Modification of ALL – SAT solver to search verification kits in testing

I A Lyapunova¹ and N A Fomenko²

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. To date, systems are being actively developed that imply automatic proof of software compliance with the so-called specification, i.e. requirements for it. However, in most projects, testing is still the basis of the verification process. Often this is done by people whose task is to develop tests for finding errors made by programmers in the code. Simplified tests represent a set of “input” parameters supplied to the entrance to the program, and upon completion of its work, the “output” parameters are compared with the “expected” ones according to the logic described in the specification. Testing can determine how repeated program execution with the intention of finding errors in it, but this in turn does not prove its correctness. This paper discusses the use of SAT solvers to solve the problem of finding verification kits for software testing and the development of an ALL-SAT solver modification).

1. Introduction

Verification is the study and justification of the fact that the program corresponds to its specifications. In other words, it is the process of finding errors in a program or proof that they're not.

The basic methods of verification tests. Consider the main stages of SOFTWARE development and approximate distribution of introduced and detected errors (figure 1). Phase one includes analysis of the subject area, user requirements and is completed development of software specifications. During the design process occurs the development of the architecture and it is detailing. Implementation – writing the code programs. Further testing is carried out (component and system), and if it does not give information about code errors, the program can be transmitted for operation [1 – 5].

SOFTWARE verification methods can be divided into testing and formal methods, based on the use of mathematical apparatus implemented in languages, methods and software specification and verification tools.

In the task of testing an important point is to justify the completeness of the test coverage. As a rule, in one form or another, this justification is reduced to the confirmation of execution of all functions specifications. It can be performed by the method of "black box" (black box testing) with full coverage of the input data and the method of "white box" (white box testing) with full coverage of the program code.
In the task of testing an important point is to justify the completeness of the test coverage. As a rule, in one form or another, this justification is reduced to the confirmation of execution of all functions specifications. It can be performed by the method of "black box" (black box testing) with full coverage of the input data and the method of "white box" (white box testing) with full coverage of the program code.

Automatic SOFTWARE validation presents a significant problem in the absence of strict standards for specification development and code writing. Besides however, these methods have the following significant drawbacks: slow speed, human intervention may be required and in General it is not possible to build a complete system of axioms and rules of withdrawal. Thus, software testing remains the most popular and effective SOFTWARE verification procedure for the majority of large projects where the introduction of formal methods is not possible. The testing process can be divided into 3 stages (figure 2):

![Testing Process Diagram](image)

**Figure 2.** Testing Process Diagram.

Specification analysis is the process of familiarizing the tester with requirements of the software. The most time-consuming and long process is the development of tests, which can be from 30 to 60% of the total complexity of creating programs. This is related with the fact that the tests are developed "manually", at a low level of automation and before the task of the tester is to develop such a number of tests that would allow find all possible errors in the program code. Often the features of the process developments require verification of the software by the "black box", when we have to work only with the input and output parameters of the program, making tests that meet the requirements, which in turn leads to the problem of finding a sufficient number of different combinations of input conditions, the complete search of which either it is not effective and leads to a significant slowdown in the test run, or it is not possible due to an infinite number of such combinations [4 – 6].

When testing the "black box" the main task is to cover the logic, presented in SOFTWARE requirements. Testing all possible paths of the algorithm generally is impossible, because the only theoretically justified way of implementation this is to iterate through all combinations of output parameters. Even if the test is to be carried out on a model consisting exclusively of parameters of Boolean type, then you will need to create a test set that contains all the input combinations. Obviously, that not only run tests, but also their generation with a large number of input parameters N, will be produced for astronomically large time $T(N)=O(2N)$. Despite the seeming simplicity of the majority of the software requirements, which can be two inputs and one output, this output can be an
internal parameter for the whole system and be supplied as an input parameter in another requirement. In such cases, the formation of one output parameter can involve a large number of inputs.

Due to the fact that testing all possible paths of the algorithm is generally impossible, combinatorial condition coverage criteria are used. It requires a sufficient set of tests that has the following properties:

- causes all possible combinations of results of conditions in each decision;
- transfers control to each entry point at least once.

In the process of developing tests, the output area of the program is indirectly divided into a finite number of equivalence classes. It occurs in such a way that it can be assumed that each test belonging to some class is equivalent to all other representatives of this class. So we can assume that there is a test case that will consist of tests that do not belong to the same class in pairs. We will call such a set a minimal test set.

As you can see, the combined coverage of conditions is not an easy task, because even with the constructed partition of the input conditions, the number of combinations is usually very large. If there is no systematic choice of a subset of combinations of input conditions, the probability of finding all errors of the program is very small.

To ensure good test coverage in complex algorithms, it is often necessary to use the method of functional diagrams. It helps to systematize the selection of good tests, as well as indirectly allows you to detect flaws in the specification.

The function diagram (figure 3) is the formal language into which the specification is translated. It can be compared with combinatorial logic network, which is adopted in electronics. It displays the logical relationships of input, internal, and output parameters. Through the methodical tracing state conditions, the charts it converted to a solution table with restricted inputs. Each column in the decision table corresponds to a test.

![Figure 3. Example of a functional diagram.](image-url)
This method allows you to find the optimal test set, but it is used infrequently.

- most of the requirements can be covered by tests compiled "in mind", because of its simplicity;
- it is often more important for experienced testers (for example, due to time constraints) to develop a set of tests that does not provide full coverage, but corresponds to their ideas about the possible localization of SOFTWARE errors.

The method of functional diagrams (figure 3) is remarkable in that it is some kind of logical formulas into which the algorithms presented in the specification can be converted.

Thus, it makes sense to consider the possibility of generating test cases based on the logical formula to which the specification is translated. Logical semantics is the simplest way to determine the logic of a program.

2. Problem statement

There are many algorithms that can be used to match a model that can be easily translated into a logical formula. In this case, all variables are of non-Boolean type and operations on them should be omitted or replaced with Booleans, if possible. Methods for solving such combinatorial problems as generating automatic template tests for integrated circuits or checking the equivalence of logic circuits are widely known. They involve the use of SAT solvers to prove the equivalence of combinational schemes. The use of SAT and SMT solvers is increasingly common in formal verification methods, but they always require not only a strict specification description, but also strictly standardized code.

Stand-alone testing tools (CAT, SAT) – a program or some script that simulates the environment for a particular module of the software under test and allows you to test this module in isolation from the system.

Typically, each module in addition to the input and output data has several dependencies in the form of additional input or data from the system. In order for the module to be tested, it is necessary to simulate or suppress these dependencies.

Module testing is a procedure of testing individual subroutines or a certain set of program functions. The implication here is that before you begin testing a program as a whole, you should test the individual small modules that make up the program. This approach allows you to control the combinatorics of testing, since the initial focus is on small modules of the program. Just do not forget about the easier debugging, testing speed and the ability to parallelize testing modules.

Testing, having obvious advantages, also faces a number of problems: for large sequential or parallel programs it is very difficult to sort through all the input data, and for dynamic structures it is impossible; the huge size of the required coverage; full coverage does not guarantee the absence of errors. At the same time, if testing allows to detect frequent errors, the formal methods are aimed at finding critical errors. This is due to the fact that the use of formal methods is often associated with much greater resources and time. It should be noted here that testing and formal methods are not mutually exclusive methods of SOFTWARE verification and can be performed together, increasing the efficiency of error detection.

Development of ALL - SAT - solver. The search of all performing sets of Boolean formulas is widely used in such areas as checking unlimited character models, QBF-problems, Boolean optimization problems and many others [7 – 9].

To implement the ALL - SAT solver, the most obvious possibility is to take the algorithm of the existing SAT solver as a basis. This way is also one of the most effective.

The problem of the feasibility of Boolean formulas (the Boolean Satisfiability Problem or SAT) is the task, which is to determine whether such variable values exist, when this Boolean formulas (the set of variables, parentheses, and the operations of conjunction, disjunction and logical negation) get the value "true". SAT is NP-complete problem, therefore, it is extremely important for the theory of algorithms, and its solution is able to give an answer to question of equality of classes P and NP.
Currently, the task of SAT is a huge number of studies – both practical and theoretical. Every year competition programs are held, so-called SAT – solvers. Such programs are actively used in applied areas: automation of chip development, cryptanalysis, artificial intelligence and much more. The "NP-completeness" of the SAT is that it boils down to a large number of problems from the NP class in polynomial time (Cook-Levin theorem). Therefore, the use of the SAT solver (figures 4, 5) may be useful for the solution common problems such as scheduling in educational institutions or construction rational route's (task salesman's.) For implementations multiplatform it was decided to write a program based on the use of the web-technologies (HTML, CSS, JavaScript). Thus, this PAT solver is a web page with an input field and a button that starts the algorithm for solving the problem.

![Figure 4](image1.png)

**Figure 4.** An example of the script of the SAT – solver when the formula is executable.

![Figure 5](image2.png)

**Figure 5.** An example of the script of the SAT-solver when the formula is not feasible.
The easiest way to find all the solving sets is to change the algorithm so that after each found solution \( l \), a disjunct of the form \( \bar{l} \), is added to the original formula, which imposes a restriction on finding the same solving set [4 – 6]. And so appends the negation of each of the decisive set to the original CNF as long as the formula does not become insoluble.

You can also change the DPLL algorithm so that in the case of finding solutions internal mechanism of conflict analysis it returned calculations on the last non-confrontational level in the assignment tree.

Due to the fact that the number of solutions to the formula is large enough, it was decided to use the second approach to finding all performing sets of Boolean formula.

Despite the fact that the SAT problem is NP-complete and the algorithms for solving it in polynomial time are unknown, there is a special case – 2-SAT, which is solved in linear time. It also consists in finding such a set of Boolean variables that the formula addressed in the unit, but the input is a formula in 2-CNF, i.e., each clause consists of two literals.

The algorithm is based on the construction of an implication graph, the vertices of which are all literals of the formula and their negations, and the edges correspond to the implicative connections.

Note here that for the 2-SAT problem to be feasible, it is necessary and sufficient any literal \( x_i \) the vertices to be in different components of the strong connectivity of the implication graph. This criterion can be checked in time \( O(N+M) \) with the help of search algorithm for strongly connected components.

Now we construct an algorithm for finding the solution of the 2-SAT problem. It is worth noting that although a solution exists, for some variables it may be true that \( x_i \) is reachable from \( x_i \), or (but not simultaneously) \( x_i \) is reachable from \( x_i \). In this case, select one of the the values of the variable \( x_i \) will lead to a contradiction, while the choice of the other will not. We will choose from two values that one that does not lead to contradictions. After selecting a value, we have to run a depth/width bypass from it and mark all values that follow from it, i.e. are achievable in the implication graph. Accordingly, there is no need to make any choice for already marked vertices, for them the value is already selected and fixed. The following rule applies only to still unmarked peaks.

Let \( \text{comp}[v] \) denote the number of the strong connectivity component to which vertex \( v \) belongs, and the numbers are ordered in the order of the topological sorting of the strong connectivity components in the component graph (i.e., the earlier ones in the topological sorting order correspond to the larger numbers: if there is a path from \( v \) to \( w \), then \( \text{comp}[v] \leq \text{comp}[w] \)). Then, if \( \text{comp}[x_i] < \text{comp}[\bar{x}_i] \), then choose the second value, otherwise choose \( x_i \).

We prove that with this choice of values we will not come to a contradiction. Let, for certainty, the vertex \( x \) is chosen (the case when vertex \( x \bar{}} \), is chosen is proved symmetrically).

First, let us prove that \( x_i \) of \( x_i \) is unreachabe. Indeed, since the number of the strongly connected component \( \text{comp}[x_i] \) is greater than the number of the component \( \text{comp}[\bar{x}_i] \), it means that the connected component containing \( x \) is located to the left of the connected component containing \( \bar{x} \), and from the first cannot be achieved last.

Second, we prove that no vertex \( x_j \) reachable from \( x_j \) is "bad", i.e. it is not true that \( x_j \bar{)} \) is reachable from \( x_j \). We prove this by contradiction. Let \( x_j \) is reachable from \( x \) and \( x_j \) is reachable from \( y \). As \( x_j \) is reachable from \( x \), then, by property implications of the graph \( x \bar{)} \) is reachable from \( y \). But, by assumption, \( x_j \) is reachable from \( x_j \). Then we get that \( x \bar{)} \) is achievable from \( x_i \), which contradicts the condition that was required to prove.

This algorithm finds the required values of variables under the assumption that for any variable \( x_i \) vertices are in different components of strong connectivity.

Now we can put the whole algorithm together and build an implication graph. Find in this graph the components of strong connectivity for the time \( O(N+M) \). Let \( \text{comp}[v] \) be the number of the strong connectivity component to which vertex \( v \) belongs. Check that for each variable \( x_i \) vertices lie in different components, i.e. \( \text{comp}[x_i] \neq \text{comp}[\bar{x}_i] \). If this condition is not executed, then return "solution does not exist". If \( \text{comp}[x_i] > \text{comp}[\bar{x}_i] \), then select true for \( x_i \) variable, otherwise - false.
3. Summary
The resulting solver can be run on most devices that have a browser. Specific interest is the possibility of implementing such algorithms on the GRID system [4]. During debugging and testing of the program it was found that ALL – SAT – solver works correctly and copes with its tasks [4, 5]. It can also be concluded that the resulting solver is significantly inferior in performance implemented C++ MiniSAT [4, 5], but during debugging and testing of the program it was found that the solver works correctly and copes with its tasks. Tests were also conducted on mobile devices and different versions of browsers.

References
[1] Chauhan N 2010 Software Testing: Principles and Practices (Oxford University Press) p 608
Leslie A. (Schad) Johnson 1992 DO-178B, 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 Proc. 22nd IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE 2007) (NY: The Association for Computing Machinery, Inc.) pp 571-2
[3] Bogdanov D S, Lyapunova I A and Tetroashvili E V 2017 The task of developing a SAT-solver for finding verification sets in software testing Available from: http://ivdon.ru/uploads/article/pdf/IVD_139_Kazharov.pdf_129d7fc19d.pdf
[4] Li A H, Lyapunova I A and Fomenko N A 2018 IOP Conf. Ser.: Mater. Sci. Eng. 450(5) 052008
[5] Kazharov H A, Lyapunova I A and Chistjakov A 2015 Software implementation of the numerical solution of the inverse problem of transport materials Available from: http://ivdon.ru/uploads/article/pdf/IVD_139_Kazharov.pdf_129d7fc19d.pdf
[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
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
[7] Awedikian Z 2009 Automatic Generation of Test Input data for MC/DC test coverage: Soccer Lab (Ecole Polytechnique de Montreal) p 38