# Re-tooling Code Structure Based Analysis with Model-Driven Program Slicing for Software Maintenance

Oladipo Onaolapo Francisca (PhD)
Computer Science Department,
Nnamdi Azikiwe University
Awka, Nigeria

*Abstract*—**Static code analysis is a methodology of detecting errors in program code based on the programmer's reviewing the code in areas within the program text where errors are likely to be found and since the process considers all syntactic program paths; there is the need for a model-based approach with slicing. This paper presented a model of high-level abstraction of code structure analysis for a large component based software system. The work leveraged on the most important advantage of static code structure analysis in re-tooling software maintenance for a developing economy. A program slicing technique was defined and deployed to partition the source text to manageable fragments to aid in the analysis and statecharts were deployed as visual formalism for viewing the dynamic slices. The resulting model was a high-tech static analysis process aimed at determining and confirming the expected behaviour of a software system using slices of the source text presented in the statecharts.**

***Keywords- software maintenance; static analysis; syntactic program behavior; program slicing.***

## I. INTRODUCTION

There had been a growing use of static analysis both in commercial and academic areas in the verification of properties of software used and locating potentially vulnerable code in critical sensitive computer systems [1]. Static code analysis is the analysis of computer software that is performed without actually executing programs built from that software. This is in contrast to the analysis performed on executing programs which is known as dynamic analysis [2]. Empirical evidence from [3] and [4] showed that software maintenance consumed between 50% and 80% of the resources in the total software budget. In addition, according to [5] and [6]; an estimated 50% to 80% of the time and material involved in software development is devoted to maintenance of existing code, hence the main justification for this work was to leverage on the most important advantage of static code structure analysis in re-tooling software maintenance for a developing economy.

The most important advantage of static code analysis lied in the possibility of considerable cost saving by defects elimination in a program; this is expected to bring about some economic benefits to a developing country as technological innovations were known to do and they can be expected to save considerable costs in software maintenance. The earlier an error is determined; the lower the cost of its correction. According to [7], correction of an error at the testing stage is ten times more expensive than its correction at the construction (coding) stage.

This paper presented a model for code structure analysis for a large component based software system. A program slicing technique was deployed to partition the code to manageable fragments to aid in the analysis and statecharts were deployed as visual formalism to view the dynamism of the static slices. The model aimed at determining and confirming the expected behavior of a software system using the source text because research had shown that during maintenance, the most reliable and accurate description of the actual behavior of a software system is its source code [8]. The rest of this paper is organized as follows: A background to the concepts of static codes structure analysis and program slicing was presented in section II; section 3 described the materials and methods adopted in the research, the resultant models were discussed in section 4 and the section 5 concluded the paper.

## II. RESEARCH BACKGROUND

Identifying errors in software during development is very important so that the end product can be error free and perform to its specification. Reference [9] believed that early identification of bugs in a developing program can be achieved through the concept of program slicing. Generally, program slice has a wider spectrum of applications that include debugging, testing, maintenance, code understanding, complexity measurement, security etc.

Static analysis is any form of analysis that does not require a system to be operated. The process complements dynamic analysis, where system operation is central. When applied to code, static analysis is typically referred to as white-box, glass-box, structural or implementation based techniques [10]. In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is sometimes applied to the analysis performed by an automated tool, with human analysis being called program understanding, program comprehension or code review. Static Analysis is performed without program execution and the process includes almost everything except conventional testing. This is because dynamic testing requires running code, some program properties such as race conditions are too hard to test for and though it may be impossible to program correctness, one can easily prove simple properties of simplified models. Static analysis can be applied earlier in development because some kinds of defects are hard to find by testing (e.g., timing-dependent errors) and because testing and analysis are complementary; each is best at finding different faults. The sophistication of the analysis performed by tools varies from those that only consider the behavior of individual statements and declarations, to those that include the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors to formal methods that mathematically prove properties about a given program [11].

One implementation technique of formal static analysis is model checking; it includes the consideration of systems that have finite state or may be reduced to finite state by abstraction. Data-flow analysis is a lattice-based static analysis technique for gathering information about the possible set of values and the abstract interpretation models the effect that every statement has on the state of an abstract machine. In this case, the model 'executes' the software based on the mathematical properties of each statement and declaration. This abstract machine over-approximates the behaviours of the system and the abstract system is thus made simpler to analyze, at the expense of incompleteness since not every property true of the original system is true of the abstract system. If properly done, though, abstract interpretation is sound as every property true of the abstract system can be mapped to a true property of the original system [12].

The sophistication of the analysis performed by the model varies from those that only consider the behavior of individual statements and declarations, to those that include the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors to formal methods that mathematically prove properties about a given program [13]. Static analysis can find weaknesses in the code at the exact location, it can be conducted by trained software assurance developers who fully understand the code and it allows a quicker turn around for fixes. In addition to all these, it permits weaknesses to be found earlier in the development life cycle, reducing the cost to fix.

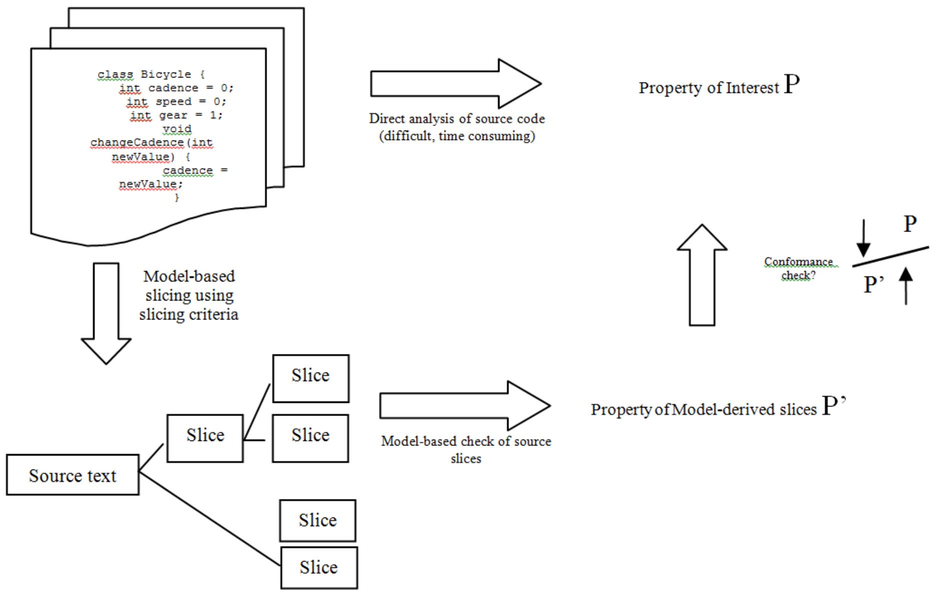
Program slicing was initially proposed to guide programmers during program debugging, but had been found to be useful for the process of understanding programs. Dynamic slicing was used to identify those parts of the program that contributed to the computation of the selected function for a given program execution. This can be used to understand program execution by adopting a commonly used high level abstraction. Program slicing is the computation of the set of programs statements, the program slice that may affect the values at some point of interest, referred to as a slicing criterion. Program slicing can be used in debugging to locate source of errors more easily. Other applications of slicing include software maintenance, optimization, program analysis, and information flow control. [13]. Slicing techniques have been seeing a rapid development since the original definition by Mark Weiser. At first, slicing was only static, i.e., applied on the source code with no other information than the source code. Reference [14] introduced dynamic slicing, which worked on a specific execution of the program; for a given execution trace. Based on the original definition of [15], informally, a static program slice S consists of all statements in program P that may affect the value of variable v at some point p. The slice is defined for a slicing criterion C=(x,V), where x is a statement in program P and V is a subset of variables in P. A static slice includes all the statements that affect variable v for a set of all possible inputs at the point of interest (i.e., at the statement x). Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies. Dynamic slicing techniques provided a means to prune unrelated computation, and it may help to narrow down this part of a program that contributed to the computation of a function of interest for a particular program input.

## III. Materials and Methods

The methodology adopted in the work was a modification of the Jakstab framework [16]. Jakstab is an Abstract interpretation-based, integrated disassembly and static analysis framework for designing analyses on executables and recovering reliable control flow graphs. In order to make the framework suitable for the research in this paper, the author added an extra layer of abstraction to the original framework to obtain a modified methodology suitable for the task at hand. While the starting point for the Jakstab framework is binary source, the approach in this paper performs the analysis on the program source code.

In addition to the modified Jakstab framework; the following materials were deployed in building the model-based high-tech source analysis system.

- A bottom-up dynamic slicing technique was defined in this work and deployed to obtain a hybrid-tech static analysis model (Fig. 1). A slice was constituted by an executable portion of the original program whose behavior is, under the same input, indistinguishable from that of the original program on a given variable ‘V’ at point ‘P’ in the program. Reference [13] had showed that bottom-up program slicing techniques could be successfully deployed to transform a large component-based program into a smaller one that contains only statements relevant to the computation of a given function.
- Statecharts notations used in [17] were deployed as a concise visual formalism that captured the dynamic behaviour of a system in representing program slices in this work. Illustrated below is a visualization of a login module Passwd.pas (Fig. 2).
- Goal models- a graph structure representing stakeholder goals and their inter-dependencies was deployed to decompose goals into sub-goals through AND/OR refinements (Fig. 3).



```
class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;
    void
changeCadence(int
  newValue) {
        cadence =
  newValue;
    }
```

Direct analysis of source code (difficult, time consuming)
Property of Interest P
Model-based slicing using slicing criteria
Conformance check?
P
P'
Slice
Slice
Slice
Slice
Slice
Slice
Source text
Model-based check of source slices
Property of Model-derived slices P'

Figure 1. Source model-based slicing for static analysis

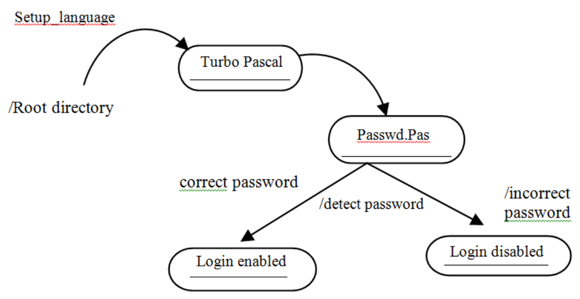
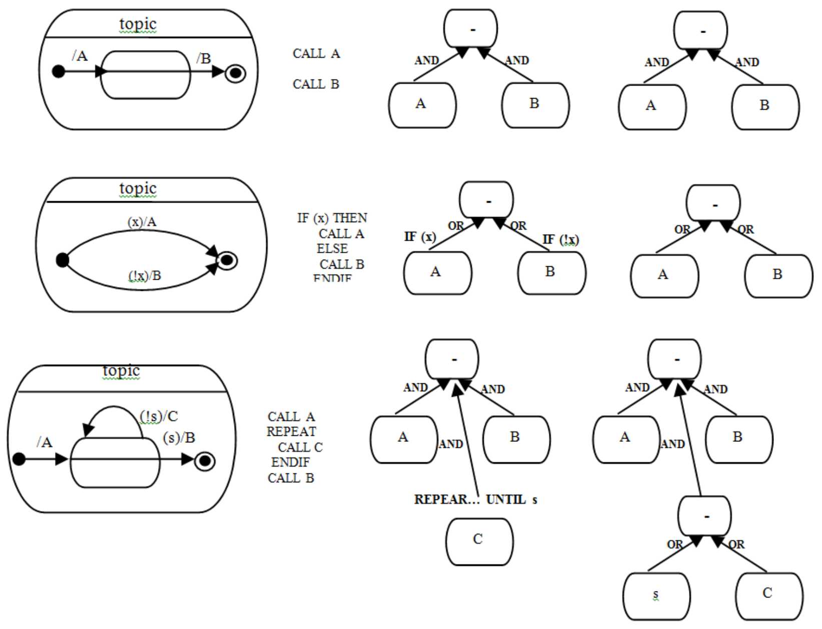


Figure 2. Sample statechart notation



```
CALL A
CALL B
```

```
IF (x) THEN
    CALL A
    ELSE
    CALL B
    ENDIF
```

```
CALL A
REPEAT
   CALL C
 ENDIF
CALL B
```

Figure 3. Patterns to extract goal models from abstract code [17]

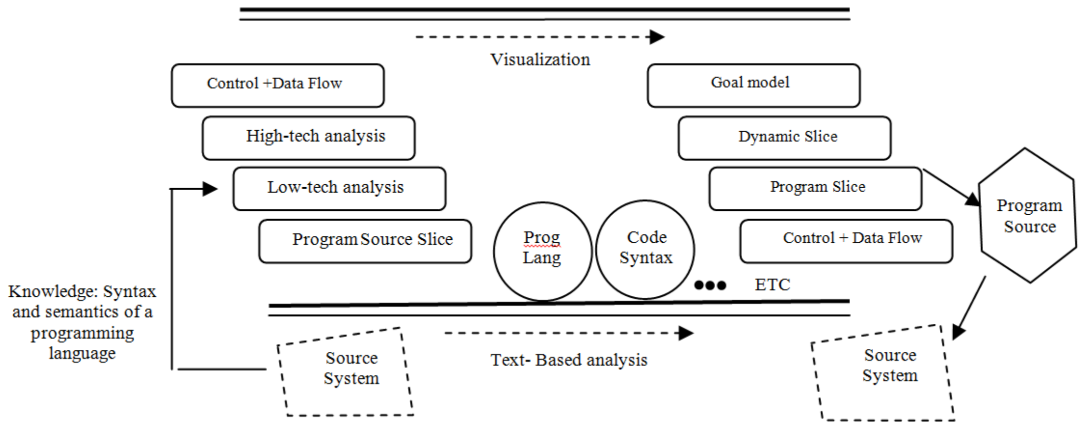


Figure 4. The model-based high-tech source analysis system

## IV. Results and Discussions

This work described a model-based high-tech methodology for static source analysis which consisted of systematically using slices of code source models as primary source artifacts throughout the analysis process (Fig.1). This was because, though proving program correctness may be wrong at static analysis phase, it is possible to prove simple properties of simplified models. The process took as input, the program source text and generated slices based on slicing criterion.

The knowledge schema at this point comprised of the knowledge of the syntax and semantics of the programming language (Fig.4). Low-tech static analysis that involved simple software inspection and manual checking of simple syntactic standards were first carried out. This was followed by the high-tech static analysis that included enforced syntactic checks, check for conformance with respect to specifications, designs, and the code data flow analysis. The model was bottom-up and generated slices were analyzed, refined if necessary and evaluated. The sum of all analyzed parts (slices) gave the whole (entire source) at the end of the day which may eventually become a candidate for analysis at a later time. Generated slices were represented by statecharts and resolved into goal models.

The Static analysis model was applied to a real-life legacy application [13]. A procedural application developed for a commercial bank in Nigeria prior to the consolidation of the banking sector in the country was chosen due to its high availability and statecharts were deployed to show the abstract description of the behaviour of the source system. Fig. 5 showed the statechart that implemented the login page of the application, passwd.pas.

Visualizations like the one above (Fig. 5) were built for different slices of the application and the different visualizations were combined to obtain a high-level meta-model that contained the entire description of the original system (Fig. 6).

The top-level statechart above was converted to an annotated goal model using the conversion process described earlier (Fig. 3), all the transitions were converted into goals using the AND/OR decomposition rules. Some tasks in the goal model contributed to quality concerns modeled by the softgoals; for example, “correctPassword” contributed to the security concern while “ErrorMessage” contributed to the usability concern (Fig.7).

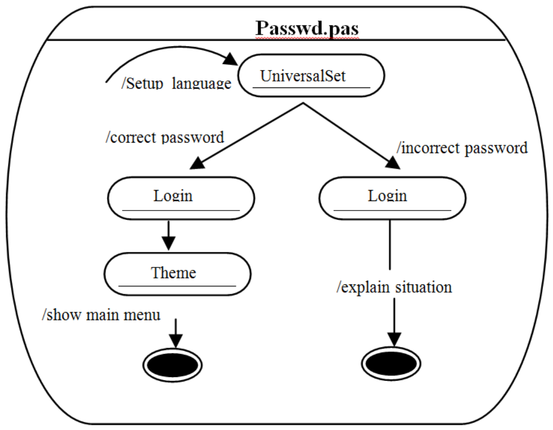


Figure 5. Sample visualization of the login module using Statechart

## V. Conclusion

Previous research had showed that the maintenance of existing software source code consumed up to ¾ of the resources in the total software budget (time and material), and that the earlier an error is determined, the lower is the cost of its correction. In addition, [18] opined that code comprehension require 47% and 62% of the total time for enhancement and correction tasks, respectively. The main justification for this work therefore is to leverage on the most important advantage of static code structure analysis in re-tooling software maintenance for a developing economy- cost saving. In this work, a framework for code structure analysis for a large component based software system with program slicing was developed as a re-tooling technique for developing economies in order to enable an early detection of bugs during a software development process thereby saving significant costs in software maintenance.

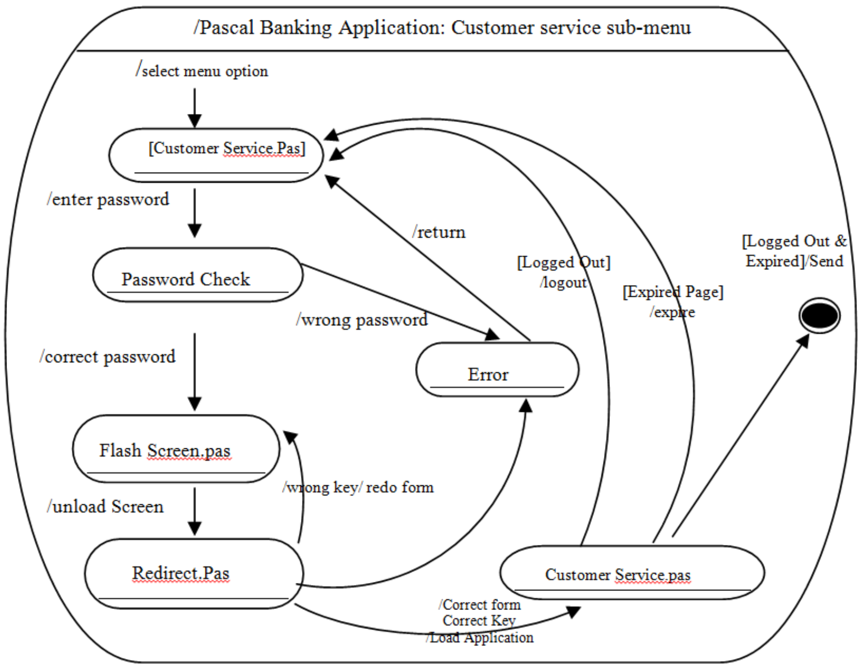


Figure 6. Sample top-level Statechart of the Procedural application

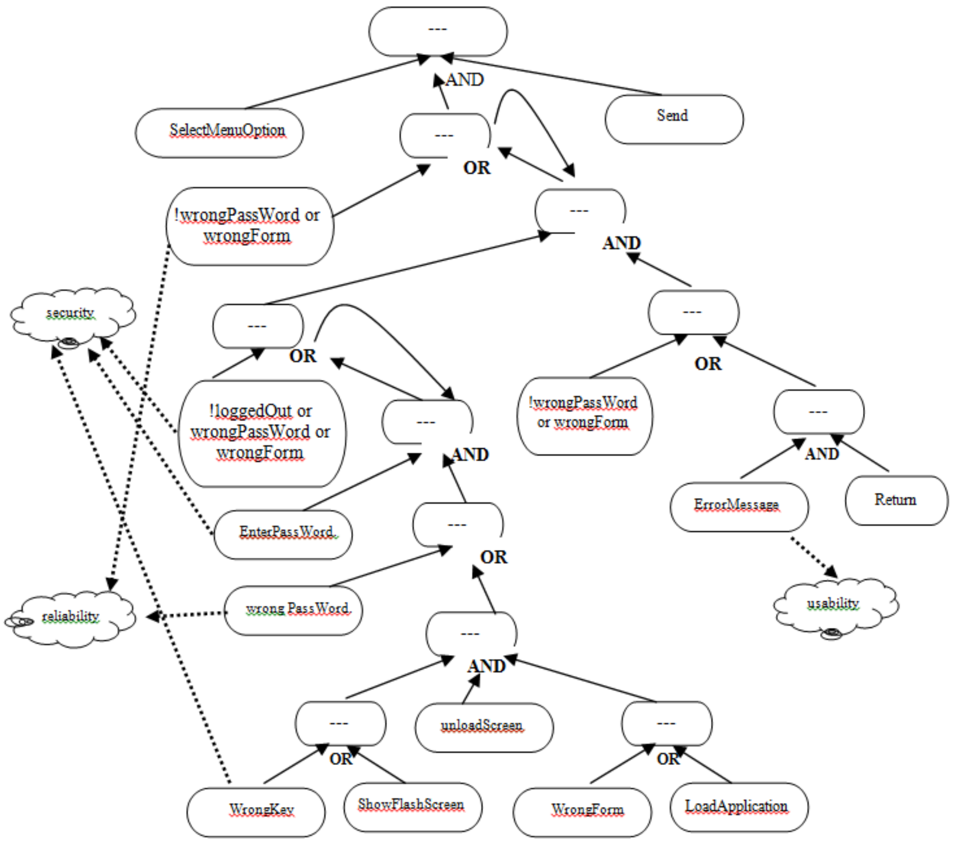


Figure 7. Sample goal model for the application [13]

## REFERNCES

[1] B. Livshits, “Improving Software Security with Precise Static and Runtime Analysis” Section 7.3 “Static Techniques for Security,” Stanford doctoral thesis, 2006. http://research.microsoft.com/en-us/um/people/livshits/papers/pdf/thesis.pdf

[2] B. A.Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsbarrow, N. J. Ward, and D. W. R. Marsh, “Industrial Perspective on Static Analysis,” Software Engineering Journal vol 10, pp. 69-75, March, 1995.

[3] B. W. Boehm, “Software engineering economics,” Prentice-Hall, Englewood Cliffs, NH, 1981.

[4] C. McClure, “The three Rs of software automation,” Prentical Hall, Englewood Cliffs, NJ, 1992.

[5] A.F. Ackerman, L. S. Buchwald, and F. H. Lewski. “Software Inspections: An Effective Verification Process,” IEEE Software, Vol. 6, No. 3, May 1989, pp. 31-36.

[6] K. Erdos, & H. M. Sneed, "Partial Comprehension of Complex Programs enough to perform maintenance," Proceedings of the IEEE Sixth International Workshop on Program Comprehension, June 24 – 26, 1998.

[7] S. McConnell, Code Complete, 2nd ed., Microsoft Press: Paperback, 2004, 914 pages, ISBN: 0-7356-1967-0.

[8] R. Klosch, “Reverse Engineering: Why and How to Reverse Engineer Software”, Proceedings of the International Conference on Software Engineering, 2001, pp. 123-132.

[9] K. Thiagarajan, C.Saravanakumar, G. Poonkuzhali, Ponnammal Natarajan, and S.Jeyabharathi, “Static program slicing for composite data using FSM-Model,” World Academy of Science, Engineering and Technology Journal, Issue 32 Aug. 2009, pp. 820-824. Downloaded December 2011 from http://www.waset.org/journals/waset/v32.php

[10] A. Ireland, “Static Analysis Techniques,” Lecture notes on F28SD2: “Software Design”, School of Mathematical and Computer Science, Heriot-Watt University, Edinburgh

[11] Wikipedia the free encyclopedia, “Static Program Analysis,” Downloaded November 2011 from http://en.wikipedia.org/wiki/Static_program_analysis

[12] P. Jones, "A formal methods-based verification approach to medical device software analysis". Journal of Embedded Systems Design, 2010.

[13] O.F. Oladipo, “Software reverse engineering of legacy applications,” Ph.D. Dissertation Computer Science Department, Nnamdi Azikiwe University, Awka Nigeria, March 2010, unpublished

[14] B. Korel and J. Laski. “Dynamic program slicing,” Information Processing. Letters, vol. 29, no 3, pp.155-163, Oct. 1988.

[15] M. Weiser. "Program slicing". IEEE Transactions on Software Engineering, vol. 10, Issue 4, pages 352–357, IEEE Computer Society Press, July 1984.

[16] Jakstab Framework homepage http://www.jakstab.org/

[17] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. Cesar Sampaio do Prado Leite, “Reverse Engineering Goal Models from Legacy Code,” In: 13th IEEE International Conference on Requirements Engineering (RE'05), 29 Aug-2 Sept 2005, Paris, France, Downloaded October 2009 from www.cs.toronto.edu/~alexei/pub/re05re.pdf

[18] M. L. Nelson, “A Survey of Reverse Engineering and Program Comprehension,” Lecture notes on CS 551: “Software Engineering Survey,” Old Dominion University, April 19, 1996, Downloaded November 2011 from arxiv.org/pdf/cs/0503068

## AUTHOR'S PROFILE

Oladipo, Onaolapo Francisca holds a Ph.D in Computer Science from Nnamdi Azikiwe University, Awka, Nigeria; where she is currently a faculty member. Her research interests spanned various areas of Computer Science and Applied Computing. She has published numerous papers detailing her research experiences in both local and international journals and presented research papers in a number of international conferences. She is also a reviewer for many international journals and conferences. She is a member of several professional and scientific associations both within Nigeria and beyond; they include the British Computer Society, Nigerian Computer Society, Computer Professionals (Regulatory Council) of Nigeria, the Global Internet Governance Academic Network (GigaNet), International Association of Computer Science and Information Technology (IACSIT ), the Internet Society (ISOC), Diplo Internet Governance Community and the Africa ICT Network.