Contents lists available at Crown Academic Publishing

International Journal of Engineering and Artificial Intelligence

Journal home page: http://www.ijeai.com

Software Bug Prediction using Tree-Based Approach

Hisham Abdullah Bin Ateya [1], Dr. Saeed Mohammed Saeed Banaeamoon [2]

[1] Department of Information Technology, Faculty of Engineering & Information Technology, Al-Rayan University, Hadhramout, Mukalla, Yemen

[2] Department of Computer Engineering, College of Engineering & Petroleum, College of Computers Information Technology, Hadhramout University, Yemen

Corresponding Author: Hisham Bin-Ateya hishamco_2007@hotmail.com

Original article

Received 26 March 2020, Accepted 10 April 2020, Available online 11 April 2020

ABSTRACT

In this paper a Smart Unit Tests are generated to identify potential bugs in the source code for the method under test. Abstract Syntax Tree (AST) is prepared and constructed for the method source code. After that the abstract syntax tree is constructed, the syntax tree is analyzed to find a set of constraints and values that help to predict the software bugs. Usually those constraints came in form of mathematical and logical expressions. The constraints are then solved to evaluate the data values which are trigger the exceptions and cause the bugs at runtime.

Keywords: Bug Prediction; Software Bugs; Software Testing; Software Quality; Test Automation; Unit Testing; White Box Testing

1. Introduction

In Software Engineering, the system development life cycle (SDLC) is the process for planning, creating, testing and deploying an information system. [1] There are usually six phases in this cycle: requirement analysis, design, development and testing, implementation, documentation and evaluation.

Software testing is plays a vital role in the SDLC, which provide stakeholders with information about the quality of the software product or service under test. [2] Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementations. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects), and verifying that the software product is fit for use. [3]

During the development of a program or software, a range of measures is taken to ensure that the program is tested prior to the release and distribution of the program. These measures are aimed at reducing the number of bugs in the program in order to improve the quality of the program. A bug in a source code program is an unintended state in the executing program that results in undesired behavior. Regardless of these measures, the program may still contain bugs.

Detecting software bugs are tricky and not an easy task to accomplish, there are two approaches to detect the software bugs before the developers can fix them: the first one named Static program analysis which is the process of analyze the computer software without executing programs. The second one named Dynamic program analysis which is the process to analyze the computer software by executing programs on real or virtual processor, in contrast with static program analysis, whereas the analysis performed without program execution. [4]

Software maintenance makes the corrective measures needed to fix software bugs after the bugs are reported by end users. Fixing the software bugs after deployment of the program hampers the usability of the deployed program and increases the cost of the software maintenance services. A better solution would be to detect and fix the software bugs prior to release of the program.

A bug-free software is the main reason to implement the software testing. Quality Assurance (QA) or Software Testing is crucial because it identifies errors/bugs from a system at the beginning. By considering the problems in the base helps to turn improvement in the quality of product and brings confidence in it.

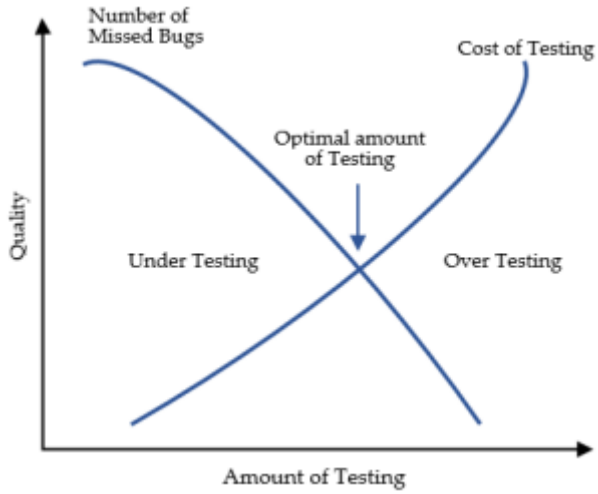

Figure 1. Software Testing & Quality Chart

According to the **Figure 1**, software testing is an important component of software quality assurance. The importance of testing can be considered from life-critical software (e.g., flight control) testing which can be highly expensive because of risk regarding schedule delays, cost overruns, or outright cancellation.

Hence the role of testing in having a good quality software, however the test measurement in the other hand allows better product quality. So, how the do you measure the test effectiveness? Counting number of lines covered by tests or taking a functional approach and see if all the features are covered.

2. Literature Review

(Floyd and Hoare, 0000), proposed a theorem proving which is based on the deductive logic [20, 21]. A program statement S is represented as a triple {p}S{q}, where p (precondition) and q (post-condition) are logical formulas over program states. This triple is valid iff for a state t satisfying formula p, executing S on t yields a state t′ which satisfies q. various inference rules are then used to verify system states.

(Floyd and Hoare, 0000), proposed a theorem proving which is based on the deductive logic [20, 21]. A program statement S is represented as a triple {p}S{q}, where p (precondition) and q (post-condition) are logical formulas over program states. This triple is valid iff for a state t satisfying formula p, executing S on t yields a state t′ which satisfies q. various inference rules are then used to verify system states.

(Cousot and Cousot, 0000), formalized abstract interpretation technique. It is a theory of semantics approximation of a program based on monotonic functions over ordered sets, especially lattices [16].

Abstract interpretation is a generic term for a family of static analysis techniques that includes both type systems and data flow analysis, among others. In abstract interpretation, a program's variables are assigned values from an abstract domain, and the program is executed on the basis of modified semantics for how each language construct applies in this new domain. For example, whereas an int variable may typically take on any concrete integer value in the range [-2^31, 2^31), in abstract interpretation the variable may be given a value of -, 0, +, or ? indicating only the sign of the variable (where "?" indicates an unknown or indeterminate value). Operators such as < are given meaning for this new domain, as shown in **Table 1**. Moving away from concrete values and operators allows automated analysis tools to evaluate programs' meanings in terms of the higher-level abstract domains. This ensures that the analysis will actually terminate on all input programs.

Table 1. Abstract interpretation rules for the < operator over the domain {-, 0, +, ?}

Input 1	Operator	Input 2	Is	Result
-	<	-	=	?
-	<	0	=	True
-	<	+	=	True
-	<	?	=	?
0	<	-	=	False
0	<	0	=	False
0	<	+	=	True
0	<	?	=	?
+	<	-	=	False
+	<	0	=	False
+	<	+	=	?
+	<	?	=	?
?	<	-	=	?
?	<	0	=	?
?	<	+	=	?
?	<	?	=	?

(Floyd and Hoare, 0000), proposed a theorem proving which is based on the deductive logic. A program statement S is represented as a triple {p}S{q}, where p (precondition) and q (post-condition) are logical formulas over program states. This triple is valid iff for a state t satisfying formula p, executing S on t yields a state t′ which satisfies q. various inference rules are then used to verify system states.

(Pacheco, C., Lahiri, S. K., Ernst, M. D., & Ball, T, 2007), Feedback Directed Random Testing (FDRT) FDRT approach is creating method sequences incrementally, and using the runtime information to guide the generation.

(Tao Xie, Nikolai Tillmann, Jonathan de Halleux and Wolfram Schulte, 2009), Program Exploration (PEX) is a white-box test generated tool as a part of Microsoft Research, which is my main inspiration for proposing this research.

PEX in nutshell using a Dynamic Symbolic Execution technique with fitness-guided path exploration. The Fitness Function (Fitnex) which was introduced in PEX as a key player in the entire Microsoft PEX Framework as search strategy to reduce the amount of exhaustive search required during the path exploration process.

(Patrice Godefroid, Nils Klarlund, Koushik Sen, 2011), Directed Automated Random Testing (DART) combines three techniques: automated interface extraction, automatic generation of a test driver for random testing and dynamic test generation to direct execution along alternative program paths.

3. Methodology

will introduces the proposal techniques for bug's prediction using Tree-based approach. At the beginning it will give an overview about the entire design for the proposed technique, then it will dig into the details for each phase of the overall design. After that it will discuss about a program analysis and execution techniques. Finally it will end up with path exploration and code coverage.

Based on all the problems and issues that mentioned previously, In this research is proposing a new approach for predicting the software bugs that may missed or occur for certain circumstances by making a static analysis for the source code of the methods that need to be tests, after constructing a model which is Abstract Syntax Trees in our case. A suite of unit tests will automatically generated, after exploring all the possible paths and finding out where are the places that lead the program to crash or raise a bug.

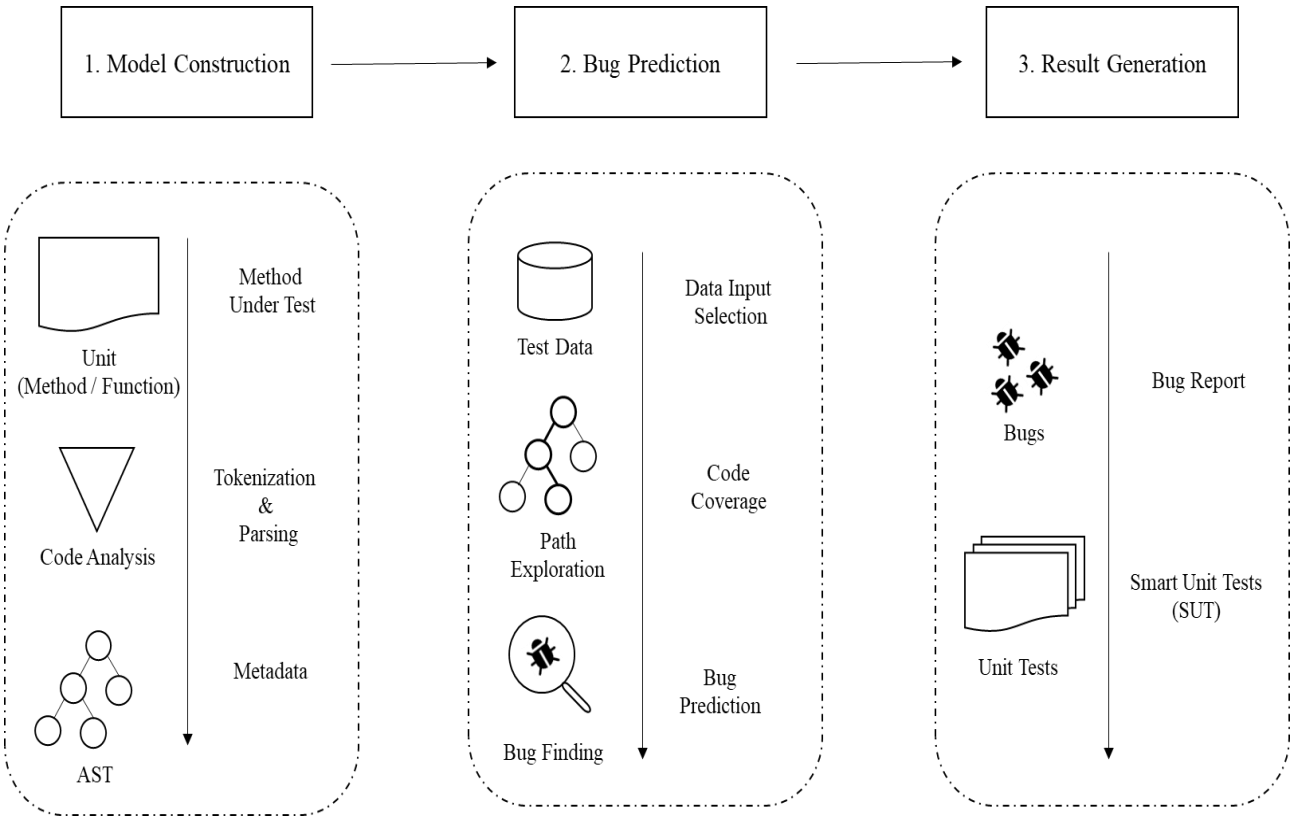

Figure 2. Overview of the Empirical Study Design

The empirical study design shown in Figure 4.1 is consist of three phases described as the following:

A. Model Construction

The first phase is concern about the model creation and construction, which is a very important process that assists the bug predictor for finding bugs.
The model is basically in form of tree representation – which is in our case AST - for the method under test.
Constructing an AST needs sort of compiler like tool to create the syntax tree for a given source code.
Analyze certain program code has two essential operations in case to construct an AST respectively: lexical and syntax analysis.

Lexical Analysis (Lexing)

Lexical analysis is the process of converting a source code of a program into a sequence of tokens (lexemes), each token is a data structure that represents a particular type such as identifiers, keywords, and operators … etc. A program that perform lexical analysis usually named lexer, tokenizer [1] or scanner. Tokens is basically a string with identified meaning. It is a structured as pair that consist of a token name an optional token value.
Common token names are:

- Identifier: names chosen by the programmers;
- Keyword: names reserved in the programming language;
- Separator: punctuation characters and paired delimiters;
- Operator: symbols that operate on arguments and produce results;
- Literal: numeric, logical, textual, reference literals;
- Comment line, block

Consider this expression in C# programming language:
`a = b + 4 / c;`
The lexical analysis of this expression yields the following sequence of tokens:
[(identifier, a), (operator, =), (identifier, b), (operator, +), (literal, 4), (operator, /), (identifier, c), (separator, ;)]

Syntax Analysis (Parsing)
Syntax analysis is the process of analyzing a string of symbols conforming to the rules of a formal grammar. A program that perform syntax analysis usually named parser. [3]
Continuing with the previous example the parser will group the tokens which are generated by the lexer and construct a syntax tree for each statement in the source code with respect of the Context Free Grammar (CFG) for a given programming language, the Figure 4.2 and Figure 4.3 show the CFG and an AST for the example that mentioned earlier.

```
<A - expression> ::= <A - expression> + <A1> | <A - expression> – <A1> | <A1>
<A1> ::= <A1> * <A2> | <A1> / <A2> | <A2>
<A2> ::= <identifier> | <number>
<identifier> ::= letter (letter | digit | '_')
<number> ::= sign? digit+
<letter> ::= [a-zA-Z]
<digit> ::= [0-9]
<sign> ::= '+' | '-'
```

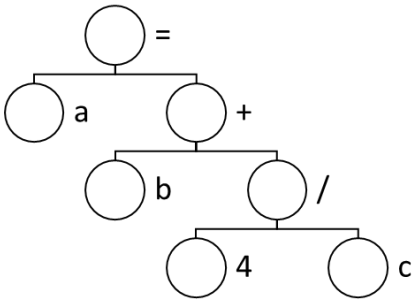

Figure 3. AST for simple assignment statement

The previous example shows a single assignment statement, but really method may contains tons line of code, that have a giant AST, but the process still the same.

B. Bug Prediction
After the AST have been constructed, it's the time to start the second phase which is concern about evaluate the syntax tree with a test data and looking forward to predicate any sort of bugs that may method under test has.

Choosing test data is one of the challenges in this research, so the proposed method suggested in this research is to use concolic execution technique over the others because it has the advantages of the static analysis and symbolic execution technique. The concolic execution will be explained in more details in the next section.

Now after that data have been generated, each path is examined with the test input to explore all the possible paths in the syntax tree.

Path exploration process indeed is another challenge, because not all the paths are feasible in almost the cases, nothing but the path exploration will be guided with the test data and path constraints to achieve a high code coverage and expected results.

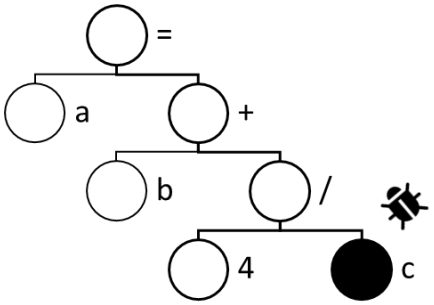

Figure 3. AST for a simple assignment statement that contains a bug

In our previous example an obvious bug has been detected in 4 / c, because all of us are already knew if the c = 0 the program will crash at this point and will throw *DivideByZeroException*

C. Report Generation

Finally after almost – if not all - the code branches have been discovered, and all the bugs have detected, it's the time report those bug and generate a proper suite of unit tests that are smart enough to examine all the source code branches that already discovered that lead to path explosion and test fails.

Those Smart Unit Tests will be automatically generate without human interfering, this is very handy to the developers and save a lot of time for discover many software bugs that may developers don't think about.
According to our previous example the following unit tests will be generated:

```
[Test]
void Test_Case_1()
{
   // Arrange
   int a;
   int b = 0;
   int c = 0;

   // Act & Assert
   try
   {
      a = b + 4 / c;
   }
   catch(DivideByZeroException ex)
   {
      Assert.Fail();
   }
}
```

```
[Test]
void Test_Case_2()
{
   // Arrange
   int a;
   int b = 0;
   int c = 2;

   // Act
   a = b + 4 / c;

   // Assert
   Assert.Equals(2, a);
}
```

As we seen from the above test cases, that the first test case is cause the bug when c = 0, while the latter ensures that the correctness of the tested code, and doing what is supposed to do.

4. Analysis

The methodology will be applied to achieve the objectives of the study, and the results will be presented.

5. Conclusion

Software Testing is very important phase on the SDLC, to ensure the quality of the software. This paper introduced a new approach for predicting the software bugs in the early stage.

References

Articles:

Albzeirat, M. K., Hussain, M. I., Ahmad, R., Al-Saraireh, F. M., Salahuddin, A., & Bin-Abdun, N. (2018). Applications of Nano-Fluid in Nuclear Power Plants within a Future Vision. *International Journal of Applied Engineering Research*, *13*(7), 5528-5533.

Albzeirat, M. K., Hussain, M. I., Ahmad, R., Al-Saraireh, F. M., & Ahmad, I. (2018). A novel mathematical logic for improvement using lean manufacturing practices. *Journal of Advanced Manufacturing Systems*, *17*(03), 391-413.

Albzeirat, M. K., Hussain, M. I., Ahmad, R., Al-Saraireh, F. M., Salahuddin, A., & Bin-Abdun, N. (2018). Applications of Nano-Fluid in Nuclear Power Plants within a Future Vision. *International Journal of Applied Engineering Research*, *13*(7), 5528-5533.

Albzeirat, M. K., Hussain, M. I., Ahmad, R., Al-Saraireh, F. M., & Ahmad, I. (2018). A novel mathematical logic for improvement using lean manufacturing practices. *Journal of Advanced Manufacturing Systems*, *17*(03), 391-413.

Book: Author Name, Title. No. of Edition, Editor, City, State or Country: Publishing house, year, pp (-).

Part of a book: Author Name, “Title”, in Book, edition, editor, City, State or Country: Publishing house, year, pp.(-).

Proceedings of the Conferences: Author Name, “Title”, in Conference, year, pp.(-).

Thesis : Author Name ,(year) title. thesis (PhD.), University Name.