A Specifications-Based Mutation Engine for Testing Programs in C#

Andreas S. Andreou
Department of Electrical Engineering and Information Technology,
Cyprus University of Technology
Limassol, Cyprus
email: andreas.andreou@cut.ac.cy

Pantelis Stylianos Yiasemis
Department of Electrical Engineering and Information Technology,
Cyprus University of Technology
Limassol, Cyprus
e-mail: pm.yiasemis@edu.cut.ac.cy

***Abstract*—This paper presents a simple and efficient engine which produces mutations of source code written in C#. The novelty of this engine is that it produces mutations that do not contradict with the specifications of the program. The latter are described by a set of pre- and post-conditions and invariants. The engine comprises two parts, a static analysis and syntactic verification component and a mutation generation component. Preliminary experiments showed that the proposed engine is more efficient than a simple mutations generator in terms of producing only valid mutations according to the specifications posed, thus saving time and effort during testing activities.**

Keywords-mutation testing; mutation engine; specifications;

I. INTRODUCTION

Technology advancements nowadays lead to the automation of a large number of activities within the software development process. The exploitation of computing power drives the need for producing better, faster and more reliable software systems. Nevertheless, the aforementioned targets increase software complexity and size, making this need hard to be satisfied. The competition in the software development market pushes companies to increase their productivity, developing software in tighter time limits usually sacrificing the quality of the resulting software.

One of the most significant reasons for the inadequate quality control in software development is the lack of efficient software testing. The latter is a way for verifying the correctness and appropriateness of a software system, or, alternatively, for ensuring that a program meets its specifications ([1], [2]). Software testing is not a simple process; on the contrary, it consumes a large percentage of the time and budget of the whole development process. In some cases it even surpasses the time needed for the creation of the software product. Its main purpose is to reveal and locate faults so as to assist developers improving the functional behavior of the system under development.

Software testing consists of two main processes, the identification of faults (testing) and their correction (debugging). Indentifying faults is the most time consuming process as it can take up to 95% of the time of software testing. Having this in mind, we can safely conclude that there will be a constant need to develop tools that will assist in accelerating and automating the testing process, guiding developers to locate and debug faults faster and more efficiently.

The aim of the present paper is to introduce a mutation engine for source code written in C#, which is the basic element 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 two components: The first offers the ability to validate the grammatical correctness of the source code and provides a form of statistical analysis for exporting useful information that can be used to process/modify the source code. The second involves the production of mutations of the original source code and facilitates the identification of faults, as well as the assessment of the quality of test data.

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, as along with a brief demonstration of the supporting software tool. Section IV describes a set of preliminary 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 McMinn [3], three different kinds of software testing techniques exist. These are White Box Testing (WBT), Black Box Testing (BBT) and the mixing of the two called Gray Box Testing (GBT). Each of these three techniques offers its own advantages and disadvantages, differing on 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 needs to have full access to the source code and know exactly the way it works. Advantages of this method are that it can locate coincidental correctness, this is the case where the final result is correct but the way it is calculated is not. Moreover, all possible paths of code execution may potentially be tested offering the ability to identify errors or/and locate parts of dead code, that is, parts that are never executed.

Different techniques have been proposed for WBT making use of the structure of the source code or the sequence of execution, giving birth to static code analysis and testing for the former and dynamic testing for the latter. We concentrate on dynamic testing where the actual flow of execution drives

 ISBN: 978-1-61208-165-6

test data production. One such technique that has gain serious interest among the research community is Mutation Testing (MT).

MT is a relatively new technique introduced by DeMillo et al. [4] and Hamlet [5], which is based on performing replacements in code statements through certain operators that correspond to specific types of errors, producing the so-called mutant programs; the latter are then used to assist in producing or/and assessing the quality of test data as regards revealing the errors in the mutants [6].

The general idea behind MT is that the faults being injected correspond to common errors made by programmers. This means that 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, pretty much the same as a slight change (mutation) in living species causing a different form of life. 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.

There are quite a few ways to represent code and provide the means for better understanding and management of the 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 its nodes usually corresponds to a single line of code, while the arcs connecting nodes represent the flow of execution. 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 [7] for automatically testing Java programs. More to that, CFG may drive the generation of test data by providing the means to construct an objective function for optimization algorithms to satisfy (e.g. by evolution, like Michael et al. [8]).

During the last years the Visual Studio (VS) platform [9] has been constantly evolving becoming one of the most wide spread platforms used today in the software industry. This is partly due to the fact that it provides to developers 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 led us to investigate its use for software testing, and more specifically for developing a new mutation testing tool.

Code Contracts (CC) are offered by VS2010 as the means to encode specifications [10]. CC may consist of pre-conditions, post-conditions and invariants. Their aim is to improve the testing process during runtime checking and static contract verification, as well as to assist in documentation generation.

The mutation engine introduced in this paper is partly based on the aforementioned concepts. More specifically, it utilizes CFG and static analysis as in [7] to extract the information needed for analyzing and describing adequately the source code under investigation. Moreover, it employs CC to embed the specifications required so that the program functions properly and static analysis (contract verification) in order to guide the production of meaningful mutant programs, that is, programs that do not violate their original specifications. The engine targets at offering the means for automatic, time-preserving software testing.

III. MUTATION ENGINE

A. Architecture

As previously mentioned, the mutation engine was implemented in the VS2010 platform. The selection of VS2010 was made partly because it is a relatively newly introduced platform, meaning that the components developed may be used as a 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 specifically designed to work with the C# programming language, but with minor changes and additions the support of the rest of the programming languages VS2010 platform offers may be enabled as well.

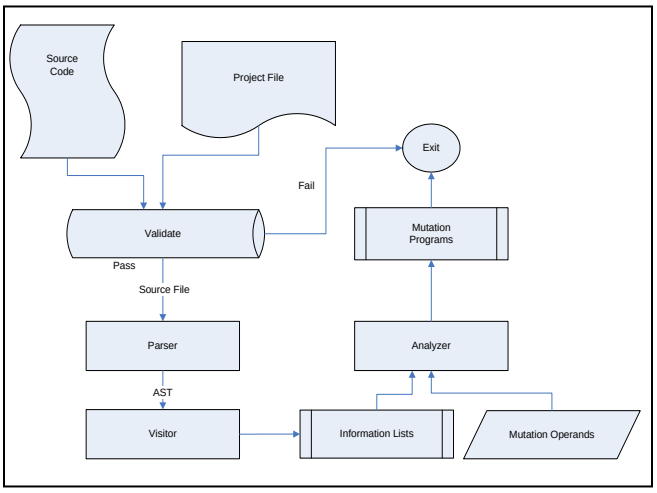

Figure 1. The mutation engine architecture

The architecture of the proposed mutation engine is depicted graphically in Figure 1 where three major components enable the execution of the engine's stages. The first is a source code validation component, which compiles the source code and presents the erroneous lines in code if such exist. This component takes as input a source code file (.cs), or an executable file (.exe), or a dynamic link library file (.dll), as well as the project file (.csproj). The project file is needed to provide the component with information for references in libraries and files that the source code must use 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.

The second component performs statistical analysis of the source code without the need of an executable form of the program under testing. By statistical analysis we mean exporting the useful information from the source code as regards the structure of the program. This component takes as input the source code file and uses the class AbstractSourceTree (AST) of SharpDevelop [11] to model the abstract syntax tree of the code. While compiling a source code file, a binary tree is created, each node of which represents a line of code. Traversing this binary tree, once the tree is completed, offers access to any part of the source code.

Analyzing the statistical component described above we can see that it consists of two sub-components, the *Parser* and the *Visitor*. The *Parser* analyses the source code and creates the AST as mentioned earlier. After it finishes, the *Visitor* passes through the tree collecting useful information, while giving the opportunity to the user to make changes and additions to the information stored. The implementation of the *Visitor* utilized the AbstractAstVisitor class of SharpDevelop, with some minor additions to help accessing all the nodes of the AST, both at the high and the low level 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.

The third and final component is actually the heart of the mutations production. 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 based on the rules each of them represents without breaching the grammatical correctness of the resulting program.

Mutations are performed at the method level via operators that are usually of arithmetic, relational, logical form, and at the class level with operators applied to a class or a number of classes and usually refer to 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 following:

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
- LOIA – logical operations insertion (assignment)
- LOD – logical operations deletion
- LODA – logical operations deletion (assignment)

Shift

- SOR – shift operations replacement
- SOIA – shift operations insertion (assignment)
- SODA – shift operations deletion (assignment)

Replacement

- PR – parameter replacement
- LVR – local variable replacement

B. 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. Therefore, when testing is based on mutations processing time may substantially increase as it is proportional to the number of mutants processed. This is a significant problem that may hinder the use of mutation testing in certain cases. Thus, there is a need to minimize mutation testing execution time. This is feasible taking into account the fact that a considerable number of useless mutations may be observed as the changes made to the code correspond to invalid forms of executions for that particular program as these are determined by the program's specifications. Therefore, we need to take these specifications into consideration when producing the mutants. This is exactly what we do via the Code Contracts supported in VS2010. Additionally, this feature is enhanced by ruling out mutation cases that have syntactical errors and are practically of no use.

The following example demonstrates how mutations are driven by the specifications inserted via CC, where class Test includes methods *Foo* and *Goo* and uses CC to express two pre-conditions (denoted by *Contact.Requires*) and one post-condition (denoted by *Contact.Ensures)*:

```
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)  }
```

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 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. Therefore, a sort of "thinking" before producing 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 in the subsequent analysis and testing activities.

C. *The software tool*

A dedicated software tool was developed to support the whole process. An example scenario is given below to demonstrate its operation: A source code file and the project file of the program tested are given as input to the system. 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 2) and the process is terminated. If there are just warnings, the user is again informed, but the system now continues to the next step. Statistical Analysis of the source code is executed next resulting the creation of the AST. The visitor component then passes through the binary tree and creates the lists that store the information found in the source code. Lastly, the third component takes as input the lists created earlier by the visitor and a set of selected mutators, applies these operators and returns the resulting mutated programs (Figure 3).

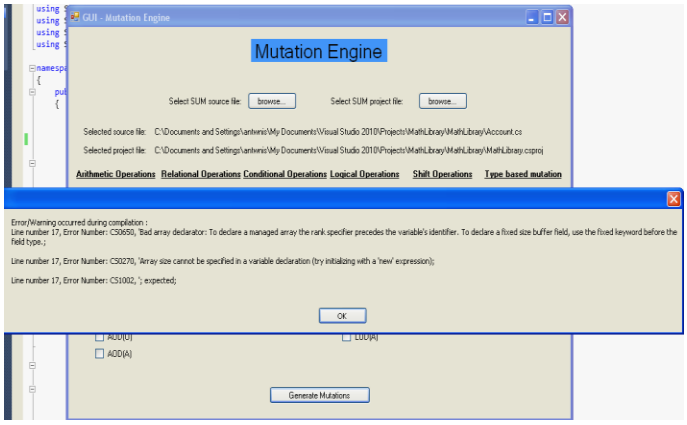

Figure 2. Execution : Errors in compilation

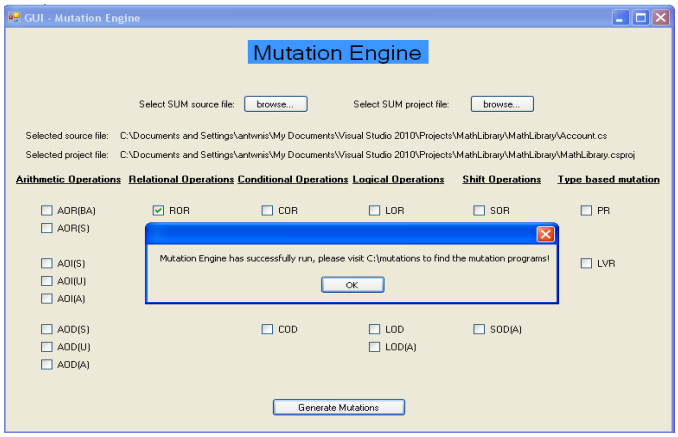

Figure 3. Execution : Mutations successfully produced

IV. EXPERIMENTAL RESULTS

A series of preliminary experiments was conducted to assess the correctness and efficiency of the proposed testing approach. The aim here was twofold: First, to demonstrate that the proposed engine works as it is supposed to, that is, it is able to produce correctly a number of mutations to be used for testing by performing atomic changes to the source code in hand according to a selected operator. Second, to assess whether the incorporation of specifications in the way mutations are produced indeed improves its performance by allowing only certain types of mutations to be executed and thus bounding the computational burden for revealing faults.

The first experiment is involved with assessing the quality (adequacy) of test cases to identify faults in a benchmark program via the use of the proposed approach. The second deals with fault detection using two sample programs with injected faults and producing mutants for detecting those faults. The final experiment compares the number of mutations produced with a standard mutation process to that of a specifications-driven production so as to assess the improvement in time performance. The experiments are analyzed below:

A. *Test-Data Quality Assessment*

This experiment used as benchmark the well-known triangle classification program listed below, which was tested against certain test data presented in Table I.

```
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; } }
```

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

i	j	k	Result
2	2	2	equilateral
0	1	2	not a triangle
3	3	1	isosceles
3	4	2	scalene

Using the values of Table I for the three variables it seems at first that we have tested adequately the TCP. Nevertheless, if we employ the mutation engine proposed we may conclude that the aforementioned set of test data is of low quality as at least one atomic change produces an error that is not recognized by the set. Indeed, the engine produced several mutations of which the one below passed the set as it successfully yields an identical result as the original program:

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

This simple code mutation suggests that indeed the proposed engine is able to assess the quality of a set of data to adequately test a given program.

B. Fault Detection

This set of experiments investigated the ability of the mutation engine to reveal errors that were injected in the initial source code of two programs, the first finds the maximum number between four integers, while the second implements division of two integer numbers and it is controlled by specifications expressed with code contracts.

In the first example, three faults were inserted in the code below, one relational, one parameter replacement and one unary.

```
public class FindMax {
  public int getMax(int num1, int num2, int
       num3, int num4)  {
    int max = 0;
    if (num1 > num2)   max = num1;
    else max = num3;
//** should have been max = num2 **//
    if (max < num3)  {
      max = num3;
      if (max > num4)   max = num3;  }
//** condition should have been (max < num4)
**//
    else {
      if (max < num4)   max = num4;  }
      return -max;  }  }
//** should have been return max **//
```

The engine applied a series of mutators, of which operators ROR, PR and AOD_U were actually the ones that revealed the injected errors. More specifically, ROR replaced relational operation '>' with '<', '>=', '<=', '-' and '!=' capturing the proper behavior. PR performed every possible combination of parameter replacement among (num1, num2, num3 and num4) resulting in the correct identification of presenting the error because of the use of num2 instead of num3. Finally, AOD_U successfully located the error in the last line after removing the minus sign.

The second example below 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 **//
```

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

C. Normal vs Specifications-Based Mutations Production

As mentioned earlier, a sort of "intelligence" was embedded in the engine that eliminates all mutants that violate the pre-conditions, post-conditions or invariants set for a program. Using class CompareParadigm listed earlier, we will compare the number of mutations produced by the mutation engine with the use of specifications to that of a normal (typical) mutations generator (in this case the engine with the CC disabled). Table II lists the mutations produced according the operator used. One may easily notice that a 58% reduction to the mutants was achieved by the "intelligent" engine, which resulted in 16 mutated programs compared to 38 produced without taking into consideration the specs. This is indeed a remarkable saving of effort and time with just a small part of code consisting of less than 20 statements. Therefore, we can safely argue that in cases of large programs the computational burden will be considerably eased, preserving at the same time the effectiveness and efficiency of the testing process.

 ISBN: 978-1-61208-165-6

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

Operator	Number of Mutations	
	Specs-based	***Normal***
AORBA	5	8
AOIS	7	10
AOIU	0	6
LOI	2	6
PR	2	3
LVR	0	5
Total	16	38

V. CONCLUSION AND FUTURE WORK

Software testing is an important, though complex, area of software development that aims at increasing the quality and reliability of software systems. Automatic software testing approaches are increasingly popular among researchers that attempt to handle the aforementioned complexity and lead to faster and cheaper software development with high quality standards.

Mutation testing is a technique that produces different versions of a program under study which differ slightly form the original one and uses these versions either to identify faults or assess the adequacy of a given set of test cases. In this context, the present paper proposed a simple, yet efficient mutation engine, which uses a number of mutation operators that can be applied at the method level and incorporating a sort of intelligence to generate only valid mutants based on the program's specifications. The engine is developed in the Visual Studio 2010 platform and utilized Code Contracts to represent the specifications that must be satisfied with pre-conditions, post-conditions and invariants.

The engine is supported by a dedicated software tool consisting of two main parts. The first part verifies the syntactical correctness of the source code and proper linking with the appropriate libraries, and provides statistical analysis of the source code, using grammatical analysis and producing the Abstract Source Tree representation of the source code. The second part uses the information gathered from the previous part and generates mutations using specific operators and obeying to the rules imposed by the encoded specifications.

A series of experiments was conducted that showed that the mutation engine constitutes a tool that may efficiently be used for identifying faults in the code and for assisting to the creation of the proper set of test data. The incorporation of the specification-based concepts can significantly improve performance by reducing the number of mutants processed, thus saving time and effort.

Future work will involve extending the proposed engine to include more class-level mutators, as well as investigating the potential of supporting other programming languages under the .Net framework. Moreover, we plan to integrate our tools with tools offered by the VS2010, like the PEX, which is responsible for unit testing and UModel, which assists in creating UML 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. Finally, our efforts will concentrate on evaluating the engine on a more systematic basis using sample programs of different size and complexity and assessing various parameters like the time for creating and processing mutations, the type of mutators used, the nature of the errors induced, etc. This systematic investigation will also address scalability issues and more specifically our future experimental evaluation will include code from large-sized, real-life software projects.

REFERENCES

[1] C. Kaner, J.H. Falk, , H.Q. Nguyen, Testing Computer Software, John Wiley & Sons Inc., New York, NY, USA, 1999.

[2] Bertolino, "Software testing research: achievements, challenges, dreams", Proc. 29th International Conference on Software Engineering (ICSE 2007): Future of Software Engineering (FOSE'07), Minneapolis, MN, USA, 2007, pp. 85–103.

[3] P. McMinn, "Search-based Software Test Data Generation: A Survey", Software Testing, Verification and Reliability Vol. 14(2), 2004, pp.105–156.

[4] 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.

[5] R.G. Hamlet, "Testing Programs with the Aid of a Compiler", IEEE Transactions on Software Engineering, Vol. 3(4), 1997, pp. 279-290.

[6] "Mutation Testing Repository", http://www.dcs.kcl.ac.uk/pg/jiayue/ repository/ , [accessed 10 May 2011]

[7] 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.

[8] 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.

[9] "Visual Studio 2010", (2009) http://www.microsoft.com/visualstudio/en-us/products/2010-editions, [accessed 18 May 2011]

[10] "Code Contracts User Manual", (2010), Microsoft Corporation, http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf [accessed 20 May 2011]

[11] "SharpCode",(2009), http://www.icsharpcode.net/opensource/sd/, [accessed 17 May 2011]

 ISBN: 978-1-61208-165-6