Propuesta de implementación de operadores de mutación de rendimiento en MuCPP

Luis Acuña-Vega, Inmaculada Medina-Bulo, Juan Jose Dominguez-Jimenez, Pedro Delgado-Pérez

Grupo UCASE de Ingeniería del Software, Departmento de Ingeniería Informática, Escuela Superior de Ingeniería, Universidad de Cádiz, Avda. de la Universidad de Cadiz 10, 11519 Puerto Real, Cádiz. luis.acunavega@alum.uca.es, {inmaculada.medina,juanjose.dominguez,pedro.delgado}@uca.es

Resumen Las prueba de mutaciones de rendimiento es una idea bastante nueva y con desafíos abiertos. En este artículo, precisamente, se aborda la aplicación de la prueba de mutaciones para la detección de problemas de rendimiento en los programas escritos en C++, mediante la extensión del sistema de prueba de mutaciones MuCPP con nuevos operadores de mutación que generen mutantes que simulen errores de rendimiento. En concreto, este artículo se centra en el estudio de estos operadores de mutación propuestos en trabajos previos, presentándose la implementación de uno de ellos.[1]

Keywords: Prueba de mutaciones, Pruebas de rendimiento, C++

1. Introducción

Debido a la necesaria evolución hacia una mayor automatización de las pruebas de software, es necesario incorporar técnicas que permitan aumentar la confianza en nuestro sistema respecto a las técnicas de prueba que se emplean tradicionalmente en la industria. Una de las técnicas más potentes en este sentido es la prueba de mutaciones [5]. La prueba de mutaciones explota fallos artificiales para medir la idoneidad de los conjuntos de prueba y orientar su mejora.

Aunque la prueba de mutaciones ha sido empleada mayoritariamente para identificar debilidades en pruebas a nivel funcional, también ha sido extrapolada a otras actividades relacionadas con la ingeniería del software [4]. Sin embargo, la aplicación de mutaciones relacionadas con propiedades no funcionales del software como el rendimiento es actualmente muy limitada. Este hecho es sorprendente teniendo en cuenta la frecuencia de este tipo de problemas [9], el impacto que provocan y lo difícil que suele ser detectarlos y subsanarlos [6].

Los defectos de rendimiento del software pueden causar una importante degradación en la experiencia de usuario y dar lugar a problemas muy costosos de

[1] Este trabajo ha sido financiado por la Comisión Europera (FEDER) y el Ministerio de Ciencia, Innovacion y de Universidades bajo el proyecto FAME (RTI2018-093608-B-C33) y la Red de Investigación en Ingeniería de Software Basada en Búsqueda (SEBASENet 2.0) (RED2018-102472-T).

detectar y resolver. Las pruebas de rendimiento persiguen detectar estos defectos y reducir su impacto. Sin embargo, los mecanismos para evaluar la calidad de las pruebas de rendimiento son actualmente muy limitados causando, en muchos casos, que estos problemas pasen desapercibidos.

En este artículo precisamente se aborda la aplicación de la prueba de mutaciones para la detección de problemas de rendimiento en los programas escritos en C++, mediante la extensión del sistema de prueba de mutaciones MuCPP [2] con nuevos operadores de mutación que generen mutantes que simulen errores de rendimiento. En concreto, este artículo se centra en el estudio de estos operadores de mutación propuestos en trabajos previos [3,7], presentándose la implementación de uno de ellos.

La estructura de este artículo es como sigue. En la sección 2, se contextualiza el estado del arte de la prueba de mutaciones de rendimiento, la herramienta a extender y se informa brevemente sobre los artículos que sustentan este trabajo. En la sección 3 se detalla la propuesta del trabajo a partir de un operador de mutación ya implementado. Finalmente, en la sección 4, se concluye el trabajo, dando a conocer lo avanzado y los trabajos futuros.

2. Estado del arte

La prueba de mutaciones consiste en la introducción controlada de modificaciones en el código que se asemejan a fallos reales [8]. Estos fallos se inyectan basándose en reglas de transformación predefinidas, conocidas como operadores de mutación [1], que simulan errores de programación comunes y que tradicionalmente provocan un cambio en la funcionalidad del software. De este modo, la salida del programa original difiere de la del programa mutado. La definición del conjunto de operadores de mutación es clave para el éxito de la técnica.

Por otro lado, la prueba de mutaciones de rendimiento [7] se basa en introducir fallos sobre la aplicación a probar, es decir, realiza cambios en el código que degraden el rendimiento del programa sin realizar cambios en su funcionalidad. La prueba de mutaciones de rendimiento está diseñada para características no funcionales, como memoria o tiempo de ejecución, y buscan generar mutantes que, con la misma funcionalidad, provoquen un cambio en las propiedades no funcionales, es decir, que la medición de estas propiedades no funcionales cuando se ejecuten las pruebas sea distinta en el programa original (salida esperada) y el mutante. Estos operadores fueron planteados inicialmente en [7] y se aplicaron de forma manual en [3]. En este trabajo se implementa uno de los operadores automatizando la generación de sus mutantes.

Desde su origen, se han desarrollado más de un centenar de herramientas que implementan diferentes operadores y que cubren la mayoría de los lenguajes de programación más utilizados. Así, tenemos para lenguajes como Fortran o C, utilizando operadores de mutación tradicionales o estándar, dando lugar a herramientas como MuCPP [2] para el lenguaje C++. Es precisamente esta herramienta MuCPP la empleada en este trabajo para la introducción de operadores de mutación de rendimiento.

3. Implementación del operador de mutación RCL

Para la aplicación de la prueba de mutaciones de rendimiento como mecanismo para evaluar y mejorar la solidez de las pruebas de rendimiento se extenderá la funcionalidad del sistema de mutación MuCPP para el lenguaje C++. MuCPP se basa en el análisis del árbol de sintaxis abstracta (AST) para la detección de localizaciones de mutación en el código. Este artículo presenta la implementación del operador de mutación "RCL: Removal of Stop Condition in Loop", uno de los operadores de mutación de rendimiento genéricos definidos en [3]. RCL se refiere al operador que elimina una de las condiciones de parada del bucle para que siga iterando hasta que otra de las condiciones se satisfaga. Esta eliminación se puede realizar de tres formas: eliminando una sentencia `break`, retrasando una sentencia `return` que se encuentre en el cuerpo del bucle para situarlo después del bucle, o eliminando, en caso de que la condición del bucle esté compuesta por varias unidas por `&&`, una de ellas. El código que aparece en la figura 1 muestra precisamente la aplicación del operador RCL en este caso, eliminando la condición asociada a la variable `testbool`.

Código original:

```
Foo o1;
bool testbool=false;
for(int i=0; i < n && !testbool; ++i){
    o1.recalculate(i);
    if (o1.get_value() == 15)
        testbool = true;
}
```

Código mutante:

```
Foo o1;
bool testbool=false;
for(int i=0; i < n /*RCL*/ ; ++i){
    o1.recalculate(i)
    if (o1.get_value() == 15)
        testbool = true;
}
```

Figura 1. Ejemplo de aplicación del operador RCL.

```
forStmt(
hasCondition(
    binaryOperator(
            anyOf(hasOperatorName("&&"), hasOperatorName("and")),
                hasRHS(unaryOperator(anyOf(
                    hasOperatorName("!"), hasOperatorName("not"))
            ))
        ).bind("RCLcondR")
    )
).bind("RCL");
```

Figura 2. Sección de código del patrón que detecta localizaciones del operador RCL.

La figura 2 muestra un fragmento del patrón del operador RCL, el cual sirve para detectar la localización de mutación en el código que se presenta en la figura 1 a partir del AST que este código genera. En concreto, el patrón busca detectar

un bucle "for" (`forStmt`) que contenga una condición de parada (`hasCondition`) con un operador lógico AND (`binaryOperator`) y una variable negada en su lado derecho (`hasRHS`), donde los (`bind`) permiten localizar posiciones. Aplicando este patrón al código original se elimina la parte derecha de la condición del bucle, sustituyéndose por el comentario, logrando generar el mutante de la figura 1.

El patrón de figura 2 se ha extendido para detectar distintas combinaciones de posiciones de la variable, si está a la izquierda o la derecha, si esta está negada o no, o si el ciclo es una sentencia "while".

4. Conclusiones y trabajo futuro

Se ha realizado la implementación y puesta en práctica de un operador de mutación de rendimiento (RCL) para bucles "for" como "while" extendiendo la funcionalidad de la herramienta MuCPP, lo que demuestra la viabilidad de la implementación de este tipo de operadores. Se continuará la implementación de los demás operadores de rendimiento definidos, desarrollando esta línea de investigación. Una vez implementados los operadores, se plantea comprobar el rendimiento real en aplicaciones reales usando repositorios abiertos con bancos de prueba disponibles determinando las pruebas de rendimiento adecuadas.

Referencias

1. Ahmed, Z., Zahoor, M., Younas, I.: Mutation operators for object-oriented systems: A survey. In: 2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE). vol. 2, pp. 614–618 (2010)
2. Delgado-Pérez, P., Medina-Bulo, I., Palomo-Lozano, F., García-Domínguez, A., Domínguez-Jiménez, J.J.: Assessment of class mutation operators for C++ with the MuCPP mutation system. Information and Software Technology 81, 169–184 (2017)
3. Delgado-Pérez, P., Sánchez, A.B., Segura, S., Medina-Bulo, I.: Performance mutation testing. Software Testing, Verification and Reliability 31(5), e1728 (2021)
4. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (2011)
5. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Chapter six - mutation testing advances: An analysis and survey. Advances in Computers, vol. 112, pp. 275–378. Elsevier (2019)
6. Segura, S., Troya, J., Durán, A., Ruiz-Cortés, A.: Performance metamorphic testing: Motivation and challenges. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER). pp. 7–10 (2017)
7. Sánchez, A.B., Delgado-Pérez, P., Segura, S., Medina-Bulo, I.: Performance mutation testing: Hypothesis and open questions. Information and Software Technology 103, 159–161 (2018)
8. Woodward, M.: Mutation testing—its origin and evolution. Information and Software Technology 35(3), 163–169 (1993)
9. Zaman, S., Adams, B., Hassan, A.E.: A qualitative study on performance bugs. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR). pp. 199–208 (2012)