Partial Automation of Sensitivity Analysis by Mutant Schemata Approach

Zuhoor Al-Khanjari
Department of Computer Science, Sultan Qaboos University, Oman

Abstract: *According to Voas et al. testability is defined as the ease with in which faults may manifest themselves as failures when the software undergoes the testing process [39]. They also went further by introducing an approach for measuring sensitivity in terms of estimates from Propagation, Infection, and Execution (PIE) analyses of software and calculating the testability of a program through sensitivity estimate. Their testability calculations 'by hand' to determine the stability of the PIE analysis technique had drawbacks such as more time-consuming, high cost and less precision in the overall results [4]. Particularly the infection analysis part is one of the most expensive, sophisticated and time-consuming component of the PIE analysis technique. In order to solve this problem an investigation has been carried out by the author for establishing the feasibility of automating (or partially automating) the PIE analysis technique by means of a fast, and computationally less expensive Mutant Schemata Generation (MSG) approach [2]. An MSG-Infection tool has been developed to automate PIE analyses partially. This paper explains the use of MSG-Infection tool in automating the sensitivity analysis of C-programs and presents the results demonstrating the performance improvements achieved due to the MSG-Approach.*

Keywords: *Testability analysis, PIE technique, mutation testing, mutant schemata approach, Mothra mutation system, PiSCES, MSG-Infection tool.*

Received June 20, 2004; accepted February 2, 2005

1. Introduction

Due to rapidly growing sophistication in software products the reliability aspect of coded programs becomes an important issue. One aspect of reliability is the level of testability embedded in a program. Testability is defined as the ease with which faults, if present, may be exposed by test data as introduced by Voas *et al.* [39]. It has been proposed that reliability can be combined with testability analysis to give a better measurement for software quality [28]. Programs with high testability reveal their faults easily; those with low testability may contain faults that are very difficult to expose. The significance of the concept is twofold. First, if it is possible to measure or estimate testability, it can guide the tester in deciding where to focus testing effort. Secondly, knowledge about what makes some programs more testable than others can guide the designer to build their software with built-in 'design-for-test' feature.

In defining testability Voas *et al.* went further by introducing an approach for measuring sensitivity in terms of estimates from Propagation, Infection, and Execution (PIE) analyses of software. Through sensitivity estimate, the testability of a program can be calculated. Using the Voas's approach, the testability of a program can be defined as the minimum sensitivity among all sensitivities of all locations of that program.

The sensitivity is defined as the prediction of the minimum probability that a fault will cause a failure in the software at a particular location under a specified input distribution. For instance, if a location has a sensitivity of 1.0 under a particular input distribution D, then it is predicted that every input in D that executes location L will result in a software failure if L were to contain a fault. In the other extreme case, if a location has shown a sensitivity of 0.0, then it is predicted that no matter what fault is present in L, no input in D that executes L will cause a failure. Sensitivity gives a rough estimate of how frequently a fault will be revealed if one exists. This means that sensitivity is simply the probability of failure. There is a continuum of sensitivity in the region <0, 1>. The greater the likelihood that a fault in location L will be revealed during testing implies the greater the sensitivity that is assigned to L. In other words, a location with low sensitivity is termed *insensitive*, and a location with a high sensitivity is termed *sensitive*. Sensitivity analysis is the process of determining the sensitivity of a location in a program. It quantifies behavioral information about the likelihood that faults are hiding. Also, it can add another dimension to software quality assurance.

The sensitivity measurement at a location requires the estimation of probabilities of execution occurring, infection occurring and propagation occurring. This involves a repeated execution of the original program and corresponding mutants and observing the corresponding results. These three probability calculations are handled by the corresponding types of analysis namely Execution (or E) analysis, Infection (or I) analysis and Propagation (or P) analysis. These three types of analysis form the basis for the PIE-sensitivity analysis or simply sensitivity analysis model.

The main aim of this research is to investigate the testability concept and its measurement using MSG-Infection tool. To accomplish this aim this experimental study has been divided into two major parts. In the first part, a number of 'manual' calculations of testability have been carried out to determine the stability of the PIE analysis technique. In the second part, a testability procedure to automate (or semi-automate) the PIE analysis technique has been developed. Since mutation analysis is found to be having limited scope, it became necessary to develop a tool based on the mutant schemata approach. It has been argued that the Mutant Schemata approach build mutants of any program faster than other existing systems for mutation analysis such as 'Mothra' [23, 24]. Another important aspect of this approach is its ability to encode all mutants of the source code into one program called a metamutant (MM) program. Accordingly the author took an initiative to develop the MSG-Infection tool by modifying the existing MSG tool [8]. The tool uses MSG approach to generate the required classes of mutants. The MSG-Infection tool can be used to perform execution and infection analyses on C programs automatically. The application of this modified MSG tool in sensitivity analysis is a novel approach. In this approach the infection analysis of the PIE technique can be simplified and its performance can be improved significantly. This paper is presenting the work involved with the development of the modified MSG tool called MSG-Infection tool and the findings about performance improvement of PIE analysis due to the use of this tool.

The structure of the remaining part of this paper is as follows. Section 2 explains PIE analysis model used to estimate the testability procedure of a program. Section 3 explains sensitivity estimation model used for the testability experiments using the MSG-Infection tool. Section 4 explores the scope and limitations of some mutation based testing tools in perspective of automating the PIE analysis. Section 5 explains the concept of MSG-Infection approach and general structure of the MSG-Infection tool. Section 6 explains the design and implementation aspect of the MSG-Infection tool. Section 7 presents the results of the evaluation of the new tool. The last section discusses pros and cons of the MSG-Infection approach and some future directions required to improve the automation aspect of the PIE analysis.

2. Sensitivity Analysis PIE Model

This section summarizes the PIE model for measuring the sensitivity of the locations and the testability of the program. PIE is a white-box analysis technique based on the syntax and semantics of the code under test [26]. It makes predictions concerning future program behavior by estimating the effect that input distribution has, syntactic mutants and changed data values in data states have on current program behavior [28].

The PIE assessment model implements the definition of testability promoted by Voas and colleagues [29, 30, 39] by performing three independent dynamic analyses: Execution, infection, and propagation, which produce a set of estimates for each location of the given program. The three probability estimates can then be integrated to derive the sensitivity of each location and the overall testability of the program. The sequence of the three analyses is sometimes called the 'fault/failure model', because it relates faults, data state errors and failures [32]. The method is dynamic in the sense that it needs to execute the code in estimating the testability of a program. A location in PIE analysis can be an assignment statement, an input statement, an output statement, or the <condition> part of an if- (or a while-) statement. This definition for a location is based on Korel's definition [15] for a single instruction.

Before conducting the PIE analysis technique, several properties of the state of the program and knowledge of its environment must be assumed: the program is close to being correct semantically and syntactically, test cases should be available from an infinite sampling distribution. Hamlet and Voas have pointed out that the "PIE model is very simplistic, because it assumes that faults occur at single locations" [10]. Voas suggested that before conducting the PIE technique one should know if a program is likely to propagate data state errors if any have been created [27]. The three independent processes of PIE are discussed below.

Execution analysis is the process for predicting the probability that a location is executed when inputs are selected according to a particular input distribution D. It is concerned with the possibility that a particular location will have an opportunity to affect the output. The execution estimate of a particular location L, denoted by ε_L, can be determined by dividing the number of inputs (selected according to D) that execute location L, by the total number of test cases (expected to be large).

Infection analysis is the process for evaluating the probability that the succeeding data state of location L is different from the succeeding data state that a specific mutant creates, given that the original location and the mutant execute on a data state that would normally precede L. A data state is a collection of all variables and their associated values at some point during program execution. It may contain Boolean variables that represent the condition part of an if- or while-statement. Infection analysis involves three stages: recording the data state immediately before a location in the code, mutating the location, executing the original location and the corresponding mutant on the data state and observing whether the resulting data states are different [37]. In other words, if a fault exists in a location and it is executed, then the fault may produce an incorrect data state for that input. The incorrect data state is referred to as containing a data state error. Infection analysis is similar to fault-based testing in that both involve changing the location syntactically. Fault-based testing aims at demonstrating that certain faults are not in a program [16, 17, 18, 19, 20, 22, 40]. Also infection analysis is similar to weak mutation testing [11, 12] in that data states are compared immediately after executing a location in its original and mutant forms. When the data states differ, a variable *count* should be incremented. The procedure should be repeated depending on the total number of inputs *n* (selected according to *D*). The *count* should be divided by *n* to give $\lambda_{L,m}$, the infection estimate of the specified mutant *m* at location *L*.

Propagation analysis is the process concerned with the evaluation of the probability that a forced change in an internal computational state causes a change in the program's output. Computing the propagation estimate of a selected variable *v* at location *L* involves several steps [37]. At first selecting an input randomly from the input distribution of the program and saving the data state immediately after location *L* are done. In the second step, a new data state is generated through changing the value of a live variable *v* selected from the data state. In the third step, the rest of the program should be executed using both the original data state and the new data state including the changed value of variable *v*. In the fourth step, the *count* variable is incremented, when the resulting outputs differ. This procedure is repeated for the total number of inputs *n* (selected according to *D*). The *count* value should be divided by *n* to give, $\Psi_{L,v}$, the propagation estimate of the live variable *v* at location *L*.

3. PIE Results and Sensitivity Estimation

As mentioned previously, the sensitivity analysis can be thought of as determining the probability of failure at each program location. Based on the 'fault/failure model' of Byers and Kamkar [5], this failure probability, P_f is expressed as the product of the three separate probabilities, but necessary, conditions: *fault execution*, *data state infection* and *infected state propagation*, i. e.,

$$P_f = P_{exe} \text{ x } P_{\inf|exe} \text{ x } P_{prop|\inf} \tag{1}$$

From this formula, one can see that the sensitivity analysis involves performing execution analysis, infection analysis and propagation analysis. Using the estimates collected from the Execution, Infection and Propagation analyses, sensitivity of all individual locations of the given program can be calculated. Since the lower bound on the associated confidence interval of each estimate is considered, it is assured that if bias occurs when determining a sensitivity value, the bias causes underestimation of the sensitivity rather than overestimation.

Voas and colleagues [28, 33, 39] have introduced a more sophisticated way of calculating sensitivity (β_L) of a location. They introduced the following equations:

1. Multiplication of the three estimates for execution, infection and propagation:

$$\beta_L = (\varepsilon_L)_{\min} * \min\left[(\lambda_{L,m})_{\min}\right] * \min\left[(\Psi_{L,v})_{\min}\right] \tag{2}$$

Where

ε_L: Execution estimate of location *L*.

$\lambda_{L,m}$: Infection estimate of location *L*, mutant *m*.

$\Psi_{L,v}$: Propagation estimate of location *L*, variable *v*.

$(\bullet)_{\min}$: Lower bound of the confidence interval for an estimate.

$\min[(\lambda_{L,m})_{\min}]$: Smallest estimate for the set of mutants (m) considered at *L*.

$\min[(\Psi_{L,v})_{\min}]$: Smallest estimate for the set of *live* variables considered at *L*.

2. The formula (2) can be modified to take account of the possible but unlikely occurrence that the proportion of data state errors that do not propagate, when created by the mutant that produces the minimum infection estimate, is exactly the proportion of data state errors that do not propagate when the minimum propagation estimate is produced. The sensitivity of location *L*, denoted β_L, is now given by the formula:

$$\beta_L = (\varepsilon_L)_{\min} * \left[\sigma\left(\min\left[(\lambda_{L,m})_{\min}\right], \min\left[(\Psi_{L,v})_{\min}\right]\right)\right] \tag{3}$$

where

$$\sigma(a,b) = \begin{cases} a-(1-b) & \text{if } a-(1-b)>0 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

After calculating sensitivities of the locations, it is possible to recognize insensitive locations. With these locations alternative techniques should be applied such as testing under a new distribution, proofs of correctness, code review, symbolic testing or exhaustive testing [28, 33, 39].

As mentioned before, sensitivity is calculated for all specified locations of a tested program. From the collected set of sensitivity estimates, testability can be measured by taking the minimum non-zero sensitivity over all the locations. This can be represented by the following equation:

$$\text{testability} = \min(\beta_L) \tag{5}$$

Where:

$1 \le L \le$ Total number of locations
β_L: Sensitivity of location L.

Sensitivity is clearly related to testability, but the terms are not equivalent. Sensitivity focuses on a single location in a program and a fault at that location can have effects on the program's inputs/output behavior. Testability encompasses the whole program and the collection of sensitivities under a given input distribution. Testability depends on faults, code and test data distribution. It can be determined by applying the minimum function on the sensitivities of all locations of the given program.

4. Automating the Estimation of Testability

As shown before, estimating the testability of a program requires conducting the sensitivity analysis with these tasks:

1. Locations should be executed to calculate the execution estimate.
2. Mutants of the tested program should be created to calculate the infection estimate.
3. Live variables of the specified locations should be perturbed to calculate the propagation estimate.
4. Sensitivity of each location of the program in hand should be calculated. The second task can be achieved by using either the Mothra Mutation System [7, 14] or the MSG approach [23, 24], while the last task can be achieved by conducting the PIE technique. However, both the first and the third tasks need some extra work to be done.

Meanwhile, testability estimation cannot automatically be performed using only one approach such as MSG approach. It needs a combination of techniques to be used. Thus to automate testability, one should be able to automate/semi-automate all techniques to get the best results. Some attempts have been made to automate/semi-automate the calculations of the PIE technique. An overview of some approaches and tools used with these approaches is given below.

4.1. Mothra Mutation System

Mothra is a flexible, complete and interactive testing environment, established on mutation testing of software systems. It provides a powerful user interface that facilitates software testing by performing mutation analysis on a given program. It can be viewed as a collection of 'plug-compatible' tools such as Godzilla [6] and Equivalencer [21]. Godzilla is a test data generator tool, while Equivalencer is equivalent mutant detector. Each major function of Mothra is implemented as a separate program that executes independently of other tools. The core of this collection of tools is a language system that contains a set of objects and programs that enable Mothra to translate, execute and modify the given programs. Mothra was developed in 1986 by a team of researchers in the Software Engineering Research Center of Georgia Institute of Technology. A comprehensive and detailed manual for the functionality of the Mothra system is given in [3].

4.2. PiSCES Software Testability Analysis Toolkit

As stated previously, automating the measurement of testability involves automating the three individual processes of sensitivity analysis: Execution, Infection and Propagation analyses. A commercial tool called the PiSCES Software Testability Analysis Toolkit^{TM} of the Reliable Software Technologies Corporation of Sterling, Virginia [9] implements the PIE algorithms. It evolved from various proof-of-concept prototypes [34]. It is the only commercial software for testability determination. It generates testability estimates by developing an instrumented copy of the original program. PiSCES is written in C++ for performing analysis on C programs.

PiSCES Toolkit^{TM} is a combination of several individual tools or packages. One of the tools is the SafetyNet tool that incorporates extended propagation analysis to get an estimate for the fault tolerance of a program or indeed, for individual modules, functions, or even lines [9, 34].

4.2.1. Limitations and Evaluations of the PiSCES System

PiSCES is one of the automated tools for performing sensitivity analysis on C programs. It produces testability predictions based on the PIE analysis technique. It creates an instrumented copy of the program in question, which is then compiled and executed. Voas *et al.* [38] approximated the size of the instrumented version in comparison with the original program to be "10 times as large as the original source code" [38]. To execute the instrumented copy of the program, an input file is needed, which can be either supplied together with the original source code or generated using the PiSCES tool.

As a limitation of PiSCES, it can run "around 3000-4000 lines of source code at a time" [36]. Since the amount of the memory that PiSCES requires increases with the size of the source code, larger systems or applications must be divided into modules. In such situation each module should be tested individually. Once all modules have received dynamic testability analysis, the results for the whole application can be deduced [13, 31, 35, 36].

The PiSCES system uses normal mutation testing that involves creating copies of the original version of the code with the required changes. In other words, PiSCES does not use the Mutant Schemata approach to create the required mutants [25]. PiSCES divides complex expressions into simpler expressions. Figure 1 shows an example derived from Voas [25].

```
Complex expression
t := 0.9 * (1.0 + sqr (1.0 + y)) * exp (em * glalxm-ga
     mmln (em + 1.0) - glg);

Simple expressions
aa := sqr (y + 1.0);
bb := aa + 1.0;
cc := em + 1.0;
dd := gammln (cc);
ee := em * glalxm – dd - glg;
ff := bb * exp (ee);
```

Figure 1. Representing a complex expression by simple expressions.

The above expression can be mutated using MSG-Infection tool as one expression. The details are found in the following section.

5. MSG-Infection Tool

Each tool explained in the previous section is able to perform their subtasks by manually running the programs that constitute the tool. This warrants a tool that can perform the tasks of PIE analysis with complete automation. The MSG approach has been proposed to automate PIE analysis. MSG-Infection tool has been developed for automating the PIE analysis partially. The MSG-Infection system retains the spirit of the MSG approach by encoding a number of mutants of each location in one single modified version of the original program. This tool has been given this name because its main focus was to use the MSG approach for calculating the infection analysis. An overview of the MSG approach is explained below.

5.1. MSG Approach

The MSG technique is used to represent program neighborhood. The program neighborhood is a collection of the original program plus the mutant programs called metaprogram. The purpose of MSG approach is to improve the performance of mutation analysis systems by generating a metaprogram [23, 24]. A mutant schema has two components, a MM and a metaprocedure set, both of which are represented by syntactically valid constructs. All mutations produced from conducting standard mutagens (also called variously as mutation operators, mutation transformations and mutation rules) can be represented by metamutations.

Metaprocedures are syntactically valid representations of the abstract entities found in mutant schemata. They can be categorized as either metaoperators or metaoperands. Metaoperator procedures perform one of a class of alternate mathematical operations. Each metaoperator is implemented using a case structure. Metaoperand procedures reference one of a set of program variables. Metaoperand procedures are unique to each program and must be generated a fresh for each program.

The MSG method has the ability to encode all mutants into one source-level program. This program is then compiled (once) with the same compiler used during development and is executed in the same operational environment at compiled-program speed. If Mutant Schemata can be combined with the PIE technique, then the sensitivities and testability of a given program can be estimated automatically.

5.2. MSG-Infection Tool

The MSG-Infection tool has been designed and developed as a prototype using C-language. The tool is designed to be flexible and maintainable. As the system is expected to be quite large, it is broken down into smaller modules to manage the complexity of the code. The modules are designed with the principles of *low coupling* and *high cohesion*.

The MSG tool parses a given C-program to generate automatically the corresponding MM program. While performing sensitive analysis the tool inputs several arguments from the user: the original program that is to be tested, input variables of the program, test cases, total number of test cases and termination identifier that identifies the end of the test cases. At the end of the analysis, the tool outputs execution and infection estimates of all selected locations of the tested program, the parse tree and the MM program. The tool will also provide the user with a file that contains the locations of the tested program and their corresponding line numbers in the original program. In addition, it will provide the user with a sorted list of individual test cases, locations and all corresponding mutants' count.

Since mutation testing is a computationally expensive process, efficiency is an important issue in the design and implementation of the tool. To accomplish this criterion, the MSG-Infection tool uses the Mutant Schemata approach with its efficient MM concept. Every effort has been taken to ensure that the tool is designed and implemented with good software engineering principles. It has been designed in a modular fashion so that it can be expanded or adapted easily for future development or maintenance. A context diagram, which represents an overview of the entire MSG-Infection tool, is given in Figure 2.

5.3. Components of the MSG-Infection Tool

The MSG-Infection tool consists of two subsystems: Execution and Infection (EI) subsystem and Testability Management (TM) subsystem. The EI subsystem does the major and the most difficult part of the work. It takes the original C program as an input, parses it and creates a meta-mutant program. The TM subsystem runs the MM program generated by EI subsystem to produce the execution and infection estimates of the given C program.

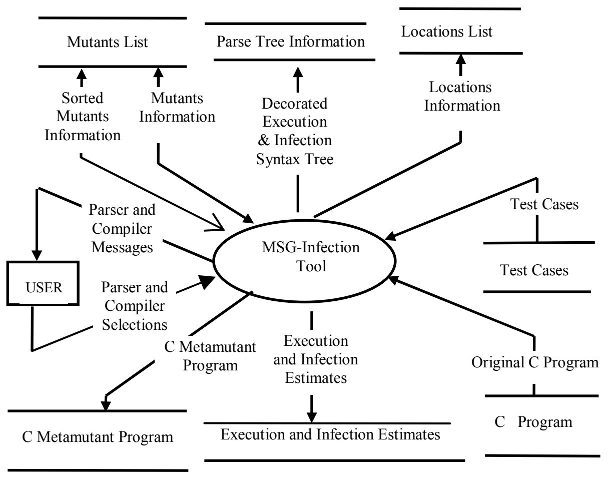

Figure 2. Context diagram of MSG-Infection tool.

5.3.1. Role of EI Subsystem

The EI subsystem is a modification of the existing MSG tool, which develops a parse tree corresponding to the given C program and modifies it to encode a variety of mutants at each location. This will be the largest and the most complicated part of the tool as it will need to analyze the C program, and produce a syntactically valid MM program which when executed can function as any mutant that was applied to the original program. The parse tree, which constructs the MM version of the given C program, will further be altered by EI subsystem so that each location *L* in the program now becomes as:

```
store pre-L;
execute L;
store post-L;
    for each mutant $L_m$ of L
    loop
        restore pre-L;
        execute $L_m$;
        compare post-$L_m$ with post-L;
    end loop;
restore post-L;
```

In the above, 'pre-*L*' and 'post-*L*' correspond to the data state immediately before and after location *L*. The result of each comparison of the post-L_m state with the post-*L* state is saved and used to determine the infection estimate of location *L*. The storing of the data state prior to a location is achieved by instrumenting with assignments to a special array storage that remains undisturbed during execution of the location or any of its mutants. The restore operation after the location can then use this array to recover the values that the variables had before the location.

As the EI subsystem applies mutants to the original C program, it will give the mutants identification labels and will put total number of each mutant in the specified loops in the MM program. The tool will determine where the mutations can be applied in the programs by studying the syntax, context, scope and semantics (meaning) of elements in the program. Elements can be looping and conditional constructs, expressions and constants. Simple search and replace or tokenizing methods do not provide the sophistication necessary for this level of program analysis, only compilation process of program parsing could do this. The tool by traversing the program's syntax tree, determines where mutations can be applied and rewrite it to implement the syntactically valid MM.

5.3.2. Role of TM Subsystem

The Testability Management subsystem manages the process of running the test cases against the original locations of the program and the corresponding mutants. It generates two files, one of which contains all required execution and infection estimates of the locations of the given program. The other file contains information about the test cases, location number and the state of the corresponding mutants whether they are killed or still alive. The TM subsystem compiles the generated MM program with the provided test cases, the available functions and macros to generate both execution and infection estimates. This subsystem will perform all tasks needed for conducting part of the sensitivity analysis: execution and infection analyses.

As mentioned before the subsystem keeps track of all mutants' count and saves them into a file with the corresponding test cases and location number. The file will be sorted according to location number and test case number before the saved information can be used to perform the tasks of sensitivity analysis of the corresponding mutants.

Several important macros have been encoded in the TM subsystem to facilitate the compilation and the execution processes. Further macros can easily be added to the subsystem.

6. Development of the MSG-Infection Tool

The EI and TM subsystems are completely independent of each other. The only interfacing requirement between them is that the input to TM is a MM program that is the output of the EI subsystem. The coupling between them is very low. Either subsystem can be modified without affecting the other as long as the interface between them is maintained consistent. The schematic view of the MSG-Infection tool particularly displaying the roles of EI and TM is shown in Figure 3.

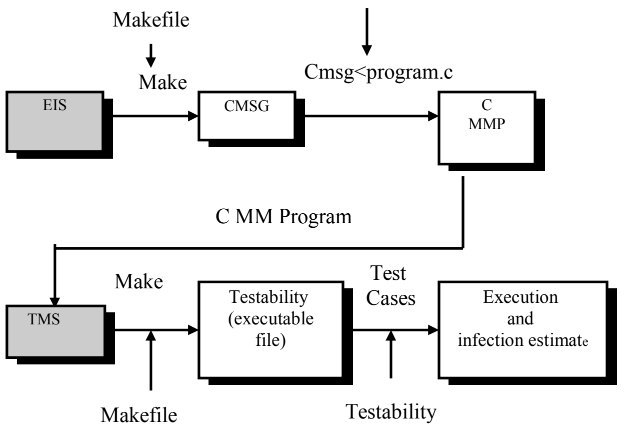

Figure 3. Schematic view of MSG-infection tool.

6.1. Implementation of EI Subsystem

The Execution and Infection subsystem will need to perform the three phases (lexical analysis, syntax analysis and semantic analysis) in order to generate the C MM program. At the beginning of the project, a skeleton C parser program was available. The skeleton parser reads a text file, parses it and outputs whether it is a valid C program. It uses the UNIX tools *lex* and *yacc*. On the other hand the MSG tool, from which some modules for constructing the parse tree with metafunctions have been taken, was available. These modules have been slightly changed to meet the needs of the MSG-Infection tool.

6.1.1. Mutating the Syntax Tree

After completing the construction of the syntax tree for the given program, the EI subsystem sets about generating the MM of the program by transforming and modifying the syntax tree. This involves the construction of the mutants. The subsystem *encoded* four mutant classes: *Arithmetic operator replacement, constant replacement, statement deletion* and *variable replacement*. These mutant classes have been used according to the recommended standard mutation types of the C language and naming conventions in [1].

Special care needs to be taken here, as many mutants can be applied to the single program construct such as an expression. In such cases, it is vital that the mutants are applied in a structured way so that they do not interfere with each other. Any such interference would cause the generated MM to be wrong and would probably produce a non-compiling version of MM. To prevent mutations interfering with each other, EI implements each mutant in such a way that the added mutant code encases the original statement in a neat way so that the whole structure becomes a single statement, allowing further mutants to be applied. Figure 4 demonstrates how the mutants are constructed for a single statement or expression.

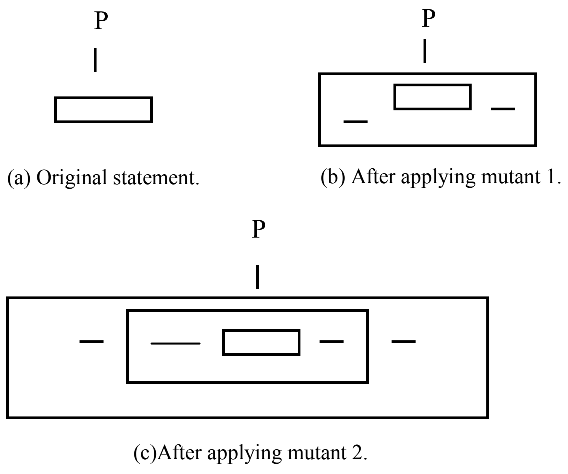

Figure 4. Multiple (or nested) mutants.

It is equally important that no mutations are applied to any code that has been generated by a previous mutant: Such an event would firstly produce mutants that should not exist, and secondly would most probably cause an infinite loop, as the code being mutated would produce yet more code that would be mutated, which would in turn produce yet more code, and so on.

EI prevents this problem by applying the mutations to the node in an in-order traversal. As each node is encountered, its production name, production identity, context and scope are analyzed to determine whether a mutation is to be applied. EI restricts itself to only being able to modify the immediate child nodes of the current node being examined. No other nodes in the subsystem may be changed. This contains the modifications to a local area in the syntax tree so that the code added by the mutation can be controlled and protected from mutation.

As the nodes are traversed and mutated in an in-fix pattern it will always be the case that, for each node that is visited, all of the ancestors of the node will already have been analyzed and probably modified by mutations. It will also be true that none of the descendants of the node will have been visited yet. This means that EI is free to examine the descendants of the current node, with the knowledge that all of them are in their original state. EI may therefore look at the descendants of the current node to determine its context. The ancestors of the node may not be looked at as they have been already mutated, and will therefore contain code that is not true to the original program.

The scope of a node is determined with the help of its ancestors, but this is now a problem as the order in which the mutants are applied has caused the ancestors of the current node to be mutated, altering their contents significantly. EI solves this problem by maintaining a stack throughout the MM generation process. When each node of the tree is first encountered, its production type is pushed on to the stack before any mutations are applied to it. The element is popped off the stack when all of the descendants of that particular node have been traversed. The stack therefore keeps a record of the productions that are in scope, and their order of scope. EI searches through this stack structure to ascertain the exact scope of the node.

6.2. Implementation of TM Subsystem

The Testability Management (TM) subsystem manages the process of running the test cases against the original locations of the program and the corresponding mutants. It generates two files as mentioned before one of which contains all the required infection and execution estimates for each location of the given program. The other file contains mutants list information that has been discussed previously. This subsystem performs tasks that are easier than the tasks performed by the EI subsystem. It contains all macros, mutation functions and execution and infection algorithms along with other functions used for checking and comparison purposes of the data used by the MM program. This means that the TM subsystem includes the built-in libraries that controls and facilitates the execution of the MM program.

6.3. Limitations and Evaluations of the MSG-Infection System

The MSG-Infection system uses an existing system called MSG [8]. The purpose of developing this tool was originally to improve the performance of infection analysis. However, the tool is found to have some limitations. The program under test should have at least one location, one input variable and any conditional statement that should use braces '{'and'}'. Secondly the program should not have 'include' statement and any variable initialization. Thirdly, the tool has no scope to handle arrays, pointers, matrix and also other data types such as character and Boolean type in the test programs. Finally the tool requires the user to specify a set of input and output variables. Reading of the input variables will be performed by a proper sequence of read statements of the generated metaprograms.

7. Evaluation Results of MSG-Infection Tool

Table 1 gives the list of programs that were selected for testing the capability of the MSG-Infection tool. The table also describes the test features of each program.

Table 2 provides some information about the tested programs and their corresponding MM program. It shows program name, number of locations, total number of test cases used and total number of lines of both the original C program and the corresponding MM program. Also it shows the size of the compiled versions of both the original C program and the corresponding MM program.

Table 1. Test programs and their testing features .

ID	Program Name	Description (Features Included)
P1	Quadratic.c	- one declaration statement of integer. - one or more operator. - more than one location. - more than one input variable.
P2	Absolute.c	- one declaration statement of type float. - one location. - one operator. - only one input variable.
P3	Sum.c	- two declaration statements of integer type. - more than two locations. - only one operator. - only one input variable of type integer.
P4	Product.c	- two or more declaration statements of the same type float. - two declaration statements of different types (i.e., integer and float). - more than two locations. - one operator. - one input variable of type integer.
P5	Average.c	- two declaration statements of different type (i. e., integer and float). - more than two locations. - only one operator. - only one input variable of type integer.
P6	Macros.c	- two locations. - one operator. - two or more input variables of the same type integer. - built-in functions which are called macros.
P7	Example.c	- two or more input variables of the same or different type (i. e., integer and float).
P8	Cast.c	- casting some variables. - casting a variable is to force that variable to be of certain type as required - built-in functions which are called macros.
P9	Try.c	- two or more assignment statements.
P10	Loc_10.c	- two or more assignment statements.
P11	Loc_50.c	- 50 assignment statements.
P11	Loc_100.c	- 100 assignment statements.
P11	Loc_150.c	- 150 assignment statements.

Table 2. Original C and MM programs sizes.

Program	L	Test Cases	Ori. Lines	MM Lines	Orig. Size	MM Size
Absolute.c	1	100	20	589	20531	80400
Example.c	1	1000	22	625	20531	84784
Macros.c	2	1000	33	981	24630	86048
Sum.c	2	100	26	946	20531	85776
Cast.c	3	1000	30	1351	41109	91496
Try.c	3	100	22	1272	24630	90840
Average.c	4	100	47	1654	20532	96480
Product.c	4	100	29	1646	20531	96360
Quadratic.c	4	10000	64	1720	45207	97088
Loc_10.c	10	1000	38	3879	41109	129016
Loc_50.c	50	1000	93	18987	45206	361416
Loc_100.c	100	1000	164	40532	45206	706952
Loc_150.c	150	1000	252	62772	49304	811189

Actually all programs have been constructed to check some properties of the tool such as declaration part, total number of locations, total number of variables, etc. The last four programs have been used to check the total number of locations that can be considered by the MSG-Infection tool. From Table 2 it can be deduced that ≈ 400 lines are added to the MM program for each location of the original C program. Therefore the sizes of the compiled MM files are larger than the sizes of the compiled original files.

To get a rough idea concerning the amount of time used for constructing, compiling and executing the MM programs, the following tables (Table 3 to Table 7) are provided. These tables show three types of time: *real time, user time* and *system time*. *Real time* can be defined as the amount of time spent in executing the command, *user time* can be defined as the amount of time spent in executing the user's process and *system time* can be defined as the amount of time spent in the system on behalf of the user's process. Since the main memory is simultaneously shared among several users, the times provided might be changed slightly every time the programs are used.

Table 3. Time for executing MM programs.

Command	RT	UT	ST
Absolute.c	0m4.14s	0m0.25s	0m0.12s
Example.c	0m8.57s	0m0.30s	0m0.12s
Macros.c	0m7.83s	0m0.45s	0m0.13s
Sum.c	0m6.82s	0m0.36s	0m0.12s
Cast.c	0m7.67s	0m0.61s	0m0.14s
Try.c	0m8.14s	0m0.18s	0m0.15s
Average.c	0m7.25s	0m0.67s	0m0.14s
Product.c	0m6.38s	0m0.65s	0m0.15s
Quadratic.c	0m9.12s	0m0.81s	0m0.16s
Loc_10.c	0m8.84s	0m1.71s	0m0.21s
Loc_50.c	0m20.12s	0m9.78s	0m0.57s
Loc_100.c	0m38.78s	0m26.79s	0m1.25s
Loc_150.c	1m25.88s	0m49.59s	0m1.94s

Table 4. Compilation time of original C programs.

Command	Real Time	User Time	System Time
Absolute.c	0m1.06s	0m0.76s	0m0.17s
Example.c	0m1.03s	0m0.75s	0m0.19s
Macros.c	0m2.20s	0m0.72s	0m0.25s
Sum.c	0m1.08s	0m0.75s	0m0.19s
Cast.c	0m1.19s	0m0.84s	0m0.23s
Try.c	0m1.19s	0m0.70s	0m0.25s
Average.c	0m1.27s	0m0.71s	0m0.24s
Product.c	0m1.04s	0m0.74s	0m0.19s
Quadratic.c	0m1.72s	0m0.82s	0m0.30s
Loc_10.c	0m1.22s	0m0.86s	0m0.23s
Loc_50.c	0m1.34s	0m0.92s	0m0.27s
Loc_100.c	0m1.48s	0m1.06s	0m0.23s
Loc_150.c	0m7.38s	0m1.06s	0m0.62s

Table 5. Time for executing MM programs.

Command (Make)	Real Time	User Time	System Time
Absolute.c	0m4.76s	0m3.32s	0m0.76s
Example.c	0m5.20s	0m3.36s	0m0.82s
Macros.c	0m5.71s	0m3.57s	0m0.81s
Sum.c	0m4.99s	0m3.51s	0m0.77s
Cast.c	0m5.55s	0m3.83s	0m0.80s
Try.c	0m6.33s	0m3.65s	0m0.94s
Average.c	0m5.39s	0m3.80s	0m0.88s
Product.c	0m6.09s	0m3.89s	0m0.78s
Quadratic.c	0m5.50s	0m4.03s	0m0.79s
Loc_10.c	0m7.20s	0m5.25s	0m1.00s
Loc_50.c	0m18.64s	0m15.00s	0m1.30s
Loc_100.c	0m39.22s	0m30.65s	0m2.17s
Loc_150.c	1m9.48s	0m46.68s	0m3.27s

Table 6. Execution time of original C programs.

Command (a.out)	RealTi me	User Time	System Time
Absolute.c	0m0.36s	0m0.04s	0m0.09s
Example.c	0m0.21s	0m0.03s	0m0.08s
Macros.c	0m0.28s	0m0.12s	0m0.06s
Sum.c	0m0.18s	0m0.03s	0m0.07s
Cast.c	0m0.53s	0m0.04s	0m0.08s
Try.c	0m0.89s	0m0.04s	0m0.09s
Average.c	0m0.51s	0m0.07s	0m0.12s
Product.c	0m1.01s	0m0.10s	0m0.14s
Quadratic.c	0m13.64s	0m1.83s	0m1.79s
Loc_10.c	0m0.30s	0m0.09s	0m0.08s
Loc_50.c	0m0.23s	0m0.10s	0m0.07s
Loc_100.c	0m0.50s	0m0.15s	0m0.09s
Loc_150.c	0m2.46s	0m0.41s	0m0.45s

Table 7. Time for executing MM programs.

Command (Testability)	RealT ime	User Time	System Time
Absolute.c	0m10.23s	0m0.09s	0m0.10s
Example.c	0m8.04s	0m1.86s	0m0.22s
Macros.c	0m6.28s	0m0.94s	0m0.15s
Sum.c	0m6.52s	0m0.08s	0m0.07s
Cast.c	0m7.71s	0m1.23s	0m0.15s
Try.c	0m6.81s	0m0.16s	0m0.09s
Average.c	0m8.29s	0m0.09s	0m0.08s
Product.c	0m5.32s	0m0.10s	0m0.08s
Quadratic.c	2m10.67s	0m30.84s	0m2.21s
Loc_10.c	0m15.92s	0m8.70s	0m0.22s
Loc_50.c	0m49.31s	0m42.07s	0m0.79s
Loc_100.c	1m4.71s	0m22.90s	0m5.79s
Loc_150.c	2m53.35s	2m28.20s	0m2.69s

Viewing the above tables, it is obvious that the execution time increases. That is, the execution time of the MM programs is roughly 60 times the execution time of the original C programs. Determining the sensitivity estimate of locations is an expensive and time-consuming process. Getting the estimates of the analyses more directly might help in solving the problem or part of it. This motivates researchers to find easier and more direct methods to estimate the testability of programs or at least to get an indication of the testability estimate of the program without conducting the actual analyses. Table 8 provides information that could be deduced from the original programs.

Table 8. Direct infection determination.

Function	**Infec. Est.**	**Comment**
f (*a*) = *a*;	high	*a* ∈]-∞, ∞[
f (*a*) = *a* - 50;	high	*a* ∈]-∞, ∞[
f (*a*) = *a* mod 2;	high	*a* ∈]-∞, ∞[
f (*a*) = SQUARE (*a*);	high	*a* ∈]-∞, ∞[
f (*a*) = sqrt (*a*);	high	*a* ∈]-∞, ∞[
f (*a*) = SQUARE (*a*) div 2;	high	*a* ∈]-∞, ∞[
f (*a*) = (*a* * 2) div 3;	high	*a* ∈]-∞, ∞[
f (*a*) = 2 * *a* – 3 + *a*;	high	*a* ∈]-∞, ∞[
f (*a*) = *a* – 5 * *a* + *a*;	high	*a* ∈]-∞, ∞[
f (*a, b, c*) = *b* * *b*- 4 * *a* * *c*;	high	*a, b, c* ∈]-∞, ∞[
f (*a*) = sqrt (*a*) * 2 + 3 - *a*;	high	*a* ∈]-∞, ∞[
f (*a*) = 3 + SQUARE (*a*) – 2 * *a* - *a*;	high	*a* ∈]-∞, ∞[
f (*a*) = 2 * sqrt (ABS (*a* * 2)) – 2 * *a*;	high	*a* ∈]-∞, ∞[
f (a) = (2 * *a* + SQUARE (9)) div (2 * SQUARE (25));	high	*a* ∈]-∞, ∞[
f (a) = sqrt (*a*) – SQUARE (*a*);	high	*a* ∈]-∞, ∞[
f (*a*) = not (*a*);	high	*a* = 0 or *a* = 1
f (*a*) = sin (*a*);	high	*a* ∈]-∞, ∞[(degrees)
f (*a*) = cos (*a*);	high	*a* ∈]-∞, ∞[(degrees)
f (*a*) = sin (*a*) + cos (*a*);	high	*a* ∈]-∞, ∞[(degrees)
f (*a*) = SQUARE (sin (*a*))+ SQUARE (cos (*a*));	high	*a* ∈]-∞, ∞[(degrees)
f (*a*) = tan (*a*);	high	*a* ∈]-∞, ∞[(degrees)
f (*a, b, c*) = (-*b* + isqrt (*b* * *b* – 4 * *a* * *c*)) div (2 * *a*);	low	*a, c* ∈ [0, 10], *b* ∈ [1, 1000]
f (*a*) = *a* div *b*;	low	*a* ∈]-∞, ∞[infection decreases as *b* increases
f (*a*) = *a* mod *b*;	low	*a* ∈]-∞, ∞[infection decreases as *b* decreases

This section discusses the generalization of the estimation of the infection analysis. By viewing the source code, one can say whether the assignment statements can produce a high or a low infection estimate. It has been seen that some operators or macros might hide faults and produce low infection estimate. Table 8 provides the infection estimate of some functions. The table shows the results for the functions that usually produce high infection estimate, 1.0 or close to 1.0 and the functions that might produce low infection estimate, close to 0.0. It helps the software engineers to detect the infection estimate of the locations directly from the macros used in those locations.

The MSG-Infection tool deals with the simple statements or locations. However, the arrays and other complex statements including arrays could be incorporated in the future investigations.

8. Conclusion

The existing tools seem to have some limitations such as time factor and unreasonable requirement of memory resources in evaluating the testability of programs. In order to tackle these limitations this paper has first reviewed the PIE analysis technique as the basis for developing an efficient tool with the support of MSG approach. Then it has presented the development of the MSG-Infection tool. Also, it showed the timing results of the MSG-Infection tool in determining the execution and infection analyses of various locations of the tested programs. A significant improvement in its performance over PiSCES is due to its ability to perform the tests based on weak mutation.

The MSG-Infection tool is not only better tool performance-wise, but also its requirement on memory resources is within the reasonable limit. The tool is interactive and made for carrying out testing on only C-programs. It is easy to maintain, adapt and expand scalable to a complete system. However there is some overhead due to extra code in MMs.

The sensitivity analysis technique used to evaluate the testability of programs has several advantages including assessment and quantification of software reliability. Development of MSG-Infection software promises for further research leading to ultra-reliable software.

Acknowledgement

Author thanks Steven Flanagan for permission to use his MSG system.

References

[1] Agrawal H., DeMillo R. A., Hathaway B., Hsu W., Krauser E. W., Martin R. J., Mathur A. P., and Spafford E., "Design of Mutant Operators for the C Programming Language," *SERC-TR-41-P Software Engineering Research Center*, Department of Computer Science, Purdue University, West Lafayette, Indiana, 1989.

[2] Al-Khanjari Z. and Woodward M. R., "Investigating the Relationship Between Testability and the Dynamic Range-to-Domain Ratio," *The Australian Journal of Information Systems (AJIS)*, vol. 11, no. 1, pp. 55-74, September 2003.

[3] Al-Khanjari Z., "Mutation Testing Using Mothra," *Dissertation for Master of Science*, Supervised by Woodward M., The University of Liverpool, 1995.

[4] Al-Khanjari Z., Woodward M. R., and Ramadhan H., "Critical Analysis of the PIE Testability Technique," *Software Quality Journal (SQJ)*, Kluwer Academic Publishers, The Netherlands, vol. 10, no. 4, pp.331-353, December 2002.

[5] Byers D. and Kamkar M., "A Hybrid Approach to Propagation Analysis," *in Proceedings of the 3rd International Workshop on Automatic*

Debugging (AADEBUG'97), Linkoping, Sweden, May 1997.

[6] DeMillo R. A. and Offutt A. J., "Constraint-Based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900-910, 1991.

[7] DeMillo R. A., Guindi D. S., McCracken W. M., Offut A. J., and King K. N. "An Extended Overview of the Mothra Software Testing Environment," *in Proceedings of the 2nd Workshop on Software Testing, Verification and Analysis*, IEEE Computer Society, Banff, Canada, pp. 142-151, 1988.

[8] Flanagan S. J., "Mutation Testing Using Mutant Schemata," *BSc Dissertation*, Computer Science Department, University of Liverpool, UK, 1997.

[9] Friedman M. A. and Voas J. M., *Software Assessment: Reliability, Safety, Testability*, Wiley, New York, USA, 1995.

[10] Hamlet D. and Voas J., "Faults on its Sleeve: Amplifying Software Reliability Testing," *in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93)*, Cambridge, Mass, ACM SIGSOFT SE Notes, USA, vol. 18, no. 3, pp. 89-98, 1993.

[11] Howden, W. E., "Completeness Criteria for Testing Elementary Program Functions," *in Proceedings of the 5th International Conference Software Engineering*, pp. 235-243, 1981.

[12] Howden W. E., "Weak Mutation Testing and Completeness of Program Test Sets," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, pp. 371-379, 1982.

[13] Khoshgoftaar T., Szabo R., and Voas J., "Detecting Program Modules with Low Testability," *in Proceedings of International Conference on Software Maintenance (ICSM'95)*, Nice, France, 1995.

[14] King K. N. and Offut A. J., "A FORTRAN Language System for Mutation-Based Software Testing," *Software Practice and Experience*, vol. 21, no. 7, pp. 685-718, 1998.

[15] Korel B., "PELAS-Program Error-Locating Assistant System," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1253-1260, 1998.

[16] Morell L. J. and Hamlet R. G., "Error Propagation and Elimination in Computer Programs," *Technical Report 1065*, University of Maryland, USA, 1981.

[17] Morell L. J., "A Model for Code-Based Testing Schemes," *in Proceedings of the 5th Annual Pacific Northwest Software Quality Conference*, 1987.

[18] Morell L. J., "A Theory of Error-Based Testing," *PhD Thesis*, Technical Report TR-1395, University of Maryland, USA, 1984.

[19] Morell L. J., "A Theory of Fault-Based Testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844-857, 1990.

[20] Morell L. J., "Theoretical Insights into Fault-Based Testing," *in Proceedings of the 2nd ACM SIGSOFT, IEEE Workshop on Software Testing*, Analysis and Verification, Banff, Canada, 1988.

[21] Offutt A. J. and Pan J., "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165-192, 1997.

[22] Richardson D. J. and Thompson M. C., "The RELAY Model of Error Detection and its Application," *in Proceedings of the 2nd Workshop on Software Testing*, Verification, and Analysis, Banff, Canada, pp. 223-230, 1998.

[23] Untch R. H., "Schema-Based Mutation Analysis: A New Test Data Adequacy Assessment Method," *PhD Thesis*, Dep artment of Computer Scie nce, Clemson University, South Carolina, USA, 1995.

[24] Untch R. H., Offutt A. J., and Harrold M. J., "Mutation Analysis Using Mutant Schemata," *in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93)*, Cambridge, Mass, ACM SIGSOFT SE Notes, vol. 18, no. 3, 139-148, 1993.

[25] Voas J. M., "A Dynamic Failure Model for Performing Propagation and Infection Analysis on Computer Programs," *PhD Thesis*, College of William and Mary, Virginia, US, 1990.

[26] Voas J. M., "Dynamic Testing Complexity Metric," *Software Quality Journal*, vol. 1, no. 2, pp. 101-114, 1992.

[27] Voas J. M., "Factors that Affect Software Testability," *in Proceedings of the 9th Pacific Northwest Software Quality Conference*, Portland, Oregon, USA, pp. 235-247, 1991.

[28] Voas J. M., "PIE: A Dynamic Failure-Based Technique," *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 717-727, 1992.

[29] Voas J. M., "Software Testability Measurement for Intelligent Assertion Placement," *Software Quality Journal*, vol. 6, no. 4, pp. 327-335, 1997.

[30] Voas J. M. and McGraw G., *Software Fault Injection: Inoculating Programs Against Errors*, Wiley, New York, USA, 1998.

[31] Voas J. M. and Miller K. W., "Dynamic Testability Analysis for Assessing Faults Tolerance," *High Integrity Systems Journal*, vol. 1, no. 2, pp. 171-178, 1994.

[32] Voas J. M. and Miller K. W., "Software Testability: The New Verification," *IEEE Software*, vol. 12, no. 3, pp. 17-28, 1995.

[33] Voas J. M. and Miller K. W., "The Revealing Power of a Test Case," *Software Testing, Verification, and Reliability*, vol. 2, no. 1, pp. 25-42, 1992.

[34] Voas, J. M., Miller K. W., and Payne J. E., "A Software Analysis Technique for Quantifying Reliability in High-Risk Medical Devices," *in Proceedings of the 6th IEEE Symposium on Computer-Based Medical Systems*, Ann Arbor, MI, 1993.

[35] Voas J. M., Miller K. W., and Payne J. E., "An Empirical Comparison of a Dynamic Software Testability Metric to Static Cyclomatic Complexity," *in Proceedings of the 18th Annual SE Workshop*, NASA-Goddard SE Laboratory Series Report 93-003, 1993.

[36] Voas J. M., Miller K. W., and Payne, J. E., "Automating Test Case Generation for Coverages Required by FAA Standard Do-178B," *in Proceedings of Computers in Aerospace 9*, CA Publisher: AIAA, San Diego, USA, 1993.

[37] Voas J. M., Miller K. W., and Payne J. E., "PiSCES: A Tool for Predicting Software Testability," *in Proceedings of the Symposium on Assessment of Quality Software Development Tools*, IEEE Computer Society, New Orleans, USA, pp. 297-309, 1992.

[38] Voas J. M., Miller K. W., and Payne J. E., "Software Testing and its Application to Avionic Software," *in Proceedings of Computers in Aerospace 9*, CA Publisher: AIAA, San Diego, USA, 1993.

[39] Voas J. M., Morell L. J., and Miller K. W., "Predicting Where Faults Can Hide from Testing," *IEEE Software*, vol. 8, no. 2, pp. 41-48, 1999.

[40] Zeil S. J., "Testing for Perturbations of Program Statements," *IEEE Transactions on Software Engineering*, vol. 9, no. 3, pp. 335-346, 1993.

Zuhoor Al-Khanjari is the assistant dean for Postgraduate Studies and Research, College of Science, Sultan Qaboos University. She is an assistant professor in software engineering, Department of Computer Science at Sultan Qaboos University, Sultanate of Oman. She received her BSc in mathematics and computing from Sultan Qaboos University, Sultanate of Oman, MSc and PhD in computer science from the University of Liverpool, UK. Her research interests include software engineering, database management, e-learning, human-computer interaction, intelligent search engines, and web data mining and development. Currently, she is the coordinator of the software engineering group in the Department of Computer Science, Sultan Qaboos University, Sultanate of Oman. Also she is coordinating e-learning facilities in the same department. She is a member in the editorial board of the International Arab Journal of Information Technology (IAJIT) and a member in the steering committee of the International Arab Conference on Information Technology (ACIT).

.