International Journal of Advanced Computer Research (ISSN (print): 2249-7277 ISSN (online): 2277-7970)
Volume-4 Number-1 Issue-14 March-2014

Smart Test Case Quantifier Using MC/DC Coverage Criterion

S. Shanmuga Priya[1], Sheba Kezia Malarchelvi[2]

Abstract

Software testing, an important phase in Software Development Life Cycle (SDLC) is a time consuming process. Information shows that nearly 40 to 50% of software development time is spent in testing. Manual testing is labour-intensive and error-prone so there is a need for automatic testing technique. Automation brings down the time and cost involved in testing. When testing software, there are often a massive amount of possible test-cases even for quite simple systems. Running each and every feasible test-case is certainly not a choice, so designing test-cases becomes a significant part of the testing process. NASA proposed Modified Condition/Decision Coverage (MC/DC) testing criterion in 1994, which is a white box testing criterion. The objective of this paper is to automate the generation of minimum number of test cases required to test a system with maximum coverage by removing the redundant test cases using MC/DC criterion. The work also gives a tool Smart Test Case Generator Tool (STCGT) that automates the minimum number of test cases required to test the source code. This will give an idea about the test cases execution for the beginners of the testing team, thereby, aids in a quality on-time product.

Keywords

Test Case, MC/DC, White Box Testing, Automation.

1. Introduction

After developing any software application, testing plays a significant role in finding the accuracy, comprehensiveness and worth of the software that is developed. Software testing process typically consumes of about 50% of the total cost involved [1]. For safety critical software, this percentage might even be higher [2].

S. Shanmuga Priya, Assistant Professor / CSE & J.J. College of Engineering and Technology, Trichy, Tamilnadu.

Sheba Kezia Malarchelvi, Professor / CSE & J.J. College of Engineering and Technology, Trichy, Tamilnadu.

One definition of testing is "the process of questioning a product in order to evaluate it", where the "questions" are things the tester tries to do with the product and the product answers with its behavior in reaction to the probing of the tester. Testing must precisely uncover diverse classes of errors in a least amount of time and with a least amount of effort. A primary objective of testing is to ensure that the software works as stated in the specifications, so the customer could be satisfied. Even though there are many software testing approaches, the software market is still demanding for effective testing of the complex product as the size of the software being developed grows every day.

The software could be tested either manually or by automation. In manual testing, the tester tests the software manually for the defects. The tester must play the role of an end user and almost all the features of the application should be ensured for its correct behaviour. The tester follows a systematic approach by writing the test plan that leads them through a set of important test cases [3]. The difficulty with manual testing are its time consuming and not reusable, does not have any scripting facility, requires great effort and some errors could remain uncovered [4]. The next most promising technique is the automated testing, which covers the difficulties of manual testing. Automation testing is carried out by automating the steps of manual testing with the aid of any automation tools [5]. Automation testing process reduces both the testing cost and it also improves the software quality. The automation testing increases the test execution speed, considered to be more reliable, repeatable, comprehensive, programmable and reusable. So, it would be better to automate the testing process rather than doing it manually which is being needed for the emerging fields of Search based software engineering, Search based software testing etc.

The primary measurement for the adequacy of testing is code coverage analysis, analysing that all source code. The output of the coverage analysis process is the percentage of code that is covered by the test cases written. So the number of test cases written also decides the quality of the testing. Hence it should be generated carefully such that for each path

in the program there should be a test case in order to ensure maximum coverage. Here an approach for automatic generation of minimal number of test cases to provide maximum coverage in the case structural testing has been tried.

The paper is organized as follows. Section 2 gives various code coverage and coverage metrics that are used in practice, Section 3 gives the related works, Section 4 gives the proposed work, Section 5 gives the developed smart test case generator tool, Section 6 gives the salient features and limitations of the proposed system and Section 7 gives the conclusion.

2. Code Coverage and Coverage Metrics

Glass box testing or white box testing is a structural testing technique that compares test program behavior against the apparent intention of the source code. This kind of testing examines the working of a program and also takes the possible pitfalls in the structure and logic. In contrast the functional testing which is also known as black-box testing compares the test program behaviour against the requirement specification without taking the program's internal working into account.

Code coverage analysis is process that indirectly measures the quality by finding the areas of a program that are not exercised by the set of chosen test cases. This might demand to produce additional test cases to increase the coverage of code; however redundant test cases do not increase code coverage. A code coverage analyser automates this process. Coverage analysis is used to guarantee quality of the set of tests, not the actual product's quality. Coverage analysis requires access to test program source code and often requires recompiling [6].

The U.S. Department of Transportation Federal Aviation Administration (FAA) has formal requirements for structural coverage in the certification of safety-critical airborne systems [DO-178B]. Small count of other organizations has such requirements, so the FAA is significant in the definitions of these metrics. To measure how well the program is exercised by a test suite, coverage criteria are used. There is a number of coverage criteria, the main ones being are [7]:

Function coverage - Assures that each function in the program has been called or not.
Statement Coverage – Find out whether each node in the program been executed or not.
Decision Coverage – Find out whether every edge in the program been executed.
Condition Coverage – Checks for both true and false for each Boolean sub-expression.
Condition/Decision Coverage - Both decision and condition coverage should be satisfied.
Parameter Value Coverage - In a method taking parameters, verifies that all the common values for such parameter has been considered.
Modified Condition/Decision Coverage - For safety-critical applications (e.g., for avionics software, medical expert system) it is often required that modified condition/decision coverage (MC/DC) must be satisfied. This criterion extends condition/decision criteria with requirements that each condition should affect the decision outcome independently.
JJ-Path Coverage – Checks that all jump to jump paths [8] are executed.
Path Coverage – Checks that every possible route through a given part of the code has been executed.
Entry/Exit Coverage – Identifies whether all the possible call and return of the function has been executed.
Loop Coverage – Checks for the execution of every possible loop zero times, once, and more than once?
Parameter Value Coverage - For each parameter in a method, checks whether the most common possible parameter values has been tested.
Safety-critical applications are often required to demonstrate that testing achieves 100% of some form of code coverage [9].

3. Related Works

In 2011, Swathi et al. [10] proposed a tool that automates the generation of test cases using control structure methods. The developed tool aimed to for 100% coverage for the given C language structural code which includes statement coverage, decision coverage, path coverage and branch coverage analysis. In 2012, Manish et al. [11] proposed an automated test case generator for C source code. The approach converted the C source code to Control Flow Graph and found all possible feasible paths to generate the test cases. In [12] Zalan Szugyi and Zoltan Porkolab analyzed several projects written in Ada programming language. They estimated the difference of the required test cases of Decision Coverage and Modified Condition / Decision Coverage. In 2010, Sangeeta Tanwer and Dharmender Kumar [13], has proposed an automatic

test case generation of C language program using CFG. The tool automates the unit testing of the software. The existing system does not address the issues like unreachable blocks during execution which could produce an unreachable code. If the exit block is unreachable from the entry block, it could lead to an infinite loop. Moreover, the redundant test cases were never eliminated.

4. Proposed Work

This work aim to reduce the test case size by automating the number of test cases required to test the system, while preserving the maximum code coverage which helps in reducing time and cost. It eliminates the redundant test cases. This paper proposes an automated test case generator for a C language. The smart quantifier is designed to produce the minimum number of test cases based on the test path with maximum coverage based on Modified Condition/Decision Coverage criterion.

The approach followed is done by generating the control flow graph, a data structure that is used to identify the independent paths for an application to be tested. From the generated control flow graph, the number of independent paths can be found by calculating the Cyclomatic complexity. The number of test cases needed is equal to the value of the Cyclomatic complexity.

The figure 1 shows the flow diagram of the proposed system. The C code to be tested is fed as input to the smart quantifier, whose job is to produce the output as the minimum number of test cases need to test the given application. The quantity of test cases is derived in such a fashion, that for each independent path that exists, a separate test case should be considered. The test cases are executed against the framework and the test coverage is measured.

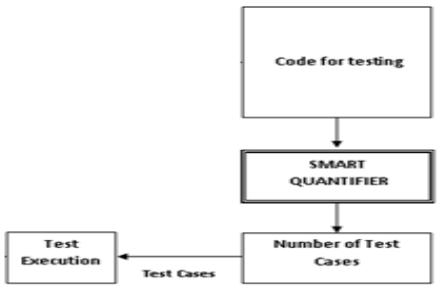

Figure 1 : Flow diagram of the proposed system

Figure 2 shows the block diagram of smart test case quantifier. The functions of the blocks are as follows:

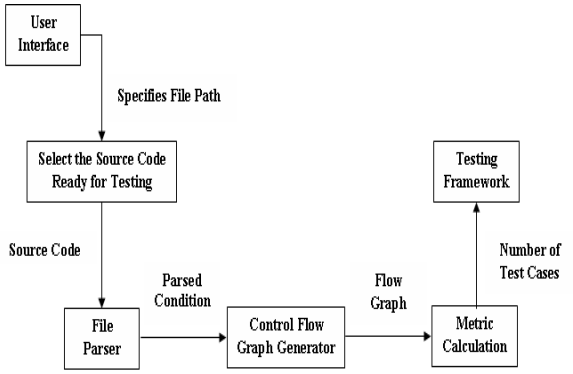

Figure 2 Block diagram of smart test case quantifier

A. User Interface

The user interface gets the path of the file which contains the source code of the product which is to be tested. The user can have all the modules related to a particular project in a same path which can be taken by the parser in a recursive manner.

B. File Parser

The algorithm used in file parser module is recursive descent parser algorithm, which is a top-down parser. The parser builds the parse tree from the top, that is, from the start symbol to the leaf. The parser modules traverse the file from starting to end and identify all the conditional statements and write them in a list and send to the control flow generation module. Perhaps the tough part of a recursive descent parser is the scanning, that is, repeatedly fetching the next token from the scanner.

C. Control Flow Graph (CFG) Generator

A graph could prop up the testers in analysing and understanding the behavior of a program that is subjected to testing. A Control Flow Graph is one such and is generated by the Control Flow Graph Generator, in order to capture the flow of control within a program to be tested. There are two possible options to construct CFG. It can be constructed either manually or could be done with the help of tools. For relatively small programs, CFG can be constructed manually without much difficulty. However, it is difficult to construct CFG as the size of the program grows, and there arises a need for any tool that could automate the generation. Such

automated generation of CFG is yielded by the Control Flow Graph Generator. The input to the control flow graph generator is the output of the file parser, that is, the list containing all the conditional statements present in the code to be tested. From the list received, the file parser generates the CFG. The conditional operators are kept as nodes and the flow between statements is indicated by means of edges.

D. Metric Calculation

To generate the number of test cases the generated control flow graph is given as input to the metric calculation. The essential parameters for calculating Cyclomatic Complexity are number of nodes and edges. From the CFG, calculate the number of edges and nodes. Check whether number of nodes is lesser than number of edges. This block is to identify the number of nodes and edges from the control flow graph and generates the number of test cases for full coverage. The steps followed to generate the minimum number of test cases are:

Step 1: The generated control flow graph is given as input.
Step 2: Calculate the number of edges and nodes.
Step 3: Check whether number of nodes is lesser then number of edges.
Step 4: Calculate the Cyclomatic complexity using the formula M = E − N + 2, where, where E is the number of edges of the graph and N is the number of nodes of the graph.
Step 5: Display the number of test cases needed.

5. Developing Smart Test Case Generator Tool

The Smart Test Case Generator Tool (STCGT) has been implemented using Microsoft Visual C++. The figure 3 shows the browser screen which takes the input, the C source code which is to be tested.

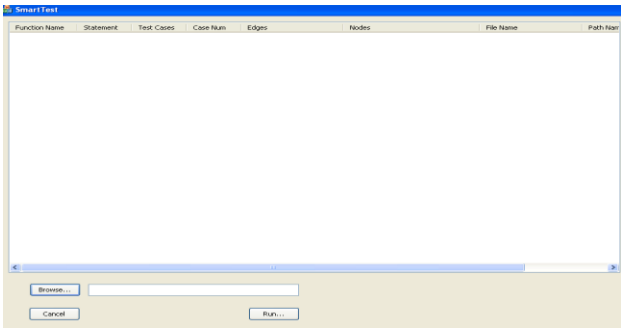

Figure 3: Browser window

The figure 4 shows the Selection window. The code that is to be tested can be chosen by clicking on the browse button, by specifying the path where the code is located.

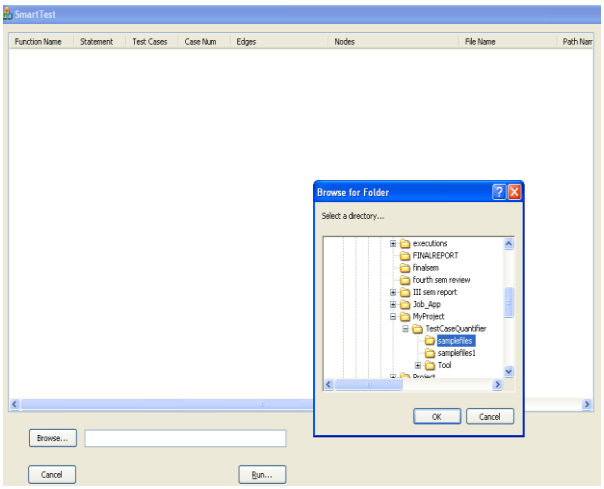

Figure 4: Selection window

Figure 5 shows the path of the selected file which contains the code to be tested. The several modules to be tested must be in the same folder.

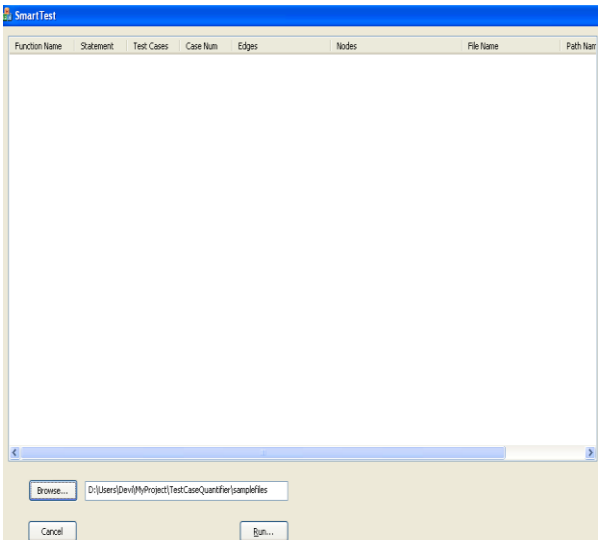

Figure 5: Displaying path of the file containing the code to be tested

Figure 6 shows the minimum number of test cases generated for the selected source code. The displayed test cases must be the minimum

requirement that a tester should take in to consideration when testing the code.

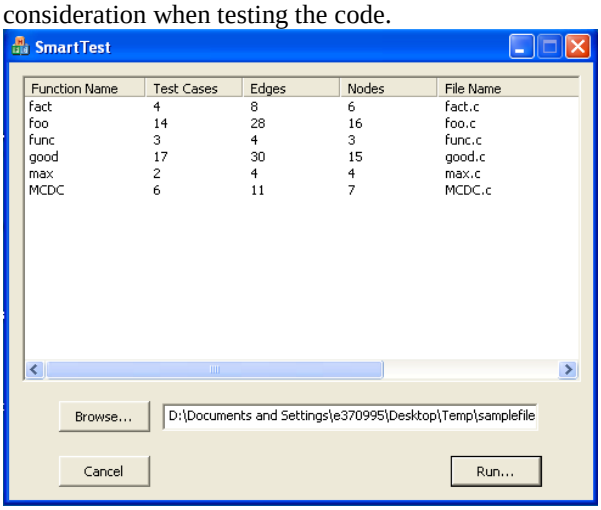

Figure 6: Smart test case generator showing minimum number of test cases required for testing the given code

6. Salient Features and Limitations of the Proposed System

The various salient features of Smart Test Case Quantifier are:

a) It uses MC/DC Coverage metrics.
b) It automates structural testing.
c) Quantifies the number of test cases needed.
d) Provides maximum coverage with minimum number of test cases.
e) Minimizes the number of iterations for getting complete code coverage and
f) Can be used for products to a higher level safety critical system.

The developed system has the following limitations:

a) It can work for C codes only.
b) The system is able to generate the test cases for all the modules of the project at the same time, provided all modules must be in the same path.

7. Conclusion

The proposed work is based on MC/DC criterion which generates the minimum number of test cases that is required to test the C source code. The proposed tool STCGT, has a limitation that it works only for C source code. In future, works need to be proposed to make this as a generic tool by using various other coverage metrics.

References

[1] Rudolf Ramler, Klaus Wolfmaier, "Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost", Proceedings of the international workshop on Automation of Software Test, pp.15-23, 2006.
[2] Michael Grottke, Kishor S. Trivedi, "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate", IEEE Computer, Vol.40, No.2, pp. 107-109, 2007.
[3] J. Chilenski and S. Miller., "Applicability of Modified Condition / Decision Coverage to Software Testing", Software Engineering Journal, pages 193–200, September 1994.
[4] Mark Fewster and Dorothy Graham, "Software Test Automation: Effective Use of Test Execution Tools", ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.
[5] Steve Cornett Code Coverage Analysis, July 29 2004 http://www.bullseye.com/coverage.html.
[6] http://idtus.com/img/UsefulAutomatedTestingMetrics.pdf.
[7] http://www.math.unipd.it/~tullio/IS-1/Dispense_2003/Software_Testing_Metrics.htm.
[8] http://covtool.sourceforge.net/#Intrumenting.
[9] http://www.lexjansen.com/pharmasug/2004/fdacompliance/fc06.pdf.
[10] Swathi.J.N, Sumaiya Thaseen.I and Sangeetha.S, "Minimal Test Case Generation for Effective Program Test using Control Structure Methods and Test Effectiveness Ratio", International Journal of Computer Applications (0975 – 8887) Volume 17– No.3, March 2011.
[11] Manish Mishra, Shashi Mishra and Rabins Porwal, "Basic Principle for Test Case Generation Automatically", VSRD International Journal of Computer Science & Information Technology, Vol. 2 (9), 2012, 772-781.
[12] Zalan Szugyi and Zoltan Porkolab, "Necessary Test Cases for Decision Coverage and Modified Condition / Decision Coverage", Electrical Engineering 52, no. 3-4 (2010): 187-195.
[13] Sangeeta Tanwer and Dharmender Kumar, "Automatic Test Case Generation of C Program Using CFG", IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 4, No 8, July 2010.

S. Shanmuga Priya is currently obtained her M.E. in Computer Science and Engineering from Anna University, Chennai, Tamil Nadu, India. She received her B.E. in Computer Science and Engineering in 2003 from Bharathidasan University, Tamilnadu, India. She has around 10 years of experience in teaching. Presently she is serving as Assistant Professor in the Department of Computer Science and Engineering at J.J. College of Engineering and Technology, Tiruchirappalli, Tamil Nadu, India. Her research interests include Software Engineering and Software Testing.

P. D. Sheba Kezia Malarchelvi received B.E. in Computer Engineering in 1991 from Madurai Kamaraj University, Tamil Nadu, India. She completed M.E. in Computer Science in 1995 from the Regional Engineering College, Tiruchirappalli, Tamil Nadu, India. She received Ph.D. in Computer Science and Engineering in the year 2010 from Bharathidasan Institute of Technology, Bharathidasan University, Tamil Nadu, India. She has around 22 years of experience in teaching. Presently she is serving as Professor and Head of the Department of Computer Science and Engineering at J.J. College of Engineering and Technology, Tiruchirappalli, Tamil Nadu, India. Her research interests include Security, Grid Computing and Cloud Computing.