CIATEQ, A. C. Centro de Tecnología Avanzada
Dirección de Posgrado

Generación automática de casos de prueba a partir de requerimientos funcionales utilizando gramáticas libres de contexto

TESIS QUE PRESENTA

Ing. Alejandra Victoria Alcaraz
Asesor: Mtro. Gustavo Espejel Salazar

Para obtener el grado de

Maestro en
Sistemas Inteligentes Multimedia

Zapopan, Jalisco
agosto, 2021

CARTA DE LIBERACIÓN DEL ASESOR

Querétaro, Querétaro, 30 de mayo del 2021.

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

Los abajo firmantes, miembros del Comité Tutorial de la Ingeniera Alejandra Victoria Alcaraz, una vez revisado su Proyecto Terminal de tesis/tesina, titulado "GENERACIÓN DE CASOS DE PRUEBA A PARTIR DE REQUERIMIENTOS FUNCIONALES UTILIZANDO GRAMÁTICAS LIBRES DE CONTEXTO" **autorizo** que el citado trabajo sea presentado por el alumno para su revisión, con el fin de alcanzar el grado de **Maestría**.

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

Maestro Gustavo Espejel Salazar
Asesor Académico

CARTA DE LIBERACIÓN DEL REVISOR

Guadalajara, Jalisco, 28 de Julio del 2021.

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

Por medio de la presente me dirijo a usted en calidad de Revisor del proyecto terminal de la alumna **Alejandra Victoria Alcaraz**, cuyo título es:

Generación automática de casos de prueba a partir de requerimientos funcionales utilizando gramáticas libres de contexto

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

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

Atentamente,

Dr. Edgar Gonzalo Cossio Franco

F31b Revisión: 01-Mar-2021

DEDICATORIA

A mis padres por todo el apoyo y amor recibido de ellos en esta etapa de la tesis, por sus preocupaciones por mí y cuidados.

A mi novio y ahora mi esposo, por su apoyo incondicional para que yo pudiera lograr este sueño, por sus desvelos conmigo, ayudas y motivaciones para seguir adelante, por sus sacrificios para que yo pudiera concentrarme.

A mis hermanos por todas sus muestras de cariño en esos días difíciles y regaños para que siguiera avanzando.

A mis amigos y compañeros que preguntando por mi avance y dándome ánimos me motivaban para seguir adelante.

A Dios, que sin su ayuda no lo hubiera logrado, por medio de todas esas personas y circunstancias que me ha enviado pude continuar.

AGRADECIMIENTOS

Gracias a mi asesor que con su guía y apoyo he logrado este trabajo, por creer en mí y dedicarme de su tiempo. Gracias por todo el conocimiento compartido, la paciencia y trabajo en equipo.

Gracias al CIATEQ que me brindó la posibilidad de ampliar mis horizontes y conocimientos y me ha dado el apoyo para continuar con la tesis a pesar de las dificultades.

* * *

RESUMEN

Los sistemas de hardware son cada vez más complejos y difíciles de validar, a la vez que la validación para asegurar su correcto funcionamiento resulta muy costosa, entre el 50 y 70% del costo de los proyectos es dedicado para validación; y las fallas que se escapen pueden resultar también muy costosas. Es por esta razón que se deben buscar áreas de oportunidad para mejorar el proceso, resulte más eficiente y menos costoso.

En este trabajo se busca automatizar la generación de casos de prueba a partir del análisis de los requerimientos funcionales con el propósito de reducir el tiempo de generación de estos casos de prueba y así reducir los costos.

Para este fin se utiliza una plataforma que analiza los requerimientos y luego genera los casos de prueba con la información obtenida del análisis. Esta plataforma está conformada por dos módulos: un analizador de requerimientos y un generador de casos de prueba. El módulo analizador está siendo reusado del sistema TCG_UML (Espejel Salazar, 2019), este módulo requiere de una gramática para habilitar el análisis. Esta gramática fue desarrollada para habilitar el análisis sintáctico y semántico de los requerimientos. El módulo generador de casos de prueba se implementó para que trabajara en conjunto con el módulo analizador, este recibe el árbol sintáctico que el analizador genera y a partir de ahí construye los casos de prueba necesarios para garantizar que los requerimientos son correctamente validados.

Al automatizar la generación de casos de prueba se logró llegar al resultado esperado, se redujo el tiempo de generación de los casos de prueba y los costos de validación en un 8%.

Palabras clave: Casos de prueba, Automatización, Validación de hardware.

ABSTRACT

Hardware systems are getting more complex and harder to validate, and at the same time, validation to guarantee its correct functionality is very expensive, it is between 50% and 70% the total cost of the project. Also, the not found bugs could be very expensive. This is the reason to try to find opportunities to enhance the validation process and make it more efficient and less expensive.

In this project the objective is to automate the test cases generation activity from functional requirements analysis to reduce the time and cost.

For this purpose, it is used a platform that analyses the requirements and then it generates the tests cases with the information produced from the analysis. This platform is comprised of two modules: the requirements analyzer and a test cases generator. The analyzer module is taken from TCG_UML (Espejel Salazar, 2019), this module requires a grammar to enable the analysis. This grammar was developed to enable the syntactic and semantic analysis. The test cases generator module was developed to work in conjunction with the analyzer module, it receives the syntactic tree that the analyzer module generates, and it generates the test cases from it to guarantee that the requirements are correctly validated.

Test cases automation achieved the expected result, the time needed for tests cases generation and the cost was reduced by 8%.

Keywords: Test cases, Automation, Hardware validation.

ÍNDICE DE CONTENIDO

RESUMEN v
ABSTRACT vi
ÍNDICE DE CONTENIDO vii
ÍNDICE DE FIGURAS ix
ÍNDICE DE TABLAS x
GLOSARIO xi
1. INTRODUCCIÓN 12
1.1 ANTECEDENTES 12
1.2 DEFINICIÓN DEL PROBLEMA 15
1.3 JUSTIFICACIÓN 16
1.4 OBJETIVOS 17
1.4.1 Objetivo general 17
1.4.2 Objetivos específicos 17
1.5 HIPÓTESIS 17
2. MARCO TEÓRICO 18
2.1 REQUERIMIENTOS 18
2.1.1 Tipos de requerimientos para dispositivos electrónicos 20
2.1.2 Recomendaciones para escribir requerimientos de hardware 20
2.2 VALIDACIÓN DE HARDWARE 22
2.2.1 Pre-silicio o Verificación de SoC 24
2.2.2 Post-silicio o Validación de SoC 25
2.3 CRITERIOS PARA SELECCIONAR CASOS DE PRUEBA DE HARDWARE 26
2.3.1 Técnicas de pruebas 26
2.4 AUTOMATIZACIÓN DE LA GENERACIÓN DE CASOS DE PRUEBA 32
2.4.1 Casos de prueba de software 32
2.4.2 Casos de prueba de hardware 34
2.5 TEORIA DE AUTÓMATAS Y LENGUAJES FORMALES 38
2.5.1 Historia breve 38
2.5.2 Lenguajes 38
2.5.3 Gramáticas formales 39
2.5.4 Gramáticas libres de contexto 41

2.5.5 Autómatas 42
2.6 COMPILADORES 42
2.6.1 Análisis léxico 43
2.6.2 Análisis sintáctico 43
2.6.3 Análisis semántico 44
3. PROCEDIMIENTO 45
3.1 ESTANDARIZACIÓN DE REQUERIMIENTOS 45
3.2 INTEGRACIÓN CON ANALIZADOR CONFIGURABLE 47
3.3 IMPLEMENTACIÓN DE SOFTWARE GENERADOR DE CASOS DE PRUEBA 49
3.4 SEMÁNTICA IMPLEMENTADA 52
3.4.1 Resultados 52
3.4.2 Condiciones 53
3.4.3 Eventos 54
3.5 USO DE MC/DC 55
4. RESULTADOS 57
4.1 CONCENTRAR REQUERIMIENTOS 57
4.2 CASOS DE PRUEBA GENERADOS 57
4.3 COMPARATIVA CON OTRAS SOLUCIONES 60
4.4 TIEMPOS DE CICLO 61
CONCLUSIONES 64
RECOMENDACIONES 65
APORTACIÓN DE LA TESIS 66
APORTACIÓN SOCIAL DE LA TESIS 67
REFERENCIAS 68
ANEXOS 72

ÍNDICE DE FIGURAS

Figura 1. Personal asignado por trimestre 13
Figura 2. Modelo del Proceso de Ingeniería de Requerimientos lineal 19
Figura 3. Ejemplo de requerimiento funcional de hardware 22
Figura 4. Etapas en el ciclo de vida de un SoC 23
Figura 5. Diagrama de componentes de plataforma TCG_UML 37
Figura 6. Jerarquía de gramáticas, lenguajes y autómatas 41
Figura 7. Extracto de gramática para requerimientos funcionales 46
Figura 8. Ejemplo de requerimiento funcional 48
Figura 9. Condiciones en el árbol sintáctico 49
Figura 10. Diagrama de clases de modulo generador de casos de prueba 51
Figura 11. Ejemplo de caso de prueba 52
Figura 12. Requerimientos funcionales de hardware 58
Figura 13 .Casos de prueba generados 59

ÍNDICE DE TABLAS

Tabla 1. Actividades de validación post-silicio en Intel 14
Tabla 2. Comparación cualitativa de pre-silicio vs post-silicio vs pruebas de manufactura 24
Tabla 3. Tipos de cobertura estructural 29
Tabla 4. Combinaciones de condiciones 54
Tabla 5. Casos que cumplen con MC/DC 56
Tabla 6. Casos de prueba generados en formato CSV 60
Tabla 7. Comparativa de generadores de casos de pruebas 61
Tabla 8. Comparativa de resultados 62

GLOSARIO

Clase: En el ámbito de la programación orientada a objetos, una clase representa un tipo particular de objetos y tiene asociado un código (definición de la clase), que determina los atributos, así como los métodos que pueden ejecutar los objetos de la clase (Pavón Mestras, 2007).

DFT: Design for Testability.

HAS: Hardware Architecture Specification.

Landing Zone: En términos de productos, describe un rango de atributos medibles que un producto debe cumplir para lograr la visión del producto. Es muy útil definirlo cuando se necesita considerar muchos requerimientos y no se tiene que perder de vista ninguno de ellos y el cómo se relacionan (Wirfs-Brock, 2011).

MC/DC: Modified Condition/Decision Coverage.

PO: Power-On.

QS: Quality Samples.

SoC: System on Chip.

TCG_UML: Test Case Generator UML.

UML: Unified Modeling Language.

VV: Volume Validation.

1. INTRODUCCIÓN

La era actual está llena de dispositivos electrónicos que dan lugar a una gran diversidad de casos de aplicación. Debido a la alta complejidad de estos diseños modernos y a la creciente presión para reducir el tiempo de salida al mercado, es más probable que se escapen errores en la etapa de verificación y aparezcan hasta después que el chip ha sido manufacturado (Chang, Markov, & Bertacco, 2007), en la etapa de validación de hardware o post-silicio.

La validación de hardware incluye un sinnúmero de tareas, incluyendo verificación de la corrección funcional, adherencia a las restricciones de potencia y rendimiento, tolerancia a los márgenes de ruido eléctrico, aseguramiento de la calidad, robustez contra estrés físico o fallas térmicas en el ambiente, etc. (Mishra, Morad, Ziv, & Ray, 2017).

1.1 ANTECEDENTES

Intel es una empresa dedicada al desarrollo de nuevas tecnologías en la industria de los semiconductores, es uno de los fabricantes de chips más grandes del mundo (Kolmar, 2021), así como de otros componentes electrónicos; es de esencial importancia que sus productos tengan la más alta calidad.

Es trabajo del equipo de validación post-silicio, asegurar que los chips de servidores y clientes que salen de las fábricas de Intel funcionan precisamente como fueron diseñados, antes de que estos sean expedidos en volumen, y esta es una parte crítica del proceso de manufactura de Intel (Kirsch, 2020).

Históricamente Intel se ha tardado 10 trimestres o cuartos (Q) en la validación post-silicio de los proyectos, en la Figura 1 se puede ver cómo es la ocupación del personal por trimestre, en el primero solamente se tiene al 50% del total del equipo haciendo actividades de validación para este proyecto, en el segundo trimestre ya se tiene al 75%, y los siguientes seis trimestres se tiene la capacidad completa del equipo involucrado, finalmente los últimos dos trimestres vuelve a bajar el porcentaje de empleados involucrados (Garcia Bedoy Torres, 2021).

Figura 1. Personal asignado por trimestre
Elaboración propia

Debido a que la validación post-silicio es la última etapa antes de que los productos sean lanzados al mercado, el cumplimiento de los tiempos de validación es un factor que influye de gran manera en el cumplimiento de las fechas propuestas para el lanzamiento del producto.

En la Tabla 1 se muestra un desglose de las actividades que se realizan durante la validación post-silicio en Intel a lo largo de los trimestres dedicados para este proceso, esta información se obtuvo de una entrevista que se tuvo con un ingeniero principal de Intel, este es un cargo importante dentro de la empresa (Garcia Bedoy Torres, 2021).

Tabla 1. Actividades de validación post-silicio en Intel

Actividad	Trimestre
Desarrollo de planes de validación	Q1
Desarrollo de contenido para pruebas	Q2
Preparación para PO y VV	Q3
PO e inicio de VV	Q4
Ejecución de pruebas y depuración en VV	Q5-Q6
PO de la siguiente versión del producto	Q7
Ejecución de pruebas y depuración en VV	Q8
QS	Q9
PRQ	Q10

Elaboración propia

Como se puede ver, en el primer trimestre se lleva a cabo el desarrollo de los planes de validación, es aquí donde se definen los casos de prueba funcionales necesarios para asegurar el correcto funcionamiento del producto, y que serán ejecutados en las fases posteriores de validación.

En Intel, esta actividad de definición de casos de prueba se ha venido haciendo de manera manual, los ingenieros de validación definen estos casos de prueba funcionales y para esto deben revisar distintas bases de datos y documentos en los que se encuentra la información sobre los requerimientos que debe cumplir el nuevo producto. Estas bases de datos y documentos son: el "Landing Zone", la base de datos de requerimientos funcionales y los documentos de especificaciones de arquitectura de hardware (HAS por sus siglas en inglés) (Merrill, 2017).

Esta actividad de revisión de documentos y bases de datos por parte de los ingenieros debe ser exhaustiva para asegurar que no se escape ningún funcionamiento que deba ser validado, es por lo tanto una actividad crucial que definirá en gran parte el éxito o fracaso en encontrar errores en la última etapa antes de liberar el producto. Los casos de prueba deben de ser efectivos, deben encontrar muchos errores y defectos.

Es por esto, que esta actividad tan importante de generación de casos de prueba no se debería confiar a la suerte de que los ingenieros logren encontrar toda la información relevante a través de todas las fuentes antes mencionadas y al mismo tiempo se debería buscar optimizar el tiempo utilizado en esta búsqueda exhaustiva, que como se vio en la Tabla 1, consume un trimestre de la mitad de los ingenieros de validación asignados a un proyecto.

1.2 DEFINICIÓN DEL PROBLEMA

El primer problema que se encuentra en la manera en cómo se han definido los casos de prueba anteriormente, es la dispersión de información sobre el funcionamiento esperado del producto que se va a validar, esto resulta en un proceso tardado y complejo de búsqueda y que afecta principalmente a los ingenieros de poca experiencia. Es tardado porque los ingenieros dedican un trimestre completo a la búsqueda de toda la información sobre el funcionamiento esperado del nuevo sistema como se muestra en la Tabla 1, y es complejo porque la información está dispersa en distintos documentos y bases de datos que suelen ser muy extensos, esto es, entre cientos y miles de páginas, y los ingenieros de validación deben revisar todos estos documentos y bases de datos para extraer de ahí los casos de prueba. Aun así puede ocurrir que la información sobre los requerimientos funcionales esté incompleta.

Enseguida se encuentra el problema de una actividad susceptible a errores humanos, suele ocurrir que incluso teniendo la información de los requerimientos funcionales completa, no se generan todos los casos de prueba necesarios para validarlos. Hasta ahora no se ha establecido el uso de alguna estrategia para la generación de los casos de prueba, lo único que se conoce entre los ingenieros es que se deben hacer pruebas positivas y pruebas negativas; así como algunas pruebas de seguridad, este es un conocimiento que los ingenieros desarrollan a través del tiempo y algunas veces es adquirido de otros ingenieros con más experiencia.

Así es como se busca responder a la pregunta ¿Es posible automatizar este proceso de generación de casos de prueba?

1.3 JUSTIFICACIÓN

La validación en la etapa de post-silicio es crítica debido a dos razones: el plazo de tiempo se debe cumplir de manera estricta ya que es la última etapa antes de comenzar la producción en masa para el lanzamiento del producto que ya se ha planeado anteriormente en base a la economía de mercado, el retraso puede significar la pérdida desde millones a miles de millones de dólares y en algunos casos la pérdida del mercado para el producto totalmente; la otra razón es el alto costo de que se escapen fallos en esta última etapa, resultando en el lanzamiento de un producto que no funciona bien y que se tendrán que perder millones de dólares en ingresos debido a la retirada del producto y el impacto en la reputación de la compañía (Mishra, Morad, Ziv, & Ray, 2017).

De esta manera, la creación de los planes de prueba es muy importante ya que sin duda constituye la actividad más crítica y fundamental de preparación para la validación de post-silicio (Mishra, Morad, Ziv, & Ray, 2017), esto es, la creación y diseño de casos de prueba efectivos. Si bien, la validación no puede garantizar la ausencia de todo error porque es imposible una validación completa debido a las restricciones de costos y tiempo (para validar exhaustivamente un sistema se podrían necesitar decenas de años), la estrategia obvia es entonces, tratar de hacer las pruebas lo más completo posible (Myers, 2004).

Entre las actividades de validación, la generación de casos de prueba es una de las tareas más intelectualmente demandantes y también es uno de los retos más críticos, ya que tiene un alto impacto en la efectividad y eficiencia del proceso completo de validación (Anand, et al., 2013).

Con base en lo anterior, la automatización en la generación de casos de prueba trae como beneficios el proveer una cobertura consistente de los requerimientos del sistema y el generar casos de prueba de una manera objetiva e imparcial, además de liberar a los ingenieros de validación de la gran parte de tedio de esta

actividad. Además de que el tiempo necesario para la generación de las pruebas se reduce, lo cual permite manejar más fácilmente los cambios inevitables en los requerimientos (Cunning & Rozenblit, 1999).

Para el grupo de validación en Intel, esta reducción del tiempo de generación de casos de prueba representa alrededor de un 6% del total de la inversión en personal total a lo largo del proyecto, esto se presenta en la Figura 1, al reducir los costos que se generan en el primer trimestre con la capacidad del equipo al 50%.

1.4 OBJETIVOS

1.4.1 Objetivo general

Automatizar la generación de casos de prueba funcionales que provean cobertura completa de los requerimientos funcionales documentados de un sistema de hardware.

1.4.2 Objetivos específicos

a. Definir un lenguaje de escritura para los requerimientos funcionales de hardware que permita estandarizar su formato.

b. Encontrar una estrategia de validación efectiva para generar casos de prueba en el grupo de validación de hardware de Intel.

c. Desarrollar una herramienta que sea capaz de analizar los requerimientos escritos, e implementando la estrategia de validación genere los casos de prueba.

1.5 HIPÓTESIS

La generación automática de casos de prueba para requerimientos funcionales, utilizando una estrategia de validación eficiente y adecuada para el grupo de validación de post-silicio en Intel, ayudará a reducir los costos totales de validación de un proyecto en un 6%, al reducir el tiempo utilizado en la generación de estos casos de prueba.

2. MARCO TEÓRICO

Como marco de este trabajo se tienen algunos conceptos clave, teorías y el conocimiento que se ha construido alrededor de la ingeniería de requerimientos, validación de sistemas de hardware y generación de casos de prueba. También se presentan los resultados de trabajos y aportaciones que otros autores han hecho con el fin de automatizar los procesos de análisis, procesamiento y validación de requerimientos, así como algunos trabajos que se han hecho para la generación de casos de prueba de manera automática; todo esto adecuándolo al contexto de validación post-silicio de hardware en Intel.

2.1 REQUERIMIENTOS

El proceso de ingeniería de requerimientos es una etapa temprana importante en el ciclo de vida de los sistemas, y se dice que la principal causa de falla de los proyectos son los requerimientos deficientes (Scott, Cook, & Kasser, 2004). También algunos estudios de campo que se hicieron en algunas compañías de Australia sobre las prácticas de ingeniería de requerimientos y las áreas de mejora, concluyeron que los problemas son más organizacionales y no técnicos por naturaleza, por ejemplo, administración de la documentación y manejo de incertidumbre. (Martin, Aurum, Jeffery, & Paech, 2002)

La meta de la ingeniería de requerimientos es desarrollar y mantener un buen conjunto de requerimientos, las actividades llevadas a cabo durante este proceso se muestran en la Figura 2. La práctica ha demostrado que esto es difícil, las dificultades se derivan de varias fuentes, primeramente, los requerimientos son comúnmente escritos en lenguaje natural, el cual es inherentemente ambiguo y puede ser mal entendido (Scott, Cook, & Kasser, 2004).

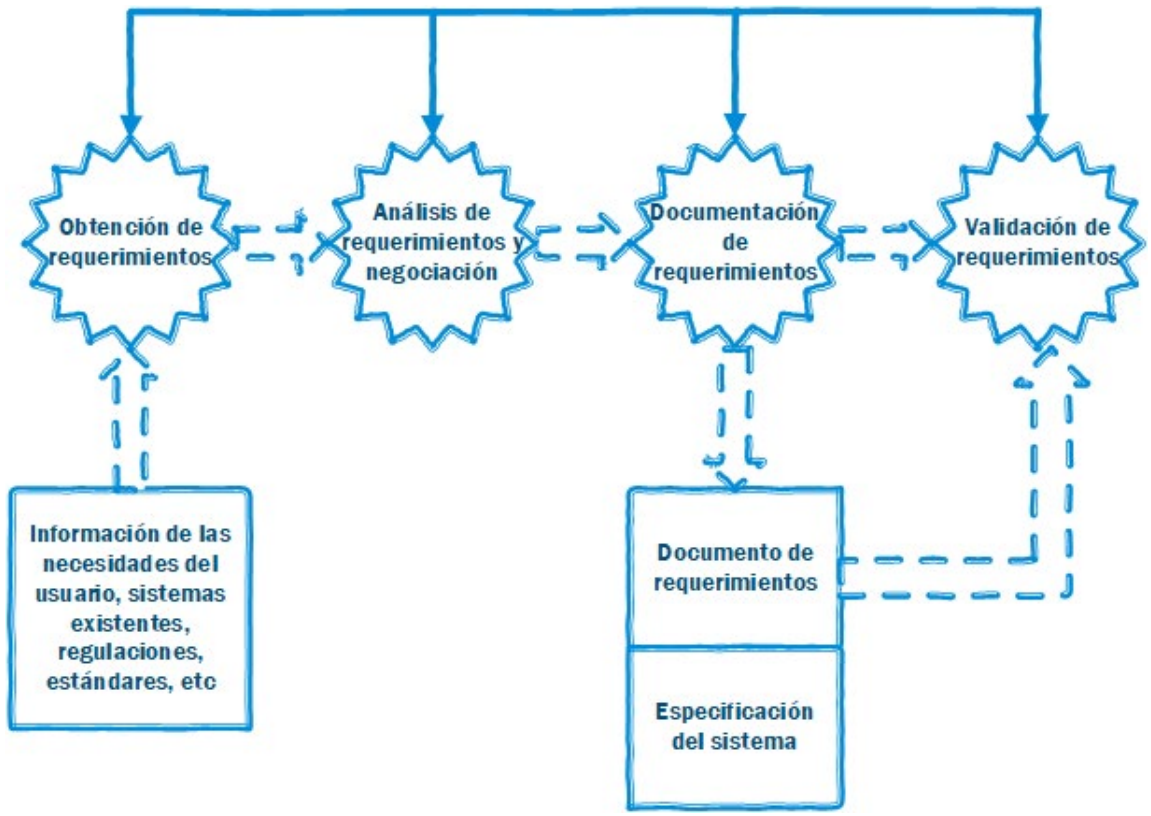


Figura 2. Modelo del Proceso de Ingeniería de Requerimientos lineal

(Martin, Aurum, Jeffery, & Paech, 2002)

Dentro del ámbito del hardware también se les da importancia a los requerimientos, a través de la experiencia se ha encontrado que algunas formas de capturarlos, expresarlos y organizarlos son más efectivas que otras, y esto resulta en hacer más eficiente y eficaz el proceso de verificación e incrementar la cobertura de pruebas del diseño. Se recomienda que los requerimientos sean escritos de forma que puedan ser verificados (Fulton & Vandermolen, 2017).

Uno de los problemas que se mencionan es que la forma en que los requerimientos se usan y en cómo son escritos, es frecuentemente influenciada por la cultura de ingeniería prevaleciente en la empresa. Las habilidades de escritura de los requerimientos son aprendidas de ejemplos en la misma empresa: los ingenieros usan documentos de requerimientos de programas anteriores como punto de entrada, y pueden también revisar documentos de los clientes y de la industria. De

esta manera el ingeniero asume que los requerimientos están bien escritos y que los debería escribir de una manera similar o idéntica (Fulton & Vandermolen, 2017).

2.1.1 Tipos de requerimientos para dispositivos electrónicos

De todos los requerimientos que existen para dispositivos electrónicos, se mencionan algunas categorías que fueron encontradas en base a observaciones sobre lo que los ingenieros escriben en los documentos de requerimientos y estos son los resultados: (Fulton & Vandermolen, 2017).

- Requerimientos funcionales
- Requerimientos de implementación
- Requerimientos de aplicación
- Requerimientos de entradas
- Requerimientos indirectos
- Requerimientos referidos
- Requerimientos globales
- Requerimientos derivados

Se dice que los requerimientos funcionales son el tipo de requerimientos más recomendado, estos describen el comportamiento y pueden ser expresados en cómo una salida responde a una entrada (o a un evento de tiempo). Además, los requerimientos funcionales son verificables, expresados en términos cuantificables, y son independientes del diseño y la implementación del hardware (Fulton & Vandermolen, 2017).

2.1.2 Recomendaciones para escribir requerimientos de hardware

En la industria de la aviación se requiere de mucha precisión por la criticidad que implica el que dependan vidas humanas de un sistema libre de fallos, las siguientes son recomendaciones que surgen de esta industria para escribir requerimientos de dispositivos electrónicos bien formados (Fulton & Vandermolen, 2017):

    - Usar la palabra “shall” para identificar un requerimiento.
- Hacer cada requerimiento único.
- Asignar un identificador o etiqueta única a cada requerimiento.
- Especificar lo que se hace, no cómo se hace.
- Especificar un comportamiento observable en una salida en el requerimiento.
- Describir cómo responden las salidas a entradas válidas.
- Describir cómo responden las salidas a entradas inválidas.
- Describir cómo responden las salidas a eventos relacionados al tiempo.
- Hacer cada requerimiento especifico y conciso.
- Expresar los requerimientos en sentido positivo, no requerimientos negativos o “shall not”.
- Para PDLs o circuitos, expresar los requerimientos en términos de comportamiento a nivel de pin.
- Usar descripciones de salidas que puedan ser observadas y entradas que puedan ser controladas.
- Especificar los requerimientos de manera cuantificable para que los resultados de las pruebas no resulten confusos.

Se recomienda también usar una plantilla para escribir los requerimientos, ya que eso permitirá que los nuevos miembros del equipo puedan contribuir al proyecto prontamente ya que se tendrá un contenido y formato estandarizado, además de que la plantilla ayuda al escritor de requerimientos a pensar en todos los aspectos de la funcionalidad. Los estándares definen el significado y uso de operadores modales de necesidad y posibilidad como “should”, “shall”, “can”, “will”, “may” y “might” (Fulton & Vandermolen, 2017).

Un ejemplo de requerimiento funcional de hardware escrito en inglés se muestra en la Figura 3.

"DAT_OUT(11:0) shall output the 12-bit unsigned binary equivalent of the AIN_1 analog input."

Figura 3. Ejemplo de requerimiento funcional de hardware

(Fulton & Vandermolen, 2017)

Otra recomendación encontrada para la escritura de requerimientos es el uso de lenguajes formales, se dice que es un primer y substancial paso hacia requerimientos de alta calidad, ya que se hacen explicitas algunas preguntas sutiles que de otra manera estarían escondidas en la ambigüedad de un lenguaje natural (Pill, et al., 2006). Y también que proveen un método preciso para especificar requerimientos de aplicación y funcionalidad del sistema que podrían ser afectados negativamente por fallos de post-silicio (Mitra, Seshia, & Nicolici, 2010).

2.2 VALIDACIÓN DE HARDWARE

El proceso de validación juega un papel muy importante en el aseguramiento de la calidad al confirmar que el hardware cumple con los requerimientos (Fulton & Vandermolen, 2017). La validación del hardware consiste en confirmar que una unidad de hardware objetivo, la cual puede incluir una combinación compleja de componentes, satisface su propósito especifico y sirve también como una ventana temprana de interoperabilidad funcional (Faria & Salazar, 2015).

Sin embargo, la validación es un proceso de alta complejidad y demanda de recursos, se estima que el 70% del total de recursos y tiempo, del desarrollo en general, son utilizados en validación (Mishra, Morad, Ziv, & Ray, 2017), a la vez que es visto en gran medida como un arte con muy pocas soluciones sistemáticas (Mitra, Seshia, & Nicolici, 2010).

Este proceso de validación se divide en tres etapas dependiendo del objeto que se usa para validar como más adelante se explica. Y son nombradas de distintas maneras dependiendo del autor o la empresa donde se hace la validación. Así por ejemplo un autor (Mozhikunnath, 2016) las llama etapas de verificación, validación y prueba, otros autores las llaman etapas de pre-silicio, post-silicio y on-field debug (Mishra, Morad, Ziv, & Ray, 2017) y hay otro artículo donde también se les llama pre-silicio y post-silicio a las primeras dos etapas, pero en la tercer etapa solo se consideran las pruebas en manufactura (Mitra, Seshia, & Nicolici, 2010). En este trabajo se llamará como en el último artículo ya que en Intel también son nombradas de este modo. En la Figura 4 se representa gráficamente cómo se dan estas etapas a través del ciclo de vida del diseño del SoC. Y en la Tabla 2 se presenta una comparativa de las características entre estas distintas etapas de validación.

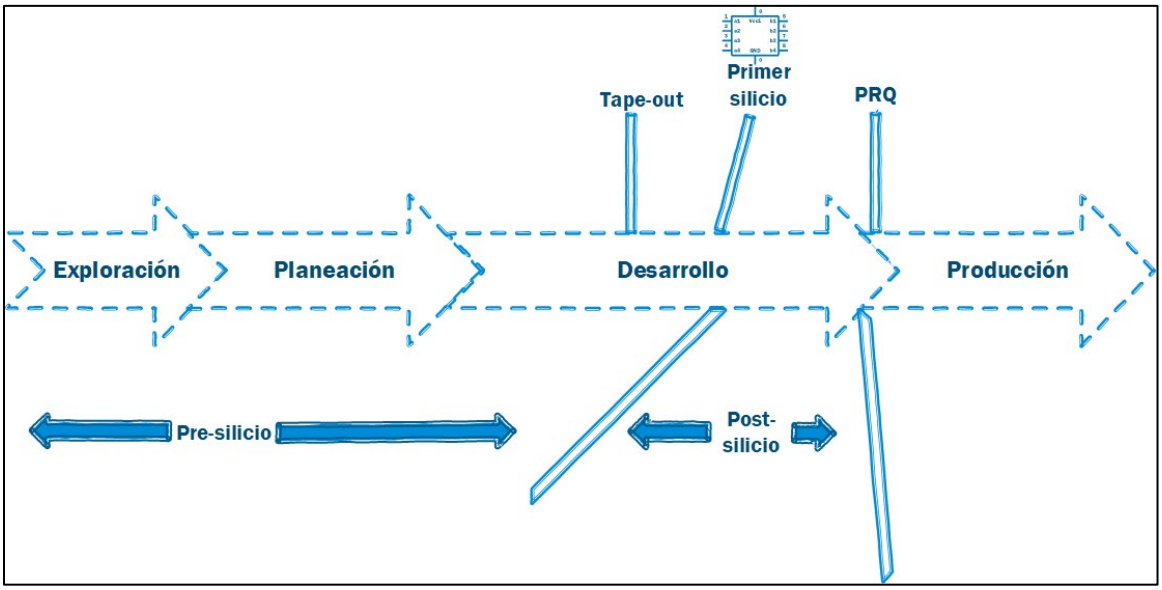


Figura 4. Etapas en el ciclo de vida de un SoC

(Mishra, Morad, Ziv, & Ray, 2017)

Tabla 2. Comparación cualitativa de pre-silicio vs post-silicio vs pruebas de manufactura

Verificación pre-silicio	Validación post-silicio	Pruebas de Manufactura
Excelente controlabilidad y capacidad de observación porque todas las señales pueden ser accedidas.	Insuficiente controlabilidad y capacidad de observación debido al limitado accesos a las señales internas. El escáner DFT es útil para algunos casos cuando una falla es repetible.	Controlabilidad y capacidad de observación principalmente a través de escaneo DFT.
Efectos físicos complejos son difíciles de modelar.	Integridad de señal, variaciones de proceso y no determinismo.	Existen varios modelos de defectos.
Simulación de diseños completos es muy lenta; verificación formal aplicable solo de forma selectiva.	La velocidad es ordenes de magnitud más rápida que en simulación.	Generalmente muy rápido (algunos segundos o minutos por chip).
Existen algunas métricas (p. ej. Cobertura de código, cobertura de aserciones, cobertura de mutaciones).	Métricas de cobertura para validación de post-silicio: Es una pregunta de investigación abierta.	Métricas de pruebas: (p. ej. Stuck-at, transition, cobertura N-detect) son ampliamente usados.
Corrección de fallos no es caro.	Corrección de fallos puede ser caro.	Corrección de fallos no es el objetivo principal.

(Mitra, Seshia, & Nicolici, 2010)

2.2.1 Pre-silicio o Verificación de SoC

Es un proceso en el cual un diseño es probado o verificado contra una especificación de diseño antes de que el circuito sea mandado a manufacturar por primera vez (tape-out). Esto sucede junto con el desarrollo del diseño y puede empezar desde el momento en que se definen la arquitectura o microarquitectura del diseño. La meta principal de la verificación es asegurar la corrección funcional del diseño antes del tape-out (Mozhikunnath, 2016). Durante esta etapa se usan

metodologías para verificación como simulación, verificación formal, emulación/prototipado en FPGA, chequeos estáticos y dinámicos, para verificar todos los aspectos del diseño (Mozhikunnath, 2016), y el objeto de la verificación es típicamente un modelo del diseño en lugar del silicio (Mishra, Morad, Ziv, & Ray, 2017). El principal medio de verificación de diseños de hardware continúa siendo la simulación de software que se hace en esta etapa de pre-silicio (Tasiran & Keutzer, 2001).

2.2.2 Post-silicio o Validación de SoC

Es el proceso en el cual el diseño manufacturado (chip) es probado en cuanto a todo lo funcional en una configuración de laboratorio. Esto se hace usando el chip real ensamblado en una plataforma de referencia junto con los demás componentes que son parte del sistema para el cual el chip fue diseñado. La meta es validar todos los casos de uso del chip que un cliente puede eventualmente tener en una implementación real y calificar el diseño para todos estos modelos de uso (Mozhikunnath, 2016), así como asegurar que el diseño del silicio funciona apropiadamente bajo condiciones operativas reales e identificar (y arreglar) errores que se pudieron haber escapado durante la verificación de pre-silicio (Mishra, Morad, Ziv, & Ray, 2017).

Las simulaciones de post-silicio se ejecutan a las velocidades de reloj reales permitiendo la exploración de estados de diseño más profundos de los que se permiten en pre-silicio. Desafortunadamente, post-silicio está limitado por en la capacidad de observación: solo unas pocas de las miles de señales internas se pueden observar durante la operación normal de un chip (Ray & Hunt, 2009).

La validación post-silicio representa uno de los componentes más importantes, caros y complejos de la metodología de validación del diseño de SoC (Mishra, Morad, Ziv, & Ray, 2017).

2.3 CRITERIOS PARA SELECCIONAR CASOS DE PRUEBA DE HARDWARE

Los casos de prueba son el conjunto de entradas aplicadas al hardware o diseño con un conjunto correspondiente de resultados esperados y los criterios de éxito/fracaso (Fulton & Vandermolen, 2017).

El proceso de validación puede ser optimizado y hacerse más eficiente con requerimientos que expresan comportamiento a nivel de pin. Una primera oportunidad para mejorar el proceso es crear los casos de prueba a partir de los requerimientos tan pronto como los requerimientos estén maduros y hayan sido revisados. Esto permite que las actividades de validación y simulación empiecen incluso antes de que el diseño sea creado, otra idea es presentar los casos de prueba de manera que fácilmente se pueda revisar si los requerimientos han sido verificados exhaustivamente y para esto propone presentarlo en tablas con las entradas y salidas como columnas y el conjunto de valores de entradas y resultados esperados a través de cada fila (Fulton & Vandermolen, 2017).

En el contexto de validación funcional de hardware, los casos de prueba únicamente provienen de los requerimientos del hardware y no del diseño, porque si se obtuvieran casos de prueba del diseño, lo único que se estaría probando es que el diseño es el diseño, pero no validan que el diseño cumple con la funcionalidad deseada (Fulton & Vandermolen, 2017).

2.3.1 Técnicas de pruebas

La meta es validar lo suficiente para asegurarse que la probabilidad de falla debido a errores escondidos es suficientemente baja para ser aceptada, se necesita de métricas que ayuden a determinar esa aceptación (Chilenski, Hayhurst J, Veerhusen, & Rierson, 2001).

En el ámbito del desarrollo de software, las pruebas de software se obtienen usando dos estrategias: caja negra y caja blanca; las pruebas de caja negra son frecuentemente usadas para validación (¿estamos construyendo el software

correcto?) y las pruebas de caja blanca son usadas para verificación (¿estamos construyendo el software correctamente?) (Nidhra & Dondeti, 2012).

2.3.1.1 Pruebas de caja negra

Las pruebas de caja negra también se les llama pruebas funcionales, es una técnica para diseñar casos de prueba basado en la información de las especificaciones, donde el validador no tiene acceso al código fuente, solamente se enfoca en las salidas generadas en respuesta a las entradas seleccionadas y a las condiciones de ejecución (Nidhra & Dondeti, 2012). Para este método se utilizan los requerimientos del sistema (W. & Y., 2004).

Las técnicas de caja negra son las siguientes:

- Partición de clases equivalente
- Análisis de valores en la frontera
- Tablas de decisión
- Diagramas de transición de estados
- Arreglos ortogonales
- Técnica de pares

2.3.1.2 Pruebas de caja blanca

Las pruebas de caja blanca también se llaman pruebas estructurales, estas se diseñan conociendo el código fuente y se concentran en los flujos de control y flujos de datos de los programas, con esta técnica es posible probar cada rama y decisión en el programa (Nidhra & Dondeti, 2012).

Algunas de las ventajas de estas pruebas de caja blanca son que encuentran errores escondidos en el código y ayuda a remover líneas extra de código. La principal desventaja es que es muy caro (Ehmer Khan, 2011).

Las técnicas de caja blanca son:

1. Pruebas de caja blanca estáticas

   a. Chequeo de escritorio
   b. Inspección de código
   c. Inspecciones formales
2. Pruebas de caja blanca estructurales
   a. Pruebas de flujo de control/cobertura
   b. Pruebas de camino básico
   c. Pruebas de ciclos
   d. Pruebas de flujo de datos

Pruebas de cobertura

Dentro del tipo de pruebas estructurales se encuentran las pruebas de cobertura. La cobertura se refiere a la medida en que una actividad de verificación ha cumplido sus objetivos, de esta manera, la cobertura estructural se refiere a la medida de la estructura de código que ya se ha analizado por las pruebas basadas en requerimientos, y se mide en términos de invocaciones a sentencias, expresiones booleanas evaluadas y construcciones de control ejercitadas (Chilenski, Hayhurst J, Veerhusen, & Rierson, 2001). La cobertura estructural ayuda a asegurar que el proceso de verificación basado en requerimientos ha puesto atención a todas las características de la implementación (Chilenski J. J., 2001). La Tabla 3 muestra las definiciones de medidas de cobertura estructural basadas en el flujo de control.

Tabla 3. Tipos de cobertura estructural

Criterio de cobertura	Cobertura de sentencias	Cobertura de decisiones	Cobertura de condiciones	Cobertura de Condición/ Decisión	MC/DC	Cobertura de múltiples condiciones
Cada punto de entrada y de salida ha sido invocado		●	●	●	●	●
Cada sentencia ha sido invocada	●					
Cada decisión ha tomado todas sus posibles salidas		●		●	●	●
Cada condición en una decisión ha tomado todas sus posibles salidas			●	●	●	●
Cada condición en una decisión ha sido demostrada que afecta de manera independiente la salida de la decisión					●	●
Cada combinación de las condiciones dentro de una decisión ha sido invocada						●

(Chilenski, Hayhurst J, Veerhusen, & Rierson, 2001)

Cobertura de sentencias

El ideal de este método es que todas las sentencias se ejecuten por lo menos una vez, sin embargo, ofrece una cobertura pobre del flujo de control de los sistemas (Anderson, 2011).

Cobertura de decisiones

En este método es suficiente que cada rama sea ejecutada por lo menos una vez y esto resultará en que también se ejecutarán todas las sentencias (Pezze & Young, Structural Testing, 2007).

Cobertura de condiciones

Este analiza las condiciones de un predicado compuesto más detenidamente y examina cada condición de manera individual (Sarabi, 2012). Se definen cuatro variedades de cobertura de condiciones (Pezze & Young, Software Testing and Analysis: Process, Principles, and Techniques, 2008): cobertura de condición básica, cobertura de decisión y condición, MCDC y cobertura de condición compuesta.

- **Cobertura de condición básica**
  Esta cobertura requiere que cada condición dentro de un predicado o una expresión booleana compuesta sea evaluada a verdadero y falso. Esta métrica cubre todos los sentencias pero no todas las ramas (Sarabi, 2012).

- **Cobertura de decisión y condición**
  Esta combina la cobertura de decisión o ramas con la cobertura de condición básica, por lo tanto, además de evaluar cada condición a falso y verdadero, el predicado de control también debe ser evaluado a falso y verdadero (Sarabi, 2012).

- **Cobertura de condición múltiple**
  Este criterio requiere casos de prueba que aseguren cada posible combinación de entradas en una decisión, es decir, requiere pruebas exhaustivas. En teoría, este criterio de cobertura de condición múltiple es el

criterio de cobertura más deseable; pero es impráctico para muchos casos. Para una decisión con *n* entradas, este criterio requiere 2^n casos de prueba (Chilenski, Hayhurst J, Veerhusen, & Rierson, 2001).

- **MC/DC**

  El criterio de cobertura modificada de condición/decisión (MC/DC por sus siglas en inglés) requiere que sea mostrado cómo afecta de manera independiente, el cambio de una condición, en la salida o resultado de una decisión. El requerimiento de independencia asegura que el efecto de cada condición sea probado en relación con las demás condiciones y por lo tanto tenga el efecto apropiado. Este criterio requiere por lo general un mínimo de *n*+1 casos de prueba para una decisión con *n* entradas. Por ejemplo para (A or B), los casos de prueba (TF), (FT), and (FF) proveen MC/DC (Chilenski, Hayhurst J, Veerhusen, & Rierson, 2001).

De acuerdo con Chilenski y Miller (Chilenski J. J., 1994), el demostrar que cada condición lógica afecta independientemente la salida de la decisión, requiere un conjunto de pruebas mínimas específicas para cada operador lógico. En este trabajo se consideran únicamente los operadores ***and*** y ***or*** porque son los operadores que se utilizan en las condiciones que tienen los requerimientos.

Operador *and*

Las pruebas mínimas para cumplir MC/DC para un operador *and* requiere lo siguiente:

1. Todas las entradas sean verdaderas con la salida observada que sea verdadera también. Esto requiere un caso de prueba.
2. Cada una de las entradas sea puesta exclusivamente en falso con la salida observada que sea falsa. Esto requiere *n* casos de prueba para un operador *and* con *n* entradas.

Operador *or*

Las pruebas mínimas para cumplir MC/DC para un operador *or* requiere lo siguiente:

1. Todas las entradas sean falsas con la salida observada que sea falsa también. Esto requiere un caso de prueba.
2. Cada una de las entradas sea puesta exclusivamente en verdadero con la salida observada que sea verdadera. Esto requiere *n* casos de prueba para un operador *or* con *n* entradas.

2.4 AUTOMATIZACIÓN DE LA GENERACIÓN DE CASOS DE PRUEBA

La generación de casos de prueba es una de las tareas de trabajo más intenso en el proceso de validación y que tiene mayor impacto en la efectividad y eficiencia de los sistemas (Anand, y otros, 2013). La generación de casos de prueba de manera automática toma importancia ya que ayuda en la minimización de tiempo y costo en el desarrollo de casos de prueba (Sahoo, Ojha, Mohapatra, & Manas, 2016).

2.4.1 Casos de prueba de software

La generación automática de casos de prueba ha sido uno de los temas más activos por varias décadas en la investigación de pruebas de software, resultando en distintas estrategias y herramientas (Anand, y otros, 2013), los que se encontraron para este trabajo se muestran a continuación:

Automatización a partir de modelos de objetos

A inicios de los 90's, cuando la programación orientada a objetos comenzó a florecer (Purdum, 2008), se hizo un trabajo en el que se integraron un conjunto de herramientas que permiten modelar software con el paradigma orientado a objetos, esto significa que se definen las especificaciones del software por medio de diagramas, y después permite usar esos modelos para generar los casos de prueba (Poston, 1994) siempre y cuando esos modelos contengan unas anotaciones especiales sobre dominio de los datos. Este trabajo se probó con un

pequeño experimento y se logró demostrar que reduciría un mes de trabajo consumido en la generación de casos de prueba.

Automatización a partir de algoritmos evolutivos

También se ha investigado la generación de datos de prueba al azar y casos de prueba utilizando algoritmos evolutivos como el algoritmo de la colonia de abejas (BCA por sus siglas en inglés), en el cual las abejas continúan buscando diferentes fuentes de comida de acuerdo con la calidad para reemplazar la anterior con una nueva solución mejorada. O también el algoritmo de búsqueda de armonía (HS por sus siglas en inglés) en el cual se busca un valor optimo a través de un proceso de improvisación musical. Finalmente el algoritmo de optimización de la nube de partículas (PSO por sus siglas en ingles) en el cual las partículas buscan su comida a través de cambios de posición y velocidad para optimizar su búsqueda (Sahoo, Ojha, Mohapatra, & Manas, 2016).

Técnicas de automatización en base a los tipos de artefactos utilizados

a) Pruebas estructurales usando ejecución simbólica

Es una técnica que analiza el código de un programa para generar automáticamente los casos de prueba, la ejecución simbólica consiste en usar valores simbólicos en lugar de valores concretos como entradas para el programa (King, 1975) y de esta manera se encuentran los caminos que puede tomar el código de acuerdo con los valores de las entradas.

b) Pruebas basadas en modelos

Esta técnica se enfoca en las pruebas de tipo de caja negra o funcionales, las cuales aceptan entradas y producen salidas. El algoritmo de selección de prueba deriva los casos de prueba del modelo seleccionando un conjunto de secuencias

especificadas por el modelo, usando un criterio de prueba basado en una hipótesis de prueba (Anand, y otros, 2013).

c) Pruebas combinatorias

El enfoque en esta estrategia es la selección de una muestra de parámetros de entrada que cubren un conjunto de combinaciones prescritas de los elementos a ser probados. Estas pruebas combinatorias se originan del campo de la estadística llamado diseño de experimentos (Fisher, 1971).

d) Pruebas aleatorias adaptables

Esta estrategia propone que los casos de prueba deben ser dispersados de manera uniforme a lo largo de los dominios de las entradas (Chen, 2010).

e) Pruebas basadas en búsqueda (Anand, y otros, 2013)

Se usan algoritmos de automatización para automatizar la búsqueda de datos de pruebas que maximicen el alcance de las metas de las pruebas y que minimicen los costos (Harman, 2001).

2.4.2 Casos de prueba de hardware

En el campo del hardware también se han hecho algunas investigaciones para la generación automática de casos de prueba como en (Jantsch, Notbauer, & Albretch, 2000), donde se hizo el esfuerzo para generar casos de pruebas para validación en ambiente de simulación de sistemas embebidos a partir de los requerimientos y especificaciones, y para esto crearon un lenguaje formal para la especificación de los casos de pruebas. Esto se parece en parte a lo que en este trabajo se pretende hacer sobre la generación de casos de pruebas para hardware a partir de las especificaciones, sin embargo, difiere en que está

enfocado en el ambiente de verificación de simulación mientras que este trabajo se enfoca en la validación post-silicio, con el silicio real.

A continuación se muestran algunos proyectos que se parecen más en la finalidad de generar casos de prueba de manera automática a partir de las especificaciones para hardware en el ambiente de post-silicio:

Generador de casos de prueba a partir de especificaciones de requerimientos para sistemas embebidos de tiempo real

Cunning y Rozenblit realizaron un trabajo en el que proponen modelar el sistema en la forma de una máquina de estados finitos, usando un lenguaje de modelado formal, a partir de este modelo se ejecuta el generador de casos de prueba y este genera un árbol llamado el árbol de escenario, donde los vértices representan los estados del sistema y las aristas representan las transiciones entre los estados. Después todos los casos de prueba son extraídos del árbol de escenario, llevando a cabo un recorrido transversal. El problema que se menciona sobre este trabajo es que solamente es práctico para sistemas pequeños porque el árbol puede crecer y dejar de ser manejable para sistemas complejos (Cunning & Rozenblit, 1999).

Sistema ASSERT

La herramienta de ASSERT™ desarrollada dentro de General Electric, acepta requerimientos formalizados para después analizarlos en base a un modelo de dominios construido a partir de un lenguaje ontológico, después se generan casos de prueba basados en estos requerimientos. Para esta herramienta fue desarrollado un lenguaje formal para la captura de requerimientos que permite analizarlos usando métodos formales. (Siu et al, 2017).

Sistema TCG_UML

TCG_UML es otra herramienta que genera casos de prueba la cual acepta requerimientos expresados en forma de diagramas de actividad UML (Lenguaje Unificado de Modelado), usando la sintaxis de PlantUML (herramienta de código abierto) y a partir de ahí genera un árbol sintáctico que después se usa en la generación automática de casos de prueba (Espejel Salazar, 2019).

La plataforma TCG_UML es un sistema desarrollado por Espejel (Espejel Salazar, 2019) que consta de un analizador léxico, un analizador sintáctico, y un generador de casos de prueba; como entrada recibe un archivo con una gramática libre de contexto y el archivo de texto que se requiere analizar; y como salida genera casos de prueba.

Esta plataforma tiene la propiedad de ser configurable, lo que le permite aceptar cualquier lenguaje que genere la gramática libre de contexto introducida. El analizador sintáctico genera un árbol sintáctico del que después se generan casos de prueba.

En el trabajo de Espejel (Espejel Salazar, 2019), se describe cómo esta plataforma fue desarrollada para aceptar diagramas de actividad UML con la sintaxis de PlantUML, de ahí que su nombre contiene el acrónimo de UML, y genera casos de prueba a partir estos diagramas de actividad. En la Figura 5 se muestra un diagrama de componentes de este sistema TCG_UML (Espejel Salazar, 2019).

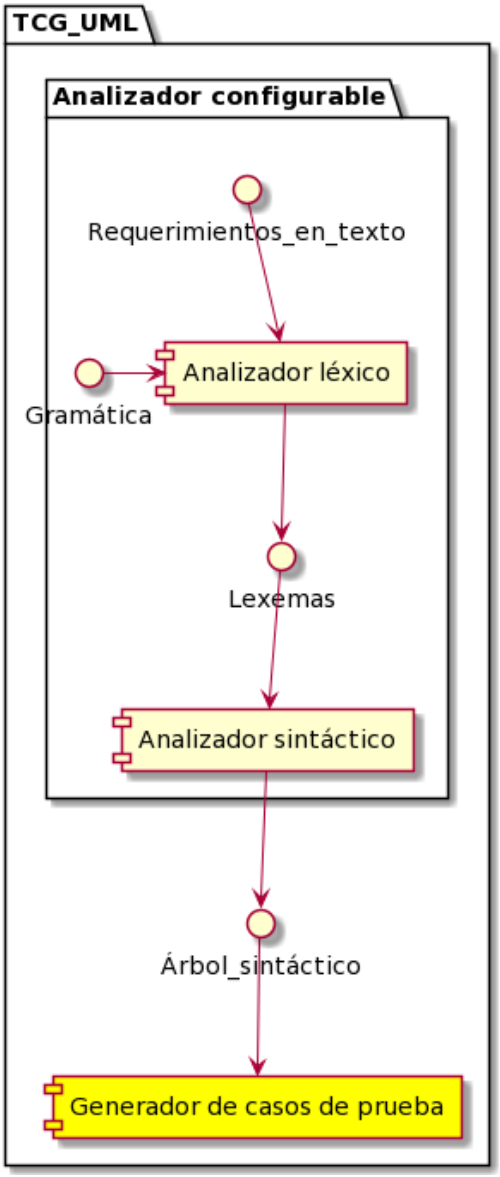


Figura 5. Diagrama de componentes de plataforma TCG_UML

Elaboración propia

Ya que en este trabajo se hace uso de los analizadores léxico y sintáctico de este sistema TCG_UML, en la siguiente sección se revisa la teoría sobre autómatas y lenguajes formales ya que es la base de estos analizadores.

2.5 TEORIA DE AUTÓMATAS Y LENGUAJES FORMALES

La teoría de autómatas es la rama de la ciencia de la computación que trata el diseño de dispositivos de computación abstractos y autopropulsados y que sigue una determinada secuencia de operaciones automáticamente (TutorialsPoint, 2016).

2.5.1 Historia breve

Antes de 1930 no había computadoras y Alan Turing introdujo una máquina abstracta que tenía todas las capacidades que tienen las computadoras de ahora.

Más tarde en 1940's y 1950's, unos tipos de máquinas simples llamados autómatas finitos fueron introducidos por algunos investigadores.

Después, en 1950's el lingüista N. Chomsky empezó su estudio de gramáticas formales, las cuales están muy relacionadas a los autómatas abstractos (Singh, 2019).

2.5.2 Lenguajes

El lenguaje de un alfabeto es un subconjunto de todas las cadenas que pueden ser formadas a partir del alfabeto (Singh, 2019).

Prácticamente hay dos vistas desde donde ver a los lenguajes (Gallier, 2020):

- Punto de vista de reconocimiento.
- Punto de vista de la generación.

Sin importar cómo veamos un lenguaje, típicamente se consideran dos cosas (Gallier, 2020):

1. La sintaxis, esto es, cuáles son las cadenas "legales" en ese lenguaje (¿cuáles son las reglas gramaticales?).

2. La semántica de las cadenas en el lenguaje, esto es, cuál es el significado o interpretación de una cadena.

Desde el punto de vista del reconocimiento, típicamente se asume que hay algún tipo de "caja negra", un autómata que toma una cadena como entrada y regresa dos posibles salidas (Gallier, 2020):

- Sí, la cadena es aceptada, lo cual significa que pertenece al lenguaje que estamos tratando de definir.
- No, la cadena es rechazada, lo cual significa que no pertenece al lenguaje.

Desde el punto de vista de la generación del lenguaje, son importantes los formalismos que especifican un lenguaje en términos de reglas que permiten la generación de cadenas "legales". El formalismo más común es una gramática formal (Gallier, 2020).

2.5.3 Gramáticas formales

Una gramática es un cuádruplo (Σ, V, S, P), donde:

1. Σ es un conjunto no vacío finito, llamado el alfabeto terminal. Los elementos de Σ son llamados los terminales.
2. V es un conjunto no vacío finito disjunto con Σ. Los elementos de V son llamados no terminales o variables.
3. $S \in V$ es un no terminal distinguido, llamado el símbolo inicial.
4. P es un conjunto finito de producciones o reglas de la forma:

$$\alpha \rightarrow \beta$$

donde a es una cadena de terminales y no terminales conteniendo al menos un no terminal y β es una cadena de terminales y no terminales (Jiag, Li, Ravikumar, & Regan, 2018).

Ejemplo: Para construir una gramática que describa enunciados en inglés, el alfabeto Σ contiene todas las palabras en inglés. V contiene los símbolos no

terminales que corresponden a los componentes estructurales de un enunciado en inglés como <enunciado>, <sujeto>, <predicado>, <sustantivo>, <verbo>, <artículo>, etc. El símbolo inicial seria <enunciado>. Algunas producciones típicas serán:

<enunciado> → <sujeto><predicado>

<sujeto> → <sustantivo>

<predicado> → <verbo><articulo><nombre propio>

<sustantivo> → Mary

<sustantivo> → algorithm

<verbo> → wrote

<artículo> → an

La regla **<enunciado>** → **<sujeto><predicado>** modela el hecho de que un enunciado puede consistir en un sujeto y un predicado. Las reglas **<sustantivo>** → **Mary** y **<sustantivo>** → **algoritmo** significan que ambos: "Mary" y "algorithm" son posibles sustantivos (Jiag, Li, Ravikumar, & Regan, 2018).

Las gramáticas formales fueron introducidas como tal por Post en 1943 y tuvieron antecedentes en el trabajo por Thue y otros. Sin embargo, el estudio de su uso rigoroso en la descripción de lenguajes formales y naturales no empezó sino hasta mediados de 1950´s (1956) por Chomsky (Jiag, Li, Ravikumar, & Regan, 2018).

Chomsky clasificó las gramáticas formales (y los lenguajes que éstas generan) de acuerdo con una jerarquía de cuatro niveles representada en la Figura 6 (Málaga Jurado, 2008).

Estas cuatro clases de gramáticas incrementan gradualmente las formas de las producciones.

La Gramática tipo 0 es una gramática sin restricciones

La Gramática tipo 1 es una gramática sensible al contexto, cada producción $\alpha \rightarrow \beta$ satisface $|\alpha| \leq |\beta|$.

La Gramática tipo 2 o libre de contexto, cada producción $\alpha \rightarrow \beta$ satisface $|\alpha| = 1$.

La Gramática tipo 3 o regular tiene una de las siguientes formas:

$$A \rightarrow cB, \quad A \rightarrow c, \quad A \rightarrow \epsilon$$

Donde A y B son no terminales (A=B está permitido) y c es una terminal (Jiag, Li, Ravikumar, & Regan, 2018).

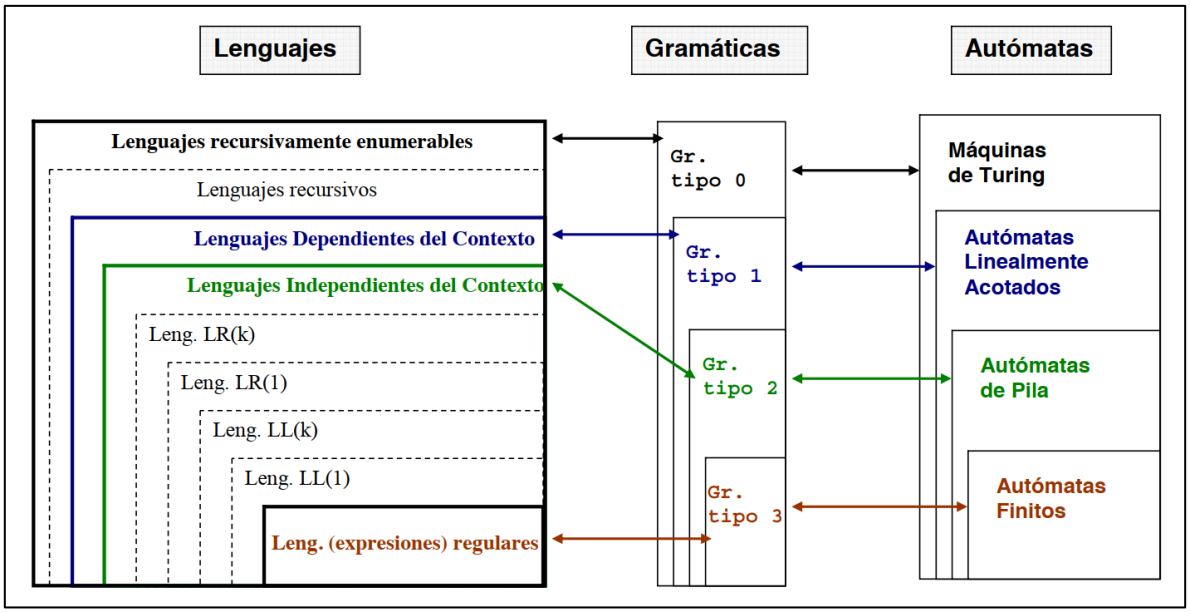


Figura 6. Jerarquía de gramáticas, lenguajes y autómatas

(Málaga Jurado, 2008)

2.5.4 Gramáticas libres de contexto

Las gramáticas libres de contexto son las que típicamente se usan para los lenguajes de programación modernos, ya que las demás gramáticas en niveles más elevados de la jerarquía de Chomsky hacen que el problema de palabra se vuelva insoluble algorítmicamente. (Gómez & Pardo, 2015).

Sin embargo el problema de palabra para gramáticas libres de contexto es decidible. Esto significa que es posible diseñar un algoritmo/programa que detecte

errores sintácticos y decida si un programa es sintácticamente válido para un lenguaje de programación. (Gómez & Pardo, 2015)

Las gramáticas libres de contexto tienen producciones de la forma como en la Ecuación 1. La conversión de A en v se realiza independientemente del contexto en que se encuentre A, de ahí su nombre (Málaga Jurado, 2008).

$$A \rightarrow v \; donde \; A \in \Sigma_{\mathrm{N}}, v \in \Sigma^{*}$$

Ecuación 1. Forma de producción de una gramática libre de contexto (Málaga Jurado, 2008)

2.5.5 Autómatas

Una máquina abstracta o autómata es un dispositivo teórico que recibe como entrada una cadena de símbolos y los procesa, cambiando de estado, de manera que genera una determinada salida. Los autómatas pueden servir, entre otras cosas, para determinar si una palabra pertenece o no a un determinado lenguaje (Málaga Jurado, 2008).

Como se ve en la Figura 6, existen cuatro tipos de autómatas: Autómatas Finitos, Autómatas de Pila, Autómatas Linealmente Acotados y Máquinas de Turing. Los Autómatas Finitos son el tipo más sencillo de autómatas y las Máquinas de Turing son los más complejos.

2.6 COMPILADORES

Un compilador es un software que convierte el código fuente en instrucciones binarias para la arquitectura del procesador, haciendo el código menos portable (Pahade & Dawale, 2019). Es por tanto, un programa complejo que trabaja en fases, cada fase transforma el programa fuente de una representación a otra. Son seis fases: análisis léxico, análisis sintáctico, análisis semántico, generador de código

intermedio, optimización de código y generación de código (Pahade & Dawale, 2019). En este trabajo solamente revisaremos las tres primeras fases referentes al análisis.

2.6.1 Análisis léxico

La primera fase del compilador es llamada analizador léxico o scanner, lee los caracteres del programa fuente, uno a uno, y va formando grupos de caracteres en secuencias significativas llamadas lexemas. Para cada lexema, el analizador léxico produce como salida un token de la forma (Aho, Lam, Sethi, & Ullman, 2007)

<nombre-token, valor-atributo>

que le pasa a la fase subsecuente, análisis sintáctico. En el token, el primer componente: nombre-token, es un símbolo abstracto que es usado durante el análisis sintáctico, y el segundo componente: valor-atributo, apunta a una entrada en la tabla de símbolos para este token (Aho, Lam, Sethi, & Ullman, 2007).

Para el diseño de los analizadores léxicos se utilizan los Autómatas Finitos.

2.6.2 Análisis sintáctico

El analizador sintáctico, también conocido como parser, recibe como entrada los tokens que genera el analizador léxico y comprueba si estos tokens siguen la sintaxis (Málaga Jurado, 2008). Es decir, consume los tokens y los agrupa en sentencias completas y expresiones, así como las palabras son agrupadas en enunciados en un lenguaje natural. Este parser es guiado por una gramática, la cual establece las reglas de composición formales de un lenguaje dado (Thain, 2020). Si no se han producido errores, la salida teórica de esta fase del compilador será un árbol sintáctico. (Málaga Jurado, 2008)

Para el diseño de los analizadores sintácticos se utilizan los Autómatas de Pila. (Málaga Jurado, 2008)

2.6.3 Análisis semántico

El analizador semántico determina si el significado de las instrucciones del programa es válido, si siguen las reglas del lenguaje. Por ejemplo comprueba que los argumentos de un operador pertenecen al conjunto de los operandos posibles o comprueba que los operandos son compatibles entre sí, etc. (Málaga Jurado, 2008).

3. PROCEDIMIENTO

En este trabajo se combinan las propuestas hechas para los sistemas de ASSERT (Siu, et al., 2017) y TCG_UML (Espejel Salazar, 2019), y se adaptan para ser compatibles con las necesidades del equipo de validación de servidores de Intel y lograr nuestro objetivo de generar casos de prueba de manera automática. En Intel los requerimientos son documentados en lenguaje inglés y la finalidad es procesarlos a partir de este formato, por lo que se tomó la idea del trabajo de ASSERT de estandarizar el formato para la escritura de requerimientos y para esto se diseñó una gramática libre de contexto. Para el procesamiento de requerimientos se toma el módulo de software del trabajo de Espejel, basado en teoría de compiladores, que procesa los archivos de entrada y genera un árbol sintáctico con los elementos clave de los requerimientos; en este trabajo se desarrolló un módulo de software que genera los casos de prueba funcionales a partir del árbol sintáctico que le entrega el módulo de Espejel.

3.1 ESTANDARIZACIÓN DE REQUERIMIENTOS

La principal dificultad encontrada para procesar los requerimientos es que estos no estén concentrados en un mismo lugar, ni escritos con un mismo estilo. Se tomó una muestra de requerimientos funcionales de los distintos componentes que se validan de los microprocesadores y se encontró un patrón en la mayoría de ellos que refleja que estos requerimientos tienen esta forma: bajo ciertas condiciones específicas, cuando ocurre alguno o varios eventos, entonces se deben disparar uno o más flujos de acción; resultando así, que un requerimiento funcional tiene una estructura como la Ecuación 2.

$$\begin{aligned} &REQUERIMIENTO\ FUNCIONAL \\ &\qquad \rightarrow Si\ CONDICIONES\ cuando\ EVENTOS, entonces\ RESULTADOS \end{aligned}$$

Ecuación 2. Estructura de un Requerimiento Funcional

Elaboración propia

Una vez encontrado este patrón en los requerimientos funcionales, la cuestión era cómo procesar computacionalmente los requerimientos escritos en inglés para que se reconozcan los elementos de la estructura de manera automática.

Para esta tarea se decidió utilizar la parte del analizador configurable del sistema TCG_UML de Espejel como se describe en la sección 3.2. Este analizador necesita de una gramática libre de contexto de la que se derive la sintaxis de los requerimientos que se quieren analizar.

Para este propósito se creó una gramática libre de contexto y en la Figura 7 se muestra un fragmento de esta gramática donde se pueden ver los símbolos no terminales más importantes que la conforman: FUNCIONAL, REQUERIMIENTO, CONTEXTO, EVENTOS y RESULTADOS.

```
<REQUERIMIENTOS>
    <derivation>
        <noterminal>FUNCIONAL</noterminal>
    </derivation>
</REQUERIMIENTOS>
<FUNCIONAL>
    <derivation>
        <noterminal>REQUERIMIENTO</noterminal>
        <noterminal>FUNCIONAL</noterminal>
    </derivation>
    <derivation>
        <noterminal>REQUERIMIENTO</noterminal>
    </derivation>
</FUNCIONAL>
<REQUERIMIENTO>
    <derivation>
        <noterminal>CONTEXTO</noterminal>
        <terminal>when</terminal>
        <noterminal>EVENTOS</noterminal>
        <terminal>then</terminal>
        <noterminal>RESULTADOS</noterminal>
    </derivation>
</REQUERIMIENTO>
<CONTEXTO>
    <derivation>
        <terminal>always</terminal>
    </derivation>
    <derivation>
        <noterminal>CONDICIONAL</noterminal>
    </derivation>
</CONTEXTO>
```

Figura 7. Extracto de gramática para requerimientos funcionales

Elaboración propia

Lo que se describe en el fragmento de la gramática en la Figura 7 es lo siguiente: esta comienza con el símbolo no terminal de REQUERIMIENTOS que actualmente solo deriva al símbolo de FUNCIONAL, la gramática solo acepta requerimientos funcionales. Después, el símbolo FUNCIONAL tiene dos opciones para derivar: en los símbolos REQUERIMIENTO y FUNCIONAL o solamente en el símbolo FUNCIONAL, esto se traduce en que FUNCIONAL puede contener uno o más requerimientos, el archivo de texto que se introduce para ser analizado puede contener varios requerimientos. Finalmente cada requerimiento deriva en los símbolos no terminales de CONTEXTO, EVENTOS y RESULTADOS, además de las palabras "when" y "then" en este orden. En el ANEXO A se encuentra la gramática completa con todos los símbolos terminales y no terminales representada en un formato más conciso.

3.2 INTEGRACIÓN CON ANALIZADOR CONFIGURABLE

Como se menciona anteriormente, se decidió utilizar el módulo analizador configurable de la plataforma TCG_UML que se describe en la sección 2.4.3 e integrarle un módulo generador de casos de prueba que se desarrolló como parte de este trabajo. Este analizador recibe como entrada la gramática diseñada, y descrita en la sección 3.1. Con esta gramática, la plataforma es ahora capaz de aceptar requerimientos funcionales escritos con el formato definido.

En la Figura 8 se muestra el ejemplo de un requerimiento escrito con el formato, se resaltan los tres elementos principales que conforman al requerimiento: Contexto, Eventos y Resultados. Al ser analizado este requerimiento por la plataforma y usando la gramática descrita en la sección 3.1, se genera el árbol sintáctico que se muestra en el ANEXO B.

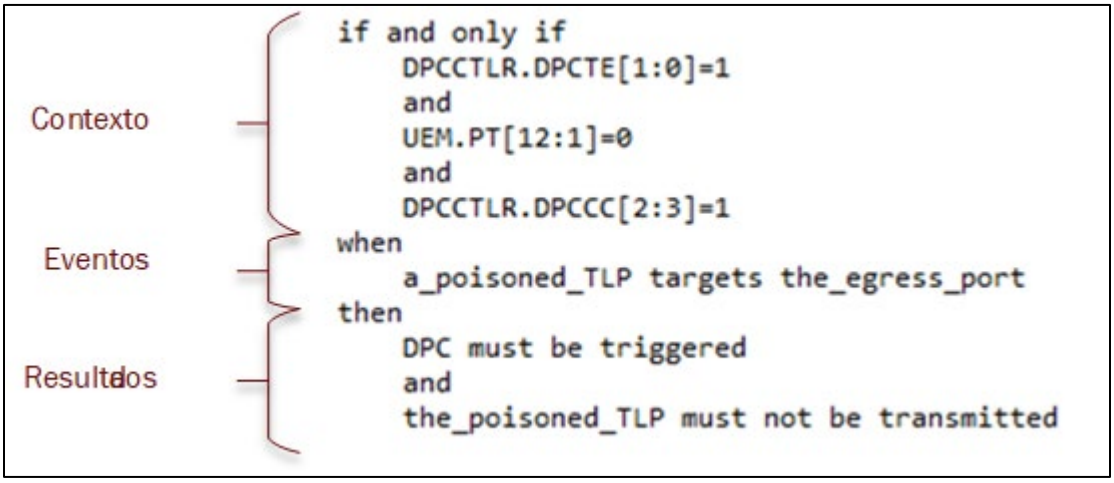


Figura 8. Ejemplo de requerimiento funcional

Elaboración propia

El árbol sintáctico contiene en los nodos los elementos que se definieron en la gramática, del nodo FUNCIONAL se derivan los nodos: CONTEXTO, EVENTOS y RESULTADOS, estos son los elementos que se definieron como los principales para formar un requerimiento funcional; a su vez, de estos nodos principales se derivan los elementos que los conforman; así por ejemplo, del nodo CONTEXTO se derivan los nodos CONDICIONAL y CONDICIONES, y de aquí se derivan todas las CONDICIONES que se deben cumplir en un requerimiento funcional. La Figura 9 es un fragmento del árbol sintáctico generado, donde se puede apreciar mejor cómo se derivan las condiciones que son parte del requerimiento.

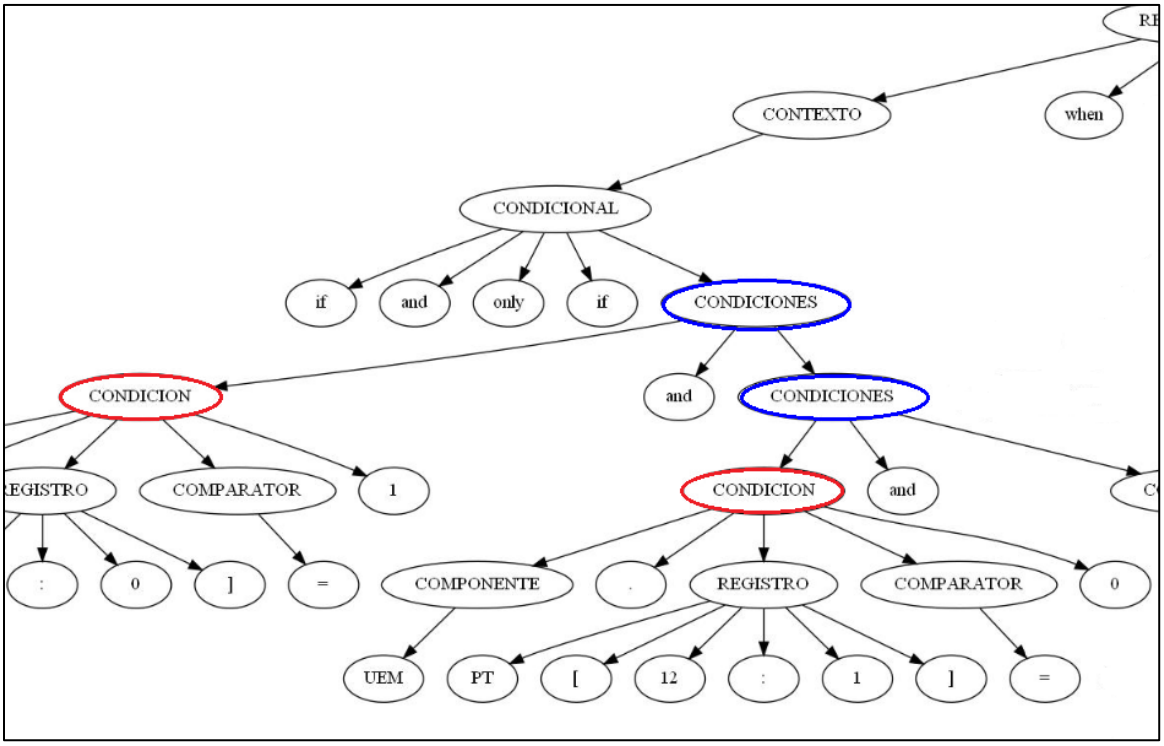


Figura 9. Condiciones en el árbol sintáctico

Elaboración propia

3.3 IMPLEMENTACIÓN DE SOFTWARE GENERADOR DE CASOS DE PRUEBA

Originalmente la plataforma TCG_UML (Espejel Salazar, 2019), además del analizador configurable que genera el árbol sintáctico, también tiene implementado un generador de casos de prueba muy específico para generar casos de prueba a partir de un diagrama de actividad en la sintaxis de PlantUML. Para efectos de este proyecto se implementó otro generador de casos de prueba, específico para los requerimientos escritos de una manera estandarizada en lenguaje inglés, como se describió en la sección 3.1. La Figura 10 muestra el diagrama de clases del módulo de software que se implementó en este proyecto para generar automáticamente los casos de prueba. En el ámbito de la programación orientada a objetos, una clase representa un tipo particular de objetos y tiene asociado un código (definición de la clase), que determina los atributos, así como los métodos que pueden ejecutar los objetos de la clase (Pavón Mestras, 2007). La programación orientada a objetos permite a los programadores

pensar de las entidades del software como si fueran entidades de la vida real (Banda, 2020).

La clase principal de este módulo es la clase "REQUIREMENTS_PROCESSOR", esta clase recibe el árbol sintáctico que previamente genera el analizador sintáctico, donde se encuentra la información de los requerimientos que están siendo analizados. Esta clase de "REQUIREMENTS_PROCESSOR" procesa cada subárbol que representa un requerimiento, por lo tanto necesita funciones que procesen cada elemento del requerimiento: condiciones, eventos y resultados. Para cada uno de estos elementos también se ha creado una clase que contiene los atributos del elemento que representa: "EVENT", "RESULT" y "CONDITION".

La otra clase importante es la clase "TEST_CASES", esta clase es instanciada por cada requerimiento que es encontrado en el árbol sintáctico y su objetivo es generar todos los casos de prueba para ese requerimiento, su estrategia es crear las distintas combinaciones de condiciones apoyados en el concepto sobre casos de prueba de Fulton y Vandermolen descrito en la sección 2.3.

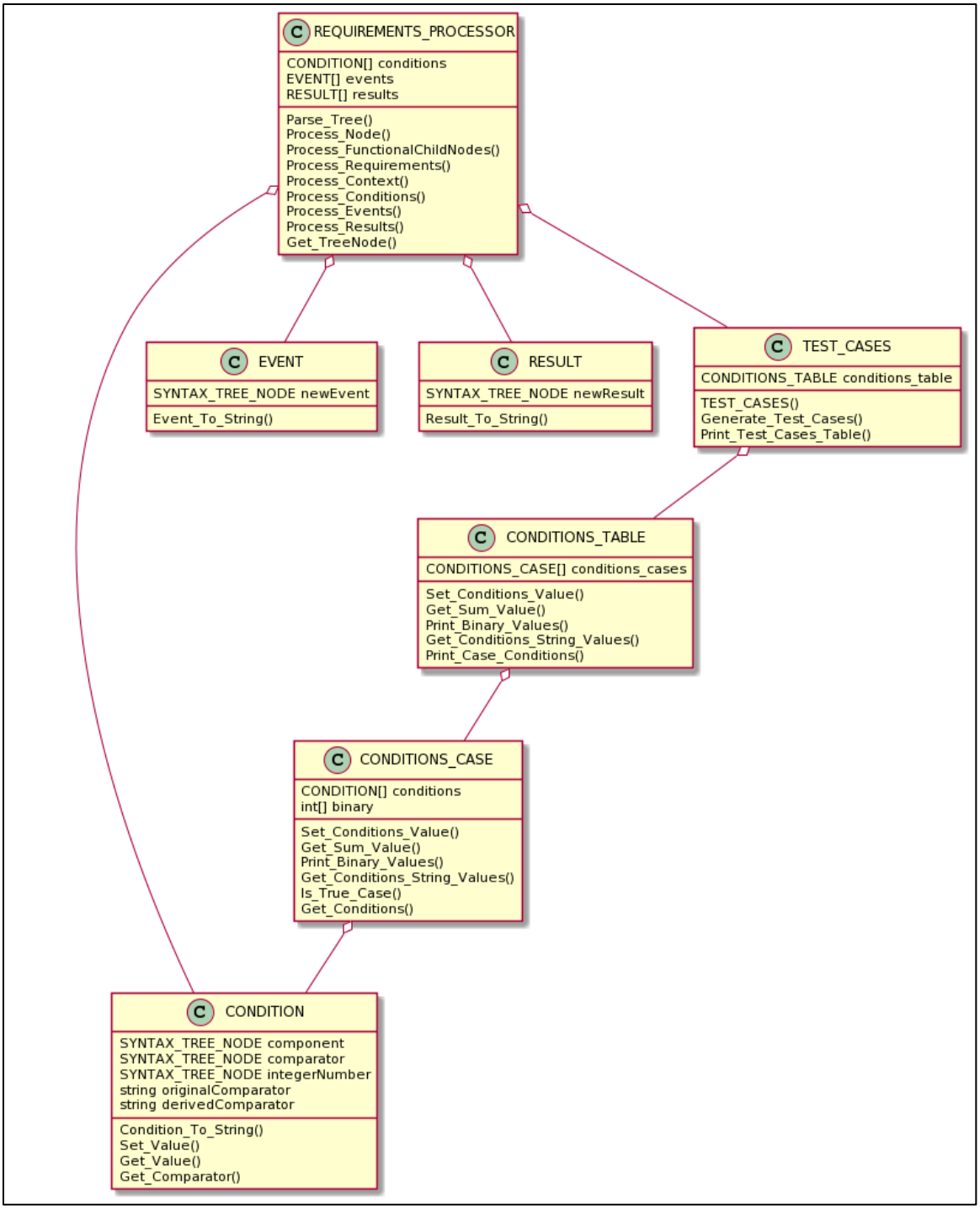


Figura 10. Diagrama de clases de modulo generador de casos de prueba

Elaboración propia

3.4 SEMÁNTICA IMPLEMENTADA

Para describir la manera en que se generan los casos prueba a partir de los requerimientos se seguirá usando como ejemplo el requerimiento mostrado en la Figura 8. En esa figura se muestra la estructura de un requerimiento funcional, formado por: Contexto, Eventos y Resultados, ahora en la Figura 11 se muestra la estructura que deben tener los casos de prueba generados por el generador de casos de prueba desarrollado en este trabajo.

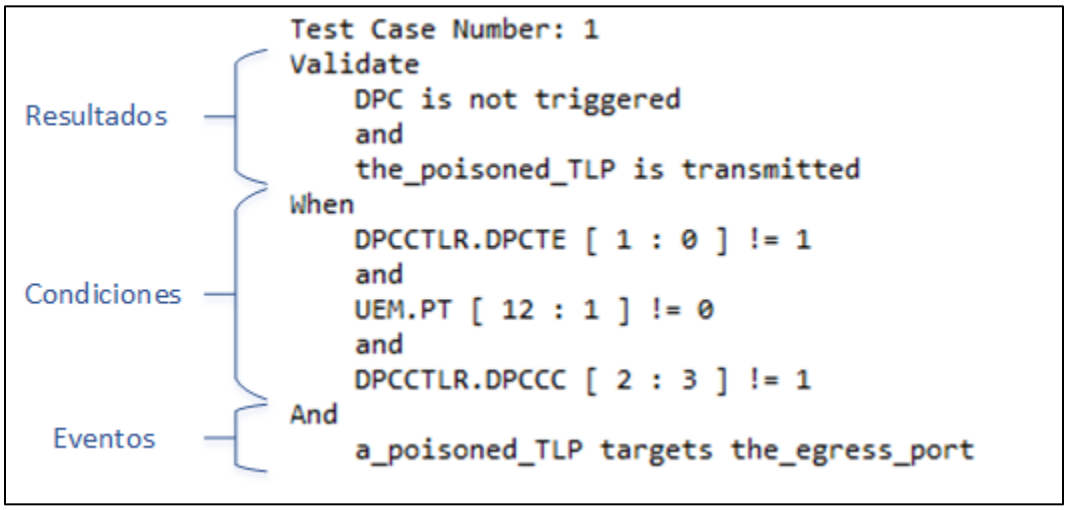


Figura 11. Ejemplo de caso de prueba

Elaboración propia

Enseguida se describen los tres elementos principales que tienen los casos de prueba generados así como una descripción de donde provienen y cómo son generados.

3.4.1 Resultados

El caso de prueba comienza con los resultados, los cuales tienen algunos cambios gramaticales con respecto a cómo se encuentran escritos en el requerimiento, en el ejemplo de este requerimiento vemos un resultado escrito de la siguiente manera:

"*DPC must be triggered*", como se ve, en el requerimiento se utiliza la palabra "*must*".
Sin embargo, en el caso de prueba, el resultado se escribe así:

"*Validate DPC is triggered*", aquí ya no se usa la palabra "must", se ha cambiado por el verbo escrito en presente. De esta manera se cambia la manera en cómo se expresan los resultados.

3.4.2 Condiciones

De las condiciones se originan los distintos casos de prueba, es decir, las condiciones pueden tomar valor falso o verdadero, y de acuerdo con el requerimiento, solo cuando se cumple que las condiciones toman unos valores específicos es cuando se generan ciertos resultados, sin embargo, se debe validar también lo que sucede cuando las condiciones toman otros valores diferentes a los que se especifican en el requerimiento.

Siguiendo con el ejemplo del requerimiento de la Figura 8, se tienen estas condiciones:

DPCCTLR.DPCTE [1: 0] = 1
UEM.PT [12: 1] = 0
DPCCTLR.DPCCC [2: 3] = 1

Cada condición puede tomar el valor de falso o verdadero, o dicho de otra manera, su signo puede cambiar de "=" a "! =" (de igual a diferente) y viceversa. De esta manera estas tres condiciones pueden generar un total de 8 combinaciones diferentes, combinando los valores que puede tomar cada condición. En la Tabla 4 se muestran estas ocho combinaciones. El valor de 0 indica que es falsa o no se cumple la condición, mientras que el valor de 1 significa que sí se cumple, por ejemplo, para la condición:

DPCCTLR.DPCTE [1: 0] = 1

Cuando es verdadera o tiene valor de 1, quiere decir que *DPCCTLR.DPCTE [1: 0]* es igual a 1(*DPCCTLR.DPCTE [1 : 0] = 1*), como está expresada en la condición. Sin embargo, cuando es falsa o tiene valor de 0, quiere decir que *DPCCTLR.DPCTE [1: 0]* es diferente a 1 (*DPCCTLR.DPCTE [1: 0] ¡= 1*), que es lo contrario a lo que expresa la condición.

Tabla 4. Combinaciones de condiciones

No.	DPCCTLR.DPCTE [1: 0] = 1	UEM.PT [12: 1] = 0	DPCCTLR.DPCCC [2: 3] = 1
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

Elaboración propia

De esta manera, se genera un caso de prueba para cada combinación diferente, resultando ocho casos de prueba; sin embargo, en la única combinación, en la cual se esperan los resultados como se mencionan en el requerimiento, es la combinación en que coinciden los valores de las condiciones con los valores del requerimiento, en este caso sería la última combinación, la combinación No. 8, la cual indica que las tres condiciones son verdaderas, en las tres el valor es igual al indicado.

3.4.3 Eventos

Los eventos son un suceso que ocurre o un lapso que se cumple, cuando algún evento se presenta ante las condiciones establecidas, entonces se esperan ciertos resultados.

Si el requerimiento tiene más de un evento, se espera que al generarse cualquiera de los eventos se disparen los resultados, por lo tanto, los casos de prueba

generados se repetirán tantas veces como eventos contenga un requerimiento. Para el ejemplo del requerimiento que se ha estado revisando, solamente se tiene un evento, por lo tanto, los 8 casos de prueba son los únicos que serán generados para este requerimiento.

3.5 USO DE MC/DC

El generador de casos de prueba fue implementado con la opción de aplicar la metodología MC/DC, es decir, solamente generar los casos de prueba necesarios para demostrar que cada condición afecta de manera independiente la salida de la decisión, tal como se mostró que funciona MC/DC en la sección 2.3.2.

De esta manera, para el requerimiento que se ha tomado como ejemplo de la Figura 8, al aplicar MC/DC, en lugar de generar los 8 casos de prueba para las ocho combinaciones, se generan solamente cuatro casos de prueba.

Estos casos de prueba son un subconjunto del total de las combinaciones que cumplen los requerimientos mínimos para la compuerta "and" según lo descrito en la sección 2.3.2.

En la Tabla 5 se muestra la tabla de verdad para la expresión que forman las condiciones del requerimiento que se ha estado revisando. Son tres condiciones y están unidas por un "and", solamente cuando todas son positivas se hace positivo el resultado, como se muestra en la combinación No. 8.

Para cumplir con los requerimientos de MC/DC en una compuerta "and", se necesitan las cuatro combinaciones que están sombreadas en la Tabla 5.

Tabla 5. Casos que cumplen con MC/DC

No.	DPCCTLR.DPCTE [1: 0] = 1	UEM.PT [12:1] = 0	DPCCTLR.DPCCC [2:3] = 1	Resultado
1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	**0**	**1**	**1**	**0**
5	1	0	0	0
6	**1**	**0**	**1**	**0**
7	**1**	**1**	**0**	**0**
8	**1**	**1**	**1**	**1**

Elaboración propia

4. RESULTADOS

4.1 CONCENTRAR REQUERIMIENTOS

Con la gramática diseñada y descrita en la sección 3.1 se hace la propuesta para escribir los requerimientos funcionales usando el formato de requerimiento descrito en la Figura 8, y al hacer la propuesta de documentarlos todos de la misma manera, también se está haciendo la propuesta de concentrarlos en un mismo lugar.

De esta manera se tendrá un mejor manejo de los requerimientos, asegurando su completitud y también estarán listos para procesarse por medio de herramientas como la que aquí se presenta.

La gramática fue diseñada de acuerdo con las recomendaciones encontradas en la revisión de literatura sobre creación de casos de prueba para hardware y que se mencionan en la sección 2.1.2.

El usar esta plantilla o gramática para escribir los requerimientos traerá los beneficios descritos también en la sección 2.1.2, como por ejemplo, ayudar a los nuevos miembros del equipo a contribuir más rápido en el proyecto y ayudar al escritor de los requerimientos a pensar en todos los aspectos de la funcionalidad.

4.2 CASOS DE PRUEBA GENERADOS

Una vez implementado el módulo generador de casos de prueba y usando la gramática diseñada, el sistema es capaz de aceptar archivos de texto que contienen requerimientos funcionales de hardware como se muestra en la Figura 12.

```
if and only if
    DPCCTLR.DPCTE[1:0]!=1
when
    a_unmasked_uncorrectable_error is detected
    or
    upstream_ERR_FATAL is received
    or
    ERR_NONFATAL_message is received
    or
    link changes from L1 to L0
then
    the_system must trigger DPC

if and only if
   DCTL2.CTRL[5:0] = 1
when
    upstream_posted_transactions_with_randomized_RID is generated
then
    the_completions must preserve the_RID_value_of_the_originating_request

if and only if
    DPCCTLR.DPCTE[1:0]=1
    and
    DPCCAPR.RPEFDPC[5:6]=1
    and
    DPCECH.PTLPEBS[6:7]=1
    and
    DPCCTLR.PTLPEBE[5:4]=1
    and
    UEM.PT[12:1]=0
    and
    DPCCTLR.DPCCC[2:3]=1
when
    a_poisoned_TLP targets the_egress_port
then
    DPC must be triggered
    and
    the_poisoned_TLP must be discarded
    and
    the_poisoned_TLP must not be transmitted
```

Figura 12. Requerimientos funcionales de hardware

Resultado del módulo de software desarrollado

A partir de estos requerimientos funcionales, el sistema genera los casos de prueba para cada requerimiento en dos formatos diferentes: el primero es un archivo PDF que muestra los casos de prueba en inglés con una estructura definida como se muestra en la Figura 13.

```
Test Case Number: 1
Validate
DPC is not triggered
When
DPCCTLR.DPCTE [ 1 : 0 ] != 1
DPCCAPR.RPEFDPC [ 5 : 6 ] != 1
DPCECH.PTLPEBS [ 6 : 7 ] != 1
And
a_poisoned_TLP targets the_egress_port


Test Case Number: 2
Validate
DPC is not triggered
When
DPCCTLR.DPCTE [ 1 : 0 ] != 1
DPCCAPR.RPEFDPC [ 5 : 6 ] != 1
DPCECH.PTLPEBS [ 6 : 7 ] = 1
And
a_poisoned_TLP targets the_egress_port


Test Case Number: 3
Validate
DPC is not triggered
When
DPCCTLR.DPCTE [ 1 : 0 ] != 1
DPCCAPR.RPEFDPC [ 5 : 6 ] = 1
DPCECH.PTLPEBS [ 6 : 7 ] != 1
And
a_poisoned_TLP targets the_egress_port
```

Figura 13 .Casos de prueba generados

Resultado del módulo de software desarrollado

El otro formato en que son presentados los casos de prueba es en un archivo de valores separados por comas (CSV por sus siglas en inglés) que puede ser abierto en Excel y mostrar los casos de prueba en una tabla, esto permite una tener una mejor visión de todos los casos de prueba. En la Tabla 6 se muestra un ejemplo de los casos de prueba que se generarían para un requerimiento.

Tabla 6. Casos de prueba generados en formato CSV

Inputs	DPCCTRL.DPCTE[1:0]! =1	Evento	Resultado
Test Case 1	FALSE	A_unmasked_uncorrectable_error is detected	The_system does not trigger DPC
Test Case 2	TRUE	A_unmasked_uncorrectable_error is detected	The_system trigger DPC
Test Case 3	FALSE	Upstream_ERR_FATAL is received	The_system does not trigger DPC
Test Case 4	TRUE	Upstream_ERR_FATAL is received	The_system trigger DPC
Test Case 5	FALSE	ERR_NONFATAL_message is received	The_system does not trigger DPC
Test Case 6	TRUE	ERR_NONFATAL_message is received	The_system trigger DPC
Test Case 7	FALSE	Link changes from L1 to L0	The_system does not trigger DPC
Test Case 8	TRUE	Link changes from L1 to L0	The_system trigger DPC

Elaboración propia

Cada fila corresponde a un caso de prueba diferente y en las columnas primero se encuentran las condiciones, después los eventos y finalmente el resultado.

En el ejemplo de la Tabla 6 solamente se tiene una condición: DPCCTRL.DPCTE[1:0]! =1 la cual solo puede ser falsa o verdadera; se tienen tres eventos diferentes y se tiene un resultado que se puede o no cumplir. Lo anterior da como resultado ocho casos de prueba: 2 valores de una condición x 4 eventos. En general, el número de casos de prueba será: $2^n.\ x$, donde n es el número de condiciones y x es la cantidad de eventos que se pueden generar según el requerimiento.

4.3 COMPARATIVA CON OTRAS SOLUCIONES

La Tabla 7 es una comparativa de los generadores de casos de prueba que se revisaron en este trabajo: TCG_UML y ASSERT, y el generador de casos de prueba que se desarrolló aquí usando el analizador configurable de TCG_UML.

Tabla 7. Comparativa de generadores de casos de pruebas

Característica	TCG_UML	ASSERT	Generador de casos de prueba usando el analizador configurable
Usado en validación	Software	Software	Hardware
Configurable	Si	No	Si
Fuente de datos	Modelos UML	Requerimientos en inglés	Requerimientos en inglés
Se define sintaxis	No	No	Si
Se define semántica	Si	No	Si

Elaboración propia

El nuevo generador de casos de prueba desarrollado en este trabajo toma ideas de los dos trabajos: TCG_UML y ASSERT.

Se está reusando el analizador configurable de TCG_UML, esto le da la capacidad de ser configurable a nuestro generador de casos de prueba, lo que le permite ser una herramienta adaptable a los proyectos, y si hubiera necesidad de cambiar la sintaxis de los requerimientos solamente se necesitaría modificar la gramática diseñada.

Este generador de casos de prueba acepta los requerimientos escritos en inglés, como lo hace ASSERT, solo con la condición de apegarse al formato definido por la gramática libre de contexto, lo que permite que la implementación de esta metodología sea más fácil de adoptar dentro de la empresa que si se pidiera cambiar por completo la forma en que se documentan los requerimientos.

4.4 TIEMPOS DE CICLO

El generador de casos de prueba se utilizó como prueba a pequeña escala en el área de validación de interrupciones de hardware.

Para esto se escribieron 20 requerimientos en el formato propuesto en este trabajo) no se pueden mostrar aquí por ser propiedad intelectual de Intel), estos requerimientos se utilizaron como entrada en el sistema generador de casos de prueba al igual que la gramática diseñada descrita en la sección 3.1; y de esta manera se generaron automáticamente los casos de prueba en cuestión de 2 o 3 minutos después de algunos clics. En la Tabla 8 se muestra una proyección de los resultados obtenidos al usar el generador de casos de prueba en el proceso de validación en comparación con la generación de casos de prueba de manera manual ya que no se ha usado aún en producción.

Tabla 8. Comparativa de resultados

Actividad	Generación manual de casos de prueba (horas)	Generación automática de casos de prueba (horas)
Escritura de requerimientos	80 (Sin formato)	120 (Con formato)
Generación de casos de prueba	480	80
Desarrollo de contenido para pruebas	480	480
Preparación para PO y VV	480	480
Ejecución y depuración	1920	1920
QS	480	480
PRQ	480	480
Total de horas	4400	4040

Elaboración propia

Como se puede ver, al utilizar el sistema de generación automática de casos de prueba, se necesita escribir los requerimientos en el formato establecido y esto genera un incremento en el tiempo necesario para la actividad de escritura de

requerimientos (40 horas), sin embargo después en la actividad de generación de casos de prueba, ese tiempo es recompensado con mucho al disminuir de 480 a solamente 80 horas. Con el uso del generador de casos de pruebas, el total de horas disminuye 360 horas, esto es un 8% del total de horas invertidas en la validación de un proyecto.

Esta disminución de tiempo representa una reducción de costos, de acuerdo con la Figura 1 donde se presenta el personal asignado por trimestre, se encuentra que la actividad de desarrollo de casos de prueba se lleva a cabo durante un trimestre en el cual la capacidad de personal está al 50% por lo cual esta reducción de los gastos representa un 4%.

CONCLUSIONES

Se ha logrado automatizar la generación de los casos de prueba funcionales a partir del análisis de los requerimientos, utilizando una plataforma que analiza y luego genera casos de prueba a partir de ese análisis. Con esto se ha logrado reducir el tiempo destinado a la definición de casos de prueba.

Esta plataforma está conformada por dos módulos, un analizador de archivos y un generador de casos de prueba. El módulo analizador de archivos se está reusando del sistema TCG_UML (Espejel Salazar, 2019), este módulo requiere de una gramática libre de contexto para habilitar el análisis de los requerimientos. La gramática fue diseñada para habilitar el análisis sintáctico y semántico de los requerimientos y con esto se cumple el objetivo de estandarizar el formato de escritura de los requerimientos.

El módulo generador de casos de prueba se implementó para que trabajara en conjunto con el módulo analizador, este recibe el árbol sintáctico que el analizador genera y a partir de ahí construye los casos de prueba necesarios para garantizar que los requerimientos son correctamente validados. Para la generación de casos de prueba se implementó la metodología MC/DC, que como se describe en la sección 3.5, se encontró que es una estrategia efectiva porque únicamente se generan los casos de prueba necesarios para validar el correcto resultado en las salidas a partir de los valores de entrada introducidos en el sistema.

En general, se logró tener la herramienta que es capaz de analizar los requerimientos funcionales de hardware en Intel y aplicando la mejor metodología encontrada se generan de manera automática los casos de prueba.

RECOMENDACIONES

Este trabajo se desarrolló para analizar los requerimientos funcionales de hardware, ya que este es el tipo de requerimientos más utilizado y los mejores requerimientos para validar (Fulton & Vandermolen, 2017), pero como se vio en la sección 2.1.1, también existen otros tipos de requerimientos de dispositivos electrónicos, la recomendación es ampliar la gramática para que acepte esos otros tipos de requerimientos. De la misma manera, se recomienda mejorar la gramática para los requerimientos funcionales, agregar más vocabulario y trabajar en la mejora de la escritura de los sustantivos, actualmente los sustantivos solo son reconocidos si están formados por una sola cadena de caracteres, ya sea una palabra o varias palabras separadas por guion bajo.

Por otro lado, se puede continuar trabajando para generar además de los casos de prueba, los scripts de las pruebas con los pasos que se deben seguir para ejecutar la prueba.

Tal como se encuentra ahora la gramática funciona para aceptar los requerimientos de hardware funcionales de manera general y no exclusivamente para Intel, de acuerdo con las fuentes encontradas, la estructura que se definió para los requerimientos es la estructura estándar de la industria. Sin embargo a pesar de que la estructura base se mantenga a través de la industria, las palabras modales pueden cambiar incluso de proyecto a proyecto y la gramática puede ser adaptada.

APORTACIÓN DE LA TESIS

Este trabajo une ideas de varios trabajos anteriores dirigidos a generación automática de casos de prueba para validación de software, en especial dos de ellos: ASSERT y TCG_UML, de ASSERT se toma la propuesta de establecer un estándar o formato para escribir los requerimientos, sin embargo ASSERT no permite que la herramienta sea configurable y se pueda cambiar esa estructura fácilmente ya que usa un lenguaje definido: SADL. De TCG_UML se toma la idea de especificar una gramática y se reúsa su analizador configurable, sin embargo TCG_UML no creó una sintaxis sino que utilizó la sintaxis de PlantUML, en este trabajo se creó una sintaxis como resultado del análisis que se hizo de los requerimientos. Siendo así que el aporte es la creación de la gramática libre de contexto que define el lenguaje utilizado para los requerimientos de hardware funcionales utilizados en Intel; así como también el módulo de software construido para la generación de casos de prueba funcionales para Intel.

APORTACIÓN SOCIAL DE LA TESIS

Este trabajo genera una gramática para expresar requerimientos de hardware funcionales con una estructura estándar en la industria de hardware. Para hardware, existe muy poca documentación sobre requerimientos y buenas prácticas, por lo que este trabajo representa algo de avance para esa falta de conocimiento público en investigaciones anteriores, casi todo es sobre software.

REFERENCIAS

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. (2007). *Compilers: Principles, Techniques, and Tools.* Boston: Pearson, Addison Wesley.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., . . . Mc Minn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software, 86*(8), 1978-2001.

Anderson, S. (2011). *Structural testing.* Obtenido de The University of Edinburgh. School of Informatics: https://www.inf.ed.ac.uk/teaching/courses/st/2011-12/Resource-folder/06_structural.pdf

Banda, F. (20 de marzo de 2020). *The four pillars of object oriented programming.* Obtenido de Key Lime Interactive: https://info.keylimeinteractive.com/the-four-pillars-of-object-oriented-programming

Chang, K.-h., Markov, I., & Bertacco, V. (2007). Automating post-silicon debugging and repair. *2007 IEEE/ACM International Conference on Computer-Aided Design*, 47-54. doi:10.1109/ICCAD.2007.4397249

Chen, T. Y.-C. (2010). Adaptive random testing: the ART of test case diversity. *Journal of Systems and Software, 83*(1), 60-66. doi:https://doi.org/10.1016/j.jss.2009.02.022

Chilenski, J. J. (1994). Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal, Vol. 7, No. 5*, 193-200.

Chilenski, J. J. (2001). *An investigation of three forms of the modified condition decision coverage (MCDC) criterion.* Washington, D.C.: Office of Aviation Research.

Chilenski, J. J., Hayhurst J, K., Veerhusen, D. S., & Rierson, L. K. (2001). *A Practical Tutorial on Modified Condition/Decision Coverage.* Springfield: NASA Center for AeroSpace Information (CASI).

Cunning, S., & Rozenblit, J. (1999). Automatic Test Case Generation from Requirements Specifications for Real-time Embedded Systems. *IEEE Xplore*, 784-789.

Ehmer Khan, M. (2011). Different Approaches to White Box Testing Technique for Finding Errors. *International Journal of Software Engineering and Its Applications*.

Espejel Salazar, G. (Noviembre de 2019). Desarrollo de infraestructura para facilitar el análisis y la verificación de diagramas de actividad identificando los caminos funcionales. *Tesis para obtener el grado de Maestro*. Querétaro, Querétaro, México: CIATEQ.

Faria, H., & Salazar, A. (2015). Prototyping Speeds Hardware Validation, Interface IP Certification. *Electronic design*.

Fisher, R. A. (1971). *The Design of Experiments, 8th Edition.* New York: Hafner Publishing Company.

Fulton, R., & Vandermolen, R. (2017). *Airborne Electronic Hardware Design Assurance*. Florida: CRC Press.

Gallier, J. (2020). *Introduction to the Theory of Computation Languages, Automata, Grammars Slides for CIS262.* Filadelfia: University of Pennsylvania.

Garcia Bedoy Torres, J. (25 de April de 2021). How are distributed the project costs at Intel. (A. Victoria Alcaraz, Entrevistador)

Gómez, D., & Pardo, L. M. (2015). *Teoría de Autómatas y Lenguajes Formales.* Santander: Universidad de Cantabria.

Harman, M. a. (2001). Search based software engineering. *Information and Software Technology*, 833-839.

Jantsch, A., Notbauer, J., & Albretch, T. (2000). Functional Validation of Mixed Hardware/Software Systems based on Specification, Partitioning and Simulation of Test Cases. *Design automation for embedded systems* , 31.

Jiag, T., Li, M., Ravikumar, B., & Regan, K. W. (2018). Formal Grammars and Languages. Riverside, California, Estados Unidos.

King, J. C. (1975). *A new approach to program testing.* New York: Computer Science Department .

Kirsch, W. (23 de Abril de 2020). *'You have been moving heaven and Earth': Keller, Koduri visit lab, offer thanks to validation teams*. Obtenido de Circuit News: https://circuit.intel.com/content/news/circuitnews/corporate/2020/-you-have-been-movingheavenandearthkellerkodurivisitlabofferthan.html

Kolmar, C. (8 de Marzo de 2021). *10 Largest semiconductor companies in the world*. Obtenido de Zippia: https://www.zippia.com/advice/largest-semiconductor-companies-world/

Málaga Jurado, E. (2008). *Teoría de autómatas y lenguajes formales* (Vol. 55). Cáceres: Universidad de Extremadura.

Martin, S., Aurum, A., Jeffery, R., & Paech, B. (2002). Requirements Engineering Process Models in Practice. *Australian Workshop on Requirements Engineering* (págs. 141-155). Melbourne: Deakin University.

Merrill, C. C. (Octubre de 2017). Platform Definition: Concept to LZ to POR Flow. Estados Unidos: Intel.

Mishra, P., Morad, R., Ziv, A., & Ray, S. (2017). Post-silicon Validation in the SoC Era: A Tutorial Introduction. *IEEE Design & Test*, 68-92.

Mitra, S., Seshia, S. A., & Nicolici, N. (2010). Post-Silicon Validation Opportunities, Challenges and Recent Advances. *Design Automation Conference* (págs. 12-17). Nueva York: Association for Computing Machinery.

Mozhikunnath, R. (18 de Septiembre de 2016). *anysilicon.com*. (CMC Microsystems) Recuperado el 19 de Septiembre de 2020, de anysilicon: http://anysilicon.com/verification-validation-testing-asicsoc-designs-differences/

Myers, G. J. (2004). *The Art of Software Testing, Second Edition.* Hoboken: John Wiley & Sons, Inc.

Nidhra, S., & Dondeti, J. (2012). Black Box and White Box Testing Techniques -A literature review. *International Journal of Embedded Systems and Applications*, 29-50.

Pahade, P., & Dawale, M. (2019). Introduction to Compiler and its Phases. *International Research Journal of Engineering and Technology*, 1318-1322.

Pavón Mestras, J. (2007). *Facultad de Informática - Universidad Complutense de Madrid.* Obtenido de Programación Orientada a Objetos : https://www.fdi.ucm.es/profesor/jpavon/poo/1.1.objetos%20y%20clases.pdf

Pezze, M., & Young, M. (2007). *Structural Testing.* Obtenido de The University of Edinburgh: http://www.inf.ed.ac.uk/teaching/courses/st/2017-18/Ch12.pdf

Pezze, M., & Young, M. (2008). *Software Testing and Analysis: Process, Principles, and Techniques.* Nueva York: Daniel Sayre.

Pill, I., Semprini, S., Cavada, R., Roveri, M., Bloem, R., & Cimatti, A. (2006). Formal Analysis of Hardware Requirements. *43rd ACM/IEEE Design Automation Conference* (págs. 821-826). San Francisco: Association for Computing Machinery.

Poston, R. M. (1994). Automated testing from object models. *Communications of the ACM*, 48-58.

Purdum, J. J. (2008). *Beginning C# 3.0.* Indianapolis: Wrox Press.

Ray, S., & Hunt, W. (2009). Connecting Pre-silicon and Post-silicon Verification. *Formal Methods in Computer-Aided Design*, (págs. 160-163). Austin.

Sahoo, R., Ojha, D., Mohapatra, D., & Manas. (2016). Automated Test Case Generation and Optimization : A Comparative Review. *International Journal of Computer Science & Information Technology , 8*, 19-32.

Sarabi, M. (2012). *Evaluation of Structural Testing Effectiveness in Industrial Model-driven Software Development.* Vasteras: Malardalen University Sweden.

Scott, W., Cook, S., & Kasser, J. (2004). Development and Application of a Context-free Grammar for Requirements. *The Systems Engineering Test and Evaluation (SETE) Conference* (págs. 333-340). Adelaide: Systems Engineering Society of Australia.

Singh, A. (2019). *Formal Language And Automata Theory 2nd Edition.* National Library of Australia.

Siu, K., Moitra, A., Durling, M., Crapo, A., Li, M., & Yu, H. (2017). Flight Critical Software and Systems Development using ASSERT. *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)* (págs. 1-10). St. Petersburg : IEEE.

Tasiran, S., & Keutzer, K. (2001). Coverage metrics for functional validation of hardware designs. *IEEE Design Test of Computers*, 36-45.

Thain, D. (2020). *Introduction to compilers and language design.*

TutorialsPoint. (2016). *Automata Theory Introduction.* Recuperado el 17 de Septiembre de 2020, de tutorialspoint.com: https://www.tutorialspoint.com/automata_theory/automata_theory_tutorial.pdf

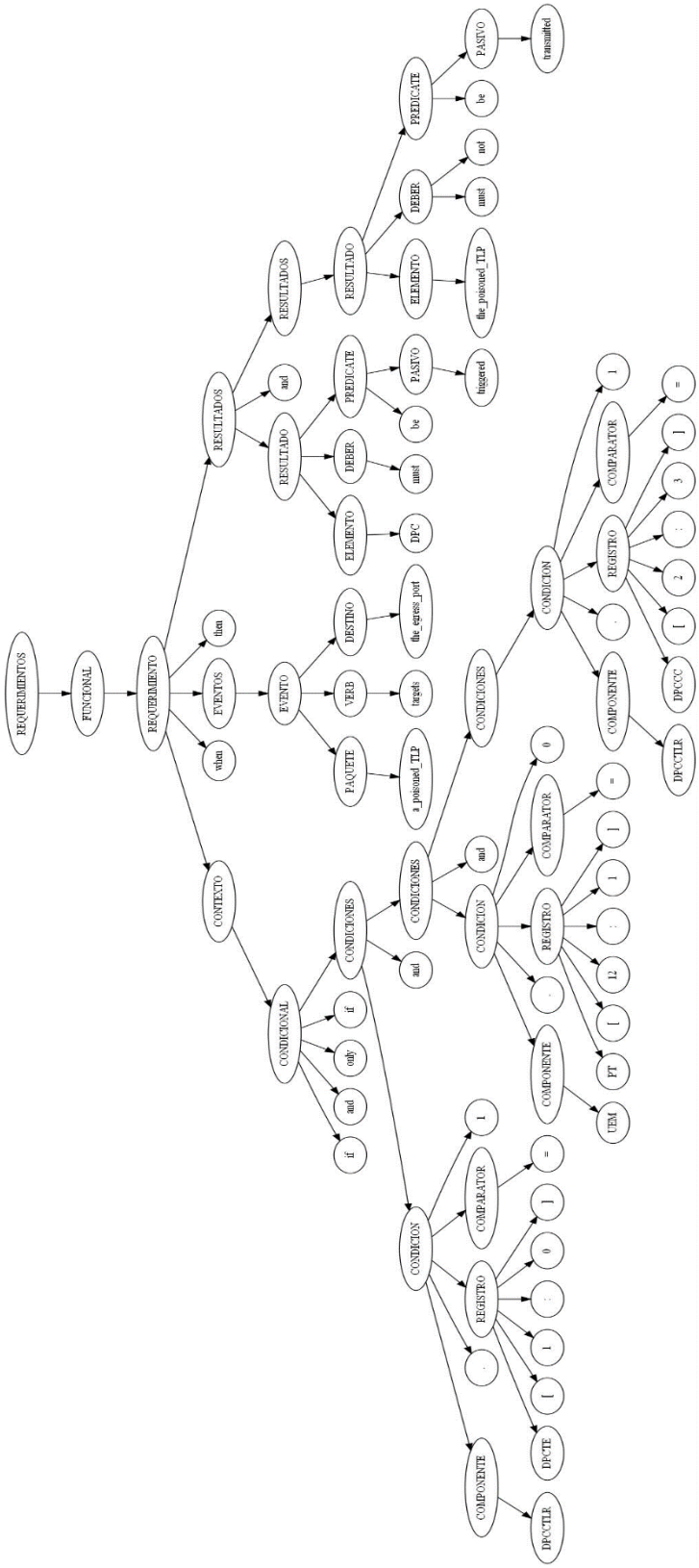
W., L., & Y., J. (2004). Generating Test Cases from UML Activity Diagrams based on Gray-Box Method. *11th Asia-Pacific Software Engineering Conference* (págs. 284-291). Busan: IEEE.

Wirfs-Brock, R. (20 de Julio de 2011). *Introducing Landing Zones*. Obtenido de The Responsible Designer: http://wirfs-brock.com/blog/2011/07/20/introducing-landing-zones/

ANEXOS

ANEXO A. Gramática libre de contexto para requerimientos funcionales de hardware

Símbolo No Terminal		**Derivación**
REQUERIMIENTOS		FUNCIONAL
FUNCIONAL		REQUERIMIENTO FUNCIONAL REQUERIMIENTO
REQUERIMIENTO		CONTEXTO when EVENTOS then RESULTADOS
CONTEXTO		always CONDICIONAL
CONDICIONAL		if and only if CONDICIONES
CONDICIONES		CONDICION and CONDICIONES CONDICION or CONDICIONES CONDICION
CONDICION		COMPONENTE REGISTRO COMPARADOR integer
COMPONENTE		identifier
REGISTRO		Identifier openbracket integer closebracket
COMPARADOR		equal notequal
EVENTOS		EVENTO and EVENTOS EVENTO or EVENTOS EVENTO
EVENTO		PAQUETE VERB DESTINO PAQUETE TO_BE PASIVO CHANGE
PAQUETE		Identifier
VERB		preserve targets trigger
TO_BE		is are
DESTINO		Identifier
PASIVO		triggered

		discarded transmitted detected received generated cleared re-enabled
CHANGE		ELEMENTO CHANGE_VERB from VALOR to VALOR
VALOR		integer identifier
CHANGE_VERB		changes change
ELEMENTO		identifier
RESULTADOS		RESULTADO and RESULTADOS RESULTADO or RESULTADOS RESULTADO
RESULTADO		ELEMENTO DEBER PREDICATE
DEBER		must not must
PREDICATE		be PASIVO VERB ADDITION
ADDITION		identifier

ANEXO B. Árbol sintáctico completo de un requerimiento funcional