Universidad ORT Uruguay

Facultad de Ingeniería

Mapeo Sistemático y Estudio de Caso sobre Técnicas de Generación Automática de Pruebas

Entregado como requisito para la obtención del título de Master en Ingeniería

Gerardo Quintana - 98649

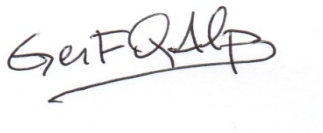
Tutor: Martín Solari

2013

Declaración de Autoría

Yo, Gerardo Quintana, declaro que el trabajo que se presenta en esta obra es de nuestra propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Master en Ingeniería;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mí;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Gerardo Quintana – 15/08/2008

Abstract

La automatización de la generación de los casos de prueba puede tener un impacto significativo en la eficacia y la eficiencia de las pruebas de software. Para reducir el trabajo manual de crear datos de prueba los profesionales pueden usar técnicas automáticas. En la literatura se reportan muchos estudios que evalúan nuevas técnicas para la generación automática de casos de prueba estructurales. Si bien se han desarrollado herramientas para la generación automática de casos de prueba que pueden alcanzar una alta cobertura, la adopción de estas por parte de la industria es limitada. Para la transferencia exitosa de estas técnicas de la academia a la industria, es esencial que la comunidad de investigadores continúe resolviendo muchos de los desafíos que quedan por delante.

Esta tesis tiene como objetivo explorar la generación automática de casos de prueba a través de la identificación de las técnicas y sus problemas e investigar su utilidad práctica. Se realizó un mapeo sistemático de la literatura, extendiendo dos trabajos relacionados a la tesis, para conocer las técnicas y los problemas investigados. A partir de este mapeo, se realizó un estudio de caso con herramientas de generación para evaluar su eficacia con respecto a la detección de defectos.

Se concluye que las técnicas que tienen más estudios son las basadas en búsqueda mediante algoritmos genéticos, que se combinan distintos enfoques para incrementar la eficiencia de la generación y que los problemas con más estudios son: generación de datos de prueba, reducción de los casos de prueba y técnicas para tratar con estructuras complejas de programas. Por otra parte, el estudio de caso con las herramientas muestra baja cobertura y que no son eficaces para la detección de defectos reales.

Palabras clave

Pruebas de software, generación automática de casos de prueba, pruebas unitarias, calidad, mapeo sistemático de la literatura, estudio de caso.

Índice

1. Introducción

1.1 Área de investigación

Dentro de la ingeniería de software, la calidad es uno de los factores más importantes que determinan el éxito o el fracaso de un producto de software, junto con el costo, la planificación y la funcionalidad [1]. De los factores mencionados anteriormente, costo, planificación y funcionalidad, **esta investigación se focaliza en la calidad de los sistemas de software.**

En la actualidad los sistemas de software forman parte de nuestra vida. Dado que estos sistemas han crecido en tamaño y complejidad, asegurar una alta calidad del software es cada vez más desafiante y costoso. Un enfoque para incrementar la calidad de los productos de software consiste en disminuir la cantidad de defectos. De acuerdo a la forma en que los defectos son tratados, existen distintas actividades de aseguramiento de la calidad que se pueden clasificar en tres categorías generales [1]:

- Prevención de defectos a través de la eliminación de errores en el código fuente y actividades de bloqueo de errores, tales como la educación y entrenamiento, especificaciones formales y verificación, y la selección y aplicación apropiada de tecnologías, herramientas, procesos, o estándares.
- Reducción de defectos a través de inspecciones, pruebas de software, y otras actividades estáticas o dinámicas, para detectar y quitar fallos del software.

- Contención de defectos a través de la tolerancia a fallos, prevención de fallos, o minimización de impacto de fallo, para asegurar la fiabilidad y la seguridad del software.

Dentro de la segunda categoría de la clasificación anterior, las pruebas de software son una de las alternativas más importantes y más usadas [1]. Las pruebas de software implican la ejecución del software y la observación del comportamiento del programa o la salida. El proceso de pruebas de software consiste en la evaluación dinámica del comportamiento de un programa en un conjunto finito de casos de prueba, debidamente seleccionados desde el dominio de ejecución (por lo general infinito), contra el comportamiento esperado [2]. Este proceso se usa ampliamente en la industria para asegurar la calidad del software. **Esta investigación aborda la calidad del software a través de las pruebas de software.**

1.2 Automatización de las pruebas de software

La historia de la investigación de la ingeniería de software ha visto el surgimiento continuo de paradigmas nuevos de desarrollo, que intentan liberar software con mayor calidad y con menos costos [3]. Las pruebas de software son una de las formas más importantes para asegurar la calidad del software, pero son consideradas como uno de los procesos más costosos del desarrollo. Algunos estudios estiman que los defectos en el software tienen un costo anual aproximado de US$ 59.5 mil millones en la economía de los EE.UU. [4].

Las pruebas de software pueden ser realizadas manualmente o automáticamente. La forma más tradicional de realizar las pruebas de software es manual, pero ambos

enfoques son complementarios. Con la automatización de las pruebas el costo del desarrollo puede disminuir significativamente. **Esta investigación se focaliza en la automatización de las pruebas de software.**

Las pruebas de software pueden verse como una secuencia de tres pasos fundamentales [5]:

1. El diseño de casos de prueba que sean efectivos para descubrir fallos en el software, o que al menos sean adecuados a un criterio de adecuación de prueba.
2. La ejecución de los casos de prueba diseñados.
3. La determinación de si el resultado es el correcto.

En la práctica actual, por lo general el único paso totalmente automatizado es el paso 2 [5]. La generación de casos de prueba es el paso más complejo y normalmente requiere la intervención de un experto. Su automatización puede tener un impacto significativo en las pruebas de software. Para reducir el trabajo manual de crear datos de prueba los profesionales pueden usar técnicas automáticas. **Esta investigación se focaliza en la generación automática de casos de prueba.**

1.3 El problema de la tesis

Se necesita realizar más estudios acerca de los beneficios y limitaciones de la automatización de las pruebas automáticas [6]. Los investigadores buscan desarrollar técnicas y herramientas eficientes que pueden ser usadas por los profesionales para probar el software [7]. La investigación en distintas áreas de las pruebas de software ha proporcionado avances que son prometedores para ayudarnos a alcanzar el objetivo de

proporcionar herramientas prácticas que puedan ayudar a los profesionales a desarrollar software de alta calidad. Se ha propuesto una hoja de ruta para las pruebas de software que conduce al destino de proveer métodos prácticos para las pruebas, herramientas y procesos para desarrollar software de alta calidad [7]. Para llegar a ese destino se requiere realizar investigación fundamental en algunas áreas específicas de las pruebas, por ejemplo demostrar la eficacia de las técnicas de prueba de software, la creación de nuevos métodos y herramientas, y la ejecución de estudios empíricos para facilitar la transferencia de la tecnología a la industria. Si bien la investigación de la generación automática de casos de prueba ha sido muy activa y existen avances en el área, es de destacar que todavía todo el esfuerzo ha producido un impacto limitado en la industria y la actividad de la generación de casos de prueba mayormente se realiza en forma manual [3]. Si bien existen múltiples técnicas que los profesionales pueden usar para la generación de automática de casos de prueba, muchas de estas técnicas son aplicables para las pruebas unitarias y puede que no sean escalables en software industrial [7].

Los casos de prueba se pueden generar principalmente a partir de las especificaciones o a partir del código [8]. Las pruebas de software en el primer caso son llamadas funcionales o de caja negra, mientras que en el último caso son llamadas estructurales o de caja blanca. Las técnicas de generación de casos de prueba estructurales analizan las estructuras de flujo o control de datos desde el código fuente y las utilizan para extraer la información necesaria para la generación de casos de prueba [9]. Dado un criterio de cobertura como por ejemplo cobertura de camino, cobertura de sentencia o cobertura de rama, el problema para la generación de casos de prueba estructurales es encontrar un conjunto de entradas que alcancen la cobertura dada [10]. Este problema en general es

indecidible, principalmente debido a las muchas posibles combinaciones de entrada de un programa [11]. Un problema indecidible es aquel para el que no es posible construir un algoritmo que para cada entrada correcta termina y da una respuesta "sí", si la entrada tiene la propiedad requerida y responde "no" en otro caso [12]. **Esta investigación se focaliza en técnicas que automáticamente generen casos de prueba estructurales.**

La **Figura 1** ilustra en forma jerárquica el enfoque general de la tesis.

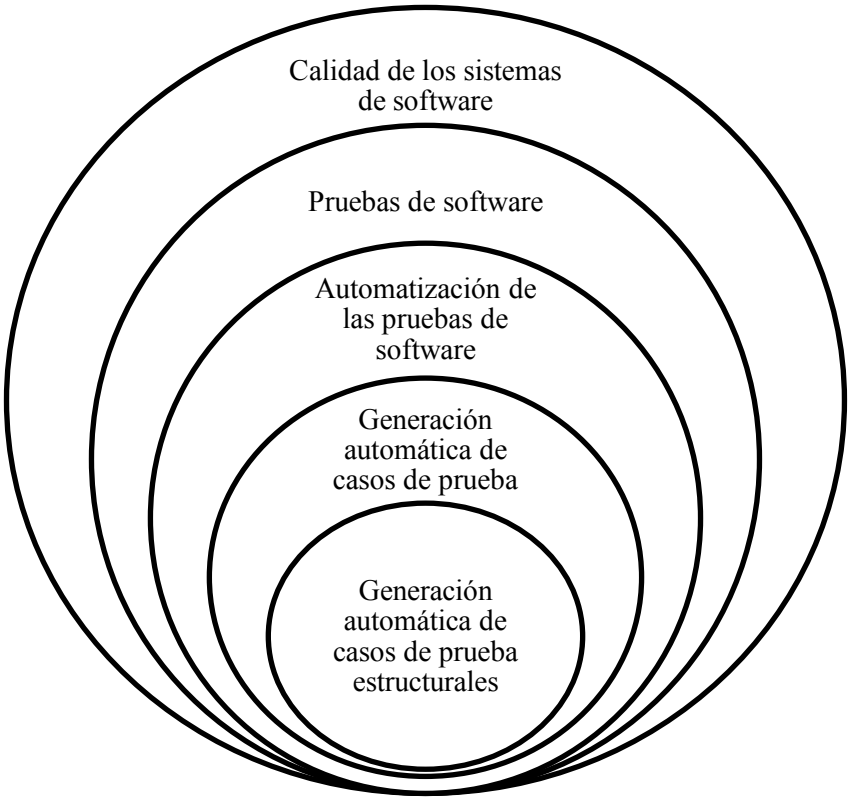


Figura 1: Jerarquía de conceptos del enfoque de la tesis.

Existen varios enfoques para la generación automática de casos de prueba. Uno de los más comunes que es encontrado en la literatura es la generación aleatoria que consiste en generar aleatoriamente entradas hasta que se genere una que cumpla con el criterio

de búsqueda. Otro enfoque consiste en la generación basada en métodos heurísticos o computación evolutiva. En este enfoque se utilizan algoritmos de optimización meta-heurísticos para la generación de datos y se llaman pruebas de software basadas en búsqueda (del inglés *Search Based Software Testing: SBST*). **Una parte de esta investigación, en particular un estudio de caso para investigar la utilidad de las técnicas de generación se focaliza en pruebas de software basadas en búsqueda.**

1.4 Objetivos

El objetivo general de la tesis es explorar la generación automática de casos de prueba, a través de la identificación y clasificación de las técnicas y sus principales problemas e investigar su utilidad práctica con respecto a la detección de defectos en el software.

En la literatura existen numerosas propuestas de técnicas para la generación de casos de prueba. Muchas de estas técnicas son aplicables para pruebas unitarias y puede que no sean escalables para sistemas grandes. Si bien aún persisten algunos problemas se han obtenido resultados prometedores para lograr obtener herramientas prácticas para ayudar a desarrollar software de alta calidad.

Desde la perspectiva de la investigación la automatización de las pruebas de software es un área madura de la investigación. Aunque, al mismo tiempo, se reconoce que existe una brecha entre la investigación académica y los beneficios y los problemas que encuentran para aplicar efectivamente las pruebas de software en la industria [6]. Los principales beneficios percibidos de la automatización de las pruebas de software son la reusabilidad, repetitividad, el esfuerzo ahorrado en las ejecuciones de prueba y mejor

cobertura de código. En cuanto a las limitaciones la automatización las herramientas necesitan ser más eficientes en la creación de casos de prueba y su mantenimiento.

Para lograr los objetivos en esta tesis se sigue el enfoque de la la ingeniería de software basada en evidencia [13] (del inglés *Evidence-based software engineering*), que tiene como objetivo mejorar la toma de decisiones relacionadas con el desarrollo y mantenimiento del software mediante la integración de la mejor evidencia actual de la investigación con la experiencia práctica. Los profesionales pueden tomar decisiones incorrectas si no consideran la evidencia científica acerca de la eficacia de las técnicas [13]. Para determinar qué técnicas pueden ser aplicadas en la práctica, se requiere realizar estudios empíricos para determinar su eficacia y su eficiencia para lograr un determinado objetivo de prueba a un costo determinado.

El objetivo general de la tesis se divide en los siguientes objetivos específicos:

1. Identificar y clasificar las técnicas de generación automática de casos de prueba.
2. Identificar los problemas investigados con las técnicas.
3. Explorar mediante un estudio de caso la efectividad para detectar defectos de las herramientas que implementen técnicas de generación automática de prueba.

1.5 Contribuciones de la tesis

Las siguientes son las principales contribuciones de la tesis:

1. Un estudio de mapeo sistemático sobre técnicas de generación automática de pruebas estructurales que tiene como objetivo la clasificación de las técnicas y la identificación de qué problemas son investigados con ellas.
2. Un estudio de caso con algunas herramientas basadas en búsqueda para explorar la utilidad de éstas para la detección de defectos en el software. Se busca aportar nuevo conocimiento a lo reportado en la literatura sobre qué casos de prueba generan las herramientas y qué defectos se pueden detectar con ellas.

1.6 Estructura de la tesis

El capítulo 2 comienza realizando una serie de definiciones necesarias para la correcta comprensión de los tema tratados en el resto de la tesis. El resto del capítulo 2 presenta el estado de la cuestión relacionado con la generación automática de pruebas de software estructurales, mostrando el marco conceptual en el que se desarrolla la investigación, e identifica los problemas y las oportunidades.

El capítulo 3 describe el método general de investigación detallando el flujo de trabajo que se sigue. Además se presentan los dos principales estudios que se llevarán a cabo junto con la motivación de los mismos y la descripción del método a seguir.

El capítulo 4 presenta el mapeo sistemático de la literatura realizado para estudiar las técnicas de generación automática de casos de prueba con las que se han realizado experimentos. Además se presenta una clasificación de las técnicas. En este capítulo se detalla la preparación del estudio, los resultados del mismo y la correspondiente discusión de ellos.

El capítulo 5 describe un estudio de caso con herramientas de prueba basadas en búsqueda. En este capítulo se describe la preparación y la ejecución del estudio de caso, los resultados obtenidos y la discusión realizada sobre sus resultados.

En el capítulo 6 se lleva a cabo el último paso que consiste en una discusión general acerca de los resultados obtenidos en ambos estudios y trabajo futuro. Dicha discusión se encuentra alineada a los objetivos específicos que llevarán a la concreción del objetivo general planteado.

2. Estado de la cuestión

Hay muchos conceptos que se utilizan frecuentemente en la investigación de la generación automática de casos de prueba. Este capítulo comienza definiendo los principales términos para una correcta comprensión de los aportes de la tesis. Recientemente surgieron nuevos enfoques para la generación automática de casos de prueba. En este capítulo se muestran algunas de las divisiones de los enfoques y la progresión de éstos a través del tiempo.

2.1 Definiciones

2.1.1 Sistema generador de datos

Los casos de prueba son un conjunto de entradas de prueba, condiciones de ejecución y resultados esperados desarrollados para un objetivo en particular, como por ejemplo ejercitar una ruta de programa en particular o para verificar el cumplimiento de un requerimiento específico [14]. Un conjunto de casos de prueba es llamado *Test Suite* [15]. La generación de datos es el proceso de identificación de datos de entrada que satisfacen los criterios de prueba seleccionados [16]. Los datos de prueba son los datos específicamente identificados para usar en las pruebas de software [15]. Un oráculo de prueba es una fuente confiable del comportamiento esperado del software que proporciona salidas para cualquier entrada especificada en las especificaciones del software y un comparador para verificar los resultados reales [17].

Habitualmente la generación automática de datos de prueba es llamada en inglés *automatic test data generation (ATDG)*. Estrictamente hablando en *ATDG* solo se generan los datos de entrada sin un oráculo [18]. Cabe señalar que el concepto de generación automática de datos de prueba y la generación automática de casos de prueba han sido utilizados indistintamente en las investigaciones reportadas en la literatura.

Una arquitectura posible de un sistema generador de datos de prueba consiste en tres partes (**Figura 2**): **analizador de programa, selector de caminos** y **generador de datos** [19].

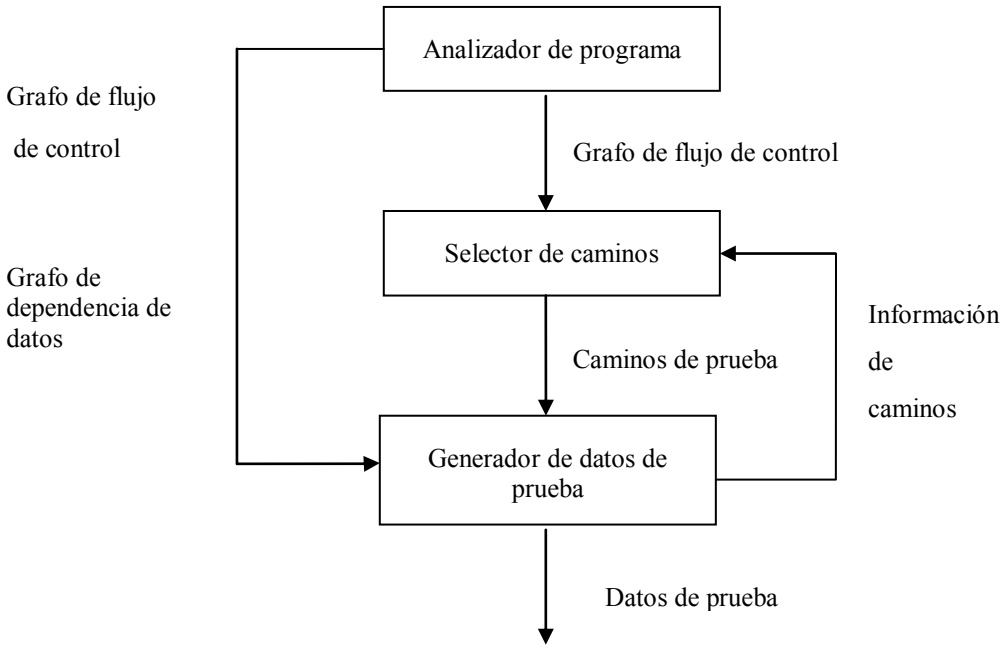


Figura 2: Arquitectura de un sistema generador de datos de prueba.

En un sistema generador de casos de prueba el código fuente se corre en un analizador de programa que genera un **grafo de flujo de control** que puede ser usado por el **selector de caminos** y el **generador de datos de prueba**. El **selector de caminos** inspecciona los datos del programa para encontrar caminos que satisfagan el criterio de

prueba seleccionado. Los caminos son proporcionados como entrada al generador de datos el cual genera los datos de prueba que ejercitan los caminos dados. El generador puede proveer al selector con retroalimentación, tales como la información de relativa a los caminos no factibles [19]. Un camino se dice que es factible si existe una entrada de programa que atraviese el camino, de lo contrario el camino es no factible [20].

En la literatura existen diferentes arquitecturas de los sistemas generadores de casos de prueba. Con base en la técnicas de implementación se pueden diferenciar las siguientes arquitecturas de los sistemas [15]: i) arquitectura basada en ejecución real (**Figura 2**), ii) arquitectura basada en ejecución simbólica, iii) arquitectura basada en ejecución simbólica y ejecución real y iv) arquitectura basada en enfoque orientado a objetos.

El modelo de la arquitectura basada en ejecución simbólica (**Figura 3**) utiliza el concepto de ejecución simbólica como lo indica su nombre. 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 [21]. La arquitectura basada en esta técnica consiste en tres partes: **selector de caminos**, **generador de restricciones y solucionador de restricciones**. El **selector de caminos** genera un conjunto de caminos desde el programa bajo prueba para satisfacer algún criterio. Uno de los criterios es cobertura de sentencia en el que se requiere seleccionar un conjunto de caminos de tal forma que todas las sentencias sean cubiertas al menos una vez. Luego que el **selector de caminos** genera los caminos, el **generador de restricciones** crea las restricciones ya sea del desde el programa bajo prueba directamente o desde los caminos generados por el **selector de caminos**. La técnica de ejecución simbólica es aplicada para buscar las

restricciones. La salida del **generador de restricciones** son un **conjunto de restricciones de camino**. Cada restricción de camino es un conjunto de igualdades y desigualdades sobre las variables de entrada y un conjunto de valores que satisfacen estas restricciones, que son el conjunto de datos de prueba requerido para el camino respectivo [15].

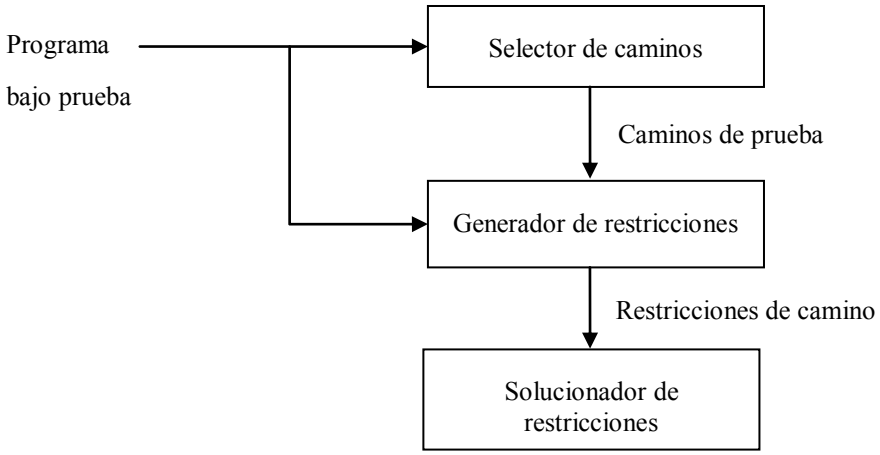


Figura 3: Arquitectura de un sistema generador de datos de prueba basada en ejecución simbólica.

La arquitectura basada en ejecución real y ejecución simbólica (**Figura 4**) consiste de un **analizador de programa,** un **instrumentador de código,** un **filtro** y un **comparador.** El **instrumentador de código** inserta líneas e forma estática al código fuente para mostrar los caminos y el **filtro** provee los únicos caminos factibles. El **comparador** compara el número de caminos en la iteración actual y la anterior para terminar con el número mínimo de iteraciones ya sea para cubrir el criterio de camino más largo o todos los caminos [15]. Por más detalles de esta arquitectura se puede consultar otros artículos [15], [22].

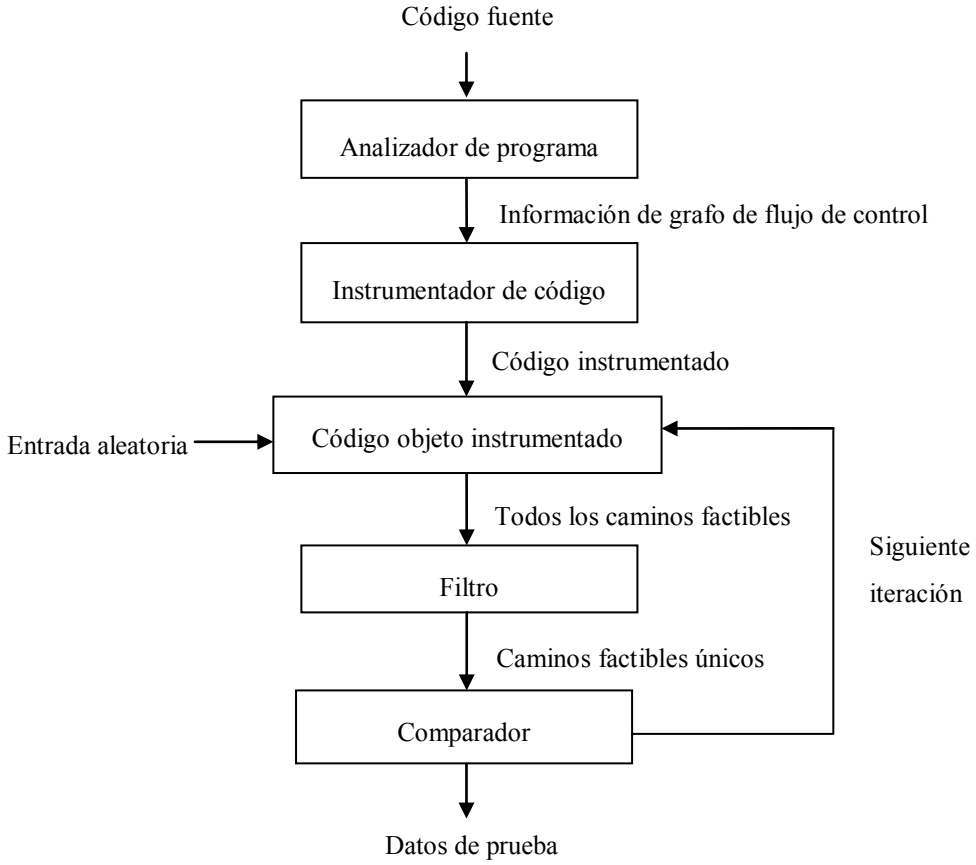


Figura 4: Arquitectura de un sistema generador de datos de prueba basada en ejecución real y ejecución simbólica

Por último, la arquitectura basada en enfoque orientado a objetos (**Figura 5**) genera los datos a partir de componentes que son modelos UML del sistema bajo prueba. Este tipo de pruebas son basadas en la especificación y no están en el alcance de la tesis.

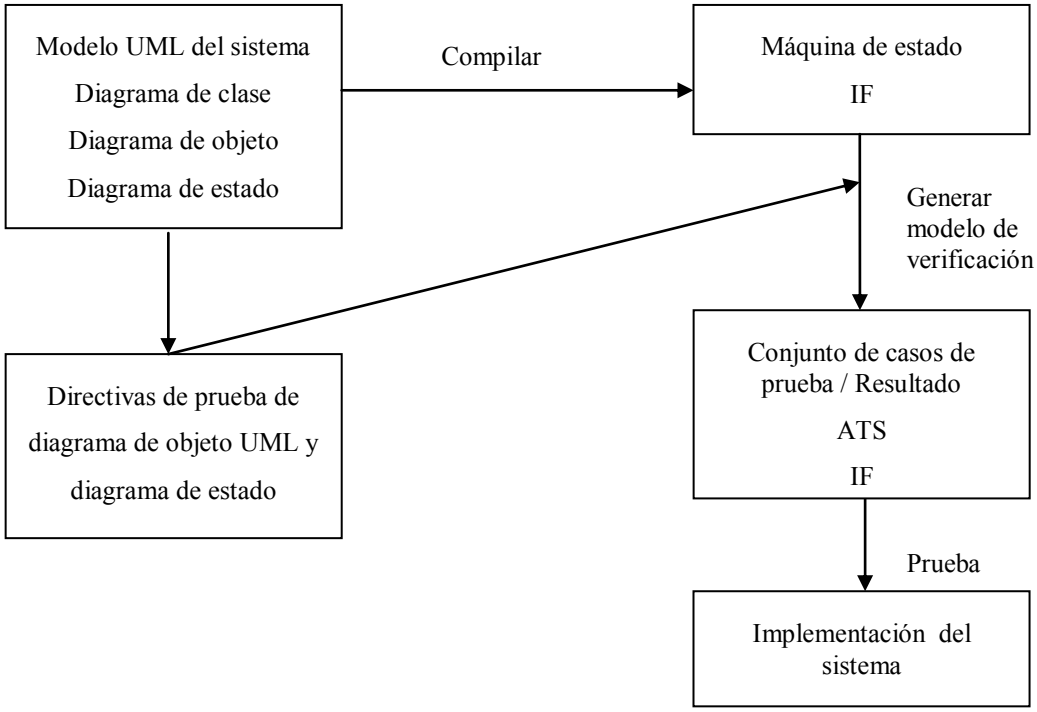


Figura 5: Arquitectura de un generador de datos de prueba basado en enfoque orientado a objetos.

2.1.2 Técnicas para la generación automática

Existe más de un enfoque para la generación de casos de prueba. Una división posible de estos enfoques es la siguiente [23]: aleatorio, orientado al objetivo y orientado al camino. Existe también otro enfoque llamado inteligente [24].

La generación aleatoria consiste en generar aleatoriamente entradas hasta que una entrada valiosa sea generada [16]. El enfoque orientado al camino es el proceso de selección de una o más rutas de programa hasta la instrucción seleccionada y luego generar datos de entrada para el o los caminos seleccionados [16]. El enfoque orientado al objetivo es el proceso de generar datos de prueba para ejecutar la instrucción seleccionada independientemente del camino tomado [23]. En este enfoque el paso de la

selección del camino es eliminado. Otro enfoque posible es llamado inteligente y se basa en un análisis sofisticado del código para guiar la búsqueda de nuevos datos de prueba [24]. Este enfoque no ha sido muy extendido y ha sido reportado por pocos investigadores [24], [25], [26], [8].

Existe además otra clasificación de los enfoques dependiendo de si requiere o no la ejecución del programa para obtener los datos de prueba, por lo que se clasifican en dinámicos (usan ejecución real) y estáticos (usan ejecución simbólica) respectivamente [19]. Los métodos estáticos generalmente usan ejecución simbólica para obtener restricciones sobre las variables de entrada para un criterio de prueba particular. En contraste a los métodos estáticos, los métodos dinámicos requieren que el software bajo prueba sea ejecutado. Los métodos dinámicos en vez de usar sustitución de variables implican una búsqueda directa de los datos de prueba que cumplen con el criterio deseado [8]. El costo computacional involucrado en la ejecución repetida puede tener un alto costo computacional aplicado a software complejo con un gran espacio de parámetros.

Cada uno de los enfoques mencionados anteriormente (aleatorio, orientado al objetivo, orientado al camino e inteligente) puede ser implementado en forma estática o dinámica [27].

Existen también los métodos híbridos que combinan generación estática y dinámica [27] [15]. Estos métodos intentan obtener lo mejor de los métodos que combinan. Por ejemplo, un prototipo de una herramienta llamada *PathCrawler* [28] tiene como objetivo satisfacer el criterio de cobertura llamado *all-paths*. Los enfoques estáticos para la generación de casos de prueba tienen problemas en la detección de caminos no

factibles, en particular en el caso de bucles con un número variable de iteraciones [28]. Los métodos dinámicos evitan los problemas de los métodos estáticos mediante el uso de técnicas como la minimización de funciones de tal forma que el objetivo es cubierto. La combinación de métodos estáticos y dinámicos ha sido utilizada en otras herramientas como *CUTE* [29] y *Palus* [30].

Mahmood [27] brinda una detallada clasificación de las técnicas (ver **Figura 6**), dividiéndolas en dos grandes categorías: funcionales (basadas en la especificación o de caja negra) y estructurales (de caja blanca).

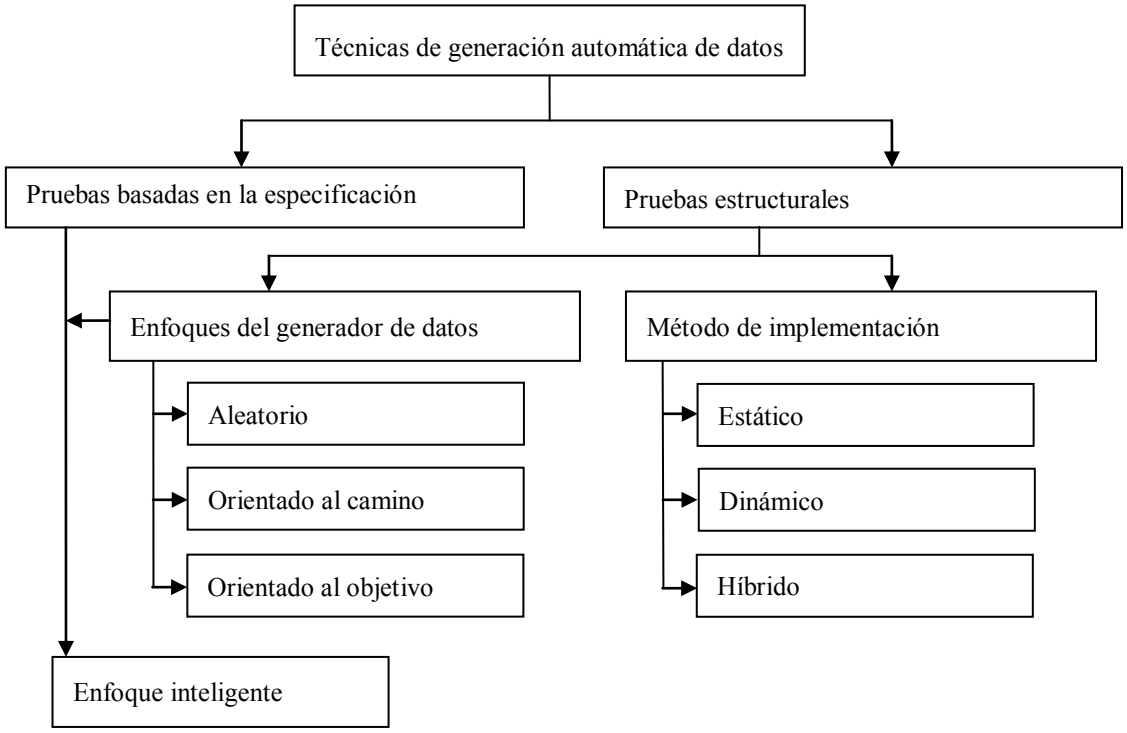


Figura 6: Enfoques para la generación automática de casos de prueba. Basado en la división propuesta por Mahmood [27]**.**

La clasificación de las técnicas incluye además una división según el enfoque del generador de datos, conteniendo el enfoque inteligente y una división según el método de implementación.

2.1.2.1 Pruebas basadas en búsqueda

En los años recientes hubo un creciente interés por los enfoques para la generación basados en algoritmos meta-heurísticos o computación evolutiva. Este enfoque es conocido como pruebas de software basadas en búsqueda (*SBST*). Los algoritmos meta-heurísticos combinan varios métodos heurísticos con el fin de encontrar soluciones a los problemas computacionalmente difíciles. Mahmood [27] clasificó las técnicas de *SBST* con la división presentada en la **Figura 6**. El trabajo de Mahmood [27] consiste en una revisión sistemática en la que clasifica con distintos criterios todas las técnicas en el período 1997 al 2006 incluyendo las de *SBST*, mostrando que las técnicas de *SBST* pueden utilizar enfoques de generación orientados al camino y al objetivo e implementadas en forma estática, dinámica o híbrida. Una extensión a la revisión de Mahmood [27] en términos de los años cubiertos y los detalles de las técnicas fue presentada por Romli *et al.* [31]. En esta revisión, Romli *et al.* [31] clasifica las técnicas en el período 1976 al 2010, incluyendo el tipo de técnica (funcional o estructural), el método de implementación (estático, dinámico e híbrido), sin considerar el enfoque del generador, la técnica para la generación de datos e identificando para cada una si usa algoritmos meta-heurísticos.

En la ingeniería de software basada en búsqueda, el término 'búsqueda' es usado para referirse a las técnicas de optimización de búsqueda meta-heurísticas que utiliza [18].

La ingeniería de software basada en búsqueda trata de reformular los problemas de la ingeniería de software como problemas de optimización basados en búsqueda.

Las pruebas basadas en búsqueda 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. La definición de una función de adecuación, que es una guía para encontrar buenas soluciones en un espacio de búsqueda infinito, es clave para usar un algoritmo meta-heurístico [32]. Las técnicas meta-heurísticas han sido aplicadas para automatizar la generación de datos para pruebas estructurales y funcionales, las pruebas de propiedades de caja gris (por ejemplo restricciones de seguridad) y también para propiedades no funcionales (como por ejemplo el peor tiempo de ejecución de un segmento de código) [20].

Muchos algoritmos se han utilizado en las pruebas basadas en búsqueda, incluyendo algoritmos evolutivos paralelos [33], *scatter search* [34], *tabu search* [35], *particle swarm optimization (PSO)* [36], *hill climbing* [37] y s*imulated annealing* [38]. Por lejos las técnicas de búsqueda más populares usadas en *SBST* pertenecen a la familia de los algoritmos evolutivos en lo que se conoce como pruebas evolutivas [39]. 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 [40]. Estos algoritmos usan una búsqueda global (más comúnmente, pero no exclusivamente implementadas con un Algoritmo Genético). Los algoritmos evolutivos son considerados como una forma de búsqueda global porque ejercitan varios puntos a la vez en el espacio de búsqueda. En

cambio, *hill climbing* es descrito como búsqueda local porque considera solo una solución a la vez y hace movimientos en la vecindad local de esas soluciones.

2.1.2.2 Ejecución simbólica dinámica

Otro enfoque que ha tenido un interés creciente en la comunidad de investigadores es *Dynamic Symbolic Execution (DSE)* [29]. Es un enfoque híbrido que entrelaza ejecución real (*concrete execution)* con ejecución simbólica (*symbolic execution)*. Debido a la combinación de ejecución real (*CONCrete Execution*) con ejecución simbólica (*SymbOLIC Execution*), *DSE* es conocido también como *Concolic Testing* (*CT*). Los programas son ejecutados simultáneamente con valores simbólicos y concretos y las restricciones simbólicas generadas a lo largo de los caminos son simplificadas usando los valores concretos. Las restricciones simbólicas son usadas para generar en forma incremental entradas de datos para obtener una mejor cobertura mediante la combinación de restricciones simbólicas para un prefijo del camino con la negación de un condicional tomado por la ejecución.

2.2 Problemas de la generación automática

Los enfoques para la generación automática de prueba tienen algunas debilidades. Por ejemplo, un generador aleatorio puede crear muchos datos de prueba, pero como la información de los requerimientos de prueba no están incorporados en el proceso de generación, el generador de datos de prueba puede ser ineficiente [24]. Los generadores orientados al camino primero identifican el camino para el cual los datos de prueba van a generarse, pero, dado que el camino puede no ser factible, el generador puede fallar en

encontrar una entrada que atraviese el camino. Los generadores orientados al objetivo generan datos para atravesar un camino no específico. Como estos enfoques utilizan el concepto encontrar cualquier camino, es difícil predecir la cobertura de código dado un conjunto de objetivos [15]. Un enfoque inteligente puede generar nuevos datos de prueba rápidamente si el análisis es preciso. No obstante, el análisis requerido para tener éxito en una gama amplia de programas puede ser muy complejo [24].

Por otra parte, una de las principales críticas que tienen las técnicas estáticas, son el alto costo de computación que tienen. Además, algunas de las restricciones se pueden volver insolubles. Esto suele suceder cuando el código fuente tiene variables del tipo punto flotante o restricciones no lineales. No obstante, las técnicas estáticas pueden ser usadas para probar la ausencia de ciertos tipos de errores, mientras que las pruebas de software pueden solo ser usadas para probar la presencia de errores. Esto hace que las técnicas estáticas indispensables especialmente en sistemas críticos [41].

Independientemente del enfoque para la generación o de la forma de implementar el método de generación, las técnicas enfrentan varios problemas que se detallan en las siguientes secciones.

2.2.1 Manejo del entorno de ejecución

Este problema ocurre con los programas que verifican la existencia de archivos o directorios, que leen o validan archivos. Asimismo, los programas que usan bases de datos tienden a incluir código que realiza acciones tales como abrir una conexión a la base de datos, agregar, actualizar y borrar datos, entre otras acciones [32]. Además, el sistema operativo subyacente puede causar problemas cuando el programa verifica la

cantidad de memoria disponible. El código que involucra accesos de red puede requerir leer o escribir valores a un *socket*, verificar la presencia de servicios, etc. El manejo del entorno de ejecución es reportado como uno de los principales problemas que impidieron la cobertura de muchas de las ramas de los programas bajo prueba en un experimento [42]. Algunos de estos problemas pueden resolverse mediante la generación de datos de prueba que se copian a un archivo o base de datos, para ser leídos de nuevo por el programa bajo prueba [32]. Otra solución es que los desarrolladores enseñen a la herramienta a instrumentar la llamada al método externo [42] o proveer objetos *mock* para simular las dependencias irrelevantes del entorno [32], [42].

2.2.2 Testeabilidad

Otro problema que enfrentan las técnicas para la generación automática de casos de prueba es mejorar la testeabilidad. La testeabilidad es el grado en que un sistema o componente facilita el establecimiento de criterios de evaluación y la realización de pruebas para determinar si se han cumplido esos criterios [14]. Existen algunos constructores de los lenguajes que dificultan la generación de entradas y por lo tanto dificultan la testeabilidad. Por ejemplo, en *SBST* las variables de tipo bandera en predicados de rama provocan problemas en las funciones de adecuación. Una variable de tipo bandera es una variable del tipo *boolean,* y el problema con este tipo de variables se da porque relativamente pocos valores de entrada existen que hagan que la variable adopte uno de sus dos valores posibles [39]. Como consecuencia, las variables de este tipo introducen grandes mesetas en el espacio de búsqueda. La función de

adecuación resultante en estos casos consiste en dos mesetas (una para el valor verdadero y otra para el valor falso). En estas situaciones, la performance de la búsqueda evolutiva no es mejor que una búsqueda aleatoria.

2.2.3 Tipos de datos dinámicos

Los programas que involucran estructuras de datos dinámicas, tales como listas o árboles son problemáticos cuando es necesario determinar el tamaño requerido y la forma para la ejecución de alguna estructura de programa. Por ejemplo, la forma de un árbol está determinada por sus ramas y el número de nodos de cada nivel. Existen algunos trabajos iniciales en esta área que utilizan búsqueda local para adaptar una estructura de datos dinámica introducida para que coincida con la ruta a ser ejecutada [43].

2.2.4 Bucles

Los bucles en el código fuente que no tienen un número constante de iteraciones son problemáticos para la generación [19]. En *SBST* para búsquedas globales, hay problemas que presentan las funciones de adecuación para guiar la búsqueda en estructuras anidadas dentro de bucles [20]. Si bien hay algunos enfoques para tratar el problema, todavía es un problema abierto. Además de los problemas de óptimos locales y mesetas de las funciones de adecuación, existen problemas con los cálculos de distancia de rama ya que en algunos casos puede llevar a abandonar la búsqueda.

2.2.5 Programas orientados a objetos

Existen también problemas para la generación de datos de prueba con los programas orientados a objetos, dado que los objetos están basados en estados. En estos programas hay dificultades para generar los estados deseados de los objetos. Este problema ha sido denominado como el problema de la creación de objetos (del inglés *object creation problem: OCP*) y es uno de los principales problemas que enfrentan las herramientas para la generación de casos de prueba estructurales [44]. La principal razón por la que ocurre este problema es que ciertas ramas de los programas bajo prueba requieren determinados estados de los objetos que no pueden ser generados por las herramientas.

2.2.6 Oráculo

Actualmente los generadores de casos de prueba normalmente no generan pruebas completas, sino que generan secuencias de llamadas a métodos [45]. Mientras que son buenas para cubrir código, su eficacia depende de buenos controles en tiempo de ejecución en el código. Para que estas secuencias sean casos de prueba adecuados, se necesitan oráculos para verificar la corrección del comportamiento observado. Se han realizado esfuerzos para automatizar los oráculos en la forma de modelado, especificaciones, desarrollo dirigido por contrato y pruebas metamórficas [32]. También se ha tratado de obtener oráculos automatizados vía transformaciones de testeabilidad. Se trata de utilizar trasformaciones de testeabilidad para producir versiones alternativas de los programas que puedan ser usadas para verificar la versión original [20].

2.2.7 Múltiples objetivos de prueba

Por último, un problema a tratar es lograr optimizar múltiples objetivos de prueba. Los enfoques de *SBST* son capaces de optimizar más de una función de adecuación por vez, lo que permite enfocarse en soluciones que satisfagan más de un criterio [32]. El resultado de la búsqueda multi-objetivo es un conjunto de solución óptimo de Pareto, donde cada miembro del conjunto no es mejor que alguno de los demás para todos los objetivos. Las búsquedas multi-objetivo proveen una ventaja sobre las técnicas de prueba tradicionales que son capaces solamente de realizar una cosa por vez.

2.3 Herramientas

Existen muchas herramientas para la generación automática de casos de prueba. Algunas son usadas en la industria y otras están bajo desarrollo. Para la generación automática existen diferentes enfoques. En la presente sección se hace una revisión de tres de los principales enfoques para la generación automática de casos de prueba: ejecución simbólica, ejecución simbólica dinámica y pruebas basadas en búsqueda.

2.3.1 Herramientas basadas en ejecución simbólica y *DSE*

Papadakis *et al*. [46] propone una herramienta basada en ejecución simbólica para la generación automática de datos. La herramienta emplea una estrategia de selección de caminos integrada con pruebas aleatorias para generar casos de prueba y usa programación lineal para resolver las restricciones.

La extensión *JPF-SE* [47] de la herramienta *Java Path Finder* [48] permite la ejecución simbólica de programas basados en Java basado en un verificador de modelos.

PET [49] es un prototipo para la generación de datos de prueba estático de *bytecode* de Java. *PET* está basado en una técnica llamada *Partial Evaluation* (*PE*) [50] desarrolladas para *CLP* (*Constraint Logic Programming*). *PE* es una técnica que especializa programas. Una extensión de *PET*, llamada *jPET* [51] realiza ingeniería inversa de los casos de prueba tomando la información obtenida a nivel de los *bytecodes* y mostrándolos en una forma comprensible a nivel del código fuente Java. Con este fin, *jPet* se puede integrar dentro del entorno de programación Eclipse y extender las funcionalidades de *PET*.

Euclide [52] es un herramienta basada en restricciones para verificar programas críticos escritos en el lenguaje C. Utiliza un enfoque basado en restricciones para la generación de datos de prueba.

PathCrawler [28] y *Dart* [53] y *Osmose* [54] son tres herramientas que usan un enfoque de generación de datos orientado al camino y que utilizan ejecución real y ejecución simbólica. Estas herramientas seleccionan caminos en una forma indirecta y habitualmente consideran todos los caminos del programa. *Osmose* [54] trabaja directamente con el código de máquina ejecutable en vez del código fuente.

Cute [29] combina ejecución real y ejecución simbólica para generar datos de prueba con el objetivo de explorar todos los caminos de ejecución posibles. Como estrategia para generar casos de prueba usa pruebas *Concolic* basada en una exploración llamada *depth-first* de todos los caminos factibles del programa bajo prueba. Una implementación de *Cute*, llamada *jCute* fue desarrollada para analizar programas

desarrollados en Java [55]. Majumdar y Sen [56] extendieron *Cute* para implemtar *hybrid concolic testing* [56] (combinación de pruebas aleatorias con pruebas *Concolic*).

EGT [57], al igual que *Cute* [29] usa una combinación de ejecución real y simbólica. *Crest* [58] es una herramienta de código abierto que genera datos de prueba para C. Usa una estrategia de búsqueda de datos guiada por la estructura estática del código, llamada grafo de flujo de control (*CFG* por sus siglas en inglés). Como estrategia para la generación de datos de prueba usa una combinación de análisis estático y dinámico y diferentes estrategias de exploración de caminos.

LCT (del inglés "*Lime Concolic Tester"*) [59] es una herramienta de código abierto que genera datos de prueba para los programas Java que se basa en las pruebas *concolic*. *LCT* toma como entrada un programa secuencial Java que ha sido compilado en *bytecode* y genera pruebas que intentan cubrir todos los caminos de ejecución del programa. *LCT* soporta múltiples estrategias de búsqueda que afectan el orden en el que se exploran las rutas de ejecución. Durante el proceso de prueba las excepciones no detectadas son reportadas como defectos.

EXE [60] (del inglés "*Execution generated Executions*") genera los casos de prueba a partir del código bajo prueba usando ejecución simbólica. Cuando el código verifica en forma condicional una expresión, *EXE* realiza la ejecución, restringiendo la expresión a que sea verdadera en la rama verdadera y falsa en la otra. Cuando un camino termina o encuentra un bug, *EXE* genera en forma automática un caso de prueba.

Klee [61] extiende *EXE* [60] para abordar el problema llamado explosión de caminos. Emplea distintas estrategias de resolución de restricciones y usa heurísticas de búsqueda. Además, usa un enfoque para tratar con el entorno de ejecución.

Una herramienta para la generación automática de casos de prueba para .NET es *Pex* [62]. *Pex* explora el código bajo prueba utilizando *DSE*.

Las pruebas de software orientado a objetos consisten en secuencias de invocaciones a métodos. *Symstra* [63] es un marco que usa ejecución simbólica para generar un número pequeño de secuencias de métodos para estructuras complejas, de forma tal de poder explorar en forma exhaustiva secuencia de métodos con argumentos simbólicos hasta un largo dado. *Kunit* [64], desarrollado sobre el motor de ejecución simbólica *Bogor/Kiasan*, permite la generación de casos de prueba para secuencias de encabezados de programas desarrollados en Java. *Kiasan* [65] es una técnica que combina ejecución simbólica, prueba de modelos, prueba de teorema y resolución de restricciones para apoyar el análisis del diseño por contrato del software orientado a objetos.

En la **Tabla 1** se presentan las herramientas de generación de casos de prueba basadas en ejecución simbólica y *DSE* encontradas en la literatura con sus respectivos lenguajes que soportan. En negrita las herramientas disponibles en forma pública. Las demás son propietarias.

Tabla 1: Herramientas basadas en ejecución simbólica y DSE y lenguajes soportados.

Herramienta	**Lenguajes**
Papadakis *et al*. [46]	Múltiples lenguajes (C, Delphi, etc.)
Java Path Finder [48] **(extensión JPF-SE** [47]**)**	*Java*
Euclide [52]	C
PathCrawler [28]	C / C++
Dart [53]	C
Osmose [54]	Código de máquina ejecutable.

Herramienta	**Lenguajes**
Cute [29]	C; Java (*jCute*)
Majumdar y Sen [56]	C
Execution Generated Testing (*EGT*) [57]	C
Crest [58]	C
Pex [62]	.NET
Symstra [63]	Java
Kiasan/KUnit [64]	Java
Sireum/Kiasan [66]	Java
EXE [60]	C
Klee [61]	C
PET [49]	Java
jPet [51]	Java
LCT (Lime Concolic Testing)	Java

2.3.2 Herramientas basadas en *SBST*

Diaz *et al*. [67] presentan una herramienta divida en módulos que genera casos de prueba para programas escritos en C y en C++, de forma tal que permite el uso de distintos generadores de casos de prueba independientes entre ellos. Actualmente implementa el algoritmo de búsqueda *Tabu*, el algoritmo de búsqueda *Scatter* y el algoritmo aleatorio.

Iguana [68] es una herramienta que utiliza enfoques basados en *SBST* desarrollada para que los investigadores puedan comparar los diferentes algoritmos de generación, como por ejemplo búsqueda aleatoria y *hill climbing*. Se puede utilizar también para comparar

funciones de adecuación, como por ejemplo obtener cobertura de rama de un programa, y para poder comparar técnicas de análisis para la generación.

eTOC [69] (*evolutionary testing of classes*) utiliza un algoritmo genético para la generación De la misma forma que *eTOC* [69] , *Evosuite* [70] genera casos de prueba con un algoritmo genético derivando candidatos individuales llamados cromosomas y usando operadores inspirados por la evolución natural (por ejemplo, selección, cruce y mutación), de tal manera que forma iterativamente mejores soluciones con respecto a un objetivo de optimización (por ejemplo, la cobertura de rama) [71].

Austin [72] (*AUgmented Search–based TestINg*) usa una variantes del método de Korel [43] llamado *Alternating Variable Method* y técnicas adaptadas de *directed adaptive random testing* [53] y de ejecución simbólica dinámica (*DSE*). Combina el algoritmo *hill climbing* para tipos de datos de entrada enteros y de punto flotante con un conjunto de reglas de resolución de restricciones para entradas de tipo punteros.

Testful [73] es una herramienta para la generación semiautomática de prueba unitaria para sistemas con estado que utiliza enfoques basados en búsqueda. Es considerada semiautomática porque se le solicita al usuario algunos datos para aumentar la eficiencia del enfoque. *Testful* aborda particularmente el problema de la generación de pruebas para los sistemas con estados. Al igual que *Evosuite* el enfoque de *Testful* no se focaliza en un objetivo de prueba a la vez (por ejemplo una rama particular). *Testful* considera que las pruebas para los sistemas con estados están conceptualmente compuestas de dos partes [74]: una que crea los estados deseados de los objetos, y otra que ejercita cada uno de estos estados.

El marco *EvoTest* (*EvoTest Framework: ETF*) [75] nació de la investigación multidisciplinaria financiada por la Unión Europea en el proyecto *EvoTest* (*IST*-33472), cuyo objetivo es encontrar soluciones a los problemas de pruebas de sistemas de software a través del uso de técnicas evolutivas. El marco representa el estado de las técnicas de prueba automáticas evolutivas y se pretende que se puede utilizar en la industria. *Evotest* se puede utilizar para generar casos de prueba para módulos de software de *ANSI C*. *ETF: Daimler's EST* [76] también usa un enfoque evolutivo como el de *EvoTest.*

Ghani *et al.* [77] propone una herramienta que extiende las técnicas de prueba basadas en búsqueda a los criterios de cobertura *MCDC* y condición múltiple. La herramienta puede ser extendida para incluir distintos algoritmos basados en búsqueda.

La combinación de análisis estático y dinámico también ha sido usada en una herramienta llamada *Palus* [30]. Esta herramienta no está basada en *SBST*. Para generar los datos usa información del análisis dinámico para crear secuencias de llamadas a métodos que no violan especificaciones implícitas del programa, llamadas secuencias legales, y usa información del análisis estático para diversificar las secuencias generadas. Las pruebas generadas por *Palus* alcanzaron una mayor cobertura estructural y detectaron más defectos que con otros enfoques aleatorios [30].

En la **Tabla 2** se presenta las herramientas de generación de casos de prueba basadas en *SBST* encontradas en la literatura con sus respectivos lenguajes que soportan. En negrita las herramientas disponibles en forma pública, con * las herramientas disponibles para uso académico. Las demás son propietarias.

Tabla 2: Herramientas basadas en *SBST* y lenguajes soportados.

Herramienta	**Lenguajes**
Diaz *et al*. [67]	*C/C++*
Iguana [68]	*C*
eTOC [69]	*Java*
Austin [72]	*C*
*Evotest** [75]	*C*
ETF: Daimler's EST [76]	*C*
Evosuite [70]	*Java*
Evacon [78]	*Java*
Ghani *et al*. [77]	*Java*
Testful [73]	*Java*

2.4 Efectividad de las herramientas

Se ha investigado la performance de los enfoques para la generación automática de casos de prueba. En particular dos enfoques para la generación han sido comparados [5]: el enfoque *concolic* de la herramienta *Cute* [29] y un enfoque basado en búsqueda de la herramienta *Austin* [72]. 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. En este estudio se investigó qué tipo de estructuras de programa hacen fallar a las herramientas en la generación. Las razones más comunes son las siguientes:

- Fallos de segmentación que son el resultado de restricciones implícitas en punteros y por asignación de valores incorrectos a los parámetros de entrada.
- Excepciones de punto flotante, que pueden ser según el estándar *IEEE* 754, operación no válida, división por cero, desbordamiento, subdesbordamiento y cálculo inexacto. Mediante una inspección de código se determinó que en todos los casos la causa de la excepción fue un error de división por cero. Este resultado era esperado ya que todas las entradas primitivas se inicializan en cero por ambas herramientas.
- Funciones no manejadas que se refiere a tipos de entradas no manejadas por las herramientas.
- Operaciones de entrada y salida en el 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 [44], donde los desarrolladores proveen una guía a las herramientas para ayudarlas a abordar los problemas. En una encuesta [6] 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 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 no pueden proveer una adecuada guía

en la búsqueda de datos. Un enfoque [79] 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* [70].

Se ha reportado la cobertura alcanzada por *Testful* y por *Evosuite*. Con *Testful*, ejecutando la herramienta 10 minutos se obtuvo una cobertura de rama de 82.3 % [74]. Con *Evosuite* se ha demostrado que cuando el programa no tiene dependencias con el entorno (por ejemplo, cuando el código no accede el 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 % [71].

Se ha evaluado la herramienta *Evotest* [75] en cuatro módulos de software industrial [80]. Los módulos tuvieron que ser adaptados para atender a las limitaciones de la herramienta, de tal manera que el mayor número posible de las funciones seleccionadas para evaluación pudieran ser probados. En la evaluación se pudieron generar casos de prueba para 37% de las funciones seleccionadas. Para las demás funciones que la herramienta no pudo manejar, las razones del fracaso fueron principalmente debidas al uso de punteros (92 % de los fallos), uso de *arrays,* variables volátiles (variables con propiedades especiales relacionadas con la optimización) e instrumentación de funciones múltiples. La instrumentación de funciones múltiples se refiere a generar casos de prueba no solo para funciones individuales en aislamiento sino que también para funciones que invocan a otras. Las limitaciones mostradas por *Evotest* [75] son compartidas con el prototipo *ETF: Daimler's EST* [76].

En lo que respecta a la detección de defectos las herramientas automáticas pueden revelar defectos no encontrados por técnicas de prueba manuales [81]. Algunos estudios indican también que las estrategias de prueba manuales y automáticas son complementarias [82], [83].

La herramienta de generación de casos de prueba aleatorios *Randoop* [84] ha demostrado tener una efectividad similar a la de las pruebas unitarias manuales en lo que respecta a la detección de defectos [85]. 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 [85], [86].

A pesar de que algunos estudios como los citados anteriormente [81], [82], [83], [85], [86] 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. En un experimento [87] 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.

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 [88]. 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* [84] fueron defectos falsos [88].

2.5 Resumen del estado de la cuestión

A lo largo de las últimas décadas se han propuesto diversas técnicas para la generación automática de casos de prueba. Además de la división de enfoques para la generación automática de casos de prueba que se muestra en la **Figura 6**, se han propuesto otras divisiones. Tracey *et al*. [8] muestra una división indicando la progresión de ideas a través del tiempo (**Figura 7**). Las flechas en la **Figura 7** indican la progresión de las ideas y los métodos a través de los diversos enfoques. Mientras que la década de 1980 vio el surgimiento de los métodos formales como un foco de atención de la investigación académica, la década de 1990 ha visto el crecimiento una vez más de la investigación de pruebas de software. Esto es en parte debido a la constatación de que el uso de métodos formales no obvia la necesidad de una buena prueba del software y también la constatación de que las pruebas de software pueden proporcionar un medio rentable de verificación [8].

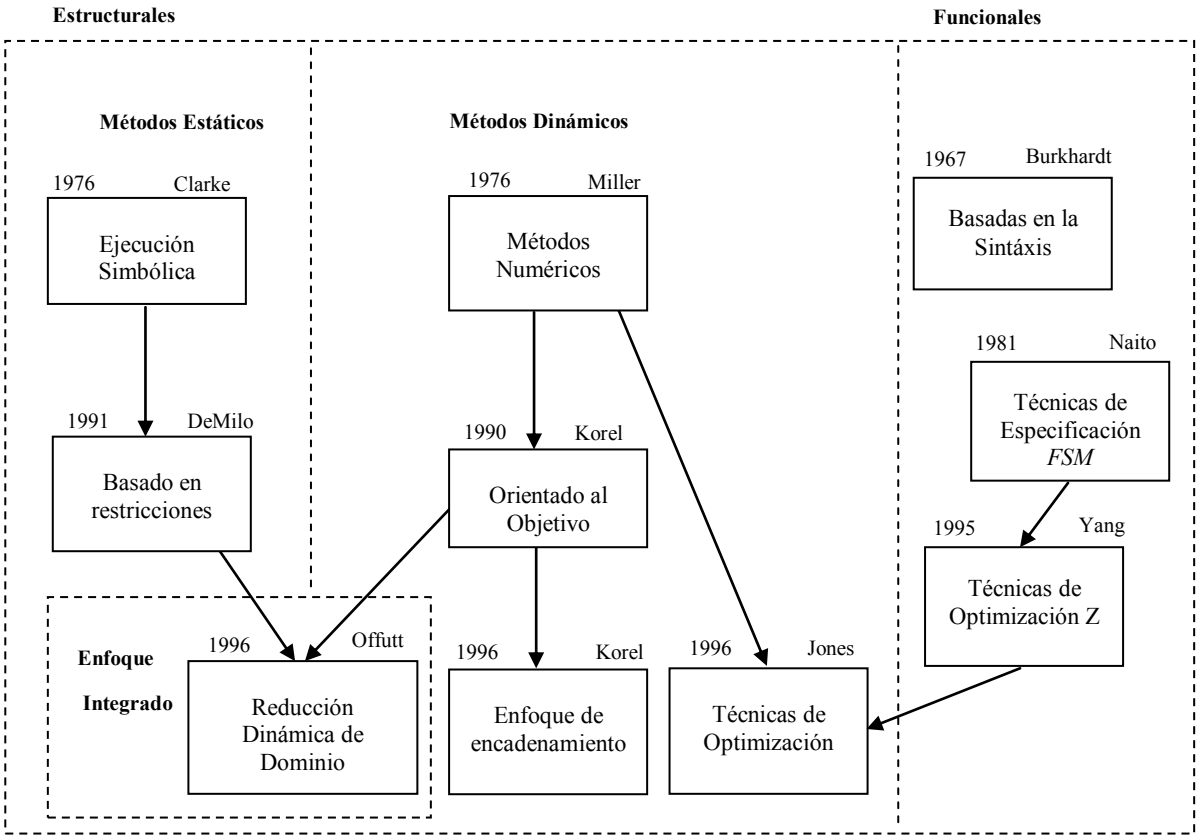


Figura 7: Progresión de enfoques para la generación automática de casos de prueba según Tracey *et al.* [8]

La eficacia de las técnicas para generar casos de prueba es crucial para lograr obtener software de mejor calidad. Esta eficacia se debe medir no solo con respecto a la cobertura del código sino que además las técnicas deben lograr generar automáticamente casos de prueba que sean relevantes para encontrar defectos en el software.

A pesar que hace varias décadas que las técnicas se han comenzado a desarrollar y que se han implementado algunas herramientas, aún presentan muchos problemas y la adopción de las técnicas por parte de los profesionales en la industria aún es muy baja. La presente investigación permite identificar los siguientes huecos en el conocimiento:

- Si bien se conocen los problemas que tienen las técnicas, aún se requiere enriquecer el cuerpo de conocimiento con un compendio de los problemas que son investigados experimentalmente y con qué técnicas, y una clasificación de las técnicas que incluya el contexto de investigación de los experimentos. Este compendio permitiría a los profesionales disponer de más información acerca de la posible aplicación de las técnicas en distintos contextos y a los investigadores saber qué problemas se están investigando empíricamente y con qué técnicas para generar nuevas hipótesis y futuras líneas de investigación.
- La investigación de la generación de casos de prueba frecuentemente se ha focalizado en los problemas técnicos de la generación, pero se desconoce la aplicación práctica de los casos de prueba generados. Es necesario mostrar con casos prácticos qué efectividad tienen los casos de prueba para la detección de defectos en el software.

3. Metodología de investigación

La ingeniería de software empírica se caracteriza por la aplicación del método científico experimental que se basa en la observación y manipulación rigurosa de la realidad para estudiar un fenómeno. La investigación empírica pretende explorar, describir, predecir y explicar fenómenos naturales, sociales, o cognitivos mediante pruebas basadas en la observación o la experiencia [89]. **Esta investigación utiliza una metodología con un enfoque empírico.**

Las metodologías de investigación sirven para diferentes propósitos: exploratorio, descriptivo, explicativo, mejora [90]. Un estudio exploratorio tiene como objetivo descubrir lo que está sucediendo, la búsqueda de nuevas ideas y la generación de ideas e hipótesis para nuevas investigaciones [91]. A su vez un estudio descriptivo tiene como objetivo retratar a una situación o fenómeno. **Debido a que el conocimiento sobre generación automática de pruebas todavía es escaso el propósito de esta investigación es descriptivo y exploratorio.**

Los enfoques para la investigación empírica pueden incorporar métodos cuantitativos y cualitativos para la recolección y análisis de datos [89], [92]. Ambos métodos emplean procesos cuidadosos, sistemáticos y empíricos para generar conocimiento [92]. Los métodos cuantitativos recogen datos numéricos y los analizan con métodos estadísticos, mientras que los métodos cualitativos recogen material en forma de texto, imágenes o sonidos extraídos de observaciones, entrevistas y pruebas documentales, y la analizan utilizando métodos que no se basan en la medición precisa para producir su conclusiones [92]. Existe un enfoque mixto que es una combinación del enfoque

cuantitativo y del cualitativo [89]. **Esta investigación utiliza un enfoque de investigación mixto.**

El método general de investigación se diseña alineado con los objetivos específicos de la tesis. En la **Figura 8** puede verse el método general de investigación alineado a los objetivos de la tesis.

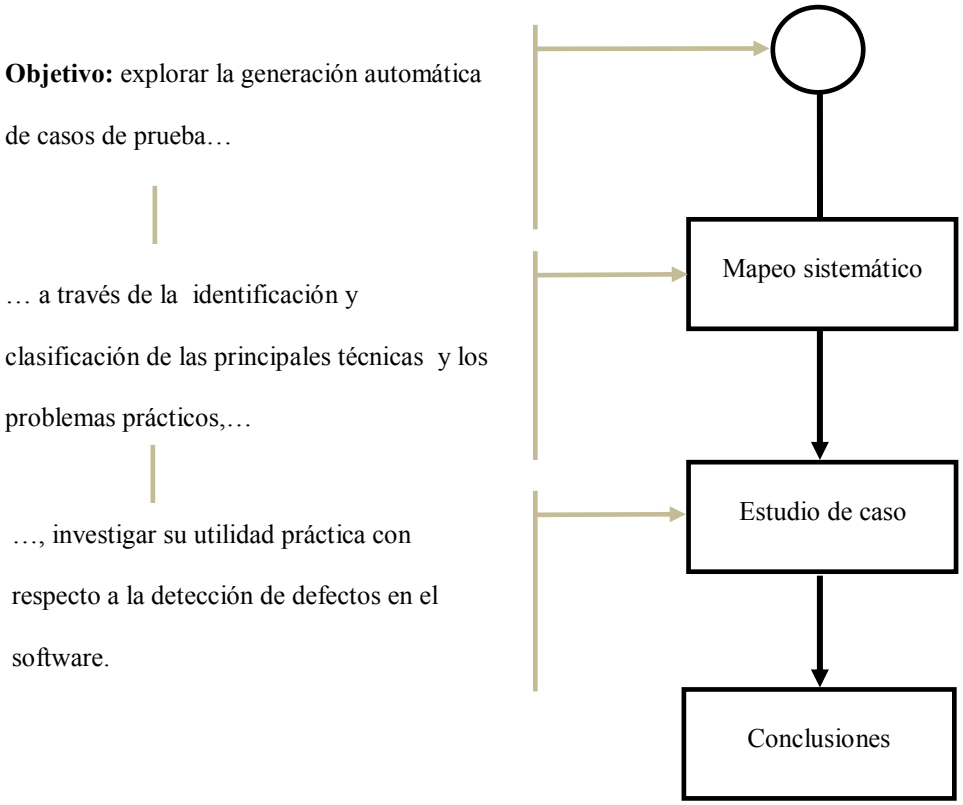


Figura 8: Método general de investigación

Para lograr los objetivos específicos del primer paso del método general de investigación (**Figura 8)** se realiza un mapeo sistemático de la literatura sobre experimentos de generación automática de casos de prueba estructurales. El estudio busca identificar cuáles son los enfoques para la generación automática de casos de prueba que se investigan experimentalmente. Para determinar qué técnicas pueden ser

aplicadas en la práctica se requiere realizar estudios empíricos que muestren su utilidad y aplicabilidad en determinados contextos según lo recomienda la ingeniería de software basada en evidencia [93]. Para obtener las técnicas en este paso se consideran estudios empíricos que proveen evidencia científica acerca de la eficacia de las técnicas y no meras especulaciones [13].

El mapeo sistemático tiene dos resultados. El primero es un mapa con las técnicas con el tipo de software con el que han demostrado ser más eficientes e identificando el método de implementación de la técnica (estático, dinámico o híbrido). El segundo resultado es una lista de los problemas que enfrenta la generación y que están siendo investigados empíricamente. El estudio usa datos de estudios publicados previamente con el propósito de sintetizar la evidencia. Los estudios considerados para sintetizar son experimentos y no otro tipo de estudios empíricos primarios (encuestas, estudios de caso, etc.), dado que los experimentos ofrecen mayor rigurosidad en las observaciones del fenómeno. El alcance del estudio además está delimitado por el tipo de técnicas investigadas que son técnicas de generación de casos de prueba estructurales.

El segundo paso de la tesis investiga la utilidad práctica de la generación automática de casos de prueba estructurales. La generación automática de prueba estructural se ha focalizado en aspectos técnicos de la generación, pero frecuentemente ha ignorado la utilidad de los casos de prueba luego de la generación [87]. La sola generación automática de casos de prueba de alta cobertura, no necesariamente mejora la capacidad para probar el software.

Del primer paso de la metodología general de investigación se obtiene un conjunto de enfoques para la generación que han demostrado empíricamente ser eficientes para la

generación en un determinado contexto. Estos enfoques son tomados como entrada para el estudio de caso. En el estudio de caso se seleccionan herramientas que implementen alguno de los enfoques mencionados con el fin de investigar específicamente la utilidad práctica con respecto a la detección de defectos en el software. En el estudio de caso, para lograr el objetivo específico se plantea realizar una investigación con las herramientas seleccionadas y compararlas con respecto a la cantidad de defectos detectados en un determinado software bajo prueba.

El paso final consiste en discutir los hallazgos realizados en los estudios anteriores, destacar los aportes de la investigación y en describir las líneas de investigación futuras. Todos los resultados intermedios de la tesis pueden ser tomados por otros investigadores para realizar futuras investigaciones y por los profesionales para tomar decisiones acerca de las técnicas a utilizar.

3.1 Mapeo sistemático

Un estudio de mapeo sistemático es un estudio secundario que tiene como objetivo construir un esquema de clasificación y estructurar un campo de interés de la ingeniería de software [94]. Un estudio secundario es aquel cuyo conocimiento no es generado con base en mediciones directas sino que surge del análisis de estudios primarios. Los estudios primarios son estudios empíricos donde se genera conocimiento con base en mediciones directas [95]. Un estudio de mapeo provee una estructura del tipo de investigación y de los resultados que se ha publicado, clasificándolos y muchas veces da un resumen visual, el mapa de sus resultados [94].

Los estudios de mapeo pueden ser de gran utilidad para los investigadores para el establecimiento de líneas de base para futuras actividades de investigación [96]. Una línea de base se puede utilizar en una variedad de maneras, ya sea como el punto de partida para la investigación de las tendencias de investigación o el punto de partida para las revisiones sistemáticas de la literatura. Las revisiones sistemáticas de la literatura son otro tipo de estudios secundarios que intentan responder preguntas de investigación específicas [97]. Las revisiones sistemáticas y los estudios de mapeo sistemáticos difieren en los objetivos, el alcance y la profundidad y que ambos métodos pueden ser usados en forma complementaria [94].

3.1.1 Motivación

El mapeo sistemático se realiza para enriquecer el cuerpo de conocimiento con un compendio de las principales técnicas y de los problemas investigados. Presenta los resultados de un estudio de mapeo sistemático para identificar y clasificar un conjunto de estudios primarios que reporten experimentos acerca de técnicas para aumentar la eficiencia en la generación automática de casos de prueba estructurales.

Particularmente los objetivos de estudio son:

- dar una visión general de qué evidencia empírica existe en la literatura acerca de generación automática de prueba estructural.
- obtener una lista de los problemas de la generación automática de casos de prueba estructural investigada y con qué técnicas se realiza la investigación.
- generar un mapa con las técnicas para la generación automática de casos de prueba estructurales incluyendo el contexto en el que se han investigado.

3.1.2 Método

Para llevar a cabo el estudio se siguen los pasos propuestos por Petersen *et al*. [94]. Cada uno de los pasos del proceso tiene un resultado, siendo el mapeo sistemático el resultado final del proceso. Los pasos definidos son los siguientes:

1. Definir las preguntas de investigación: las preguntas de investigación definen el alcance de la investigación y son definidas de acuerdo a los objetivos principales de la tesis.
2. Determinar las fuentes de datos y la estrategia de búsqueda: las fuentes de datos fueron seleccionadas teniendo en cuenta las más relevantes en la ingeniería de software. Para determinar la estrategia de búsqueda, se obtuvo términos relacionados con área de investigación [27] que se fueron refinando en sucesivas búsquedas exploratorias.
3. Seleccionar los estudios: en este paso se definen los criterios de inclusión y exclusión que son usados para incluir los estudios relevantes para responder las preguntas de investigación.
4. Clasificar los estudios: los artículos obtenidos en la selección de estudios se clasifican según cierto esquema de clasificación definido para contestar las preguntas de investigación planteadas. Dicha clasificación se realiza en función del título y resumen de los artículos seleccionados en el paso anterior.
5. Extraer y realizar el análisis de los de datos: una vez obtenido el esquema de clasificación, los artículos relevantes son leídos completamente y los resultados de la lectura son registrados en una planilla para su correspondiente análisis.

3.2 Estudio de caso

El segundo paso de la metodología general de investigación es un estudio de caso. El estudio de caso es un método empírico que tiene como objetivo investigar un fenómeno contemporáneo en su contexto [91]. El estudio de caso normalmente tiene como objetivo el seguimiento de un determinado atributo o el establecimiento de relaciones entre los diferentes atributos. Los estudios de caso son particularmente útiles cuando se intenta responder una pregunta de "cómo" o "por qué" acerca de un conjunto de eventos de un fenómeno [98].

Por otra parte, los experimentos son estudios caracterizados por medir los efectos de la manipulación de una variable sobre otra variable. Además, los sujetos se asignan a los tratamientos en forma aleatoria [99]. Los cuasi-experimentos son similares a los experimentos controlados, excepto que los sujetos no se asignan aleatoriamente a los tratamientos [91].

En esta investigación se explora el aporte que realizan dos herramientas de generación desde el punto de vista de las pruebas unitarias. A través del estudio de caso se plantea explorar qué cantidad de casos de prueba generan y qué cobertura logran. Además, se plantea determinar qué aportan las herramientas desde el punto de vista de la detección de defectos en el software. El sujeto de prueba es solo un programa que ha sido utilizado en otros experimentos de prueba, aplicando otras técnicas manuales, que no son parte del estudio de caso.

Si se tiene en cuenta la formalidad del estudio experimental, el estudio realizado en esta investigación se focaliza en un solo proyecto (programa), por lo que el método de

investigación es un estudio de caso. Si el estudio se focaliza en un solo proyecto Kitchenham *et al.* [100] prefiere llamar al estudio como estudio de caso, dado que no es posible tener un experimento formal sin replicación. Se debe tener en cuenta que en el estudio de esta investigación tampoco se tiene aleatoriedad, condición necesaria para un experimento, ya que se han investigado las herramientas con un solo programa. Se debe considerar también que un estudio de caso es un estudio observacional que coincide con el propósito de esta investigación.

3.2.1 Motivación

Explorar la utilidad con respecto a la detección de defectos de las herramientas que implementan enfoques para la generación automática de casos de prueba estructural, así como también comparar las herramientas consideradas con respecto a la cobertura alcanzada y a los defectos encontrados.

3.2.2 Método

Para realizar un estudio de caso hay 5 pasos que deben realizarse [91]:

1. Diseño del estudio de caso: se definen los objetivos y se hace un plan del estudio de caso. Los objetivos son refinados en un conjunto de preguntas de investigación que han de ser respondidas a través del análisis del estudio de caso. El plan debe contener algunos elementos como por ejemplo el objetivo que indica qué se espera conseguir, qué se estudia, un marco de referencia, preguntas de investigación que establecen lo que se necesita saber con el fin de cumplir con el objetivo del estudio y los métodos técnicos para recopilar los datos.

2. Preparación para la recolección de datos: se definen los procedimientos y los protocolos para la recolección de datos. El protocolo del estudio de caso es un contenedor para las decisiones de diseño en el estudio de caso, así como los procedimientos para su realización. El protocolo es un documento modificado continuamente que se actualizan cuando se cambian los planes del estudio de caso.
3. Recolección de datos: trata de la recolección de datos en el caso estudiado.
4. Análisis de los datos: se realiza el análisis de los datos obtenidos en el estudio.
5. Reporte: se comunican los hallazgos del estudio.

4. Ejecución de estudio de mapeo

4.1 Procedimiento

Para llevar a cabo el presente estudio se siguen los pasos mencionados por Petersen *et al*. [94] para un estudio de mapeo sistemático de la literatura, y se construye un protocolo con el propósito de evitar sesgos en el estudio [97]. Un protocolo de revisión especifica los métodos que se utilizarán para llevar a cabo el estudio de mapeo.

4.1.1 Preguntas de Investigación

Las preguntas de investigación se definen de acuerdo a los objetivos principales del estudio que son identificar los principales problemas de la generación automática de casos de prueba estructurales que se investigan y clasificar las técnicas. Entonces, para poder lograr los objetivos, se definieron dos preguntas de investigación (PI).

La pregunta PI1, define la base del estudio y brinda una perspectiva general de los experimentos con las técnicas de interés. La pregunta PI1 se define de la siguiente forma:

- ¿qué experimentos se realizaron relacionados con la generación automática de casos de prueba estructurales y con qué objetivos?

La pregunta PI2 clasifica las técnicas de acuerdo a una de las clasificaciones dada por Mahmood [27], que categoriza a las técnicas según el tipo de implementación: estático, dinámico o híbrido (combinación de los enfoques anteriores). Se define de la siguiente forma:

- ¿cuáles son las técnicas de generación de datos de prueba estructurales reportadas en los experimentos?

4.1.2 Fuentes de datos y estrategia de búsqueda

Se consideraron las siguientes bibliotecas digitales para realizar la búsqueda: *Scopus* e *IEEEXplore*. La razón para seleccionarlas es que según Dieste *et al*. [101], *Scopus* tiene menos debilidades que otras bases, cubre una amplia gama de publicaciones en el campo de la ciencia de la computación y mantiene una base de datos completa y consistente. Además de *Scopus*, algunos autores recomiendan el uso de otras bibliotecas digitales como *IEEEXplore, Inspec, Compendex* y *ACM Digital Library* [102]. De las últimas posibles se selecciona *IEEEXplore*.

Los términos utilizados en las búsquedas se formaron mediante la combinación de términos derivados del tema de interés y de los términos relacionados con experimentos. Los términos del tema de interés fueron obtenidos de algunas sugerencias de Mahmood [27] y se fueron ajustando durante las búsquedas preliminares realizadas. La lista de sinónimos relacionados con el término *experiment*, se obtuvo de Dieste *et al*. [101]. Mediante el uso del comodín *, en las bibliotecas digitales quedan incluidos en la búsqueda los sinónimos de dicho término. En la **Tabla 3** se presentan los términos usados en la cadena de búsqueda con los respectivos sinónimos considerados.

Tabla 3: Términos y sinónimos utilizados para componer la cadena de búsqueda

Términos	**Sinónimos**
Automatic	*Automated*
	Automation
Experiment	*Experimental study*

Términos	Sinónimos
	Experimental comparison
	Experimental analysis
	Experimental setting
Test case generation	*Test-case generation*
	Test data generation
	Test data creation
	Test generation
Test data	*Test values*
	Test input

La cadena de búsqueda empleada se conformó con los términos de la **Tabla 3** y con operadores *booleanos*. La cadena resultante es: (*automatic OR automated OR automation*) *AND ("test case generation" OR "test-case generation" OR "test generation" OR "test data generation" OR "test data creation" OR "test values" OR "test input") AND experiment* AND software*. En cada una de las bibliotecas digitales seleccionadas las búsquedas se realizaron considerando los campos palabras clave, título, resumen y artículos publicados entre los años 2006 y el 2012. La búsqueda en las bibliotecas digitales fue realizada el 08 de mayo de 2012.

El motivo de acotar la búsqueda al período 2006-2012, es que permite extraer una muestra amplia de artículos recientemente publicados y permite comparar algunos resultados con los dos artículos más similares y anteriores al período de búsqueda [27], [31] y recientes al presente estudio.

4.1.3 Selección de estudios

Para seleccionar los estudios primarios se definieron los siguientes criterios de inclusión y exclusión. Los criterios de inclusión definidos son los siguientes:

- Experimentos de la disciplina Ingeniería de Software que reporten enfoques que aporten mejoras en la generación automática de casos de prueba estructurales. Por ejemplo, técnicas específicas para la generación, técnicas para detectar caminos no factibles y estrategias para mejorar la performance de los algoritmos de generación.

Los criterios de exclusión definidos son los siguientes:

- Pruebas estructurales de interfaces de usuario gráficas y *web services*. Este tipo de investigación se excluye ya que son dos temas muy específicos y no están dentro del alcance de la presente investigación.
- Estudios de casos, reportes de experiencias, estudios observacionales, evaluaciones o comparaciones teóricas de técnicas.
- Experimentos cuyo objetivo sea principalmente la evaluación de una herramienta y no la evaluación de una técnica.
- Artículos no disponibles para ser descargados.

El proceso de selección de estudios primarios comprende cuatro fases, las cuales se llevan a cabo por dos investigadores. La **Figura 9** muestra una visión general del proceso de búsqueda para la selección de estudios primarios. De la búsqueda en las bibliotecas digitales resultaron 1483 artículos; 1140 obtenidos en *Scopus* y 343 obtenidos en *IEEEXplore*. La primera fase de la selección consiste en excluir los artículos duplicados. De 1483 artículos, se quitaron 142 artículos duplicados. Los 1341

artículos resultantes de la primera fase, se tomaron como entrada de la segunda fase, en la que se aplicaron los criterios de inclusión y exclusión basados en el título de los artículos. En la segunda fase se excluyeron 1199, por lo que quedaron 142 artículos. Los artículos excluidos en la segunda fase principalmente pertenecen a otro dominio, como por ejemplo artículos relacionados con microprocesadores y electrónica. Además se encontraron artículos de pruebas de software que tratan de pruebas de performance y seguridad, por lo que se descartaron.

En la fase 3, se aplican los criterios de inclusión y exclusión basados en el resumen de los artículos primarios. De un total de 142 artículos se excluyen 62. La razón principal de la exclusión en esta fase es que algunos artículos no reportan experimentos relacionados con pruebas estructurales (por ejemplo pruebas basadas en modelos o funcionales), estudios secundarios, pruebas de software para interfaces gráficas y para *web services*.

En la última fase, la fase 4, se aplicaron los criterios de inclusión y exclusión basados en el texto completo de los artículos primarios. A los 80 artículos resultantes de la fase 3 se les aplicó los siguientes criterios de calidad tomados de Elberzhager *et al*. [102], para comprobar su adecuación a las preguntas planteadas en el presente estudio:

- Los objetivos de la investigación se describen con claridad.
- El enfoque se explica suficientemente.
- Los factores contextuales y ambientales se presentan con claridad.
- Los datos de entrada y salida de usar el enfoque son explícitamente mencionados.
- La performance del enfoque se aclara suficientemente.

- La evidencia del enfoque se documenta.

Aplicando los criterios de calidad mencionados, de 80 artículos, en la fase 4, se descartaron 32 artículos, resultando 48 artículos primarios que se toman como entrada para el proceso de extracción de datos.

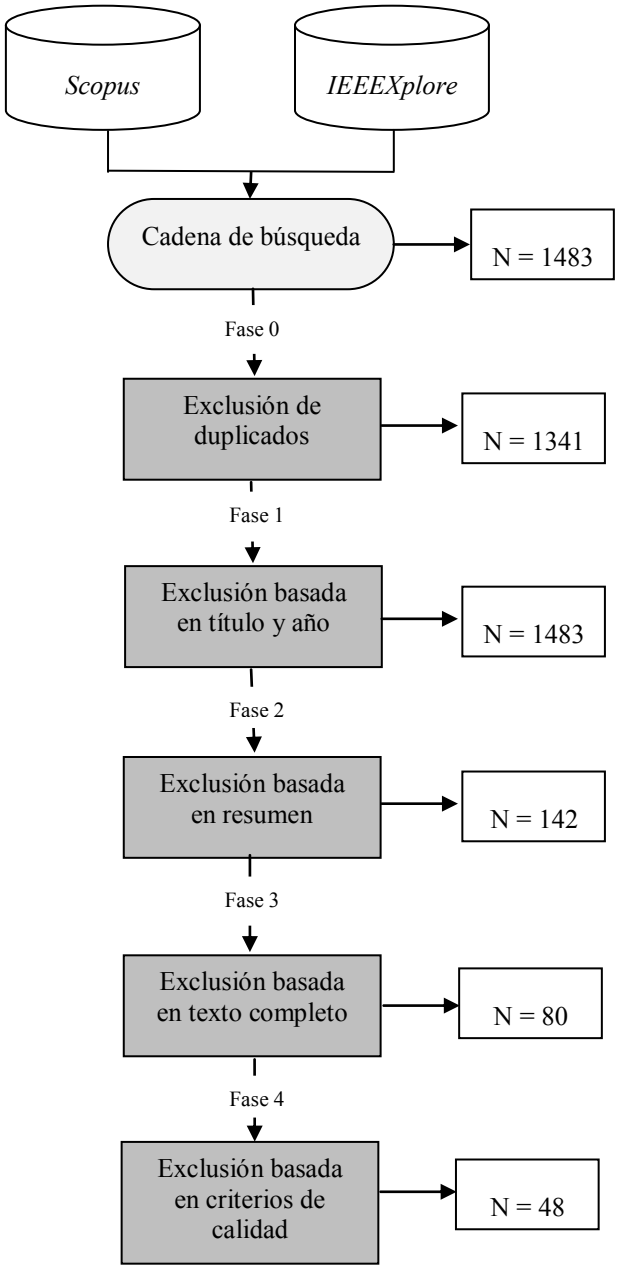


Figura 9: Proceso de búsqueda para la selección de estudios primarios

4.1.4 Extracción de Datos y Síntesis

Para la extracción de datos se desarrolló un formulario (**Tabla 4**) con el fin de extraer toda la información relevante para contestar las preguntas de investigación.

Tabla 4: Formulario de extracción de datos

Dato	**Valor**	**Notas adicionales**
Nombre del artículo		
Año		
Fuente		
Canal de publicación		
Objetivo del estudio		
Tipo de Técnica		
Algoritmos de generación		
Método de implementación		
Aspectos estudiados		
Contexto		

Los artículos seleccionados se clasificaron según el siguiente esquema de clasificación:

- Tipo de técnica: se refiere al tipo de técnica de prueba de software usada en el experimento. Las técnicas encontradas en los experimentos son las siguientes: *Search Based Software Testing (SBST)*, *Random Testing (RT)* [103], *Constraint-Based Testing (CBT)* [104] e *Hybrid Concolic Testing (HCT)* [56] y combinaciones de las anteriores.
- Aspectos estudiados: se refiere a los principales aspectos estudiados en el experimento y conceptos que reflejen la contribución del experimento.

Los aspectos y conceptos encontrados en los objetivos de los artículos primarios se agruparon en las categorías que se describen en la **Tabla 5**.

- Contexto: se refiere al tipo de programa empleado para realiza los experimentos. Permite diferenciar experimentos que se realizaron con programas de pequeña escala o programas de escala industrial.

Tabla 5: Taxonomía de clasificación

Clase	**Descripción**
Optimización del conjunto de casos de prueba y reducción del espacio de búsqueda.	Agrupa a los métodos para minimizar los conjuntos de casos de prueba, métodos para optimizar el número de casos de prueba generados y reducción del dominio de entrada.
Tratamiento de estructuras de programas.	Agrupa las propuestas para tratar con punteros, *arrays*, librería de funciones, pruebas de clase, problema de la forma, problemas de estado en la programación orientada a objetos.
Generación de datos de prueba.	Agrupa a las técnicas de generación y búsqueda de datos.

Los 48 artículos primarios obtenidos en el proceso de selección de estudios, se clasificaron según la taxonomía presentada en la **Tabla 5**, para abordar las 2 preguntas planteadas en la presente investigación. La asignación de un artículo a una determinada categoría se realizó considerando los problemas en los que se focalizan los estudios o las técnicas que plantean probar para optimizar los distintos aspectos de la generación automática de datos. Cada una de las categorías no son excluyentes, es decir que un artículo se puede haber clasificado en una o más categorías, dado que cada uno de los estudios tiene uno o más objetivos.

4.2 Resultados

Los resultados del estudio de mapeo, se presentan respondiendo cada una de las preguntas de investigación. Cada una de las categorías de la **Tabla 5** representa uno o un conjunto de temas de investigación y permiten responder la pregunta de investigación PI1.

En la **Figura 10** se muestran los resultados de la clasificación de los artículos en las distintas categorías. De los 48 artículos primarios seleccionados, 35 (72.91%) fueron clasificados en el área de *SBST*. De *RT* se encontraron 3 estudios (6.25 %). Otras áreas de pruebas de software como *Concolic Testing, Hybrid Concolic Testing* y combinaciones de las anteriores se clasificaron como *Miscellaneous Testing (MiscTesting)*, y son en total 10 estudios que representan el 20.83 % del total.

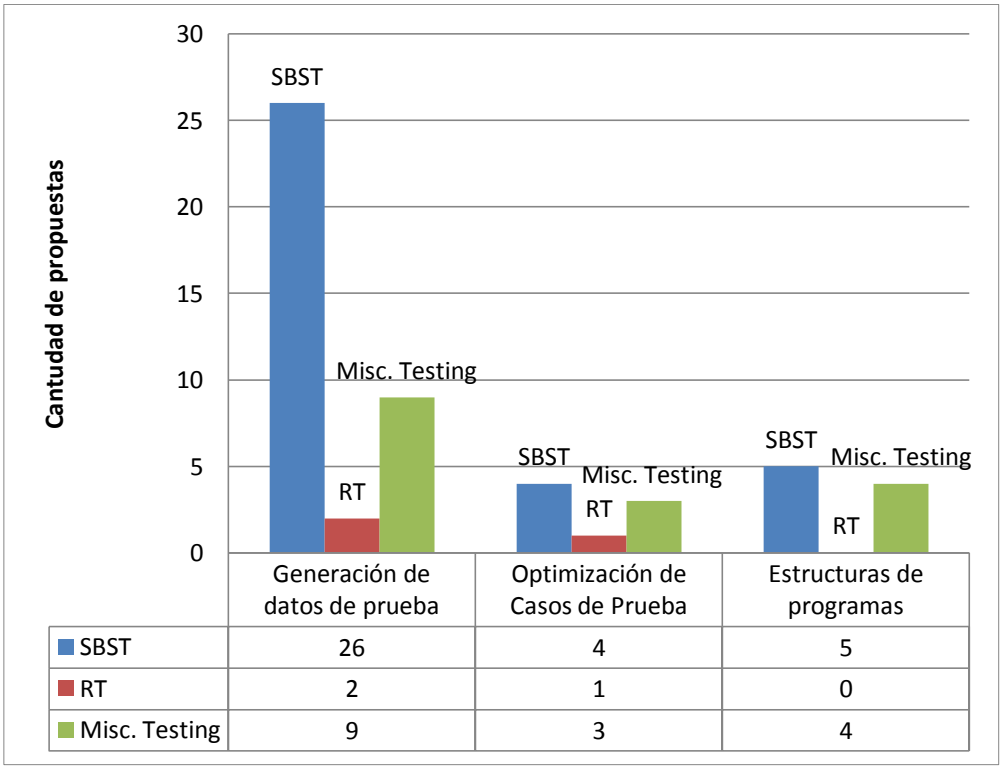


Figura 10: Clasificación de estudios según el área de prueba y el tipo de estudio

Las fuentes y los canales de publicación de los experimentos son diversos. Considerando todos los artículos seleccionados, se identifican 36 fuentes de publicación. En lo que respecta a los canales de publicación, se identifican 3 canales de publicación: revistas, conferencias y simposios. Más del 65% (31 artículos) se publicaron en conferencias, 16 artículos en revistas y uno en un simposio.

4.2.1 PI1: experimentos de generación automática

Para abordar esta pregunta se describe el foco principal de cada estudio según cada categoría de clasificación.

4.2.1.1 Optimización del conjunto de casos de prueba

El propósito de la reducción del conjunto de casos de prueba es obtener la mínima colección de casos de prueba sin reducir la calidad de las pruebas de software. De esa forma el costo de las pruebas de software se reduce y el proceso de desarrollo de software es acelerado [105]. La integración de métodos de prueba aleatorios, tecnología invariante (propiedades en un punto del programa que pueden revelar la extensión de la cobertura de datos del programa y otra información) y algoritmos genéticos, permite la reducción eficiente de los casos de prueba [105], [106].

Para *SBST* se propone un algoritmo basado en un grafo de control de dependencias que realiza un análisis estático para reducir el número de casos de prueba generados y lograr también la reducción del costo del oráculo [107]. Un algoritmo basado en razonamiento de restricciones (del inglés *constraint reasoning*) que permite crear conjuntos de casos de prueba que minimicen el número de rechazos (datos de prueba que ejecuten otro camino de flujo de control), es presentado para *RT* [108]. Razonamiento de restricciones

consiste en métodos genéricos para resolver problemas complejos que involucran restricciones.

El espacio de búsqueda de potenciales entrada de datos puede ser muy extenso, aún para programas pequeños. Su tamaño, es un factor que afecta la eficiencia de la generación de datos de prueba en cualquier enfoque de *SBST*. Para la reducción del espacio de búsqueda, una propuesta es usar una técnica llamada estrategia de eliminación de variable de entrada irrelevante [109]. La técnica consiste en eliminar del espacio de búsqueda las variables que no afectan si una estructura objetivo puede ser ejecutada o no, y por lo tanto puede ser removida. Para reducir el espacio de búsqueda en *CBT* en pruebas orientadas al camino, se investigan heurísticas para descartar caminos no relevantes, logrando una reducción del espacio de búsqueda [110]. También, se investiga una solución [111] para la reducción del espacio de búsqueda de los contenedores (*arrays, lists, vectors, trees,* etc.), en la programación orientada a objetos.

4.2.1.2 Tratamiento de estructuras de programas

La generación de datos de prueba con algoritmos genéticos para clases en la programación orientada a objetos tiene algunos defectos. Por ejemplo, para la generación de datos solo tratan con tipos de datos básicos y no soportan suficientemente tipos de datos definidos por el usuario. Además, durante la generación, algunos individuos pueden ser generados y seleccionados repetidamente, lo que resulta en una convergencia temprana. Para resolver el problema, se presenta una propuesta [112] que combina *tabu search* y algoritmos genéticos (*GA*) llamada *Tabu GA*. Otro problema para la generación son los programas que tienen punteros y estructuras de datos dinámicas. Una solución a este problema se propone llamada enfoque de asignación de

direcciones [113]. Un enfoque compuesto de dos fases para la generación de datos de estructuras dinámicas [114] probó ser más eficiente que el de asignación de direcciones, así como también que puede detectar caminos no factibles. Otro problema relacionados con los caminos es el llamado explosión de caminos, que ocurre principalmente por llamadas anidadas en el código de los programas, bucles y condiciones. Tres heurísticas complementarias se proponen para tratar con este problema [110].

Una limitación de la ejecución simbólica es el tratamiento de algunas estructuras como punteros y *arrays*. Para poder solventar las limitaciones algunos enfoques combinan ejecución simbólica con ejecución real. Uno de estos enfoques es *Concolic Testing (CT)* [53]. Este enfoque tiene algunas limitaciones al fallar en la ejecución de algunas ramas y al incrementar el número de casos de prueba redundantes. Una propuesta [115] para complementar ejecución simbólica de una forma distinta a la que lo hace *CT* permite solventar algunas de las limitaciones que tiene ejecución simbólica con respecto a los punteros, *arrays* y funciones de bibliotecas.

En la programación orientada a objetos, por lo general los programas tienen estados internos, los cuales son problemáticos porque la cobertura de algunas estructuras puede depender de su estado interno actual. Para lograr cubrir una determinada rama del código en este tipo de programas, por lo general se requiere una secuencia de llamadas a funciones para establecer el estado interno en la configuración adecuada. Se ha investigado la importancia del largo de las secuencias a las llamadas a funciones tienen en las pruebas de software [116], [111].

Para apoyar la búsqueda de datos de los algoritmos de generación de datos, se presenta una nueva técnica para realizar transformaciones de capacidad de prueba (del inglés *testeability transformations*) para tratar específicamente con contenedores en la

programación orientada a objetos [111]. Los programas que tienen en el código variables del tipo "bandera", no permiten que la generación de datos mediante *SBST* sea eficiente, ya que en presencia de este tipo de variables, la búsqueda de datos degenera en una búsqueda aleatoria. Un algoritmo permite realizar transformaciones de capacidad de prueba [117] para tratar con programas que tengan este tipo de variables. Los predicados anidados también pueden causar problemas en la búsqueda de datos en *SBST*, porque la información requerida para guiar la búsqueda solo está disponible una vez que se satisface cada condicional anidado. Esto puede enlentecer el proceso de búsqueda y restringir el espacio de búsqueda disponible para la satisfacción de las restricciones. Una solución [118] presentada como transformación de capacidad de prueba permite la evaluación temprana de todos los condicionales anidados en el programa.

4.2.2 PI2: clasificación de técnicas

Para responder la segunda pregunta de investigación se clasifican las técnicas de generación de datos de prueba que en los estudios seleccionados se reportan como más eficaces o eficientes, adoptando una de las clasificaciones utilizadas por Mahmod [27] y extendida por Romli *et al*. [31]. La clasificación utilizada en el presente artículo modifica y extiende la de Romli *et al*. [31], agregando el tipo de técnica y el contexto en el que se realizan los experimentos. La clasificación de la **Tabla 6** es realizada de acuerdo al método de implementación que utiliza la técnica (estático, dinámico o híbrido) y se agrupan en función del tipo de técnica (*SBST, Random Testing (RT) y Misc. Testing*). La columna contexto (Cont.) se refiere al tipo de software (industrial

(ind.) o software de pequeña escala (peq.)), en el que se realizan los experimentos con las técnicas. Además para cada artículo se incluye una breve descripción de la técnica usada para la búsqueda y generación de datos de prueba.

Para complementar el presente estudio se agregaron 7 artículos clasificados en la revisión de Romli *et al*. [31] que no se encontraron como estudios primarios. Estos artículos reportan experimentos con técnicas del tipo *SBST* y son identificados en la con un (*).

Tabla 6: Clasificación de técnicas para la generación.

Tipo	**Art.**	**Cont.**	**Técnica de generación**	**Método**
SBST	[111]	Peq.	*MA*	Dinámico
	[119]	Peq.	*MA* basado en *GA*	Dinámico
	[120]	Peq.	*Ant Colony*	Dinámico
	[121]	Peq.	*Artificial Immune Algorithm (AIA)*	Dinámico
	[122]	Ind.	*Differential Evolution (DE)*	Dinámico
	[123]	Ind.	*GA*	Dinámico
	[124]	Peq.	*GA*	Dinámico
	[125]	Peq.	*GA*	Dinámico
	[126]	Peq.	*GA*	Dinámico
	[127]	Peq.	*GA* basado en una técnica de evaluación de adecuación (*FEP*)	Dinámico
	[128]	Peq.	Combinación de *GA* con *Artificial Immune System (AIS)*	Dinámico
	[129]	Peq.	*GA* con una función de adecuación basada en *dominance relations*	Dinámico
	[130]	Peq.	*GA (many paths coverage)*	Dinámico
	[131]	Peq.	*Genetic PSO*	Dinámico
	[132]	Peq.	*Genetic PSO*	Dinámico

Tipo	Art.	Cont.	Técnica de generación	Método
	[133]	Peq.	*Genetic SA*	Híbrido
	[38]	Peq.	*Genetic SA*	Dinámico
	[40]	Ind.	*Hybrid MA*	Dinámico
	[134]	Peq.	*Improved PSO*	Dinámico
	[135]	Peq.	*Improved Adaptive PSO (APSO)*	Dinámico
	[136]	Peq.	*Multi-population GA*	Dinámico
	[137]	Peq.	*Multi-population GA*	Dinámico
	[138]	Peq.	*Multi-population GA*	Dinámico
	[139]	Peq.	*Random Walk Based Algorithm*	Dinámico
	[112]	Peq.	*Tabu GA*	Dinámico
	[140]	Peq.	*Tabu GA*	Dinámico
	[141] (*)	Peq.	*Ant Colony*	Híbrido
	[142] (*)	Peq.	*Hybrid self-adaptive and multi-objective evolutionary algorithms*	Híbrido
	[143] (*)	Peq.	*Batch-optimistic, close-up algorithms*	Dinámico
	[144] (*)	Peq.	*Domain reduction and evolutionary testing*	Dinámico
	[145] (*)	Peq.	Funciones de adecuación*: normalized extended Hamming distance* y *branch predicate* con *GA*	Dinámico
	[146] (*)	Peq.	Función de adecuación (MC/DC)	Dinámico
	[147] (*)	Peq.	Data state scarcity search strategy	Dinámico
RT	[148]	Peq.	*Path-oriented random test data generator*	Dinámico
	[108]	Peq.	*Path-oriented random test data generator - constraint reasoning*	Híbrido
Misc. Testing	[149]	Ind.	Combinación de *Concolic Testing* con *SBST*	Híbrido
	[106]	Peq.	Combinación de RT con *Invariant Technology y GA*	Dinámico
	[150]	Peq.	*Concolic Testing y Abstraction Refinement and Coarsening (ARC)*	Híbrido
	[151]	Peq.	Combinación *de GA con Concolic Testing (DSE)*	Híbrido

Tipo	Art.	Cont.	Técnica de generación	Método
	[56]	Ind.	*Hybrid Concolic Testing*	Híbrido
	[115]	Peq.	Combinación de *Lazy symbolic execution , ejecución simbólica y real*	Híbrido
	[114]	Peq.	Algoritmo para *dynamic pointer data (path-oriented)*	Dinámico
	[152]	Peq.	*SBST* y un esquema dinámico	Dinámico
	[153]	Peq.	*path-wise (IIRM)*	Dinámico

Nota: *GA* es *Genetic Algorithm* (Algoritmo Genético). *PSO* es *Particle Swarm Optimization*. *SA* es *Simulated Annealing*. *MA* es *Memetic Algorithms*. *RT* es *Random Testing* (Pruebas Aleatorias). Art. es Artículo.

4.2.2.1 Comparación con revisiones anteriores y tendencias

Los resultados de la **Tabla 6** indican que las técnicas para la automatización de la generación de datos de prueba estructurales con más estudios empíricos son las de *SBST*. La tendencia hacia el uso de *SBST* puede verse representado gráficamente en un artículo [109] y también en la clasificación de técnicas de Romli [31]. Para la generación de datos con este tipo de técnica se experimenta con distintos algoritmos meta-heurísticos (*GA, AIA, DE, PSO, SA*), con mejoras a éstos y con algoritmos meméticos (del inglés *memetic algorithms: MA*). Los algoritmos meméticos [154] son metaheurísticas que usan búsqueda global y local (por ejemplo se combina *GA* con *SA*). En el área de *SBST* el algoritmo más utilizado es *GA* y sus combinaciones, seguido por *PSO* y sus mejoras y combinaciones. El uso frecuente de *GA* en *SBST* también es observado en Ali *et al*. [155]. Algunas de las posibles razones que explican el uso frecuente de los *GA* son [155]:

- Existen numerosas publicaciones de la aplicación de *GA* y datos empíricos de las configuraciones de los parámetros requeridos para su aplicación, sumado al hecho de que hay libros acerca de los *GA* que permiten que sea fácil de aplicarlos en distintos contextos.
- En algunos casos los *GA* han demostrado tener mejor performance que otros algoritmos de búsqueda locales, pero no hay evidencia de que los *GA* sean mejores que otros algoritmos de búsquedas globales.
- Los *GA* tienen buenas implementaciones en la forma de herramientas comerciales y *frameworks*, que generalmente facilitan su aplicación.

Mediante la utilización de distintos algoritmos y funciones se investiga algunas propuestas que consideran más de un objetivo de prueba [136], [137], [138]. Estas propuestas son realizadas para obtener técnicas que consideren más de una característica en los datos de prueba. Por ejemplo, capacidad para revelar cierto tipo de defecto, tiempo de ejecución reducido y consumo de memoria, y otros factores asociados al entorno de desarrollo. El enfoque de éstas técnicas es considerar a la generación como un problema multi-objetivo, dado que depende de múltiples variables [136]. La funciones de adecuación son también uno de los temas de investigación reportado en revisiones anteriores [27], [31]. En el presente estudio se identifican algunos experimentos con el objetivo de proponer y comparar nuevas funciones de adecuación [124], [127], [126], [129], [125].

Con respecto a los métodos de implementación de las técnicas de la **Tabla 6**, son dinámicos e híbridos, no encontrándose técnicas cuyo método de implementación sea puramente estático. El 74,46 % de las técnicas son implementadas en forma dinámica y

el resto en forma híbrida. La tendencia hacia métodos de implementación dinámicos e híbridos se debe a los problemas que presentan los métodos puramente estáticos. Así como también, los métodos de implementación híbridos tratan de obtener los beneficios de los estáticos y dinámicos. La tendencia hacia los métodos de implementación dinámicos e híbridos ha sido observada también en los estudios de Mahmod [27] y Romli [31]. Con respecto al tipo de software con el que se experimenta los resultados indican que aproximadamente el 88,88 % de la investigación se realiza con software de pequeña escala.

En la **Tabla 7** se hace un resumen de las principales tendencias de investigación encontradas en el estudio.

Tabla 7: Tendencias de investigación

Tendencia	
Técnicas más estudiadas	Las técnicas más estudiadas son las de *SBST*
Tipos de algoritmos y funciones	Se experimenta con *GA, AIA, DE, PSO, SA* y con algoritmos meméticos. Se investigan también algoritmos multi-objetivo. Hay nuevas propuestas y comparaciones de funciones de adecuación [124], [127], [126], [129], [125]
Algoritmos con más experimentos	En el área de *SBST* el algoritmo más utilizado es *GA* y sus combinaciones, seguido por *PSO* y sus mejoras y combinaciones
Métodos de implementación de las técnicas	Los métodos de implementación con los que se experimenta son: • Dinámicos (74,46 %) • Híbridos (25,54 %)
Tipo de software con el que se experimenta	El 88,88 % de la investigación se realiza con software de pequeña escala

4.3 Amenazas a la validez

Los resultados de un estudio de mapeo pueden ser afectados por los investigadores que realizan la investigación, por las bibliotecas digitales seleccionadas, por los términos de búsqueda y por la ventana de tiempo seleccionada [102]. Naturalmente, esto plantea una serie de amenazas a la validez de los resultados que son analizadas en la presente sección.

4.3.1 Validez de conclusión

La validez de conclusión se refiere a la relación entre el tratamiento y el resultado [156]. Una amenaza a la validez de conclusión se puede dar en la extracción de datos. Para disminuir esta amenaza se desarrolló un formulario de extracción de datos que asegura que los datos relevantes extraídos sean consistentes. Otra amenaza a la validez el bajo número de estudios por categoría.

4.3.2 Validez del constructo

La validez del constructo asegura que la construcción del estudio esté relacionada con el problema de investigación y que las fuentes seleccionadas sean relevantes [156]. Una posible amenaza a la validez del constructo puede ser el sesgo en el proceso de selección de estudios primarios. Esta amenaza se disminuye con la construcción de un protocolo de investigación. El protocolo incluye las preguntas de investigación, los criterios de inclusión y exclusión, la cadena de búsqueda, la estrategia de búsqueda y selección de datos. Con el objetivo de obtener la máxima cantidad de estudios primarios relevantes, la cadena de búsqueda se construyó en forma sistemática. Los términos del

tema de interés en primera instancia se obtuvieron de algunas sugerencias de términos de revisiones anteriores [27] y se fueron ajustando durante las búsquedas preliminares realizadas. La lista de sinónimos relacionados con el término experimento se obtuvo de un artículo [101] que presenta estrategias para obtener experimentos relevantes.

4.3.3 Validez interna

La validez interna se refiere a la conexión entre la observación y las explicaciones propuestas [156]. Esto asegura que las conclusiones extraídas sean verdaderas. Esta amenaza se disminuye con un proceso definido de selección de artículos. La inclusión y exclusión de artículos se realizó por dos investigadores en forma independiente lo que disminuye la amenaza de la validez interna.

4.3.4 Validez externa

La validez externa se refiere a la posibilidad de generalización de los resultados [156]. En la presente revisión se consideran solo estudios a partir del año 2006, lo que puede ser una amenaza para la generalización de los resultados. Para disminuir esta amenaza, los resultados se han comparado y complementado con los de otras revisiones similares. La exclusión de algunos estudios (por ejemplo, los relativos a las herramientas) puede implicar una amenaza a la representatividad del mapeo. Por otra parte, dado el procedimiento sistemático seguido durante el estudio de mapeo, los resultados y las conclusiones generales derivadas de ellos tienen validez dentro del dominio de investigación abordado, permitiendo que los resultados del estudio puedan ser usados como un punto de partida para otras líneas de investigación.

5. Ejecución de estudio de caso

5.1 Diseño del estudio de caso

5.1.1 Objetivo

El objetivo del estudio de caso es explorar el aporte que realizan dos herramientas basadas en *SBST,* desde el punto de vista de la generación de prueba unitaria. *SBST* surgió como uno de los más estudiados experimentalmente en el estudio de mapeo. A través del estudio de caso se explora qué cantidad de casos de prueba generan, qué cobertura de rama se obtiene, si los defectos en los programas impiden que las herramientas logren alcanzar una alta cobertura, así como también explorar qué aportan desde el punto de vista de la detección de defectos en el software. El aporte con respecto a la detección de defectos se refiere no solo a la cantidad de defectos encontrados sino que además se plantea explorar cuál es el formato de los casos de prueba generados por cada una de las herramientas.

5.1.2 Preguntas de investigación

Se definieron las siguientes tres preguntas de investigación (PI) de acuerdo a los objetivos del caso de estudio.

- PI1: ¿Cuántos casos de prueba generan cada una de las herramientas?
- PI2: ¿Existen defectos en el código que pueden afectar la cobertura alcanzada por las herramientas?

- PI3: ¿Qué cantidad y qué tipo de defectos se detectan con cada una de las herramientas?

5.1.3 Selección de sujetos

5.1.3.1 Sujeto de prueba

El sujeto de prueba es un programa escrito en Java llamado *Conversor* cuyo propósito es convertir una lista de medidas en el sistema de unidades estadounidense al sistema internacional (métrico). El programa ha sido utilizado en otros experimentos de prueba, aplicando otras técnicas manuales, que no son parte de este estudio. Se utiliza este programa porque se conocen los principales defectos y porque brinda la posibilidad de hacer comparaciones posteriores.

El programa *Conversor* pide como entrada una línea de la forma <valor> <unidad>, que puede contener una sola medida, aunque en algunos casos la medida pueden expresarse utilizando una segunda unidad para la fracción. Al procesar la línea el programa muestra por la consola una línea de salida. En el caso de que no se pueda procesar una línea del archivo de entrada se muestra el mensaje de error correspondiente. El programa tiene 665 instrucciones en 147 líneas de código con un total de 92 ramas.

En la **Tabla 8** se presentan los defectos que han sido sembrados en el programa *Conversor*. De los muchos defectos que se pueden sembrar en un programa Java, se sembraron algunos que habitualmente introducen los programadores y con el propósito de explorar si afectan la cobertura de las herramientas. Los defectos del 1 al 7 y el

defecto 11 se pueden detectar con técnicas de prueba estructurales. Los demás son defectos encontrados a partir de la especificación del programa.

Tabla 8: Defectos encontrados en el programa *Conversor*.

#	Categoría	Fallo
1	Asignación de valor a un boolean (=), en vez de usar un (==)	Permite combinaciones de unidades no válidas
2	Error de escritura en sentencia *switch*	Retorna como no válida una unidad válida
3	División por cero	Error lógico
4	Uso de (==), en vez de *equals*	No identifica correctamente las unidades
5	Excepción de conversión de *string* a entero no controlada	Lanza una excepción cuando se introducen caracteres no numéricos.
6	Código no alcanzable	Error al controlar la validez de uno de los valores de entrada del programa.
7	Punto y coma luego de un *if*	Condición en el código no evaluada
8	Omisión de etiqueta en error cálculo	Permite una mayor cantidad de onzas por libra.
9	Defecto en los mensajes de la interfaz de usuario	Error en mensaje "Combinación de unidades"
10	Acepta entradas extendidas	No da error si hay más de dos decimales.
11	Omisión de control, error de operador de comparación	Permite que el segundo valor sea igual al unitario.

5.1.3.2 Herramientas para la generación de casos de prueba

Las herramientas seleccionadas generan casos de prueba para clases escritas en código Java y oráculos. Para generar los casos de prueba usan un enfoque basado en búsqueda con el objetivo de maximizar la cobertura estructural y generan los oráculos conjuntos de *assert*. Los *assert* permiten a los profesionales detectar desviaciones del comportamiento esperado del programa aunque no son oráculos completos dado que no son generados a partir de una especificación del programa.

Las herramientas seleccionadas son *Testful* [73] y *Evosuite* [70]. No se han encontrado comparaciones directas de ambas herramientas.

5.1.3.2.1.1 Herramienta Evosuite

Evosuite[1] usa un enfoque evolutivo para generar los conjuntos de casos de prueba. 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) [71]. 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. El operador cruce produce descendencia de conjunto de casos de prueba mediante el intercambio de casos de prueba de dos padres individuales, y el operador de mutación o bien añade casos de prueba generados aleatoriamente, o muta casos de prueba individuales. La mutación de

[1] http://www.evosuite.org/

los casos de prueba puede agregar, eliminar o cambiar las llamadas a métodos en una secuencia. La aptitud (del inglés *fitness*) se calcula con respecto a la cobertura de rama, utilizando un cálculo basado en la medición de distancia de rama. La distancia de rama estima que tan cerca una rama va a evaluar a verdadero o falso para una ejecución en particular. Para cada rama se considera la distancia de rama mínima sobre todos los casos de prueba de un conjunto de pruebas. La aptitud general de un conjunto de pruebas es la suma de estos valores mínimos, de tal manera que un individuo con cobertura de rama 100% tiene aptitud 0 [71].

Para generar conjunto de casos de prueba *Evosuite* requiere el *bytecode* de la clase Java bajo prueba y sus dependencias. El *bytecode* es analizado e instrumentado, y al final de la búsqueda genera conjunto de casos de prueba *JUnit* para una determinada clase [70].

En las pruebas estructurales, las pruebas son generadas desde el código fuente con el objetivo de satisfacer un criterio de cobertura [157]. Las herramientas comúnmente seleccionan un objetivo de cobertura a la vez, por ejemplo una rama del programa o un camino de flujo de control, y generan un caso de prueba para que ejercite este objetivo particular. Hay un problema con los enfoques que abordan un objetivo de prueba por vez: asume que todos los objetivos de cobertura son igualmente importantes, igualmente difíciles de alcanzar, e independientes el uno del otro. Asumir estas hipótesis conlleva algunos problemas. Muchos criterios de cobertura no son factibles, lo que significa que no existen pruebas que los ejerciten. Aún si son factibles, algunos objetivos de cobertura son más difíciles de satisfacer que otros. Por lo tanto, el resultado de la generación de prueba depende del orden de los objetivos de cobertura elegidos y cuántos de ellos son factibles [70]. Para evitar los problemas que conllevan el enfoque de abordar un

objetivo de prueba a la vez, *Evosuite* usa un enfoque llamado generación de casos de prueba total (del inglés *whole test suite generation*). Este enfoque no produce casos de prueba individuales para objetivos de cobertura individuales, sino que se centra en conjuntos de casos de prueba dirigidos a un criterio de cobertura completo al mismo tiempo. Esto evita que los resultados sean afectados por el orden o la dificultad o la inviabilidad de los objetivos de prueba individuales.

Evosuite propone mantener la cantidad total de conjuntos de prueba tan pequeña como sea posible. Este enfoque tiene algunas ventajas, como por ejemplo, su eficacia no se ve afectada por el número de objetivos no alcanzables en el código. Para producir un conjunto reducido de *assertions Evosuite* usa pruebas de mutación que maximizan el número de defectos sembrados en una clase [70].

Evosuite además de generar casos de prueba, genera un reporte que consiste en archivos de datos *CVS* (datos de cada una de las ejecuciones) y un reporte en *HTML*.

Evosuite no tiene restricciones con respecto a los tipos de datos, ya que puede manejar *arrays* y objetos de cualquier clase [70], pero tiene las siguientes limitaciones:

1. Solo trata con aplicaciones de hilos únicos [70].
2. Soporte básico para el manejo de *strings* [71].
3. Soporte limitado para *Java generics* y el manejo de tipos de datos dinámicos [71].
4. Manejo de ramas en código de manejo de excepciones [71].

En la **Tabla 9** se presenta un resumen de las características principales de *Evosuite*.

Tabla 9: Características de *Evosuite*.

Característica	Valor
Enfoque	Evolutivo
Algoritmo de generación	Algoritmo genético
Tipo de software	*Java*
Tipo de casos de prueba	*JUnit*
Tecnología de la herramienta	• *SBST* • Generación de casos de prueba total • Conjunto reducido de *assertions*
Efectividad y escalabilidad	Ver: • [158] • [71]

5.1.3.2.1.2 Testful

Testful[2] es una herramienta para la generación semiautomática de prueba unitaria para sistemas con estado que utiliza enfoques basados en búsqueda. Es considerada semiautomática porque se le solicita al usuario algunos datos para aumentar la eficiencia del enfoque.

Testful aborda particularmente el problema de la generación de pruebas para los sistemas con estados. En estos sistemas los objetos se deben llevar a estados adecuados y se debe proporcionar los valores correctos para los parámetros de entrada de los métodos bajo prueba. Al igual que *Evosuite* el enfoque de *Testful* no se focaliza en un objetivo de prueba a la vez (por ejemplo una rama particular). *Testful* considera que las pruebas para los sistemas con estados están conceptualmente compuestas de dos partes: una que crea los estados deseados de los objetos, y otra que ejercita cada uno de estos estados.

Testful genera casos de prueba *JUnit* para clases y métodos Java con técnicas basadas en búsqueda haciendo que los estados de los objetos evolucionen. Usa tanto cobertura de

[2] https://code.google.com/p/testful/

sentencia como de rama en la clase bajo prueba, combinando un algoritmo evolutivo (búsqueda global) con uno *hill climbing* (búsqueda local). Los algoritmos evolutivos son considerados como una forma de búsqueda global porque ejercitan varios puntos a la vez en el espacio de búsqueda. En cambio, *hill climbing* es descrito como búsqueda local porque considera solo una solución a la vez y hace movimientos en la vecindad local de esas soluciones.

En *Testful* la búsqueda local integra la búsqueda global para formar un enfoque híbrido. El algoritmo evolutivo lleva a los objetos a estados útiles que son usados por el algoritmo local para ejercitar partes no cubiertas de la clase.

Testful ha sido comparado con *jAutoTest* [159], *eToc* [69] y *Randoop* [84] con respecto a la cobertura lograda demostrando tener una mejor performance [74].

En la **Tabla 10** se presenta un resumen de las características principales de *Testful*.

Tabla 10: Características de *Testful*.

Característica	**Valor**
Enfoque	Evolutivo
Algoritmo de generación	Algoritmo memético
Tipo de software	*Java*
Tipo de casos de prueba	*JUnit*
Tecnología de la herramienta	• *SBST* • Se focaliza en más de un objetivo de prueba a la vez • Crea los estados deseados de los objetos, y otra ejercita cada uno de estos estados
Efectividad y escalabilidad	Ver: • [74]

5.1.3.3 Configuración del estudio de caso

La versión de *Evosuite* es la 20130318 y la de *Testful* testful-2.0.0.alpha y su correspondiente instrumentador instrumenter-2.0.0.alpha. El entorno de desarrollo para el experimento es el siguiente: SUN Java 1.6.0_34 y Eclipse Juno IDE. Para calcular la

cobertura de rama se utiliza la herramienta *EclEmma* [160] que es una herramienta libre para evaluar la cobertura de código Java. La versión de *EclEmma* es la 2.2.0.

Para que las herramientas no tengan problemas con el acceso a métodos por ser privados, todos los métodos de la clase del programa *Conversor* se cambiaron a públicos.

En la **Tabla 11** se presenta un resumen de la configuración del estudio de caso.

Tabla 11: Configuración del estudio

Configuración	Valor
Versión de *Evosuite*	20130318
Versión de *Testful*	• testful-2.0.0.alpha • instrumenter-2.0.0.alpha
Versión de *EclEmma*	2.2.0
Entorno de desarrollo	• SUN Java 1.6.0_34 • Eclipse Juno IDE

5.1.4 Proceso de estudio y variables

La ejecución del estudio de caso consiste en tres pasos:

- Paso1: se generan casos prueba aplicando las herramientas en el programa *Conversor con los defectos conocidos*. En este se responde la pregunta de investigación PI1 y se obtienen las siguientes medidas y datos:
    - Tiempo que tardan las herramientas en generar los casos de prueba.
    - Cantidad de casos de prueba.
    - Cobertura de rama.
    - Ejemplos de cada uno de los casos de prueba.
- Paso 2: se generan casos prueba aplicando las herramientas en el programa *Conversor* sin los defectos conocidos. En este paso se responde la pregunta de investigación PI2 Para contestar esta pregunta se quitaron todos los defectos

conocidos en el código y se generaron sobre él los casos de prueba. Se compara la cantidad de casos generados con los defectos y sin los defectos. Las medidas y datos obtenidos en este paso son los siguientes:

  - Cobertura de rama con y sin los defectos conocidos en el software.
  - Identificación de defectos que impiden lograr mayor cobertura.

- Paso 3: se generan casos prueba aplicando las herramientas en el programa *Conversor* sin los defectos que impidan la máxima cobertura por parte de las herramientas. En este paso se responde la pregunta de investigación PI3. Las medidas y datos obtenidos en este paso son los siguientes:
  - Defectos encontrados con cada una de las herramientas.

Las variables independientes del estudio incluyen el número y el tipo de defectos sembrados en el programa *Conversor*, el programa elegido y las herramientas de generación de casos de prueba. Las variables dependientes son el número de casos de prueba generados y el número de defectos encontrados con los casos de prueba generados.

5.2 Resultados

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

5.2.1 PI1: Cantidad de casos de prueba

En la **Tabla 12** se presenta la cantidad de casos de prueba y el tiempo empleado para generarlos por cada una de las herramientas. Durante la generación *Testful* no finalizó

correctamente. Se investigó la razón y se concluyó que uno de los defectos en el código, el defecto #5 de la **Tabla 8**, no permitía que la generación finalizara. Por lo que se tuvo que agregar en el código una excepción controlada para este error para poder finalizar con la generación de los casos de prueba.

Tabla 12: Tiempo y cantidad de casos generados

Herramienta	Tiempo	#Casos
Evosuite	9’ 58’’	38
Testful	13’ 06’’	18

Evosuite generó una mayor cantidad de casos de prueba y empleó un tiempo menor que *Testful* en generarlos. Los resultados de la **Tabla 12** muestran que *Evosuite* es más eficiente que *Testful*, pero surgen algunas preguntas considerando este resultado. Una pregunta posible es si al generar más cantidad de casos *Evosuite* logra una mayor cobertura. Otra pregunta es qué diferencia hay entre los casos de prueba generados por las dos herramientas. Un escenario común en las pruebas de software es que después de generados los datos de prueba el profesional agregue los oráculos de prueba [157]. En este contexto el profesional debe entender las pruebas generadas para poder generar oráculos eficaces. Si bien las herramientas investigadas generan oráculos en términos de *assert,* éstos aún no son eficaces y puede que se requiera adaptarlos. Para ayudar al profesional es importante también generar pequeños conjuntos de prueba con una buena cobertura de código. *Evosuite* genera casos de prueba *JUnit* con el formato que se puede ver en la **Figura 11**.

```
@Test
public void test0()  throws Throwable  {
       Conversor conversor0 = new Conversor();
       String string0 = conversor0.procesarLinea("   --> Linea  vacia");
       assertNotNull(string0);
       assertEquals("  --> Linea vacia --> No se procesa, linea con 5 o más
       elementos", string0);
}
```

Figura 11: Ejemplo de caso de prueba generado por *Evosuite*.

El reporte *HTML* de *Evosuite* contiene una entrada para cada una de las ejecuciones (por ejemplo una para cada ejecución en una clase), lista de los parámetros de ejecución, los casos de prueba y la cobertura. Es posible también, configurando un propiedad en la generación, obtener gráficos de la aptitud, largo, y tamaño de las historias.

Un ejemplo de los casos de prueba generados por *Testful* se puede ver en la **Figura 12**. *Testful* incluye uno o más casos de prueba en métodos públicos.

```
public void testFul5() throws Exception {
       mesconv.Conversor mesconv_Conversor_0 = null;
       java.lang.String java_lang_String_0 = null, java_lang_String_1 = null,
       java_lang_String_2 = null;
       java_lang_String_0 = "";
       java_lang_String_1 = "\u00D5\r=}5C";
       mesconv_Conversor_0 = new mesconv.Conversor();
       java.lang.String tmp0 = mesconv_Conversor_0.procesarLinea(java_lang_String_0);
       java_lang_String_2 = (java.lang.String)  tmp0;
       assertEquals(" --> Linea vacia", tmp0);
       java.lang.String tmp1 = mesconv_Conversor_0.procesarLinea(java_lang_String_1);
       assertEquals("\u00D5\r=}5C --> No se procesa, linea con 1 elemento", tmp1);
       java.lang.String tmp2 = mesconv_Conversor_0.procesarLinea(java_lang_String_2);
       assertEquals(" --> Linea vacia --> Primera unidad no valida", tmp2);
}
```

Figura 12: Ejemplo de casos de prueba generados por *Testful*.

Los casos de prueba generados por las herramientas se pueden comparar cualitativamente en términos de legibilidad. Cuanto mayor sea la legibilidad de las pruebas son más fáciles de entender por parte de los profesionales. La legibilidad es definida como el juicio de un humano acerca de la dificultad de entendimiento de un texto [161]. La legibilidad de un programa está relacionada con su capacidad de mantenimiento. *Testful* puede agrupar más de un caso de prueba en un solo método accediendo a diferentes clases y utilizando diversas variables para poder satisfacer los requerimientos del caso. En cambio, *Evosuite* genera casos de prueba más pequeños enfocados en una funcionalidad específica. En este sentido, a juicio del autor, *Evosuite* genera casos de prueba más legibles que *Testful*.

5.2.2 PI2: Cobertura de las herramientas

Para contestar la pregunta de investigación PI2 se quitaron todos los defectos sembrados en el código y se generaron sobre él los casos de prueba. Se puede observar en la **Tabla 13** la cobertura alcanzada por las herramientas con y sin los defectos identificados en el código. La cobertura de rama de *Evosuite* es significativamente mejor que la alcanzada por *Testful* en ambos casos.

Tabla 13: Cobertura de rama con y sin los defectos.

Herramienta	% Cobertura con los defectos	% Cobertura sin los defectos
Evosuite	76,8	90,2
Testful	43,9	68,3

La cobertura de las dos herramientas aumentó notoriamente luego que se quitaron los defectos. El defecto #1, que es la asignación de un valor a un *boolean* en el código

impedía que se lograra una mayor cobertura. Esta asignación está localizada en un punto del código que tiene involucrada una decisión que de ser verdadera se ejecutan otros métodos. Aún luego de haber quitado el defecto en el código hay ramas que no son cubiertas por las herramientas. La rama 1 en la **Figura 13** no es cubierta totalmente. Existe 1 de las dos ramas no cubierta.

```
1: if (combinacionUnidadesValida(unidad1,
unidad2)) {

2:  if (segundoValorValido(valor2, unidad2)) {

    valorSalida = convertirValor(valor1, unidad1)
    + convertirValor(valor2, unidad2);

    salida = valorSalida + " " +
    unidadSalida(unidad1);

3:  } else {

    salida = "Segunda cantidad igual o mayor al
    valor unitario de la primera unidad";

    }

4: } else {

    salida = "Combinacion de valores no valido";

   }
```

Figura 13: Parte del código parcialmente cubierto por las herramientas.

Para la condición 2 de la **Figura 13**, las herramientas no consiguen cubrir ninguna de las dos ramas correspondientes. El código del método consiste en las líneas de código mostradas en la **Figura 14**.

```
public boolean segundoValorValido(double valor,
UnidadUS unidad) {
    boolean valido = false;
    if ((unidad == UnidadUS.in && valor <= 12)
    || (unidad == UnidadUS.ft && valor <= 3)
    || (unidad == UnidadUS.oz && valor <= 19)) {
              valido = true;
    }
    return valido;
}
```

Figura 14: Código de método no cubierto.

La cobertura de rama obtenida en el estudio de caso con *Testful* (68,3 %) es baja comparada con la obtenida en evaluaciones anteriores de la herramienta [73], [74], [162]. En estas evaluaciones *Testful* obtiene una mayor cobertura que otras herramientas como *jAutotest* [163], *Randoop* [84], *eToc* [69]. La cobertura reportada en las evaluaciones varía entre 79,0 % y 85,2 %. 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 [71].

5.2.3 PI3: Defectos detectados

En esta sección se responde la última pregunta de investigación PI3: ¿Qué cantidad y qué tipo de defectos se detectan con cada una de las herramientas?

Con los casos de prueba generados por *Evosuite*, en el código con los defectos reportados en la **Tabla 8**, se pudo detectar los defectos #2 y #3. El defecto #2 correspondiente a un error de escritura en el código de una sentencia *switch*, se detecta por el caso de prueba de la **Figura 15**.

```
@Test
public void test25()  throws Throwable  {
  Conversor conversor0 = new Conversor();
  Conversor.UnidadUS conversor_UnidadUS0 =
  conversor0.identificarUnidad("in");
  assertEquals("noValida", conversor_UnidadUS0.name());
}
```

Figura 15: Caso de prueba generado por *Evosuite* que detecta defecto de conversión.

En este caso la salida del caso de prueba es "in" pero se espera una salida con el texto "no Valida". Para el caso del defecto en el código #5, *Evosuite* genera el caso de prueba de la **Figura 16**. Si bien *Testful* cubre algunas de las líneas de código donde están los defectos #4 y #7, los casos de prueba generados no reportan ningún defecto en el programa.

```
@Test
 public void test26()  throws Throwable  {
  Conversor conversor0 = new Conversor();
  // Undeclared exception!
  try {
    conversor0.convertirMedidaCompuesta("yd", "yd", "yd", "yd");
    fail("Expecting exception: NumberFormatException");
  } catch(NumberFormatException e) {
        /*
         * For input string: \"yd\"
         */
  }
 }
```

Figura 16: Caso de prueba generado por *Evosuite* que detecta la división por cero.

En resumen, la tasa de detección de defectos de cada una de las herramientas calculada como el cociente entre la cantidad de defectos detectados y la cantidad de defectos en el

código detectables por técnicas de caja blanca (defectos del 1 al 7: **Tabla 8**) es 0,28 (2/7) para *Evosuite* y 0,0 (0/7) para *Testful* (ver **Tabla 14**).

Tabla 14: Datos obtenidos en el estudio de caso.

Variable / Herramienta	***Evosuite***	***Testful***
Tiempo	9’ 58’’	13’ 06’’
Cantidad de casos de prueba	38	18
Cobertura con los defectos	76,8	43,9
Cobertura sin los defectos	90,2	68,3
Cantidad de defectos detectados	2	0

En este estudio, no se encontró la misma efectividad para la detección de defectos reportadas en otros [82], [83], en los que algunas herramientas aleatorias han demostrado tener una efectividad en la detección de defectos similar a la de las pruebas unitarias manuales. En este sentido, el estudio de caso concluye que 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. Además, 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 los generan actualmente, ejecutando código incorrecto [87], la detección de defectos es poco probable.

5.3 Amenazas a la validez

Existen varias amenazas a considerar en un estudio que compara dos herramientas. La primera amenaza a considerar es la validez interna del estudio de caso, es decir si hubo algún sesgo en el diseño experimental que pueda afectar los resultados obtenidos.

5.3.1 Validez interna

Una posible fuente de sesgo es que una determinada configuración de una herramienta pueda favorecerla sobre la otra. En este estudio no hubo que modificar la configuración estándar de las herramientas. Otra posible amenaza a la validez interna es la cantidad, el tipo de defectos inyectados en el programa y en qué lugar del código son inyectados. El presente estudio es exploratorio y plantea determinar si los defectos pueden afectar o no la cobertura y no es el objetivo final determinar específicamente cuánto se ve afectado.

5.3.2 Validez externa

Otra posible fuente de sesgo es la elección del programa usado en el estudio de caso, que 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 un programa pequeño 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.

6. Conclusiones

El campo de la generación automática de casos de prueba ha hecho enormes progresos en los últimos años. Muchos de los enfoques para la generación como las pruebas aleatorias, pruebas basadas en búsqueda y ejecución simbólica dinámica pueden lograr una alta cobertura. Pero, a pesar de los avances logrados, todavía las herramientas que implementan estos enfoques presentan algunas deficiencias que limitan su amplio uso en la industria.

De la clasificación de las técnicas realizada en el estudio de mapeo de la tesis se puede determinar que una amplia mayoría de la investigación se realiza con software de pequeña escala. La evidencia obtenida el estudio indica que la investigación está focalizada en técnicas de *SBST* mediante el uso de algoritmos genéticos. Si bien la investigación se está diversificando en los tipos de algoritmos utilizados y en los problemas abordados, se requieren más estudios empíricos para demostrar su aplicabilidad con software más grande y complejo.

En lo que respecta a los problemas para la generación, en el estudio de mapeo realizado en la tesis se identificaron los más investigados y resultaron las siguientes categorías:

- Generación de datos de prueba: para buscar aumentar la eficacia y la eficiencia de la generación de datos con este tipo de técnica se experimenta con distintos algoritmos meta-heurísticos, con combinaciones y mejoras a éstos y con algoritmos meméticos. Se puede concluir que los factores principales con los que se experimenta para lograr una mayor eficacia y eficiencia en la generación

automática de datos de prueba, son las funciones de adecuación y la combinación y mejora de algoritmos.

- Tratamiento de estructuras de programas: se investigan diversas propuestas para el tratamiento de tipos de datos definidos por el usuario, estructuras de datos dinámicas y punteros, funciones de bibliotecas y estados internos de los objetos en la programación orientada a objetos.
- Optimización del conjunto de prueba y reducción del espacio de búsqueda: para solucionar estos problemas se encontraron algunos experimentos para probar algoritmos y técnicas de análisis estático.

El estudio de mapeo realizado en la tesis provee a los profesionales una visión general de los enfoques existentes y de los objetivos que se busca lograr con cada uno y en qué contexto puede ser eficiente su aplicación. Para los investigadores, proporciona una visión general de los distintos enfoques para la generación con el fin de identificar huecos en la investigación y poder establecer futuras líneas de investigación.

En el contexto de las pruebas estructurales, esta tesis se enfocó primero en identificar las técnicas de generación automática de casos de prueba y sus problemas, para luego seleccionar del conjunto resultado algunas de las herramientas que implementen los enfoques principales con el objetivo de explorar su utilidad para los profesionales. Para lograr el objetivo se plantaron tres preguntas de investigación en un estudio de caso.

Las conclusiones generales del estudio de caso son que las herramientas utilizadas no lograron una alta cobertura como se esperaba y la efectividad en la detección de defectos es baja comparada con la obtenida en otros estudios como los que se reportan de las herramientas de generación aleatoria. Asimismo, los defectos pueden causar

problemas a las herramientas para lograr una buena cobertura. Esta situación condiciona la forma en la que los profesionales pueden utilizarlas. Para lograr una cobertura satisfactoria el profesional, de forma manual, por ejemplo por medio de la inspección de código debería detectar y eliminar la mayor cantidad de defectos previo a la generación de los casos de prueba con las herramientas. Si bien las herramientas seleccionadas integran diferentes técnicas para lograr mayor detección de defectos y una mayor cobertura, las herramientas se podrían beneficiar con la integración de otras técnicas como el análisis estático para analizar previamente el código en búsqueda de defectos conocidos y luego sí comenzar con la generación de casos de prueba.

Por otra parte, en el estudio de caso encontró que las herramientas no son eficaces en la detección de defectos. Para mejorar la tasa de detección de defectos se debe continuar investigando la generación de oráculos de prueba más efectivos. Con estos resultados su utilidad se podría limitar a la generación de casos para pruebas de regresión y también para generar casos de prueba como base para el profesional. En todos los casos es particularmente importante la legibilidad de las pruebas generadas ya que el profesional debe interpretarlas correctamente para poder modificarlos.

A continuación se realiza un resumen de los resultados obtenidos en el estudio de caso para cada una de las preguntas de investigación:

- La primera pregunta de investigación (PI1) trata de identificar cuántos casos de prueba y cuál es la cobertura lograda por cada una de las herramientas en el código con los defectos inyectados. La respuesta es que *Evosuite* generó más casos de prueba y logró una mayor cobertura de rama que *Testful*. La cobertura lograda por *Evosuite* fue de un 76,8 %, mientras que la de *Testful* fue de 43,9 %.

- La segunda pregunta de investigación (PI2) explora si los defectos inyectados en el código afectan la cobertura de las herramientas. Se determinó que uno de los defectos, la asignación de una condición a una variable de tipo *boolean*, impide una mejor cobertura. La cobertura alcanzada luego de quitar los defectos conocidos es de 90,2 % para *Evosuite* y de 68,3 % para *Testful*.
- La tercera pregunta de investigación (PI3) explora qué cantidad de defectos son detectados por las herramientas. En el caso de *Evosuite* detecta 2 de los 11 defectos sembrados. Uno de los defectos es detectado por una condición no ejercitada en el código, y la otra es una excepción de conversión de *string* a entero no controlada. *Testful* no detectó ninguno de los defectos y ocurrió un error durante la generación debido a unos de ellos. Es de notar que aún luego de haber quitado el defecto en el código todavía quedan ramas en el código que no son cubiertas por las herramientas. En el estudio de caso se han presentado ejemplos de estas ramas para futuras investigaciones.

6.1 Publicaciones asociadas a este trabajo

Durante el transcurso de la investigación presentada en la tesis se elaboraron dos artículos. El primero se titula "Estudio de Mapeo Sistemático sobre Experimentos de Generación Automática de Casos de Prueba Estructurales" y fue aceptado en la XXXVIII Conferencia Latinoamericana en Informática (CLEI 2012) y presentado en Medellín (Colombia) el día 3 de Octubre de 2012 [164]. Esta publicación resume la primera parte de la investigación llevada a cabo para esta tesis.

El segundo artículo se denomina “Un Estudio de Caso de Herramientas de Prueba Basadas en Búsqueda”. Próximamente, este artículo se enviará a una conferencia para su correspondiente evaluación.

6.2 Líneas futuras de investigación

En esta sección se presentan las posibles líneas de investigación a seguir, tomando como base los aportes realizados en esta tesis.

- Para continuar con la investigación en lo que respecta al primer objetivo (identificar técnicas y problemas), se podría hacer una revisión sistemática por ejemplo de los distintos mecanismos para abordar el problema del oráculo en la generación automática de casos de prueba. La generación de oráculos en la parece ser un punto clave para la detección de defectos. Adicionalmente, se podría tomar cualquiera de los problemas encontrados referentes a la generación para profundizar aún más el tema con una revisión sistemática de la literatura.
- Realizar una revisión sistemática de la literatura de las técnicas para la generación que utilizan *DSE* y *SBST*, ya que esta combinación parece ser promisoria para solucionar los problemas específicos que tienen los enfoques *constraint-based testing* y *SBST*.
- Establecer un *benchmark* para la realización de experimentos con técnicas basadas en búsqueda para la generación automática de casos de prueba estructurales, que incluya la preparación de programas para ser probados, la inyección de defectos en los programas y la definición de la métricas para la evaluación de la eficiencia y la eficacia de las técnicas.

- Extender el estudio de caso de la tesis a otras herramientas, como por ejemplo algunas de *DSE*, así como incorporar otros programas que permitan generalizar las conclusiones.
- Realizar experimentos controlados y estudios de caso en contextos realistas para explorar el uso de herramientas de generación por parte de sujetos humanos y la combinación de las pruebas automáticas con las manuales. Es necesario continuar evaluando la fiabilidad de las pruebas generadas por las herramientas y en qué medida las pruebas automáticas pueden complementar a las manuales.

7. Referencias Bibliográficas

[1] J. Tian, *Software quality engineering: testing, quality assurance, and quantifiable improvement*. 1st. ed. Hoboken, New Jersey: John Willey & Sons, 2005.

[2] P. Bourque, R. Dupuis, and A. Abran, "The guide to the software engineering body of knowledge," *Software, IEEE*, vol. 16, no. 6, pp. 35–44, Nov. 1999.

[3] A. Bertolino, "Software Testing Research : Achievements, Challenges , Dreams," in *Future of Software Engineering. FOSE '07*, 2007, pp. 85–103.

[4] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, 2002.

[5] K. Lakhotia, P. McMinn, and M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN," *Journal of Systems and Software*, vol. 83, no. 12, pp. 2379–2391, Dec. 2010.

[6] K. Petersen and M. V. Mantyla, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," *7th International Workshop on Automation of Software Test (AST)*, pp. 36–42, Jun. 2012.

[7] M. J. Harrold, "Testing: A Roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 61–72.

[8] N. Tracey, J. Clark, and K. Mander, "Automated Program Flaw Finding using Simulated Annealing," in *ISSTA '98 Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis*, 1998, pp. 73 – 81.

[9] T. E. J. Vos, A. I. Baars, F. F. Lindlar, P. M. Kruse, A. Windisch, and J. Wegener, "Industrial Scaled Automated Structural Testing with the Evolutionary Testing Tool," in *Third International Conference on Software Testing, Verification and Validation*, 2010, pp. 175–184.

[10] N. Tracey, J. Clark, K. Mander, and J. Mcdermid, "An Automated Framework for Structural Test-Data Generation," in *Automated Software Engineering. Proceedings. 13th IEEE International Conference on*, 1998, pp. 285–288.

[11] B. Beizer, *Software testing techniques*. 2nd ed. New York: Van Nostrand Reinhold, 1990.

[12] V. Halava, "Decidable and undecidable problems in matrix theory," 1st ed. Turku, Finland: University of Turku, Department of Mathematics, 1997.

[13] T. Dyba, B. Kitchenham, and M. Jorgensen, "Evidence-based software engineering for practitioners," *Software, IEEE*, vol. 22, no. 1, pp. 58–65, Jan. 2005.

[14] J. Radatz, A. Geraci, and F. Katki, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std*, vol. 610121990, p. 121990, 1990.

[15] H. Tahbildar and B. Kalita, "Automated software test data generation: Direction of research," *International Journal of Computer Science and Engineering Survey*, vol. 2, no. 1, pp. 99–120, 2011.

[16] B. Korel, "Automated test data generation for programs with procedures," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 209–215, May 1996.

[17] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim, "A Comparative Study on Automated Software Test Oracle Methods," in *Fourth International Conference on Software Engineering Advances*, 2009, pp. 140–145.

[18] M. Harman, S. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*, pp. 1–78, 2009.

[19] J. Edvardsson, "A survey on automatic test data generation," in *Proceedings of the 2nd Conference on Computer Science and Engineering*, 1999, pp. 21–28.

[20] P. Mcminn and R. Court, "Search-based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[21] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[22] H. Tahbildar and B. Kalita, "Heuristic Approach of Automated Test Data Generation for Program having Array of Different Dimensions and Loops with Variable Number of Iteration," *arXiv preprint arXiv:1011.0594*, Nov. 2010.

[23] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 1, pp. 63–86, Jan. 1996.

[24] R. Pargas, M. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[25] K.-H. K. Chang, I. Cross, JamesH, W. H. Carlisle, D. Brown, and J. C. II, "A framework for intelligent test data generators," *Journal of Intelligent and Robotic Systems*, vol. 5, no. 2, pp. 147–165, 1992.

[26] C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton, "Genetic algorithms for dynamic test data generation," in *Automated Software Engineering. Proceedings., 12th IEEE International Conference*, 1997, pp. 307–308.

[27] S. Mahmood, "A Systematic Review of Automated Test Data Generation Techniques," M.S. thesis, School of Engineering, Blekinge Institute of Technology, Ronneby, Sweden, 2007.

[28] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *Dependable Computing-EDCC 5*, 2005, pp. 281–292.

[29] 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.

[30] S. Zhang, "Palus : A Hybrid Automated Test Generation Tool for Java," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1182–1184.

[31] R. Romli, S. Sulaiman, and K. Zamli, "Automatic programming assessment and test data generation a review on its approaches," in *Information Technology (ITSim), International Symposium in*, 2010, vol. 3, pp. 1186–1192.

[32] P. McMinn, "Search-Based Software Testing: Past, Present and Future," *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 153–163, Mar. 2011.

[33] E. Alba and F. Chicano, "Observations in using parallel and sequential evolutionary algorithms for automatic software testing," *Computers & Operations Research*, vol. 35, no. 10, pp. 3161–3183, Oct. 2008.

[34] R. Blanco, J. Tuya, E. Diaz, and B. A. Diaz, "A scatter search approach for automated branch coverage in software testing," *Engineering intelligent systems for electrical engineering and communications*, vol. 15, no. 3, pp. 135–141, 2007.

[35] E. Díaz, J. Tuya, R. Blanco, J. J. Dolado, and J. Javier Dolado, "A tabu search algorithm for structural software testing," *Computers & Operations Research*, vol. 35, no. 10, pp. 3052–3072, Oct. 2008.

[36] A. Windisch, S. Wappler, and J. Wegener, “Applying particle swarm optimization to software testing,” in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, New York, New York, USA, 2007, p. 1121.

[37] M. Harman and P. McMinn, “A theoretical & empirical znalysis of evolutionary testing and hill climbing for structural test data generation,” in *Proceedings of the international symposium on Software testing and analysis - ISSTA ’07*, New York, New York, USA, 2007, pp. 73–83.

[38] A. I. Baars, K. Lakhotia, T. E. Vos, and J. Wegener, “Automatic generation of test data for path testing by adaptive genetic simulated annealing algorithm,” in *Computer Science and Information Systems (FedCSIS), Federated Conference on*, 2011, pp. 917–923.

[39] A. I. Baars, K. Lakhotia, T. E. Vos, and J. Wegener, “Search-based testing, the underlying engine of Future Internet testing,” in *Computer Science and Information Systems (FedCSIS), Federated Conference on*, 2011, pp. 917–923.

[40] M. Harman, P. Mcminn, and R. Court, “A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search,” *Software Engineering, IEEE Transactions on*, vol. 36, no. 2, pp. 226–247, Mar. 2010.

[41] K. Lakhotia, “Search–Based Testing,” M.S. thesis, Dept. Comp. Science, King's College London, London, 2009..

[42] X. Xiao, T. Xie, N. Tillmann, J. de Halleux, and J. De Halleux, “Precise identification of problems for structural test generation,” in *Proceeding of the 33rd international conference on Software engineering - ICSE ’11*, 2011, pp. 611–620.

[43] B. Korel, Z. Terms-automated, and W. State, “Automated software test data generation,” *Software Engineering, IEEE Transactions on*, no. X, pp. 870–879, Aug. 1990.

[44] 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.

[45] G. Fraser and A. Zeller, “Generating parameterized unit tests,” in *Proceedings of the International Symposium on Software Testing and Analysis - ISSTA ’11*, New York, New York, USA, 2011, pp. 364–374.

[46] M. Papadakis and N. Malevris, “A symbolic execution tool based on the elimination of infeasible paths,” in *Software Engineering Advances (ICSEA), Fifth International Conference on*, 2010, pp. 435–440.

[47] 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*, 2007, pp. 134–138.

[48] W. Visser, C. S. Păsăreanu, and S. Khurshid, “Test input generation with java PathFinder,” in *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis - ISSTA ’04*, New York, New York, USA, 2004, pp. 97–107.

[49] E. Albert, M. Gómez-Zamalloa, and G. Puebla, “PET: a partial evaluation-based test case generation tool for Java bytecode,” *Proceedings of the ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 25–28, 2010.

[50] E. Albert, M. Gómez-Zamalloa, and G. Puebla, “Test data generation of bytecode by CLP partial evaluation,” in *Logic-Based Program Synthesis and Transformation*, 2009, pp. 4–23.

[51] E. Albert, I. Cabanas, A. Flores-Montoya, M. Gómez-Zamalloa, and S. Gutiérrez, “jPET: an Automatic Test-Case Generator for Java,” in *Reverse Engineering (WCRE), 18th Working Conference on*, 2011, pp. 441–442.

[52] A. Gotlieb, “Euclide: A constraint-based testing framework for critical c programs,” *Software Testing Verification and Validation. ICST’09. International Conference on*, pp. 151–160, 2009.

[53] P. Godefroid, N. Klarlund, and K. Sen, “DART: directed automated random testing,” in *ACM Sigplan Notices*, 2005, vol. 40, pp. 213–223.

[54] S. Bardin and P. Herrmann, “OSMOSE: automatic structural testing of executables,” *Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 29–54, 2011.

[55] K. Sen and G. Agha, “CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools,” in *Computer Aided Verification*, 2006, pp. 419–423.

[56] R. Majumdar and K. Sen, “Hybrid concolic testing,” in *Software Engineering. ICSE. 29th International Conference on*, 2007, pp. 416–426.

[57] C. Cadar and D. Engler, “Execution generated test cases: How to make systems code crash itself,” in *Model Checking Software*, 2005, pp. 2–23.

[58] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation,” in *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering*, 2008, pp. 443–446.

[59] K. Kähkönen, T. Launiainen, O. Saarikivi, J. Kauttio, K. Heljanko, and I. Niemelä, "LCT: An open source concolic testing tool for Java programs," in *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2011, pp. 75–80.

[60] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, Dec. 2008.

[61] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.," in *OSDI*, 2008, vol. 8, pp. 209–224.

[62] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *Tests and Proofs*, 2008, pp. 134–153.

[63] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2005, pp. 365–381.

[64] X. Deng and J. Hatcliff, "Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION. TAICPART-MUTATION*, 2007, pp. 3–12.

[65] X. Deng and J. Hatcliff, "Kiasan: a verification and test-case generation framework for Java based on symbolic execution," in *Leveraging Applications of Formal Methods, Verification and Validation. ISoLA. Second International Symposium on*, 2006, pp. 137–137.

[66] X. Deng and J. Lee, "Robby. Efficient symbolic execution algorithms for programs manipulating dynamic heap objects,", Kansas State University, Kansas, Rep. SAnToS-TR2009-09-25, 2009.

[67] E. Díaz, J. Tuya, and R. Blanco, "A modular tool for automated coverage in software testing," in *Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on*, 2003, pp. 241–246.

[68] P. McMinn, "IGUANA: Input generation using automated novel algorithms. A plug and play research tool," *Department of Computer Science, University of Sheffield, Tech. Rep. CS-07-14*, 2007.

[69] P. Tonella, "Evolutionary testing of classes," *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 119–128, 2004.

[70] G. Fraser and A. Arcuri, “EvoSuite: Automatic Test Suite Generation for Object-Oriented Software,” in *SIGSOFT FSE*, 2011, pp. 416–419.

[71] 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.

[72] 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 International Symposium on Search Based Software Engineering*, pp. 101–110, Sep. 2010.

[73] L. Baresi and M. Miraz, “TestFul: Automatic Unit-Test Generation for Java Classes,” in *Software Engineering, ACM/IEEE 32nd International Conference on*, 2010, vol. 2, pp. 281–284.

[74] L. Baresi, P. L. Lanzi, and M. Miraz, “TestFul: An Evolutionary Test Approach for Java,” *Third International Conference on Software Testing, Verification and Validation*, pp. 185–194, 2010.

[75] M. Dimitar, I. Dimitrov, and I. Spasov, “Evotest-Framework for customizable implementation of Evolutionary Testing,” *International Workshop on Software and Services*, Oct. 2008.

[76] J. Wegener, A. Baresel, and H. Sthamer, “Evolutionary test environment for automatic structural testing,” *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, Dec. 2001.

[77] K. Ghani and J. Clark, “Automatic Test Data Generation for Multiple Condition and MCDC Coverage,” in *Software Engineering Advances. ICSEA’09. Fourth International Conference on*, 2009, pp. 152–157.

[78] K. Inkumsah and T. Xie, “Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 425–428.

[79] 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.

[80] H. Gross, P. M. Kruse, J. Wegener, and T. Vos, “Evolutionary White-Box Software Test with the EvoTest Framework: A Progress Report,” in *International Conference on Software Testing, Verification, and Validation Workshops*, 2009, pp. 111–120.

[81] 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.

[82] 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.

[83] C. Pacheco and M. Ernst, “Eclat: Automatic generation and classification of test inputs,” *ECOOP Object-Oriented Programming*, 2005.

[84] 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.

[85] 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.

[86] 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.

[87] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, “Does Automated White-Box Test Generation Really Help Software Testers?,” in *ISSTA*, 2013.

[88] F. Gross, G. Fraser, and A. Zeller, “Exploring Realistic Program Behavior,” Saarland University, Saarbrücken, Germany, 2012.

[89] D. I. Sjoberg, T. Dyba, and M. Jorgensen, “The future of empirical methods in software engineering research,” in *Future of Software Engineering. FOSE’07*, 2007, pp. 358–378.

[90] C. Robson, *Real World Research*. Chichister, 2nd. ed. West Sussex: John Willey & Sons, 2011.

[91] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, Apr. 2009.

[92] R. Sampieri, C. Collado, P. Lucio, and M. Pérez, *Metodología de la investigación*. 3rd. ed. México: McGraw Hill. 2003.

[93] B. a. Kitchenham, T. Dyba, and M. Jorgensen, “Evidence-based software engineering,” in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 273–281.

[94] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, “Systematic mapping studies in software engineering,” in *12th International Conference on Evaluation and Assessment in Software Engineering*, 2008, vol. 17, p. 1.

[95] D. Budgen, “ASE-EVAL : Software Evaluation Glossary,” *Evidence-Based Software Engineering*, (2013, March, 19). [Online]. Available: https://www.dur.ac.uk/ebse/glossary.php.

[96] B. A. Kitchenham, D. Budgen, and P. Brereton, “The value of mapping studies-a participant-observer case study,” in *Proceedings of Evaluation and Assessment of Software Engineering, EASE*, 2010.

[97] B. A. Kitchenham, “Guidelines for performing systematic literature reviews in software engineering,” Dept. Comp. Sci., Univ. of Durham, Durham, UK, 2007.

[98] R. K. R. Yin, “Case study research: Design and methods,” 3rd. ed. Thousand Oaks, California: Sage Publications, 2003.

[99] C. Wohlin, M. Höst, and K. Henningsson, “Empirical research methods in software engineering,” in *Empirical Methods and Studies in Software Engineering*, 2003, pp. 7–23.

[100] B. Kitchenham, L. Pickard, and S. Pfleeger, “Case studies for method and tool evaluation,” *Software, IEEE*, vol. 12, no. 4, pp. 52–62, Jul. 1995.

[101] O. Dieste, A. Grimán, and N. Juristo, “Developing search strategies for detecting relevant experiments,” *Empirical Software Engineering*, vol. 14, no. 5, pp. 513–539, Oct. 2009.

[102] F. Elberzhager, J. Münch, and V. T. N. Nha, “A systematic mapping study on the combination of static and dynamic quality assurance techniques,” *Information and Software Technology*, vol. 54, no. 1, pp. 1–15, Jan. 2012.

[103] J. Duran and S. Ntafos, “An evaluation of random testing,” *Software Engineering, IEEE Transactions on*, no. 4, pp. 438–444, May 1984.

[104] S. Bardin, B. Botella, F. Dadeau, F. Charreteur, A. Gotlieb, B. Marre, C. Michel, M. Rueher, and N. Williams, “Constraint-based software testing,” *Journée du GDR-GPL*, vol. 9, 2009.

[105] N. Pan, F. Zeng, and Y. Huang, “Test case reduction based on program invariant and genetic algorithm,” in *Wireless Communications Networking and Mobile Computing (WiCOM), 6th International Conference on*, 2010, pp. 1–5.

[106] F. Zeng, Q. Cao, L. Mao, and Z. Chen, “Test case generation based on invariant extraction,” *Wireless Communications, Networking and Mobile Computing. WiCom’09. 5th International Conference on*, pp. 1–4, 2009.

[107] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, “Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem,” in *Software Testing, Verification, and Validation Workshops (ICSTW), Third International Conference on*, 2010, pp. 182–191.

[108] A. Gotlieb and M. Petit, “A uniform random test data generator for path testing,” *Journal of Systems and Software*, vol. 83, no. 12, pp. 2618–2626, Dec. 2010.

[109] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun, and J. Wegener, “Input Domain Reduction through Irrelevant Variable Removal and Its Effect on Local, Global, and Hybrid Search-Based Structural Test Data Generation,” *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 453–477, Mar. 2012.

[110] S. Bardin and P. Herrmann, “Pruning the search space in path-based test generation,” in *Software Testing Verification and Validation. ICST’09. International Conference on*, 2009, pp. 240–249.

[111] A. Arcuri and X. Yao, “Search based software testing of object-oriented containers,” *Information Sciences*, vol. 178, no. 15, pp. 3075–3095, Aug. 2008.

[112] Y. Bo, Q. Ye-mei, L. Feng-ye, and M. Can, “Tabu search and genetic algorithm for generating test data of class testing,” in *Information Engineering and Computer Science. ICIECS . International Conference on*, 2009, pp. 1–6.

[113] S. Sai-ngern, C. Lursinsap, and P. Sophatsathit, “An address mapping approach for test data generation of dynamic linked structures,” *Information and Software Technology*, vol. 47, no. 3, pp. 199–214, Mar. 2005.

[114] R. Zhao and Q. Li, “Automatic test generation for dynamic data structures,” in *Software Engineering Research, Management & Applications. SERA. 5th ACIS International Conference on*, 2007, pp. 545–549.

[115] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, “Bringing white-box testing to service oriented architectures through a service oriented approach,” *Journal of Systems and Software*, vol. 84, no. 4, pp. 655–668, Apr. 2011.

[116] A. Arcuri, “Longer is better: On the role of test sequence length in software testing,” in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, 2010, pp. 469–478.

[117] D. Binkley, M. Harman, and K. Lakhotia, “FlagRemover: A testability transformation for transforming loop-assigned flags,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 3, p. 12, Aug. 2011.

[118] P. McMinn, D. Binkley, and M. Harman, “Empirical evaluation of a nesting testability transformation for evolutionary testing,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 3, p. 11, 2009.

[119] M. R. Keyvanpour, H. Homayouni, and H. Shirazee, “Automatic software test case generation,” *Journal of Software Engineering*, vol. 5, no. 3, pp. 91–101, May 2011.

[120] K. Li, Z. Zhang, and W. Liu, “Automatic test data generation based on ant colony optimization,” in *Natural Computation. ICNC’09. Fifth International Conference on*, 2009, vol. 6, pp. 216–220.

[121] J. Ye, Z. Zhan, Z. Zhang, W. Dong, and Z. Qi, “Design of Some Artificial Immune Operators in Software Test Cases Generation,” in *Young Computer Scientists. ICYCS. The 9th International Conference for*, 2008, pp. 2302–2307.

[122] R. Becerra, R. Sagarna, and X. Yao, “An evaluation of Differential Evolution in software test data generation,” in *Evolutionary Computation. CEC’09. IEEE Congress on*, 2009, pp. 2850–2857.

[123] A. S. Andreou, K. A. Economides, and A. A. Sofokleous, “An automatic software test-data generation scheme based on data flow criteria and genetic algorithms,” in *Computer and Information Technology. CIT. 7th IEEE International Conference on*, 2007, pp. 867–872.

[124] Y. Cao, C. Hu, and L. Li, “An approach to generate software test data for a specific path automatically with genetic algorithm,” in *Reliability, Maintainability and Safety. ICRMS. 8th International Conference on*, 2009, pp. 888–892.

[125] Y. Chen, Y. Zhong, T. Shi, and J. Liu, “Comparison of Two Fitness Functions for GA-Based Path-Oriented Test Data Generation,” in *Fifth International Conference on Natural Computation*, 2009, pp. 177–181.

[126] Y. Chen and Y. Zhong, “Experimental Study on GA-Based Path-Oriented Test Data Generation Using Branch Distance Function,” *Third International*

Symposium on Intelligent Information Technology Application, pp. 216–219, 2009.

[127] S. Y. Lee, H. J. Choi, Y. J. Jeong, T. H. Kim, H. S. Chae, and C. K. Chang, "An improved technique of fitness evaluation for evolutionary testing," in *Computer Software and Applications Conference Workshops (COMPSACW), IEEE 35th Annual*, 2011, pp. 190–193.

[128] K. Liaskos and M. Roper, "Hybridizing evolutionary testing with artificial immune systems and local search," in *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, 2008, pp. 211–220.

[129] A. Ghiduk and M. Girgis, "Using genetic algorithms and dominance concepts for generating reduced test data," *Informatica (Slovenia)*, vol. 34, no. 3, pp. 377–385, Mar. 2010.

[130] D. Gong, W. Zhang, and X. Yao, "Evolutionary generation of test data for many paths coverage based on grouping," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2222–2233, Dec. 2011.

[131] S. Zhang, Y. Zhang, H. Zhou, and Q. He, "Automatic path test data generation based on GA-PSO," in *Intelligent Computing and Intelligent Systems (ICIS), IEEE International Conference on*, 2010, vol. 1, pp. 142–146.

[132] K. Li, Z. Zhang, and J. Kou, "Breeding Software Test Data with Genetic-Particle Swarm Mixed Algorithm," *Journal of computers*, Feb. 2010.

[133] B. Li, Z. Li, J. Zhang, and J. Sun, "An automated test case generation approach by genetic simulated annealing algorithm," in *Natural Computation. ICNC. Third International Conference on*, 2007, vol. 4, pp. 106–111.

[134] H. Cui, L. Chen, B. Zhu, and H. Kuang, "An efficient automated test data generation method," in *Measuring Technology and Mechatronics Automation (ICMTMA), International Conference on*, 2010, vol. 1, pp. 453–456.

[135] X. Zhu and X. Yang, "Software test data generation automatically based on improved adaptive particle swarm optimizer," in *Computational and Information Sciences (ICCIS), International Conference on*, 2010, pp. 1300–1303.

[136] G. H. L. Pinto and S. R. Vergilio, "A Multi-Objective Genetic Algorithm to Test Data Generation," *22nd IEEE International Conference on Tools with Artificial Intelligence*, pp. 129–134, Oct. 2010.

[137] Y. Chen and Y. Zhong, “Automatic Path-Oriented Test Data Generation Using a Multi-population Genetic Algorithm,” in *Fourth International Conference on Natural Computation*, 2008, pp. 566–570.

[138] M. Alshraideh, B. a. Mahafzah, and S. Al-sharaeh, “A multiple-population genetic algorithm for branch coverage test data generation,” *Software Quality Journal*, vol. 19, no. 3, pp. 489–513, Oct. 2010.

[139] J. Xuan, H. Jiang, Z. Ren, Y. Hu, and Z. Luo, “A Random Walk Based Algorithm for Structural Test Case Generation,” in *Software Engineering and Data Mining (SEDM), 2nd International Conference on*, 1990, no. 60805024, pp. 583–588.

[140] S. Xiajiong, W. Qian, W. Peipei, and Z. Bo, “Automatic generation of test case based on GATS algorithm,” in *Granular Computing, GRC . IEEE International Conference on*, 2009, pp. 496–500.

[141] K. Ayari, S. Bouktif, and G. Antoniol, “Automatic mutation test input data generation via ant colony,” in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007, pp. 1074–1081.

[142] F. Pinte, N. Oster, and F. Saglietti, “Techniques and tools for the automatic generation of optimal test data at code, model and interface level,” in *Companion of the 30th international conference on Software engineering*, 2008, pp. 927–928.

[143] A. Sofokleous and A. Andreou, “Automatic, evolutionary test data generation for dynamic software testing,” *Journal of Systems and Software*, Nov. 2008.

[144] M. Harman, F. Islam, T. Xie, and S. Wappler, “Automated test data generation for aspect-oriented programs,” in *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, 2009, pp. 185–196.

[145] Y. Cao, C. Hu, and L. Li, “Search-based multi-paths test data generation for structure-oriented testing,” in *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, 2009, pp. 25–32.

[146] Z. Awedikian, K. Ayari, and G. Antoniol, “MC/DC automatic test input data generation,” in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, 2009, pp. 1657–1664.

[147] M. Alshraideh, L. Bottaci, and B. Mahafzah, “Using program data-state scarcity to guide automatic test data generation,” *Software Quality Journal*, Mar. 2010.

[148] A. Gotlieb and M. Petit, “Path-oriented random testing,” in *Proceedings of the 1st international workshop on Random testing - RT*, New York, New York, USA, 2006, p. 28.

[149] M. Harman, Y. Jia, and W. Langdon, “Strong higher order mutation-based test data generation,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 212–222.

[150] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, “Enhancing structural software coverage by incrementally computing branch executability,” *Software quality journal*, Dec. 2011.

[151] J. Malburg and G. Fraser, “Combining search-based and constraint-based testing,” in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 436–439.

[152] M. Papadakis and N. Malevris, “Automatic mutation based test data generation,” in *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, 2011, pp. 247–248.

[153] L. Ming-Hao, G. You-Feng, S. Jin-Hui, L. Jiang-Hong, Z. Lu, and S. Jia-Su, “An approach to test data generation for killing multiple mutants,” in *Software Maintenance. ICSM. 22nd IEEE International Conference on*, 2006, pp. 113–122.

[154] P. Moscato, “On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms,” *Caltech concurrent computation program, C3P Report*, 1989.

[155] S. Ali, S. Member, L. C. Briand, H. Hemmati, and R. K. Panesar-walawege, “A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation,” *Software Engineering, IEEE Transactions on*, vol. 36, no. 6, pp. 742–762, 2010.

[156] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, and B. Regnell, *Experimentation in software engineering*, Heidelberg, Germany: Springer, 2012.

[157] G. Fraser and A. Arcuri, “Evolutionary generation of whole test suites,” in *Quality Software (QSIC), 2011 11th International Conference on*, 2011, pp. 31–40.

[158] G. Fraser and A. Arcuri, “Whole Test Suite Generation,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276 – 291, Feb. 2012.

[159] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva, “Contract driven development= test driven development-writing test cases,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 425–434.

[160] M. R. Hoffmann, B. Janiczak, and E. Mandrikov, “Java Code Coverage for Eclipse", (2013, March, 01) [Online]. Available: http://www.eclemma.org/index.html.

[161] R. P. L. Buse and W. R. Weimer, “Learning a Metric for Code Readability,” *IEEE Transactions on Software Engineering*, vol. 36, pp. 546–558, 2010.

[162] M. Miraz, P. Lanzi, and L. Baresi, “TestFul: using a hybrid evolutionary algorithm for testing stateful systems,” in *Software testing, verification and validation (ICST), third international conference on*, 2010, pp. 185–194.

[163] B. Meyer, I. Ciupa, A. Leitner, and L. L. Liu, “Automatic testing of object-oriented software,” *SOFSEM Theory and Practice of Computer Science*, vol. 2007, pp. 114–129, Jan. 2007.

[164] G. Quintana and M. Solari, “A systematic mapping study on experiments with automatic structural test case generation,” in *Informatica (CLEI), XXXVIII Conferencia Latinoamericana En*, Medellin, CO-ANT, 2012, pp. 1 – 10.