Estudio de Caso sobre Herramientas de Generación Automática de Casos de Prueba

Gerardo Quintana, Martín Solari, Santiago Matalonga

Centro de Investigación e Innovación en Ingeniería de Software – CI3S, Universidad ORT Uruguay
Montevideo, Uruguay
gquintana@uni.ort.edu.uy, martin.solari@ort.edu.uy, smatalonga@uni.ort.edu.uy

Resumen. Dos de los objetivos primarios de las pruebas de software son alcanzar una alta cobertura estructural y encontrar la mayor cantidad de defectos desconocidos. Las herramientas para la generación automática de casos de prueba pueden alcanzar una alta cobertura, pero se requieren más estudios para determinar la eficacia y el tipo de defectos que pueden detectar. El objetivo del estudio de caso es analizar dos herramientas, *Evosuite* y *Randoop,* con el propósito de comparar la eficacia y los tipos de defectos que detectan. Se utilizaron dos programas bajo prueba con defectos sembrados (según la clasificación ortogonal de defectos (ODC): algoritmo, de control, asignación e interface). Estos tipos de defectos han sido reportados como los más frecuentes en estudios anteriores. Los resultados indican que las herramientas no siempre logran una alta cobertura y que tienen una baja eficacia, en oposición a algunos estudios anteriores, para detectar algunos tipos de defectos.

Palabras clave: Ingeniería de software. Estudio de caso. Herramientas de generación de casos de prueba. Pruebas unitarias.

1 Introducción

La generación de casos de prueba es una de las actividades clave en las pruebas de software, pero es considerado un proceso costoso. La automatización de la generación de los casos de prueba puede disminuir los costos de las pruebas de software. Por este motivo, uno de los sueños de las pruebas de software es que sea 100% automatizado [1]. Pero, este sueño aún no se ha logrado. Se requieren herramientas con suficiente capacidad para cumplir con los objetivos de prueba o análisis.

En las últimas décadas se puede observar un avance importante en lo que respecta a la generación automática de casos de prueba. En un trabajo previo [2], se identificaron los principales temas que se investigan en el área de automatización de las pruebas estructurales y se dio una clasificación de las técnicas que se reportan como más eficaces y eficientes para la generación de casos de prueba.

Para la automatización de la generación existen diversas técnicas, de las cuales se pueden distinguir las siguientes: pruebas aleatorias [3], pruebas de ejecución simbóli-

adfa, p. 1, 2011.

ca [4], pruebas de ejecución simbólica dinámica (del inglés Dynamic Symbolic Execution: DSE) también llamadas pruebas concolic [5] y pruebas de software basadas en búsqueda (del inglés Search Based Software Testing: SBST) [6].

Las pruebas aleatorias consisten en elegir al azar los valores en el dominio de entradas posibles. Esta estrategia a menudo genera un número significativo de casos de prueba en un corto tiempo. No obstante, puede tener dificultades en cubrir partes de un programa de difícil acceso porque por ejemplo, para cubrirlo, se requiere valores específicos [7].

Ejecución simbólica es una técnica estática de análisis de código fuente en la que los caminos de los programas son descritos como un conjunto de restricciones que implica solamente los parámetros de entrada de un programa [8]. Las técnicas estáticas de análisis de código, como es ejecución simbólica, no requieren que el código bajo prueba sea ejecutado. Existen algunas herramientas que usan como base ejecución simbólica, como por ejemplo *JPF* [9]. Una de las principales críticas que tienen las técnicas estáticas es su alto costo de computación. Además, algunos de las restricciones se pueden volver insolubles.

Las pruebas basadas en búsqueda (por sus siglas en inglés SBST) se basan en algoritmos meta-heurísticos para automatizar o parcialmente automatizar las tareas de prueba. Las técnicas o algoritmos meta-heurísticos son un conjunto de algoritmos genéricos usados para encontrar soluciones óptimas o casi óptimas a los problemas que tienen espacios de búsqueda de alta complejidad. Las pruebas evolutivas (del inglés Evolutionary Testing) son un sub-campo de SBST en las que los algoritmos evolutivos son usados para guiar la búsqueda [10].

En este trabajo se van a estudiar dos de las principales técnicas mencionadas anteriormente para la generación automática: SBST y pruebas aleatorias. SBST recientemente ha atraído gran interés en la comunidad de investigadores. Desde que Xanthakis aplicó algoritmos genéticos al problema de la generación de datos de prueba [11], hubo un aumento creciente de publicaciones relacionadas con este tema [12], demostrando ser un enfoque muy promisorio para resolver el problema de la generación [12]. Por otra parte, las pruebas aleatorias a pesar de las dificultades que pueden presentar para lograr una amplia cobertura del código, estudios relacionados han demostrado resultados valiosos con respecto a la cobertura y a la detección de defectos [13], [14], [15], [16].

A través del presente estudio se explora cuál es la utilidad de las herramientas desde el punto de vista de la cobertura de código y de la detección de defectos en el software. El objetivo del estudio de caso es analizar dos herramientas de generación de casos de prueba, con el propósito de comparar y entender con respecto a la eficacia en revelar defectos y a los tipos de defectos que detectan. Para llevar a cabo el estudio, se seleccionó un lenguaje orientado a objetos, específicamente Java, dado que ofrece las herramientas más actualizadas para la generación automática de pruebas unitarias. Como sujetos de prueba se seleccionaron dos programas Java pequeños a los que se les sembraron algunos defectos. Estos defectos han sido reportados como los que más frecuentemente que insertan en el código los programadores [17], [18]. Con respecto a las herramientas, se seleccionaron dos representativas de las técnicas selecciónadas para el estudio (pruebas aleatorias y SBST) que generan casos de

prueba JUnit para programas Java. Hubo un proceso para seleccionar las herramientas más representativas de las técnicas (explicado con detalle en la sección 3.3.) y algunas fueron descartas por dificultad de configuración y uso. Asimimso, las herramientas seleccionadas *Randoop* (pruebas aleatorias) y *Evosuite* (SBST), han demostrado obtener una alta cobertura y ser eficaces en la detección de defectos.

Los resultados obtenidos en el estudio de caso muestran que los casos de prueba no siempre logran una alta cobertura. Los defectos en el código pueden impedir que las herramientas no logren una alta cobertura. La eficacia en la detección de defectos es baja, en oposición a algunos estudios anteriores, para detectar los tipos de defectos sembrados en el estudio de caso según la clasificación ortogonal de defectos (ODC): algoritmo, de control, asignación e interface.

El resto del artículo está organizado de la siguiente forma. La sección 2 presenta el trabajo relacionado al estudio de caso. La sección 3 describe el estudio de caso con su objetivo, sus correspondientes preguntas de investigación, las herramientas utilizadas y los programas bajo prueba y su configuración. Los resultados del estudio de caso son presentados en la sección 4 y en la sección 5 se realiza una discusión de éstos. Las amenazas a la validez son abordadas en la sección 6 y la sección 7 presenta las conclusiones y el trabajo futuro.

2 Trabajo Relacionado

Existen muchos temas investigados en la generación automática de casos de prueba. Los principales temas investigados son la performance de las técnicas y los problemas para la generación.

En lo que respecta a la performance, dos enfoques para la generación han sido comparados [19]: el enfoque concolic de la herramienta *Cute* [20] y un enfoque basado en búsqueda de la herramienta *Austin* [21]. En esta investigación las herramientas se aplicaron a programas complejos y no se realizaron ajustes en ellas ni en los programas. Los resultados del estudio indican que las dos herramientas enfrentaron varios problemas para la generación de los casos de prueba y ninguna de ellas logró cubrir más del 50 % de las ramas del código. También el estudio sugiere combinar estos enfoques que se desarrollaron completamente independientes dado que puede aumentar la eficacia en la generación automática de las pruebas de software.

En la práctica cuando las herramientas se enfrentan con sistemas complejos no resuelven todos los problemas para la generación de prueba sin la intervención humana. Se ha propuesto una metodología llamada pruebas del desarrollador cooperativas [22], donde los desarrolladores proveen una guía a las herramientas para ayudarlas a abordar los problemas. En una encuesta [23] acerca de la utilidad de las herramientas automáticas de prueba, el 80 % de los profesionales rechazaron la visión de que las pruebas de software serán totalmente automatizadas.

En lo que respecta a la detección de defectos las herramientas automáticas pueden revelar defectos no encontrados por técnicas de prueba manuales [16]. Algunos estudios indican también que las estrategias de prueba manuales y automáticas son complementarias [24], [25].

En el contexto de SBST se han presentado enfoques para aumentar la performance en la generación de las pruebas, ya que hay escenarios en los que las funciones de adecuación que son las que determinan el grado de aptitud de una solución candidato, no pueden proveer una adecuada guía en la búsqueda de datos. Un enfoque [26] consiste en incluir al profesional con conocimiento en el dominio de la aplicación, en el proceso de generación, para que provea retroalimentación y lograr de esta forma mayor cobertura. Para implementar este enfoque se usa la herramienta *Evosuite* [27]. Con *Evosuite* se ha demostrado que cuando el programa no tiene dependencias con el entorno (por ejemplo, cuando el código no accede a un sistema de archivo o a la red) puede alcanzar una cobertura promedio de hasta el 90 %, pero cuando existen este tipo de dependencias obtuvo en promedio una cobertura del 48 % [28].

En un experimento [29] que involucra a sujetos humanos que se les solicita construir conjunto de casos de prueba JUnit de forma manual o con la asistencia de *Evosuite*, se obtuvo en una de las clases estudiadas una tasa de detección de defectos (porcentaje de mutantes detectados por el conjunto de casos de prueba) de 0,38 con la asistencia de *Evosuite* y de 0,89 sin la asistencia de la herramienta. Este estudio, si bien confirma que las herramientas son efectivas para lograr una alta cobertura comparada con la alcanzada por los humanos, no se puede afirmar que las herramientas mejoren nuestra habilidad para probar software y cuestionan cómo la comunidad de investigadores evalúan las herramientas. En otro estudio [7] en el que se aplica *Evosuite* en 100 proyectos de software de código abierto seleccionados al azar, revela que SBST es muy adecuado tanto para revelar defectos en el software como para producir conjuntos de casos de prueba representativos para cualquier criterio de cobertura. *Evosuite* ha sido evaluado en términos de cobertura y de mutation score con las herramientas *t2* [30], *DSC* [31] y *Randoop* [32] en una competición [33] donde obtuvo el primer lugar.

La herramienta de generación de casos de prueba aleatorios *Randoop* [32] ha demostrado tener una efectividad similar a la de las pruebas unitarias manuales en lo que respecta a la detección de defectos [15]. Se encontró también que los casos de prueba generados detectan distintos defectos que los encontrados por las pruebas unitarias manuales, demostrando que ambos enfoques para la generación son complementarios [15], [13]. La performance para la generación unitaria de programas orientada a objetos, mediante *Randoop*, ha demostrado [14] ser útil para encontrar defectos en software industrial. Además, encuentra una gran cantidad de defectos en pocos minutos.

A pesar de que algunos estudios como los citados anteriormente [16], [24], [25], [15], [13], indican que las herramientas pueden generar casos de prueba en forma eficiente y pueden ser utilidad para los profesionales en lo que respecta a la detección de defectos, la adopción de las herramientas por parte de la industria ha sido muy limitada y no todos los estudios concluyen que las herramientas ayuden a los profesionales a encontrar defectos [29], [34]. Las herramientas de generación automática de prueba unitaria pueden generar casos de prueba no esenciales, esto es que representan ejecuciones que nunca ocurren en la realidad [34]. Estas ejecuciones no esenciales pueden revelar defectos falsos, que indican defectos en las pruebas en lugar de

defectos en el código. En un estudio exploratorio con 5 sujetos se encontró que el 100 % de las pruebas generadas por *Randoop* [32] fueron defectos falsos [34].

3 Estudio de Caso

La presente sección brinda una descripción del estudio de caso, el material de estudio y su configuración.

Estudio de caso es un método empírico que tiene como objetivo investigar un fenómeno contemporáneo en su contexto [35]. El estudio de caso normalmente tiene como objetivo el seguimiento de un determinado atributo o el establecimiento de relaciones entre los diferentes atributos.

3.1 Objetivos del Estudio de Caso

Los objetivos del estudio de caso se definen utilizando el paradigma GQM [36]. El objetivo del estudio de caso es analizar dos herramientas de generación automática de casos de prueba, con el propósito de comparar y entender con respecto a la eficacia en revelar defectos y a los tipos de defectos que detectan.

El estudio de caso es exploratorio en el sentido que busca determinar qué sucede con estas herramientas, busca determinar nuevos puntos de vista y generar ideas y nuevas hipótesis para nuevas investigaciones.

Para lograr los objetivos, se definieron las siguientes tres preguntas de investigación (PI).

- PI1: ¿Cuántos casos de prueba generan cada una de las herramientas?
- PI2: ¿Qué cobertura de rama se obtiene con los casos de prueba generados por las herramientas?
- PI3: ¿Qué cantidad y qué tipo de defectos se detectan con los casos de prueba generados por las herramientas?

3.2 Proceso de Estudio y Variables

El estudio ha sido dividido en 3 fases: (1) la fase de preparación, (2) la ejecución del estudio, y (3) la fase que consiste en el análisis y la evaluación. La primera fase trata acerca de la preparación de las herramientas y el sembrado de defectos a los programas. La ejecución del estudio consiste en la generación por parte de las herramientas de los casos de prueba en cada uno de los programas. Los casos de prueba se van a generar en los programas con los defectos y sin los defectos sembrados para determinar si los defectos no permiten que las herramientas logren una alta cobertura. Por último, en la fase de análisis y evaluación se realizaron las medidas de las variables del estudio con los casos de prueba generados.

El estudio se concentra en las pruebas unitarias dado su importancia en las pruebas de software y en programas orientados a objetos. Para realizar el estudio de caso se sembraron en dos programas Java un conjunto de defectos. Los tipos de defectos

sembrados surgen de una investigación [17] que presenta una clasificación de los defectos específica para los programas orientados a objetos y el lenguaje Java, basada en un conjunto de defectos encontrados en aplicaciones Java, y proponen una extensión a la clasificación anterior [18]. En la investigación se clasifican los defectos más representativos que surgen de analizar 574 defectos de software Java. Los resultados obtenidos son comparados y muestran que los errores que tienen los programadores siguen un patrón. Existen tres grandes categorías de los defectos más representativos: missing construct, wrong construct y extraneous construct. Dentro de cada categoría se identificaron defectos específicos que se clasificaron según la clasificación de defectos ortogonal [37] (Orthogonal Defect Classification: ODC).

Las variables independientes del estudio incluye la elección de las herramientas, la duración del estudio y el número y el tipo de defectos sembrados. Las variables dependientes son el número de casos de prueba generados, el número y el tipo de defectos detectados y el número de falsos positivos (código correcto pero incorrectamente reportado con defectos).

3.3 Herramientas

De acuerdo a los objetivos del estudio se seleccionaron dos herramientas efectivas, representativas de las técnicas selecciónadas para el estudio (pruebas aleatorias y SBST) y que generan casos de prueba JUnit para programas Java. Hubo un proceso para seleccionar las herramientas más representativas de las técnicas. Dentro de las herramientas que generan casos de prueba para programas Java se evaluaron las siguientes: *t2* [30] (pruebas aleatorias), *JCrasher* [38] (pruebas aleatorias), *Randoop* (pruebas aleatorias), *Testful* [39] (SBST) y *Evosuite* (SBST) [27]. Las herramientas seleccionados son: *Randoop* y *Evosuite*. Se encontró que estas herramientas son las más efectivas reportadas en diversos estudios en lo que respecta a la cobertura y la cantidad de defectos encontrados, y las más fáciles de usar y configurar.

Randoop. Es una herramienta que genera pruebas unitarias mediante el uso de pruebas aleatorias dirigidas por la retroalimentación (del inglés feedback-directed random testing) [32]. Esta técnica inspirada en las pruebas aleatorias usa retroalimentación de ejecución obtenida mediante la ejecución de entradas de prueba a medida que son creadas, para evitar la generación de entradas ilegales y redundantes [32]. *Randoop* ha sido aplicado para la generación de casos de prueba en diferentes estudios [15], [40], [34].

Randoop puede generar tanto pruebas de regresión JUnit, como casos de prueba para revelar defectos. Las pruebas de regresión generadas son pruebas que no violan contratos provistos, mientras que las pruebas que violan contratos pueden ser para revelar defectos. *Randoop* crea secuencia de llamadas a métodos incrementalmente, seleccionando en forma aleatoria una llamada a método para aplicar y mediante la selección de argumentos de secuencias construidas previamente.

Evosuite. Usa un enfoque evolutivo para generar los conjuntos de casos de prueba [27]. Mediante un algoritmo genético deriva candidatos individuales llamados cromosomas usando operadores inspirados por la evolución natural (por ejemplo, selección, cruce y mutación), de tal manera que forma iterativamente mejores soluciones con

respecto a objetivo de optimización (por ejemplo, la cobertura de rama) [28]. Los cromosomas en *Evosuite* son conjuntos de casos de prueba que consisten en un número variable de casos de prueba, que son secuencias de llamadas a métodos.

3.4 Sujetos y Configuración del Estudio de Caso

Para definir el estudio de caso se instancia el siguiente marco metodológico [41] para evaluar técnicas de prueba de software. El objetivo del marco es permitir a los investigadores definir más fácilmente los estudios de caso. Al mismo tiempo garantiza que se han cumplido las numerosas directrices y listas de control para hacer el trabajo empírico. Además, dado que los estudios de caso se ejecutarán de acuerdo con un diseño similar, será más fácil comparar los resultados obtenidos, y por lo tanto un cuerpo de evidencia puede ser construido de manera que permita investigar las características generales sobre las técnicas y herramientas evaluadas en diferentes estudios de caso se podrían especificar. La instanciación del marco para el estudio de caso se presenta en la Tabla 1.

Tabla 1. Descripción de las herramientas evaluadas.

Prerrequisitos	
Estático, dinámico	Dinámico.
Tipo de software	Clases Java.
Fase del ciclo de vida	Pruebas unitarias para programas Java.
Entorno	Eclipse IDE
Escalabilidad	Programas Java pequeños.
Entrada	Clases Java compiladas.
Conocimiento	Pruebas unitarias JUnit para Java.
Experiencia	Conocimiento básico de pruebas unitarias.
Resultados	
Salida	Casos de prueba JUnit.
Eficacia	Investigado por el estudio de caso.
Tipos de defectos	Investigado por el estudio de caso.
Operación	
Interacción	A través de la línea de comando para *Evosuite*. A través de Eclipse IDE para *Randoop*.
Fuente de información	Manuales, artículos y páginas web. *Evosuite:* http://www.evosuite.org, [27]. *Randoop:* http://randoop.googlecode.com/hg/plugin/doc/index.html, [32].
Aplicabilidad de la tarea	Pruebas unitarias; Generación automática de casos de prueba.
Madurez	Herramientas de investigación académicas bajo desarrollo.
Obtención de las herramientas	
Licencia	*Evosuite:* GNU General Public License V3. *Randoop:* MIT License.
Costo	Libre.
Soporte	Ninguno.

Como se puede observar en los prerrequisitos descritos en la Tabla 1, las herramientas utilizadas en el estudio de caso son dinámicas, dado que requieren la ejecución del código bajo prueba para generar los casos de prueba. Para la generación además, se requiere las clases compiladas de los programas bajo prueba y se utiliza el entorno de desarrollo integrado (sigla en inglés IDE: integrated development environment) de Eclipse.

Los sujetos de prueba son dos programas escritos en Java. Los programas han sido utilizados en otros experimentos de prueba, aplicando otras técnicas manuales, que no son parte de este estudio. Se utilizan estos programas porque se conocen los principales defectos y porque brindan la posibilidad de hacer comparaciones posteriores.
Los programas son los siguientes:

- Conversor: tiene como propósito convertir una lista de medidas en el sistema de unidades estadounidense al sistema internacional (métrico). El programa tiene 665 instrucciones en 147 líneas de código con un total de 92 ramas.
- Export: tiene como propósito exportar a distintos formatos la información contenida en una tabla de movimientos contables. Solo analiza la línea de comandos proporcionada y evalúa su corrección sintáctica y semántica. El programa tiene 663 instrucciones en 136 líneas de código con un total de 60 ramas.

En ambos programas, para que las herramientas no tengan problemas con el acceso a métodos por ser privados, todos los métodos de la clase se cambiaron a públicos.

En la Tabla 2 se presentan los tipos de defectos sembrados en los programas. Para cada tipo de defecto, se indica el tipo de defecto ODC y la cantidad sembrada en cada programa. Estos defectos surgen de los más frecuentes encontrados en dos artículos [17], [18] y son los siguientes: missing if construct plus statements (MIFS), missing function call (MFC), missing functionality (MFCT), missing if construct around statements MIA, wrong data types or conversion used (WSUT), wrong variable used in parameter of function call (WPFV), wrong logical expression used as branch condition (WLEC) y Wrong return value (WRV). En los programas bajo prueba se sembraron estos tipos de defectos menos el tipo de defecto MFCT, dado que solo puede ser detectado por técnicas de prueba basadas en la especificación [42]. Las herramientas del estudio de caso generan basándose en el código fuente. En el programa Export se sembraron los tipos de defectos del #1 al #3, y en el programa Conversor se sembraron los defectos del #4 al #7.

Tabla 2. Defectos sembrados en los programas, el tipo de defecto ODC y la cantidad sembrada en cada proyecto.

#	Tipo de defecto	Tipo de defecto ODC	Cantidad
1	MIFS	Algoritmo	2
2	MFC	Algoritmo	1
3	MIA	De Control	1
4	WSUT	Asignación	1

#	Tipo de defecto	Tipo de defecto ODC	Cantidad
5	WPFV	Interface	1
6	WLEC	De Control	2
7	WRV	Interface	1

Para aumentar las posibilidades de lograr una mayor cobertura y una mayor tasa de detección de defectos, se invocaron las herramientas 5 veces con diferentes argumentos. Para *Randoop* se modificó la configuración por defecto, aumentando el tiempo de generación a 120 segundos. También se incrementó la relación de nulo por encima del 0,0. Esta configuración es recomendada en investigaciones anteriores [15] y con mejores resultados que la configuración por defecto, de forma tal que el nulo sea pasado como parámetro en las llamadas a métodos. Para *Evosuite* se configuró para correr 120 segundos. Se utilizó un tamaño de población de 50, longitud máxima de prueba de 20 sentencias y los parámetros por defecto determinados en un estudio de ajuste de parámetros [43]. Esta configuración de la herramienta se ha utilizado en investigaciones anteriores [7].

4 Resultados

Esta sección presenta los resultados con respecto a las preguntas de investigación planteadas.

4.1 PI1y PI2: Cantidad de Casos de Prueba y Cobertura

La Tabla 3 presenta los resultados que responden a la pregunta de investigación 1, respecto a la cantidad de casos de prueba generados, y a la pregunta de investigación 2 que trata acerca de la cobertura de rama obtenida. Se midió la cobertura obtenida generando los casos de prueba en el código con y sin defectos para comparar.

Tabla 3. Resultados de la generación (casos de prueba y cobertura).

Programa	Herramienta	Con defectos		Sin defectos	
		# Casos	%Cobertura	# Casos	%Cobertura
Conversor	*Evosuite*	25	82,7	38	90,2
	Randoop	5002	45,7	5002	40,2
Export	*Evosuite*	12	82,2	14	90,4
	Randoop	5004	48,7	5003	50,4

Se puede observar que en todos los casos *Evosuite* obtuvo una cobertura mayor que *Randoop*, a pesar de que el primero genera menos cantidad de casos de prueba. Por otra parte, la cantidad de casos de prueba generados con *Evosuite* aumenta en los programas sin los defectos sembrados. Asimismo, la cobertura de los casos de prueba generados con *Evosuite* aumenta cuando el programa no tiene los defectos sembrados.

En cambio, *Randoop* genera casi la misma cantidad de casos de prueba para los dos programas independientemente de si el programa tiene los defectos sembrados. A diferencia de *Evosuite*, la cobertura obtenida por *Randoop* disminuye en el programa Conversor con los defectos a pesar de que la cantidad de casos de prueba se mantiene.

4.2 PI3: Defectos Detectados

Los casos de prueba generados no revelan la existencia de los defectos sembrados en ninguno de los programas. *Randoop* encuentra dos null pointer exception (NPE) en cada uno de los programas que surgen de pasar valores nulos como parámetros a los métodos. Uno de los fallos en la ejecución de los casos de prueba es un falso positivo dado que es generado por pasarle a las variables de tipo enumerado valores nulos. Los demás surgen por pasarle valores nulos a los métodos.

En cambio, generando los casos de prueba sin los defectos sembrados en el código y ejecutándolos con JUnit en el código con los defectos sembrados en el programa Conversor, los casos de prueba a través de los assert revelan defectos en el código, dado que las salidas esperadas no son las mismas que las obtenidas. Es decir que el criterio de prueba establecido en el assert no se cumple. Se puede observar que los fallos de ejecutar los casos de prueba de *Randoop* son lanzados muchas veces dado que genera casos de prueba redundantes. Por ejemplo, en el programa Conversor la ejecución de los casos de prueba generados por *Randoop* lanzan 2860 fallos y los generados por *Evosuite* lanzan 15 fallos.

5 Discusión

La cobertura de rama obtenida en el estudio de caso con *Randoop* es baja comparada con la obtenida en promedio en otros estudio (77,86 %) [40]. En cambio, con *Evosuite* en el estudio de caso se obtuvo una cobertura similar a la cobertura promedio (90 %) obtenida por la herramienta en otros estudios [28]. Las herramientas de pruebas aleatorias como *Randoop,* generan un gran número de casos de prueba, pero pueden tener el problema de lograr una alta cobertura dado que algunos estados de los objetos son difíciles de alcanzar en particular con datos generados aleatoriamente. Por otra parte, las herramientas como *Evosuite*, a pesar de que generan conjuntos de casos de prueba más reducidos que las aleatorias, logran una mayor cobertura. *Evosuite*, que está basada en SBST, utiliza funciones de adecuación que guían la búsqueda de datos. Además, *Evosuite* en lugar de optimizar los casos de prueba para cada rama, tiene como objetivo optimizar un conjunto de casos de prueba con respecto a todas las ramas [44], ayudando así a lograr una mayor cobertura.

A partir de los resultados del presente estudio de caso se puede observar que las herramientas tienen baja eficacia en la detección de defectos. En este estudio, no se encontró la misma efectividad para la detección de defectos reportadas en otros [24], [25], en los que algunas herramientas aleatorias han demostrado tener una efectividad en la detección de defectos similar a la de las pruebas unitarias manuales. De acuerdo a los datos observados, las herramientas evaluadas no ayudan efectivamente a los

profesionales a detectar defectos cuando son usadas de la forma en que se utilizaron en el estudio. Los oráculos de prueba para estas herramientas dinámicas basadas en el código fuente de los programas, están limitadas a un reducido conjunto de defectos, como por ejemplo a caídas del programa o excepciones no controladas. Los defectos frecuentes que quedan en los programas, como los sembrados en los programas del estudio de caso, pueden ser detectados verificando que el comportamiento del programa coincida con el comportamiento esperado. Las herramientas generan asserts de prueba, que son el primer paso para la generación de oráculos de prueba. Estos asserts no son realmente oráculos de prueba porque encapsulan el comportamiento observado mediante la ejecución del caso de prueba, en lugar del comportamiento deseado [45]. Para mejorar la tasa de detección de defectos se debe continuar investigando la generación de oráculos de prueba más efectivos. De la forma en que este tipo de herramientas los generan actualmente, ejecutando código incorrecto [29], la detección de defectos es poco probable. Con estos resultados, la utilidad de las herramientas se podría limitar a la generación de casos para pruebas de regresión y también para generar casos de prueba como base para el profesional.

Con respecto a la facilidad de uso de las herramientas, ambas resultaron ser muy amigables. Las dos se integran al entorno de desarrollo integrado de Eclipse y sus parámetros son fácilmente modificables. *Evosuite* también genera un reporte que consiste en archivos de datos CVS (datos de cada una de las ejecuciones) y un reporte en HTML para mostrar los casos de prueba generados y la cobertura.

6 Amenazas a la Validez

Como cualquier otro estudio empírico, el presente estudio de caso enfrenta amenazas a la validez. La mayor amenaza para este estudio es la validez externa. La elección de los programas usados en el estudio de caso, potencialmente puede afectar la validez externa, es decir el grado en el que es posible generalizar los resultados obtenidos. Para los objetivos del estudio de caso se necesitaba contar con programas pequeños y con un dominio conocido y con un conocimiento de sus defectos. Para poder generalizar los resultados se requiere hacer más experimentos con otros programas de variados tamaños y complejidad.

En lo que respecta a la validez interna, una posible fuente de sesgo es la cantidad, el tipo de defectos sembrados en los programas y en qué lugar del código son sembrados. Más estudios deben realizarse para reducir esta amenaza. Otra posible fuente de sesgo es la configuración de las herramientas. Para mitigarla se utilizó la configuración que ha reportado más efectiva en estudios anteriores.

7 Conclusiones y Trabajo Futuro

La generación automática de casos de prueba ha hecho grandes avances en la última década, pero aún enfrenta importantes problemas. El presente estudio de caso tiene como objetivo explorar la utilidad desde el punto de vista de la cobertura y de los defectos detectados de las pruebas generadas por dos herramientas *Evosuite y Rando-*

op. El estudio de caso está centrado en dos programas pequeños que se ha utilizado en otros experimentos y a los que se les han sembrado tipos de defectos reportados como más frecuentes durante la codificación.

Para lograr el objetivo del estudio se plantearon tres preguntas de investigación. La pregunta de investigación PI1 trata de identificar cuántos casos de prueba generan las herramientas, la pregunta PI2 plantea cuál es la cobertura lograda por cada una de las herramientas y la tercera pregunta de investigación PI3, explora qué cantidad de defectos son detectados por las herramientas.

Comparado los resultados obtenidos para las preguntas de investigación, se puede afirmar que *Evosuite* a pesar de que generó menos casos de prueba que *Randoop,* logró una mayor cobertura de rama. Para una de las herramientas, *Evosuite*, los defectos en el código hacen que los casos de prueba generados logren una menor cobertura de rama. No sucede lo mismo con *Randoop* que en uno de los programas alcanza una cobertura menor sin los defectos sembrados.

El objetivo general del presente estudio de caso es investigar la utilidad de las herramientas. De los resultados del estudio se puede concluir que se obtuvo una baja cobertura en el código con los defectos sembrados y la efectividad en la detección de defectos es baja. Esta baja efectividad se debe principalmente al problema del oráculo de prueba. El mecanismo por el cual estas herramientas generan los oráculos de prueba se debe continuar investigando.

Las herramientas resultaron ser muy amigables comparando su usabilidad con otras herramientas. Las dos se integran al entorno de desarrollo integrado de Eclipse y sus parámetros son fácilmente modificables.

Como trabajo futuro se plantea comparar las herramientas con respecto a los objetivos del presente estudio de caso, con otras herramientas que implementan otras técnicas como *Palus* (ejecución simbólica) y *DSC* (ejecución simbólica dinámica), así como incorporar otros programas que permitan generalizar las conclusiones.

Referencias

1. A. Bertolino, “Software Testing Research: Achievements , Challenges , Dreams,” in *Future of Software Engineering. FOSE ’07*, 2007, pp. 85–103.
2. G. Quintana and M. Solari, “A systematic mapping study on experiments with automatic structural test case generation,” in *Informatica (CLEI), XXXVIII Conferencia Latinoamericana En*, 2012, pp. 1 – 10.
3. J. Duran and S. Ntafos, “An evaluation of random testing,” *Softw. Eng. IEEE Trans.*, no. 4, pp. 438–444, May 1984.
4. C. S. Păsăreanu and W. Visser, “A survey of new trends in symbolic execution for software testing and analysis,” *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 4, pp. 339–353, Aug. 2009.
5. K. Sen, “Concolic testing,” *Proc. twenty-second IEEE/ACM Int. Conf. Autom. Softw. Eng. - ASE ’07*, p. 571, 2007.
6. P. Mcminn and R. Court, “Search-based Software Test Data Generation: A Survey,” *Softw. Testing, Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, Jun. 2004.
7. G. Fraser and A. Arcuri, “1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite,” *Empir. Softw. Eng.*, 2013.

8. J. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.
9. S. Anand, C. S. Păsăreanu, and W. Visser, “JPF–SE: A Symbolic Execution Extension to Java PathFinder,” in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2007, pp. 134–138.
10. M. Harman, P. Mcminn, and R. Court, “A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search,” *Softw. Eng. IEEE Trans.*, vol. 36, no. 2, pp. 226–247, Mar. 2010.
11. S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, “Application of genetic algorithms to software testing,” in *5th International Conference on Software Engineering and its Applications*, 1992, pp. 625– 636.
12. P. McMinn, “Search-Based Software Testing: Past, Present and Future,” *2011 IEEE Fourth Int. Conf. Softw. Testing, Verif. Valid. Work.*, pp. 153–163, Mar. 2011.
13. C. Pacheco, S. Lahiri, and T. Ball, “Finding errors in. net with feedback-directed random testing,” in *Proceedings of the international symposium on Software testing and analysis*, 2008, pp. 87–96.
14. I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, “Experimental assessment of random testing for object-oriented software,” *Proc. 2007 Int. Symp. Softw. Test. Anal. ISSTA 07*, vol. 127, p. 84, 2007.
15. R. Ramler, D. Winkler, and M. Schmidt, “Random Test Case Generation and Manual Unit Testing: Substitute or Complement in Retrofitting Tests for Legacy Code?,” in *Software Engineering and Advanced Applications (SEAA), 38th EUROMICRO Conference on*, 2012, pp. 286–293.
16. A. Bacchelli, P. Ciancarini, and D. Rossi, “On the Effectiveness of Manual and Automatic Unit Test Generation,” in *The Third International Conference on Software Engineering Advances*, 2008, pp. 252–257.
17. T. Basso and R. Moraes, “An investigation of java faults operators derived from a field data study on java software faults,” *Work. Testes e Tolerância a Falhas*, pp. 1–13, 2009.
18. J. Duraes and H. Madeira, “Emulation of software faults: a field data study and a practical approach,” *Softw. Eng. IEEE Trans.*, vol. 32, no. 11, pp. 849–867, 2006.
19. K. Lakhotia, P. McMinn, and M. Harman, “An empirical investigation into branch coverage for C programs using CUTE and AUSTIN,” *J. Syst. Softw.*, vol. 83, no. 12, pp. 2379–2391, Dec. 2010.
20. K. Sen, D. Marinov, and G. Agha, “CUTE: a concolic unit testing engine for C,” in *ESEC/FSE-13 Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005, vol. 30, pp. 263–272.
21. K. Lakhotia, M. Harman, and H. Gross, “AUSTIN: A Tool for Search Based Software Testing for the C Language and Its Evaluation on Deployed Automotive Systems,” *2nd Int. Symp. Search Based Softw. Eng.*, pp. 101–110, Sep. 2010.
22. X. Xiao, “Problem Identification for Structural Test Generation: First Step Towards Cooperative Developer Testing,” in *Software Engineering (ICSE), 33rd International Conference on*, no. 1, pp. 1179–1181.
23. K. Petersen and M. V. Mantyla, “Benefits and limitations of automated software testing: Systematic literature review and practitioner survey,” *7th Int. Work. Autom. Softw. Test*, pp. 36–42, Jun. 2012.
24. A. Leitner, I. Ciupa, B. Meyer, C.- Zürich, and M. Howard, “Reconciling Manual and Automated Testing: the AutoTest Experience,” in *40th Annual Hawaii International Conference on System Sciences (HICSS’07)*, 2007, pp. 1–10.

25. C. Pacheco and M. Ernst, "Eclat: Automatic generation and classification of test inputs," *ECOOP Object-Oriented Program.*, 2005.
26. Y. Pavlov and G. Fraser, "Semi-automatic Search-Based Test Generation," in *IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 777–784.
27. G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," in *SIGSOFT FSE*, 2011, pp. 416–419.
28. G. Fraser and A. Arcuri, "Sound Empirical Evidence in Software Testing," in *Proceedings of the International Conference on Software Engineering*, 2012, no. September, pp. 178–188.
29. G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does Automated White-Box Test Generation Really Help Software Testers?," in *ISSTA*, 2013.
30. I. Prasetya, T. Vos, and A. Baars, "Trace-based reflexive testing of OO programs with T2," in *1st Int. Conf. on Software Testing, Verification, and Validation (ICST)*, 2008, pp. 151–160.
31. Mainul Islam and C. Csallner, "Dsc+Mock: A Test Case + Mock Class Generatorin Support of Coding Against Interfaces," in *Proceedings of the Eighth International Workshop on Dynamic Analysis*, 2010, pp. 26–31.
32. C. Pacheco and M. Ernst, "Randoop: feedback-directed random testing for Java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.
33. G. Fraser and A. Arcuri, "EvoSuite at the SBST 2013 Tool Competition," *evosuite.org*.
34. F. Gross, G. Fraser, and A. Zeller, "Exploring Realistic Program Behavior," Technical report, Saarland University, Submitted for publication, 2012.
35. P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empir. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, Apr. 2009.
36. V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, 1994, vol. 2, no 1994, p. 528-532.
37. R. Chillarege, "Orthogonal defect classification," *Handb. Softw. Reliab. Eng.*, 1996.
38. C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Softw. Pr. Exper.*, vol. 34, pp. 1025–1050, 2004.
39. L. Baresi, P. L. Lanzi, and M. Miraz, "TestFul: An Evolutionary Test Approach for Java," *Third Int. Conf. Softw. Testing, Verif. Valid.*, pp. 185–194, 2010.
40. K. Inkumsah and T. Xie, "Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution," *2008 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pp. 297–306, Sep. 2008.
41. T. E. J. Vos, B. Marínt, M. J. Escalona, and A. Marchetto, "A methodological framework for evaluating software testing techniques and tools," in *12th International Conference on Quality Software, QSIC 2012*, 2012, vol. Xi'an, Sha, pp. 230–239.
42. J. Edvardsson, "Techniques for Automatic Generation of Tests from Programs and Specifications," Doctoral dissertation, Linköping. 2006.
43. A. Arcuri and G. Fraser, "On Parameter Tuning in Search Based Software Engineering," *Search Based Softw. Eng.*, 2011.
44. G. Fraser and A. Arcuri, "Whole Test Suite Generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276 – 291, Feb. 2012.
45. F. Pastore, L. Mariani, and G. Fraser, "CrowdOracles: Can the Crowd Solve the Oracle Problem," *2013 IEEE Sixth Int. Conf. Softw. Testing, Verif. Valid.*, 2013.