Propuesta de Arquitectura para un Generador de Casos de Prueba a partir de un RLD

[a] *I.S.C. Raquel Lizue Martínez Ramos.*, [b] *M.C.E. Beatriz Alejandra Olivares Zepahua.*, [c] *M.C.C. Celia Romero Torres.*, [d] *M.S.C. Luis Ángel Reyes Hernández.*, [e] *M.R.T. Ignacio López Martínez.*

Instituto Tecnológico de Orizaba, División de Estudios de Posgrado e Investigación, Maestría en Sistemas Computacionales,
[a] raquelizue@outlook.es, [b] bolivares@ito-depi.edu.mx,
[c] cromerotorres@hotmail.com, [d] l_r_h01@hotmail.com,
[e] ilopez@ ito-depi.edu.mx, Orizaba, Veracruz, México.

Resumen

Las pruebas son necesarias en la creación de cualquier producto de software, ya que validan la funcionalidad de éste a través de la detección de errores y garantizan el hecho de que lo que se construyó es lo que se espera que haga el producto; las pruebas pueden ser unitarias (verifican el comportamiento de un componente), integrales (comprueban la funcionalidad entre varios componentes) o funcionales (validan el funcionamiento de subconjuntos del sistema), entre otras; el conjunto de pasos para reproducir la prueba se le conoce como caso de prueba. Por otra parte, el tiempo para probar es limitado, por lo que los esfuerzos actuales se centran en generar casos de prueba automáticamente, para optimizar el tiempo y presupuesto de prueba, garantizando la certeza de dichos casos; por lo tanto, es importante que el origen de los casos de prueba no introduzca errores o ambigüedades en el proceso. Existen diferentes soluciones a esta cuestión, como las definiciones formales y semiformales. En este trabajo se plantea la creación de un generador automático de casos de prueba funcionales que utilice como entrada casos de uso descritos en un lenguaje de definición de requerimientos (RDL) específico, para obtener un grafo dirigido y posteriormente, aplicar métodos que permitan obtener los casos de prueba tomando en consideración el porcentaje de cobertura, el tiempo disponible para probar y/o el tipo de escenario.

Palabras clave—Casos de Prueba, Casos de Uso, Cobertura, Lenguaje de Definición de Requerimientos.

Abstract

The tests are necessary in the creation of any software product, since they validate the functionality of this one through the detection of errors, in addition, they guarantee the fact that what was constructed is what the product is expected to do, such tests can be unitary (they verify the behavior of a component), integral (they check the functionality between several components) or functional (they validate the operation of subsets of the system), among others; the set of steps to reproduce the test is known as test case. On the other hand, the time to test is limited, so current efforts are focused on generating test cases automatically, to optimize test time and budget, ensuring the certainty of such cases, therefore, it is important that the origin of test cases does not introduce errors or ambiguities in the process; there are different solutions to this issue, such as formal and semiformal definitions. In this work we propose the creation of an automatic generator of functional test cases that uses as input use cases described in a specific requirements definition language (RDL), to obtain a directed graph and later, apply methods that allow obtaining test cases taking into consideration the percentage of coverage, the time available to test or type of scenario.

Keywords— *Coverage, Requirements Definition Language, Test Case, Time to Test, Use Case.*

1. INTRODUCCIÓN

Anteriormente, probar era una actividad de poco interés entre los programadores, incluso se consideraba una pérdida de tiempo, sin embargo, esto ha cambiado, y actualmente probar es una práctica supervisada que da paso a los casos de prueba (especificaciones del proceso de ejecutar un producto de software para encontrar problemas en éste). Detectar errores en el código del software, identificar defectos o comportamientos alejados del esperado son ventajas que ofrecen los casos de prueba, ya que, si tales fallas son corregidas a tiempo, se asegura el comportamiento del producto final.

Debido a la creciente demanda de sistemas cada vez más complejos, un componente de software es capaz de dar origen hasta a cientos de casos de prueba y es común que el tiempo para probar no sea amplio, lo que lleva al equipo de pruebas a trabajar con el mínimo indispensable de casos para determinar que el elemento funciona adecuadamente.

Para dar solución a lo antes expuesto, los esfuerzos actuales se centran en generar casos de pruebas de forma automática, para aprovechar al máximo el tiempo de prueba de un producto programado y generar software más apegado al funcionamiento deseado por el usuario. Sin embargo, cuando se trata de pruebas funcionales, el proceso se complica debido a que a) el origen de las pruebas puede introducir errores como al utilizar Lenguaje Natural (NL), o b) incurrir en un mayor consumo de recursos si se usan especificaciones formales; por otra parte, en los trabajos relacionados que se revisaron, se encontró que no siempre se consideran las limitantes que tiene el proceso de prueba (tiempo disponible por ejemplo).

Las empresas de desarrollo de software a la medida tienen una necesidad creciente en el campo de las pruebas; en este proyecto se trabaja con una empresa líder en el mercado, misma que por políticas de privacidad será mencionada como “empresa A”; dicha empresa proporcionó para la realización de este proyecto un lenguaje propio de definición de requerimientos (RDL).

En este artículo se presenta un proceso para la generación automática de casos de prueba funcionales, que toma como punto de partida los casos de uso descritos en un lenguaje de definición de requerimientos propio de la “empresa A”,

tomando en consideración limitantes como el grado de cobertura y el tiempo disponible para pruebas.

2. MARCO TEÓRICO

En esta sección, se exponen los conceptos más necesarios e importantes para comprender mejor el presente artículo.

2.1 Pruebas de software

G. J. Myers [1] definió que una prueba de software es el proceso de ejecutar un programa con la intención de encontrar errores; este proceso ayuda a verificar el comportamiento del producto y se describe en un formato conocido como caso de prueba.

Las pruebas se clasifican de la siguiente manera [2]:

a) Pruebas unitarias: se enfocan en la verificación de la unidad más pequeña del diseño de software: como clase, componente o módulo de software. En estas pruebas se consideran criterios de calidad tanto técnicos como funcionales.
b) Pruebas integrales: tienen como objetivo tomar los componentes probados de manera individual y verificarlos como un todo.
c) Pruebas funcionales: verifican el comportamiento de conjuntos de componentes desde el punto de vista del usuario final, es decir, se centran en verificar las funciones del sistema, dejando de lado los elementos técnicos de calidad.

2.2 Criterios de adecuación de pruebas

Los evaluadores de software necesitan un marco para decidir qué elementos y datos de prueba son necesarios para considerar que los esfuerzos de prueba son lo suficientemente adecuados para finalizar el proceso con la confianza de que el software funciona correctamente. Dicho marco existe en forma de criterios de adecuación de prueba.

Formalmente, un criterio de adecuación de datos de prueba es una regla que sirve para determinar si se realizaron suficientes pruebas. Estos criterios representan estándares mínimos para probar un programa, por lo que una forma acertada para referirse a dichos criterios es el análisis de **cobertura**.

Cuando un objetivo de prueba relacionado con la **cobertura** se expresa con un **porcentaje**, se le denomina **grado de cobertura** [3]**.**

2.3 Grafo dirigido

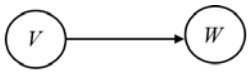
En [4], los grafos se definen como una estructura de datos no lineal, la cual sirve para modelar diversas aplicaciones. Un grafo *G* consta de un conjunto de vértices o nodos *V* y un conjunto de arcos *A*, cada uno de los cuales une un vértice con otro, esto se presenta en la siguiente fórmula.

$$G = (V, A) = (N, A) \quad [1]$$

Los arcos que unen los nodos también se llaman *aristas del grafo* y se representan por medio de un par de elementos, *(vi, vj)*, donde los elementos son los nodos que une el arco. Si en un grafo los arcos tienen una dirección, el grafo se llama *grafo dirigido u orientado.*

En un grafo orientado, cada arco se representa por medio de un par ordenado *(v, w)* donde el primer elemento es el nodo origen o fuente, y el segundo es el nodo destino de ese arco, por lo tanto, se puede decir que el arco que va desde *v* a *w,* es adyacente a *v*. Un grafo orientado también se llama *dígrafo* y se aprecia un ejemplo de éste en la Figura 1.

Figura 1. Representación de un grafo dirigido.



Fuente: elaboración propia a partir de [4].

2.3 Lenguaje de definición de requerimientos

Usualmente, los integrantes de un equipo de desarrollo de software, utilizan el NL (*Natural Language,* Lenguaje Natural) para describir las especificaciones de requerimientos, lo que hace que estas descripciones sean más propensas a fallas debido a que cada individuo suele interpretar o usar el lenguaje con diferentes variaciones [5].

Un RDL (*Requirements Definition Language,* Lenguaje de Definición de Requerimientos), es aquel que permite representar de forma estandarizada las características deseables de un software específico [6]; esto es porque, siendo estandarizado, el riesgo de ambigüedades disminuye.

Por lo anterior, se considera deseable el uso de lenguajes más estandarizados como los formales o los de creación propia, como es el caso del RDL propuesto por la "empresa A".

2.4 Casos de Uso

Un caso de uso (UC) es una unidad coherente del funcionamiento de un sistema de software que describe una secuencia de acciones que realiza dicho sistema y que lleva a un resultado de valor a un actor específico. Un UC muestra la descripción escrita en lenguaje natural de los pasos y demás características del mismo [2].

En los casos de uso se identifican tres tipos diferentes de escenarios o flujos de ejecución (secuencias de acciones específicas que el actor requiere, y las respuestas apropiadas del sistema):

- Principal o básico: es el flujo más común de acciones entre el sistema y los actores.

- Alterno: flujo independiente que suele dispararse por alguna decisión o acción en el flujo principal y su flujo se reintegra al escenario principal.
- Excepción: representan situaciones que se anticipan a errores que pueden ocurrir durante la ejecución del caso de uso, establecen mecanismos para manejar tales incidencias.

3. TRABAJOS RELACIONADOS

El Modelo Basado en Pruebas (MBT) es un enfoque para alcanzar los niveles de seguridad requeridos para los sistemas críticos, sin embargo, éste está sujeto a fallas debido a que las anotaciones de entrada se hacen en Lenguaje Natural (NL), por lo que en [7] se propuso NAT2TESTSCR como una estrategia para la generación automática de casos de prueba a partir de requisitos escritos en Lenguaje Natural Controlado (CNL) gracias a SysReq-CNL. Como resultado, se logró la construcción de casos de prueba hasta 30 minutos más rápido en comparación con los casos escritos por los desarrolladores de software.

En [8] se propuso la Generación de Pruebas del Modelado de Casos de Uso (UMGT), un enfoque capaz de extraer información de los casos de uso a partir del Procesamiento de Lenguaje Natural (NLP). UMGT restringe los casos de uso con el Modelado de Casos de Uso Restringido (RUCM), que es una plantilla con reglas y restricciones que ayudan a reducir la imprecisión de los casos de uso. El enfoque funcionó favorablemente en un caso de estudio con especificaciones de casos de uso para un sistema de sensores automotrices.

Para probar la suficiencia del software en escenarios reales del negocio, Yazdani et al., [9] definieron una técnica para generar casos de prueba a partir de especificaciones de requisitos utilizando modelos de procesos de negocios. Por lo que, primero se genera un BPMN (*Business Process Model and Notation*, Modelo y Notación de Procesos de Negocio) para representar el modelo de proceso de negocio, para luego transformarlo en un grafo de estado del cual se extraen los casos de prueba utilizando una herramienta llamada Spec Explorer, logrando así, ahorrar tiempo y costos para la generación de casos de prueba, y aumentando la confiabilidad del software, por la compatibilidad de las pruebas con el flujo real del proceso de negocio.

En [10] se analizó que se origina ambigüedad en los casos de prueba debido a que el proceso de escritura de los requisitos y de los mismos casos se hace de forma manual, por lo que se propuso un enfoque para definir criterios de análisis de pruebas. Para la definición de requisitos, se usa NL y RUCM, que fue desarrollado a partir de TCS (*Test Case Specifications*, Especificación de Casos de Prueba). Finalmente, se obtuvo una herramienta llamada Toucan4Test, que implementa criterios de cobertura de estructura en las especificaciones RUCM, logrando generar TCS y casos de prueba de forma sistemática y automática a partir de las especificaciones de RUCM.

En la literatura estudiada se hace una transformación de casos de uso o de requerimientos para eliminar la ambigüedad del NL, llegando a la generación automática de casos de prueba de forma más rápida; si bien el presente trabajo también busca generar casos de prueba a partir de casos de uso, la aportación del mismo va en el sentido de utilizar un RDL y considerar filtros para la generación de casos de prueba: grado de cobertura deseado, tiempo para probar y tipos de flujos de ejecución.

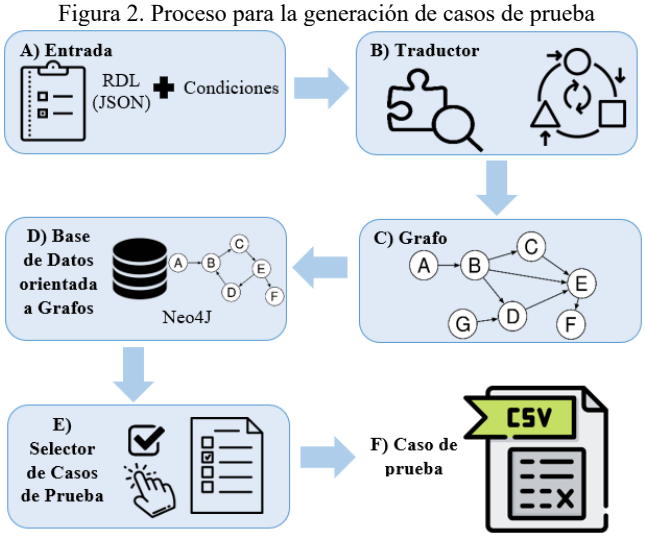
4. PROCESO DE LA HERRAMIENTA GENERADORA DE CASOS DE PRUEBA

Los requisitos a cubrir por el generador, determinados por la "empresa A" son los siguientes:

- Obtener casos de prueba automáticos de forma más rápida y eficiente que la generación manual.
- Procesar casos de uso escritos en RDL bajo el formato JSON.
- Representar los casos de prueba mediante grafos dirigidos.
- Priorizar los escenarios de prueba y obtener, gracias a una serie de algoritmos, los escenarios que correspondan a cierto grado de cobertura tomando en consideración el tiempo disponible para probar.
- Obtener los casos de prueba resultantes en archivos de texto separados por comas (.csv).

Los pasos que debe seguir la herramienta para cumplir con los requisitos mencionados se ilustran en la Figura 2.

Figura 2. Proceso para la generación de casos de prueba



A) Entrada: el punto de partida para la generación automática de casos de prueba son casos de uso descritos en un RDL propio de "la empresa", en formato JSON, que sumados a las condiciones que dictamine el usuario (si

desea abarcar cobertura, tiempo o conjuntos de escenarios para probar), forman la entrada al proceso. El RDL no sólo representa los pasos de los casos de uso, también incluye detalles como los tipos de datos a capturar y los rangos de valores válidos en la captura. Además, considera información adicional para la construcción del grafo dirigido, por ejemplo, la probabilidad de tomar un curso alterno o de realizar la extensión de un caso de uso.

B) Traductor: en esta fase se procesan la entrada y las condiciones del usuario para crear un grafo dirigido, donde la estructura de éste contempla los pasos del caso de uso, por lo que los caminos reconocibles en el grafo inicialmente representan los escenarios (principales, alternos y de excepción) que forman al caso de uso. Además, los nodos contienen información relevante para expandir el grafo, debido a que un paso del caso de uso no necesariamente equivale a un paso en el caso de prueba; retomando el ejemplo del campo de captura de tipo numérico, el caso de uso sólo especifica qué acción tomar si el valor capturado no es correcto, pero en el grafo para pruebas debe representarse, en nodos separados, qué hacer en caso de que se haya capturado un valor por encima del rango, por debajo de éste, o si no se escribió nada; por lo cual, los nodos de grafo original que representan capturas de datos, deben incluir información del tipo y rango de valores aceptables en dichas capturas.

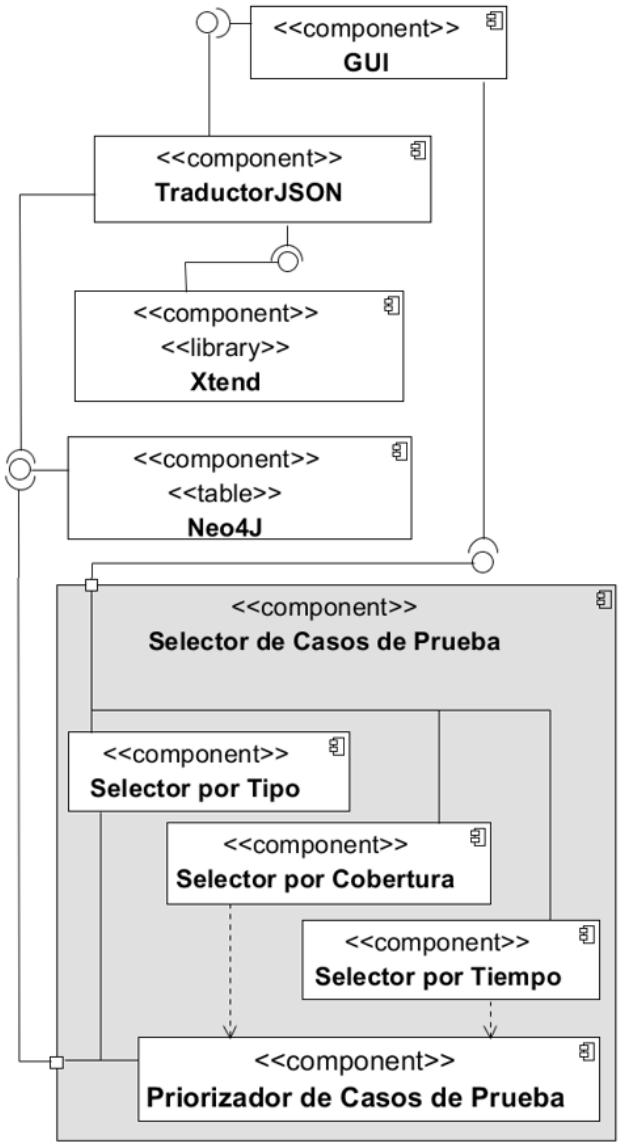
C) Grafo: una vez creado el grafo para pruebas, es decir, considerando los nodos extra para validaciones, éste se guarda en la **D) Base de Datos Orientada a Grafos** Neo4J, y hasta este punto, cabe destacar que desde la misma base es posible realizar consultas para obtener el camino más corto o más largo según la secuencia de nodos tomado en consideración tanto el escenario básico como los escenarios alternos y/o de excepción. No basta con reconocer todas las secuencias de nodos posibles, por lo que se hace uso de un conjunto de algoritmos sobre el grafo con el fin de priorizar los escenarios de prueba, determinar cuáles de estos escenarios son el número mínimo que se necesitan probar para decir que funciona adecuadamente el componente (cobertura) y saber qué cantidad de escenarios relevantes es posible probar en un tiempo.

E) Selector de Casos de Prueba: continuando con el proceso, el grafo almacenado y las consideraciones de priorización mencionadas antes alimentan al **Selector de Casos de Prueba**, donde se eligen los caminos que representan los escenarios del caso de prueba en función de las condiciones especificadas por el usuario desde la entrada; es decir, se hacen las operaciones necesarias sobre Neo4J para obtener una serie de combinaciones de pasos que satisfagan las condiciones del usuario, como cuestiones de cobertura, tiempo disponible para realizar la prueba y/o tipo de escenario del Caso de Uso a probar; finalmente, el resultado de este paso, se exporta para dar origen al **F) Caso de Prueba Resultante**, que es un archivo de texto de plano separado por comas que contiene las especificaciones del caso de prueba con las condiciones especificadas por el usuario.

5. ARQUITECTURA DE LA HERRAMIENTA

La arquitectura propuesta toma como base el proceso descrito en la sección anterior para identificar los elementos de software necesarios para cumplirlo y las tecnologías de las que se va auxiliar: JavaFx, Xtend y Neo4J; en la Figura 3 se muestra el Diagrama de Componentes resultante.

Figura 3. Arquitectura para el generador de casos de prueba.



Como lenguaje de programación, por requerimiento de "la empresa", se trabaja con *Xtend*, un lenguaje de programación para la máquina virtual de Java y que es capaz de utilizar bibliotecas propias de Java.

El componente *GUI* (*Graphic User Interface,* Interfaz Gráfica de Usuario) representa el conjunto de clases y archivos FXML, basados en JavaFX, que controlan la interacción entre el usuario final y la aplicación. Permite la

captura de 1) el o los archivos JSON, que representan los casos de uso bajo el RDL de "la empresa", 2) el porcentaje de cobertura deseado, 3) el tiempo disponible para probar y 4) tipo de curso del UC (básico, de excepción, alterno) a incluir en la prueba. Es necesario aclarar que normalmente sólo se requiere un archivo JSON por caso de uso, pero si dicho caso tiene extensiones o inclusiones, se necesita proveer los archivos de los casos relacionados.

El componente *TraductorJSON*, se auxilia de componentes de otras tecnologías como son *Xtext* y *Neo4J*. El componente *Xtext* representa una biblioteca para Java que permite generar editores de código y que, en este caso, va a proveer algunas de las funciones para el tratamiento de las cadenas JSON. El componente *TraductorJSON* va a tomar los objetos incluidos en las cadenas JSON para transformarlos en nodos del grafo dirigido que se almacena en el gestor representado por el componente *Neo4J*; además, va a interpretar la información relacionada con validaciones, incluida en el JSON, para crear nuevos nodos que representan pasos en las pruebas de datos. En este punto, el grafo representa el caso de prueba completo, con todos los pasos posibles.

El componente *Selector de casos de prueba,* es el responsable de obtener los pasos del caso de prueba funcional a partir del grafo almacenado en *Neo4J* y entregarlos al componente *GUI* para que éste los exporte como un archivo de texto separado por comas. El componente *Selector de casos de prueba* incluye 4 componentes más: *Priorizador de Casos de prueba, Selector por Tipo, Selector por Cobertura y Selector por Tiempo;* éstos tienen como responsabilidad aplicar una serie de operaciones particulares sobre el grafo para seleccionar los pasos del caso de prueba que serán incluidos en el resultado final tomando en cuenta las condiciones indicadas por el usuario en el componente *GUI.*

El componente *Priorizador de Casos de Prueba,* es el responsable de ordenar los escenarios del caso de prueba, es decir, de identificar todas las rutas (secuencias de nodos) sobre el grafo y darles una prioridad. La mayor prioridad está relacionada con la cercanía al curso básico o principal del caso de uso original, mientras que la menor prioridad se relaciona con los cursos de excepción.

Los componentes *Selector por Cobertura* y *Selector por Tiempo* toman en cuenta la prioridad asignada por el componente anterior, debido a que éstos determinan cuántos y cuáles elementos, del conjunto obtenido del *Priorizador,* cubren el porcentaje de cobertura o el tiempo disponible, respectivamente, de acuerdo a lo que haya indicado el usuario inicialmente.

El comportamiento del componente *Selector por Tipo* es similar a los antes descritos, sin embargo, no toma en cuenta las prioridades asignadas por el componente *Priorizador,* ya que elige directamente, sobre el grafo almacenado, los nodos cuyo origen, en la cadena JSON, corresponde al tipo de escenario de caso de uso elegido por el usuario.

6. RESULTADOS

Actualmente, la primera versión del generador, producto de la arquitectura antes presentada, demostró ser capaz de recibir, a través de su interfaz gráfica, un archivo JSON correspondiente a un caso de uso descrito con el RDL de "la empresa". El componente *TraductorJSON* convierte las cadenas JSON en objetos, genera un nodo del grafo con cada uno de los pasos incluidos en los objetos y almacena el conjunto como un grafo dirigido en *Neo4J*. Es importante destacar que, por el momento, sólo se está considerando el curso básico o principal del caso de uso y, además, la extensión de casos, es decir, el unir dos grafos bajo ciertas condiciones.

El *Selector de Casos de Prueba* toma el grafo almacenado y genera los escenarios del caso de prueba para el caso de uso, tomando en cuenta sólo el curso básico o principal que por ahora es el único representado en el grafo dirigido. Dichos escenarios del caso de prueba se exportan al usuario en archivos de texto separados por comas, con extensión .csv.

7. CONCLUSIONES Y RECOMENDACIONES

La utilización de un RDL para limitar las expresiones incluidas en un caso de uso dio los resultados esperados, ya que no inserta errores en los casos de prueba finales; además, resulta muy provechosa la serialización del caso de uso como un archivo JSON, pues estandariza las características de los pasos de los diferentes casos de uso de tal forma que permite hacer reglas genéricas para armar grafos.

El uso de un grafo dirigido también mostró los efectos deseados, ya que es más fácil obtener pruebas apegadas al comportamiento descrito por el usuario debido a la multitud de algoritmos existentes para implementar recorridos sobre este tipo de estructuras. Además, *Neo4J* facilitó mucho el trabajo de representación y almacenado del grafo, pues gracias a su servicio operable desde el *browser* es posible visualizar el grafo, lo que facilitó el desarrollo de los componentes al contar con elementos visuales durante las pruebas del generador.

Con lo anterior, se demostró que la arquitectura propuesta, al menos en la primera versión implementada, logra el objetivo de crear casos de prueba más rápido y con menos errores que si se hacen de forma manual, lo que da paso a continuar con el proyecto para completar la funcionalidad del resto de los componentes de la arquitectura.

8. TRABAJO A FUTURO

Para continuar con el proyecto, se tiene contemplado construir la segunda versión del generador, que incluye robustecer el grafo para obtener la representación de todos los cursos que incluya el caso de uso, completar el componente *Selector de Casos de Prueba,* para entregar

casos de prueba tomando en cuenta cobertura, tiempo y tipo de escenario.

Finalmente, se espera probar el Generador de Casos de Prueba con casos de uso definidos por la "empresa A", para comprobar su funcionamiento frente a situaciones empresariales reales.

9. AGRADECIMIENTOS

Los autores agradecen el financiamiento del consejo Nacional de Ciencia y Tecnología (Conacyt) y el apoyo del Tecnológico Nacional de México.

10. REFERENCIAS

[1] G. J. Myers, *The Art of Software Testing, Second Edition*, 2.ª ed. New Jersey: Wiley, 2004.
[2] R. S. Pressman, V. Campos Olguín, J. Enríquez Brito, B. J. Ferro Castro, y C. Villegas Quezada, *Ingeniería del software: un enfoque práctico*. México D. F.: McGraw-Hill, 2010.
[3] I. Burnstein, *Practical Software Testing*. New York: Springer-Verlag, 2003.
[4] *Introducción a la Teoría de Grafos*. ELIZCOM S.A.S.
[5] H. M. Sneed, «Testing Against Natural Language Requirements», en *Proceedings of the Seventh International Conference on Quality Software*, Washington, DC, USA, 2007, pp. 380–387.
[6] «Notas_Analisis_Requerimiento.pdf». .
[7] G. Carvalho *et al.*, «NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications», *Sci. Comput. Program.*, vol. 95, pp. 275-297, dic. 2014.
[8] C. Wang, F. Pastore, A. Goknil, L. Briand, y Z. Iqbal, «Automatic Generation of System Test Cases from Use Case Specifications», en *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, New York, NY, USA, 2015, pp. 385–396.
[9] A. Yazdani Seqerloo, M. J. Amiri, S. Parsa, y M. Koupaee, «Automatic test cases generation from business process models», *Requir. Eng.*, vol. 24, n.º 1, pp. 119-132, mar. 2019.
[10] M. Zhang, T. Yue, S. Ali, H. Zhang, y J. Wu, «A Systematic Approach to Automatically Derive Test Cases from Use Cases Specified in Restricted Natural Languages», en *System Analysis and Modeling: Models and Reusability*, 2014, pp. 142-157.