International Journal of Research in Advent Technology, Vol.7, No.3, March 2019
E-ISSN: 2321-9637
Available online at www.ijrat.org

Automated Test Case Generation Tool Based On Pycparser

Bhumikaben Patel (PG Student)[1], Jignesh Patoliya (Assistant Professor) [2]
V.T.Patel Department of Electronics & Communication, Chandubhai S. Patel Institute of Technology (CSPIT), Charotar University of Science and Technology (CHARUSAT)
Email: 17pgevd012@charusat.edu.in, patelbhumi6964@gmail.com

Abstract: The cost of testing software and handling errors within a development cycle rather than the subsequent cycles, has been estimated very high. This emphasizes that current testing methods are often inadequate, and that helping reduce software bugs and errors is an important area of research with a substantial payoff. This is particularly true for the increasingly complex, distributed systems used in many applications from embedded control systems to military command and control systems or (for our research) critical avionics applications or systems. The purpose of producing the tool is to help in software testing as well as to reduce cost of it for safety critical avionics applications. These systems may exhibit intermittent or transient errors after prolonged execution that are very difficult to diagnose. Our goal is to help reduce the high cost of developing test cases for safety-critical software applications that require a certain level of coverage for certification, for example, safety critical avionics systems that need to demonstrate MC/DC (modified condition and decision) coverage of the code. This paper explores strategies for automatic test case generation using Pycparser with different code coverage criteria or structural coverage criteria. This criteria includes function coverage, statement coverage, branch coverage and condition coverage to measure what percentage of code has been exercised by a test suite. Coverage criteria are usually defined as rules or requirements, which a test suite needs to satisfy. We show that how the automated test case generating tool can be used to automatically generate test scenarios.

Index Terms- Introduction, Scope, Out of Scope, Structural Coverage Analysis, Related Work, System Framework, Conclusion and Future work

1. INTRODUCTION

Software development for critical avionics control systems, such as the software controlling aeronautics applications like Aircraft Flight Control System, Engine Control System, Flight Management Control Systems are costly, time consuming, and error prone process.[1] In such projects, the validation and verification phase (V&V) consume approximately 50%-70% of the software development resources. Thus, if the process of deriving test cases for V&V could be automated and provides requirements-based and code based test suites that satisfy the most stringent standards (such as DO-178B-the standard governing the development of flight-critical software for civil aviation , dramatic time cost savings would be realized.[2]

This paper presents a method for automatically generating test cases to structural coverage criteria, which presents a method for automatically generating test cases for Statement Coverage, Brach Coverage and Condition Coverage of test suit. We show how a tool can be used to generate complete test cases that provide a predefined coverage of any software development artifact. Software testing is one of the most expensive parts of software development. The goal of testing is to detect as many errors as possible with minimum cost. Often some coverage criteria is specified that needs to be satisfied during testing. Testcases should be selected to achieve the desired coverage and detect maximum possible errors. Selecting testcases is a challenging task, usually performed manually on a case-by-case basis. Clearly, a tool that will automatically generate useful testcases for a class of software modules will be extremely useful. [3] We have used Pycparser in this automatic test case generation tool which helps to generate tokens from C source code and these tokens can be used to generate test cases.

2. SCOPE

Experiment and Test conditions age of (switch case, if-else, while circle and for circle with various administrators) of test suit.

3. OUT OF SCOP:

This device won't take some other documents with the exception of .C records as an info. So it can't be utilized for testing some other source code document.

It won't consider some other activities of C code out of these contingent explanations.

4. STRUCTURAL COVERAGE ANALYSIS (SCA)

In programming testing auxiliary inclusion investigation is a standout amongst the most critical part and here we are going to utilize this for produce experiments. Every necessity in application at least one tests which demonstrate that it has been executed effectively. Auxiliary inclusion demonstrates that these tests practice the majority of the code. As indicated by DO-178B Structural inclusion will be articulation inclusion, choice inclusion and MC/DC inclusion relying upon the product level. DO-178B (like necessity based testing), which perform basic inclusion has some key advantages: prerequisites are finished as for code, experiments are finished, no code is conveyed that shouldn't be there, code for use in different designs is unmistakably distinguished. Auxiliary inclusion incorporates: Statement inclusion, Decision inclusion, MC/DC inclusion. Explanation inclusion estimates whether every announcement experienced. This is influenced by computational proclamations than by choices.

For example:

```
If((x>1)&&(y=0))
{
z=z/x;
}
If((z=2)||(y>1))
{
z=z+1;
}
```

By x=2, y=0, z=4 as input every statement is executed once.

Articulation inclusion likewise incorporate condition inclusion, various condition inclusion, circle inclusion.

Choice inclusion estimates whether Boolean articulations, for example, if explanation and keeping in mind that announcement assessed to both genuine and false.

For example:

```
If(a>b)
{
Print("hello");
}
Else
{
Print("bye");
}
```

Here either evident case or false case so both genuine and false experiment will be produced.

This Structural Analysis is strategy which will break down the entire code of the product covering the conditions, circles branches and articulations and as needs be it will create fitting experiments naturally which needn't bother with human connection for check.

This investigation will likewise check for the Standards a flying programming ought to pursue which will consider for legitimacy to get FAA Certificate (for confirming appropriate working of the product for flying machine frameworks).[5]

5. RELATED WORK

Most previous work on test data generation for structural testing of sequential programs addresses the problem of finding data to cover a test objective in the form of given node, branch or path of the control flow graph.

Static approaches to test case generation typically extract the constraints on input values (path predicate) corresponding to a path from the control flow graph covering the test objective and then solve these constraints to find a test case which activates the path. In theory, symbolic execution can be used to construct the path predicate. However, in practice symbolic execution encounters problems in the detection of infeasible paths (notably in the case of loops with a variable number of iterations), the treatment of aliases and the complexity of the formula which are gradually built up. Various ways around these shortcomings have therefore been proposed.[4]

Dynamic approaches avoid the problems of symbolic execution by not using the path predicate. Instead, the program is instrumented so as to evaluate, at each execution, the "distance" from the test objective and general heuristic function minimization techniques are used to search for input values to reduce this distance to zero. The disadvantages of these techniques are that they may need a great many executions before a test case is found and they may fail to find a test case even when one exists.[4]

We maintain that, for full structural coverage, we do not need to construct the control flow graph. If each path to be covered is selected from the control flow graph then the feasibility of each one must be checked. This problem is reduced in our approach. Like the dynamic approach to test data generation, our method is based on dynamic analysis. Because of Pycparser used in our tool we can easily find conditional or looping statements from input C source code and after finding all the statements we can generate test cases related to that statements using dynamic analysis method. We suffer neither from the approximations and complexity of static analysis, nor from the number of executions demanded by the heuristic algorithms used in function minimization.

6. SYSTEM FRAMEWORK

Create an automated test case generation system or tool as shown in figure. A brief introduction on the functions of major components of the system:

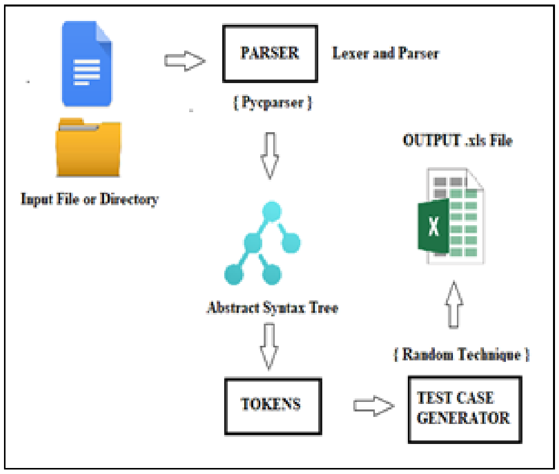

Fig2. Block Diagram of the System

6.1. Input Data

Data shall be C source code file or directory. Preprocessed file shall also be accepted for the procedure. Input shall be taken by user by choosing path where the input file is situated. User shall also select whole directory where number of source code files are situated.

6.2. Pycparser

We have used Pycparser in our tool for parsing C source code file as well as for generating tokens. These tokens are used for generate test cases. Here Pycparser works as a Lexer and Parser. We can say that it is a combination of both Lexer and Parser. Pycparser is a parser which can parse C code file and it is written in pure Python. The major need of our application is to parse C code file which is a input of the system and Pycparser is designed to parse this kind of files and therefore we have used it to make our tool better and easy to work.

Pycparser is unique in the sense that it's written in pure Python - a very high level language that's easy to experiment with and tweak. To people familiar with Lex and Yacc, **Pycparser's** code will be simple to understand. It also has no external dependencies (except for a Python interpreter), making it very simple to install and deploy.

Pycparser aims to support the full C99 language. Some features from C11 are also supported, and patches to support more are welcome.

Pycparser supports very few GCC extensions, but it's fairly easy to set things up so that it parses code with a lot of GCC-isms successfully. **Pycparser** very closely follows the C grammar provided in Annex A of the C99 standard. Pycparser was tested on Python 2.7, 3.4-3.6, on both Linux and Windows. Pycparser has no external dependencies. The non-stdlib library it uses is PLY, which is bundled in Pycparser/ply.

Note that **pycparser** (and PLY) uses docstrings for grammar specifications. Python installations that strip docstrings will fail to instantiate and use **pycparser**. You can try to work around this problem by making sure the PLY parsing tables are pre-generated in normal mode; this isn't an officially supported/tested mode of operation, though.

In order to be compilable, C code must be preprocessed by the C preprocessor-cpp. cpp handles preprocessing directives like #include and #define, removes comments, and performs other minor tasks that prepare the C code for compilation. If you import the top-level parse file function from the **pycparser** package, it will interact with cpp for you, as long as it's in your PATH, or you provide a path to it.

C code almost always #includes various header files from the standard C library, like stdio.h. While (with some effort) **pycparser** can be made to parse the standard headers from any C compiler, it's much simpler to use the provided "fake" standard includes in utils/fake_libc_include. These are standard C header files that contain only the bare necessities to allow valid parsing of the files that use them. As a bonus, since they're minimal, it can significantly improve the performance of parsing large C files.

The key point to understand here is that **pycparser** doesn't really care about the semantics of types. It only needs to know whether some token encountered in the source is a previously defined type. This is essential in order to be able to parse C correctly.[6]

6.3. Lexer

The **lexer**, also called **lexical analyzer** or **tokenizer**, is a program that breaks down the input source code into a sequence of lexemes. It reads the input source code character by character, recognizes the lexemes and outputs a sequence of tokens describing the lexemes. This process will perform lexical analysis on the source code i.e input. Examples of tokens are listed below:

Token name	**Sample values**
Identifier	Val1,y,var
Keyword	If , while , return
Separator	{ , (, ;
Operator	+ , > , =

Literal	False , 3.02e37 , "name"
Comment	/*function to add numbers*/ , //display output

Table1. Types of Tokens

It will divide the code into tokens and store in database. These tokens can be identifier, keyword, separator, operator, literal, or comment.

A **lexeme** is a single identifiable sequence of characters, for example, keywords (such as class, func, var, and while), literals (such as numbers and strings), identifiers, operators, or punctuation characters (such as {, (, and .).

A **token** is an object describing the *lexeme*. A token has a **type** (e.g. Keyword, Identifier, Number, or Operator) and a **value** (the actual characters of the described lexeme). A token can also contain other information such as the line and column numbers where the lexeme was encountered in the source code.

A **lexer** can be implemented as a class, whose constructor takes an input string in parameter (representing the source code to perform lexical analysis on). It exposes a method to recognize and return the next token in the input.

6.4. Parser

Parser is used as a compiler. Lexer is also a part of Parser. Parser considers the tokens as input, it will evaluate the conditional expressions using different stacks and grammar given to it.

A parser is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure, giving a structural representation of the input while checking for correct syntax.

6.5. Generate Tree (AST)

Now the Abstract Syntax Tree will be generated according to grammar. It will represent structure of source code written in C programming language.

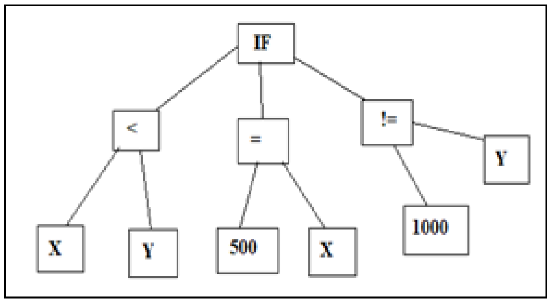

Fig3. Example of AST

It would contain variables and those will be consider for decision table. These variables given automatically generated values and using those values output will be generated for test case.

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural, content-related details. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees, which are typically built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing.

6.6. Output Data

The decision table will store all the variables and its values for generated test cases covering all the conditions (True-True, True-False, False-True, False-False) in .xls file. Output will be shown in this .xls file, where input file name, function name, line number, conditional statement, test cases related to that statement and outputs generated according to those test cases are listed.

7. CONCLUSION AND FUTURE WORK

We have demonstrated an approach of automate the test-case generation for avionics software engineering artifacts of source code. We use different coverage criteria to define what test cases are needed and the test cases are then generated using the automated test-case generation tool. We have used Pycparser to better result of lexical analysis as well as to parse .C file in a best way. We have included all conditional and looping statements to cover all the conditions in the source code for verification.

Results of the tool indicate that the approach has potential to dramatically reduce the costs associated with generating test-cases to high levels of coverage.

Future scope includes the Automated Test Case generation tool required a certain level of coverage MC/DC (Modified Condition Decision Coverage) that needs to be added for avionics systems DO-178C level A V&V. In future we can include switch case or any other requirement for verification of the software.

REFERENCES

[1]. D.J.Berndt and A. Watkins. High Volume Software Testing using Genetic Algorithms. College of Business Administration, University of South Florida.

[2]. Sanjai Rayadurgam and Mats P.E. Heimdahl. Coverage based test-case generation using model checker. Department of Computer Science and Engineering, University of Minnesota, USA.

[3]. Pankaj Jalote and Mallaku G. Caballero. Automated Testcase Generation for Data Abstraction. Department of Computer Science and Institute for Advanced Computer Studies, University of Marylend.

[4]. Nicky Williams, Bruno Marre and Patricia Mouy On-the-Fly Generation of K-Path Tests for C Functions. France.

[5]. Bhumikaben Patel and Jignesh Patoliya. Coverage Based Test_Case Generation using Automated Test_Case Generating Tool. V.T.Patel Department of Electronics & communication, Chandubhai S. Patel Institute of Technology (CSPIT), Charotar University of Science and Technology (CHARUSAT)

[6]. Online resource: http://github.com/eliben/pycparser

[7]. Online resource: aerointerview.com

[8]. Online resource: rapitasystems.com

[9]. Online resource: Wikipedia