Dynamic Localisation and Automatic Correction of Software Faults using Evolutionary Mutation Testing

Pantelis Stylianos Yiasemis and Andreas S. Andreou
Department of Computer Engineering and Informatics, Cyprus University of Technology,
31 Archbishop Kyprianou ave., Limassol, Cyprus

Keywords: Mutation Testing, Fault Localization and Correction, Genetic Algorithms.

Abstract: Software testing has recently turned to more "sophisticated" techniques, like automatic, self-adaptive mutation testing, aiming at improving the effectiveness of the testing process and handling the associated complexity that increases the level of time and effort spent. Mutation testing refers to the application of mutation operators on correctly functioning programs so as to induce common programming errors and then assess the ability of a set of test cases to reveal them. In the above context the present paper introduces a novel methodology for identifying and correcting faults in Java source code using dynamic slicing coupled with mutation testing and enhanced by Genetic Algorithms. A series of experiments was conducted to assess the effectiveness of the proposed approach, using different types of errors and sample programs. The results were quite encouraging as regards the ability of the approach to localise the faults, as well as to suggest the appropriate correction(s) to eliminate them.

1 INTRODUCTION

A significant research topic in the area of Software Engineering is how to increase the productivity and quality of the software products. A major factor that affects negatively the aforementioned issues is the presence of faults. Faults are the source of erroneous behavior on behalf of a system known as failure. According to the IEEE standard Glossary of Software Engineering Terminology failure is defined as the inability of a system or component to correctly execute the functions defined by the required specifications. Fault is an incorrect step, procedure or data definition in a program, which summed they lead to failure. They are sometimes called as problems, errors, anomalies, inconsistencies or bugs (Patton, 2006).

There are cases in literature where the existence of faults in software systems of great importance and cost made the headlines of newspapers. Some examples of such significant failures include Disney's Lion King in 1994-1995, Intel's Pentium Floating-Point Division Bug in 1994, NASA's Mars Polar Lander in 1999, Patriot Missile Defense System in 1991, Y2K (Year 2000) Bug originated decades back - circa 1974 and the Dangerous Viewing Ahead in 2004 (Patton, 2006).

The software testing process may be divided into two sub-processes; failure detection, which is performed during the execution of a program and debugging, where faults that lead to failure are being identified and corrected. Most of the developing firms spend around 50% to 80% of the total development time in software testing activities, trying to reduce the number of faults in their source code. Furthermore, the testing process may be considered as one of the most demanding, hard and frustrating set of activities in software development. The process of actually detecting faults in a software system takes up to 95% of the whole code debugging sub-process (Myers, 1979). It is obvious, therefore, that there is a need to develop highly effective fault detection methods to improve the effectiveness and ease the code testing process. This will subsequently reduce the amount of time code testing requires, increasing the quality of the code and the productivity of software developers.

In this study an innovative method is proposed for detecting and correcting faults in Java code by combining: Dynamic Code Slicing, Mutation Testing and Genetic Algorithms (GA). Specifically, the aim of the paper is to utilize the benefits of dynamic slicing so as to change the execution flow of an erroneously behaving program by replacing

Stylianos Yiasemis P. and S. Andreou A..
Dynamic Localisation and Automatic Correction of Software Faults using Evolutionary Mutation Testing.
DOI: 10.5220/0003992000150026
In *Proceedings of the 14th International Conference on Enterprise Information Systems* (ICEIS-2012), pages 15-26
ISBN: 978-989-8565-11-2

parts in the particular slice of code that contains one or more faults. The process of finding the erroneous line(s) and selecting the correct replacement(s) for eliminating the faults is a difficult problem with a large solution space. GA may successfully tackle this problem by reducing it to a search optimization problem.

The rest of the paper is organized as follows: Section 2 presents a short literature overview on fault localization and mutation testing; this section also outlines some Computational Intelligence techniques utilized in this area. Section 3 describes the technical background behind the algorithms adopted in the proposed solution. Furthermore, the basic principles behind Mutation Testing are introduced along with mutation operations and the MU Java system. Section 4 describes the aspects of the proposed methodology, while in section 5, the application prototype and the experimental results are presented. This section also demonstrates the design of the experiments and discusses the corresponding results. Section 6 provides a critical stand towards possible limitations of the proposed approach. Finally, section 7 draws our conclusions and outlines future research steps.

2 LITERATURE OVERVIEW

Code testing is a time and effort consuming process, therefore there is a strong need to improve its effectiveness and reduce the cost associated with completing its tasks. This led to studies trying to develop automated code testing methods or/and tools, which introduce novel algorithmic tasks that take advantage of computers' processing power to reduce human effort. In this context various methods have been suggested in literature, introducing Delta debugging, variations of dynamic programming, failure inducing chops and predicate switching. Below, we describe a few of these methods that lie within the area of fault localization and/or program slicing, as the latter constitutes a very promising approach that may assist in identifying and isolating faulty statements.

Fault Localization (FL) is the process of identifying suspicious code that may contain faults in the program and then examine it to decide if the identified code actually contains faults. FL is a heuristic method using dataflow tests, which supports execution slicing and dicing based on test cases (Agrawal, et al., 1995). Agrawal and Horgan (1990) observed that a fault is located in a slice that corresponds to a test case which failed during execution. The search for the fault may then concentrate on that slice only (dice) and the rest of the program may be ignored. The results showed that this method was able to detect some but not every fault inserted by independent observers, while in some cases the faults were not even included in the slices selected. Black et al. (2005) studied the characteristics of a program's slices in an attempt to identify those components that could contain faults. A slicing profile was formed with slicing metrics and dependence clusters to investigate if a reliable tool could be developed so as to identify the more fault prone components, yielding encouraging preliminary results. An experimental evaluation using dynamic slices for fault localization was performed by Zhang et al. (2005), who proposed a framework for dynamic slicing of programs with a long execution path. Three different dynamic slicing algorithms were implemented and compared, namely data slicing, full slicing and relevant slicing. The results concluded that the slices created with data slicing were smaller than the other two algorithms but they contained on average less faults. The full slices were larger than data slices and contained more faults, but were a bit smaller than the relevant slices which contained even more faults. Speed-wise data slicing was the fastest of the three, while relevant slicing the slowest.

Delta debugging is the process of defining the causes that specify how a program acts by focusing on the differences (deltas) between the current and the previous version of that program, as stated by Gupta in Gupta et al. (2005). Continuing, the method simplifies and isolates the input that causes the failure. For isolating and investigating the behavior of a certain input, delta debugging searches for the smallest test case scenario that becomes successful when this input is deleted from the test case. By utilizing the delta debugging dynamic algorithm, a method was introduced using the minimal failure inducing input that delta debugging identifies, to compute a forward dynamic slice and afterwards intersect the forward slice with the statements in the backward slice of the erroneous output to create a failure-inducing chop, Zhang et al. (2005). The results showed that these failure-inducing chops may aid in reducing the size of the search space without losing the ability to locate the erroneous code.

Mutation Testing (MT) is another technique that is based on the insertion of faults in a program and was first introduced in the late 70s by Hamlet (1977) and Demillo et al. (1978), while nowadays it is reported in various research studies dealing with software testing.(e.g. Nica et al. and Harman et al.). The general idea of MT is that faults commonly

made by programmers are induced in the initial programs to create a set of erroneous programs called mutants, each of them containing a specific change-fault. The mutants are executed with a set of test cases and the quality of the set is assessed by measuring how many faults are detected. Quality is evaluated using an adequacy score known as Mutation Score (MS), which is defined as the number of "killed" mutants (revealed faults) to the number of the non-equivalent mutants (undetected faults). Mutation analysis targets at increasing the mutation score, bringing it closer to 1, so that the set of test cases used are adequate to detect all the faults included in the mutants. MT has high computational costs, as it needs to execute all the test cases on every mutant, and requires substantial effort on behalf of the programmers.

Genetic Algorithms (GA) are search algorithms based on the principles of natural selection and genetic reproduction (Goldberg, 1989); (Jiang et al., 2008). Essentially, GA constitute a special class of optimization techniques that maintain a population of individuals each of which represents a possible solution to the problem in hand. The population is evolved using genetic operators that cross-over and mutate individuals trying to reach to better solutions in each generation. The fitness of each individual is evaluated using a dedicated function. Despite the fact that GA and program slicing algorithms have been extensively used in literature, no work has been reported thus far that combines the two for fault localization and correction. The work of Jiang et al. (2008) which describes how software faults may be localized based on a set of testing requirements and program slicing inspired the present paper as regards the encoding of the GA population. The authors presented an approach to identify dependence structures in a program that searches a superset of all possible slices to identify the set of slices that achieves maximum coverage. The framework yielded accurate results, showing in practice that it is possible to express problems of dependency analysis as search problems and that good solutions can be achieved in a reasonable time frame by using this technique.

Arcuri and Yao (2008) proposed a framework for automatic software bug fixing which used co-evolution where both programs and test cases co-evolve, aiming at fixing bugs in programs by influencing each other. The framework requires as input the faulty program and its formal specification. The work included some preliminary experiments that showed its potential applicability for any implementable program; essentially, it attempts to evolve the whole program tree, something which may prove costly and not efficient for large scale programs. Also, formal specifications are not always provided, a fact that makes the framework unusable in these cases.

Genetic Programming (GP) and program analysis were used by Weimer et al. (2010) in order to repair off the shelf legacy systems. The GP takes as input the source code to be repaired, the negative test cases that exercise the fault as well as several positive test cases that result in the correct behavior of the system. It then evolves a modified candidate repair that does not fail the negative test cases and still passes the positive test cases. The first candidate that passes all the negative and positive test cases is called the primary repair, which is reduced to the minimized repair after the use of program analysis to get rid of the irrelevant changes. Using bugs that already existed in the systems and were not manually injected provided better conclusions and the study was the first to work on real programs with real faults.

3 TECHNICAL BACKGROUND

3.1 Mutation Testing (MT)

As previously mentioned, the general idea behind MT is that the faults introduced by mutation are similar to common programming errors. MT is proven effective in finding a satisfying number of test cases, which can be used to identify real faults made by programmers (Hamlet, 1977). The number of possible faults is quite large, thus traditional MT targets only those groups of faults which are closer to the original code. This theory is based on two hypotheses: Competent Programmer Hypothesis (CPH) and Coupling Effect (CE). CPH states that programmers tend to write nearly correct code, while CE states that test data used to identify simple faults is sensitive enough to identify complex errors as well (Offut, 1989). Although there are some recent studies in literature that deal with high order mutations, like Harman et al. (2011) and Fraser and Zeller (2011), this paper focuses only on first order mutants as these may be considered good enough for performing adequate testing of program code: Simple faults may be represented with simple mutations created with syntactic changes, while complex faults are being represented with more complex mutations consisting of more than a single change in the code.

Traditional MT states that from a program P we get a set of faulty programs (mutants) after applying

some single syntactic changes on the original code. The transformation rule that generates a mutant of the original program is called Mutation or Mutant Operator (MO). MOs are designed to modify variables and expressions by replacement, insertion or deletion (Offutt and Untch, 2001). Before starting mutation analysis, the original program must be executed with the test cases to check if it is executed correctly. Afterwards the set of test cases will be used to check all the mutants created. If the execution results of a mutant are different compared to those of the original program then we may say that the mutant has been killed, otherwise we say that it has survived. After execution of all test cases some mutants may survive, so we need to provide some additional test data to kill those as well. In the end, after these additional executions, some mutants may still have survived as they keep returning the same results as the original program. These are called Equivalent Mutants and while they are syntactically different from the original program they provide the same functionality.

Testing with mutation methods is completed with calculating the adequacy score, known also as Mutation Score (MS), which defines the quality of the set of test cases given as input to the program. MS is the analogy of the number of "killed" mutants against the number of the non-equivalent mutants. The purpose of mutation analysis is to increase the mutation score, bringing it closer to 1, which means that the set of test cases is adequate for detecting all the faults included in the mutants.

3.2 MuJava and Mutation Operators

A variety of different tools for MT have been suggested: Mothra (King and Offutt, 1991) is a versatile environment for FORTRAN, which is the first full MT software tool. Jester is a simple open source tool for Java code MT (Binkley et al., 2006) that is integrated with JUnit, a well-known code testing environment. Jester does not use any sophisticated algorithm to accelerate the mutation process thus resulting in slow performance, while the number of MO supported is restricted. These limitations make the use of Jester ineffective and non-practical for large programs. MuJava (Offutt et al., 2005) is another system for MT of Java programs. Its primary objective is to study MO relevant to Object Oriented programming languages. At present it is regarded as one of the most complete tools in terms of MO supported. It offers mutations for traditional testing, but also for testing at the class level, by combining two basic technologies, Mutant Schemata Generation (MSG) and byte-code translation. Using MSG it creates "meta-mutants" of the program in at source code level that integrate a number of mutations. Working directly on the byte-code means that only two compilations are needed, the one of the original program and the one of the meta-mutants that MSG created. This improves the performance of the tool over other mutation testing tools that compile all the mutants. Based on the above useful characteristics, we decided to utilize MuJava in our study; therefore we outline its basic features below.

MuJava uses two kinds of MOs, method-level (Seung and Offutt, 2005) and class-level operators (Wang and Roychoudhury, 2004). Method-level operators change the source code by replacing, deleting and inserting primitive operators. There are six different primitive operators: arithmetic, relational, conditional, shift, logical and assignment operators. Some of these operators consist of a number of other sub-operators (e.g. binary arithmetic and shortcuts.

The class mutation operators MuJava uses are categorized in four different sets based on the characteristics of the programming language they affect. These are Encapsulation, Inheritance, Polymorphism and Java-Specific features.

4 AUTOMATIC, EVOLUTIONARY MUTATION TESTING

The goal of the proposed approach is to offer an efficient, automatic way to define the specific line or the smaller possible set of lines responsible for a fault present in a Java program. More to that, the approach aims at suggesting the necessary correction(s) that remove the fault. The use of dynamic slicing enables isolating those lines of a program that affect a variable at a given point of interest for certain executions using specific input values. We set manually the slicing criterion in an input file which contains the source code file's name and the line with the criterion for the creation of the slice. Also, the normalized version of the initial source code is fed as input to the dynamic slicing algorithm. The number of possible corrections, though, to be performed on the slice for removing a fault is usually large thus making its manual processing hard and time consuming. Mutation testing techniques applied to the code contained in the slice may be considered as the answer removing

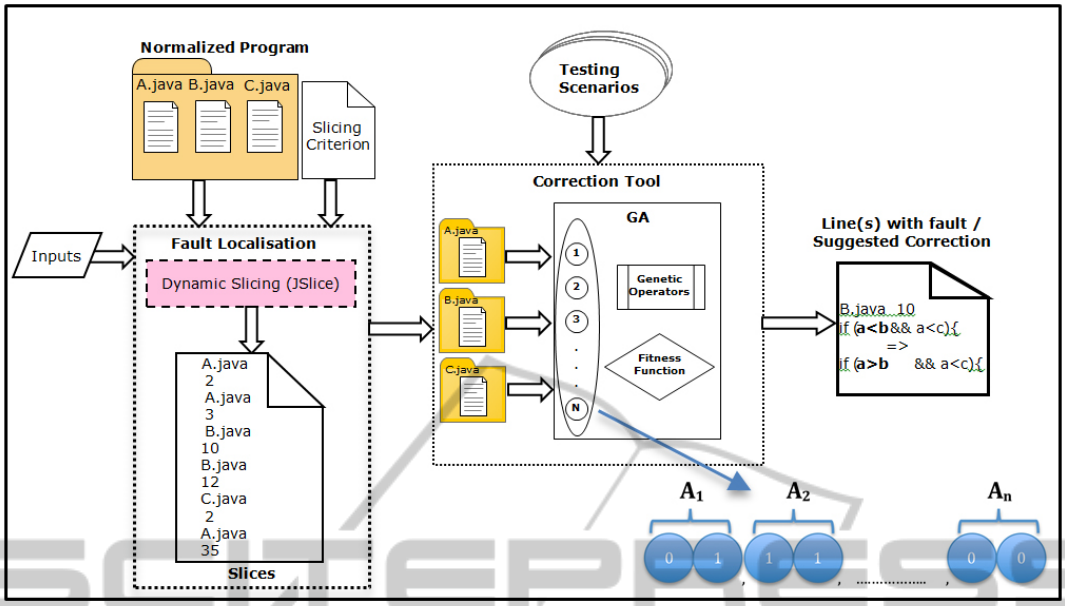

Figure 1: The proposed hybrid approach for fault localization and correction combining dynamic slicing, mutation testing and Genetic Algorithms.

a fault is usually large thus making its manual processing hard and time consuming. Mutation testing techniques applied to the code contained in the slice may be considered as the answer to this problem. Therefore, the proposed approach combines dynamic slicing with mutation testing, with the two being further enhanced by the use of genetic algorithms to evolve mutant solutions for fault correction (see Fig. 1).

Every possible solution to our problem is represented as a chromosome of size N, where N is the number of lines contained in a slice. Each line in the slice is represented as a gene that may take any value in the domain [0, K]. K is the maximum number of supported mutation operators. Any value different than zero, corresponds to a specific mutation operation that is applied to that line. The search space of the genetic algorithm based on our encoding scheme for a dynamic slice of size N is:

$$(A_1 + 1) * (A_2 + 1) * (A_3 + 1) * \ldots * (A_N + 1) \quad (1)$$

where A_x is the number of replacements for line x in the slice. As the case of no replacements is valid as well, it is necessary to add 1 to each number of replacements for each line so that the minimum size of the search space is 1 (i.e. no mutations are applied on any line).

To drive the algorithm to the best possible solution we use a Fitness Function that takes into account the results of the execution of each of the mutated programs, using a number of predefined successful and faulty test cases, that is, test scenarios that execute correctly or not respectively on the original unmodified program. Then the algorithm assesses a specific replacement based on two elements:

- The number of successful test cases that remain successful after the replacement
- The number of faulty test case scenarios that become successful after the replacement has taken place.

The probability of a line to include a fault increases proportionally to the fitness of its "best" replacement. A specific solution suggests one or more lines that contain a fault. The number of lines contained in each solution affects the fitness of that specific solution. Specifically, each suggested solution (chromosome) is evaluated using the following formula:

$$\frac{\sum_{n=1}^{L}\left(SS_n \cdot SW_n \sum_{j=1}^{S}\left(SSC_{j,n}\right) + FS_n \cdot FW_n \sum_{k=1}^{F}\left(FSC_{k,n}\right)\right)}{N \cdot SLW} \quad (2)$$

where:

SS_n and FS_n are the numbers of successful test case scenarios that remained successful, and of faulty scenarios that turned to successful respectively after replacement n,

SW_n and FW_n are weights defining the significance of the successful and faulty test case scenarios respectively,

Table 1: Mutation Operators used in our algorithm.

Method Level Mutation Operators	
AOR - Arithmetic Operator Replacement	**COD** - Conditional Operator Deletion
AOI - Arithmetic Operator Insertion	**SOR** - Shift Operator Replacement
AOD - Arithmetic Operator Deletion	**LOR** - Logical Operator Replacement
ROR - Relational Operator Replacement	**LOI** - Logical Operator Insertion
COR - Conditional Operator Replacement	**LOD** - Logical Operator Deletion
COI - Conditional Operator Insertion	
Class Level Mutation Operators	
IHD - Hiding variable deletion	**PRV** - Reference assignment with other comparable variable
IOP - Overriding method calling position change	**OAC** - Arguments of overloading method call change
ISI - super keyword insertion	**JTI** - this keyword insertion
ISD - super keyword deletion	**JTD** - this keyword deletion
IPC - Explicit call to a parent's constructor deletion	**JSI** - static modifier insertion
PNC - new method call with child class type	**JSD** - static modifier deletion
PMD - Member variable declaration with parent class type	**JID** - Member variable initialization deletion
PPD - Parameter variable declaration with child class type	**EOA** - Reference assignment and content assignment replacement
PCI - Type cast operator insertion	**EOC** - Reference comparison and content comparison replacement
Polymorphism PCC Cast type change	**EAM** - Acessor method change
PCD - Type cast operator deletion	**EMM** - Modifier method change
PCC - Cast type change	

$SSC_{j,n}$ is a constant score for a specific successful test case scenario j after replacement n (in case we want to give a specific successful scenario higher importance over the others)

$FSC_{j,n}$ is a constant score for a specific faulty test case scenario j after replacement n (in case we want to give a specific faulty scenario higher importance over the others)

SLW is a weight that reflects the importance of the slice size,

L is the number of lines contained in the proposed solution, i.e. the number of genes that were graded with a value different of 0,

S and F are the number of successful and faulty test scenarios respectively, and N the slice size.

The algorithm terminates if: (i) The predefined maximum number of generations has been reached, or, (ii) A chromosome has been evolved that yields the highest possible fitness score as expressed by eq.(2) - this is achieved when the chromosome involves only one line that contains the fault and the proposed replacement converts all faulty test cases to successful, or (iii) For the last M generations the fitness of the best chromosome becomes equal or lower compared to that of the previous generation and M exceeds the 25% of the total number of generations set.

4.1 Application Issues

A dedicated software tool was developed to support the proposed approach (screenshots are provided in Fig. 2). First, the user defines the folder that contains the source files and packages of the Java code under testing, as well as the mutation operators to be applied. The operators supported are part of those provided by the MuJava tool, specifically those that fit the purposes of this work (see Table 1).

The tool applies the mutation operators on the selected program and creates a new copy of the original file for each mutation. When the production of all code mutation cases is completed, the user sets the slicing criterion according to which the dynamic slice will be created, which is basically the line where a "faulty" result or an erroneous value of a certain variable is observed. The dynamic slice of the program is produced through a call to the JSlice tool *(*jslice.sourceforge.net). Continuing, the user executes the algorithm by defining the test cases files that contain the successful and faulty testing scenarios to be used, as well as the weights that involve three decimal numbers defining the grading weights for the mutations. Next, the definition of the initial values for the parameters of the Genetic Algorithm takes place, with the domain value for each gene being based on the maximum allowable number of mutations for the line it represents.

5 EXPERIMENTAL RESULTS

This section reports on the results of the

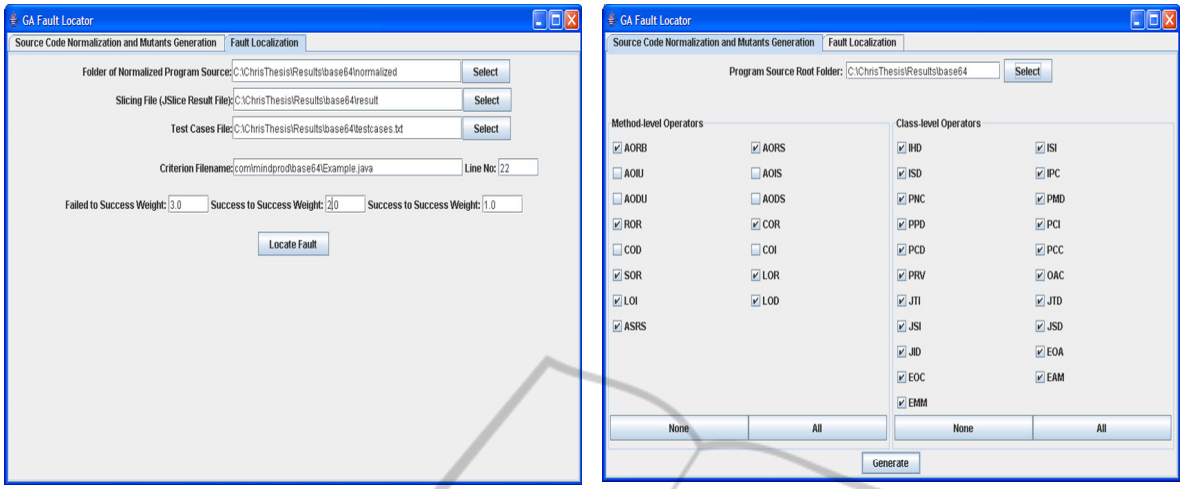

Figure 2: Screenshots of the supporting software tool.

experimental process. Three series of experiments were conducted: Category A includes mutant programs with errors at the level of methods, while category B involves errors induced at the level of classes. Category C evaluates the scalability of the proposed approach on large real-world programs. It is worth noting that the experiments were performed on a Dell Inspiron I6000 machine with Intel Pentium M processor at 1.73GHz and 2.00 GB of RAM.

The initial settings of the GA and its fitness function were as follows: For every slice FW=3.0, SW=1.0 and the slice size weight SLW=1.0. The number of generations was set equal to 200, thus calculating M to 50. Mutation and crossover probabilities were set to 0.01 and 0.3 respectively, while roulette wheel was used as the selection mechanism. Finally, equal importance was given to all scenarios, with SSC and FSC being set to the constant value of 10 for every scenario. The experimental results reported below for the former two categories are the average of 25 GA runs.

5.1 Experiment Series A'

This group corresponds to mutations performed at the method level. More specifically, single errors producing first order mutations have been induced for the following operator categories: (i) Arithmetic, (ii) Relational, (iii) Conditional, (iv) Logical and (v) Assignment. The latter was performed via a shift operation thus covering also this specific category of mutation operators.

The sample programs used in the experiments correspond to programming solutions to well-known problems that are usually treated as benchmarks; these programs are available at http://www.cut.ac.cy/staff/%20andreas.andreou/files and are briefly described below.

Credit Card Validation: This sample program reads a credit card number of x digits and returns its vendor, which may be one of the following: Visa, AMEX, Diners/Carte Blanche, JCB and MasterCard (or none of these). The program may also check the digits of the card using an algorithm that is suitable for that particular vendor so as to validate the card number.

Triangle Classification: This program, given the three sides (lengths) of a triangle, performs classification in certain categories (e.g. equilateral, isosceles, scalene etc.).

Base 64: This program receives an input string, encodes it using a 64-character set representation and finally decodes it back to string, which returns it as output.

Table 2 lists the results: the sample program, the mutation performed and the line where the error was located are indicated in the first column, while the subsequent columns include the exact statement(s) with the proposed correction and the number of testing scenarios, both successful and unsuccessful, that were used to guide the evolutionary process. The selection of these scenarios was performed automatically by a search-based module of the supporting tool that parsed the program under test and recognized the different cases that should be described through scenarios (success/fail) in order for the algorithm to identify the correct functioning. An example of testing scenarios is given in Table 3 for the Credit Card Validation program. Finally, the last column of Table 2 indicates the time for execution.

It is worth noting that in all cases the error pre-

Table 2: Results obtained using mutations performed at the method level.

Sample Program	Initial Statement Vs Mutated Statement	Testing Scenarios	Time (sec)
Credit Card Validation MO: AOD LOC: 57	cachedLastFind=i <vs> cachedLastFind=i++	2 Success 6 Fail	337
Triangle Classification MO: ROR LOC: 30	if ((i + j <= k) || (j + k <= i) || (i + k <= j)) <vs> if ((i + j == k) || (j + k <= i) || (i + k <= j))	5 Success 1 Fail	136
Credit Card Validation MO: COR LOC: 56	if (ranges[i].low<= creditCardNumber && creditCardNumber<= ranges[i].high) <vs> if(ranges[i].low<= creditCardNumber || creditCardNumber<= ranges[i].high)	7 Success 2 Fail	696
Base64 MO: LOR LOC: 147	combined |= b[i + 2] & 0xff <vs> combined |= b[i + 2] | 0xff	2 Success 1 Fail	448
Base64 MO: ASR LOC: 144	combined <<= 8 <vs> combined >>= 8	2 Success 3 Fail	191

Table 3: Sample of the testing scenarios used for the Credit Card Validation sample program.

```
\mindprod\creditcard\ValidateCreditCard.java 1
F;379999999999999;vendor_name.equals("AMEX")
S;4000000000000;vendor_name.equals("Visa")
S;4999999999998;vendor_name.equals("Visa")
S;0;vendor_name.equals("Err:no enough digits")
F;6011222233334444;vendor_name.equals("AMEX")
```

sent was successfully detected and corrected, with the time-frame of execution ranging from less than 2,5 to almost 12 minutes, depending on how difficult the error was for the algorithm to locate and provide the proper correction.

5.2 Experiment Series B'

This group includes programs that were fed with single errors corresponding to mutations at the class level. More specifically, the errors induced relate to special features of the Java programming language and also to inheritance.

The sample programs used (available at http://www.cut.ac.cy/staff/%20andreas.andreou/files) were the following:

Person Sorted List: This program receives a set of numbers corresponding to the identity card numbers of persons and sorts them in a list. An object of type Person is inserted in the list if its ID is not already part of it. Finally, it returns the number of persons in the list. The program uses inheritance for different types of persons like for example employee and student. This program was modified in line 12 of the file "Student.java", using the IHD mutation operator, which inserted variable *id* in child class Student so as to hide the corresponding variable in the parent class Person. Also, this program was used in another experiment where line 83 of the file "Person.java" was modified using operator EOC so as to replace method *equals* with the "==" operand thus providing comparison at the level of reference instead of content.

Shapes: It is a simple program that receives four input parameters (numbers) that correspond to the dimensions of a circle (first parameter – radius), a square (second parameter – side) and a rectangle (third and fourth parameters –sides A and B). The program calculates the area of each shape based on the input parameters and then it sorts the shapes using a Splay Tree according to their area. Finally, it returns the name of the shape with the maximum area. The "Circle.java" file was modified using the IOP operator and more specifically, lines 19 and 20 were interchanged so as to call the method that calculates the area of the shape prior to assigning values to variables *x1* and *y1* which correspond to the sides of the shape.

Graph-Shortest Path: This program receives a file as input which includes the description of a graph, along with the cost of each edge. More specifically, the program requires three input parameters, the first

Table 4: Results obtained using mutations performed at the class level.

Sample Program	Initial Statement vs *Mutated Statement*	Testing Scenarios	Time (sec)
Person Sorted List MO: IHD LOC: 12	`public Student( java.lang.Integer id, java.lang.String n, int ag, java.lang.String ad,java.lang.String p, double g)` <vs> *`public java.lang.Integer id = new java.lang.Integer(0);`* *`public Student( java.lang.Integer id, java.lang.String n, int ag, java.lang.String ad, java.lang.String p, double g)`* *<<LINE 12 DELETED>>*	3 Success 2 Fail	81
Shapes MO: IOP LOC: 19, 20	`super.setValues(x1,y1);` `Area();` <vs> *`Area();`* *`super.setValues(x1,y1);`*	4 Success 3 Fail	677
Graph Shortest Path MO: JTI LOC: 17	`this.cost=cost;` <vs> *`cost=cost;`*	3 Success 2 Fail	392
Graph Shortest Path MO: JSD LOC: 24	`public int scratch=0` <vs> *`public static int scratch=0`*	5 Success 3 Fail	778
Person Sorted List MO: EOC LOC: 83	`boolean equals = otherId.equals(thisId)` <vs> *`boolean equals = otherId == thisId`*	4 Success 3 Fail	95
Order Set MO: EAM LOC: 260	`int size2 = s2.getSetLast() + 1;` <vs> `int size2 = s2.getActualSize() + 1;`	1 Success 2 Fail	37

two corresponding to the start and end of a certain route respectively and the third defining the algorithm which will be used to calculate the shortest path and its corresponding cost. The program returns the sequence of nodes which constitute the shortest path and the cost of that path according to the input values. The program was modified in line 17 of the "Edge.java" file so as to include an error caused by the use of the JTI mutation operator. More specifically, the keyword "*this*" was removed during the assignment of the local variable *cost*. Therefore, the value of *cost* was assigned to the variable itself instead of the local variable *cost*. Also, this program was used in another experiment, where line 24 of the file "Vertex.java" was modified using operator JSD so as to insert the keyword "static" in the definition of variable *scratch*, thus causing all instances (objects) of class Vertex to have the same value for that variable.

Order Set: This program receives two sorted arrays as inputs and returns a new array which contains only the elements that are common between the input arrays (line 260 in file "OrderSet.java" was modified with the EAM operator by substituting the call to method *getSetLast()* with the call to method *getActualSize()*).

Table 4 presents the results of the second category of experiments. Again the number of replacements (lines) suggested by the GA was always of size 1, while in all cases the error present was successfully detected and corrected. The time of execution ranged from less than 1 to almost 13 minutes, again depending on the type of the error.

Concluding, the two series of experiments provided strong indications that the proposed approach works quite satisfactory, covering a relatively wide variety of errors, both in terms of type and complexity, locating and successfully correcting the erroneous statement in 100% of the benchmark cases.

5.3 Experiment Series C'

The last series of experiments involved programs implemented as assignments by a group of undergraduate students at the University <blinded for review purposes> enrolled in an Object Oriented Programming course using Java. The general idea here was to evaluate the proposed approach with large, real-world code with actual, not hand-seeded, faults that are accidentally made while programming.

Two projects with different functional targets

were assigned to two classes of students; the first numbered 43 students and the second 38. Therefore, a total of 81 programs were used in the experiments; the programs ranged in size from 160 to 1400 lines of code.

The first project assignment required the development of a simple library system, which would allow the user to store, borrow and delete books. The library was to be implemented using an array of Book objects. The user would be able to insert a new book from the main menu by entering relevant book details. In case the user wanted to borrow a book, the attribute Free of that Book object would be changed to false. The deletion of a book would remove the object from the Library array. Both Deletion and Borrowing required the book's ISBN number.

The second assignment was about a text-based labyrinth game. Each time the game would start with a random labyrinth of NxN size, where N was a user-defined variable entered at the start of the game execution. The labyrinth structure would be created as an array of NxN random rooms from the predefined Room objects, with each room containing traps that would affect the player when entering that room. The user would try to escape from the labyrinth by choosing which door in each room he/she should enter. The player would win the game when he/she would reach the exit.

The algorithm was executed on the first version of each student's code, i.e. after the first level of debugging and prior to final testing (proofing). We observed that the algorithm fixed approximately 80% of the errors present in the programs (an average of 3 errors in the first assignment and 5 in the second). For these faults the algorithm correctly evolved and eventually proposed the correct repair(s). What differed from the results of the previous two experiment series was the size of the resulting slices which was larger in this case. This was expected due to the fact that, contrary to the previous experiments where there was only a single hand-seeded fault in the sample programs, this series of programs contained more complicated faults affecting an increased number of lines of code.

The rest 20% of the faults that remained "uncorrected" were actually not addressed at all by the tool as they resided in statements that could not be handled by the parsing module of the supporting tool, or did not fall in the categories of operators that MuJava supports. In addition, some of the faults required repairs in lines that were previously changed when repairing a former fault in top-down sequence. This fact is not currently handled by our approach so the faults remained intact (see next section for more details). Therefore, in all of the aforementioned cases we had to skip those particular faults as practically no input could be fed to the algorithm, or its execution would have resulted in regression faults.

Further analysis of the type of errors found and the repairs applied was not among the targets of this experimental series as our goal was just to show how the algorithm scales up with programs of larger size that contain "real" faults made by the programmers. In this context the algorithm performed well as it covered a large amount of the errors and yielded a reasonable size of slice where the appropriate corrections were successfully suggested irrespectively of the total size of the program under investigation.

6 DISCUSSION ON POSSIBLE THREATS TO VALIDITY

This section briefly presents and discusses some considerations:

(i) The programs used in the two first experiment series were small and contained seeded, not real faults, and therefore one may argue that this may affect the validity of the results in some way. This, actually, does not constitute a threat to the validity of the proposed approach as: (a) The programs used in series A and B were of the order of some hundreds of LOC, which are lower than some studies reported in the literature (e.g. Harman et al.; Fraser and Zeller). This size, though, is among the acceptable average sizes for classes and methods within classes. Our approach works at the level of units, therefore the overall size of the program is not so important, as the proposed algorithm will concentrate on the smaller, independent parts of the source code each time. This was exactly the process followed for the larger programs of series C, therefore we consider the proposed approach able to cope with practically any size of code. Additionally, the faults induced simulate the actual omissions or mistakes made by programmers, thus we believe that the impact of the "fakeness" of the errors used in this study is minimal. (b) A set of programs with "real" faults was also used in the experimentation which were implemented by university students at their final stage before graduating; thus, these subjects may be considered very close or similar in skills with young programmers recruited by SMEs in the software industry and consequently their faults

may be considered identical to those of professional programmers.

(ii) As stated above the faults were hand-seeded so it was easy to know what affected the normal operation of the program so as to set as the slicing criterion. Under normal circumstances this would be hard as the users would have to understand what could possibly create the fault to include it in the slicing criterion. This was true for experiment series A and B. With series C, though, it became evident that this aspect depends only on the programming (debugging) skills of developers rather than the method used. Therefore, this does not threaten the validity of our results.

(iii) The selection of the testing scenarios was performed manually, based on the type of the program and the relevant functional specifications. The test cases were chosen so as to cover the largest possible number of different paths of execution and this is an issue that deserves research in its own merit; tools that are able assess the quality of the test cases against the specifications of a program could be used in order to select the best possible set of test data automatically, and this is something we plan to pursue in the future.

(iv) One of our concerns was the fact that a change being made to a part of the code could influence other areas of the program possibly creating new (regression) faults. In this case we may identify two different scenarios: (a) In the case where the fault repaired affects a line of code that is below the changed statements as we move from top to bottom in sequence of execution then this would not really be considered a problem as the new fault would be identified at a subsequent stage in a new slice. (b) If the algorithm attempted to change a previously repaired statement that removed a formerly addressed fault then this would have gone undetected by the algorithm. At the moment what we do to handle this problem is that we create a log-file that contains all previous changes (repairs) made in the code; a controller module consults this log-file prior to exercising mutations and prevents the algorithm from attempting to make any changes to repaired statements. In this case the algorithm skips the error, puts it in a separate file with "uncorrected" faults and continues with the next fault with proper notification of the user.

7 CONCLUSIONS

Software development suffers from low product quality and high percentage of project failure. The presence of faults is one of the major factors affecting the quality of delivered products; that is why a high percentage of development time is devoted to testing. Most of the time spend in software testing is devoted on actually locating the faults in the source code instead of correcting them. It is obvious that there is a strong need to develop a highly effective method for fault detection so as to reduce the time required by the testing process and assist in increasing the quality of the code and the productivity of software developers.

In this paper we proposed a novel approach that is able to automatically detect and correct faults in Java code. The approach utilizes dynamic code slicing for localising a fault and suggests possible corrections with the use of Mutation Testing. Jslice was used for creating the slices, while MuJava was the tool adopted for applying different mutation operators selected by the user, both at method, as well as at class level. The process of fault detection and correction through statement replacement is a problem difficult to tackle, with a large solution space; thus, we resorted in using Genetic Algorithms so as to reduce it to a search optimization problem. The GA evolved a number of candidate solutions-replacements of statements that were assessed by a dedicated fitness function.

Two series of experiments with hand-seeded errors were conducted using sample programs corresponding to well-known problems that are normally used as benchmarks for testing. Series A' involved programs with method-level errors, specifically arithmetical, relational, conditional, logical and assignment errors, while series B' included programs containing class-level errors related to Java features and inheritance. The results suggested that the proposed approach works quite satisfactory, covering a wide range of errors, both in terms of type and complexity, while it always yields the smallest possible slice containing the error and suggests the correct replacement that removes the fault.

A third series of experiments was also conducted, using two different programming assignments of undergraduate students delivered in the context of an O-O Programming course. The algorithm was applied on the first version of the code after the first level of debugging and the results were really encouraging as they showed that the algorithm scaled up nicely with large programs containing "real" faults made by programmers and not hand-seeded ones.

There are quite a few research steps that may be performed based on the present work: First, we will

attempt to collect more real case examples of programs so as to have a richer close-to-reality set of results. Second, a better analysis of the time, results and performance of the students' assignments will be made, as well as assessment of the performance of the algorithm with respect to different types of faults, their number per slice and their average complexity. Third, we plan to investigate the potentials of integrating a supporting module to the existing tool that will enable a more "sophisticated" way for selecting the appropriate test case scenarios automatically based on pre- and post- conditions that express the functional specifications of a program. Fourth, we will address the problem of regression faults by attempting to provide simultaneous repairs to more than one fault. To this end we will modify our Genetic Algorithm so as to support multi-objective optimization through multithreading and parallel processing. Fifth, the tool will be upgraded to include the statements of Java that our current implementation of the parser does not support so as to cover an even larger number of errors. Finally, our future research plans also involve increasing the set of mutation operators supported, with the inclusion of more complicated replacement operators.

REFERENCES

Agrawal, H., Horgan, J.R., 1990. Dynamic program slicing. In *ACM SIGPLAN Conference on programming Language Design and Implementation*, White Plains, New York, U.S.A., ACM Press, pp. 246-256.

Agrawal, H., Horgan, J. R., London, S., Wong, W. E., 1995. Fault Localization Using Execution Slices and Dataflow Tests. In *Sixth International Symposium on Software Reliability Engineering*, Volume-Issue: 24-27, pp. 143 – 151.

Arcuri, A., Yao, X., 2008. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation IEEE World Congress on Computational Intelligence*, pp. 162-168

Binkley, D., Danicic, S., Gyimothy, T., Harman, M., Kiss, A., Korel, B., 2006. Theoretical foundations of dynamic program slicing. *Theoretical Computer Science*, pp.23-41.

Black, S., Counsell, S., Hall, T., Wernick, P., 2005. *Using Program Slicing to identify Faults in Software*. Dagstuhl, Seminar Nº 05451.

DeMillo, R. A., Lipton, R. J., Sayward, F. G., 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11(4), pp. 34-41.

Fraser, G., Zeller, A., 2011. Generating Parameterized Unit Tests. In *International Symposium on Software Testing and Analysis (ISSTA'11)*, July 17-21, Toronto, Canada.

Goldberg, D. E., 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

Gupta, N., He, H., Zhang, X., Gupta, R., 2005. Locating Faulty Code Using Failure-Inducing Chops. In *IEEE/ACM International Conference On Automated Software Engineering*, Long Beach, CA.

Hamlet, R. G., 1977. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, 3(4), pp. 279-290.

Harman, M., Jia, Y., Langdon, W. B., 2011. Strong Higher Order Mutation-Base Test Data Generation. In *ESEC/FSE'11*, September 5-9, Szeged, Hungary.

Jiang, T., Gold, N., Harman, M., Li, Z., 2008. Locating Dependence Structures Using Search Based Slicing. *Journal of Information and Software Technology*, Volume 50, No.12, pp. 1189-1209.

King, K. N., Offutt, J., 1991. A Fortran Language System for Mutation-Based Software Testing. *Software Practice and Experience*, 21(7), pp. 686-718.

Ma, Y.S., Offutt, J., 2005. Description of Method-level Mutation Operators for Java. http://cs.gmu.edu /~offutt/ mujava/mutopsMethod.pdf.

Myers, G. J., 1979. *The Art of Software Testing*. Wiley-Interscience.

Nica, M., Nica, S., Wotawa, F., 2012. On the use of mutations and testing for debugging. *Software-Practice and Experience*, Article published online.

Offutt, A. J., 1989. The Coupling Effect: Fact or Fiction. In *ACM SIGSOFT '89 - Third symposium on Software testing, analysis, and verification* ACM New York, NY, USA ACM.

Offutt, A. J., Untch, R. H., 2001. Mutation 2000: Uniting the Orthogonal. In *1st Workshop on Mutation Analysis (MUTATION'00)*, San Jose, California, pp.34-44.

Offutt, J., Ma, Y. S., Kwon, Y. R., 2005. MuJava : An Automated Class Mutation System. Software Testing, *Verification and Reliability*, vol. 15, pp. 97-133.

Patton, R., 2006. Software Testing, *Sams Publishing*, 2nd edition.

Wang, T., Roychoudhury, A., 2004. Using compressed bytecode traces for slicing Java programs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 512-521.

Weimer, W., Forrest, S., Le Goues, C., Nguyen, T., 2010. Automatic program repair with evolutionary computation. *Communications of the ACM Research Highlight* 53:5 pp. 109-116.

Zhang, X., He, H., Gupta, N., Gupta, R., 2005. Experimental Evaluation of Using Dynamic Slices for Fault Location. In *Sixth International Symposium on Automated and Analysis-Driven Debugging*, Monterey, California.