May. 31
International Journal of advanced studies in Computer Science and Engineering
IJASCSE Volume 4 Issue 5, 2015

Multi-Path Error Propagation: A Case for Defensive Programming in Software Implementation

Monday O. Eze
Department of Computer Science/Maths/Informatics
Federal Univ. Ndufu-Alike, Ikwo (FUNAI),
Ebonyi State, Nigeria

Shakeel A. Kamboh
Department of Maths & Statistics
Quaid-e-Awam University of Engineering, Science and Technology, Pakistan

***Abstract*— In system quality assurance, medical sciences, aviation and many other scientific fields, it is an accepted norm that prevention is better than cure. In other words, it is better to prevent errors than to spend valuable resources trying to unravel or fix such errors. In software engineering, the importance of defensive programming cannot be ignored. The concept of defensive programming emphasizes that a number of coordinated steps should be taken right from the software design stage in order to drastically reduce the possibility of system errors. Thus, defensive programming is a form of error-avoidance and vulnerability reduction mechanism. The current research is a study on the importance and practice of defensive programming in software projects. Particular interest is on a class of problems termed as multi-path error propagation. This is a scenario where a programming decision which has several possible options, and is dependent on the attributes of a given input parameter, leads to the propagation of system errors. Moreover, such an error could affect the overall system output, thus the need for defensive programming. Appropriate case studies and practical solutions were based on the three programming platforms - Java, C/C++ and Matlab.**

Keywords- Defensive Programming, Error-Avoidance, Software Engineering, Software Testing, Software Project, Error Propagation.

I. INTRODUCTION

Defensive programming is defined as a programming technique that saves debugging and testing time, and encourages the avoidance of unpredictable errors [1]. Defensive programming has also been described as the art of writing safe and sound programs [2]. One of the principles a software expert bears in mind when embarking on defensive programming is that a user may attempt to break a program either intentionally or unintentionally by providing flawed input to it.

Defensive programming is necessary in order to build bug-free software, and also to avoid software security hitches. For instance, [3] demonstrated the defensive programming techniques for dealing with e-mail spoofing in the PHP platform. The importance of defensive programming has been stressed by a number of *software engineering* practitioners [4]. A common strategy suggested by [5] is for programmers to apply defensive programming 'checks' within the source code so as to detect some design errors. The two strategic positions suggested for such precautionary measures are at the entry to a sub-program and after the sub-program execution. System debugging could become very complex and daunting [6], especially if the programmer fails to take early precautions against system bugs in large programming projects.

The choice of a programming platform could also have a lot of effects on both the programmer's productivity and system integrity. A particular programming language could have varied programming environments and inbuilt capabilities. For instance, a user-friendly integrated development environment [7] could be more preferable to a user-unfriendly platform. A syntax-aware editor [8] could help a programmer to track errors as early as possible, unlike others that allow errors to pile up till the debugging stage [9].

In defensive programming, a software expert always keeps in mind that things could go wrong long after a *software project* launch. With this mindset, a programmer tries to apply proactive measures [10] at the early stage of the system development so as to mitigate software failures in the future.

II. RELATED RESEARCH

It is important to outline some previous works in defensive programming. A research by [11] studied defensive programming from the object oriented point of view. One of the key findings is that objects may alter data previously declared as public. This scenario which was described as potentially dangerous has been solved through encapsulation, for instance in the C programming language.

A key defensive programming techniques in C++ is the outright declaration of constants using 'const' keyword within the program. As outlined by [12], such a practice could drastically reduce the possibility of errors in software projects. About ten serious web programming issues were listed by [13]. These were reportedly solved through the application of defensive programming in JavaScript 2.0. One of those issues is the use of an old browser that does not support modern JavaScript objects. A recent study [14] has focused on the application of defensive programming techniques to deal with common issues resulting from database related changes. In this category are changes to such database objects as tables, constraints, columns, and stored procedures.

The use of defensive programming to secure web applications is an active research issue. A security breach in

this category termed as SQL injection attack targets sensitive web data through manipulation of the original SQL queries [15]. At least four defensive programming approaches were outlined by [16] for dealing with SQL injection attacks, one of which is to limit the user privileges. An offshoot of defensive programming is the use of automated proofs to verify program correctness. A recent work in this regard used the SPARK toolset to ensure the absence of run-time errors in Ada software [17]. As already reported by [18], a number of modern programming languages such as Java use assertions to deal with defensive programming issues. The general workflow for defensive programming will be presented in the next section of this work.

III. GENERAL WORKFLOW

The diagram in Fig. 1F shows a general defensive programming workflow. No doubt, a singular illustration may not address all issues related to defensive programming. Therefore, this workflow is by no means viewed as an exhaustive evolution, but rather as a template to be improved upon in future works.

ramifications. For instance, the two programming languages C++ and Prolog may address a particular software implementation issue from different points of views. It is left for a software specialist to choose which programming platform the person views as appropriate for solving a particular software problem.

As shown in the workflow, each of the objects labeled DPS-1, DPS-2, DPS-3 and DPS-4 points towards a particular software development operation. The acronym DPS stands for 'defensive programming strategy'. Thus, DPS-2 represents a set of defensive programming strategies which could be applied at the software design stage, while DPS-3 represents those which could be introduced at implementation.

Similarly, the workflow also shows the spherical objects labeled as PErr-1, PErr-2, PErr-3 and PErr-4. The acronym PErr is used in this work to illustrate the concept of 'preventable errors'. This is hinged on the understanding that every successful implementation of a defensive programming strategy is a direct or indirect prevention of a particular error or set of errors. Similarly, each time a programmer fails to take a requisite defensive precaution, there are chances that one or more errors may pass into the

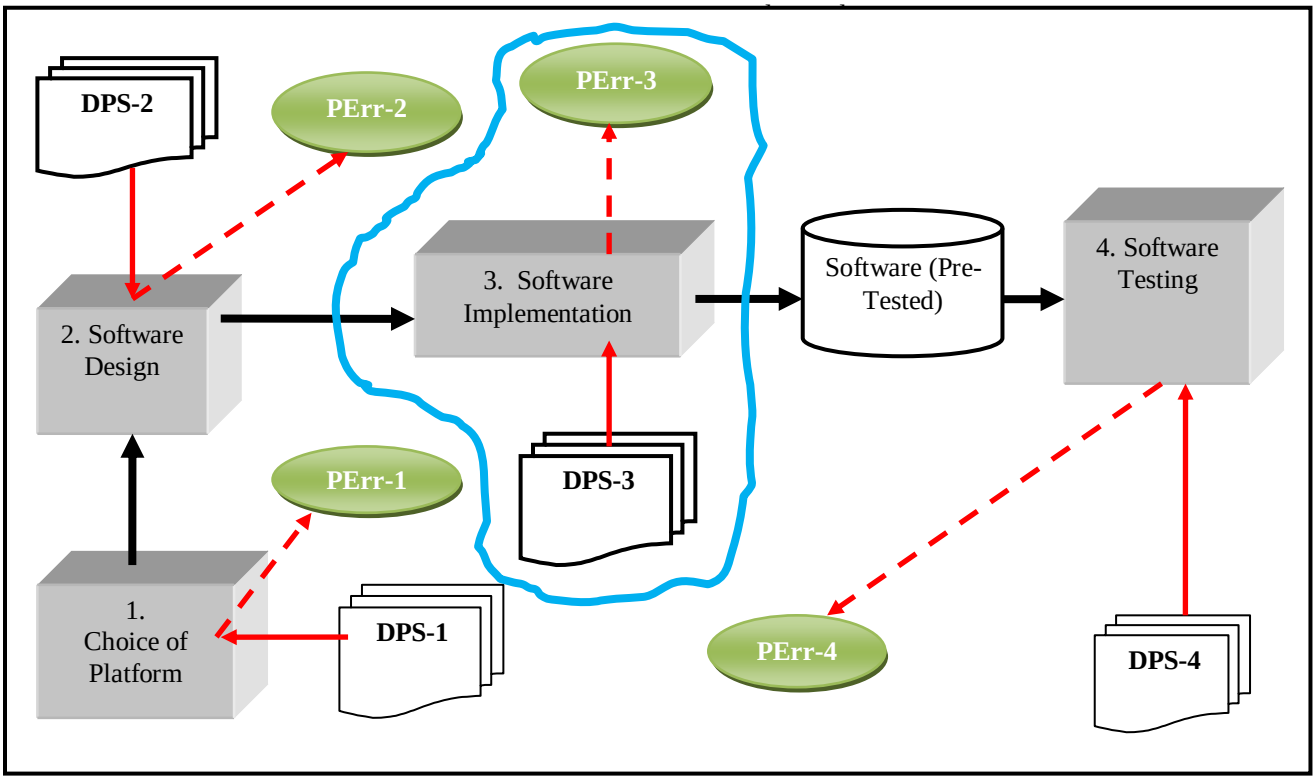

Fig. 1F: General Defensive Programming Workflow

The cuboids shaped objects numbered as 1 to 4 represent some key stages or operations in a typical software development project where defensive programming could be applied. Thus, defensive programming should begin right from the choice of a programming platform. This is because no two programming platforms are exactly the same in all

In other words, PErr-3 is a collection of errors that could possibly be prevented as a result of the application of defensive programming strategies DPS-3. As shown in the diagram, the PErr objects are linked to the cube-shaped objects by red coloured broken arrows. The broken arrows signify that a particular preventable error PErr-x could be

mitigated through the implementation of a particular defensive strategy DSP-x, where x = 1, 2, … as earlier explained.

Some of the defensive programming approaches applied by the past researchers were mentioned in the literature survey section of this work. An exhaustive listing of all the possible preventable errors and their corresponding defensive programming strategies may not be possible in a single research paper of this capacity. However, the focus of this work is on the necessity for defensive programming in multi-path decisions. This falls within the system implementation stage, and has been indicated in the workflow by grouping the objects set {PErr-3, DPS-3, Software Implementation} using a blue marker.

IV. MULTI-PATH PROGRAMMING DECISION

At this stage, a more elaborate discussion on the multi-path *error propagation* will be appropriate. Such an exploration will go a long way to underline the importance of defensive programming, particularly in reducing the possibility of avoidable errors in software projects. A multi-path propagation error usually originates from a multi-path programming decision.

This is a class of programming decision which presents several entry and exit paths. The decision is triggered by an input parameter, which determines which section of the program where system control will be transferred to. Usually, the input parameter is followed by a number of pre-decision paths, and then an equal number of post decision paths. No doubt, a multi-path programming decision requires defensive programming checks.

This is because a faulty decision could lead to the propagation of errors to other sections of software. The resultant effect could be an erroneous software project. This scenario is of serious concern since the propagated error is not necessarily syntax related, and therefore may not be spotted by the debugging and *software testing* process. A typical multi-path programming decision studied in this research is the Case-Switch construct. Fig. 2F, 3F and 4F show the syntax rules for implementing the Case-Switch in three different programming languages. The first illustration is for Java [19, 20], the second is for MATLAB [21], while the third is for C/C++ [22, 23] respectively.

V. THE MULI-PATH COMPONENTS

The components of each of the Switch-Case constructs in Fig. 2F, 3F and 4F are summarized in Table I. Based on the table, the set of input parameters {J-Expr, M-Expression, and C-Expression} were used for Java, Matlab and C/C++ platforms respectively.

For every Case-Switch execution, the system has multiple paths from where to choose, depending on the value of the input parameter. For instance, in the case of Java, the system follows the second path if the input parameter is Value2.

JAVA SYNTAX

```
switch (J-Expr)
        {
    case Value1:
      J-Statements-1;
           break;
    case Value2:
      J- Statements-1;
           break;
            ...
           ….
 Default:
     J-Statements –N;
        }
```

Fig. 2F: Switch – Case Construct for Java

MATLAB SYNTAX

```
switch M-Expression
 % expr  is scalar or string
     case Option1
           M-Stm -1
     case Option2
           M-Stm -2
           ……      .
           ........
     otherwise
           M-Stm -N
     end
```

Fig. 3F: Switch – Case Construct for MATLAB

C/C++ SYNTAX

```
switch (C-Expression)
                {
 case Label1 : C-statement-1

 case Label2 : C-statement-2
              ……
             .........

 default : C-statement-N
                }
```

Fig. 4F: Switch – Case Construct for C/C++.

Similarly, a Matlab program would follow the second path if the input parameter is Option2. Also, a C/C++ program would follow the second path, if the input parameter is *Label2*.

TABLE I MULTI-PATH TABLE

Parameters	Programming Languages		
	Java	*Matlab*	*C+++*
Input Parameter	J-Expr	M-Expression	C-Expression
First path	Value1	Option1	Label1
Second Path	Value2	Option2	Label2
Third Path	Value3	Option3	Label3
……..	……	……	……
Path	Default	Otherwise	Default

The problem of *error propagation* is that system control is always transferred to a designated path the moment a multi-path error occurs. This is usually the last path in every multi-path construct. Thereafter, the program execution still continues. The implication is that even if a wrong input parameter was used in a multi-path construct, the system still goes ahead to perform all the computations defined in the last path. This path is usually referred to as the 'default' path. Thus, if care is not taken, the errors which may arise from the execution of the default lines of code could be propagated into the entire system.

VI. MULI-PATH ERROR PROPAGATION

A pictorial illustration of how a multi-path error is propagated in many high level programming languages is shown in Fig. 5F. Based on the diagram, the input parameter is shown as InParam. Furthermore, the set {*EPath1, EPath2, EPath3 … EpathN*} constitutes the entry paths into the multi-path decision function (MPDF). The decision function represents the inbuilt Case-Switch construct of the high level programming language in question.

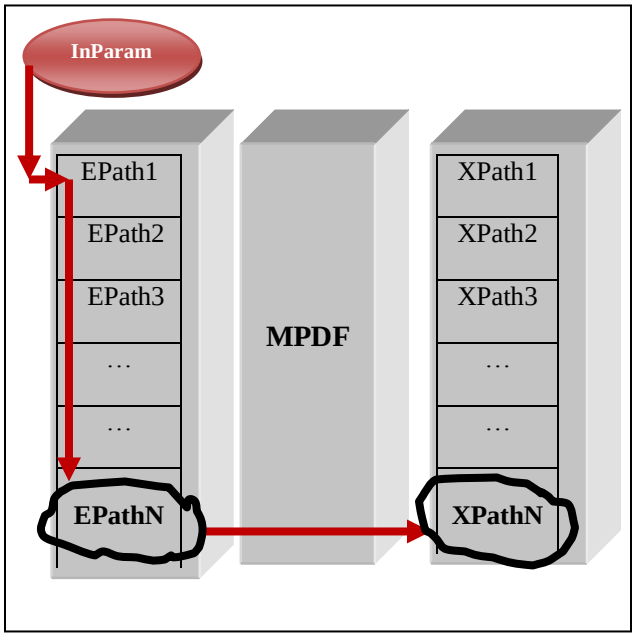

Fig. 5F: Multi-Path Decision Diagram

The set {*XPath1, XPath2, XPath3… XpathN*} constitutes the exit paths. It is through the exit paths that errors are propagated. As depicted in the diagram, system always passes program control through *EPathN* to *XPathN* whenever there is an error with the input parameter.

This *error propagation* study focused on the three programming languages – Java, Matlab and C/C++. A practical demonstration was made using the Matlab source code named *SwitchProg.m*, which is listed in Fig. 6F.

As shown in the figure, a variable known as *testVar* is used to pass an input into the multi-path programming construct. During program execution, the user is prompted with a narration "Input Possible Values [1-4] : ". The program output is however far from the user expectation as will be analyzed in the following section. Defensive programming is therefore necessary in order to avoid errors of this type.

VII. OUTPUT ANALYSIS AND IMPLICATIONS

The program execution screen is displayed in Fig. 7F. As clearly shown, the user entered the input value '3', with a possible expectation that the system would execute the third option in the Case-Switch construct. However instead of following the user expected path, the system executed the fifth option in the construct.

A careful analysis was done in order to unravel the reason for the disparity. It was found that though the user entered '3' with numeric value in mind, the Case-Switch was however programmed to be driven by non-numeric values. Due to this lack of synchronization between the input data type and the Case-Switch decision data types, control was automatically switched to the default path. Thus, instead of the third option, the fifth was executed.

The implication is that a serious error could be propagated into the system as a result of such a disparity. It is therefore advisable that a programmer should apply defensive programming at the point of entry to a Case-Switch construct. This is necessary in order to avoid multi-path *error propagation* into the larger sections of the program.

```
%Demonstrating Multi-Path Error propagation
%based on Case-Switch Construct.
testVar=input('Input Possible Values [1-4]  :')
switch testVar
        case '1'
            disp 'Value of Program Input is ONE'
        case '2'
            disp 'Value of Program Input is TWO'
        case '3'
             disp 'Value of Program Input is THREE'
        case '4'
            disp 'Value of Program Input is FOUR'
        otherwise
            disp 'Value of Program Input is FIVE'
end
```

Fig. 6F: Multi-Path Sample Code in Matlab

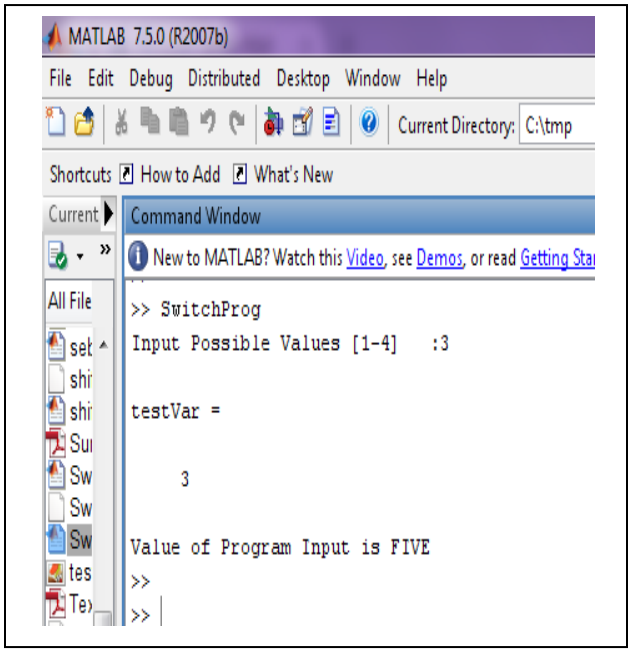

Fig. 7F: Multi-Path Program Output in Matlab

VIII. SOLUTION STRATEGIES

This research recommends two major defensive programming strategies for tackling the multi-path *error propagation* problem. These are the Programmer Double Checklist and Input Data Certification respectively.

A. *Programmer Double Checklist*

This is a procedural solution-based defensive programming strategy [24]. Here the programmer maintains what is termed as a 'double checklist' [25]. This is a list containing the major parameters or variables that a programmer wishes to cross check (or double check) for correctness before assuming that a software project has been completed. The importance of this procedural solution is that it acts as a guide or a reminder to the programmer that nothing should be left undone.

A double checklist could be kept in form of a spreadsheet [26] such as Microsoft Excel. Alternatively, a physical notebook could also be used. In a typical software project, a programmer could review the checklist to ensure that the data type specified in each of the Case-Switch paths coincide with those of the corresponding input parameters. The process of building the double checklist as well as performing the actual review may or may not be automated. However, this procedural strategy could save the system from a lot of programming errors.

B. *Input Data Certification*

Another defensive programming strategy that could be applied to deal with multi-path *error propagation* is data certification [27]. Here, the programmer applies requisite data inquiries and type casting before utilizing a particular data structure or variable in a program. Data type casting [28] involves the conversion from one data type to another within a source code. Fig. 8F outlines three possible applicable areas for data certification. These are at the input, pre-processing and pre-output formatting stages of a computer program.

It is important to carry out a simple verification to ascertain the data type of the user inputs. This is based on the obvious fact that the processing defined on any given data will depend on its data type. For instance, a user may be required by a computer application to enter a value designated as *BudgetAmount* which is supposed to be a real number. A user interface that is not intelligent enough may accept string values even when it is wrong to do so. Through defensive programming, a professional programmer thinks out of the box, and ensures that extra checks are done at the point of entry [29]. As shown in Fig.8F, such defensive checks can also be done before attempting to process a given data or attempting to format the outputs from a set of computer processing operations.

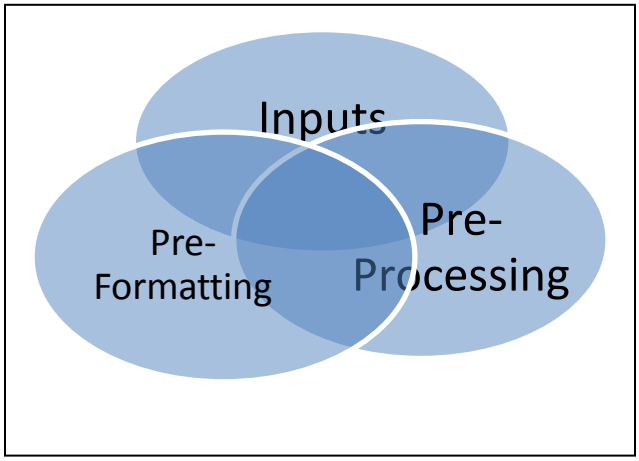

Fig.8F: Certification Areas in Defensive Programming

In a number of programming languages, data certification operations come as inbuilt programming commands which the programmer invokes at the appropriate stage. Let the symbol DCert signify a general data certification operation. The essence of DCert in a programming construct is mainly to carry out double checks in order to avoid data type-related programming errors. The data certification commands are usually Boolean in nature [30]. The general syntax of a typical data certification operation is shown in equation (1).

$$DCert\ (x) = \begin{cases} YES, & If\ x\ is\ a\ data\ type \\ NO, & If\ it\ is\ otherwise \end{cases} \qquad (1)$$

where *DCert* = a data certification command and x is a named data variable.

For instance, a typical DCert could determine if a programming data is a real number, a character, an integer, a logical data, and so on. MATLAB implements *DCert* based on a set of logical keywords {isfloat, ischar, isinteger,

islogical, …}. The multi-path error propagation problem could therefore be solved by incorporating requisite data certification commands as part of the program decision process. In other words, a programmer could invoke the keyword 'isfloat' to determine if the data used as input is a real number or not [31].

IX. CONCLUSION/ FUTURE RESEARCH

The importance of defensive programming cannot be overemphasized. A number of past researches in defensive programming were reviewed in order to establish their link with the current work. This research focused on the issue of multi-path *error propagation* in software programming. Effort was made to trace the origin of the problem particularly in the Case-Switch construct of some high level programming languages.

The scope if this work covered the application of relevant defensive programming techniques to mitigate multi-path errors in modern programming. Practical illustrations were based on Java, Matlab and C/C++. The solution strategies for tackling this issue were also presented. The key benefit of this research is that it attempts to evolve proactive strategies for dealing with computer programming errors. This is an important goal in software engineering since attempting to debug a system after its release could be very costly [32]. Therefore any intellectual effort geared towards discovering and implementing *error-avoidance* strategies will no doubt be of immense benefit to the field of computing. A number of future enhancements are proposed. One area for future research is a full automation of defensive programming checks. Such a full automation will make it possible for system to enhance defensive programming with little human intervention.

REFERENCES

[1] F. Preparata. "The unpredictable deviousness of models", Theoretical Computer Science Vol. 408, 2008, p99-105

[2] F. Schindler. "Coping with Security in Programming", Acta Polytechnica Hungarica Journal, Vol. 3, No. 2, 2006, p65-72.

[3] D. R. Sahu, D. S. Tomar. "Defensive Programming to Reduce PHP Vulnerabilities", International Journal of Advances in Computer Networks and Its Security– IJCNS, Vol. 4: Issue 2, June 2014, p71-75

[4] I. Sommerville. "Sofware Engineering, 9th Ed", Pearson Education, Inc., Massachusetts USA, 2011

[5] D. Bell. "Software Engineering for Students: A Programming Approach", Prentice Hall International, Essex England, 2005.

[6] I. Marsic. "Software Engineering", Department of Electrical and Computer Engineering, Rutgers University, New Jersey, 2012.

[7] K. Muslu, Y. Brun, R. Holmes, M. Ernst and D. Notkin. "Speculative Analysis of Integrated Development Environment Recommendations", OOPSLA'12, Tucson, Arizona, USA, October 19–26, 2012, p669-682.

[8] G. Costagliola, V. Deufemia and G. Polese. "Towards Syntax-Aware Editors for Visual Languages", Electronic Notes in Theoretical Computer Science Vol. 127, 2005, p107–125.

[9] Y. Lilis and A. Savidis. "An Integrated Approach to Source Level Debugging and Compile Error Reporting in Metaprograms", Journal of Object Technology, Vol. 12, No. 3, 2013, p1–26.

[10] Y. Liu, K. Liu and M. Li. "Passive Diagnosis for Wireless Sensor Networks", IEEE/ACM Transactions on Networking, Vol. 18, No. Aug. 2010, p1132-1144

[11] B. Rick, T. Mohiuddin, and M. Nawrocki. "LabVIEW Advanced Programming Techniques", Boca Raton: CRC Press LLC, 2001

[12] J. Swan. "Defensive C++ Programming Guidelines for those who dislike Debugging", Dept. of Computing Science and Maths, Univ. of Stirling, Scotland, Nov. 2012

[13] T. Powell and F. Schneider. "JavaScript 2.0-The Complete Reference, 2nd Ed", McGraw-Hill/Osborne, New York, 2004.

[14] A. Kuznetsov. "Defensive Database Programming with SQL Server", Simple Talk Publishing, 2010

[15] B. Shehu, A. Xhuvani, and S. Ahmeta. "Methods of Identifying and Preventing SQL Attacks", IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 6, No 2, November 2012, p403-406

[16] V. Kumar and J. Das. "Advanced Detecting and Defensive Coding Techniques to prevent SQLIAs in Web Applications: A Survey", International Journal of Science and Modern Engineering, Vol. 1, Issue-6, May 2013, p26-31

[17] D. Foulger and S. King. "Using the SPARK Toolset for Showing the Absence of Run-Time Errors in Safety-Critical Software", Ada-Europe 2001, p229-240.

[18] J. C. Demay, F. Majorczyk, E. Totel and F. Tronel. "Detecting illegal system calls using a data-oriented detection model", IFIP SEC'2011, Jun 2011, Lucerne, Switzerland, p12

[19] J. Farrell. "Java Programming: Fifth Edition", Course Technology Cangage Learning, Boston USA, 2010

[20] H. Schildt. "Java™:The Complete Reference, Seventh Edition", The McGraw-Hill Companies, 2007

[21] S. R. Otto, J. P. Denier, "An Introduction to Programming and Numerical Methods in MATLAB", Springer-Verlag London Limited, London, 2005

[22] S. Prata. "C++ Primer Plus: Sixth Edition", Pearson Education, Inc., New York, 2012

[23] B. Jones and P. Aitken. "SAM's Teach yourself C in 21 Days", Sams Publishing, Indianapolis USA, 2003.

[24] I. Parberry. "Designer Worlds: Procedural Generation of Infinite Terrain from Real-World Elevation Data", Journal of Computer Graphics Techniques Vol. 3, No. 1, 2014, p74-85

[25] A. Jones, R. K. Stephens, R. R. Plew, R. F. Garrett and A. Kriegel. "SQL Functions Programmer's Reference", Wiley Publishing, Inc., Indianapolis, 2005

[26] S. Powell, K. Baker and B. Lawson. "Errors in Operational Spreadsheets", Journal of Organizational and End User Computing, Vol. 21, No. 3, 24-36, Jul-Sep. 2009

[27] M. Fitzpatrick and J. D. Crocetti. "Introduction to Programming with Matlab", Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, 2010.

[28] D. Liang. "Introduction to Java Programming, 8Ed.", Prentice Hall Publishers, New York, 2011

[29] J. Castagnetto, Ha. Rawat, S. Schumann, C. Scollo and D. Veliath. "Professional PHP Programming", Wrox Press Ltd,Birmingham UK, 1999

[30] D. Radojevi. "[0,1] - Valued Logic: A Natural Generalization of Boolean Logic", Yugoslav Journal of Operations Research, Vol. 10, No. 2, 2000, p185-216

[31] MathWork. "MATLAB:The Language of Technical Computing", Function Reference Volume 1: A - E, Version 7, The MathWorks, Inc., Natick, 2006.

[32] C. Liu, L. Fei, X. Yan, J. Han and S. Midkiff. "Statistical Debugging: A Hypothesis Testing-Based Approach", IEEE Transactions on Software Engineering, Vol. 32, No. 10, Oct 2006, p1-17