

Herramienta para la Prueba de Mutaciones en el Lenguaje C++

Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Juan José Domínguez-Jiménez

Grupo UCASE de Ingeniería del Software, Departamento de Ingeniería Informática,
Universidad de Cádiz, Cádiz, España

{pedro.delgado,inmaculada.medina,juanjose.dominguez}@uca.es

Resumen La prueba de mutaciones es una técnica basada en fallos en torno a la cual se han elaborado herramientas para un amplio abanico de lenguajes de programación. Sin embargo, el desarrollo de un marco de prueba de mutaciones no comercial para C++ estaba pendiente. En este artículo se presenta una herramienta que permite analizar código C++, generar mutantes y ejecutar un conjunto de casos de prueba para obtener resultados que nos permitan determinar su efectividad en la detección de errores en el código. La herramienta está diseñada para permitir la inclusión de nuevos operadores para cubrir cualquier característica del lenguaje. En este documento, el uso de la herramienta se muestra a través de un operador de mutación al nivel de clase.

Palabras clave: Prueba de software, prueba de mutaciones, C++

1. Introducción

La prueba de mutaciones es una técnica que trata de evaluar la efectividad de los conjuntos de casos de prueba detectando fallos insertados intencionadamente [7]. Esta técnica consiste en introducir simples cambios sintácticos en el programa original mediante los *operadores de mutación*. Cada uno de estos cambios genera un nuevo programa al que se le conoce como *mutante*. Un buen conjunto de casos de prueba para nuestro programa debe poder detectar todos los cambios que afecten a su funcionamiento.

En la actualidad existe una gran variedad de herramientas que emplean esta técnica con éxito para diferentes lenguajes [6]. No obstante, el desarrollo de la técnica en torno a C++ no ha sido apenas abordado, pudiendo encontrar algunas herramientas comerciales [2], como Insure++ o PlexTest, las cuales en general solo realizan algunas mutaciones simples y aplican la técnica de una manera selectiva. Este artículo tiene como propósito introducir a la comunidad científica un marco para la aplicación de la prueba de mutaciones al lenguaje C++. Esta herramienta automatiza las distintas fases que implica la técnica, desde el análisis del código hasta la ejecución de las pruebas para comparar el resultado de los mutantes con el programa original. La herramienta sigue un esquema de funcionamiento similar a MuBPEL para composiciones WS-BPEL [5], permitiendo ejecutar distintos comandos para cada una de estas etapas.

Respecto al conjunto de operadores, en un trabajo previo se definió un conjunto de operadores al nivel de clase para C++ [3], el cual fue posteriormente evaluado [4] y se ha implementado en la herramienta. En cualquier caso, la herramienta de mutación facilita la implementación de nuevos operadores para llevar a cabo la técnica a cualquier nivel del lenguaje. En este documento se explica su funcionamiento general (Sección 2) y se hace una demostración de su uso mediante la aplicación de un operador a nivel de clase en un programa de prueba, explorando las distintas opciones de la herramienta (Sección 3). Finalmente, se expondrán brevemente las conclusiones y el trabajo futuro.

2. Funcionamiento de la Herramienta

El marco presentado reutiliza el árbol de sintaxis abstracta (AST) que el compilador *Clang* [1] genera como representación intermedia del código. El AST representa los distintos elementos del lenguaje de manera estructurada. De esta manera, se recorre el árbol en busca de los nodos que cumplen las características prescritas en los operadores de mutación. *Clang* proporciona sus bibliotecas para realizar esta búsqueda y controlar las diversas construcciones del lenguaje [2]. A continuación, se muestra cómo la herramienta maneja las etapas de esta técnica:

- *Análisis*: El AST es recorrido en busca de localizaciones apropiadas respecto a los patrones de búsqueda creados previamente para cada operador. Se permite realizar el análisis de varios ficheros de código fuente en la misma ejecución. Dos ficheros pueden incluir una misma cabecera, cuyo código estará presente en el AST de ambos ficheros, pero la herramienta está preparada en este caso para evitar crear mutantes generados previamente.
- *Generación*: Los nodos detectados en el análisis son manipulados adecuadamente para insertar la mutación oportuna. La estructura del programa, incluyendo el sistema de compilación, es replicada en cada mutante mediante el sistema de control de versiones *Git*, lo cual permite que todo mutante pueda ser ejecutado independientemente, además de ahorrar espacio en disco al almacenar en cada rama únicamente la mutación introducida.
- *Ejecución*: La ejecución de casos de prueba no está sujeta a un marco de pruebas concreto, permitiendo al probador ejecutar un conjunto de pruebas ya implementado, siempre que se informe apropiadamente al sistema de la salida que se genera. Para pruebas independientes de un marco de pruebas, como en la Sección 3, se hace uso de una biblioteca creada a tal fin.

3. Demostración de uso

La demostración del empleo del marco de prueba de mutaciones se ilustrará mediante el operador *IOD*, cuyo objetivo es eliminar métodos que están sobrescritos en clases que heredan [3]. A continuación, se muestra la salida de las opciones principales incluidas en la herramienta al aplicar el operador *IOD* en el programa “ejemplo.cpp” (Fig. 1), que cuenta con tres métodos sobrescritos. Para este ejemplo, se aplican los dos casos de prueba mostrados en la Fig. 2:

```

1 class Vehiculo{           class Coche:           class Biplaza:
2                             public Vehiculo{         public Coche{
3 public:                     public:                 public:
4     string tipo(){          string tipo(){         string tipo(){
5         return "genérico";   return "coche";       return "biplaza";
6     }                       }                       }
7     string matricula(){     int plazas(){         int plazas(){
8         return mat;         return 5;             return 2;
9     }                       }                       }
10    string mat;             };                     };
11 };
    
```

Fig. 1. Código fuente de “ejemplo.cpp”. Resaltados los métodos sobrescritos.

```

TestFunctions tf; /*Biblioteca para informar de los resultados*/
/* Prueba 1 */    | /* Prueba 2 */
Coche c;         | Biplaza b;
tf.save_result(c.tipo()=="coche"); | tf.save_result(b.plazas()==2);
    
```

Fig. 2. Dos casos de prueba para “ejemplo.cpp”

- Analyze:** Muestra un informe de los mutantes por cada operador. El formato de la salida es “Operador Localizaciones Variantes”, siendo “localizaciones” el número de lugares donde insertar el fallo, y “variantes” las distintas mutaciones que se pueden insertar por cada localización:

> **tool analyze ejemplo.cpp**

IOD 3 1 (Nota: Variante=1 → única acción es ‘eliminar’)

La herramienta evita la creación de un mutante duplicado [2], ya que desde la clase “Biplaza” se hereda el método “tipo” desde las clases “Coche” y “Vehiculo”.

- Applyall:** Esta opción genera todos los mutantes posibles en orden según se encuentran en el código. El nombre sigue el formato “m” + “código_localización_variante_fichero”, siendo “código” un número identificador del operador (ejemplo: IOD = “01”):

> **tool applyall ejemplo.cpp**

Mutantes: m01_1_1_ejemplo, m01_2_1_ejemplo, m01_3_1_ejemplo

- “m01_1_1_ejemplo” eliminará las líneas 4-6 de la clase “Coche”.
- “m01_2_1_ejemplo” eliminará las líneas 4-6 de “Biplaza”.
- “m01_3_1_ejemplo” eliminará las líneas 7-9 también de “Biplaza”.

Nota: También existe la opción *apply* para generar solo un mutante concreto.

- Compare:** Este comando lleva a cabo la comparación del resultado de la ejecución de los mutantes con respecto al programa original:

```
> tool compare m01_1_1_ejemplo m01_2_1_ejemplo  
               m01_3_1_ejemplo
```

	Prueba 1	Prueba 2	
<i>m01_1_1_ejemplo</i>	1	0	Nota:
<i>m01_2_1_ejemplo</i>	0	0	* 0: Misma salida.
<i>m01_3_1_ejemplo</i>	0	1	* 1: Distinta salida.

Los mutantes “m01_1_1_ejemplo” y “m01_3_1_ejemplo” se dice que han sido “matados” por la “Prueba 1” y la “Prueba 2” respectivamente, al tener una salida distinta al programa original. Para detectar el fallo presente en el mutante “m01_2_1_ejemplo” será necesario un nuevo caso de prueba.

4. Conclusiones y Trabajo Futuro

Este artículo presenta un marco para aplicar la prueba de mutaciones a programas escritos en C++, el cual servirá para poder obtener resultados sobre la efectividad de esta técnica para este lenguaje. Esta herramienta es una importante contribución al abordar la mutación del lenguaje de una manera general y robusta, para así poder investigar sobre la técnica en torno a este lenguaje.

Como trabajo futuro, se pretende seguir refinando la implementación de los operadores y del sistema en general, así como añadir nuevos operadores, mejoras y funcionalidades a la herramienta a fin de conseguir nuevos resultados de investigación sobre la prueba de mutaciones.

Agradecimientos: Este trabajo fue financiado por la beca de investigación PU-EPF-FPI-PPI-BC 2012-037 de la Universidad de Cádiz.

Referencias

1. Clang: a C language family frontend for LLVM, <http://clang.llvm.org>
2. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J.: Generación de mutantes válidos en el lenguaje de programación C++. In: XIX Jornadas de Ingeniería del Software y Bases de Datos. Cádiz, Spain (2014)
3. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J., García-Domínguez, A.: Operadores de mutación a nivel de clase para el lenguaje C++. In: XII Jornadas sobre Programación y Lenguajes, PROLE 2013. Madrid, Spain (2013)
4. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J., García-Domínguez, A., Palomo-Lozano, F.: Class mutation operators for C++ object-oriented systems. *Annals of telecommunications* 70(3-4), 137–148 (2015)
5. García-Domínguez, A., Estero-Botaro, A., Medina-Bulo, I., Palomo-Lozano, F.: Herramienta de mutación firme para WS-BPEL 2.0. In: Actas de las XVII Jornadas de Ingeniería del Software y Bases de Datos. pp. 415–418 (2012)
6. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on* 37(5), 649–678 (Oct 2011)
7. Woodward, M.R.: Mutation testing - its origin and evolution. *Information and Software Technology* 35(3), 163–169 (Mar 1993)