Coverage verification by tests of program branches and conditions of software in environments of automatic testing

A H Li\(^1\), I A Lyapunova\(^1\) and N A Fomenko\(^2\)*

\(^1\) Southern Federal University, 105/42 Bolshaya Sadovaya Str., Rostov-on-Don, Russian Federation, 344006
\(^2\) Gubkin Russian State University of Oil and Gas, 65-1, Leninsky prospect, Moscow, Russian Federation, 119991

*E-mail: fomenko.nata.alex@gmail.com

Abstract. Software verification is a discipline of software engineering that focuses on compliance of software requirements. In critical areas such as the aviation industry, the quality of military software is subject to strict control. Covering the requirements allows you to assess the completeness of a set of tests in relation to the functionality of the system, but does not allow you to assess the completeness in relation to its software implementation. The same function can be implemented using completely different algorithms that require a different approach to the organization of testing. Our approach uses "concolic" testing to automatically generate test cases.

1. Introduction

In testing, an important aspect is the quantitative assessment of the quality of the test set \([1, 2]\). Qualitative can be considered a set of tests that meet a certain criterion of completeness. As a rule, the ratio of the tested part of the system to its full volume is taken as a measure of completeness. A complete set of tests according to the criterion of completeness makes it possible to assert that the system implements all the functionality specified in the requirements.

The basic and most frequently used method of determining the completeness of the set of tests is the ratio of the number of test requirements for which test is composed of the total number of test requirements. That is, we are talking about covering all test requirements with a test set. Coverage of requirements gives an opportunity to assess the degree of completeness of the set of tests in relation to the functionality of the system, but does not allow to assess the completeness in relation to its software implementation. The same requirement can be implemented using radically different algorithms that require a different approach to the organization of testing \([3-8]\).

For a more detailed assessment of the completeness of the set of tests when testing by the white box method, the software code coverage is investigated. During the operation of each test case, a certain section of program code is executed; all code sections must be executed when the entire set of tests is executed. If there are sections of code that are not executed when the test set is run, the test set is potentially incomplete (that is, it does not test all the functionality of the system). To study the coverage of the program code should (white box strategy) proceed after the full coverage of the requirements (black box strategy). After all, full code coverage does not guarantee that the test Suite covers all requirements.
One of the first James C. King introduced the concept of symbolic execution of the program, which is closely related to the normal concept of execution of the program. This method has the advantage that a single symbolic execution can be a large, usually infinite, class of normal executions.

CUTE [9] means "Concolic Unit Testing Engine", which combines symbolic execution and normal execution to generate test inputs to explore all possible execution paths.

DART [10] means "Directed Automated Random Testing", a tool developed by Patrice Godefroid, Nils Klarlund and Koushik Sen as a single testing method. The tool uses the concept of "concolic" testing to cover the path. It uses the following methods:

- Automatic interface extraction;
- Generation of automatic test driver for random testing;
- Dynamic analysis of program behavior to automatically create test inputs to control execution on alternative paths.

Zeina Avedikian [9] proposed an approach to automatically generate test inputs according to MC/DC criteria. The algorithm steps are the following:

1. For each solution, calculate the MC/DC coverage set using a truth table.
2. The suggested fitness features are given below:
   - Branch Fitness: this is a function of the distance between branches, describing how far the decision statement is checked to make its result true.
   - Dependency Fitness: this function describes a series of acceptance statements from the input that will help to achieve a solution obtained using the control Flow Graph (CFG).
3. Generate test inputs using metaheuristic algorithms.

Godbole, S, and others [10] proposed an approach to increasing the MC/DC using the program code of the transformer. The Transformer program code was based on the Quine-McCluskey minimization method. The purpose of this article was to automatically create a test Suite MC/DC.

Vitthal and others [11] proposed an approach called the Modified Quain – McCluskey method (MQM). Using the MQM method, the performance of digital circuits can be increased by reducing the number of min-literals or min-terms in a Boolean expression. MQM is much easier and faster than the Quine – McCluskey method due to fewer comparisons required. Thus, this method can be used to achieve the rate of minimization of the Boolean function manually and improve the performance of the conventional method [10].

2. General structure of the algorithm
At the entrance we get a test program P and a set of manual tests M. Our goal is to show the uncovered areas of the code by a set of M, and also to generate a set of tests to check the code by MC/DC tests.

![Figure 1. General scheme of the algorithm.](image-url)
As you can see in figure 1, the original program p is fed to the input of the CREST utility, which performs two operations for us. First, it transforms code using the CIL utilities; second, it makes every branch and every cycle by additional code. With this additional code, we can find out which parts of the code were uncovered by manual tests.

Next, the transformed program P* and a set of tests M we submit to the entrance to the CAT (ATT) which runs the program on a set of tests and produces a report. In addition to the test results, the report contains information about uncovered code sections. If all tests succeed, then P* is passed to the module automatic test generation (AGT), which generates the test code and passes the test set I in analyzer cover the CA that issues the report in the form of percent cover in the metric of the critical issues.

CREST is a ready-made solution for "concolic" testing [12]. The latest version can use a search strategy that is guided by the static structure of the program under test, namely the control flow graph (CFG). But on large programs, the utility shows only a third of the coverage (from 20% to 35%). We need to increase this, perhaps by sacrificing performance. The utility also generates inputs only for numerical variables (int, short, char). In our approach, we also implement symbolic variables for some other types of variables (arrays, pointers, floating-point numbers). The CIL conversion and code instrumentation modules are well implemented in the utility, so they were used in our approach.

Code transformation includes splitting solutions in a program into a nested If / Else structure, according to CIL. This nested structure is based on how conditions are evaluated in C, for (A&&B) type Boolean expressions, condition B is evaluated only if condition A is true. Similarly for (a / / B) type Boolean expressions, if A is true, condition B is not evaluated, and if condition A is false, then only condition B is evaluated.

Since the CAT is not something unified, and created for a specific product, we will not consider the process of testing. We only care about the report, in which, in addition to the test results, we will see uncovered branches.

The tester on the basis of this report explores the uncovered parts of the code, identifying the reason for not covering. Often, this will be due to the inability to get to this site using test inputs, protected code, code for different versions or simply unused code. But there is always a chance that it will be a missed or not fully tested requirement from the specification.

The module of automatic test set generation receives the program transformed with the help of CREST. As shown in figure 2, the P* program is sent to a static analyzer to determine character variables, level nested conditions, and generate a control Dependency Graph (CDG).

The program is then routed to the Leveling Module and the CDG module. Which generate Path Condition-a conjunct of conditions from the program by which the solver generates test input data.

2.1. Static analyzer
Most of the work on static processing of the program the utility CREST takes. For it, you need manually specify which variables are control variables, that is, which variables should be considered as input and to be tested. In our campaign we have manual tests in which all possible variables for testing are specified. Also the ability to explicitly specify which variables should be considered input is implemented. This is done to test system code sections that cannot be reached by external interface parameters.

Symbolic execution is a programmatic method of analysis that means executing a program with symbolic values rather than specific values, where symbolic values are the programmatic variables that control its behavior. Each assignment operator in a program is represented as a function or in terms of symbolic variables. Each conditional expression is expressed as a preposition formula in terms of symbolic variables. In our implementation, we store a symbol table for program variables that contain symbolic values and specific values. For the assignment operator, the symbol table is viewed for each variable on the right side and replaced with its symbolic value. In our approach, symbolic execution for data types such as int and float is obtained by storing the symbol table as described
above. Symbolic execution for complex data types, such as structures, arrays or pointers is achieved by preserving the structure.

![AGT Module Diagram](image)

**Figure 2.** Block diagram of the AGT module.

After initialization, character variables static analyzer puts the levels for conditions in a nested structure of if–else is needed for test generation in accordance with the criteria of MC/DC.

2.2. **Alignment module**

The moment the P** program is started with random values, it turns out PC (path condition) to direct the program through other branches, we negate one or more conditions obtained from the previous run. Resulting PC sends the new to the constraint solver (or as in our case the theorem proving) Z3 [13] to check if a new path is possible, if so, we obtain values for character variables, and if not, we search for a new path by negating the next level conditions.

The disadvantage of the alignment module is that we get less coverage when there are control-dependent operators, to overcome this, a second approach using the CDG module [14] is used.

2.3. **CDG module**

The alignment module cannot provide adequate coverage if the program is dependent on the control operators. These instructions depend on some other operator in the program. We use the information
about the control dependencies of a program P with the purpose of studying the maximum number of conditions to ensure high coverage. In the module, we choose a path so that it contains the maximum number of uncovered conditions, that is, conditions in which at least one side was not visible. The phases of the CDG module are as follows:

- building CDG (this phase is in the static analyzer, because it needs a program P*, and the CDG module is given P**);
- CDG initialization;
- Search for upper K-paths;
- * find workable paths with MAXSAT;
- CDG update.

After each test run that uses test cases for paths derived from the previous phase, the number of nodes is updated by first updating the leaf node score in the resulting side (true/false block) to 0 and therefore propagating that score up towards the root using a similar procedure. This process of updating scores, finding paths, using MAXSAT, getting test programs, running programs on generated test cases is repeated until all paths are explored or there are more than ten failed attempts to increase coverage.

2.4. Testing module for automatic test generation
The alignment module, in which we provide the nesting level of conditions and direct the path of conditions (PC), so that conditions only at a certain level are inverted, do not handle impracticable constraints, if they are found, we look for a new condition that can be directed. The module searches for new valid PCs until there are more than ten consecutive attempts that lead to impossible path constraints, path constraints, so that all conditions are visible, or there are no more PCs to study. If any of these values are true, the module stops and displays the results in the form of coverage, test sets received, and time spent for the full run.

The CDG module consists of constructing a CDG, determining an estimate for each node, finding upper K-paths, using MAXSAT to find a valid set of constraints, running the program through the best paths, where the estimate determines the total number of uncovered conditions in the subtree. When we run the module, while the conditions haven’t been covered, one test case covers several conditions at once. Results are produced until there are no paths left in the CDG, or there are more than ten unsuccessful attempts to increase coverage. The first alignment module examines all possible conditions in accordance with MC / DC. To initialize a CDG, the state of the program, that is, the number of covered conditions, from the alignment module is used and the maximum number of uncovered conditions is examined.

The summary of the received results is shown in table 1. For each program the covering, the number of test sets produced and the time spent by the module, including execution of a code and the static analysis is displayed.

About 52% of programs received covering of more than 80%. The average covering is 78.275%. The time taken by the module is approximately 2 seconds and 500 milliseconds that include all the tools and static code analysis along with time of performance of the alignment module and the CDG module. The maximum number of test sets received with use of this algorithm is 20.

This approach was developed in the Sukhoi Design Bureau Company. Testing was carried out on several modules FS (functional software) for which a complete test set according to requirements of the specification has been written. First, the source code of the module is passed through the CREST utility for transfer to CIL and primary instrumentation. Further the program along with manual tests is sent to the modified by CAT for testing. Modified CAT in addition to the report on results of testing (on the strategy of a black box), generates the additional report on uncovered condition lines.

The combined approach made it possible to increase the covering in comparison with the separate use of the alignment module and the CDG module.
Table 1. The results received by testing the AGT module.

| №  | Program | Function   | AGT module | Covering | Test | Time       |
|----|---------|------------|------------|----------|------|------------|
|    |         |            |            |          |      |            |
| 1-10 | Linpack | idamax     | 72         | 6        | 2.180 s |
|      |         | dscal ur   | 85         | 6        | 2.056 s |
|      |         | ddot ur    | 65         | 4        | 2.212 s |
|      |         | daxpy ur   | 67         | 5        | 2.244 s |
|      |         | dscal r    | 83         | 4        | 1.932 s |
|      |         | ddot r     | 61         | 3        | 2.064 s |
|      |         | daxpy r    | 65         | 4        | 2.160 s |
|      |         | dgesl      | 95         | 17       | 2.628 s |
|      |         | dgefa      | 80         | 8        | 1.940 s |
|      |         | matgen     | 75         | 2        | 2.160 s |
|      |         | select     | 66         | 4        | 3.104 s |
|      |         | expint     | 71         | 4        | 2.040 s |
| 11-15 | WCET  | qurt fabs  | 100        | 2        | 1.800 s |
|      |         | qurt sqrt  | 57         | 2        | 2.072 s |
|      |         | qurt       | 66         | 2        | 1.896 s |
|      |         | selectMedian | 31      | 3        | 2.192 s |
| 16-19 | BRNS  | UpCounter  | 95         | 10       | 1.876 s |
|      |         | AAS        | 97         | 15       | 2.360 s |
|      |         | GroupAAS   | 64         | 10       | 3.000 s |
|      |         | adjust     | 93         | 5        | 1.868 s |
| 20-22 | Heap Sort | heapify    | 80         | 3        | 2.056 s |
|      |         | heapSort   | 100        | 2        | 1.720 s |
| 23    | Insertion Sort | insertionSort | 83 | 5 | 2.052 s |
| 24    | Bubble Sort  | bubbleSort | 83         | 4        | 1.924 s |
| 25-26 | Quick Sort | partition  | 100        | 4        | 1.829 s |
|      |         | quickSort  | 60         | 2        | 2.004 s |
| 27    | Merge two arrays | merge | 96         | 5        | 1.956 s |
| 28-29 | Tic Tac Toe | checkwin   | 98         | 20       | 2.568 s |
|      |         | ticTactoeMain | 56   | 6 | 2.532 s |
| 30    | Triangle   | triangleCheck | 79   | 9 | 3.496 s |
| 31    | Binary Search in ATM | binarySearch | 91   | 5 | 1.928 s |
| 32    | ATM        | atm        | 95         | 8        | 1.960 s |
| 33    | Median of 2 Sorted | getMedian | 87         | 5        | 2.048 s |
|      |         | Own Below Threat | 100 | 2 | 1.744 s |
|      |         | Own Above Threat | 100 | 2 | 1.724 s |
|      |         | Positive RA Alt | 100 | 5 | 1.724 s |
|      |         | Inhibit Biased Climb | 100 | 2 | 1.728 s |
| 34-40 | TCAS | Non Crossing Biased Climb | 37   | 2 | 2.048 s |
|      |         | Non Crossing Biased Descend | 37   | 2 | 2.004 s |
|      |         | alt sep test | 61   | 5 | 2.068 s |
The disadvantages of this approach are:
• The method does not support inter-functional calls;
• The method does not support some data types (for example, lines, enumerations, unions, and dynamic arrays);
• The method does not support bitwise operators;
• methods do not support file descriptors or any manipulation with a heap;
• on large programs, the generation of test sets can take hours (it is apart from that the manual tests will be carried out several hours).

Pros of this approach:
• This method allows you to assess quickly the completeness of manual tests, as well as see all parts of the code that are not available for testing on the black box strategy;
• The method automatically generates test sets based on the white box strategy, which considerably saves time for the tester;
• using two testing strategies (white and black boxes) allows you to be more confident in the quality of software verification.

References
[1] Leslie A 1992 Software Considerations in Airborne Systems and Equipment Certification Available from http://www.dcs.gla.ac.uk/~johnson/teaching/safety/reports/schad.html
[2] Sen K 2007 Concolic Testing (Invited Mini Tutorial) Proc. 22nd IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE 2007, NY) pp 571-2
[3] Necula G C, McPeak S, Rahul S P and Weimer W 2002 CIL: Intermediate Language and Tools for Analysis and transformation of C Programs Proc. of Conference on Compiler Construction pp 213-28
[4] Chilenski J J and Miller S P 1994 Applicability of Modified Condition/Decision Coverage to Software Testing Software Engineering Journal 9(5) 193-200
[5] Stafford J A 2000 A Formal, Language-Independent, and Compositional Approach to Interprocedural Control Dependence Analysis (Boulder: University of Colorado) p 124
[6] Sen K, Marinov D and Agha G 2005 A Concolic Unit Testing Engine for C 13th ACM Sigsoft International Symposium Jointly with 10th European Software Engineering Conference (University of Illinois) pp 263-72
[7] Godefroid P, Klarlund N and Sen K 2005 DART: Directed Automated Random Testing PLDI ’05 Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (University of Illinois at Urbana-Champaign) pp 213-23
[8] Awedikian Z 2009 Automatic Generation of Test Input data for MC/DC test coverage: Soccer Lab (Ecole Polytechnique de Montreal) p 38
[9] Godbole Y, Prashanth G, Mohapatro D P and Majhi B 2013 Increase in modified condition/decision coverage using program code transformer Advance Computing Conference (IACC) IEEE 3rd International (Amsterdam, The Netherlands) pp 1400–7
[10] Jadhav V and Buchade A 2012 Modified quine-mccluskey method (Preprint arXiv:1203.2289)
[11] Burnim J and Sen K 2008 Heuristics for Scalable Dynamic Test Generation (Berkeley: University of California)
[12] The Z3 Theorem Prover from Microsoft Research Available from: http://www.github.com/Z3Prover/z3
[13] Kenneth M A 1999 Lecture 25: Control Dependence Graph (Computer Science university of Colorado boulder)