un bloque:** Cada bloque se vuelve más complejo a mayor número de entradas que interactúen con éste, la Figura 26 muestra resaltadas las entradas al bloque AND.

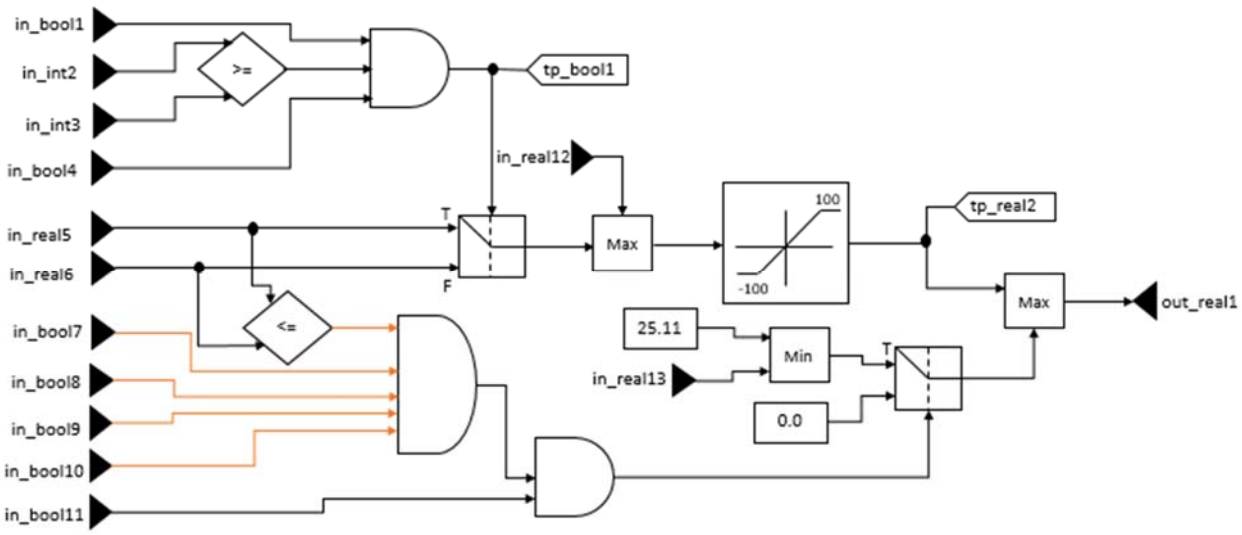


Figura 26 Entradas a un bloque

**Número de salidas de un bloque:** Al igual que con el número de entradas, el número de salidas a calcular para cada bloque hace más elaborada la prueba. Aunque en el ejemplo de la Figura 27 sólo hay bloques con una sola salida (resaltada con rojo), es decir, cada compuerta lógica o cada bloque MAX por mencionar algunos, sólo tienen un valor por calcular, en BEACON existen bloques para los cuales esto no se cumple; son bloques que además de la salida hay valores intermedios e internos que se tendrían que calcular para poder realizar la simulación de dicho bloque.

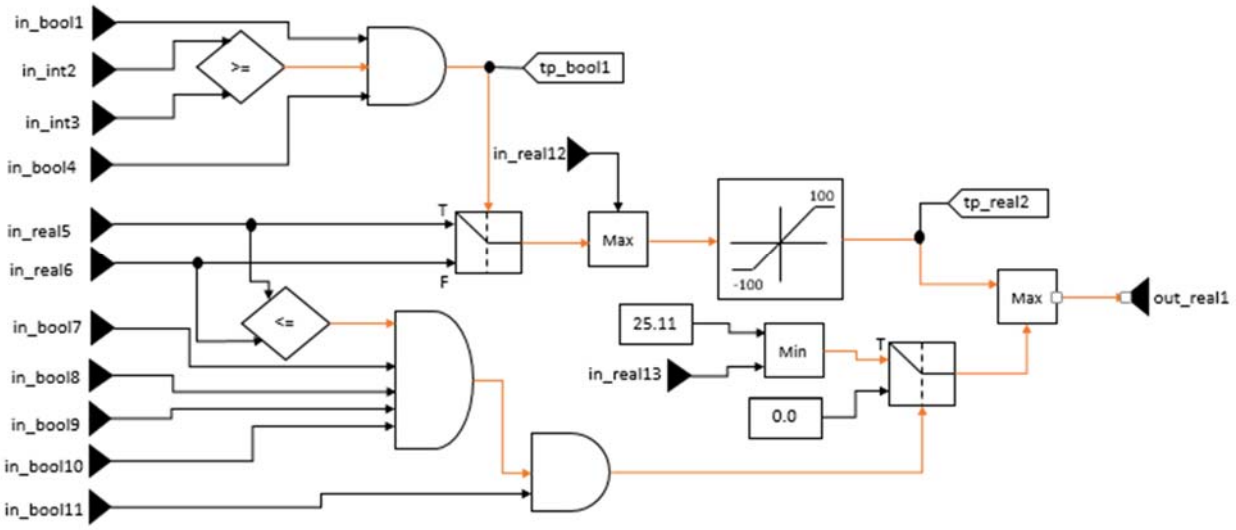


Figura 27 Salidas de bloques.

**Coberturas aplicables a cada bloque:** Es más evidente que entre más coberturas se tengan que aplicar a cada bloque, más casos de prueba necesita y más documentación se tiene que generar. Si numeramos los bloques como lo muestra la Figura 28, podemos resumir en la Tabla 3 el número de coberturas por cada elemento.

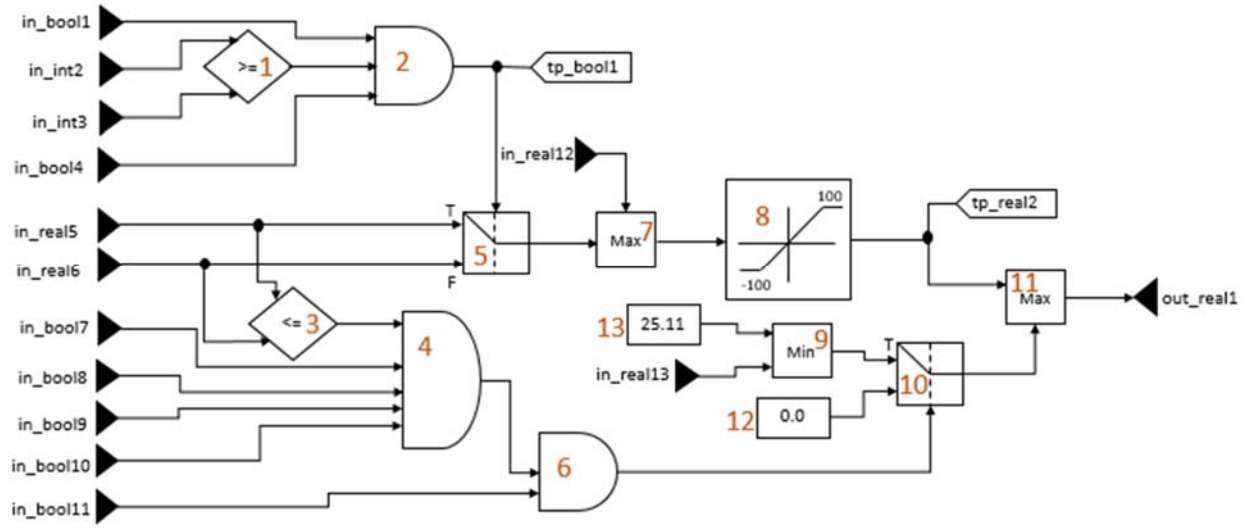


Figura 28 Numeración de bloques

NOTA: Debido a la confidencialidad con los datos y estándares de la empresa se usaran de forma ilustrativa las siguientes coberturas:

A: Prueba una comparación.

B: Prueba una compuerta lógica.

C: Prueba las fronteras entre secciones de código generado por el bloque.

D: Prueba la funcionalidad de una asignación.

E: Prueba el rango de valores para ciertos tipos de datos y operaciones aritméticas.

F: Prueba un bloque iterativo

G: Prueba una tabla de verdad

| Elemento | Número de Coberturas | Coberturas |
|---|---|---|
| **Bloque 1** | 1 | A |
| **Bloque 2** | 1 | B |
| **Bloque 3** | 1 | A |
| **Bloque 4** | 1 | B |
| **Bloque 5** | 1 | C |
| **Bloque 6** | 1 | B |
| **Bloque 7** | 2 | A,C |
| **Bloque 8** | 3 | A, C |
| **Bloque 9** | 2 | A, C |
| **Bloque 10** | 1 | C |
| **Bloque 11** | 2 | A, C |
| **Bloque 12** | 1 | D |
| **Bloque 13** | 1 | D |
| **Entrada 1** | 1 | E |
| **Entrada 2** | 1 | D |
| **Entrada 3** | 1 | D |
| **Entrada 4** | 1 | E |
| **Entrada 5** | 1 | D |

| | | |
|---|---|---|
| **Entrada 6** | 1 | D |
| **Entrada 7** | 1 | E |
| **Entrada 8** | 1 | E |
| **Entrada 9** | 1 | E |
| **Entrada 10** | 1 | E |
| **Entrada 11** | 1 | E |
| **Entrada 12** | 1 | D |
| **Entrada 13** | 1 | D |
| **Punto de Prueba 1** | 1 | D |
| **Punto de Prueba 2** | 1 | D |
| **Salida 1** | 1 | D |

Tabla 3 Resumen de coberturas por elemento

Como se puede observar, sólo se necesitan cinco tipos de coberturas (A, B, C, D y E) para toda la lógica del ejemplo según el proceso de pruebas unitarias definido usado en este proyecto; las coberturas F y G no aplican a la unidad debido a que no tiene ni bloques iterativos ni tablas de verdad, también se puede notar en la columna "número de coberturas" de la Tabla 3 que para el bloque 8, hay 3 coberturas pero sólo fueron registradas dos en la columna "Coberturas" de la misma tabla, esto quiere decir que una de las dos coberturas se tiene que hacer dos veces, en este caso, la cobertura "A" aplica dos veces debido a que de forma interna, éste bloque tiene dos comparaciones.

**Casos de prueba necesarios para las coberturas:** Cada cobertura tiene valores estándar ya definidos en el proceso de pruebas unitarias, éstos definen el criterio de satisfacción de la cobertura y son específicos para las entradas en cada elemento al cual se le está aplicando la cobertura, es decir, si la cobertura *X* tiene dos valores establecidos digamos 0 o 1 y se está aplicando al elemento *Y* con entrada *Z*, cuando *Z* haya tomado éstos valores podemos decir que la cobertura *X* ya se cumplió para el elemento *Y*.

Los casos de prueba para cada cobertura están directamente relacionados con el número de valores estándar para ésta, siguiendo el ejemplo del párrafo anterior, la cobertura *X* requiere de dos casos de prueba.

**Documentación de las coberturas:** Todos los valores estándar deben ser documentados para cada cobertura, entonces, entre más valores estándar existan en la lógica más documentación se genera y esto hace más compleja la prueba.

**Documentación extraordinaria (desviaciones a los estándares):** Éste tipo de documentación es para justificar y/o explicar el por qué alguna cobertura no puede ser considerada como satisfactoria en su totalidad, como por ejemplo, la naturaleza de la lógica, es decir, la misma lógica puede no permitir poner el valor estándar para alguna cobertura.

Siguiendo con el ejemplo de la cobertura *X*, digamos que la entrada *Z* es una constante cuyo valor es 0, entonces, el valor estándar 1 nunca podría ser documentado, aquí es donde se debe agregar documentación para justificar la ausencia del valor 1 para la cobertura *X*.

**Interacción con otros bloques:** Ésta factor es también importante para poder calcular la complejidad, si un bloque interactúa con otros bloques se vuelve más complicado el diseño de casos de prueba.

Cuando un elemento interactúa con otros de forma anterior, es decir, los elementos con los que está ligado se ejecutan antes que el elemento que estamos probando, podría resultar muy complicado alcanzar el valor estándar para la cobertura aplicada.

Por otro lado, cuando la interacción es posterior, o bien, los elementos ligados a éste se ejecutan después del elemento que estamos probando, podría ser de muy alta complejidad poder afectar directamente una salida con los valores estándar para la cobertura aplicada.

Cada una de éstas interacciones pueden ser directas e indirectas, las primeras son la conexión con el elemento inmediato en ambas direcciones (anterior y posterior), y la segunda es las interacciones de los inmediatos con otros elementos.

La Figura 29 muestra con rojo la interacción directa e indirecta anterior y con verde la interacción directa e indirecta posterior para el bloque 6, se puede notar que las entradas in_real5 e int_real6 interactúan para ambos casos.

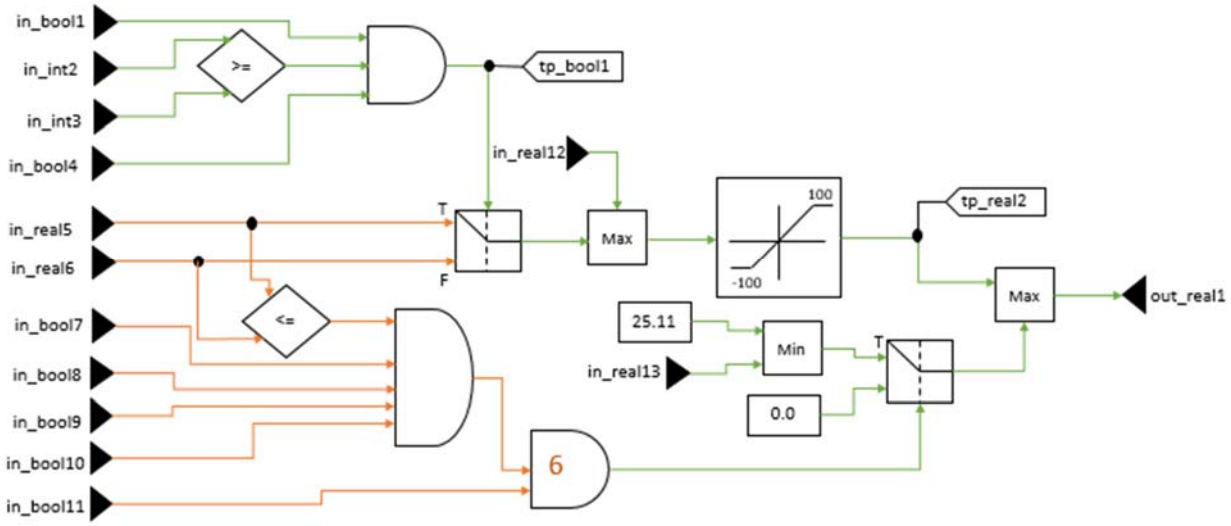


Figura 29 Interacción entre bloques.

Existen herramientas de ayuda para el bloque por ejemplo las que sugieren casos de prueba para los elementos, esto facilita un poco la prueba y debe ser considerado para el cálculo de la complejidad.

**Número de casos de prueba totales:** Éste factor es muy importante para el peso específico general ya que como se explicó anteriormente está directamente relacionado con la complejidad, además de que el tiempo de ejecución para la simulación puede llegar a ser considerable para aquellas unidades con muchos casos de prueba.

## 3.3 IMPLEMENTACIÓN DE LA METODOLOGÍA

Tomando en cuenta las dos etapas anteriores del procedimiento de éste proyecto, se desarrolló una herramienta que fuera capaz de calcular la complejidad total por unidad. Los requerimientos de alto nivel para ésta herramienta fueron los siguientes:

1. La herramienta debe de realizarse en lenguaje Perl: Perl es un lenguaje de programación desarrollado principalmente para la manipulación de texto [36], por ésta razón y debido a que los diagramas pueden ser manipulados en modo texto se escogió éste lenguaje.
2. La herramienta debe recibir enlistadas las unidades a analizar: La lista debe ser un archivo nombrado <lista>.txt donde "lista" puede ser cualquier nombre que pueda ser usado como referencia posterior. La Figura 30 es un ejemplo de la lista en <lista>.txt; no existe un mínimo o máximo número de unidades en la lista, sin embargo y por obvias razones, el mínimo número de unidades listadas es uno.

```
unidad1
unidad2
unidad3
```

Figura 30 Lista de unidades en <lista>.txt

3. Para cada unidad se debe de analizar el archivo .net como ya se mencionó en modo texto y realizar lo siguiente.

   NOTA: Debido a la confidencialidad con los datos y estándares de la empresa las siguientes formulas y/o pesos específicos son de forma ilustrativa.

   a. Identificar tipo de lógica (flujo o señal).
   b. Cuantificar número de entradas a la unidad: El número de entradas a la unidad suma la siguiente complejidad.

$$A = (N * 0.012)$$

Donde:

A: Complejidad por número de entradas a la unidad

N: Número de entradas a la unidad

c. Cuantificar número de salidas: el número de salidas en el diagrama suma la siguiente complejidad.

$$B = (N * 0.15)$$

Donde:

B: Complejidad por número de salidas en la unidad

N: Número de salidas en la unidad

d. Cuantificar número de puntos de prueba: el número de puntos de prueba en el diagrama suma la siguiente complejidad.

$$C = (N * 0.05)$$

Donde:

C: Complejidad por número de puntos de prueba en la unidad

N: Número de puntos de prueba en la unidad

e. Identificar los bloques contenidos: el número de bloques en el diagrama suma la siguiente complejidad.

$$D = (N * 0.3)$$

Donde:

D: Complejidad por número de bloques

N: Número de bloques

f. Calcular el peso específico para cada uno de los bloques usando lo siguiente.

$$E = (X + Y) * Z$$

Donde:

E: Complejidad sumada por cada bloque

X: Número de entradas al bloque

Y: Número de salidas del bloque

Z: Factor de complejidad

El factor de complejidad fue definido por los expertos en el proceso para cada uno de los bloques permitidos en la lógica, éste factor está en función de número de coberturas, número de casos de prueba necesarios para el bloque, documentación extraordinaria

requerida, complejidad del algoritmo y herramientas de ayuda para el bloque.

g. Totalizar el número de casos de prueba requeridos para la unidad derivados de bloques.

$$F = (T_1 + T_2 + \cdots + T_n) * 0.1$$

Donde:

F: Complejidad sumada por casos de prueba totales

T: Casos de prueba por cada bloque

n: Número total de bloques

h. Calcular la complejidad total de la unidad

$$G = A + B + C + D + \left(\sum_{1}^{n} E\right) + F + W$$

Donde:

G: Complejidad total

A: Complejidad por número de entradas a la unidad

B: Complejidad por número de salidas en la unidad

C: Complejidad por número de puntos de prueba en la unidad

D: Complejidad por número de bloques

E: Complejidad sumada por cada bloque

n: Número total de bloques

F: Complejidad sumada por casos de prueba totales

W: Factor de complejidad por unidad con valor de 2.5

Este factor de complejidad por unidad fue definido por los expertos en el proceso y está en función de pasos en el proceso requeridos para cada unidad como por ejemplo la simulación.

4. La herramienta debe de generar una lista con el número de complejidad para cada unidad: La lista de resultados generados por la herramienta es en un archivo <resultados>.csv, donde “resultados” coincide con <lista> de <lista>.txt. La Figura 31 muestra un ejemplo de <resultados>.csv.

| | A | B |
|---|---|---|
| 1 | unidad1 | 12.5 |
| 2 | unidad2 | 25.8 |
| 3 | unidad3 | 23.7 |

Figura 31 Lista de resultados en <resultados>.csv

Como se puede observar, el nombre de la unidad es en la columna “A” y el resultado en la columna “B” en el renglón correspondiente a la unidad. Éste resultado es un número decimal que la herramienta lo calcula analizando los elementos a al f del punto 3 y tomando en cuenta los pesos específicos del punto siguiente.

5. Los pesos específicos deben ser configurables: Los pesos específicos de cada elemento a analizar es configurable ya que se debe de tener la capacidad de ajustar éstos mismos debido a generaciones de herramientas que faciliten la prueba unitaria o bien cabios en el proceso de ésta misma.

# CAPÍTULO 4. RESULTADOS

Siguiendo con el ejemplo del capítulo anterior e identificando los bloques como se muestra en la Figura 32, al ejecutar la herramienta para calcular la complejidad de la unidad para un proceso de pruebas unitarias se obtiene lo siguiente:

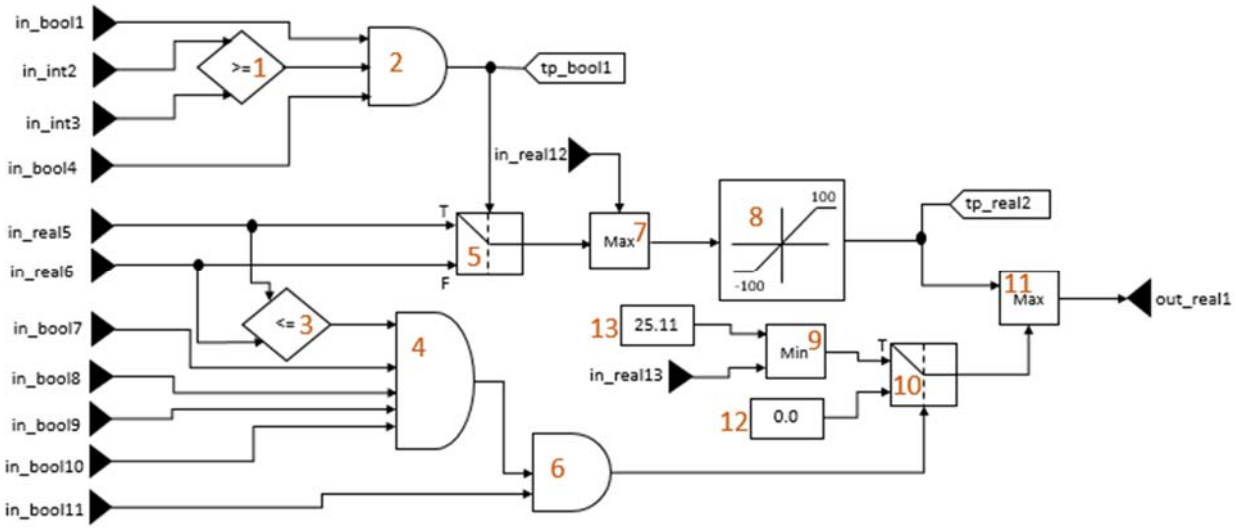


Figura 32 Identificación de bloques.

**Identificar tipo de lógica**

| Tipo de lógica |
| --- |
| **Señal** |

Tabla 4 Tipo de lógica

**Cuantificar número de entradas por unidad**

| Entradas a la unidad |
| --- |
| **13** |

Tabla 5 Número de salidas

Con este número de salidas se obtiene:

$$A = (13 * 0.012) = 0.156$$

**Cuantificar número de salidas por unidad**

| Salidas |
| --- |
| 1 |

Tabla 6 Número de salidas.

Obteniendo así:

$$B = (1 * 0.15) = 0.15$$

**Cuantificar número de puntos de prueba**

| Puntos de prueba |
| --- |
| 2 |

Tabla 7 Puntos de prueba (test points).

Con estos puntos de prueba se calcula lo siguiente:

$$C = (2 * 0.05) = 0.1$$

**Identificar los bloques contenidos**

| Bloques |
| --- |
| Bloque 1 |
| Bloque 2 |
| Bloque 3 |
| Bloque 4 |
| Bloque 5 |
| Bloque 6 |

| |
|---|
| **Bloque 7** |
| **Bloque 8** |
| **Bloque 9** |
| **Bloque 10** |
| **Bloque 11** |
| **Bloque 12** |
| **Bloque 13** |

Tabla 8 Bloques contenidos en la unidad.

Con éste número de bloques se obtiene:

$$D = (13 * 0.3) = 3.9$$

**Cuantificar número de entradas por bloque**

| Bloques | Entradas |
|---|---|
| **Bloque 1** | 2 |
| **Bloque 2** | 3 |
| **Bloque 3** | 2 |
| **Bloque 4** | 5 |
| **Bloque 5** | 3 |
| **Bloque 6** | 2 |
| **Bloque 7** | 2 |
| **Bloque 8** | 1 |
| **Bloque 9** | 2 |
| **Bloque 10** | 3 |
| **Bloque 11** | 2 |
| **Bloque 12** | 0 |
| **Bloque 13** | 0 |

Tabla 9 Número de salidas por cada bloque.

**Cuantificar número de salidas por bloque**

| Bloques | Salidas |
|---|---|
| **Bloque 1** | 1 |
| **Bloque 2** | 1 |
| **Bloque 3** | 1 |
| **Bloque 4** | 1 |
| **Bloque 5** | 1 |
| **Bloque 6** | 1 |
| **Bloque 7** | 1 |
| **Bloque 8** | 1 |
| **Bloque 9** | 1 |
| **Bloque 10** | 1 |
| **Bloque 11** | 1 |
| **Bloque 12** | 0 |
| **Bloque 13** | 0 |

Tabla 10 Número de salidas por cada bloque.

**Analizar el factor de complejidad para cada bloque**

La tabla Tabla 11 muestra los bloques usados en la unidad ejemplo así como el peso específico calculado para cada uno.

NOTA: Debido a la confidencialidad con los datos y estándares de la empresa los pesos específicos son de forma ilustrativa.

| Bloques | Peso específico |
|---|---|
| **Bloque 1** | 0.2 |
| **Bloque 2** | 0.3 |

| | |
|---|---|
| **Bloque 3** | 0.2 |
| **Bloque 4** | 0.3 |
| **Bloque 5** | 0.25 |
| **Bloque 6** | 0.3 |
| **Bloque 7** | 0.27 |
| **Bloque 8** | 0.6 |
| **Bloque 9** | 0.27 |
| **Bloque 10** | 0.25 |
| **Bloque 11** | 0.27 |
| **Bloque 12** | 0.1 |
| **Bloque 13** | 0.1 |

Tabla 11 Pesos específicos para cada bloque.

Usando la información mostrada en Tabla 9, Tabla 10 y Tabla 11y la formula mostrada a continuación, se puede obtener la complejidad que sumará cada bloque a la complejidad total de la unidad, ésta información es mostrada en la Tabla 12.

$$E = (X + Y) * Z$$

| Bloques | Entradas (X) | Salidas (Y) | Peso Específico (z) | Complejidad (E) |
|---|---|---|---|---|
| **Bloque 1** | 2 | 1 | 0.2 | 0.6 |
| **Bloque 2** | 3 | 1 | 0.3 | 1.2 |
| **Bloque 3** | 2 | 1 | 0.2 | 0.6 |
| **Bloque 4** | 5 | 1 | 0.3 | 1.8 |
| **Bloque 5** | 3 | 1 | 0.25 | 1 |
| **Bloque 6** | 2 | 1 | 0.3 | 0.9 |
| **Bloque 7** | 2 | 1 | 0.27 | 0.81 |
| **Bloque 8** | 1 | 1 | 0.6 | 1.2 |
| **Bloque 9** | 2 | 1 | 0.27 | 0.81 |
| **Bloque 10** | 3 | 1 | 0.25 | 1 |

| | | | | |
|---|---|---|---|---|
| **Bloque 11** | 2 | 1 | 0.27 | 0.81 |
| **Bloque 12** | 0 | 0 | 0.1 | 0 |
| **Bloque 13** | 0 | 0 | 0.1 | 0 |

Tabla 12 Complejidad que suma cada bloque.

Lo cual nos lleva a decir que la sumatoria de las complejidades individuales de cada bloque suma a la complejidad total de la unidad 10.73.

**Totalizar el número de casos de prueba requeridos para la unidad**

| Bloques | Casos de prueba |
|---|---|
| **Bloque 1** | 3 |
| **Bloque 2** | 4 |
| **Bloque 3** | 2 |
| **Bloque 4** | 6 |
| **Bloque 5** | 2 |
| **Bloque 6** | 3 |
| **Bloque 7** | 4 |
| **Bloque 8** | 7 |
| **Bloque 9** | 4 |
| **Bloque 10** | 2 |
| **Bloque 11** | 4 |
| **Bloque 12** | 0 |
| **Bloque 13** | 0 |

Tabla 13 Casos de prueba necesarios para cada bloque.

Como se puede observar, al hacer la sumatoria de los casos de prueba de la tabla el número total de cosos de prueba para la prueba unitaria es 41, lo cual suman una complejidad de 4.1, esto usando la formula siguiente:

$$F = (T_1 + T_2 + \cdots + T_n) * 0.1$$

**Hacer la sumatoria de complejidades para calcular la complejidad**

Usando la fórmula que se muestra a continuación y un factor *W* de 2.5, se obtiene una complejidad total *G* para la prueba unitaria de ejemplo.

$$G = A + B + C + D + \left(\sum_{1}^{n} E\right) + F + W$$

$$G = 1.56 + 0.15 + 0.1 + 3.9 + 10.73 + 4.1 + 2.5 = 23.04$$

La Tabla 14 muestra los elementos principales que influyen en la complejidad total de 23.04 para nuestra prueba unitaria ejemplo.

| Bloques | Entradas | salidas | Número de Coberturas | Casos de prueba | Peso específico |
|---|---|---|---|---|---|
| **Bloque 1** | 2 | 1 | 1 | 3 | 0.2 |
| **Bloque 2** | 3 | 1 | 1 | 4 | 0.3 |
| **Bloque 3** | 2 | 1 | 1 | 2 | 0.2 |
| **Bloque 4** | 5 | 1 | 1 | 6 | 0.3 |
| **Bloque 5** | 3 | 1 | 1 | 2 | 0.25 |
| **Bloque 6** | 2 | 1 | 1 | 3 | 0.3 |
| **Bloque 7** | 2 | 1 | 2 | 4 | 0.27 |
| **Bloque 8** | 1 | 1 | 3 | 7 | 0.6 |
| **Bloque 9** | 2 | 1 | 2 | 4 | 0.27 |
| **Bloque 10** | 3 | 1 | 1 | 2 | 0.25 |
| **Bloque 11** | 2 | 1 | 2 | 4 | 0.27 |
| **Bloque 12** | 0 | 0 | 1 | 0 | 0.1 |
| **Bloque 13** | 0 | 0 | 1 | 0 | 0.1 |

Tabla 14 Resumen de los elementos de la unidad.

## 4.1 COMPARACIÓN CON OTRAS MÉTRICAS

Como ya se mencionó anteriormente en la documentación de éste proyecto, existen otras formas de medir la complejidad de software, pero no hay una métrica que nos diga la complejidad de una prueba unitaria para un proceso ya definido.

La siguiente gráfica, muestra para 10 unidades reales una relación entre la métrica propuesta en este proyecto y otras métricas que se han usado dentro de la empresa para hacer una estimación de tiempos basado en la complejidad arrojada por dichas métricas, éstas son complejidad ciclomática y número de líneas de código.

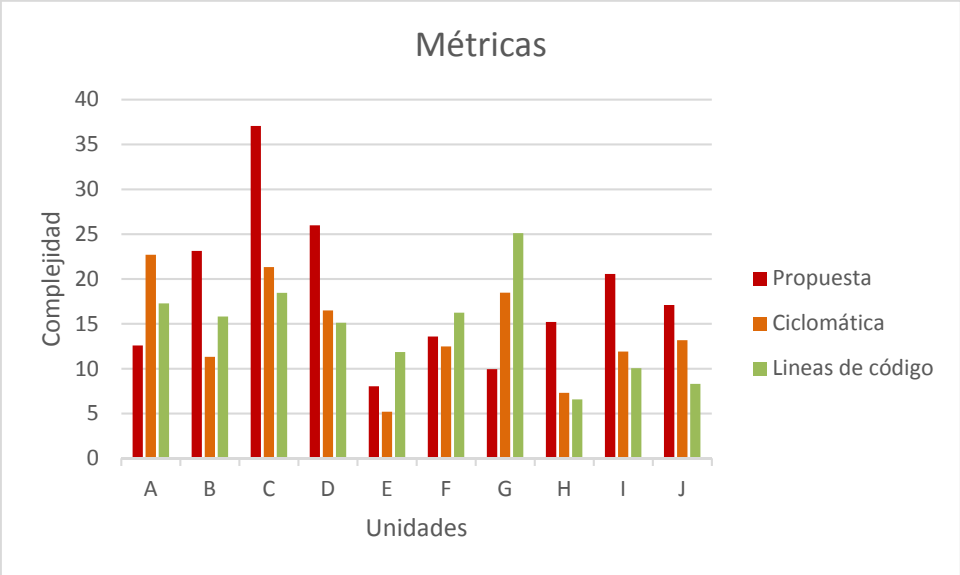


Figura 33 Relación entre métricas

La Figura 33 muestra que podría haber unidades con una complejidad alta calculada con líneas de código o complejidad ciclomática sin embargo, podrá ser fácil de aplicar la prueba unitaria con el proceso que la empresa utiliza, y por otro lado, podría haber unidades con complejidad baja calculada con las mismas metodologías diferentes a la propuesta, sin embargo, la prueba unitaria podría ser de grado de complejidad alto por la misma naturaleza del proceso.

## 4.2 COMPARACIÓN CON TIEMPOS DE CICLO REALES

Las Figura 34 y Figura 35 muestran la poca relación entre la complejidad calculada con las metodologías complejidad ciclomática y líneas de código respectivamente.

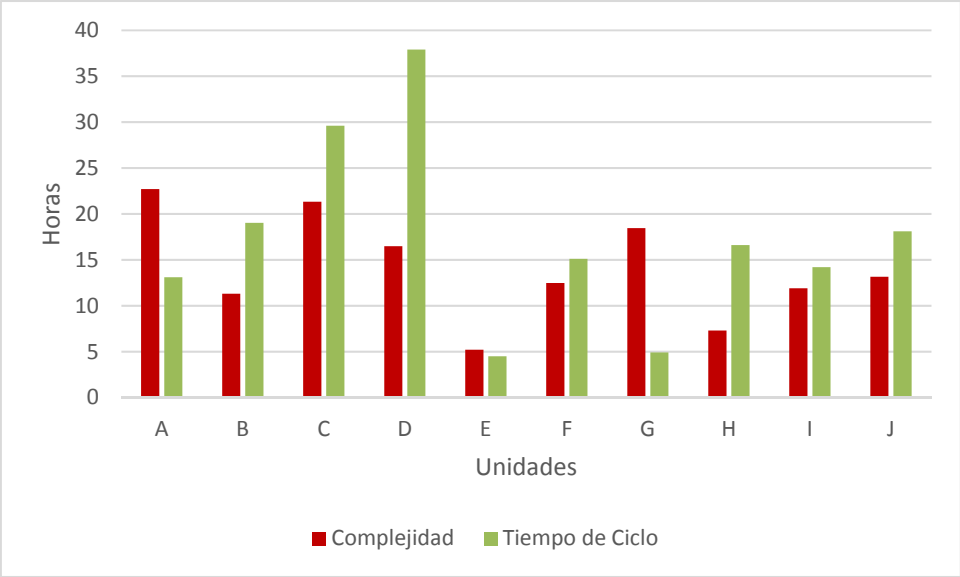


Figura 34 Relación entre complejidad ciclomática y tiempo de ciclo.

En la muestra de 10 unidades, podemos observar que algunas unidades podrían tener muy poca relación, en la Figura 34 resalta esta poca relación en las unidades “D” y “G”.

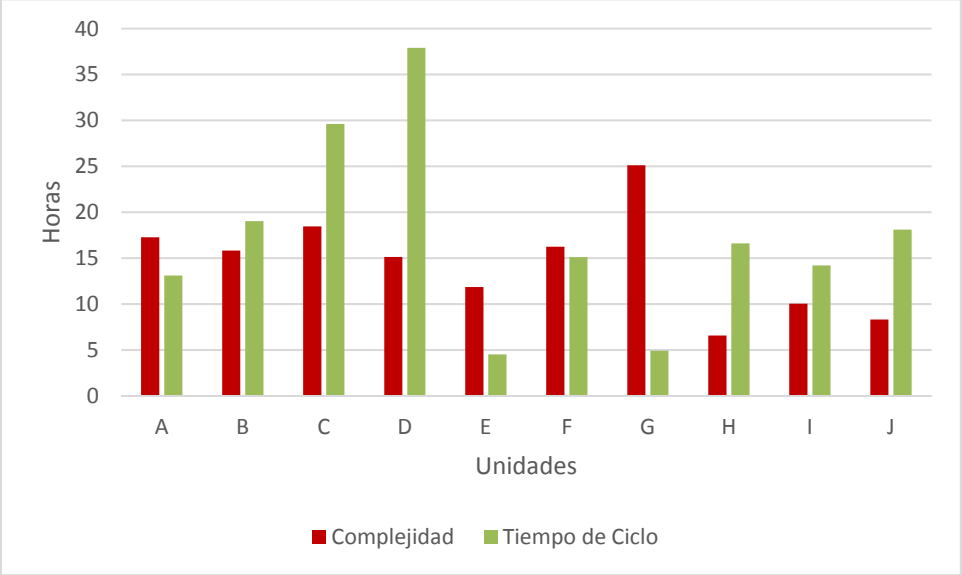


Figura 35 Relación número de líneas y tiempo de ciclo.

Ahora para este caso es prácticamente lo mismo, poca relación entre las unidades “D”, “E” y “G”.

A continuación, se muestra una gráfica con la relación entre la métrica calculada por la metodología propuesta y el tiempo de ciclo real para las mismas unidades del ejemplo en el tema anterior.

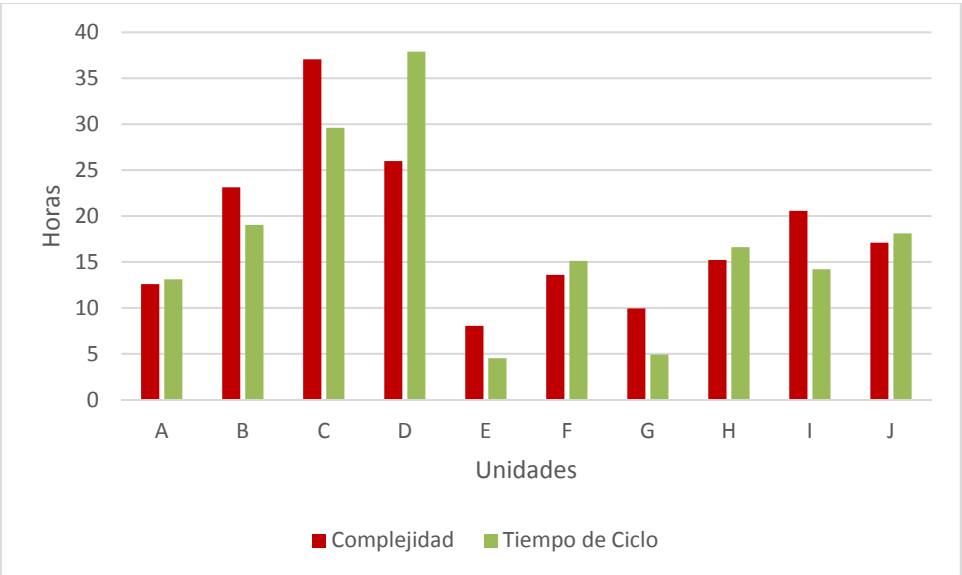

Figura 36 Relación Complejidad-Tiempo de Ciclo.

En la Figura 36 podemos observar la fuerte relación que existe entre la complejidad calculada por la metodología propuesta y el tiempo de ciclo que tomó realizar la prueba unitaria para cada unidad, incluso tomando en cuenta el factor humano que podría afectar mucho el tiempo de ciclo.

# CONCLUSIONES

En base a la investigación y desarrollo del proyecto en este tema de tesis, se puede concluir que la metodología propuesta, de forma acertada logró calcular la complejidad de una prueba unitaria para lógicas de control de Software aéreo crítico (Nivel A). Es preciso mencionar que dicha metodología toma en cuenta las consideraciones necesarias para llevar a cabo un proceso ya definido. Durante la investigación requerida para realizar la propuesta, se analizaron desde la lógica a tratar hasta el más mínimo detalle del proceso definido, contemplando herramientas de ayuda, documentación requerida, entre otros.

La metodología desarrollada arrojó valores de forma eficiente proporcionando un tiempo más real para la ejecución de las pruebas unitarias. Por esta razón se concluye de igual manera que dicha metodología permite planear de manera más acertada los proyectos, proporcionando fechas de entrega más precisas al cliente que requiere las pruebas de las unidades de software. Así mismo mediante el uso de esta metodología es posible que las lógicas a trabajar sean asignadas de forma inteligente para lograr una curva de aprendizaje óptima para los ingenieros nuevos en el proceso, evitando de esta manera el incremento en tiempo de trabajo en cada unidad.

## RECOMENDACIONES

- Resultó sumamente complejo hacer un análisis de la interacción entre bloques mediante la herramienta de tal forma que nos diera un número sin incertidumbre considerable para cada bloque, por lo tanto, los expertos en el proceso y expertos en las lógicas de los programas en los cuales la metodología será aplicada decidieron agregar una métrica que sume complejidad en función del número de bloques, es decir, entre más bloques existan en la unidad, más probabilidad de interacción entre ellos hay. Se recomienda hacer un análisis más profundo y diseñar un algoritmo que pueda hacer el cálculo de la complejidad que suma esta interacción entre bloques para que la métrica sea más certera.
- Podría diseñarse otra herramienta que analice los entregables una vez que se haya terminado la prueba unitaria, ésta herramienta debería de estar calibrada con los pesos específicos de este proyecto sólo que ahora tomará en cuenta los casos de prueba reales y no los estimados al inicio además de las desviaciones a las coberturas documentadas, esto con la finalidad de comparar ambas complejidades calculadas. El resultado de la comparación podría resultar útil para recalibrar los pesos específicos, o bien, sacar una complejidad promedio que pudiera resultar en una complejidad más apegada a la realidad.
- Para usar la complejidad propuesta por ésta metodología como entrada para calcular estimaciones de tiempo de ciclo por unidad, es decir, calcular una estimación de tiempo en función de ésta complejidad es necesario un análisis numérico profundo para tener una estimación de tiempo de ciclo más real.

# REFERENCIAS BIBLIOGRÁFICAS


[1] «Real Academia Española,» 2016. [En línea]. Available: http://dle.rae.es/?id=0tZAhTX.

[2] José Antonio Tena Sendra, «En El Aire,» 03 Junio 2015. [En línea]. Available: http://enelaire.mx/el-futuro-del-vuelo-sobre-aviacion-e-inteligencia-artificial/.

[3] «Sky Brary,» 2013. [En línea]. Available: http://www.skybrary.aero/index.php/Autoland.

[4] T. Schultz, «Coverity,» Mayo 2009. [En línea]. Available: https://www.coverity.com/library/pdf/Coverity-Meeting-DO-178B-Requirements.pdf.

[5] RTCA, «DO-178B,» 1992.

[6] R. Burkey, «Birds Project,» 2006. [En línea]. Available: http://www.sandroid.org/birdsproject/4dummies.html.

[7] «DO-178B Industry Group,» 2008. [En línea]. Available: http://www.do178site.com/do178b_questions.php.

[8] J. J. Chilenski, «Software Development Under DO-178B,» 2002. [En línea]. Available: http://www.opengroup.org/rtforum/jan2002/slides/safety-critical/chilenski.pdf.

[9] «The Standish Group,» 2016. [En línea]. Available: http://standishgroup.com/about.

[10] S. W. Shane Hastie, «InfoQ,» 2015. [En línea]. Available: https://www.infoq.com/articles/standish-chaos-2015.

[11] ISO, «ISO/IEC,» [En línea]. Available: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=43447. [Último acceso: 21 Febrero 2017].

[12] D. Cantone, Implementacion y Debugging, MP EDICIONES, 2008.

[13] I. Somerville, Ingeniería De Software, Pearson, 2011.

[14] D. M. S.Balaji, «WATEERFALL VS V-MODEL VS AGILE: A COMPARATIVE STUDY ON SDLC,» 29 Junio 2012. [En línea]. Available: http://www.jitbm.com/Volume2No1/waterfall.pdf. [Último acceso: 2017 Febrero 2107].

[15] D. L. Williams, A (Partial) Introduction to Software Engineering and Methods, 2010-2011.

[16] D. Gelperin, «IEEE Standard for Software Unit Testing 1008-1987,» 1993.

[17] A. Bertolino, Capítulo 5 - IEEE SWEBOK Guide to the Software Engineering Body of Knowledge, 2001.

[18] R. D. C. a. S. P. Jaskiel, Systematic Software Testing, 2002.

[19] R. Paulo, «Swansea University Prifysgol Abertawe,» 12 Enero 2007. [En línea]. Available: http://www.cs.swan.ac.uk/~csmarkus/CS339/dissertations/PauloRA.pdf. [Último acceso: 1 Marzo 2017].

[20] IEEE, «IEEE Standard Glossary of Software Engineering Terminology,» *IEEESTD,* vol. 10, nº 1109, pp. 1-84, 1990.

[21] M. Y. &. M. Pezze, «Computer and Information Science at University of Oregon,» 25 02 1998. [En línea]. Available: http://ix.cs.uoregon.edu/~michal/Classes/W98/LecNotes/10-Testing-system.pdf. [Último acceso: 02 03 2017].

[22] D. G. Frost, «Ricardo,» [En línea]. Available: https://www.ricardo.com/Documents/IA/ControlsandElectronics/Downloads/Autocodegen_paper.pdf. [Último acceso: 03 03 2017].

[23] I. The MathWorks, «MathWorks,» 2017. [En línea]. Available: https://www.mathworks.com/products/embedded-coder.html. [Último acceso: 03 03 2017].

[24] E. Denney, «National Aeronautics and Space Administration,» [En línea]. Available: https://ti.arc.nasa.gov/m/profile/edenney/papers/Denney-BigSky-08.pdf. [Último acceso: 03 03 2017].

[25] E. T. SA, «Esterel Technologies,» 2014. [En línea]. Available: http://www.esterel-technologies.com/products/scade-suite/automatic-code-generation/scade-suite-kcg-ada-code-generator/. [Último acceso: 06 03 2017].

[26] ADI, «International, Applied Dinamics,» 2007. [En línea]. Available: https://www.adi.com/pdfs/product/BEACON_DS3.pdf. [Último acceso: 08 03 2017].

[27] G. Booch, Análisis y Diseño Orientado a Objetos con Aplicaciones, Pearson Educación, 1996.

[28] S. N. &. C. Sandros, «Gothenburg University,» 1999. [En línea]. Available: https://gupea.ub.gu.se/bitstream/2077/1050/1/Nystedt.Sandros.ia5840.pdf. [Último acceso: 09 03 2017].

[29] H. D. M. &. P. B. Dyson, «University of Tennessee,» 1990. [En línea]. Available: http://trace.tennessee.edu/cgi/viewcontent.cgi?article=1056&context=utk_harlan. [Último acceso: 2017 03 11].

[30] V. B. &. A. Turner, «Iterative Enhancement: A Practical Technique For Software Development,» *IEEE Transaction On Software Enigineering,* Vols. %1 de %2SE-1, nº 4, pp. 390-396, 1975.

[31] C. Y. &. X. Shiyi, «Exploration of Complexity in Software Reliability,» *TSINGHUA SCIENCE AND TECHNOLOGY,* vol. 12, nº S1, pp. 266-269, 2007.

[32] S. Y. &. X. Shiyi, «A New Method for Measurement and Reduction of Software Complexity,» *TSINGHUA SCIENCE AND TECHNOLOGY,* vol. 12, nº S1, pp. 212-216, 2007.

[33] F. S. Vacas, Complejidad y Tecnologías de la Información, Madrid, 2009.

[34] M. K. W. A. H. &. A. R. D. Kenneth Magel, «Applying Software Complexity Metrics to Program Maintenance,» *IEEE Computer,* pp. 65-79, 1982.

[35] M. &. M. Frappier, «University of Ottawa,» 21 Enero 1994. [En línea]. Available: www.dmi.usherb.ca/~frappier/Papers/tm2.pdf. [Último acceso: 30 Marzo 2017].

[36] «Perldoc,» [En línea]. Available: http://perldoc.perl.org/perlintro.html#What-is-Perl%3f. [Último acceso: 10 07 2017].

# ANEXOS

CIATEQ

25 de Julio de 2017

Mtro. Geovany González Carlos
Coordinador Académico

Los abajo firmantes, miembros del Comité Tutorial del alumno Ing. Ismael Martínez García, una vez revisada la Tesis o tesina titulada: "Metodología Para El Cálculo De Complejidad En Pruebas Unitarias De Código Autogenerado", autorizamos que el citado trabajo sea presentado por el alumno para la revisión del mismo con el fin de alcanzar el grado de Maestro en Sistemas Inteligentes Multimedia durante el Examen de Titulación correspondiente.

Y para que así conste se firma la presente a los 25 días del mes de Julio del año 2017.

| | |
|---|---|
| Grado y nombre completo | M.I.E Ramón Reyes Robles |
| Asesor Académico | Asesor en Planta |

4 de Agosto de 2017

**Respetables miembros del Jurado:**

Me ha tocado el honor de haber sido designado Revisor del trabajo titulado "**Metodología para el Cálculo de Complejidad en Pruebas Unitarias de Código Autogenerado**" del Ing. **Ismael Martínez García**.

Después de haber leído detalladamente el trabajo que me fue entregado, he tenido la oportunidad de intercambiar información con el sustentante y como resultado de estas acciones he concluido que:

El trabajo tiene los siguientes aspectos positivos:

1.- La tesis aborda un tema oportuno y complejo, de alto impacto para el ciclo de vida de software aéreo, en concreto para los procesos conocidos como "lado derecho de la V de desarrollo". Por años, el problema que aborda la tesis ha sido relevante para el contexto industrial en el que el autor colabora, y su solución puede convertirse en un gran diferenciador y ventaja competitiva para grupos enfocados al diseño de pruebas de software.

2.- La metodología y métricas propuestas por la tesis complementan de una manera muy rica las métricas de complejidad tradicionales. La comunidad científica en Ingeniería de Software se ha concentrado en desarrollar métricas para medir la complejidad, acoplamiento, cohesión del producto de software, pero no ha abordado de manera frontal el desarrollo de métricas para evaluar la

complejidad de una *prueba* de software. En este aspecto, el presente trabajo ofrece una aproximación novedosa a un problema frecuente en un ámbito industrial, como lo es la evaluación de la complejidad de las pruebas de software.

3.- La metodología y métricas propuestas por este trabajo pueden coadyuvar a la estimación de esfuerzo requerido para la implementación de pruebas unitarias. La estimación de esfuerzo en *desarrollo* de software es un tema ampliamente abordado por conferencias y revistas académicas, sin embargo, la estimación del esfuerzo de pruebas requiere estudios más profundos y evaluados, como el ofrecido por este trabajo.

4.- La elaboración del estado del arte es eminentemente práctica, y no se limita o distrae con contenido exclusivamente académico. Esto es de vital importancia para trabajos de tesis con pertinencia industrial, que se recomienda se orienten a un contexto de aplicación práctica.

El trabajo tiene las siguientes oportunidades de mejora:

1.- El caso de uso para la implementación de la métrica es limitado a una sola unidad de software, en un solo generador automático de código. En el ejemplo en BEACON, el lector apreciaría mucho la implementación de la metodología y el cálculo de las métricas en unidades de software diferentes a la representada en la Figura 22 y siguientes (repetitividad). Adicionalmente, además de BEACON, el lector apreciaría mucho la implementación y evaluación de la métrica reproduciendo el diagrama en otra herramienta para la generación automática de código (cualquiera de las mencionadas en la sección 2.4), para discutir con amplitud la efectividad de la métrica más allá de su implementación en BEACON (reproducibilidad). Los aspectos referentes a repetitividad y reproducibilidad son esenciales para el diseño de experimentos.

2.- El lector se beneficiaría mucho de una discusión concreta de cada elemento de la hipótesis propuesta en el Párrafo 1.5: la sección "Conclusiones" discute

solamente de manera muy genérica el contenido de la hipótesis. En concreto, esto deja en el lector abierta la pregunta respecto de:

- tener un comentario o un juicio claro respecto de la efectividad de la metodología;
- tener un comentario o un juicio claro sobre los tiempos requeridos para diseñar de la prueba, y su asociación con la magnitud calculada por la métrica;
- una discusión extensa sobre la métrica de complejidad de la unidad de software y la métrica de complejidad de la prueba de software, así como su correlación.

3.- El documento de tesis se beneficiaría ampliamente de una revisión editorial y corrección de estilo, por ejemplo: inconsistencias inglés-español, control de viudas y huérfanas, numeración de ecuaciones ausente, entre otros detalles editoriales.

Adicionalmente:

1.- Se sugiere dar continuidad a este trabajo a través una posible segunda tesis de maestría supervisada por el mismo asesor académico, que se concentre en la estimación del esfuerzo de pruebas usando la metodología propuesta aquí. Este trabajo de maestría podría tomar un proyecto real de prueba de software, y determinar la efectividad de métrica propuesta aquí implementándola en un ejercicio estimación de esfuerzo en un contexto real. Esta segunda tesis se enfocará a evaluar la utilidad y corrección de la métrica, encontrará oportunidades de mejora identificadas a partir del uso continuo de la métrica.

Haciendo un análisis crítico del trabajo y balanceando lo positivo y las oportunidades de mejora, considero **RECOMENDAR** al Jurado que le otorgue el Grado de Maestro en Sistemas Inteligentes Multimedia al Ing. **Ismael Martínez García** por lo que acepto se imprima el trabajo de tesis.

No obstante lo anterior, le solicitaría al sustentante me responda las siguientes preguntas:

1.- ¿Existe una correlación observable cuantitativamente entre la métrica de complejidad del módulo respecto a la métrica de complejidad de la prueba asociada?

2.- ¿Además de la complejidad, cómo se estima el impacto de otras métricas como cohesión o acoplamiento en la métrica de complejidad de la prueba?

Le agradecería al Honorable Jurado tenga en consideración la propuesta de otorgar el Grado que pongo a su consideración.

Atentamente,

**Luis Ricardo Corral Velázquez**
**Doctor en Ciencia y Tecnología Informática**

Luis Corral

Dr. Luis Ricardo Corral Velázquez
(Documento firmado digitalmente)

Querétaro, Qro. a 25 de Julio de 2017

**M. en C. Fernando Talavera Sanchez**
**Mtro. Geovani González Carlos**
**CIATEQ, A.C.**

PRESENTE.

A petición del interesado, **Ing. Ismael Martínez García**, se emite la presente **constancia de satisfacción** por parte de la empresa **Centro de Ingeniería Avanzada en Turbomáquinas, S. de R. L.** (GEIQ) para la cual proporciona sus servicios. El Ing. **Martínez García** se encuentra inscrito en la **Maestría en Sistemas Inteligentes Multimedia en CIATEQ A. C.**, sede Querétaro, y presenta la Tesis titulada: **"Metodología para el cálculo de complejidad en pruebas unitarias de código autogenerado"**.

El estudio **cumple satisfactoriamente** los objetivos planteados en su ingreso al posgrado, con la finalidad de reforzar su formación académica y tecnológica dentro de la empresa y el sector. El estudio realizado agrega valor al área de **Aviations Controls Software Engineering**, donde el interesado se desempeña. Adicionalmente, una selección del contenido de la tesis ha sido revisada y publicada como artículo científico-tecnológico en la plataforma de divulgación GE Technical University.

Tanto la tesis como el artículo han sido revisados internamente por el supervisor y el líder técnico del área, y con base en la retroalimentación emitida por el asesor de tesis, se emite el presente documento de conformidad para que conste donde convenga en los procesos necesarios para la finalización del posgrado del interesado.

Sin otro particular de momento, me despido y quedo atento para cualquier aclaración que así se requiera.

Atentamente,

**Dr. Luis Ricardo Corral**
GEIQ Technical Education Leader
T + 52 442 456 7810
luis.corral@ge.com

Av. Campo Real #1692.
Col. Ampliacion El Refugio.
76146 Querétaro, Qro. México.