Análisis de beneficios y riesgos de usar la cobertura de código

Analysis of benefits and risks of using code coverage

Joan Andrés Buriticá Salazar[1]
jhoan.buritica00@usc.edu.co

Carlos Andrés Tavera Romero Ph. D.[2]
carlos.tavera00@usc.edu.co

Universidad Santiago de Cali, Facultad de Ingeniería, Programa de Ingeniería en Sistemas (1)
Universidad Santiago de Cali, Facultad de Ingeniería, Programa de Ingeniería en Sistemas (2)

Resumen

La cobertura de código es el proceso por el cual se identifica código ejecutado cuando un conjunto de casos de prueba se ejecutan. Los fallos ocasionados en programas informáticos son causados por condiciones entrantes que derivan en la ejecución de código con baja probabilidad de ejecución. Los casos de prueba aplicando la cobertura de código permite observar el comportamiento del programa informático ante una entrada de datos inválida o válida, identificando bloques de código con baja o nula probabilidad de ejecución. Existen dos aproximaciones conceptuales al aplicar la cobertura de código: instrumentación por sobrecarga fuera de línea e instrumentación por sobrecarga en tiempo de ejecución. La cobertura de código mejora la fiabilidad de los casos de prueba, permite priorizar casos de prueba y ofrece una medida cuantificable del progreso realizado en la etapa de pruebas. Un problema con la cobertura de código es la falta de una teoría que indique que tanto mejorar la fiabilidad de los casos de prueba. Una cobertura de código alta puede lograrse con el uso de la metodología TDD (Test Driven Development). La cobertura de código ofrece beneficios más allá de evaluar la eficacia de los casos de prueba, también permite identificar zonas de código con alta frecuencia de ejecución y la refactorización de software.

Palabras Clave: cobertura de código; casos de prueba; pruebas de regresión; instrumentación fuera de línea; instrumentación en tiempo de ejecución; análisis; riesgos

Abstract

Analysis of benefits and risks of using the code coverage. Code coverage is the process by which unexecuted code is identified when a set of test cases are executed. Software failures are caused by incoming conditions that result in code execution with low probability of execution. The test cases applying the code coverage allows observing the behavior of the computer program in the face of invalid or valid data entry, identifying blocks of code with low probability of execution. There are two conceptual approaches when applying code coverage, offline overload instrumentation and runtime overload instrumentation. Code coverage improves the reliability of test cases, allows prioritizing test cases, and provides a quantifiable measure of the progress made in the test stage. One problem with code coverage is the lack of a theory that indicates how much it improves test case reliability. High code coverage can be achieved with the use of TDD (Test Driven Development) methodology. Code coverage offers benefits beyond assessing the effectiveness of test cases, it also allows the identification of areas of code with high frequency of execution and software refactoring.

Keywords: code coverage; test cases; regression testing; offline instrumentation; runtime instrumentation; analysis; risks

2 INTRODUCCIÓN

Un caso de prueba es una especificación de una entrada y/o conjunto de actividades secuenciales (inválidas o validas) como argumento a una función de un programa informático que ejecuta una serie de condiciones a través del código en una etapa de pruebas y del cual se espera un resultado que define si el caso de prueba logra un objetivo en particular. Una de las características para los caso de prueba es la existencia de una precondición y una postcondición que deben de cumplirse.

La cobertura de código es el proceso por el cual "mediante métricas" se genera un reporte que identifica líneas de código, bloques de código, flujos de control, funciones y variables de un programa informático que han ejecutado una funcionalidad con la ejecución de casos de prueba. Los reportes generados por el uso de la cobertura de código, evalúan la efectividad de los casos de prueba y determinan el comportamiento del programa informático ante una entrada y/o actividad válida o inválida.

La refactorización de código, es el proceso mediante el cual se cambia la estructura interna de un programa informático pero sin afectar su comportamiento externo. El objetivo de la refactorización es mejorar el diseño, la estructura y la implementación del programa informático preservando en el proceso la funcionalidad.

La cobertura de código, además de enfocar los reportes generados en la evaluación de casos de prueba, también enfoca estos reportes en la refactorización de código, identificación de código con altas frecuencias de ejecución (Hot Spot) y depuración relacionada con el rendimiento.

Los beneficios y riesgos de la cobertura de código son variables con cada proyecto que lo utiliza, el presente trabajo ofrece una lista de beneficios percibidos y riesgos inherentes al aplicar la cobertura de código.

2.1 Planteamiento del problema

"La revisión eficaz de un programa es imprescindible para cualquier programa informático complejo" (Miller y Maloney, 1963).

Miller y Maloney (1963) en su artículo argumentaba como tesis principal que los fallos ocasionados en los programas informáticos en producción aún después de haber pasado por la etapa de pruebas, eran causados por condiciones entrantes que derivaron en la ejecución de uno o varios bloques de código que se ejecutaba en condiciones poco probables.

Concretamente, la tesis de Miller y Maloney (1963), se concentraba en los problemas que se presentaban en los programas informáticos tiempo después de estos haber entrado en operación, y que eran causados por dos razones, una manipulación incorrecta del mismo o la entrada de datos que producían la ejecución de bloques de código que no habían sido correctamente probados, es decir bloques de código que se ejecutaban con poca probabilidad.

La pregunta que surge es, ¿cómo confiar en un programa informático de cual no se tiene certeza si funcionara correctamente ante la entrada de condiciones anormales?, sin embargo, Miller y Maloney (1963) propusieron que la verdadera pregunta era, ¿se puede confiar en un programa informático para que cumpla sus especificaciones funcionales con cada nueva condición de entrada anormal?.

Los primeras aproximaciones hacia la cobertura de código fueron realizados por Rapps y Weyuker (1985) quienes propusieron emplear la técnica de análisis de flujo de datos en la etapa de pruebas (técnica que consistía en generar un entrada inválida de datos y registrar su paso por el código), permitiendo obtener una "fotografía" del comportamiento del programa ante una entrada inesperada y así identificar las partes del programa que habían sido ejercidas. Esta

técnica, tiene la desventaja de requerir que el programa sea re-escrito para aplicar esta técnica, (algunas) herramientas de cobertura de código modernas eliminan esta desventaja, pero, por otro lado, introducen nuevas.

Como cualquier procedimiento, la cobertura de código tiene beneficios percibidos y riegos presentes, las preguntas abordadas en el presente artículo, responde a dos interrogantes: ¿Cuáles son los beneficios percibidos con el empleo de la cobertura de código?, ¿Cuáles son los riesgos que surgen con el empleo de la cobertura de código?

2.2 Justificación

La cobertura de código, referida por algunos autores como cobertura estructural, Andrew (2008) y Gittens, Romanufa, Godwin y Racicot (2006), es el conjunto de métricas empleadas en los casos de prueba de un programa informático, que busca determinar de forma exhaustiva las líneas de código, los bloques de código, los flujos de control, las variables usadas y las funciones que han sido ejecutadas dentro de la estructura del programa informático.

La cobertura de código, se centra en el aspecto estructural del código y más concretamente en la ejecución del aspecto estructural. La importancia de medir la ejecución dentro del aspecto estructural del código, es permitir la evaluación de los casos de prueba, estos casos de prueba son comúnmente llamados white-box test, es decir, casos donde la persona que realiza las pruebas (tester), conoce de antemano la estructura interna del programa.

Aunque las investigaciones realizadas sobre la cobertura de código se emplean en la evaluación de los casos de prueba (porque ahí es donde reside el interrogante planteado por Miller y Maloney (1963)), también es necesario resaltar nuevas áreas de enfoque donde la cobertura de código es un recurso valioso, donde la información generada mitigaría el gasto de recursos y tiempo empleado en actividades como: la refactorización de código, la identificación de código con una alta tasa de ejecución y la depuración relacionada con el rendimiento.

Con estas nuevas áreas de enfoque, la cobertura de código ya no solamente se restringe a un solo interrogante, ahora también es empleada para dar respuesta a interrogantes como: ¿Es posible la identificación temprana de bloques de código con altas tasas de ejecución?, ¿Cómo identificar bloques de código que no aportan funcionalidad al programa?, ¿Es posible eliminar bloques de código sin que esto afecte la funcionalidad mínima del programa?

Los beneficios y riesgos de la cobertura de código se han visto en constante evolución, y es el objetivo del presente artículo dar una idea general de estos en el tiempo presente.

2.3 Estado del Arte

La cobertura de código aplicado en los casos de prueba, es el proceso por el cual se mide el porcentaje del software que es ejercido durante la ejecución de los mismos, y es aplicable tanto en la etapa de pruebas como en el uso de los pruebas unitarias (Unit Testing), pruebas de integración (Integration Testing) y pruebas del sistema (System Testing).

Para Gittens et al. (2006) y Yang, Li y Weiss (2009) la cobertura de código puede ser basada en especificaciones funcionales Black-Box Testing o en la estructura interna del programa White-Box Testing. El tratamiento de ambas modalidades se observa en los trabajos de Whalen, Rajan, Heimdahl y Mille (2006) quienes definen las métricas para la cobertura estructural (White-Box Testing) basados en las especificaciones funcionales expresadas formalmente (Black-Box Testing).

Dado que la cobertura de código basada en las especificaciones funcionales depende de la disponibilidad de las mismas, no suele ser tan usada como su contraparte, es decir, la cobertura de código basada en la estructura, que lo único en lo que depende es de la implementación.

La cobertura de código, para lograr su cometido de evaluar los casos de prueba, emplea varias métricas. Las más empleadas son:

• La métrica Block Coverage que reporta una secuencia de declaraciones ejecutadas, y cuya característica es que no hay flujo de controles internos. Es resumen, si las primeras declaraciones de un bloque son ejecutadas todas las declaraciones subsecuentes en el mismo bloque también son ejecutadas. (Véase Figura 1.)

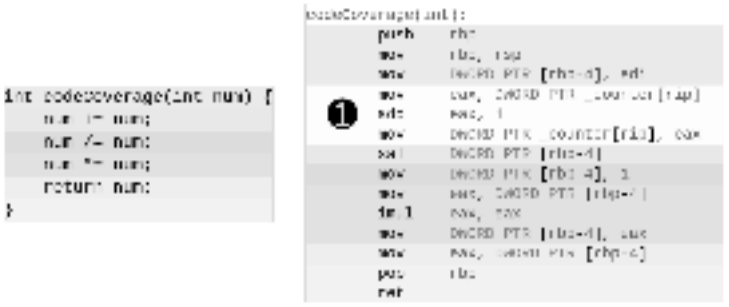

Figura 1. Instrumentación para función sin flujos de control

Las herramientas de cobertura de código optimizan las instrucciones insertadas en bloques de código donde no existen flujos de controles internos, véase la marca que identifica la única instrucción que se ha insertado para toda la función.

Fuente: Los Autores

• La métrica Statement Coverage reporta las declaraciones que han sido ejecutadas.

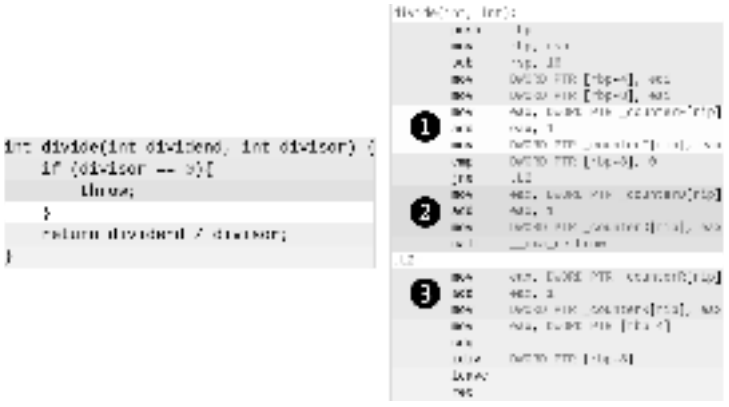

Figura 2. Instrumentación para función con flujo de control

Para bloques de código con flujos de controles internos, las herramientas de instrumentación de código han de insertar instrucciones, cuando la función es llamada (marca 1), cuando arroja una excepción (marca 2) y cuando devuelve un resultado (marca 3).

Fuente: Los Autores

Las observaciones realizadas por Gittens et al. (2002) y Piwowarski, Ohba y Caruso (1993) mostraban que la productividad de las pruebas de regresión y las pruebas de sistema (en el descubrimiento de errores) descendían una vez que se lograba alcanzar una cobertura de declaraciones del 70%.

2.3.1 Evaluación de casos de prueba

Para Yang et al. (2009) los principales motivos para aplicar la cobertura de código son: incremento de la fiabilidad de los casos de prueba, ya que permite eliminar casos redundantes que no aportan una mejora en la detección de defectos y proporcionar una medida cuantificable del progreso realizado en las pruebas relacionadas con la cobertura de código, punto además apoyado por los trabajos de Kirner (2009). Es resumen, trabajos como Gupta et al. (2015); Kirner (2009); Rothermel et at. (2001); Yang et al. (2009) concluyen que la cobertura de código permite priorizar pruebas con el fin de mejorar la tasa de detección de defectos.

Sin embargo, Cai y Lyu (2005) concluyen que una de las desventajas más obvias para la cobertura de código es la falta de información empírica que prediga qué tanto mejora la calidad de los casos de prueba con el incremento de la cobertura de código. Si bien es cierto la falta de una teoría, trabajos realizados por Gokhale y Mullen (2006) presentan una seria de técnicas que permite medir los límites prácticos de precisión que se pueden lograr con la cobertura de código. Para Yang et al. (2009) ha sido necesario resaltar que el 100% de la cobertura de código no garantiza la ausencia de errores en el programa.

Debilidades de la cobertura de código suelen ser encontradas en sus métricas individuales, Yang et al. (2009) menciona el caso de la métrica Statement Coverage, aunque fácil de usar, es insensible a condiciones, y si se opta por usar su contraparte, Path Coverage se hallará que no solamente es costosa de usar, además presenta problemas en su interpretación.

No todas las investigaciones realizadas ven con pesimismo la cobertura de código, Derezińska (2008) da puntos a favor y en contra de algunos aspectos sobre la cobertura de código, los más favorables son: incremento significativo (95.7% para funciones) de la cobertura de código con ayuda de herramientas, usadas por programadores sin experiencia y un alto porcentaje de cobertura (92.4 % para funciones) que se logra con el uso de la metodología TDD.

2.3.2 Herramientas de Cobertura de Código

Existen dos aproximaciones conceptualmente diferentes al aplicar la cobertura de código, una primera aproximación conocida como instrumentación por sobrecarga fuera de línea, realiza inserciones estáticas de instrucciones dentro del código fuente del programa anterior al proceso de compilación (está inserción de código consta de pocas líneas, que al ser ejecutadas por el programa, genera un reporte que indica que la ejecución ha pasado por ese punto), la segunda aproximación, conocida como instrumentación por sobrecarga en tiempo de ejecución, inserta y elimina instrucciones de forma dinámica. Ninguna de las dos altera el comportamiento del programa. (Gittens et al., 2006; Horváth et al., 2019; Tikir & Hollingsworth, 2002).

La aproximación de instrumentar en tiempo de ejecución permite reducir la sobrecarga en tiempo de ejecución del programa informático en un 38-90% (este porcentaje varía según la herramienta seleccionada) comparado con su contraparte de instrumentar fuera de línea. (Tikir & Hollingsworth, 2002; Yang et al., 2009).

Más allá de la diferencia presentada en el rendimiento, una ventaja que presenta la inserción dinámica frente a la inserción estática es el no requerir del código fuente para insertar las instrucciones, debido a que estas son insertadas dentro del binario o el bytecode y no en el código fuente, permitiendo instrumentar código de terceras partes (en caso de ser necesario)(Horváth et al., 2019; Yang et al., 2009).

Existe un gran trasfondo en donde son comparadas varias herramientas de análisis de código para la cobertura de código, los trabajos que comparan la instrumentación en tiempo de ejecución contra la instrumentación fuera de línea se centran en aspectos como la operabilidad, su usabilidad y las características propias de cada herramienta, ejemplos de estos trabajos son los realizados por Lingampally, Gupta y Jalote (2007); Yang et al. (2009), que pese a los resultados entregados por ambos estudios estos no dan importancia a los posibles riesgos de usar la cobertura de código.

Kajo-Mece y Tartari (2012); Li, Meng, Offutt, y Deng (2013); Tengeri, Horvath, Beszedes, Gergely y Gyimothy (2016) también hacen estas comparaciones (instrumentación fuera de línea contra instrumentación en tiempo de ejecución), los trabajos de Li et al., (2013) se centran en relación a Branch Coverage y Statement Coverage, sus conclusiones es que dado la diferencia entre ambas métricas la instrumentación fuera de línea es más apropiada para Branch Coverage, por otra parte, Kajo-Mece y Tartari (2012) evalúan ambas aproximaciones aplicadas a pequeños programas y concluyen que la instrumentación fuera de línea es más dependiente en su uso a la hora de evaluar los casos de prueba, Tengeri et al. (2016) por su parte evalúa estas dos aproximaciones en un programa de gran escala y junto con los trabajos de Horváth et al. (2019), concluye que la instrumentación en tiempo de ejecución arroja resultados bastante diferentes que su contraparte (instrumentación fuera de línea), dado las técnicas y conceptos diferentes empleadas en ambas aproximaciones.

Alemerien y Kenneth (2014), realizaron un experimento con el fin de investigar cómo los resultados arrojados por el uso de la cobertura de código con ayuda de varias herramientas son consistentes en términos de líneas, declaraciones, flujos de control y métodos. Concluyeron que la métricas flujo de control en comparación con la métrica por métodos fueron significativamente diferentes y que la métrica por declaración en comparación con la métrica por línea es ligeramente diferente. Además, encontraron que el tamaño del programa afecta de forma significativa la efectividad de las herramientas de cobertura de código.

Kessis, Ledru y Vandome (2005), investigaron la usabilidad de los análisis de la cobertura de código desde un punto de vista práctico, aplicado a un programa informático de gran escala escrito en Java, y encontraron que algunas de las herramientas para la cobertura de código empleadas no tenían la madurez suficiente para manejar de forma apropiada programas de gran escala.

2.3.3 Nuevas formas de usar la cobertura de código

Las nuevas propuestas para utilizar la cobertura de código como fuente externa de información además de determinar las instrucciones ejecutadas, es la identificación de código hotspot, es decir, código con una alta frecuencia de ejecución, refactorización de código y depuración relacionada con el rendimiento.

La cobertura de código, al emplear métricas para determinar la estructura del programa informático sin ejecutar y ejecutadas y evaluar la fiabilidad de los casos de prueba, ayudaría en la simplificación de código, eliminando bloques de código que no aportaron funcionalidad con la ejecución del EME, y permitiendo la reconstrucción de la funcionalidad perdida en busca de en aumentar la legibilidad del código y su modernización.

Llegado este punto, Tikir y Hollingsworth (2002) propone la instrumentación de código en producción, es decir, que el uso de la Cobertura de Código se mantenga ya llegado el momento de usar el programa informático en entornos de producción, permitiendo obtener una retroalimentación de los bloques de código cubiertos con cada ejecución que el usuario realizaba, todo con el fin de proporcionar información acerca de la funcionalidad pobremente utilizada por los usuarios y patrones de uso inesperados por parte del usuario que permitieran mejoras en el diseño.

3 METODOLOGÍA

La metodología CDIO (acrónimo de concebir, diseñar, implementar y operar), ha alcanzado gran reconocimiento dentro de la ingeniería como una metodología de excelentes resultados, permitiendo comprender la importancia y impacto estratégico de la investigación y el desarrollo tecnológico en la sociedad, a su vez que permite el aumento de la productividad en ambientes altamente complejos basados en la tecnología y los procesos tecnológicos. (Crawley, Johan Malmqvist, Sören Östlund, Brodeur, Edström, 2014).

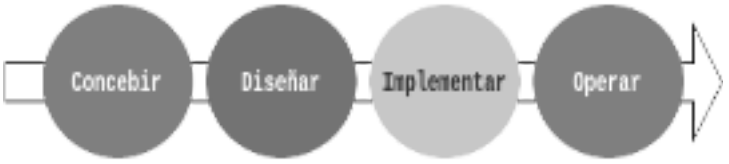

Figura 3. Etapas en la metodología CDIO

Fuente: Los Autores

El marco de trabajo CDIO Standard, permite crear un programa que se organiza alrededor de una disciplina técnica y un proceso tecnológico, este programa es conducido por la experiencia obtenida del estudiante en espacios modernos de trabajo en las etapas de diseño e implementación. Una de las características de este programa, es el aprendizaje activo y experimental, permitiendo una mejora continua de los procesos involucrados.

3.1 La etapa de concebir

3.1.1 Tecnologías consideradas

Se ha elegido como software a Maximilian, software de procesamiento de señales y Pugi, software para manejos de archivos XML con soporte a Xpath, ambos escritos en C++17 y del cual se obtendrá los reportes de cobertura de código. Como entorno de desarrollo integrado (IDE), se optó por usar dos alternativas: Clion© y Eclipse TDT©, como herramientas de compilación se buscó el soporte a C++17 , gcc y clang soporta C++17 en sus versiones 7 y 4 respectivamente y como herramientas de instrumentación de código a gcov y llvm-gcov, necesarios para generar reportes de cobertura de código.

3.1.2 El diseño conceptual

Para el caso de Maximilian, se observará módulos donde reside código que posiblemente concentre la mayoría del tiempo computacional y que cargará con altas tasas de ejecución, especialmente bloques con bucles de repetición. Para Pugi se evaluará los casos de prueba y se determinará que tan bien cubren la totalidad del código, como el lenguaje utilizado para implementar el par de software es compilado, la compilación es un requisito y esta ha de hacerse bajo ciertas banderas de compilación que le indicarán al compilador que agregue información extra que le permitirá a las herramientas de instrumentación de código obtener información desde el binario.

Figura 4. Etapas en el proceso de la cobertura de código

Fuente: Los Autores

Como segunda tarea, se selecciona las métricas que se desean registrar. El registro de las métricas es realizado por las herramientas de cobertura de código. La información registrada debe ser organizada para subsecuentes análisis. La cantidad de código instrumentado y el número de métricas a observar determinarán en qué porcentaje la cobertura de código cubre la estructura del programa.

El tercer paso en el uso de la cobertura de código es analizar las métricas. Se necesita excluir del análisis información irrelevante (como áreas de nula o baja prioridad) y enfocarse en el significado de la información registrada.

3.2 La etapa de diseño

3.2.1 Registrando las Métricas

Para Andrew (2008); Gittens et al. (2006), cada registro de las métricas tiene una puntuación asociada, que señala bloques de código con una alta prioridad. En el caso de Maximilian, los bloques de código marcados de alta prioridad han de ser aquellos que interactúan con las bibliotecas encargadas de manejar la tarjeta de sonido. Este procedimiento de asignar una puntuación más alta a áreas de alta prioridad aumenta la eficacia al observar la totalidad de los registros, permitiendo que bloques de código de escasa o nula prioridad dentro del programa sean descartados (si es necesario) en favor de observar el registro de bloques más importantes.

3.2.2 Analizando las Medidas

Andrew (2008) afirma que antes de examinar las métricas generadas por el uso de la cobertura de código, es necesario implementar todo aquel requerimiento funcional que se desee examinar. La cobertura de código no dará una evaluación a aquella funcionalidad no implementada. Unos de los retos más laboriosos al trabajar con un conjunto de programas informáticos heredados es el diseño y ejecución de casos de prueba cuando estos son inexistentes, la única forma de observar una regresión del programa informático es a través de los ejemplos que toman el papel de caso de prueba. El análisis en el caso de Maximilian busca determinar aquellos bloques de código redundantes o con altas tasas de ejecución.

3.2.3 Filtrando Métricas

Una vez que se tienen los reportes de la cobertura de código, es necesario filtrar la información que se tiene. Los reportes deben filtrar y excluir toda condición ilegal conocida, tal como lógica sin usar o de baja prioridad, así como áreas de código cuyo funcionamiento haya sido verificado. Aquellas condiciones ilegales son cláusulas "if" y "else" que evalúan casos sin uso o segmentos de código predeterminados que no deben de suceder bajo ciertas condiciones de entrada, como la ejecución de código específico a una plataforma o biblioteca.

3.3 La etapa de implementar

La elección de gcov como herramienta de instrumentación de código se debió a su uso como herramienta de profiling para descubrir áreas de código con altas tasas de ejecución y costosas en tiempo de computación, la baja sobrecarga de ejecución en el código instrumentado, su uso en conjunto con otras herramientas de profiling (como gprof), la integración con entornos de desarrollo integrado (IDE) y herramientas de compilación (como gcc y llvm).

Para el proceso de instrumentación y compilación de ambos programas se usó, gcc 9.2.1 con soporte a C++17, banderas de optimización deshabilitadas, necesarias para obtener resultados congruentes y banderas de cobertura habilitadas, requisito de las herramientas de cobertura de código.

4 Resultados y Discusión

Para el caso del programa informático Maximilian, el caso de prueba es la ejecución constante de una onda sideral de 440 Hz por un lapso de 2 segundos, la primera identificación de una zona de código con una alta frecuencia de ejecución es en la función **startCallbackFunction**.

```
void LinuxAlsa::startCallbackFunction()
{
    for (int i = 0; i < stream .bufferSize; i++){
        audioCallback(data);
        for (int j = 0; j < 2; j++){
            bufferConvert[indexOfBuffer] = data[j];
            indexOfBuffer += 1;
        }
    }
```

Figura 5. Función con alta frecuencia de ejecución

Los reportes de cobertura de código muestran que la función es llamada 91 veces.

Subsecuentes ejecuciones con los mismos datos de entrada y parámetros de configuración muestran que el número de ejecuciones reportadas permanece invariante. El reporte de cobertura de código muestra que 88% de los archivos con código fuente fueron ejecutados y un 20% de líneas de código fueron ejecutadas, sin embargo no es concluyente, ya que este no toma en cuenta el paradigma multiplataforma del programa informático, obviando el hecho de existe código que no es capaz de ejecutarse por haber sido diseñado para otra plataforma.

Table 1: Número de ejecuciones para la función startCallbackFunction

Línea de Código	Número de Ejecuciones
1.047	93.275
1.048	93.184
1.049	279.552
1.050	186.368
1.051	186.368

Gracias a los reportes de cobertura de código, se pudo simplificar bloques redundantes de código (commit 099c9), mostrando así el potencial de la cobertura de código como herramienta de refactorización de software.

Para el caso de Pugi, la metodología de desarrollo elegida fue TDD, se usó desde el la etapas iniciales de la implementación y los reportes de cobertura de código muestran que al ejecutar los casos de prueba (946 en total) se ejecutaron el 100% de los archivos con código fuente y el 99% de las líneas de código. Esto entra a reforzar las conclusiones de Derezińska (2008), quien afirma el alto porcentaje de cobertura de código que se debe de lograr usando metodologías como TDD.

5 CONCLUSIONES

Para el personal encargado de los casos pruebas, la cobertura de código les asegurará que los planes de verificación y pruebas trazadas para módulos y funciones cubre la totalidad del código a probar, además, los desarrolladores han de asegurarse que la totalidad de líneas escritas para añadir funcionalidad a un programa informático han sido probadas y para equipos de desarrollo centrados en las pruebas de regresión, la cobertura de código simplifica su labor, identificando código no ejecutado por la casos de prueba existentes. Así que los beneficios directos de la cobertura de código son determinar la eficacia de los casos de prueba. Igualmente existen otras áreas donde la cobertura de código es un recurso valioso, permitiendo identificar zonas de código con altas frecuencias de ejecución, la depuración de programas con problemas de rendimiento y la refactorización del código, identificando bloques de código con nula funcionalidad y/o redundantes.

La cobertura de código aplicada a propuestas como las de Tikir y Hollingsworth (2002), permitiría utilizar la cobertura de código como herramienta de retroalimentación para determinar funcionalidad pobremente usada por los usuarios y patrones de uso inesperados. Otros beneficios de la cobertura de código son la priorización de zonas de código, donde la cobertura de código es baja y la alta medida cuantificable del progreso realizado en las pruebas. Metodologías como TDD permiten una alta cobertura de código desde el inicio y su facilidad permite que principiantes logren una alta tasa de cobertura de código con poco esfuerzo.

Uno de los riesgo al usar la cobertura de código es el desenfoque de áreas prioritarias en el proyecto, el diseño, la preparación y la implementación de casos de prueba es una tarea ardua que necesita el dominio del área específica y los conocimiento necesarios para decir que un programa informático está bien probado, el solo hecho de implementar casos de prueba con el fin de aumentar la cobertura de código en el programa no garantizará la ausencia de errores. Además de que no existe una teoría subyacente que determine qué tanto mejora la eficacia de los casos de prueba con una cobertura de código alta. Los problemas de interpretación en las métricas de cobertura de código también son otro problema, ya que confunden a quien la aplica, obteniendo resultados dispares entre diferentes métricas a un mismo caso de prueba ejecutado. Las métricas de la cobertura de código se ven afectadas negativamente en su fiabilidad entre más grande es el programa informático.

REFERENCIAS

Alemerien, K., & Kenneth, M. (2014). *Examining the effectiveness of testing coverage tools: An empirical study. International Journal of Software Engineering and its Applications*, 8, 139-162. https://doi.org/10.14257/ijseia.2014.8.5.12

Andrew, P. (2008). *Code Coverage. En Functional Verification Coverage Measurement and Analysis* (pp. 79-95). Springer US. https://doi.org/10.1007/978-1-4020-8026-5

Cai, X., & Lyu, M. R. (2005). *The effect of code coverage on fault detection under different testing profiles. ACM SIGSOFT Software Engineering Notes*, 30(4), 1. https://doi.org/10.1145/1082983.1083288

Crawley, E. F., Johan Malmqvist, Sören Östlund, Brodeur, D. R., & Edström, K. (2014). *Rethinking engineering education the CDIO approach. Springer.*

Derezińska, A. (2008). *Experiences from an Empirical Study of Programs Code Coverage. En T. Sobh (Ed.), Advances in Computer and Information Sciences and Engineering* (pp. 57-62). Springer Netherlands. https://doi.org/10.1007/978-1-4020-8741-7_11

Gittens, M., Lutfiyya, H., Bauer, M., Godwin, D., Kim, Y., & Gupta, P. (2002). *An empirical evaluation of system and regression testing. Commun*. ACM, 3. https://doi.org/10.1145/782115.782118

Gittens, M., Romanufa, K., Godwin, D., & Racicot, J. (2006). *All Code Coverage is Not Created Equal: A Case Study in Prioritized Code Coverage*. Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research. https://doi.org/10.1145/1188966.1188981

Gokhale, S., & Mullen, R. (2006). *The Marginal Value of Increased Testing: An Empirical Analysis using Four Code Coverage Measures*. J. Braz. Comp. Soc., 12, 13-30. https://doi.org/10.1007/BF03194493

Gupta, A., Mishra, N., Tripathi, A., Vardhan, M., & Kushwaha, D. S. (2015). *An Improved History-Based Test Prioritization Technique Technique Using Code Coverage*. En H. A. Sulaiman, M. A. Othman, M. F. I. Othman, Y. A. Rahim, & N. C. Pee (Eds.), Advanced Computer and Communication Engineering Technology (Vol. 315, pp. 437-448). Springer International Publishing. https://doi.org/10.1007/978-3-319-07674-4_43

Horváth, F., Gergely, T., Beszédes, Á., Tengeri, D., Balogh, G., Gyimóthy, T. (2019). *Code coverage differences of Java bytecode and source code instrumentation tools. Software Quality Journal*, 27(1), 79-123. https://doi.org/10.1007/s11219-017-9389-z

Kajo-Mece, E., & Tartari, M. (2012). *An evaluation of java code coverage testing tools*. 920, 72-75.

Kessis, M., Ledru, Y., & Vandome, G. (2005). *Experiences in coverage testing of a Java middleware. Proceedings of the 5th International Workshop on Software Engineering and Middleware* - SEM '05, 39. https://doi.org/10.1145/1108473.1108483

Kirner, R. (2009). *Towards Preserving Model Coverage and Structural Code Coverage. EURASIP Journal on Embedded Systems*, 2009. https://doi.org/10.1155/2009/127945

Li, N., Meng, X., Offutt, J., & Deng, L. (2013). *Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools* (Experience Report). 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 380-389. https://doi.org/10.1109/ISSRE.2013.6698891

Lingampally, R., Gupta, A., & Jalote, P. (2007). *A Multipurpose Code Coverage Tool for Java*. 2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07), 261b-261b. https://doi.org/10.1109/HICSS.2007.24

Miller, J. C., & Maloney, C. J. (1963). *Systematic Mistake Analysis of Digital Computer Programs. Commun.* ACM, 6(2), 58–63. https://doi.org/10.1145/366246.366248

Piwowarski, P., Ohba, M., & Caruso, J. (1993). *Coverage measurement experience during function test*. Proceedings of 1993 15th International Conference on Software Engineering, 287-301. https://doi.org/10.1109/ICSE.1993.346035

Rapps, S., & Weyuker, E. J. (1985). *Selecting Software Test Data Using Data Flow Information*. IEEE Transactions on Software Engineering, SE-11(4), 367-375. https://doi.org/10.1109/TSE.1985.232226

Rothermel, G., Untch, R. H., Chengyun Chu, & Harrold, M. J. (2001). *Prioritizing test cases for regression testing*. IEEE Transactions on Software Engineering, 27(10), 929-948. https://doi.org/10.1109/32.962562

Tengeri, D., Horvath, F., Beszedes, A., Gergely, T., & Gyimothy, T. (2016). *Negative Effects of Bytecode Instrumentation on Java Source Code Coverage*. 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 225-235. https://doi.org/10.1109/SANER.2016.61

Tikir, M. M., & Hollingsworth, J. K. (2002). *Efficient Instrumentation for Code Coverage Testing*. SIGSOFT Softw. Eng. Notes, 27(4), 86–96. https://doi.org/10.1145/566171.566186

Whalen, M. W., Rajan, A., Heimdahl, M. P. E., & Miller, S. P. (2006). *Coverage metrics for requirements-based testing*. Proceedings of the 2006 International Symposium on Software Testing and Analysis - ISSTA'06, 25. https://doi.org/10.1145/1146238.1146242

Yang, Q., Li, J. J., & Weiss, D. M. (2009). *A Survey of Coverage-Based Testing Tools*. The Computer Journal, 52(5), 589-597. https://doi.org/10.1093/comjnl/bxm021