

Testing Object-Oriented Code Through a Specifications-Based Mutation Engine

Pantelis Stylianos Yiasemis

Department of Computer Engineering and Informatics,
Cyprus University of Technology
Limassol, Cyprus
email: pantelis.yiasemis@cut.ac.cy

Andreas S. Andreou

Department of Computer Engineering and Informatics,
Cyprus University of Technology
Limassol, Cyprus
email: andreas.andreou@cut.ac.cy

Abstract—This paper presents a simple, yet efficient and effective mutation engine that can produce mutations of object-oriented source code written in the C# and Visual Basic languages as an extension of previous work on the topic [1]. The engine produces mutants based on user selected mutation operators the number of which is bounded by the specifications declared in the source code with the aid of Code Contracts. The specifications are described using a set of pre- and post-conditions and invariants. The engine consists of four distinct and integrated components; a syntactic verification component, a static analysis component, a mutation generation component, and a test case quality assessment component. A series of experiments are conducted which show that the proposed engine is able to locate a fault and efficiently propose the proper correction. In addition, the scalability of the proposed approach is assessed in terms of time and performance with respect to different program sizes.

Keywords—mutation testing; mutation engine; specifications;

I. INTRODUCTION

The rapid evolution of technology has lead to the creation of a large variety of tools that automate a number of activities within the process of software development. Computing power increases in almost exponential rates, a fact that supports the development of better and faster software systems, which, in turn, exercises pressure on their reliability as typically these systems become increasingly more complex. The competition that exists between software development companies pushes them to increase their productivity by developing the software in tighter time frames having a direct effect on the quality of the software developed.

Software Testing efficiency, or better the lack of it, is one of the most important reasons for inadequate quality control in today's software development. Software testing is a way of making sure that a software system meets its specifications while being correct and appropriate ([2], [3]). Software testing is a quite complex process that needs to be correctly performed; it thus consumes a large percentage of the time and budget of the whole development process. In some occasions it even surpasses the time and budget needed for the creation of the software product [4]. Its main purpose is the improvement of the functional behavior of a system under development, by revealing and locating faults in source code.

The software testing process comprises two main activities, the correct identification of faults and their correction (debugging). Faults can be incorrect steps or data definitions in a program that when executed together lead to

failure. Such faults are also called errors, anomalies, inconsistencies or bugs [5]. Identifying faults is more time consuming than correcting faults in software testing. This leads to the conclusion that there is a constant need for developing tools that will aid the acceleration, correctness and automation of the testing process, by guiding developers to locate and correct faults more efficiently and effectively.

The aim of this paper is to introduce a mutation engine for source code written in two popular object-oriented programming languages, namely C# and VB. Mutations are replacements of code statements performed through certain operators that correspond to specific types of errors. These replacements produce the so-called mutant programs which are then used in order to assess the quality of a test case set as regards to its ability to identify faults in code. The proposed engine constitutes the backbone of a novel mutation testing technique that takes into consideration the specifications of the program for creating only valid mutants. The engine is implemented in Visual Studio 2010 and consists of four components: The first offers the ability to validate the grammatical correctness of the source code; the second provides a form of static analysis for exporting useful information that can be used to process/modify the source code; the third involves the production of mutations of the original source code, while the fourth facilitates the identification of faults, as well as the assessment of the quality of test data.

This work constitutes an extension of previous work on the topic [1], which introduced a Mutation Engine for C# making use of Code Contracts to limit the number of produced mutants thus decreasing the time needed for fault localization. The new ground investigated in this paper may be summarised to the following: (a) The Mutation Engine has been extended to work with code written not only in C# but in VB as well. (b) The assessment of the resulted mutants and the correct identification and correction of faults for a series of examples are performed for both programming languages with comparisons between their results. (c) The engine was tested against larger versions of code and the time performance for locating the faults and producing all the mutants was assessed for multiple program sizes, with the lines of source code being increased by two, four, six and eight times respectively.

The rest of the paper is structured as follows: Section II describes briefly the basic concepts that form the necessary technical background of this work. Section III presents the mutation engine, its architecture and key elements ruling the

generation of mutations, along with a brief demonstration of the supporting software tool. Section IV describes a set of experiments and the corresponding results that indicate the correctness and efficiency of the proposed approach. Finally, Section V concludes the paper and suggests some steps for future work.

II. TECHNICAL BACKGROUND

According to Khan [6], there are three kinds of software testing techniques. These are White Box Testing (WBT), Black Box Testing (BBT) and the mixing of the two called Gray Box Testing (GBT). Each techniques offers its own advantages and disadvantages, differing in the way test cases are created and executed. In BBT the test cases are created based on the functions and specifications of the system under testing without the need for actual knowledge of the source code. WBT requires that the tester has full access to the source code and knows exactly the way it works. An advantage of this method is the ability to locate coincidental correctness, that is, the case where the final result is correct but the way it is calculated is not. Furthermore, all possible paths of code execution may potentially be tested offering the means to identify errors or/and locate parts of dead code, that is, parts that are actually never executed. GBT combines the testing methodology of WBT and BBT, meaning that it tests a system against the specifications defined but also it uses information from the source code to create the test cases. It needs more knowledge of the internals of a system than BBT but less than WBT.

Different techniques have been proposed for WBT [6] making use of the structure of the source code or the sequence of execution, giving birth to static code analysis for the former and dynamic testing for the latter. This paper concentrates on dynamic testing where the actual flow of execution drives test data production. One such technique that has gained serious interest among the research community is Mutation Testing (MT).

Various research studies propose Mutation Testing as the basic element of their approach to software testing (e.g., [7],[8]). MT is a relatively new technique introduced by DeMillo et al. [9] and Hamlet [10], which is based on the replacement of code statements through certain operators that correspond to specific types of errors, producing the so-called mutant programs. These programs are then used to assist in producing or/and assessing the quality of test data as regards revealing the errors in the mutants [11].

The general idea behind MT is that the faults being injected correspond to common errors made by programmers. The mutants are slightly altered versions of programs which are very close to their correct form. Each fault is actually a single change of the initial version of the program. The quality of a produced set of test cases is assessed by executing all the mutants and checking whether the injected faults have been detected by the set or not. This assessment is based on a Mutation Score (MS), which is the ratio of “killed” mutants against the non-equivalent mutants. The purpose of mutation analysis is to aid in creating a test case set of high quality, that is, a set able to produce a MS closer to 1. Such a test can be used to detect all the faults that may exist in the code.

It is possible to produce a large number of variations of a program and the faults that may contain, thus traditional MT targets only groups of faults that are closer to the original version of the code. This practice is based on the Competent Programmer Hypothesis (CPH) and the Coupling Effect (CE). The CPH states that the code written by programmers is almost correct. CE states that when identifying simple faults with a set of test data, the same test data can also identify larger and more complex faults [12]. While recent work in the field of MT deals with high order mutations [11] [13], this paper targets only on first order mutants (simple mutants) as these may be considered good enough, based on CPH and CE, for performing adequate testing of program code. Complex faults are represented by complex mutations consisting of more than one change in the code, whilst simple faults are represented by a single mutation (syntactic change) to a program.

There are a number of ways to represent program code. Each provides a particular way to understand a program and manage its source code. Most of them use graphs or/and binary trees that are able to depict graphically how the program actually works. The Control Flow Graph (CFG) is one such way of graphically representing the possible execution paths. Each of the nodes in a CFG corresponds to a single line of program source code. The arcs connecting nodes represent the flow of execution. A CFG may be used as the cornerstone of static analysis, where its construction and traversing offers the ability to identify and store information about the type of statements present in the source code and the details concerning the alternative courses of execution. A fine example is the BPAS framework introduced by Sofokleous and Andreou [14] for automatically testing Java programs. A CFG may also drive the generation of test data by providing the means to construct an objective function for optimization algorithms to satisfy (algorithms by evolution such as the one proposed in Michael et al. [15]).

The Visual Studio (VS) platform [16] has been constantly evolving becoming one of the most widely used platforms today in the software industry. This is partly due to the fact that it provides the ability to create a number of different types of applications, like window-apps, web-apps, services, classes etc. The wide acceptance of VS has driven the development of a number of third party tools and plug-ins that enhance the platform with even more functionality, making development of special-purpose applications simpler and easier. The aforementioned advantages of VS2010 suggest that its use might be quite beneficiary for software testing, and more specifically for developing a new mutation testing tool.

Code Contracts (CC) were introduced by VS2010 as a means to encode specifications [17], but can be installed on other versions of Visual Studio as well. CC may consist of pre-conditions, post-conditions and invariants. The aim is to improve the testing process through both runtime and static checking. Runtime checking takes place while the program executes and produces an exception when the specifications in the code are not met. Static contract verification is performed while the project is under development. It produces a warning when a condition is not satisfied and also proposes a solution to fix the relevant code. CC also assist in documentation

generation by producing an XML file with information from the CC. CC can be used on any .Net platform that contains the Contracts class, or, if building a project on a platform that does not support CC (e.g., older versions of Silverlight, Windows Phone 7, etc), a reference to the assembly Microsoft.Contracts.dll should be added to the project.

Code Contracts were developed from knowledge obtained from the Spec# programming system, an attempt made by Microsoft to provide a way for more cost effective and higher quality software. SPEC# is in essence a formal language for API contracts that permits specification and reasoning of object invariants, even in parallel processing environments or when callbacks exist in the code. Using a CC enables a programmer to create a detailed set of specifications that will be used to verify a program with the use of the static program verifier. The latter checks if a program satisfies the specifications with no runtime errors. SPEC# is being developed as a research project by Microsoft Research's Programming Languages and methods group [18].

The mutation engine introduced in this paper is partly based on the aforementioned concepts. More specifically, it utilizes CFG and static analysis as in [14] to extract the information needed for analyzing and describing adequately the source code under investigation. Moreover, it employs CC to embed the specifications required in order to assess whether a program functions properly. The mutation engine utilizes runtime checking to limit the production of meaningful mutant programs, that is, programs that do not violate their original specifications. Lastly, it incorporates an automatic test case assessment module that either evaluates the quality of a given test case set, or identifies faults in the original code, and proposes the proper correction that also satisfies the specifications. The engine offers a means for both the automation of software testing and a reduction in the time required for software testing.

III. MUTATION ENGINE

The mutation engine is implemented in the VS2010 platform. VS2010 was selected partly because it is a relatively newly introduced platform, meaning that the components developed may be used as the backbone for future tools and studies based on this platform, without facing any incompatibility issues compared to the use of older platforms. Also, to the best of our knowledge, at present no other such system exists. The engine was originally designed to work with the C# programming language. A number of additions and enhancements were introduced to the engine for supporting Visual Basic as an extension to our previous work [5].

A. Research Strategy

The first step to design and implement the mutation engine was the selection of particular technologies to be incorporated in the proposed tool. Therefore, CFG were employed to aid in the static analysis and CC were chosen to limit the number of produced mutants.

CFG were chosen to represent the code as they give information about the flow of execution, but most importantly they identify and store information about the

type of statements present in the code and additional details regarding alternative courses of execution. This information can then be used for static analysis of the code, which is a preprocessing stage that enables the gathering of critical information as regards specific parameters of the program under testing. This assists in the application of the mutation operators on the source code as it identifies everything present in the code (variables, classes, statements) and the proper mutations can then be applied to each of the elements identified. CC were selected as a means to describe the conditions that exist in the specifications of the program in order to help eliminate those mutations that do not satisfy the specifications.

B. Architecture

The architecture of the proposed mutation engine is depicted graphically in Figure 1 where four major components enable the execution of the engine's stages.

1) Syntactic verification component

The first is a source code validation component, which compiles the source code and presents any erroneous lines. This component takes as input a source code file (.cs or .vb), or an executable file (.exe), or a dynamic link library file (.dll), as well as the project file (.csproj or .vbproj). The project file provides the validation component with information for references in libraries and files that the source code uses and are part of the program. Validation includes compiling the source code and making sure that no syntactic or other compilation errors exist so as to proceed with the second stage of the engine which is the production of mutations. Otherwise the engine terminates.

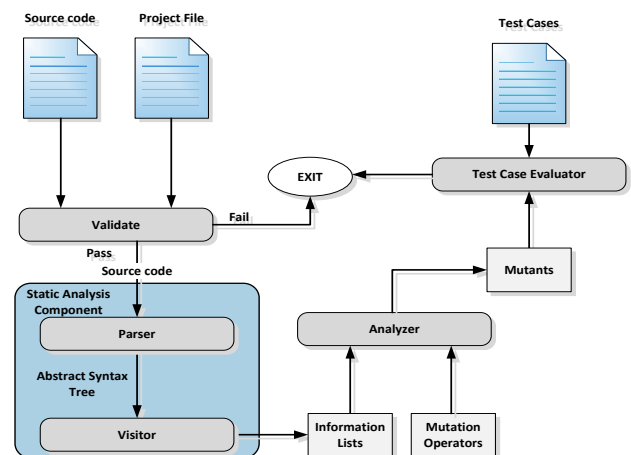


Figure 1. The mutation engine architecture

2) Static analysis component

The second component performs static analysis of the source code without the need of an executable form of the program under testing. Static analysis is the extraction of useful information from the source code concerning the structure of the program. This component takes as input the source code file and uses the class `AbstractSourceTree` (AST) of SharpDevelop [19] to model the abstract syntax tree of the code. While compiling a source code file, a binary tree (the

AST) is created, each node of which represents a line of code. Traversing this binary tree, offers access to any part of the source code.

The static analysis component described above consists of two sub-components, the *Parser* and the *Visitor*. The *Parser* analyses the source code and creates the AST as mentioned earlier. The *Visitor* passes through the AST collecting useful information, while giving the opportunity for the user to make changes and additions to the information stored. The implementation of the *Visitor* utilised the *AbstractAstVisitor* class of SharpDevelop, with some minor additions to help accessing all the nodes of the AST, both at the high (classes and their parameters, inheritance, etc.) and the low level (assignments, conditional statements, unary statements, etc.) characteristics of the programming language. The *Visitor* recursively visits each node and stores in stack-form lists all the information identified according to the node's type. In the experiments described in the next section thirteen such lists were created; nevertheless, the way the *Visitor* is structured enables the addition of any new lists or the modification of existing ones in a quite easy and straightforward manner.

3) Mutation generation component

The third component is the heart of the mutation engine. This component analyses the information stored in the lists created by the *Visitor* so as to identify the structure and content of the source code, and creates mutated programs by applying a number of predefined operators to the initial program. These mutators are responsible for creating a number of different variations of the initial source code. Each mutation is based on one or more grammatical rules that do not breach the grammatical correctness of the resulting program.

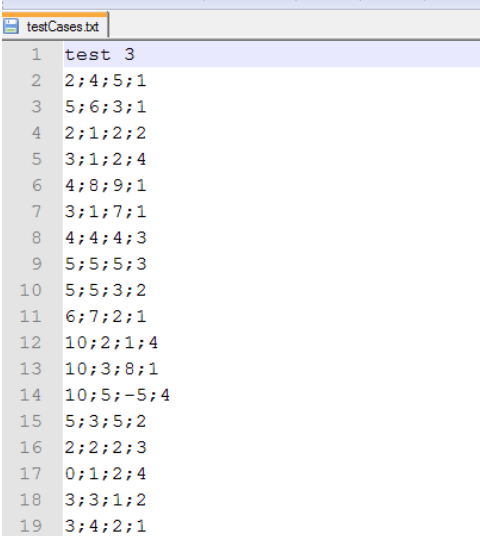
4) Test case quality assessment component

The final component of the mutation engine is the automatic test case quality assessment and fault detection component. It takes as input a text file containing the test cases. The test case file includes a header containing the name of the function to be called and the number of arguments that are needed as input, while the rest of the file contains the test cases values and the expected results for each set of inputs. Figure 2 presents an example of a test case file. The function that is going to be tested is called `test` and takes 3 parameters, as seen in the header of the file. The rest of the file contains, for each test case, the input values and expected results separated by semi-columns. The test case evaluator then loads all the test cases found in the test case input file and applies them to the original program.

If the program returns the expected values for each of the test cases, the tool continues applying each mutation to the original code and evaluating the results of each test case.

When all the test cases have been applied to all the mutants, the engine calculates the Mutation Score, along with information about which mutants were detected, which were not, the mutants that could not be compiled and some other run-time information. In the case where the initial source code fails to give the expected output for a specific test case, the tool tries to locate a possible solution by finding a mutant that gives the expected results for all the test cases defined in the file. This mutation is then logged as a possible valid

correction for the fault, while the engine continues to look for more possible solutions until all the mutation operators have been applied. The results are logged in .txt files containing useful information about which test cases managed to detect errors, which mutations were identified, the possible corrections for the initial code, etc.



```

1 test 3
2 2;4;5;1
3 5;6;3;1
4 2;1;2;2
5 3;1;2;4
6 4;8;9;1
7 3;1;7;1
8 4;4;4;3
9 5;5;5;3
10 5;5;3;2
11 6;7;2;1
12 10;2;1;4
13 10;3;8;1
14 10;5;-5;4
15 5;3;5;2
16 2;2;2;3
17 0;1;2;4
18 3;3;1;2
19 3;4;2;1

```

Figure 2. Format of the Test Cases File

C. Supported Mutation Operators

Mutations are performed at the method level using operators that are either arithmetic, relational or logical. At the class level, mutation is performed with operators applied to a class or a number of classes, and usually involves changing calls to methods or changing the access modifiers of the class characteristics (public, private, friendly etc.). The operators supported by the proposed mutation engine are the:

Arithmetic

- AOR_{BA} – arithmetic operations replacement (binary, assignment)
- AOR_S – arithmetic operations replacement (shortcut)
- AOI_S – arithmetic operations insertion (shortcut)
- AOI_U – arithmetic operations insertion (unary)
- AOI_A – arithmetic operations insertion (assignment)
- AOD_S – arithmetic operations deletion (shortcut)
- AOD_U – arithmetic operations deletion (unary)
- AOD_A – arithmetic operations deletion (assignment)

Relational

- ROR – relational operations replacement

Conditional

- COR – conditional operations replacement
- COI – conditional operations insertion
- COD – conditional operations deletion

Logical

- LOR – logical operations replacement
- LOI – logical operations insertion
- LOI_A – logical operations insertion (assignment)

- LOD – logical operations deletion
- LOD_A – logical operations deletion (assignment)

Shift

- SOR – shift operations replacement
- SOI_A – shift operations insertion (assignment)
- SOD_A – shift operations deletion (assignment)

Replacement

- PR – parameter replacement
- LVR – local variable replacement

For example if the AOR_{BA} operator is applied on the following line of code

```
jkreturn this.num / this.den;
```

then the result will be the creation of four different mutations by replacing the division (/) operator with either addition (+), multiplication (*), subtraction (-) and modulo (%). The four cases for the produced mutated line of code are shown below:

- (a) return this.num * this.den;
- (b) return this.num + this.den;
- (c) return this.num - this.den;
- (d) return this.num % this.den;

D. Specification-Based Mutations

The number of possible mutated programs for a certain case-study may be quite large depending on the type and number of statements in the source code. Mutations processing time is proportional to the number of mutants processed. This is a significant problem that may hinder the use of mutation testing in certain cases. There is a need to minimize mutation testing execution time. This is feasible if useless mutations are removed or avoided. Such mutations correspond to invalid forms of executions for that particular program which may be determined by the program's specifications. Therefore, the specifications must be taken into consideration when producing a mutant. These Specifications are implemented as CC in VS2010. This feature enhances the fault detection part of our tool, as it removes any possible mutations that do not satisfy the specifications defined in the source code.

```
public class Test {
    private int Foo(int a, int b) {
        Contract.Requires(a > b);
        Contract.Requires(b > 0);
        Contract.Ensures(Contract.Result<int>()>0;
        ...
        return (a / b);
    } ...
    private void Goo( ) {
        int x, y;
        ...
        x = y + 10;
        int result = Foo ( x , y) }
}
```

Figure 3. Class Test Example with CC specifications

Figure 3 demonstrates how mutations are driven by the specifications inserted via CC. Class *Test* includes methods *Foo* and *Goo* and uses CC to express two pre-conditions (denoted by *Contract.Requires*) and one post-condition (denoted by *Contract.Ensures*).

In *Goo* the assignment of *x* affects the values with which *Foo* is called. The first pre-condition requires that $x > y$. The engine normally would perform operation replacement substituting '+' with '-', '/', '%' and '*'. Due to the aforementioned pre-condition the engine will drop the first three replacements and use only the last one as it is the only replacement that will still satisfy the pre-condition. The same applies for $b > 0$, where any arithmetic replacement should not set *b* equal or less than zero. In this way the engine produces only valid mutations and ensures that a certain mutation is implemented in the engine which enables the production only of valid mutants thus ensuring that the minimum possible time and effort will be spent on the subsequent analysis and testing activities. This approach also limits the search for a possible solution by the user, when a number of solutions are identified by the engine.

E. The software tool

A dedicated software tool has been developed to support the process of MT. An example scenario is given below to demonstrate its operation: A source code file, the project file of the program tested and a test case file (optional) are given as input to the system. If no test case file is provided the program continues with only the creation of the mutations and nothing more. The project file and all the references to other files or libraries are automatically located and linked, and the source code file is compiled through the validation component. In the case of compilation errors a pop up window is presented to the user with the corresponding information (Figure 4) and the process is terminated.

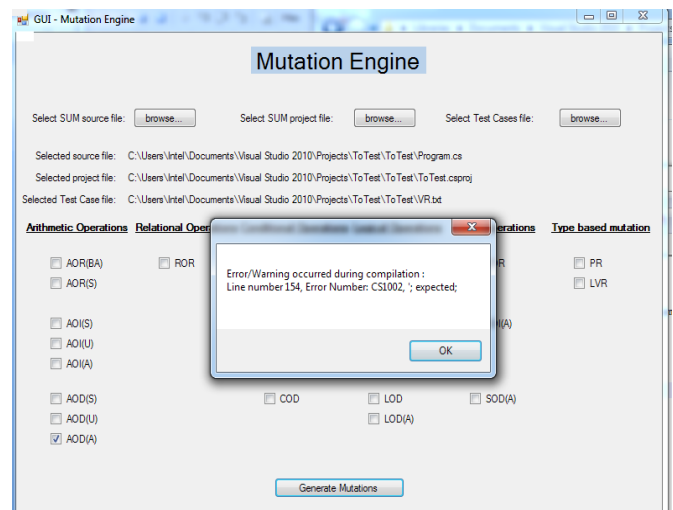


Figure 4. Execution : Errors in compilation

If there are just warnings, the user is again informed, but the system now continues to the next step. Static analysis of

the source code is performed, resulting in the creation of an AST. The visitor component then passes through the AST and creates the lists that store the information found in the source code (variables, classes, statements, etc.). The third component takes as input the lists created by the visitor and a set of mutators selected by the user, applies these operators and returns the resulting mutated programs in the path defined (Figure 5). The last component, the automatic test case quality assessment component, reads the test case file provided by the user and executes the initial program with those inputs. If the execution fails on any test case, the user is notified that the original program does not validate all the test cases correctly compared to their expected results and then searches for a mutated version that does. If the initial program validates correctly all test cases, then it continues with assessing the quality of the test data set in order to report the ability of the test data set to identify faults in the mutants.

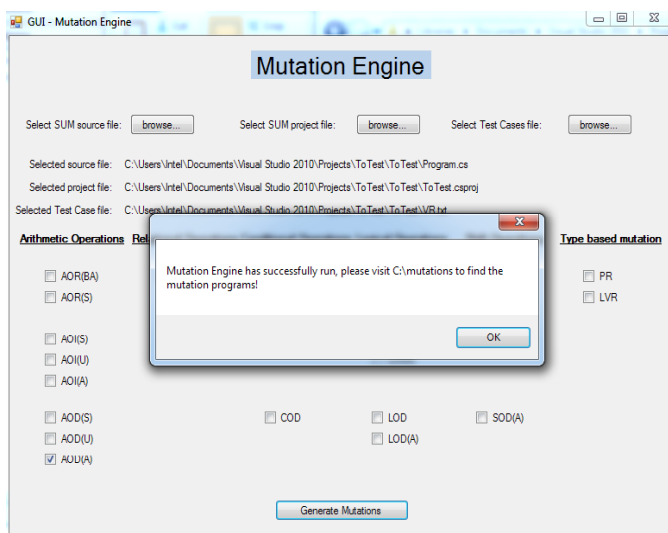


Figure 5. Execution : Mutations successfully produced

IV. EXPERIMENTAL RESULTS

In order to test the mutation engine and the corresponding trial a series of experiments were performed that would help us assess both the correctness and the efficiency of our testing approach using programs written in both the C# and VB programming languages. Four categories of experiment were conducted as follows:

Category A addressed the quality (adequacy) of a given test case set against a benchmark program. A file containing test cases and the expected results was fed into the tool along with the source code that verified all of the test cases. Then all the mutations produced by the engine were tested against the test cases, logging which mutants were discovered or which ones produced the same result as the original program.

Category B assessed the ability of the tool to discover a fault in the original code and provide a solution to correct it. In the case a test cases set fed into the tool does not validate the original program the tool continues to create mutations and propose possible solutions in order to validate all the test cases found in the test file, whilst satisfying all the specifications found in the code contracts.

Category C demonstrated that the proposed engine works as supposed on both C# and VB code, by producing correctly a number of mutations based on atomic changes to the source code according to the user's selected types of mutation operators. The same functions are developed in both programming languages and the mutations produced were compared. Also, both the results of C# and VB mutations were tested against a set of test cases to see if the mutation engine could identify the same mutants for both languages.

Category D evaluates the scalability of the proposed approach on large, real-world programs. Benchmark programs were used, and the type and number of mutations was recorded. It is worth mentioning that the experiments were performed on an Intel i7-2600 CPU at 3.4 GHz with 4 GB of RAM, while the programs used are available in various sites on the Internet (e.g., <http://www.c-program-example.com>).

Lastly, category E demonstrated that the mutation engine could eliminate the mutants that violate the pre-conditions, post-conditions or invariants set for a program. Comparisons of the number of mutations produced when using code that contains specifications against code that does not contain specifications.

The experiments are analysed below:

A. Test-Data Quality Assessment

This experiment used a specific benchmark program, the triangle classification program, which is shown in Figure 6. This program was tested against the 19 different test cases shown in Table I, the meaning of the values used in its last column is as follows: 1 equals to a scalene triangle, 2 equals to an isosceles triangle, 3 to an equilateral and 4 does not correspond to a triangle.

```
int triang(int i, int j, int k) {
    if ((i <= 0) || (j <= 0) || (k <= 0))
        return 4;
    int tri = 0;
    if (i==j)
        tri+=1;
    if (i==k)
        tri+=2;
    if (j==k)
        tri+=3;
    if (tri==0) {
        if ((i+j==k) || (j+k==i) || (i+k==j))
            tri=4;
        else tri=1;}
    else {
        if (tri>3) tri=3;
        else {
            if ((tri==1) && (i+j>k))
                tri=2;
            else{
                if ((tri==2) && (i+k>j))
                    tri=2;
                else {
                    if ((tri==3) && (j+k>i))
                        tri = 2;
                    else tri = 4; } } } }
    return tri; } }
```

Figure 6. Triangle Classification Program Source Code

Using the values in the first three columns of Table I for the corresponding variables it appears at first that the TCP has been adequately tested. The source code, the project file and the test cases file were fed into the tool and all of the available mutation operators were selected. After the engine finished both creating the mutants and testing them with the test case set, a general results file was created (Figures 7 and 8) which contained all the mutations and the verdict whether there is at least one test case in the test set that could identify the alteration performed or not. The file also includes the number of total mutations, the mutations that failed to compile, the number of mutations that were successfully discovered or not, the mutation score and the time needed for the engine to produce and test the mutations.

TABLE I. TEST DATA THAT COVER ALL POSSIBLE OUTPUTS OF THE TRIANGLE CLASSIFICATION PROGRAM (TCP)

i	j	k	Result
2	4	5	1
5	6	3	1
2	1	2	2
3	1	2	4
4	8	9	1
3	1	7	1
4	4	4	3
5	5	5	3
5	5	3	2
6	7	2	1
10	2	1	4
10	3	8	1

10	5	-5	4
5	3	5	2
2	2	2	3
0	1	2	4
3	3	1	2
3	4	2	1

Along with the general results file created at the end of the process, a results file for each mutation is created. This file contains the results of applying each of the test cases, to the program and which test case identified the error, if such a case exists.

The results show that 517 mutants were created, from which 58 could not be compiled so they were discarded. From the remaining of 459 mutants, 175 could not be identified by the test case set as they successfully yielded an identical result as the original program. Finally, 284 mutations were identified by at least one test case, leading to a mutation score of 61%.

This experiment demonstrates that the proposed engine is able to assess the quality of a set of data to adequately test a given program.

Such a change that yields the same result is the following change on the second statement of the code in Fig. 6:

```

if ((i <= 0) || (j <= 0) || (k <= 0))
    to
if ((i <= 0) || (j < 0) || (k <= 0))
    
```

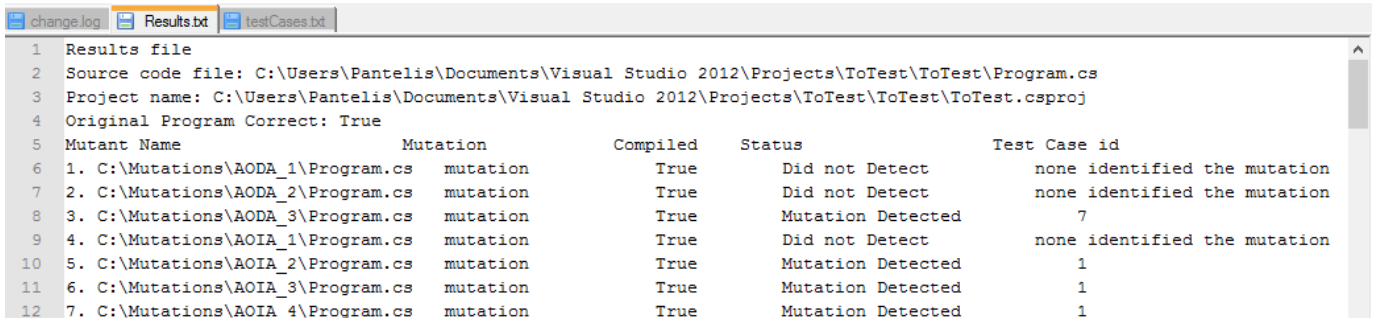


Figure 7. Beginning of general results file using Triangle Classification program written in the C# language

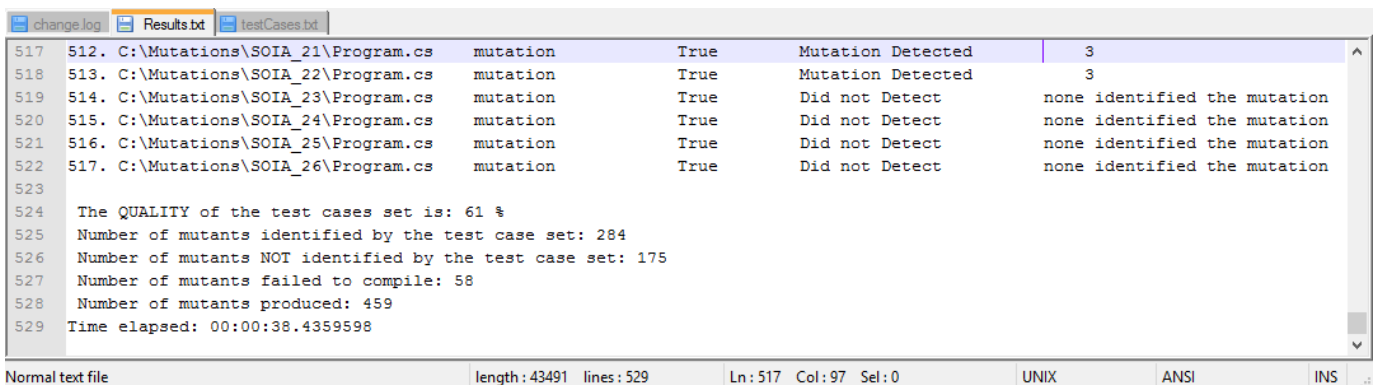


Figure 8. End of general results file using Triangle Classification program written in the C# language

B. Fault Detection

This set of experiments was concerned with the ability of the mutation engine to reveal errors that were injected in the initial source code of the triangle classification program. A number of faults were manually injected into the code and is described below.

Figure 9 shows two faults inserted in the code, one relational and one unary.

```
int triang(int i, int j, int k) {
    if ((i <= 0) || (j != 0) || (k <= 0)){
/**1. should have been ((i<=0) || (j<=0) || (k<= 0))**/
        return 4;}
    int tri = 0;
    if (i==j)
        tri+=1;
    if (i==k)
        tri+=2;
    if (j==k)
        tri+=3;
    if (tri==0) {
        if ((i+j==k) || (j+k<=i) || (i+k<=j))
            tri=4;
        else tri=1;}
    else {
        if (tri>3)
            tri+=3;
/** 2. should have been tri=3;**/
        else {
            if ((tri==1) && (i+j>k))
                tri=2;
            else{
                if ((tri==2) && (i+k>j))
                    tri=2;
                else {
                    if ((tri==3) && (j+k>i))
                        tri = 2;
                    else tri = 4; } } }
        return tri; }
}
```

Figure 9. Faults injected into Triangle Classification Program

The first change was the replacement of the \leq relational operand with \neq in the second line of the original code. The tool suggested 6 possible solutions (Table II). It's clear that correction #5 is the one that reverted the faulty program to the original version, but the other 5 proposed fixes yield the same results with number 5. This can be due to the quality of the test cases set and its inability to detect all the mutants in the first place. This means that it is possible to check only 6 out of the 615 working mutants to find a correct version. Consequently only 1% of the work is needed compared to checking all mutations for a possible correction.

The second alteration involved changing the assignment $tri=3$ to adding 3 to the variable ($tri+=3$). The engine applied all mutation operators in the original version and suggested 2 possible corrections for this case. The first was the arithmetic operations deletion (AOD_A_4) mutation, which concerned the deletion of the plus operand from the line changed. The second suggestion was the arithmetic operations

replacement (AORBA_33) that suggested the change of the + operand to $-$. This suggestion again resulted in a version that satisfied all test cases; as we can see from the code the assignment is executed when the value of tri is equal to 6, so subtracting 3 will result in tri taking the value of 3 and validating correctly the test cases.

TABLE II. PROPOSED FIXES FOR THE FIRST INJECTED FAULT

No.	Name	Proposed Fix
1	COI_2	if ((i <= 0) (!(j != 0)) (k <= 0))
2	COI_4	if ((i <= 0) !(j != 0) (k <= 0))
3	COR_1	if ((i <= 0) (j != 0) && (k <= 0))
4	COR_2	if ((i <= 0) && (j != 0) (k <= 0))
5	ROR_9	if ((i <= 0) (j <= 0) (k <= 0))
6	ROR_10	if ((i <= 0) (j == 0) (k <= 0))

The example of Figure 10 employs CC with three pre-conditions, one post-condition and one invariant, and involves two errors inserted in class `CompareParadigm` that cannot be traced by the static analyzer in VS2010.

```
class CompareParadigm {
    int num,den;

    public CompareParadigm(int numerator, int denominator){
        Contract.Requires(0 < denominator);
        Contract.Requires(0 <= numerator);
        Contract.Requires(numerator>denominator);
        this.num += numerator;
/** should have been this.num = numerator **/
        this.den = denominator;
    }
    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(this.den > 0);
        Contract.Invariant(this.num >= 0);
    }

    public int ToInt() {
        Contract.Ensures(Contract.Result<int>()>=0);
        return this.num * this.den; }
    }
/** should have been this.num / this.den **/
}
```

Figure 10. CompareParadigm Class with embedded Code Contracts

The engine is once again capable of bringing these errors to light using the arithmetic operation replacement (AOR_{BA}) and arithmetic operations deletion (AOD_A) mutators.

C. C# and VB comparative evaluation and compatibility issues

The tool has been extended to support both C# and VB. In order to assess the behavior of both C# and VB versions of the triangle classification program were used and the results compared.

The choice of analyzing only the arithmetic mutation operators was made, as they produce a large number of mutants allowing the extraction of some safe conclusions.

TABLE III. MUTATED PROGRAMS CREATED BY THE ENGINE FOR VB AND C#

Mutation type	Number of Mutations	
	Visual Basic	C#
AOD _A	3	3
AOI _A	65	65
AOI _S	66	66
AOI _U	41	41
AORB _A	36	36
Total	211	211
Failed to Compile	22	0
Identified Mutations	70	125

Table III shows that the mutation engine produces the same number of mutations for each operator in both cases of coding languages. In C# all of the mutations were compiled successfully, but when dealing with the VB source file, 22 mutations out of 211 could not be compiled. Further investigation of the mutated VB code files that could not be compiled highlighted that the mutations produced a form of syntax that is not always allowed in VB. An example of such a case is the production of the line below:

```
tri %= 2
```

VB does not support the use of the % operator, as it uses the mod operator to divide two numbers and return their remainder. After carefully checking all 22 mutations that failed to compile the observation that all of them failed because of the use of the % operator was made.

Continuing this evaluation, a comparison between the results files of the C# and VB versions of the code revealed that all of the identified mutants of the VB version were included in the C# mutants as well. Focusing on cases that were identified in C# but not in VB all of them were cases where the ++ and -- operators were introduced before a variable, as for example:

```
if ( ++tri = 1),
```

or cases with the += and -= operators being introduced as in:

```
tri -= 2
```

The use of these four operators is something that VB's compiler does not report either as a warning or an error; therefore the corresponding statements are compiled correctly, but they have no meaning and functionality. Because these statements are ignored, the mutated program yields the same behavior as the original version.

All 55 mutations that were not identified by the engine were mutations that used the four operators. This is one of the main compatibility issues raised in the extension of the proposed engine and will be addressed in future work possibly by removing these operators when dealing with VB source code.

Further investigation of the implications of the VB support took place by assessing test cases for a sample program based on the Find Max function; the program takes as input four numbers and returns the largest one (Figure 11). This would

further validate the mutation engine's support for locating and correcting faults in VB programs.

```
Dim max As Integer = 0
If num1 > num2 Then
    max = num1
Else
    max = num2
End If
If max < num3 Then
    max = num3
    If max < num4 Then
        max = num3
    End If
Else
    If max < num4 Then
        max = num4
    End If
End If
```

Figure 11. FindMax Program implemented in VB Programming Language

The evaluation used a test case file that described 20 cases with their expected results. An excerpt of the file can be seen in Table IV.

TABLE IV. PART OF THE FINDMAX PROGRAM TEST CASES SET

Num1	Num2	Num3	Num4	Result
5	6	7	8	8
5	6	3	1	6
4	8	9	1	9
-9	-4	-2	-1	-1
3	4	2	1	4

The tool was executed and the selection of all of the available mutation operators to be applied on the source code was made. This resulted in the production of a total of 336 mutations, from which 84 failed to compile due to the reasons described previously. From the remaining 252 produced mutants, 130 were identified by at least one test case, while 122 were not, something that computes a mutation score of 51% (Table V), indicating that the engine was able to detect 51% of the produced mutations with the test cases set fed.

TABLE V. MUTATED PROGRAMS CREATED BY THE ENGINE FOR VB IMPLEMENTATION OF FINDMAX FUNCTION

Mutation type	Number of Mutations
	Visual Basic
AOI _A	65
AOI _S	28
AOI _U	14
COI	4
LOI	14
LVR	96
PR	69
ROR	20
SOI _A	26
Total	336
Failed to Compile	84
Identified Mutations	130

In order to complete the conclusions of the review of the VB support the fault locating part of the tool was further investigated. For doing so the next line of code was changed as shown below:

```
from   If max < num3 Then
to     If max > num3 Then
```

Again the same test cases set was fed into the engine, as before, and the results of the Mutation Engine proposed two Relational Operator Replacements that could possibly fix the fault. These were:

```
(i) If max < num3 Then
(ii) If max <= num3 Then
```

The first replacement brings the program to its original state (i.e., before injecting the fault), while the second one again yields the same results as the first, as it does not affect the rest of the code in a way that alters the results for any of the test cases. As seen in bold letters in Figure 11, in either of the two cases the result would be that variable `max` gets the contents of the `num3` variable.

In general, the extension of the engine with the VB support module, although presenting some compatibility issues to resolve in the future, provided some encouraging results showing that the proposed engine is quite useful for testing source code written in VB exhibiting comparable performance to that when using C# code.

D. Time Behavioral Analysis

The fourth category of experiments, involved time analysis and measurements on differently sized C# programs. To this end, replication of the code of the Triangle Classification program was decided, by 2, 4, 6 and 8 times producing double the size of the program in each case. The test case quality assessment module of the tool was used with the same test cases shown in Table I. The `Stopwatch` class of the `System.Diagnostics` library was used to measure the time needed to produce the mutations.

TABLE VI. BENCHMARKS ON C# CODE

Lines Of Code	Mutations	Time (seconds)
67	468	46
134	905	77
268	1677	157
402	2480	216
536	3290	299

Table VI indicates that the time and number of mutations increases almost linearly and proportionally to the number of lines of code. An apparent analogy exists between the three values: doubling the lines of code nearly doubles both the number of mutations and hence the time needed for the engine to create them, as well as to test them. Notably, approximately 50% of the mutants corresponded to the number of failed to detect mutations, which were executed (tried) at least 20 times each, while the rest of the “normal” mutations were run a

variable number of times, ranging from 1 to 20. This emphasizes the importance of controlling the number of “useless” mutants addressed by the proposed mutation engine via the specification-driven mutation production and evaluation, as explained in the next section. For example, in the first case shown on Table VI where 67 lines of code exist in the source file for which 468 mutations were produced, it took the mutation engine 46 seconds to generate and test the mutants. This is roughly 0.1 seconds spent on producing and testing each mutant. If a programmer would have to create manually the mutants and evaluate them against the test cases, he would have needed at least 2 minutes to make each change, compile the code and run it against all the test cases. Also, he would have to document the results and keep track of all the mutators applied, something which would have taken extra time as well. The benefits of the automatic tool against the manual creation and evaluation of the mutants are clear and significant in terms of the time and effort needed.

In summary, the proposed solution was successfully tested on a large number of automatically created errors injected in the code against a number of test cases, reporting the mutants that identified (or not) each error in a reasonable time span. The time was less than the time needed for manually creating modified versions of the initial code and testing them one by one using the test cases.

E. Normal vs Specifications-Based Mutations Production

The fifth category of experiments involved the use of the CC. Using CC the tool can eliminate the mutants that violate the pre-conditions, post-conditions or invariants set for a program.

First, class `CompareParadigm` listed earlier, which includes a number of code contracts, was selected for experimentation. The number of mutations produced with the use of the specifications was compared to that of the same class with no specifications defined in code (in this case the engine with the CC disabled). Table VII lists the number of mutations that were produced according to the mutation operator used. A 58% reduction in the number of mutants is achieved when using the code contracts version of the code, which resulted in the engine generating 16 mutants compared to 38 that were produced without taking into consideration the specifications.

TABLE VII. MUTATED PROGRAMS CREATED BY THE ENGINE WITH (SPECS-BASED) AND WITHOUT THE USE OF SPECIFICATIONS (NORMAL)

Operator	Number of Mutations	
	Specs-based	Normal
AOR _{BA}	5	8
AOI _S	7	10
AOI _U	0	6
LOI	2	6
PR	2	3
LVR	0	5
Total	16	38

This reduction is quite significant, as the code consisted of less than 20 statements. Therefore, one can safely argue that in cases of large programs the computational burden will be considerably eased, preserving the effectiveness and efficiency of the testing process. Moreover, when used in conjunction with the fault locating part of the engine, it will obtain a smaller number of solutions. Written specifications can be used to constrain the creation of mutant solutions and the tool can propose only one solution to fix the fault.

V. CONCLUSIONS AND FUTURE WORK

Software development is prone to producing lower than expected quality software while the chance of project failure is high. Software Testing is an important, though complex, area of software development that mainly affects the quality and reliability of delivered software systems. A high percentage of software development time is devoted to testing.

Automatic software testing approaches are increasingly popular among researchers. They develop effective methods for fault locating and debugging so as to reduce testing complexity and lead to faster and cheaper software development steps with high quality standards.

Mutation testing is a technique that produces different versions of a program under study, each of which differ slightly from the original one, often mimicking common mistakes that programmers tend to make. These mutated versions are used to either identify faults or to assess the adequacy of a given set of test cases. In this context, a this paper proposes a simple, yet efficient mutation engine, in which a user-selectable number of mutation operators can be applied at the method level and incorporating CC to generate only valid mutants based on the program's specifications. The engine is developed in the Visual Studio 2010 platform and utilizes Code Contracts to represent the specifications that must be satisfied with pre-conditions, post-conditions and invariants for both C# and VB programming languages.

The engine is supported by a dedicated software tool consisting of four main parts. The first part verifies the syntactical correctness of the source code and proper linking with the appropriate libraries. The second part statically analyses the source code using grammatical analysis and produces the Abstract Syntax Tree representation of the source code. The third part uses the information gathered from the AST and generates mutations using specific operators selected by the user and obeying the rules imposed by the encoded specifications. The last part is the test case assessment component which either calculates the quality of a given test cases set or proposes possible corrections of faults that exist in code.

Five series of experiments were conducted that showed that the mutation engine is a tool that may be used for identifying faults in the code and for assisting the creation of the proper set of test data, both in C# and VB. Furthermore, the experiments demonstrated that the engine scales up smoothly as programs become larger in a time effective manner for creating and testing the mutants. Lastly, the incorporation of specification-based concepts allows for the significantly improved performance of the mutation engine by

reducing the number of mutants processed and solutions proposed according to the desired functionality expressed in the specifications, thus saving time and effort.

Future work will involve extending the proposed mutation engine to include more class-level mutators. Further additions and enhancements will be performed for both the C# and VB modules of the tool, while for the VB support the problem of applying mutators that produce invalid statements will be addressed. Moreover, integration of the engine with tools offered by the VS2010 is under investigation such as PEX, which is responsible for unit testing in order to automatically create test cases sets that have high code coverage [20] and UModel, which assists in creating UML diagrams. The UML diagrams from UModel can then generate source code that incorporates specifications that were set in the diagrams. This integration will enable the formation of a complete testing environment with dynamic user interaction, both at the flow of control level and at the diagrammatical level.

Our work can be compared only to a limited number of similar studies in literature: Saleh and Kulczycki [21] investigated how formal specifications can detect implementation errors in C# with the use of Creator of Mutants (CREAM) tool [22] and Boogie verifier [23] of SPEC# specifications. Their work tries and succeeds in showing how formal methods can affect the creation of bug-free programs by assessing their ability to detect design-time errors based on the SPEC# specifications. They concentrate on identifying faults created by mutations, which do not satisfy the specifications, at design level, without the need for executing the code. Our approach has a different purpose as it aims at assessing the quality of a test case set to identify faults in code and propose corrections for them. CCs are used, instead of SPEC#, for defining specifications which are verified dynamically upon code execution against a test case set and reports on any input values that do not satisfy the specifications. This makes possible the elimination of mutants that, although their code verifies statically the specifications, their execution against specific input values fails those specifications. Also our choice of CCs over SPEC# provides support for specifications in any language offered by VS, while SPEC# is designed to work only with C# code.

For MT in Java the work of Nica et al. [24] tries to answer the question if MT is really suitable for use in real-world environments. They evaluate the use of three different mutation tools for Java, MuJava, Jumble and Javalanche on some of the Eclipse IDE's source code, while they use the included JUnit tests provided with the source code on the Eclipse's repository to evaluate them. They neither try to locate and fix faults in code, nor do they assess the quality of the test cases set. Also, they use Java source code, while our work proposes a mutation engine to be used with both C# and VB programming languages.

Further validation of the proposed mutation engine will take place with the use of projects developed by graduate students. This will enable a more systematic evaluation of the engine using programs of different size and complexity that will include real faults made by programmers, while assessing various parameters, such as the time for creating and processing mutations, the type of mutators used and the nature

of the errors introduced. This systematic investigation will also bring to light any scalability issues not detected in this version of the engine. Moreover, efforts for increasing the performance of the Mutation Engine will be made with the use of parallel programming and multithreading, coupled with benchmarking tasks on a variety of different processing power systems. Lastly, the problem of regression faults will be addressed by exploring the feasibility of providing a correction to more than one fault without affecting any previous corrections.

REFERENCES

- [1] A.S. Andreou and P. Yiasemis, "A Specifications-Based Mutation Engine for Testing Programs in C#", Proceedings of the Sixth International Conference on Software Engineering Advances (ICSEA), Barcelona, Spain, 2011, pp. 70-75.
- [2] C. Kaner, J.H. Falk and H.Q. Nguyen, "Testing Computer Software", John Wiley & Sons Inc., New York, NY, USA, 1999.
- [3] A. Bertolino, "Software testing research: achievements, challenges, dreams", Proceedings of 29th International Conference on Software Engineering (ICSE 2007): Future of Software Engineering (FOSE'07), Minneapolis, USA, 2007, pp. 85-103.
- [4] B. Gauf and E. Dustin, "The case for Automated Software Testing", Future Directions in Software Engineering Journal, Vol. 10 (3), 2007, pp. 29-34.
- [5] R. Patton, "Software Testing", Sams Publishing, 2nd edition, 2006.
- [6] M.E. Khan, "Different Forms of Software Testing Techniques for Finding Errors," International Journal of Computer Science Issues (IJCSI), 2010.
- [7] "Mutation Testing Repository", <http://www.dcs.kcl.ac.uk/pg/jiayue/repository/>, [accessed 10 May 2011].
- [8] M. Nica, S. Nica and F. Wotawa, "On the use of mutations and testing for debugging," Software-Practice and Experience, Article published online, 2012.
- [9] R.A. DeMillo, R.J. Lipton and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", IEEE Computer Vol. 11(4), 1978, pp. 34-41.
- [10] R.G. Hamlet, "Testing Programs with the Aid of a Compiler", IEEE Transactions on Software Engineering, Vol. 3(4), 1997, pp. 279-290.
- [11] M. Harman, Y. Jia and W.B. Langdon, "Strong Higher Order Mutation-Base Test Data Generation", ESEC/FSE'11, Szeged, Hungary, September 5-9, 2011.
- [12] A.J. Offutt, "The Coupling Effect: Fact or Fiction", ACM SIGSOFT '89 - Third symposium on Software testing, analysis, and verification ACM, New York, USA, 1989.
- [13] G. Fraser and A. Zeller, "Generating Parameterized Unit Tests", International Symposium on Software Testing and Analysis (ISSTA'11), Toronto, Canada, July 17-21, 2011.
- [14] A.A. Sofokleous and A.S. Andreou, "Automatic, Evolutionary Test Data Generation for Dynamic Software Testing", Journal of Systems and Software, Vol. 81(11), 2008, pp. 1883-1898.
- [15] C.C. Michael, G. McGraw and M.A. Schatz, "Generating software test data by evolution", IEEE Transactions on Software Engineering (12), 2001, pp. 1085-1110.
- [16] "Visual Studio 2010", (2009) <http://www.microsoft.com/visualstudio/en-us/products/2010-editions>, [accessed 18 May 2011].
- [17] "Code Contracts User Manual", (2010), Microsoft Corporation, <http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf> [accessed 20 May 2011].
- [18] "SPEC#", (2004), <http://research.microsoft.com/en-us/projects/specsharp/>, [accessed 04 August 2012].
- [19] "SharpCode - The Open Source Development Environment for .NET", (2009), <http://www.icsharpcode.net/opensource/sd/>, [accessed 17 May 2011].
- [20] "Pex and Moles - Isolation and White box Unit Testing for .NET", (2004), <http://research.microsoft.com/en-us/projects/pex/>, [accessed 04 August 2012].
- [21] I. Saleh and G. Kulczycki, "Design-Time Detection of Implementation Errors Using Formal Code Specification", RESOLVE 2010 Workshop: Advances in Automated Verification, Denison University, Granville, Ohio, June 8, 2010.
- [22] A. Derezinska and A. Szustek, "CREAM - a System for Object-oriented Mutation of C# Programs", Information Technologies, Vol.13 (5), Gdansk, 2007, pp. 389-406.
- [23] "Boogie: An Intermediate Verification Language", <http://research.microsoft.com/en-us/projects/boogie/>, [accessed 08 December 2012].
- [24] S. Nica, R. Ramler and F. Wotawa, "Is Mutation Testing Scalable for Real-World Software Projects?", Third International Conference On Advances in System Testing and Validation Lifecycle (VALID), Barcelona, Spain, 2011.