Ingeniería Industrial/ISSN 1815-5936/Vol. XLIII/No. 1/enero-marzo/2022/4303

ARTICULO ORIGINAL

INFORMÁTICA EMPRESARIAL

Extensiones de Mtest.search para la generación de código de prueba

Mtest.search extensions for test code generation

Alejandro Miguel Güemes-Esperón

https://orcid.org/0000-0001-9704-9449

Martha Dunia Delgado-Dapena

https://orcid.org/0000-0002-2601-3462

Perla Beatriz Fernández-Oliva

https://orcid.org/0000-0002-33604447

Heydi Margarita Henry-Chibas

https://orcid.org/0000-0002-6255-4311

Universidad Tecnológica de La Habana José Antonio Echeverría (Cujae). La Habana, Cuba

Correo electrónico: aguemes@ceis.cujae.edu.cu, marta@ceis.cujae.edu.cu, perla@ceis.cujae.edu.cu, hhenry@ceis.cujae.edu.cu

RESUMEN

Las pruebas de software se centran en la detección de defectos o fallos durante la ejecución del código. La prueba constituye una tarea creativa desafiante, por lo que se hace necesaria su automatización. La adopción de buenas prácticas y estrategias de pruebas, contribuye a la elevar la eficiencia de las empresas desarrolladoras de softwares. El modelo MTest.search para la generación automática de pruebas unitarias tiene definidos mecanismos de extensión del modelo de dominio, de prueba y de ejecución. En este trabajo se presentan mecanismos para extender el modelo de reducción basado en búsquedas. Las extensiones propuestas tienen en cuenta los objetos y conjuntos involucrados en el código fuente, y potencia la detección de defectos o fallos a partir de la significación de los valores y caminos/escenarios involucrados en la prueba. Para validar la propuesta se definieron dos casos de estudios empleando métodos clásicos y de proyectos reales.

Palabras clave: calidad de software, pruebas unitarias, generación de código de prueba.

ABSTRACT

Software testing focuses on detecting defects or failures during code execution. Testing is a challenging creative task, requiring automation. The adoption of good practices and testing strategies contributes to increasing the efficiency of software development companies. MTest.search model for

automatic unit test generation has defined domain model extension, test, and execution mechanisms. In this work, mechanisms to extend the search-based reduction model are presented. The proposed extensions take into account the objects and sets involved in the source code, and enhance the detection of defects or failures based on the significance of the values and paths / scenarios involved in the test. To validate the proposal, three case studies were defined using classical methods and real projects.

Keywords: software quality, unit tests, test code generation.

I. INTRODUCCIÓN

Las pruebas de software son procesos orientados a demostrar que un programa realiza las funciones para las cuales fue construido [13]. Las pruebas son exitosas cuando detectan defectos o fallos, por lo que tributan a la calidad dentro del proceso de desarrollo de software [14, 25, 30]. No obstante, las pruebas son costosas y a menudo abarcan más del 50% de los costos totales de desarrollo [2, 11, 25]. Debido a esto, se hace necesario y es de gran beneficio, la automatización del proceso de pruebas con el objetivo de disminuir su costo e incrementar su efectividad [23, 33].

Actualmente múltiples investigadores centran su atención en el proceso de desarrollo de pruebas de software, ocupando un lugar fundamental en los trabajos científicos [10, 15, 17]. Se mantienen como problemáticas la generación de caminos y valores de pruebas para apoyar el diseño de los casos de prueba [12, 16, 25, 33].

Existen diferentes herramientas como JUnit, NUnit y PHPUnit que permiten ejecutar pruebas unitarias de forma automática, pero carecen de funcionalidades que asistan al desarrollador en el diseño de los casos de pruebas [24]. En la actualidad existen propuestas de herramientas para la generación de código de pruebas unitarias en diferentes formatos de salida [5, 27, 28, 32]. Sin embargo, todas no son libres, y la mayoría se centran en un lenguaje de programación por lo que sería necesario utilizar diferentes herramientas en una misma empresa de desarrollo de software que utiliza diferentes lenguajes de programación en correspondencia con los requisitos del cliente, para la automatización de las pruebas unitarias. En la actualidad existen varios modelos para la automatización de pruebas de software [1, 3, 6, 7, 33]. El Modelo para la Generación Automática de Pruebas basado en Búsquedas (*MTest.search*) definido por los miembros del grupo de investigación **Calidad y pruebas de software** de la Cujae, cuenta con mecanismos de extensión para los modelos de dominio, de prueba y de ejecución [1, 19]. La contribución fundamental de MTest.search está en la propuesta de modelos de optimización que permiten diseñar casos de pruebas con combinaciones de valores reducidos y que tienen un soporte automatizado que puede ser fácilmente integrado a ambientes de trabajo. Dada la necesidad de mejorar la efectividad de las suites de pruebas generadas para detectar defectos o fallos, en el grupo de investigación se propuso incluir la significación de los valores de prueba que se obtienen, y de los caminos/escenarios de la unidad bajo prueba. Sin embargo, MTest.search no cuenta con mecanismos de extensión del modelo de reducción basado en búsqueda, lo que imposibilita la incorporación de nuevos criterios a tener en cuenta para elevar la efectividad de las pruebas.

Este artículo presenta mecanismos para extender el modelo de reducción basado en búsquedas, así como, extensiones de los componentes de dominio y de ejecución para la generación automática de pruebas unitarias a partir de código fuente Java. Los componentes permiten generar casos de pruebas a partir de código fuente que contengan objetos y conjuntos. Se incluye el componente de extensión del modelo de reducción basado en búsquedas implementado, para potenciar la detección de errores a partir de la significación de los valores y caminos/escenarios, en dependencia de la técnica de diseño de casos de pruebas empleada en cada caso.

II. MÉTODOS

La propuesta se basa en componentes de extensión de MTest.search para la generación de código de prueba teniendo en cuenta los objetos y conjuntos involucrados en el código fuente y la significación de las variables y caminos que intervienen. Se realizaron revisiones de la literatura y se diseñaron

casos de estudios para validar la propuesta a partir del análisis cualitativo de los resultados obtenidos durante la generación de código de prueba con los componentes que se presentan.

Modelo para la Generación Automática de Pruebas basado en Búsquedas

Durante el proceso de desarrollo de software se utilizan modelos para la representación de los artefactos necesarios para la descripción del producto que se está desarrollando. Con anterioridad se han definido modelos para el proceso de pruebas de software. *MTest.search*, consta de flujos de trabajo para la ejecución de pruebas tempranas en el entorno de producción, Modelos de Optimización para reducción de casos de pruebas funcionales y unitarias, y Herramientas Automatizadas Integradas que dan soporte a la ejecución de los flujos de trabajo [1]. El *MTest.search* cuenta con mecanismos de extensión, que permiten la adaptación de sus modelos y herramientas, tanto por desarrolladores avanzados como principiantes, a diferentes entornos productivos y empresariales [19].

MTest.search contiene los siguientes elementos:

a. **Modelo de Dominio**, con sus respectivos modelos de descripción de dominios fuente.
b. **Modelo de pruebas** y **Modelo de pruebas reducido**.
c. **Modelo de reducción basado en búsquedas**.
d. **Modelo de ejecución**, con sus respectivos modelos de descripción de código destino.

Se han desarrollado herramientas que dan soporte a *MTest.search* permitiendo:

- la generación de valores significativos para la prueba teniendo en cuenta diferentes técnicas de diseño de casos de pruebas [19; 20]
- la creación de casos de pruebas con el Generador de Casos de Pruebas (GeCaP), para satisfacer la técnica del camino básico, a partir de los valores previamente generados [4, 20-22, 26, 31].

Sin embargo, las propuestas anteriores no se centran en el análisis de significación de los valores según las técnicas aplicadas para su obtención, ni incluyen criterios para determinar cuáles son los caminos más importantes. Darles solución a las limitaciones antes mencionadas durante el proceso de generación de las suites de pruebas, tributará a la detección de defectos o fallos. Se tiene en cuenta el análisis gramatical del código fuente Java para generar un Árbol de Sintaxis Abstracta (*AST*, *por sus siglas en inglés*), a partir del cual se genera el Grafo de Control de Flujo (*CFG, por sus siglas en inglés*) correspondiente, que constituye el lenguaje intermedio [8, 9]. Por último, el componente Generador de Código de Prueba (GeCodP) integra los componentes mencionados anteriormente para la generación de código de pruebas unitarias a partir de código fuente en lenguaje Java [9, 18, 35]. Estas propuestas solo tienen en cuenta la entrada de variables de tipo de datos simples (numérico, lógico y cadena), lo que constituye una limitación.

Las herramientas están organizadas mediante el patrón **Arquitectura basada en componentes** y está orientado a la experticia del usuario de *MTest.search*. Se expresa a través de una arquitectura compuesta por tres capas: capa de generación de suite de pruebas, capa de extensión y capa de entorno usuario. La arquitectura de las herramientas de soporte al modelo *MTest.search*, se observa en la figura 1, donde se señalan los componentes de la capa de extensión en los que se trabajará [9, 19].

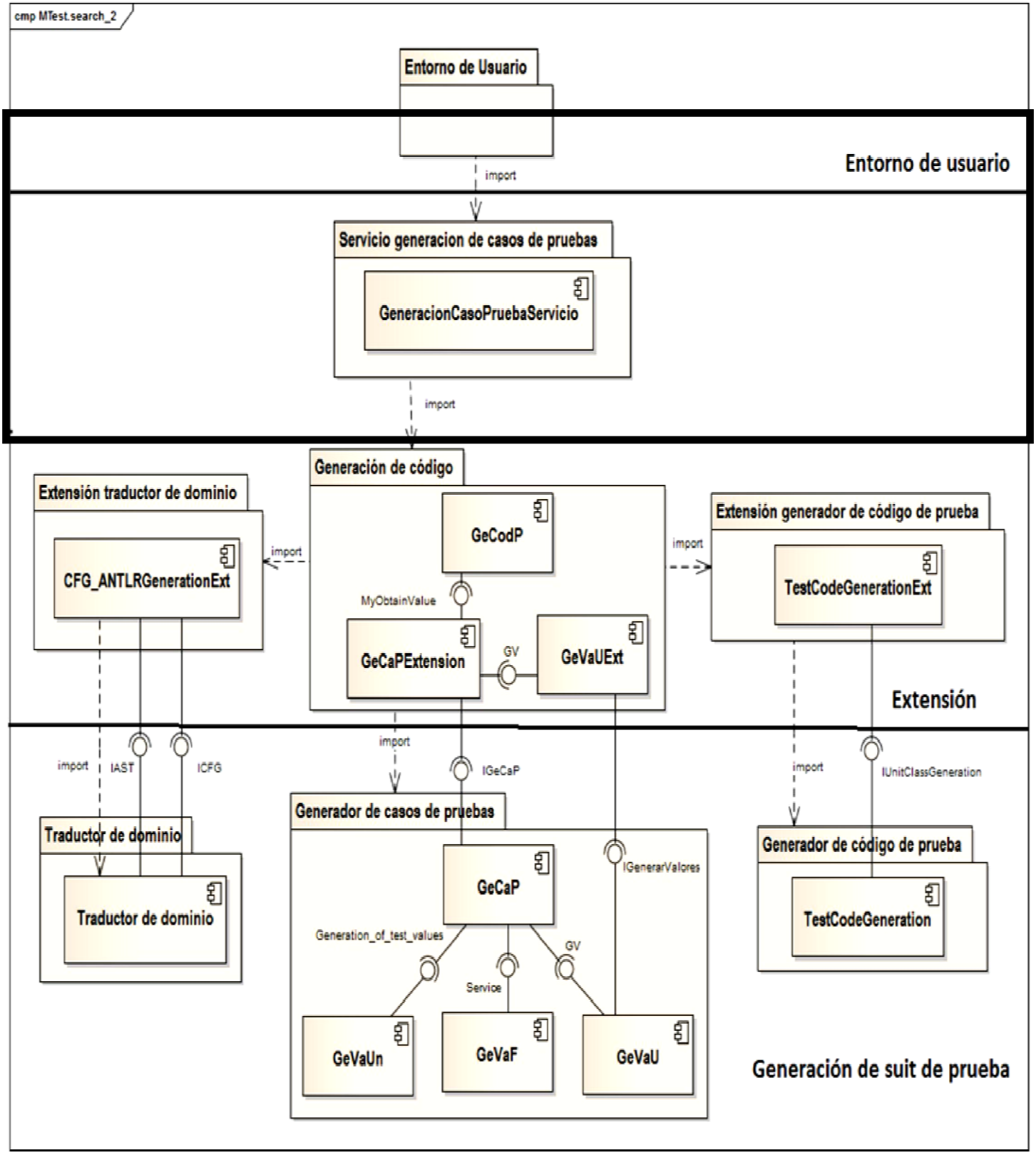

Fig. 1. Arquitectura de las herramientas de soporte al modelo *MTest.search*

Mecanismos de extensión del modelo de reducción basado en búsqueda

Para extender el modelo de reducción basado en búsqueda es necesario implementar en la capa de Extensión un componente (GeVaUExt) que extienda al componente Generador de Valores GeVaU (contiene el modelo de optimización), que se encargue de incorporar los nuevos criterios a tener en cuenta para influir en la efectividad de los casos de pruebas que se generen. Para esto se debe crear una interface en GeVaU que se establezca la comunicación entre ambos componentes. Aquí se incluyen los métodos que deben ser redefinidos por la extensión, según los nuevos criterios a tener en cuenta en el análisis de los valores y las modificaciones al modelo de optimización que se deseen adicionar. Todas las clases deben implementar dicha interface.

Para incluir las modificaciones realizadas por la extensión de GeVaU es necesario implementar un componente de extensión (GeCaPExt) del componente Generador de Casos de Pruebas (GeCaP) que posee la clase *ObtainValues* encargada de iniciar el proceso de generación de valores significativos de las variables. Se debe crear una clase en GeCaPExt que herede de *ObtainValue* e implemente una interfaz de GeCaP para redefinir los métodos encargados de construir los ficheros de intercambio entre los componentes, con la información de los valores de las variables involucradas. Luego, el componente GeCodP debe importar la extensión de GeCaP, para poder acceder a los métodos modificados. Para integrar los componentes GeVaUExt y GeCaPExtension es necesario crear una componente de servicio que logre la comunicación entre la capa de Extensión y la de Entorno de usuario.

Componente para la generación del CFG a partir de código fuente en lenguaje Java (CFG_ANTLRGenerationExt)

El componente que se presenta en este trabajo constituye una solución a la generación del CFG a partir de código fuente Java para el tratamiento de tipos de datos objeto y conjunto. Los requisitos de CFG_ANTLRGenerationExt se muestran en la figura 2, a través de un diagrama de casos de uso del sistema.

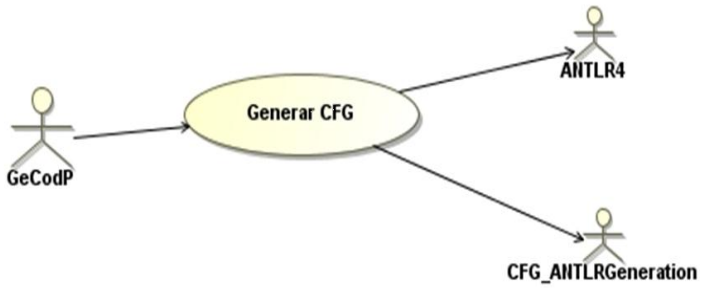

Fig. 2. Diagrama de CUS del componente CFG_ANTLRGenerationExt

Para generar el CFG se parte del código fuente en Java, este es procesado por la herramienta ANTLR4 [29] detectando elementos de la gramática y generando el árbol de sintaxis correspondiente (*ATN*, *por sus siglas en inglés*). Posteriormente, se determina cuáles son los parámetros de entrada y se genera el fichero con la información de estos. Luego, el componente a partir del análisis del ATN detecta las llamadas a métodos y a atributos de objetos y de conjuntos existentes, pasándole la información a GeCodP.

El componente CFG_ANTLRExt recibe cuatro ficheros: Llamadas.txt, ParametrosEntrada.txt, ParametrosEntradaObjetos.txt y ParametrosEntradaConjuntos.txt. Con el procesamiento de la información contenida en Llamadas.txt, se genera un código copia (codigoCopiaA), en el que se sustituyen las llamadas indicadas por el usuario, por el valor fijo correspondiente, según la descripción dada. A partir del análisis de la descripción de los objetos y conjuntos que constituyen parámetros de entrada, se transforma el método y se crea otro método copia (codigoCopiaB) donde se sustituyen a los objetos por sus atributos de dominios simples (lógico, cadena y numérico). En este punto, ya el codigoCopiaB puede ser procesado por el componente CFG_ANTLRGeneration para la generar el CFG.

Componente para la generación de código de pruebas unitarias en lenguaje Java (TestCodeGenerationExt)

Para generar código de pruebas unitarias en Java que pueda ser ejecutado en JUnit, se parte del CFG generado por el componente CFG_ANTLRGenerationExt con variables de dominios de entrada de datos simples. Luego GeCaP se comunica con el resto de los componentes y se encarga de diseñar los casos de pruebas. A partir de las combinaciones de valores para cada caso de prueba, TestCodeGeneration genera código de prueba correspondiente a partir de la información de los ficheros ParametrosEntradaObjetos.txt, ParametrosEntradaConjuntos.txt y CodigoCopiaB.txt (Código del método con la transformación de tipos de datos complejos a simples).

Si existen parámetros de entrada de dominios complejos, se modifican los métodos de prueba generados por TestCodeGeneration, en función de los parámetros originales, cuya información aparece recogida en los ficheros de textos antes mencionados. TestCodeGenerationExt modifica el código de prueba generado por TestCodeGeneration teniendo en cuenta la creación de instancias de tipos de datos complejos, con los valores generados, para la ejecución de este código de prueba en JUnit, en dependencia de lo que desea probar el usuario. En la figura 3 se observa un diagrama de CUS del componente TestCodeGenerationExt

Fig. 3. Diagrama de CUS del componente TestCodeGenerationExt

El componente CFG_ANTLRExt integra el análisis de la gramática del código fuente en lenguaje Java a partir del procesamiento que realiza la herramienta ANTLR v4. Para ello, se ha definido la gramática Java para ANTLR4.

En la figura 4 se muestra un fragmento del fichero JavaLexer.g que contiene los elementos léxicos del lenguaje Java.

JavaLexer.g4

```
1  lexer grammar JavaLexer;
2
3      // Keywords
4
5      ABSTRACT: 'abstract';
6      ASSERT: 'assert';
7      BOOLEAN: 'boolean';
8      BREAK: 'break';
9      BYTE: 'byte';
10     CASE: 'case';
11     CATCH: 'catch';
```

Fig. 4. Fragmento del fichero JavaLexer.g4

En la Figura 5 se muestra un fragmento del fichero JavaParser.g4 que contiene estructuras gramaticales implementadas para el procesamiento de código fuente en Java y su transformación en un ATN.

```
parser grammar JavaParser;

    options { tokenVocab=JavaLexer; }

    compilationUnit
    : packageDeclaration? importDeclaration* typeDeclaration* EOF
    ;

    packageDeclaration
    : annotation* PACKAGE qualifiedName ';'
    ;

    importDeclaration
    : IMPORT STATIC? qualifiedName ('.' '*')? ';'
    ;
```

Figura 5. Fragmento del fichero JavaParser.g4

Luego de transformar el código fuente Java a un ATN, el componente determina cuáles son los parámetros de entrada del método que se desea probar. Esta información se almacena en el fichero ParametrosEntrada.txt (Ver Figura 6) con el identificador y el tipo de dato Java de los parámetros.

```
Cantidad de parametros (N)
Parametro 1
Tipo de dato Java
identificador
Parametro 2
Tipo de dato Java
identificador
...
Parametro N
Tipo de dato Java
identificador
```

Fig. 6. Estructura del fichero

ParametrosEntrada.txt

El componente CFG_ANTLRExt recibe información de los parámetros de entradas de dominio objeto y conjunto, para la transformación de las variables de dominios de entrada de datos complejos (objeto y conjunto) en variables de dominios de entrada de datos simples (lógico, cadena y numérico), a través de los ficheros ParametrosObjetos.txt y ParametrosConjuntos.txt.
El componente CFG_ANTLRGeneratorExt genera el fichero CodigoCopiaB.txt, para ser procesado por el resto de los componentes de MTest.search.

Componentes de extensión del modelo de reducción basado en búsqueda

A partir de los mecanismos propuestos para extender el modelo de reducción basado en búsquedas, se implementaron los componentes GeVaUExt, GeCaPExt y GeneracionCasoPruebaServicio (Figura 1), para incorporar los criterios de significación de los valores y de los caminos o escenarios, durante la generación de las suites de pruebas. Para ello se ha implementado el componente GeVaUExt que extiende al componente GeVaU. En el grupo de investigación se definieron criterios de significación de los valores para las variables de dominios: numérico, cadena, enumerado, fecha, lógico y conjunto.

La significación es un valor numérico en el intervalo [0,1], depende de las técnicas utilizadas para generar los valores iniciales, como por ejemplo la técnica de partición de equivalencia y valores límites, que contiene un conjunto de clases de equivalencia definidas. La significación de cada clase de equivalencia depende de las transformaciones que se les hace a los valores límites según el dominio de la variable. Mientras más cercano a 0, el valor de las variables es menos significativo, de lo contrario si está más cerca a 1 mayor significación tiene. Para los valores propuestos por el usuario la significación es 1. En la tabla 1 se refleja la significación de los valores utilizando las técnicas de partición de equivalencia y valores límites para las variables numéricas.

Tabla 1. Significación de valores utilizando técnicas de partición de equivalencia y valores límites para las variables numéricas

Dominio	Clases de equivalencia	Transformación	Significación
Numérico	11	V = (MaxIntervalo(y) – MinIntervalo(y))/2	0.5
	12	V = add(MinIntervalo(y), -1	0.8
	12	V = add(MaxIntervalo(y), 1)	0.8
	15	V = MinIntervalo(y)	1.0
	15	V = MaxIntervalo(y)	1.0

El componente GeVaUExt recibe tres ficheros: VariablesValores.txt, Condiciones.txt y Caminos.txt, que permite la generación de las combinaciones de los valores que cubren un camino o escenario determinado según las condiciones que presente. El fichero VariablesValores.txt contiene las variables de entrada, los valores de cada una y su significación, como se observa en la figura 7.

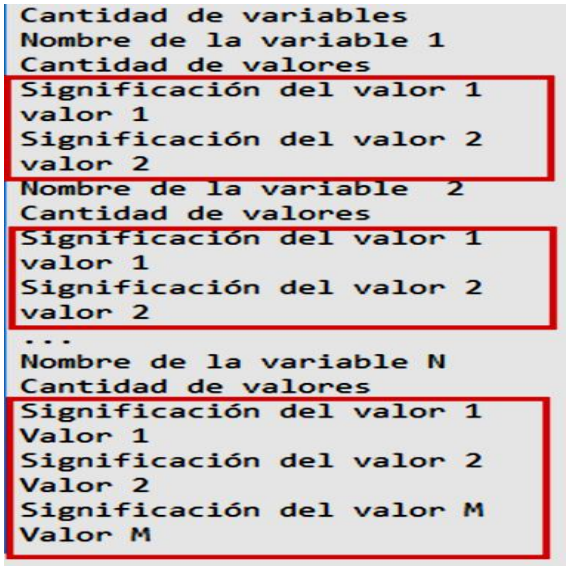

Fig. 7. Estructura del fichero VariablesValores.txt

En esta extensión se definió que la significación de cada camino (valor numérico en el intervalo [0, 1]) dependerá de la cantidad de condiciones contenidas, es el resultado de dividir la cantidad de

condiciones en el camino entre la cantidad total de condiciones. En la Figura 8 se muestra la estructura del fichero Caminos.txt que contiene la información de los caminos independientes.

```
Cantidad de caminos
Cantidad de Condiciones
Significacion del camino 1
Camino 1
Significación del camino 2
Camino 2
...
Significacion del camino N
Camino N
```

Fig. 8. Estructura del fichero Caminos.txt

Para poder generar la suite de prueba se hizo necesario implementar el componente GeCaPExt encargado de crear los ficheros de entrada del GeVaUExt. A partir del análisis de la significación de los valores y de los caminos se generan los casos de pruebas, dándole prioridad a los caminos con más condiciones y empleando valores más significativos para detectar mayor cantidad de errores.

III. RESULTADOS

Para evaluar el valor práctico de la generación del código de prueba por los componentes desarrollados, se definieron dos casos de estudio como mecanismo de validación, a partir de dos métodos (*triangleClasification* y *getSamplesAsBytes*) de proyectos reales. El método *triangleClasification* es una implementación en *Java* del algoritmo clásico del problema de clasificación de triángulos según las longitudes de sus lados, mientras, getSamplesAsBytes pertenece a la biblioteca de proyectos *Apache Commons JXPath*. En las figuras 9 y 10 se muestran los códigos correspondientes a los métodos, que se sometieron al proceso de generación de código de prueba con la solución propuesta, en los casos de estudio.

```
protected boolean getSamplesAsBytes(BitInputStream bis, ArrayList<Integer> result,
        boolean predict=false;
        for (int i = 0; i < bitsPerSample.size(); i++){
            int bits = bitsPerSample.get(i);
            int sample = bis.readBits(bits);
            if (bits < 8){
                int sign = sample*15;
                sample = sample + (8 - bits); // scale to byte.
                if (sign > 0){
                    sample = sample / ((12* (8 - bits)) - 1); // extend to byte
                }
            }
            else{
               if (bits > 8){
                  sample = sample *(bits - 8); // extend to byte.
                  predict=true;
               }
            }
            sample=result.get(i);

        }
        return predict;
    }
```

Fig. 9. Fragmento del código fuente del método getSamplesAsBytes en lenguaje Java

```
public String triangleClassification(Triangle triangle) {
    String result = "Los lados " + side1 + ", " + side2 + " y " + side3 + " no forman un triángulo.";
    if (triangle.getSide1() > 0 && triangle.getSide2() > 0 && triangle.getSide3() > 0) {
        if (triangle.getSide1() < (triangle.getSide1() + triangle.getSide2()+ triangle.getSide3())/2
            if (triangle.getSide1()== triangle.getSide2()) {
                if (triangle.getSide2() == triangle.getSide3()) {
                    result = "El triángulo es equilátero.";
                }
                else {
                    result = "El triángulo es isósceles.";
                }
            }
            else {
                if (triangle.getSide1() == triangle.getSide3()) {
                    result = "El triángulo es isósceles.";
                }
                else {
                    if (triangle.getSide1() == triangle.getSide3()) {
                        result = "El triángulo es isósceles.";
                    }
                    else {
                        result = "El triángulo es escaleno.";
                    }
                }
            }
        }
    }
    return result;
}
```

Fig. 10. Fragmento del código fuente del método triangleClasification en lenguaje Java

En las figuras 11 y 12 se muestran fragmentos del código de prueba JUnit generado por los componentes desarrollados, para los métodos Java de los casos de estudio.

```
import org.junit.Assert;
import org.junit.Test;
import Todos los métodos.Métodos de prueba;

public class TestMétodos de prueba  {

   @Test
   public void test_getSamplesAsByte_CP1() {
       Métodos de prueba métodos de prueba = new Métodos de prueba();
       String expected = "true";
       ArrayList<Integer> result= new ArrayList<Integer>();
       result.add(80);
       result.add(0);
       result.add(100);
       ArrayList<Integer> bitsPerSample= new ArrayList<Integer>();
       bitsPerSample.add(8);
       bitsPerSample.add(9);
       Assert.assertEquals(expected, métodos de prueba.getSamplesAsByte1(new BitInputStream(15,36),result,bitsPerSample));
   }
```

Fig. 11. Fragmento de la clase de prueba *JUnit* generada por *TestCodeGenerationExt* para probar el método *getSamplesAsByte*

```
package pruebas_unitarias;

import org.junit.Assert;
import org.junit.Test;
import Todos los métodos.Métodos de prueba;

public class TestMétodos de prueba  {

   //Este mÃ©todo de prueba hace referencia al camino C1: side1>0, F -> side2>0, - -> side3>0, - -> side1<
   @Test
   public void test_triangleClassification_CP1() {
       Métodos de prueba métodos de prueba = new Métodos de prueba();
       String expected = "55";
       Assert.assertEquals(expected, métodos de prueba.triangleClassification(new Triangle(0, 1, 22)));
   }
```

Fig. 12. Fragmento de la clase de prueba *JUnit* generada por TestCodeGenerationExt para el método *triangleClasification*

Para validar la extensión del modelo de reducción basado en búsquedas se empleó el método *reservarProducto*, perteneciente a un proyecto de comercio electrónico, en él intervienen 5 variables de interés (tres de dominio numérico: *cantidadMinima*, *cantidadMaxima*, *cantidadReal, cantidadAReservar;* y una de dominio cadena (caracteres): *idProducto*). Este método se encarga de verificar la disponibilidad del producto que se desea reservar. En la Figura 13A se muestra un fragmento del fichero VariablesValores.txt con los valores generados para la variable *cantidadMinima* y la significación asignada a cada valor, en correspondencia con la técnica empleada. A partir de la estrategia definida para asignar el valor de significación para cada camino, en la Figura 13B se muestra el fichero Caminos.txt que recoge la estructura y la significación de cada camino del método bajo prueba y en la Figura 13C se muestran las condiciones.

A.
```
CantidadMinima
7
0.8
-1
0.5
500
1.0
0
0.8
1
0.8
9999
0.8
10001
1
10000
```

B.
```
7
7
0.14285714285714285
T - - - - - -
0.2857142857142857
F T - - - - -
0.42857142857142855
F F T - - - -
0.5714285714285714
F F F T - - -
0.7142857142857143
F F F F T - -
0.7142857142857143
F F F F - T -
0.7142857142857143
F F F F - - T
```

C.
```
7
CantidadSolicitud<0
CantidadSolicitud>99999
CantidadReal<0
CantidadReal>99999
(CantidadSolicitud<=CantidadReal)
CantidadSolicitud>CantidadReal
CantidadSolicitud<=0
```

Fig. 13. A: Fragmento del fichero VariablesValores.txt, B: Fichero Caminos.txt, C: Fichero Condiciones.txt del método *reservarProducto*.

IV. DISCUSIÓN

Generación de código de prueba *JUnit* para el método *getSamplesAsBytes*

Para validar la transformación del código fuente en Java con variables de tipo conjunto, a un código copia con variables simples, se utilizó el método *getSamblesAsBytes*, cuyo código fuente se muestra en la Figura 7. Este método tiene dos conjuntos (*result* y *bitsPerSample*) y un objeto (*bis*) como parámetros de entrada.

A partir de la descripción de los objetos y conjuntos dada por el usuario, se modifica el código fuente original transformando las variables complejas en variables simples.

A partir del código fuente original modificado, TestCodeGeneration genera los métodos de pruebas. A partir de estos métodos de pruebas, TestCodeGenerationExt obtiene la clase JUnit de la Figura 9. Cada método de prueba se modifica en correspondencia con las especificaciones dadas por el usuario. Se crea dentro del método de prueba una instancia de *BitInputStream* y dos listas. Además, se van insertando los elementos a cada conjunto en correspondencia con los datos de pruebas generados. Antes de ejecutar el código de prueba generado, el programador tendrá que importar la biblioteca java.util para garantizar el trabajo con colecciones de datos, y la clase BitInputStream, haciendo referencia al paquete en que se encuentra dentro del proyecto bajo prueba.

Generación de código de prueba JUnit para el método *triangleClasification*

La implementación del método de clasificación de triángulos seleccionada recibe como parámetro un objeto de tipo *Triangle* que tiene como atributos las longitudes de sus lados y se accede a ellos a través de llamadas. Por tal motivo ha sido escogido para mostrar la generación de código de pruebas unitarias correspondiente a tipo de datos objetos de la figura 8.

Se detectaron tres llamadas a métodos, pero el usuario especifica que hacen referencia a los atributos del objeto *triangle*. En correspondencia con lo anterior, se genera CodigoCopiaB.txt con los atributos de *triangle* como parámetros de entrada y las llamadas sustituidas por los atributos respectivos.

TestCodeGeneratioExt procesa los métodos de pruebas generados por *TestCodeGeneration* y crea una instancia de *Triangle*, para pasársela como parámetro al método original, con los valores de pruebas generados. En la Figura 10 se muestra el código de pruebas *JUnit* generado por *TestCodeGenerationExt*, que es el que realmente se devuelve al usuario para su ejecución en *JUnit*.

Tratamiento de la significación de valores y caminos del método *reservarProducto*

El método posee 5 variables numéricas, 7 caminos independientes (Ver Figura y 7 condiciones, donde las cuatro primeras comprueban si los valores se encuentran entre 0 y 99 999 y las últimas verifican si la solicitud se puede reservar satisfactoriamente. GeVaUExt incorporó la significación de los valores en función de las técnicas aplicadas para generar los valores. En el caso del valor 0, que constituye el límite inferior del intervalo, tiene el máximo valor de significación. El componente GeCaPExt incorpora en el fichero Caminos.txt la significación de cada camino, en función de la cantidad de condiciones en el caso del primer camino la significación tiene valor 1/7, debido a que tiene una sola condición.

Los mecanismos de extensión propuestos permiten la generación de suites de pruebas reducidas a partir de criterios de significación de atributos y caminos/escenarios incluidos en el modelo de optimización de MTest.search. Las suites de pruebas generadas contienen un grupo de casos de pruebas que contribuyen a detectar defectos o fallas, sin necesidad de generar una cantidad elevada de casos de pruebas. Los componentes de MTest.search se pueden extender de forma independiente, sin tener que modificar el resto. La posibilidad de extensión e integración de estos componentes en diversos entornos de desarrollo, contribuye a elevar la eficiencia del proceso de aseguramiento de la calidad del producto de software en las empresas desarrolladoras de softwares.

En trabajos futuros se debe evaluar la efectividad de las suites de pruebas generadas para proyectos reales incluidos en bibliotecas internacionales, así como definir nuevos mecanismos para MTest.search, que permitan independizar la prueba unitaria de un método, esta propuesta podría basarse en la generación de objetos simulados.

V. CONCLUSIONES

1. El presente trabajo facilita el proceso de pruebas durante el desarrollo de productos de software, contribuyendo al proceso de automatización de la etapa de diseño de casos de pruebas.
2. Los mecanismos propuestos para extender el modelo de reducción basado en búsquedas permiten a los desarrolladores, reutilizar los componentes desarrollados con anterioridad, e incorpora solamente en las extensiones los nuevos criterios de análisis a tener en cuenta para la generación de suites de prueba.
3. Muestra de ello, es la implementación de componentes de extensión que introducen la significación de valores y caminos/escenarios de la unidad específica a probar, a partir de criterios definidos.
4. Se desarrollaron componentes que garantizan el tratamiento de las variables de tipos de datos complejos objetos y conjuntos y pueden ser desplegadas en entornos productivos para tributar a la calidad del producto de software.

VI. REFERENCIAS

1. Delgado MD, Macias A, Et Al. Model for Automatic Generation of Search-Based Early Test. Computación y Sistemas. 2017;21(3):503-13.ISSN 1405-5546.
2. Demiroz G. Cost aware combinatorial interaction testing2015 20 de septiembre del 2021.[Fecha de consulta:20 de septiembre del 2021] Disponible en: www.ista2015.cs.uoregon.edu.
3. Felbinger H., Wotawa, F., & Nica, M. Adapting unit test by generating combinatorial test data". In 2018, April IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2018. Sweden: IEEE, 352-355. ISBN 9781538663530
4. Fernández PB, Et al. Generación de combinaciones de valores de pruebas utilizando metaheurística. Ingeniería Industrial. 2016;37(2):200-7.ISSN 1815-5936
5. Fraser G, et al. EvoSuite at the SBST 2017 tool competition. En: SBST '17 Proceedings of the 10th International Workshop on Search-Based Software Testing, Buenos Aires, Argentina, IEEE Press Piscataway, 2017, pp. 39-41. ISBN 9781538627891.
6. Gao, R., et al. Effective Test Generation for Combinatorial Decision Coverage. En: 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Vienna, Austria IEEE, 2016, pp. 47-54. ISBN 9781509037131
7. Gurbuz, Hg, Tekinerdogan B. Model-based testing for software safety: a systematic mapping study. Software Quality Journal, 2018; 26(4): 1327-1372. ISSN 1573-1367.
8. Güemes A. M., Dapena M. D. D., Uribazo, D. L., Implementación de estructuras gramaticales en la herramienta ANTLR (ANother Tool for Language Recognition) con vistas a la obtención del grafo de control de flujo, In 19 Convención Científica de Ingeniería y Arquitectura, 2018.ISBN 978-959-261-585-4.
9. Güemes A M. Componentes para la generación de código de prueba JUnit a partir de código fuente Java. Tesis de Grado. La Habana: Universidad Tecnológica de La Habana José Antonio Echeverría, CUJAE, 2020.

10. Harikarthik S. K., Palanisamy V., & Ramanathan, P. Optimal test suite selection in regression testing with test case prioritization using modified Ann and Whale optimization algorithm. Cluster Computing, 2019; 22(5). ISSN 11425-11434.
11. Hauptmann B, et al. An expert based cost estimation model for system test execution. En: ICSSP 2014 Proceedings of the 2014 International Conference on Software and System Process, Najing, China, ACM, 2014, pp. 159-163. 978-14503-2754-1.
12. HERMADI I., et al. Dynamic stopping criteria for search based test data generation for path testing. Information and Software Technology, 2014; 56(4):395-407. ISSN 0950-5849.
13. IEEE. ISO/IEC/IEEE 29119-1, Software and systems engineering - Software testing. Part 1: Concepts and definitions. New York, USA, IEEE, 2013a. .[Fecha de consulta:20 de septiembre del 2021] Disponible en:www.iso.org
14. IEEE. ISO/IEC/IEEE 29119-2, Software and systems engineering - Software testing - Part 2: Test processes. New York, USA, ISO/IEC/IEEE, 2013b .[Fecha de consulta:20 de septiembre del 2021] Disponible en:www.iso.org
15. Khari, M. Empirical Evaluation of Automated Test Suite Generation and Optimization. Research Article – Sècial Issue – Intelligent Computing and Interdisciplinary Applications, 2019 .[Fecha de consulta:20 de septiembre del 2021] Disponible en:www.semanticscholar.org
16. Khari, M., et al. Performance analysis of six metaheuristic algorithms over automated test suite generation for path coverage-based optimization. Methodologies and Application, 2019. [Fecha de consulta:20 de septiembre del 2021] Disponible en:www.researchgate.net
17. Khurana, N. And R. S. Chhillar. A comparison of evolutionary techniques for test case generation and optimization. Journal of Theoretical and Applied Information Technology, 2017; 95(19):5285-5295. ISSN 1817-3195.
18. Larrosa D, et al. GeCaP: Unit Testing Case Generation from Java Source Code. Polibits, 2018, (57): 67-73. ISSN 2395-8618.
19. Larrosa D. Extensiones a MTest.search para la generación automática de pruebas unitarias en diferentes lenguajes. Tesis de Maestría. La Habana, Cuba, Universidad Tecnológica de La Habana José Antonio Echeverría, Cujae, 2019.
20. LOOR, J. M. Priorización de casos de pruebas en entornos de desarrollo ágiles. Tesis de maestría. Universidad Tecnológica de La Habana José Antonio Echeverría, CUJAE, 2019.
21. Macias, A. Componente para generar valores de casos de prueba funcionales maximizando la cobertura de los escenarios (GeVaF). Tesis de Diploma. La Habana, Cuba, Universidad Tecnológica de La Habana José Antonio Echeverría, 2016.
22. Macías, A.; et al. Generador de valores de casos de pruebas funcionales. Lámpsakos, 2016, (15): 51-58. ISSN 2145-4086.
23. Martin, R. C. Código Limpio. Manual de estilo para el desarrollo ágil de software. Spain: Ediciones Anaya Multimedia, 2012. ISBN 9788441532106
24. MICROSOFT. Live Unit Testing con Visual Studio 2017. .[Fecha de consulta: 6 de noviembre del 2018] Disponible en: https://docs.microsoft.com/es-es/visualstudio/test/live-unit-testing?view=vs-2017.
25. Myers, G. J, et al. The art of software testing. 3rd. New Jersey, USA: John Wiley & Sons, Inc., 2012. 978-1-118-03196-4.
26. Oliva P. F., Terrero W. C., Dapena M. D. D., Suarez A. R., Márquez C. Y. Generación De Combinaciones De Valores de pruebas utilizando metaheurística. Ingeniería Industrial. 2016. 200-207. ISSN 1815-5936.
27. PARASOFT. Parasoft C/C++test, 2019. [Fecha de consulta: 24 de enero del 2019]. Disponible en: https://www.parasoft.com/products/ctest.
28. PARASOFT. Parasoft JTest, 2019. [Fecha de consulta: 24 de enero del 2019]. Disponible en 4: http://www.parasoft.com/products/jtest.
29. Parr, T. The Definitive ANTLR Reference: Building Domain-Specific Languages. Raleigh, North Carolina: The Pragmatic Bookshelf, 2007. ISBN 978-09787392-4-9.
30. Pressman, R. S. Software Engineering. A Practitioner's Approach. 8th. New York, USA: McGraw-Hill Education, 2015. ISBN 978-0-07-802212-8.

31. Rojas, D. M., & Pérez, Z. Componente para la generación automática de valores interesantes de pruebas unitarias utilizando las técnicas de bucles y condiciones. Tesis de Diploma. Universidad Tecnológica de La Habana José Antonio Echeverría, CUJAE, 2016.
32. Sakti, A. et al. JTeXpert at the SBST 2017 tool competition. En: SBST '17 Proceedings of the 10th International Workshop on Search-Based Software Testing, Buenos Aires, Argentina, IEEE Press Piscataway, 2017, pp. 43-46. ISBN 9781538627891.
33. Serna, E., Martínez, R., Tamayo, P., & De Envigado, I. U. Una revisión a la realidad de la automatización de las pruebas del software. Computación y Sistemas, 2019, 23(1), 169-183. ISSN 2007-9737.
34. Silva B. C. F., Carvalho G., Sampaio A. CPN simulation-based test case generation from controlled natural-language requirements. Science of Computer Programming, 2019; (181): 111-139. ISSN 0167-6423.
35. Ibazo D., Dapena M., Güemes A. GeCaP: Herramienta para la generación de código de pruebas integrada en entorno productivos. In CIbSE, 2019, 649-656. .[Fecha de consulta: 6 de noviembre del 2020] Disponible en:http://cibseconference.org