$Para\mu$ - A Partial and Higher-Order Mutation Tool with Concurrency Operators

Pratyusha Madiraju
Advanced Empirical Software Testing and Analysis Research Group (AVESTA)
Department of Computer Science
Texas Tech University
Lubbock, TX, USA
Email: pratyusha.madiraju@ttu.edu

Akbar Siami Namin
Advanced Empirical Software Testing and Analysis Research Group (AVESTA)
Department of Computer Science
Texas Tech University
Lubbock, TX, USA
Email: akbar.namin@ttu.edu

***Abstract*—The mutation operators implemented in a mutation tool typically mutate the entire programs thoroughly and thus generate enormous number of mutants spread all over the programs. However, the complexity and difficulty level of methods implemented in a given program is less evenly distributed all over the program. Hence, the non-uniform distribution of complexity of methods in a program is an indication of testing efforts required for each method.**

We introduce partial mutations where only the complex parts of the programs are mutated instead of the entire programs. $Para\mu$ is a mutation tool for Java programs enabling partial mutations. In addition to the class mutation, $Para\mu$ implements concurrency mutation operators to address the recent advances in multicore systems and hence mutation testing of parallel and multi-threaded programs. Furthermore, $Para\mu$ allows higher-order mutations by which the users are allowed to specify the order and the types of mutation operators and thus perform a combinatorial higher-order mutation testing.

***Keywords*-mutation testing; higher-order mutation; object-oriented programs; software testing tools;**

I. Introduction

Mutation testing is a fault-based testing technique where synthetic faults are seeded into a program. In addition to assessing the adequacy of test cases, the application of mutation analysis in experimental studies and assessing software testing techniques have been of interest [1]. It has been reported that seeded faults behave similarly to real faults and thus mutation is an appropriate tool for testing experiments [2].

A typical mutation procedure applies a set of mutation operators to a given program and transforms *all* instances of a particular element into similar elements in the same class. More subtly, one may mutate the entire program thoroughly regardless of the type and complexity level of methods implemented in a given code. Though mutating and testing the entire code enhances the quality and thoroughness of test cases developed, the complete mutation of a program generates an enormous number of mutants, where detecting most of which is less important for the test practitioners.

Though selective mutations and determination of a sufficient set of mutation operators may be useful in reducing the cost of mutations and hence enhancing the feasibility of mutations, a more pragmatic approach is needed to prevent generating less important mutants. A motivating fact is that a program consists of a number of methods with various degrees of complexity. Thus, it makes sense to spread the testing efforts *less* evenly throughout the given program.

We introduce *partial* mutations by which some parts of a given code are mutated instead of the entire program. The partial mutation testing directs testing focus towards more complex parts of the programs and hence generate mutants only for the parts the test practitioners are interested in. This paper introduces $Para\mu$, a partial mutation tool enabling the users to mutate only the important and more complex parts of programs. In addition to partially mutating Java programs, the $Para\mu$ mutation tool allows higher-order mutations and generates a combinatorial set of mutants based on some of the operators selected.

A recent survey on the development of mutation testing indicates that mutation testing has been increasingly widely studied and there has been much research work on the various kinds of techniques seeking to turn mutation testing into a practical testing approach [3]. Hence, the tool support for mutation is an important need. The $Para\mu$ mutation tool addresses the recent technology advances in the multicore technology and the tremendous needs for developing and thus testing parallel and multi-threaded applications by implementing concurrency mutation operators. The mutants generated based on concurrency operators are capable of testing parallel programs for concurrency faults, e.g., deadlock, starvation and interleaving. *The main contributions of this paper are:*

- *Introducing partial mutations to mutate some parts of a given code instead of the entire program.*
- *Highlighting the advantages of bytecode mutations and comparing bytecode instrumentation tools from the mutation's point of view.*
- *Introducing $Para\mu$, a partial and higher-order mutation tool with concurrency mutation operators developed for Java programs.*

This paper is organized as follows: We review the related work in Section II. Section III introduces partial mutations. Section IV introduces $Para\mu$, a partial and higher-order mutation tool with concurrency mutation operators developed for mutating Java programs. We discuss some issues regarding partial mutations and concurrency operators in Section V. Section VI concludes.

II. Related Work

A. Mutation Tools

Mothra, a mutation system for testing Fortran programs, implements only 22 mutation operators designed for imperative languages [4]. Proteum is a mutation tool for the assessment of test adequacy for C programs [5]. Proteum implements a large number of mutation operators. The 108 mutation operators implemented by Proteum are based on a comprehensive set of mutation operators designed by Agrawal et al. [6]. Milu [7] is C mutation testing tool designed for both first order and higher order mutation testing. MILU allows customization of the set of mutation operators to be applied.

MuJava is a mutation system for Java programs [8]. MuJava implements both method and class level mutations known as traditional [9] and class mutation operators [10], [11]. MuJava implements only 5 traditional and 24 class mutation operators. MuClipse is the eClipse version of the MuJava mutation tool [12].

Javalanche [13] is an open source framework for mutation testing Java programs with a special focus on automation, efficiency, and effectiveness. Javalanche assesses the impact of individual mutations to effectively filter out equivalent mutants. Jumble [14] is a mutation tool for Java programs that mutates the bytecode of Java programs. Jumble provides only traditional method level mutations.

B. Concurrency Mutation Operators

The technology advances in the multicore systems emphasize the importance of parallel programming. Bradbury et al. have designed a set of mutation operators for concurrent Java programs [15]. Bradbury et al. [15] classify the concurrency mutation operators into five categories: a) Modify parameters of concurrent methods, b) Modify the occurrence of concurrency method calls (removing, replacing and exchanging), c) Modify keywords (addition and removal), d) Switch concurrent objects, and e) Modify critical regions (shift, expand, shrink and split). MutMut is an exploration tool for mutation testing of multi-threaded applications [16].

C. Higher-Order Mutations

Historically, mutation testing creates mutants by injection of a single synthesis fault. The Higher-Order Mutations seed more than one fault in generating mutants. Harman et al. [17] argue that higher-order mutations are better able to simulate real faults than the typical first-order mutations. Jia and Harman [18] introduce the concept of subsuming Higher-Order Mutations (HOM).

Langdon et al. [19] propose semantic mutations instead of syntactic ones and state that the generated mutants should behave similarly to the original program in terms of the faults injected. They explored the relationship between syntactic and semantic using genetic programming systems. It has been stated that the syntactic difference between mutants and the original program is based on the changes made by the genetic programming, whereas, the semantic difference is defined as the number of tests which kill the mutant.

D. The Byte Code Instrumentation Tools

Tool kits including Javaassist (Java Programming Assistant) [20], JMangler [21], BCA (Binary Component Adaption) [22], BCEL (Byte Code Engineering Library) [23], JOIE (Java Object Instrumentation Environment) [24], and ObjectWeb ASM [25] are Java libraries publicly available for byte code manipulation.

BCA is highly flexible instrumentation tool that offers a wide range of modifications libraries for manipulating the object-oriented features. The adaptations are distributed and performed as "Just-In-Time" (JIT), i.e., on the fly. BCA also incurs only a small amount of overhead [22]. The major issue using BCA is that it requires a modified JVM.

ObjectWeb ASM has a different mechanism than other instrumentation tools in which it uses the "visitor" design pattern without explicitly representing the visited tree with objects. This enables better performance compared to other tool kits. ObjectWeb ASM completely hides the constant pool management from the user. Some tools and projects that have used ObjectWeb ASM toolkit are *AspectWerkz*, *Speedo*, *Groovy*, *dynaop*, and *BeanShell*.

BCEL has been widely used in many tools which require byte code manipulation. It allows all kinds of modifications to be made on the byte code. There are two separate hierarchies of components within BCEL - one for inspecting existing code and the other for creating new code. BCEL enables complex class transformations. It enables the modifications which explicitly require the manipulation of the `constant pool`. It also solves the serialization and de-serialization issue as well as instruction address management problem. Several software analysis tools have been developed using BCEL including *fprofiler*, *cglib*, *Cougaar Memory Profiler*, and *Gretel (residual test coverage)*. Table I summaries the pros and cons of these instrumentation tools.

III. Partial Mutations

Mutation testing is a powerful technique for unit testing, but it incurs high computational costs. Regular mutation of the entire source code generates a large number of mutants. All the generated mutants may not be required for further testing purposes. Research [26] shows that the number of

Table I
A COMPARISON OF THE BYTE CODE INSTRUMENTATION TOOLS.

Toolkit	Advantages	Disadvantages	Suitable for
BCEL	– Solves serialization and de-serialization problems – Solves the instruction address management problem – Allows all kinds of modification of class files – No need to modify JVM	– Manual update of the `constant pool` – Difficult to use	Class mutations
ASM	– Solves the serialization and de-serialization problem – Hides the `constant pool` management from the user – Solves the instruction address management problem – Allows all kinds of modifications of class files – No need to modify JVM – Provides tools for data-flow and control-flow analysis	– Difficult to use	Class mutations
BCA	– Easy to use	– Some restrictions on modifications of class files – Requires modifications to JVM	Traditional mutations
Javaassist	– Easy to use – Enables source level abstraction	– Some restrictions on modifications of class files	Traditional mutations

mutants generated for a program are approximately proportional to the product of the number of data references times the size of the set of data objects. Typically, this production is a large number. Since, at least every test case has to be executed on each of the mutants generated.

Not all parts of a given code are similar. It is not necessary to mutate all parts of the code. Also, mutation of some parts of the code might generate some mutants which may not be useful and easy to be killed. In order to reduce the computational costs involved, one way is to reduce the number of mutants generated by applying selective mutation. Selective mutation can be done in many ways. One of the approaches proposed previously by Offutt et al. [27] is to avoid the application of some of mutation operators on the source code, which generate a large number of mutants.

In this paper, we suggest that only the complex parts of the code, which need to be inspected, be mutated[1]. We introduce the partial mutation approach where fault injections occur into specific areas of the code rather than the entire program. This can be done by first selecting the parts of the code, i.e., methods, to be mutated, selecting the mutation operators to be applied, and then generating the mutants by applying the selected operators on the selected parts of the code. More subtly, the partial mutations performs mutations at the method level instead of class level.

IV. $Para\mu$ - A PARTIAL AND HIGHER-ORDER JAVA BYTE CODE MUTATION TOOL WITH CONCURRENCY OPERATORS

A number of Java class and concurrency mutation operators have been implemented in a tool called $Para\mu$. The tool has three major functionalities a) mutants generation b) mutants analysis c) running the test suite provided by the tester to measure the effectiveness of the test suite. Initially the tool was developed to address the first and second functionalities and to support mutation analysis in experimental studies. However, we plan to extend the tool by incorporating the test suite executions and measuring mutation score (MS). The tool enables partial mutation by allowing the user to select the parts of the code which have to be mutated. In addition to partial mutations, $Para\mu$ allows higher order mutations at bytecode level. The tool is a great asset for mutation testing of concurrent programs and in particular multi-threaded Java applications, since it implements concurrency mutation operators as designed by Bradbury et al. [15].

[1]We are not aware of any research work/mutation tool discussing/performing partial mutations.

A. The Mutation Operators Implemented

Our initial motivation of developing $Para\mu$ was to provide the research community a mutation tool that implements concurrency mutation testing and capable of testing concurrent and multi-threaded applications. As Bradbury et al. [15] point out the traditional and class mutation operators are not sufficient to test the faults caused by the concurrency in Java programs. It is believed that the concurrency mutation operators, when used along with the traditional and class level operators provide a more comprehensive set of mutation operators for the comparison and improvement of quality assurance of test cases and analysis for concurrency [15].

The operators implemented in $Para\mu$ so far are listed in Table II. We have implemented three concurrency mutation operators as well as two class mutation operators. We plan to continue implementing the concurrency mutation operators as designed by Bradbury et al. [15] to support mutation testing of concurrent Java programs. In addition to the concurrency mutation operators, we would like also implement some of the class mutation operators, to make $Para\mu$ a more comprehensive and multi-purpose mutation tool.

B. Partial Mutations

The tool developed lets the user select the methods of the class to be mutated. The user can also select the type of mutation operator to be applied, depending on the feature of the class that needs to be modified. The selection of operators can be done from class and concurrency mutation operators implemented. We do not consider the traditional mutation operators designed for procedural languages as we

Table II
THE MUTATION OPERATORS IMPLEMENTED SO FAR IN $Para\mu$.

OP	Description	Type
AMC	Access Modifier Change	Class
OMD	Overloaded Method Deletion	Class
ASTK	Add Static Keyword to Method	Concurrency
RSTK	Remove Static Keyword from Method	Concurrency
RSK	Remove Synchronized Keyword from Method	Concurrency

aim to generate mutants which affect the object-oriented and concurrency features of an object-oriented programming language like Java.

C. The Implementation Detail

A use interface lets the user browse the class files and select a class file to be mutated. After the selection of the class file to be mutated, the user interface provides a list of methods present in the class. We used API provided by BCEL to obtain the name of methods instead of using Java reflection. The user can choose the methods of the class file selected to be mutated by selecting one or more methods from the list displayed enabling partial mutations. $Para\mu$ lets users choose the order of mutations. So far the $Para\mu$ mutation tool supports the first and second order mutations.

A list of class and concurrency mutation operators is populated in a list box dynamically, depending on the type of mutation operators selected by the user. The class file to be mutated is stored in an object of the class, `JavaClass` provided by the BCEL instrumentation package. The object-oriented information about the class file is retrieved and the required mutation is performed on the Java class file. The `constant pool` of the Java class is updated and the modified class file is dumped, i.e., saved, to the hard disk. The mutated class file can be executed using the Java class loader to verified the modified behavior of the original class file.

BCEL (Byte Code Engineering Library) tool kit has been used to generate mutants at the byte code level. The tool developed makes use of the *reflection* property. It is a feature of the Java programming that allows an executing of Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. A package in BCEL reflects the Java class and provides information about the static constraints of the class file. It allows the retrieval of information such as finding the methods of a class, access modifiers of the class, and methods, etc. This is necessary before performing the mutation, as we need to know the existing constraints of the class file in order to modify them. Instead of `reflection API` provided by Java 5.0, the BCEL API has been used to retrieve object-oriented information about the class files under mutations.

V. DISCUSSION

A. Source vs. Byte Code Mutations

The mutation procedure can be applied either at the source code level or the byte code format. Bytecode translation is a technique that has some abilities in common with *reflection*, but uses a very different approach. Bytecode translation inspects and modifies the intermediate representation of Java programs, bytecode. Since the bytecode translation directly deals with the bytecode format, it has some advantages over source-level translation. First, it can process an off-the-shelf program or library that is supplied without source code. Second, it can be performed on demand at load time, i.e., on-fly, when the Java virtual machine loads a class file.[8]. Third, it avoids the recompilation issues, thereby reducing the overhead incurred in recompilation.

1) The Structure of Java Class Files: The header section of a class file contains a magic number and the version number. The `constant pool` can be thought of as the text segment of an executable[23]. The constant pool followed by the `access rights` section which describes the access rights of the class file which are encoded by a bit mask. The implemented interfaces section lists the interfaces implemented by the class. The `fields` and `methods` sections list the fields and methods of the class. Finally, the `class attributes` section describes the attributes of the class file. The class attributes provide a mechanism to insert additional and user-defined information into the class file data structures.

B. Limitations of Bytecode Mutations

There are certain limitations with implementing some of mutation operators. For example, the mutation operators corresponding to switching concurrent objects, e.g. Exchange Explicit Lock Objects (**EELO**), might be very cumbersome to implement. As a statement in source code is associated with one or more instructions in the low level representation, byte code transformations tend to become more complex with increasing number of instructions associated with a code block. In order to implement operators related to mutation of switching concurrent objects, the instructions corresponding to the concurrent objects and concurrent code have to be traced out. Not all statements of Java program have associated keywords in byte code. In order to identify the locks, a region of byte code has to be traced out and information has to be retrieved and analyzed before making changes. Also, some core Java classes may be excluded from instrumentation. As stack operations are the primary way of manipulating byte code, it is possible that writing invalid index values onto it could cause inconsistencies in the byte code.

C. Concurrency Mutation Operators and the Non-Deterministic Executions of Parallel Programs

Bradbury et al. [15] propose 24 concurrency mutation operators classified into five categories. The concurrency operators designed include a broad set of mutation operators including mutating concurrency keywords as well as modifying critical regions. Though the set of concurrency mutation operators proposed by Bradbury et al. is comprehensive enough to address possible concurrency defects, the non-deterministic nature of parallel and multi-threaded programs raises some concerns regarding the effectiveness of these concurrency operators.

The parallel and multi-threaded applications are usually non-deterministic primarily because of interleaving of threads sharing some memory variables. The thread's interleavings are imposed by the scheduling algorithms implemented by the underlying operating systems or virtual machines. It is believed that not only developing parallel programs is hard, testing concurrent systems is even harder mainly due to the non-deterministic behaviour of these programs.

A typical mutation testing procedure starts with generating some mutants, executing a number of test cases on mutants as well as on the original program, and finally comparing the outputs produced by mutants and the original program. Any instance of differences in the outputs produced is an indication of detecting mutants and thus adequate test cases. The mutation testing procedure, however, requires the original program to be deterministic. This issue raises the concerns of whether the purpose of mutations on testing concurrent programs is for assessing the adequacy of test cases?

Consider, for instance, the concurrency operator *Remove Synchronized Keyword from Method* (**RSK**). The application of this operator poses non-determinism to the program by enabling multiple threads to access a critical region where some shared variables are set or used. The mutants generated by this operator seemingly produce different outputs than the original programs. The non-deterministic behaviour of the mutants generated by this operator indicates that the application of RSK operator produces either *easy to kill*, i.e., interleaving faults, or *equivalent*, i.e., no data races, mutants. Similar to the RSK operator, the application of some other concurrency operators may generate either easy to detect or equivalent mutants indicating an improper use of mutation testing and hence inappropriate use of mutations for assessing the adequacy of test cases developed.

We argue that mutations can be used to explore some possible thread's interleavings rather than for assessing the adequacy of test cases. For instance, consider the concurrency operator *Modify Method-X Timeout* (**MXT**) where numerical values are passed as parameters to `wait()`, `sleep()`, `join()`, and `await()` method calls. This operators simulates possible timings that *may* expose possible interleaving faults. However, it remains a question whether using mutations to explore *some* thread's interleavings is an appropriate way for testing whether there is any instance of interleaving faults in concurrent programs?

D. *Higher-order Mutations*

Higher order mutants are generated by applying the mutation operators more than once to the original program. The higher-order mutation injects more than one fault into the same program. Higher order mutants can be generated in several ways. Two simple higher-order mutations include: a) Applying the same mutation operator more than once on the same class file, b) Applying different mutation operators on the same version of a program.

Traditionally, mutation testing is concerned with first-order mutations where only one instance of fault seeding occurs. Traditional mutation considers only first-order mutants created by inserting a single fault [18]. The seeded faults are trivial and are informative for determining the adequacy of test cases. Empirical research is required to investigate the cost-effectiveness of the higher-order mutations and in particular determining the relationship between the mutation order, e.g. second or third-order mutations, and the fault-detection ability of adequate test cases developed.

VI. Conclusion and Future Work

We introduced $Para\mu$, a mutation tool for mutating the bytecode of Java programs. The tool implements higher-order mutations enabling users to specify the order and the type of operators to be applied in mutations. In this paper, we introduced partial mutations where specific parts of code are mutated instead of the entire programs. The partially mutating programs reduces the cost of mutations significantly by preventing the generation of unnecessary mutants. The $Para\mu$ mutation tool has its primary focus on implementing concurrency mutation operators and mutating multi-threaded Java applications.

The $Para\mu$ mutation tool is still in its early stages of development. So far, only 5 mutation operators, 2 class and 3 concurrency operators, are implemented. We plan to implement the entire set of concurrency mutation operators as designed by Bradbury et al. [15]. The tool, so far, only generates mutants suitable for mutation analysis. The future work is to integrate the mutation analysis, test case executions, measuring mutation score, and identifying killed and alive mutants into the GUI end thus enabling $Para\mu$ performing mutation testing.

The idea of partial mutation introduced by $Para\mu$ may introduce a new research direction in mutation testing. Further empirical studies are needed to investigate the trade-offs of using partial mutations. In particular, we would like to integrate an intelligent mechanism using machine learning techniques to analyze the complexity of methods and thus automatically make suggestions for users to identify and mutate complex methods. We also would like to integrate a bytecode verifier tool to automatically verify the soundness of the bytecode manipulations and mutants generated. Furthermore, experimentations are required to investigate the cost-effectiveness of applying higher-order mutations and the order of mutations.

References

[1] J. H. Andrews, L. C. Briand, Y. Labiche, and A. Siami Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608 – 624, August 2006.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, May 2005.

[3] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 99, 2010.

[4] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software, Practice Experiment*, vol. 21, no. 7, pp. 685–718, 1991.

[5] M. E. Delamaro and J. C. Maldonado, "Proteum – a tool for the assessment of test adequacy for C programs," in *Conference on Performability in Computing Systems (PCS 96)*, New Brunswick, NJ, July 1996, pp. 79–95.

[6] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," Department of Computer Science, Purdue University, Tech. Rep. SERC-TR-41-P, April 2006.

[7] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language," in *Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08)*, Windsor, UK, 29-31 August 2008, pp. 94–98.

[8] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system," *Software Testing, Verification, and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[9] Y.-S. Ma, "Description of method level mutation operators for java," Electronics and Telecommunications Research Institute, Korea, Tech. Rep., March 2005.

[10] J. Offutt, Y.-S. Ma, and Y. R. Kwon, "The class-level mutants of MuJava," in *Proceedings of International Workshop on Automation of Software Test (AST)*, 2006, pp. 78–84.

[11] Y.-S. Ma, Y. R. Kwon, and J. Offutt, "Inter-class mutation operators for java," in *Proceedings of International Symposium on Software Reliability Engineering (ISSRE)*, 2002, pp. 352–366.

[12] B. Smith and L. Williams, "An empirical evaluation of the mujava mutation operators," in *Testing: Academic and Industrial Conference Practice and Research Techniques*, 2007, pp. 193–202.

[13] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for java," in *ESEC/SIGSOFT FSE*, 2009, pp. 297–298.

[14] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, and M. Utting, "Jumble java byte code to measure the effectiveness of unit tests," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 169–175.

[15] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *Proceedings of the 2nd Workshop on Mutation Analysis (Mutation 2006)*, November 2006, pp. 83–92.

[16] M. Gligoric, V. Jagannath, and D. Marinov, "Mutmut: Efficient exploration for mutation testing of multithreaded code," in *Proceedings of International Conference on Software Testing (ICST)*, 2010, pp. 55–64.

[17] M. Harman, Y. Jia, and W. B. Langdon, "A manifesto for higher order mutation testing," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ser. ICSTW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 80–89.

[18] Y. Jia and M. Harman, "Higher order mutation testing," *Information Software Technology*, vol. 51, pp. 1379–1393, October 2009.

[19] W. B. Langdon, M. Harman, and Y. Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *Journal of Systems and Software*, vol. 83, no. 12, pp. 2416–2430, 2010.

[20] S. Chiba, "Java programming assistant (javassist)," www.javassist.org/, October 2010.

[21] G. Kniesel, P. Costanza, and M. Austermann, "Jmangler - A framework for load-time transformation of java class files," in *First IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, November 2001, pp. 98–108.

[22] R. Keller and U. Hölzle, "Binary component adaptation," in *Proceedings of 12th European Conference Object-Oriented Programming(ECOOP)*, 1998, pp. 307–329.

[23] M. Dahm, "Byte code engineering with the bcel, api," Freie Universität Berlin – Institute für Informatik, Tech. Rep. B–17–98, 1998.

[24] G. Cohen, J. Chase, and D. Kaminsky, "Automatic program transformation with joie," in *Proceedings of the 1998 USENIX Annual Technical Symposium*, 1998.

[25] E. Bruneton, *ASM 3.0 – A Java bytecode engineering library*, 2007.

[26] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, New Haven, CT, USA, 1980.

[27] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of International Conference on Software Engineering (ICSE)*, 1993, pp. 100–107.