

Propuesta de optimización en la prueba de mutaciones en Java

Inmaculada Medina Buló
Dpto. de Lenguajes y Sistemas
Informáticos
Escuela Superior de Ingeniería
Universidad de Cádiz
11002 Cádiz
inmaculada.medina@uca.es

Lorena Gutiérrez Madroñal
Dpto. de Lenguaje y Sistemas
Informáticos
Escuela Superior de Ingeniería
Universidad de Cádiz
11002 Cádiz
lorena.gutierrez@uca.es

Juan José Domínguez
Jiménez
Dpto. de Lenguajes y Sistemas
Informáticos
Escuela Superior de Ingeniería
Universidad de Cádiz
11002 Cádiz
juanjose.dominguez@uca.es

Resumen

Actualmente existen diversas aplicaciones de prueba de mutaciones para programas escritos en Java, la mayoría de ellas ejecutan todos los mutantes generados o emplean técnicas de optimización básicas. En este trabajo se propone emplear la *mutación evolutiva*, una técnica de prueba novedosa que utiliza algoritmos evolutivos para reducir el número de mutantes generados sin pérdida significativa de efectividad, como técnica de optimización de la prueba de mutaciones en programas escritos en Java. En esta propuesta se pretende realizar una nueva herramienta, que empleando esta técnica de optimización basada en algoritmos evolutivos, disminuya su alto coste computacional, que es el principal inconveniente de estas aplicaciones.

1. Introducción

La prueba de mutaciones es una técnica basada en fallos, que mide la efectividad de los conjuntos de los casos de pruebas para la localización de errores, para detectar fallos introducidos de forma intencionada [1, 2]. La efectividad de detección de errores de los conjuntos de los casos de pruebas, está definida como el porcentaje de errores que pueden ser detectados por esos conjuntos de casos de pruebas [3]. La prueba de mutaciones ha sido validada como una poderosa herramienta para la evaluación de la calidad de los casos de prueba [4].

Esta técnica ha sido empleada con éxito en diversos lenguajes, encontrándose herramientas que automatizan la generación de mutantes para diversos lenguajes, tales como Mothra [5] para Fortran, MuJava [6, 7, 8] para Java, GAmera [4] para WS-BPEL, SQLMutation [9] para SQL, etc.

Sin embargo, uno de los problemas a los que se enfrenta la prueba de mutaciones es el elevado número de mutantes que se dispone para un programa original, con el consiguiente coste computacional que implica la ejecución de todos ellos. En este sentido, la mutación evolutiva [10] se presenta como una técnica que permite reducir el número de mutantes generados sin pérdida de efectividad. Esta técnica ha sido implementada en la herramienta GAmera para el lenguaje WS-BPEL.

El objetivo principal de este trabajo es sentar las bases para elaborar una arquitectura de código abierto para la realización de prueba de mutaciones en Java, utilizando *mutación evolutiva* [10] para reducir el número de mutantes generados.

El resto del artículo se organiza de la siguiente forma. En la sección 2 se especifican los conceptos fundamentales asociados con la prueba de mutaciones y el estado actual del tema. En la sección 3 se exponen las hipótesis planteadas en este trabajo. Por último, la sección 4 presenta las conclusiones y el trabajo futuro.

2. Antecedentes y estado actual

En esta sección se presenta, en primer lugar, los conceptos fundamentales relacionados con la prueba de mutaciones. A continuación, se profundiza en una de las herramientas que se ha empleado en el estudio realizado y se contrasta la optimización empleada en dicha herramienta con las utilizadas en el resto de aplicaciones.

Finalmente, se valora dicha optimización para emplearla en nuestro trabajo.

2.1. Prueba de mutaciones

La prueba de mutaciones es una técnica de prueba basada en fallos [1,11], que consiste en introducir fallos simples en el programa original mediante la aplicación de operadores de mutación. Los programas resultantes reciben el nombre de *mutantes*. Cada operador de mutación se corresponde con una categoría de error típico que puede cometer el programador. Si un caso de prueba es capaz de distinguir al programa original del mutante, es decir, la salida del mutante y la del programa original son diferentes, se dice que mata al mutante. Si por el contrario, ningún caso de prueba es capaz de diferenciar al mutante del programa original, es decir, la salida del mutante y del programa original es la misma, se habla de un mutante vivo para el conjunto de casos de prueba empleado.

Aplicar la prueba de mutaciones tiene diversas dificultades, la principal de ellas es la existencia de mutantes equivalentes. Un mutante es equivalente cuando provoca la misma salida que el programa original, por lo que no va a existir ningún caso de prueba que permita matarlo; es decir, ambos tienen el mismo comportamiento. Desafortunadamente, determinar cuál de los mutantes es equivalente al programa original es una actividad extremadamente tediosa y propensa a errores [3].

La calidad de los conjuntos de casos de prueba se mide mediante la puntuación de mutación (*mutation score*), que está definida como el cociente entre el número de mutantes muertos y el número total de mutantes no equivalentes [12].

El número total de mutantes no equivalentes viene de la diferencia entre el número total de mutantes y el número de mutantes equivalentes. La puntuación de mutación es una medida cuantitativa de la calidad del conjunto de casos de pruebas. El conjunto de casos de prueba de mejor calidad es el que obtiene la mayor puntuación de mutación posible, es decir, es el que consigue matar al mayor número de mutantes. Por otro lado, dados dos conjuntos de casos de prueba con la misma puntuación de mutación será preferible el que tenga un menor número de casos de prueba [13].

Gracias a las pruebas de mutaciones se pueden obtener nuevos casos de pruebas que matan a los mutantes que permanecen vivos, de tal modo que la calidad del conjunto inicial de casos de prueba mejora. En 1998, fue Offutt [14] el que propuso por primera vez, la utilización de prueba de

mutaciones para la generación de casos de prueba, detallando las condiciones a cumplir por un caso de prueba para matar a un mutante:

1. La sentencia mutada debe ser alcanzada.
2. El estado de ejecución del mutante debe ser diferente al del programa original una vez ejecutada la sentencia mutada.
3. El estado de ejecución tras la sentencia mutada debe propagarse a la salida final del mutante.

En [15] van un paso más allá, ya que no sólo quieren generar casos de prueba para matar mutantes, sino que también intentan minimizar el número de casos de prueba generados para matar al mayor número posible de mutantes.

2.2. Herramientas de prueba de mutaciones en Java

Hoy en día, para el lenguaje de programación Java, hay disponibles diversas herramientas de pruebas de mutaciones. Entre estas aplicaciones se pueden destacar: MuJava [7], Jester [16], Jumble [17], RI [18], JavaMut [19] y Judy [3].

La diferencia entre ellas radica en los operadores de mutación que soportan así como los métodos empleados para la generación de mutantes y las técnicas de aceleración u optimización utilizadas.

La tabla 1 recoge las principales diferencias entre estas herramientas. En ella no aparece JavaMut al no poder disponer del código de la misma. De todas ellas, Judy es la más reciente, y según [3] la más completa de las que existen hasta el momento, según las características anteriormente comentadas.

Características	MuJava	Jester	Jumble	RI	Judy
Operadores de mutación de selección y tradicionales (ABS, AOR, LCR, ROR, UOI)	Sí	No	No ¹	No ²	Sí
Operadores de mutación orientados a objetos	Sí	No	No	No	Sí ³
Nivel de generación de mutantes	Bytecode ⁴	Código fuente	Bytecode	--	Código fuente
Produce mutantes no ejecutables	Sí ⁵	Sí	Sí ⁴	--	Sí
Formato de los mutantes	Ficheros de clases separados	Ficheros fuentes separados	En memoria	--	Agrupados en ficheros fuentes
Soporta JUnit	No	Sí	Sí	Sí	Sí
Soporte para herramienta de construcción o de línea de comando	No	Sí	Sí	--	Sí
Disponible	Sí	Sí	Sí	No	Sí

Tabla 1: Comparativa de las herramientas de prueba de mutaciones en Java [3]

¹Los operadores de selección no se proporcionan. Éstos deberían ser implementados, y no es algo fácil, éstos se obtienen modificando el código de la herramienta.

²El prototipo RI usa sólo unos operadores de mutación simples, por ejemplo, cambiar los tipos primitivos y los objetos de la clase Cadena.

³Solo en los niveles entre métodos.

⁴Según Offutt [6], se han implementado tres versiones de MuJava usando diferentes técnicas de implementación: modificación del código fuente, modificación bytecode y esquema de los mutantes. Hubo algunos problemas con las dos últimas técnicas, pero finalmente, se combinaron [7]. Esto significa que algunas de las mutaciones se pueden seguir haciendo con el análisis del código fuente usando el esquema de los mutantes.

⁵Aunque, los mecanismos de generación basados en la transformación de bytecode reducen la posibilidad de la creación de mutantes no ejecutables en comparación con aquéllos basados en la transformación de código fuente, ellos no pueden garantizar que todos los mutantes creados usando esta técnica serán ejecutados (por ejemplo, el cambio del modificador de visibilidad de `public` a `private` (operador de mutación AMC [20]) mediante un método que sea invocado externamente en otros paquetes de proyectos, crearán un mutante no ejecutable, pero esto puede realizarse usando una transformación bytecode).

Todos estos sistemas se caracterizan por estar todos los mutantes posibles de acuerdo a los operadores de mutación empleados en una posible selección previa.

3. Optimizando la prueba de mutaciones en Java

Uno de los principales inconvenientes que presenta la prueba de mutaciones, es el alto coste computacional que supone la ejecución de la gran cantidad de mutantes que se generan para el programa de prueba. En [4] se desarrolla la herramienta GAmEra para la aplicación de prueba de mutaciones a composiciones WS-BPEL, que introduce un mecanismo de optimización que permite seleccionar sólo un subconjunto de los mutantes totales que puedan generarse. Para ello GAmEra incorpora un algoritmo genético que

selecciona sólo los mutantes de mayor calidad, reduciéndose el coste computacional que supondría la ejecución de todos los mutantes.

Actualmente no existe ninguna herramienta de prueba de mutaciones con lenguaje Java que utilice esta técnica de optimización. De ahí nace el estudio que se presenta en este trabajo, el cual quiere reutilizar esta técnica para la generación de mutaciones con un coste computacional menor gracias a la previa selección de aquellos mutantes de mayor calidad.

Esta arquitectura va a constar de tres partes: análisis del programa original, generación de mutantes y ejecución de los mutantes. La figura 1 muestra un esquema de la arquitectura del sistema de prueba de mutaciones en Java que se propone. En ella se recibe como entrada el lenguaje original y un conjunto de casos de prueba.

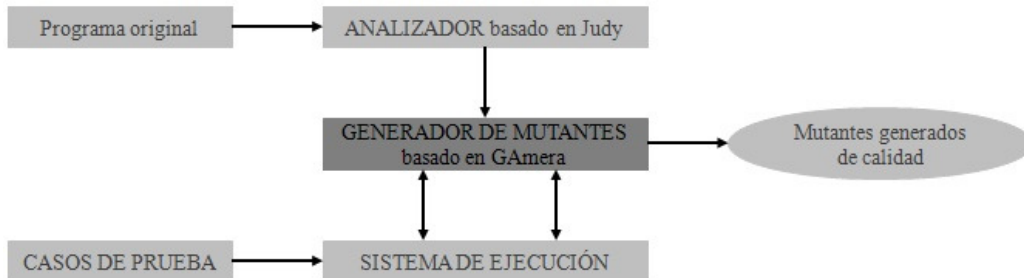


Figura 1: Esquema de la arquitectura del sistema de prueba de mutaciones en Java

El primer componente, el *analizador*, tomaría el programa original y determinaría los operadores de mutación que se pueden aplicar, así como el número de localidades donde puede usarse. Se propone la reutilización de alguna de las arquitecturas de prueba de mutaciones ya existentes para el lenguaje Java, y adaptarla para su integración en GAmEra.

Esta información es tomada por el segundo componente, el *generador de mutantes*. Para ello se quiere utilizar el sistema GAmEra [4] del grupo de investigación SPI&FM (Software Process Improvement and Formal Methods). En dicho grupo se emplea el lenguaje WS-BPEL y en este estudio, se quiere reutilizar este motor de *generador de mutantes* para el mismo propósito, pero con un lenguaje de programación diferente, Java. En este caso, tendríamos que establecer los mecanismos adecuados para la conexión de ambos componentes. En el siguiente punto se describirá el generador de mutantes del sistema GAmEra.

Finalmente, se añadiría un componente de ejecución de mutantes, el sistema de *ejecución*, para que el sistema generador de mutantes pudiese determinar la calidad de los mutantes que va generando. Esto permitiría al sistema generador producir un subconjunto de mutantes adecuados.

El empleo de la mutación evolutiva y su herramienta GAmEra a Java se sustenta bajo las siguientes hipótesis:

- Posibilidad de obtener un menor coste computacional: podemos afirmar que el número de mutantes a ejecutar es menor, por lo que, en este caso, se obtendrá un coste menor que si se ejecutasen todos los mutantes, tal y como hacen la mayoría de las herramientas citadas anteriormente.

- Ejecución de mutantes de mayor calidad: Se utilizan algoritmos genéticos para la selección de mutantes de mayor calidad. De este modo a la hora de realizar la ejecución, se ejecutarían los mutantes que hemos obtenido más representativos.

Una vez adaptada la técnica de mutación evolutiva al lenguaje Java podremos realizar comprobaciones con otras herramientas ya existentes. Esto permitiría también contrastar de manera más eficiente esta técnica de optimización. Por tanto, si la adaptación de esta técnica al lenguaje de programación Java es satisfactoria sería una técnica a considerar por el resto de herramientas que emplean otros lenguajes de programación para que el coste computacional no sea el principal inconveniente.

3.1. Generador de mutantes de GAmEra

Los mutantes se generan a partir de la información que se recibe del analizador. La herramienta nos da la posibilidad de generar todos los mutantes posibles, o bien, un subconjunto de éstos que va a ser seleccionado por el algoritmo genético. En este último caso, se llamará al componente denominado Generador de mutantes, que está compuesto por dos elementos. El primero, denominado *Búsqueda Genética de Mutantes*, es un algoritmo genético en el que cada individuo representa a un mutante, capaz de generar y seleccionar de forma automática un conjunto de mutantes. Esta selección se realiza aplicando una función de aptitud que mide su calidad en función de si hay o no casos de prueba que lo matan. El segundo elemento es el

Conversor, que transforma un individuo del algoritmo genético en un mutante WS-BPEL. Para realizar esta conversión, se utilizan hojas de estilos XSLT, una por cada operador de mutación.

3.2. Operadores de mutación en Java

Son varias las clasificaciones para los diferentes operadores de mutación que se proponen en los distintos estudios realizados para el lenguaje de programación Java. Algunas de las encontradas no son realmente una clasificación, ya que muchos de los estudios se centran en un tipo concreto de operador: concurrentes [21], captura de excepciones [22], tradicionales [3, 23], propios del lenguaje [3, 23], orientados a objetos [3], clases [3, 23], ocultación de información [24], herencia [24], polimorfismo [24] y sobrecarga de métodos [24].

Uno de los primeros pasos que necesitaremos para la realización es la creación de una clasificación única de operadores de mutación. Esto permitiría disponer de una herramienta que abarcara el conjunto completo de operadores de mutación disponibles para el lenguaje Java.

3.3. Análisis del programa original

Uno de los componentes de la arquitectura a desarrollar será el analizador. Éste se encargará de determinar el conjunto de operadores que se puede aplicar al programa original.

Para ello se quiere reutilizar la aplicación Judy, que consta de un analizador, pero limitado a un subconjunto de los operadores de mutación disponibles.

Nuestro trabajo será, por un lado, ampliar Judy para el reconocimiento de todos los operadores de mutación disponibles, y por otro, realizar la conexión con el motor de generación de GAmEra.

Antes de desvelar los motivos que nos han llevado a la selección de Judy para elaborar nuestro trabajo, describimos las diferentes herramientas de pruebas de mutación para el lenguaje Java.

3.3.1. JavaMut

JavaMut [19] no está disponible de manera gratuita para descargar, no soporta la ejecución de mutantes y compila de forma separada cada clase de mutante (lo que es un proceso muy lento). Esto

dificulta su elección para el analizador al no disponer del código y poder así reutilizarlo y adaptarlo a nuestra arquitectura.

3.3.2. MuJava

MuJava [6, 7, 8] puede verse como un sucesor de JavaMut que ofrece un gran conjunto de operadores de mutación tradicionales y orientados a objetos para el lenguaje Java. Sin embargo tiene algunas limitaciones:

- No proporciona una interfaz de línea de comando.
- No trabaja con JUnit. Sin embargo el reciente plugin Muclipse actúa como un puente entre el motor de mutación de MuJava y el IDE de Eclipse, parece que resuelve esta limitación. De hecho Muclipse muestra algunas limitaciones referentes a la automatización de la generación de mutantes.
- También hay limitaciones con el proceso de prueba (no es posible seleccionar más de una clase de forma simultánea, y es necesario seleccionar una prueba adicional por cada clase de forma explícita).

No obstante, es la herramienta de prueba de mutaciones más extendidas. El problema es que el proceso de generación de mutantes en bytecode dificultaría su integración con GAmEra que emplea código fuente.

3.3.3. Jester

Jester [16] es una herramienta que inicialmente tuvo bastante aceptación en el ámbito de la prueba del software en Java. Sin embargo, Offutt [25] argumentó que Jester resultó ser una forma muy cara para aplicar las pruebas de ramificación, más que las pruebas de mutación, debido a un mecanismo simplificado de la generación de mutantes. Los problemas que conciernen al rendimiento y fiabilidad de la herramienta, así como un rango limitado de posibles operadores de mutación (basados solo en la sustitución de cadenas). Además, los operadores de mutación ofrecidos por Jester no son sensibles al contexto, y a menudo producen mutantes no válidos. Debido a esta gran desventaja, Jester se excluyó como herramienta a reutilizar.

3.3.4. Jumble

Jumble [17] opera directamente en nivel de bytecode para acelerar la prueba de mutaciones. Soporta JUnit y ha sido desarrollado en un entorno industrial, aunque después fue destinado a la investigación. Desafortunadamente, Jumble soporta un conjunto limitado de operadores de mutación, siendo versiones simplificadas de AOR, ASR, COR, LOR, ROR and SOR estudiados en [26]. Por tanto, Jumble tiene un analizador del lenguaje muy simplificado puesto que modela sus propios operadores de mutación, lo que unido a la generación de mutantes bytecode dificulta su integración en GAmEra.

3.3.5. Judy

Judy [3] es una implementación del enfoque de FAMTA Light (rápido algoritmo de prueba de mutaciones orientado a aspectos) desarrollado en Java con extensiones AspectJ. Las características del núcleo de Judy es su alto rendimiento del proceso de la prueba de mutaciones así como su fácil integración con herramientas de desarrollo en el ámbito de la prueba del software en Java, lo que facilita la total automatización del proceso de la prueba de mutaciones y soporte para la última versión de Java. Sin embargo, Judy soporta únicamente 16 operadores de mutación predefinidos, por lo que necesitaría extenderse para el soporte del resto de operadores de mutación. No obstante, la disponibilidad de código de la herramienta y su documentación, la convierten en una opción válida.

3.3.6. Response Injection (RI)

RI [18] es un prototipo que se aprovecha de los mecanismos orientados a aspectos para generar mutantes. Se usa solamente para operadores de mutación simples, por ejemplo cambiar los tipos primitivos y los objetos de la clase Cadena. A diferencia de MuJava y Judy, la única mutación aplicada a los objetos es el valor nulo. Además, no está claro cómo RI puede soportar el ancho rango de operadores de mutación diseñados para Java. Otra limitación es el enfoque de RI que no permite cambios en las pruebas en objetos pasados como parámetros de un método. La documentación, así como el código del prototipo no están disponibles. Por ello, RI no nos parece adecuada para su reutilización.

Analizadas las distintas herramientas disponibles, nos decantamos por Judy, por sus características

de tratamiento de operadores de mutación, su disponibilidad y documentación.

4. Conclusiones y trabajo futuro

Se ha propuesto emplear la técnica de mutación evolutiva para programas escritos en Java. En ella se pretende obtener un menor número de mutantes a ejecutar, pero siendo éstos los de mayor calidad, de tal forma que el coste computacional no sea el principal inconveniente para este tipo de aplicaciones. Esto se obtiene tras el empleo de algoritmos genéticos para la generación de los mutantes. Si se demuestra que el coste computacional de esta nueva propuesta es menor que la del resto de herramientas que ejecutan todos los mutantes, llegaremos a la conclusión de que la técnica de mutación evolutiva es una de las optimizaciones a tener en cuenta a la hora de realizar estudios de prueba de mutaciones.

Se propone una arquitectura para la optimización de las pruebas de mutaciones en Java. Esta arquitectura reutilizaría el sistema generador de mutantes de GAmEra, y se ampliaría con un analizador basado en Judy, así como un sistema de ejecución.

Como trabajo futuro está la implementación de la arquitectura propuesta en este artículo. En primer lugar, habría que ampliar Judy para el reconocimiento de todos los operadores de mutación disponibles, y luego, realizar la conexión con el motor de generación de GAmEra.

Referencias

- [1] DeMillo, R.A., Lipton, R.J., and Sayward, F.G.: "Hints on Test Data Selection: Help for the Practicing Programmer", IEEE Computer, 1978, 11, (4), pp. 34 – 41.
- [2] Hamlet, R.G.: "Testing Programs with the Aid of a Compiler", IEEE Trans. Softw. Eng. 1977, 3, (4), pp. 279 – 290.
- [3] Madeyski, L. & Radyk, N. 2010, "Judy—A Mutation Testing Tool for Java", IET Softw, vol. 4, no. 1.
- [4] Estero Botaro, A., Domínguez Jiménez, J. J., Gutiérrez Madroñal, L.: "GAmEra: una herramienta para la generación y selección mediante algoritmos genéticos de mutantes WS-BPEL", en XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD), San Sebastián (2009), págs. 66 – 69.

- [5] King, K. N., Jefferson Offutt, A., "A Fortran language system for mutation-based software testing, Software"—Practice & Experience, v.21 n.7, p.685-718, July 1991.
- [6] Ma, Y., Offutt, J., Kwon, Y.R.: "MuJava: an automated class mutation system", *Softw Test Verif Rel*, 2005, 15, (2), pp. 97-133.
- [7] Offutt, J., Ma, Y.S., and Kwon, Y.R.: "An Experimental Mutation System for Java", *SIGSOFT Softw. Eng. Notes*, 2004, 29, (5), pp. 1-4.
- [8] Ma, Y.S., Offutt, J., and Kwon, Y.R.: "MuJava: A Mutation System for Java". *Proc. 28th Int. Conf. on Software Engineering*, Shanghai, China, May 2006, pp.827-830.
- [9] Tuya, J., Suarez-Cabal, M. J., & de la Riva, C. (2006). *SQLMutation: A tool to generate mutants of SQL database queries*. Mutation Analysis, 2006. Second Workshop on, 1-1.
- [10] Domínguez Jiménez, J. J., Estero Botaro, A., García Domínguez, A., Medina Bulo, I.: "Mutación evolutiva". En *XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, Valencia (2010). Aceptado para su publicación.
- [11] Offutt, A.J., Untch, R.H.: "Mutation 2000: uniting the orthogonal. Mutation testing for the new century", pág. 34–44 Kluwer Academic Publishers, Norwell, MA, USA (2001).
- [12] Zhu, H., Hall, Patrick.A.V., and May, J.H.R.: "Software Unit Test Coverage and Adequacy". *ACM Comput. Surv.* 1997, 29, (4), pp. 366 – 427.
- [13] Ayari, K., Bouktif, S., Antoniol, G.: Automatic mutation test input data generation via ant colony. En: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pág. 1074-1081 (2007).
- [14] Offutt, J.: Automatic test data generation. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA (1988).
- [15] Estero-Botaro, A., Domínguez-Jiménez, J. J., Medina-Bulo, I., "Una arquitectura para la generación de casos de prueba de composiciones WS-BPEL basada en mutaciones".
- [16] Moore, I.: "Jester - a JUnit test tester". *Proc. 2nd Int. Conf. on Extreme Programming and Flexible Processes in Software Engineering*, Sardinia, Italy, May 2001, pp. 84-87.
- [17] Irvine, S. A., Tin, P., Trigg L., Cleary, J. G., Inglis, S., and Utting, M.: "Jumble Java Byte Code to Measure the Effectiveness of Unit Tests". *Proc. Testing: Academic and Industrial Conf. Practice and Research Techniques*, Windsor, UK, September 2007, pp. 169-175.
- [18] Bogacki, B., and Walter, B.: "Aspect-Oriented Response Injection: An Alternative to Classical Mutation Testing". *Proc. IFIP Work. Conf. on Software Engineering Techniques*, Warsaw, Poland, October 2006, pp. 273-282.
- [19] Chevalley, P., Thévenod-Fosse, P.: "A mutation analysis tool for Java programs", *Int. J. Softw. Tools Tech. Transfer*, 2003, 5, (1), pp. 90-103.
- [20] Ma, Y.S., Offutt, J.: "Description of Method-level Mutation Operators for Java", November 2005, <http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf>
- [21] Bradbury, J.S., Cordy, J.R. & Dingel, J. 2006, "Mutation Operators for Concurrent Java (J2SE 5.0)", *Mutation Analysis*, 2006. Second Workshop on, pp. 11.
- [22] Changbing, J., Zhenyu, C., Baowen, X. & Ziyuan, W. 2009, A New Mutation Analysis Method for Testing Java Exception Handling.
- [23] Smith, B.H. & Williams, L. 2009, "Should software testers use mutation analysis to augment a test set?", *The Journal of Systems & Software*, vol. 82, no. 11, pp. 1819-1832.
- [24] Yu-Seung, M., Yong-Rae, K. & Offutt, J. 2002, "Inter-class mutation operators for Java", *Software Reliability Engineering*, 2002. *ISSRE 2002. Proceedings. 13th International Symposium on*, pp. 352.
- [25] Offutt, J.: "An analysis of Jester based on published papers". <http://cs.gmu.edu/~offutt/jester-anal.html>, accessed September 2006.
- [26] Ammann, P., and Offutt, J.: 'Introduction to Software Testing' (Cambridge University Press, 2008, 1st edn.).
- [27]).