Model Checking for Generation of Test Suites in Software Unit Testing

Vasilios Almaliotis Panagiotis Katsaros Konstantinos Mokos
Department of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
{valmalio,katsaros}@csd.auth.gr, mokosko@otenet.gr

***Abstract*—Model checking is a technique for exhaustively searching the model's state space for possible errors. Testing is a common method for enhancing the quality of a software product by checking for errors in program executions sampled according to some criterion called coverage criterion. Testing is a costly process especially if it is not supported by an appropriate method (and tool) for generating test suites, i.e. sets of test cases that fulfil a specific coverage criterion. This work introduces a method based on model checking, that supports generation of test cases for coverage based unit testing. As a proof of concept, we provide results obtained from the developed prototype tool support. Our method relies on automatically deriving from the source code a SPIN model with injected breakpoints. Test cases are obtained as counterexamples of violated Linear Temporal Logic (LTL) formulae that are automatically produced based on the selected coverage criterion.**

***Keywords*—Software testing, Model Checking, Unit Testing, Coverage Criteria.**

I. Introduction

Software economics reports and related bibliography[1] mention that more than 50 percent of the total cost of a software project is expended in testing . According to [2], software testing can show the presence of bugs, but never their absence. On the other hand, model checking is an effective technique applied to a model of the system that is exhaustively checked in order to detect all existing errors. Although software testing and model checking have been traditionally dealt as separate verification and validation activities a few recent works [3, 4, 5] invest on the potential of model checking towards reducing the cost of software testing.

These approaches are based on the capability of model checking algorithms to discover a model execution trace for system properties that are violated. Every such trace is known as counterexample and in fact constitutes an execution sample that can be used as a test case. The potential of model checking to be used in the generation of test cases is hampered by its limited applicability in the analysis of real-scale software systems, due to the well-known problem of state space explosion.

The work described in this article focuses on the use of model checking in the testing of the smallest units of software design that are commonly referred as components or modules. The generated test cases form the test suites for unit testing, an activity that aims to uncover errors within the boundary and the constrained scope of the module.

In the first step of the proposed method the source program is parsed and abstracted into what we call a model program. Model programs are expressed in PROMELA, the model input language of the SPIN model checker [6]. Basic blocks of the model program are differentiated by the injection of a breakpoint, which is used as target in the path selection phase. In this second phase breakpoint targets are selected based on the chosen coverage criterion and for every selected path an appropriate linear temporal logic formula is derived. The counterexamples obtained from the model checking of the generated LTL formulae form the test suite for the chosen coverage criterion. The prototype tool support that is currently available implements the multiple condition coverage criterion [1], which is considered as the most comprehensive control flow coverage alternative. It is relatively straightforward to implement a less costly coverage criterion such as edge/condition coverage but in the current prototype this requires intervention in the source code of the second level of processing.

In section II we review related work and at the same time we highlight the differences of our proposal. Section III sets the definitions for the representation of the model program and section IV presents the automated verification of the model program for the generation of the test suite. Section V provides the results obtained for an example Java program and the paper concludes with a summary of the presented method, comments on its potentialities and future research prospects.

II. Related Work

White box testing based on test cases generated by model checking has been previously addressed in [7, 8, 9, 10, 11]. The common denominator of the aforementioned techniques is the development of an appropriate set of LTL formulae that when applied to the model program produces the requested set of test cases. The model program represents the control flow of the unit under test and although its existence is a prerequisite, little has been done for how to automatically derive it from the source code. Some of the mentioned works try to reduce the cost for the generation of test cases by employing heuristic algorithms [10] in order to minimize the set of LTL formulae in the test suite.

In this article we focus on the automated extraction of the model program from the source, in an attempt to further reduce the cost for the generation of test cases. The created model

```
int method(int a, int b, int c) {
    int d = 0;
    if(a<10) {
        if(b>=20 && a<5) d=50;
        else d=0;
    }
    else {
        if(b<50) d=30;
        else d=10;
    }

    if(c==3) d=40;

    return d;
}
```

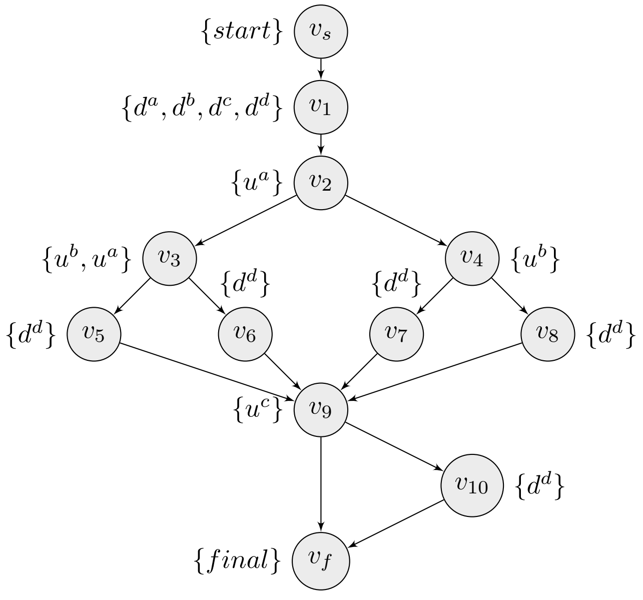

Fig. 1. Source code and the corresponding flow graph representation of a software unit

program is constructed exclusively for test case generation, which means that additional information is embedded and the program undergoes suitable transformations, in order to aid the specific model checking process that yields the expected test cases.

Also, in the works reported in related bibliography the LTL formulae used in the generation of the test suite are instances of property patterns whose syntax is relatively straightforward for a human being, but their production is not easily automated. For example, in [11] LTL formulae contain the values used before and after the execution of a selected transition, as well as the guard that enables its execution. In general it is difficult to devise an algorithmic approach that will collect this information from the model program. Our approach eliminates the need to specify the aforementioned information within the used LTL formulae by abstracting it at the level of the model program. In this way, the needed LTL formulae are kept as simple as possible and it is thus possible to support end-to-end automation from the source program to the generation of the expected test cases.

III. Problem Definition

This section introduces the adopted representation of the software unit's source code as a model program and discusses the use of LTL for the generation of the test suite corresponding to a coverage criterion.

A. Model Program

The system under test (SUT) is represented by a Kripke structure $\mathcal{T} = (S, I, \mathcal{A}, \delta)$, where S is a finite set of program states, $I \subset S$ is a non-empty set of initial states, $\mathcal{A}$ is a labeling function such that $\mathcal{A} : S \rightarrow 2^{AP}$ with AP the set of all atomic propositions and $\delta \subseteq S \times S$ is a total transition relation.

A *flow graph* $G = (V, v_s, v_f, A)$ is a directed graph, where V is the finite set of total states, $v_s \in V$ is the initial vertex, $v_f \in V$ is the final vertex and $A \subseteq V \times V$ is the finite set of arcs, that connect the states. Each vertex represents a statement and arcs determines the flow between the statements. The point of interest in each vertex, is the definition and use of variables that take part in program. For a variable a, we denote its definition as d^a and its use as u^a.

The Kripke structure T represents the SUT if $S = V$, $I = \{v_s\}$ and $\delta = A \cup \{(v_f, v_f)\}$. By expressing the SUT with a Kripke structure, which is a widely used system representation in model checking, we set a basis to automatically obtain the program paths needed for the coverage criterion of preference. Each program path is a test case and the set of test cases obtained for a given coverage criterion forms a test suite.

B. Linear Temporal Logic

The Kripke structure T reflects all possible program behaviors and it is used for model checking LTL formulae defined over the set AP of atomic propositions. LTL makes it possible to express patterns of succession of events, without having to explicitly refer to the time that events occur. A syntactically correct LTL formula is produced by the use of the following grammar rules: $\varphi ::= true \mid \alpha \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 U \varphi_2$ with α representing atomic propositions over the set of program's variables, O for the temporal operator next and U for until.

Two other operators derived from those shown in the grammar rules are the operator $\Diamond$ (eventually) and the operator $\Box$ (always).

Related works on the generation of test cases use LTL formulae to search for paths that contain a specific sub- path or a given state. These approaches can be used for developing test suites that provide coverage of the branches of the source code and the coverage criteria used are the most common criteria in the practice of software testing [11]. The obtained path is given by predicates defined over the program's variables as precondition and postcondition, as well as by the guard clauses of the intermediate transitions.

Model checking has been also used for data flow testing [10]. In this case an appropriate LTL formula searches for program paths between specific operations, like the definition and the use of one or more named variables. For the control flow graph of Figure 1 branch coverage will generate test cases that will cover vertices $\{v_3, v_4, v_5, v_6, v_7, v_8, v_{10}\}$ and dataflow coverage would search for example all possible paths between definition and usage of program variable d. However, as stated in [9] control flow statements affect dataflow coverage and this dependence can be encoded in LTL formulae.

IV. Model Checking for Coverage Based Unit Testing

The proposed method for coverage based unit testing relies on the model program representation of the previous section. The control flow branches and the states of the model program suffice for the expression of LTL formulae that detect program paths for a given coverage criterion, but the form of an appropriate LTL formula is inevitably complicated. Instead of this, the implemented approach is not based solely on the derivation of an appropriate formula, but enhances the model program representation with additional information for the construction of the alternative program paths. A direct impact is that the formulae used for the detection of program paths are simplified, i.e. part of the work required for the development of appropriate LTL formulae is transferred to the construction of the model program. We believe that this approach opens prospects for automating the development of LTL, formulae thus raising the degree of automation in the construction of a test suite.

A. Translation of Control Flow Statements

Program execution flow is affected by control flow statements like *if-then, if-then-else, switch* and condition controlled loops. In all cases, program paths are determined by the decisions made as a consequence of conditional expressions with one or more predicates combined with Boolean operators. The fundamental idea behind branch coverage is to create test cases that will cover all edges of the program's control flow graph. Multiple condition coverage is another criterion that requires a test case for each different combination of decisions in a conditional expression.

Every predicate that cannot be broken into smaller predicates is called a clause. The mutations of a given clause, say c_i, are in the set $M_i = \{c_i, \neg c_i\}$ and consequently each decision made for a conditional expression is given as an l-ary boolean function $d : M_i^l \rightarrow B$, where $B = \{0, 1\}$ and l is the number of clauses in the expression. We define the function f that for each $d(m_i^l)$, where $m_i^l \in M_i^l$, returns the basic block b for the edge corresponding the decision. Thus, for a conditional statement with an else part, say e, we have

$$(f(d(m_i^l)) = b) \wedge (f(\neg d(m_i^l)) = e).$$

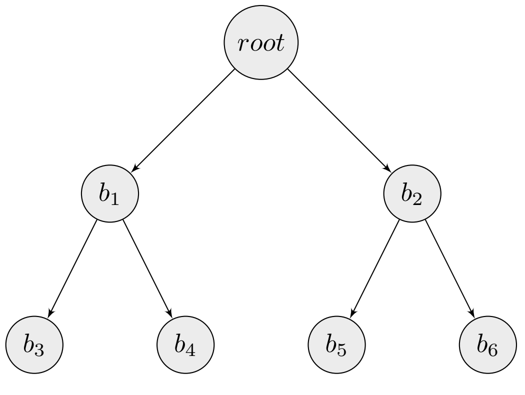

Fig. 2. Breakpoint hierarchy tree of program in Figure 1

The described interpretation leads to different program paths that a model checking algorithm can distinguish if there is a way to identify the different states reached by the model program. To support this we insert additional information during the source to model program translation. This information is used as an analogue to the breakpoint and it is utilized in path selection. For an *if-then-else* statement the proposed abstraction results in the following model program representation

$$(f(d(m_i^l)) = b \wedge breakpoint_{z,j,n}) \wedge \\ (f(\neg d(m_i^l)) = e \wedge breakpoint_{z,j,n+1})$$

for all $m_i^l \in M_i^l$ and for the n_{th} decision of the zth control flow statement found in the basic block of the j_{th} level of nesting. The model program is enhanced with one breakpoint namely $breakpoint_{s,n}$ for every entry point and a separate breakpoint, say $breakpoint_{f,n}$ for each return statement of the unit's code. From now on, SP and EP will respectively represent the set of all entry points and the set of all exit points.

In a post-processing phase, the breakpoints are organized into tree structures, one for each non nested control flow statement, that are used for optimizing the number of test cases needed for the coverage criterion at hand. Leafs of the aforementioned trees cite the breakpoints that do not dominate any other breakpoint of the hierarchy. In essence, as it is shown in Figure 2, breakpoints are organized by their level of nesting, also called depth. Each tree includes all possible data flows between breakpoints in the corresponding control flow statement with possibly multiple nesting levels.

B. Patterns of LTL formulae for the generation of test cases

The enhanced model program provides a basis for finding program paths (i.e. test cases) returned as counterexamples of model checking runs. Appropriate formulae that describe program paths comply with a simple property pattern and we call them witnesses. Due to their simplicity they can be produced automatically, if we assume access to the model program and the trees of breakpoints.

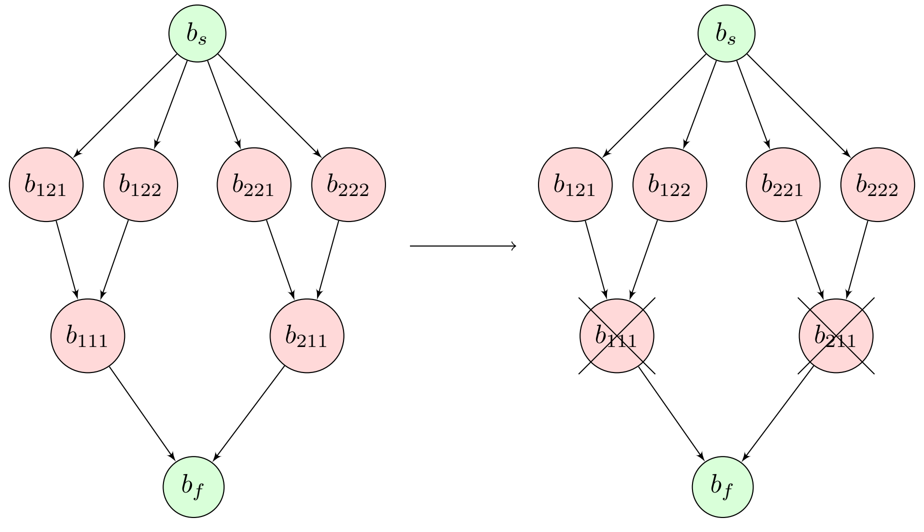

Fig. 3. Reduction of necessary breakpoints of program in figure 1

For utilizing the capability of model checking algorithms to detect a counterexample, every witness is turned into a trap property, i.e. if p is a witness we obtain a test case by model checking $\neg p$.

The pattern of LTL witnesses is

- **ltl** $::= \Diamond\,(breakpoint_s \wedge \Diamond\,(\textbf{subltl}))$ with $breakpoint_s \in SP$
- **subltl** $::= breakpoint_i \wedge \Diamond\,(\textbf{subltl})\ \mid breakpoint_f$ with $breakpoint_f \in EP$

All paths to be covered are represented by LTL trap properties that are produced automatically. The number of needed LTL properties depends on the coverage criterion at hand. Multiple condition coverage requires comparatively higher number of test cases form other control coverage alternatives. Finally, we note the requirement that at least one test case should be included in the test suite for all entry points and all exit points.

C. Deriving Combinations of breakpoints automatically

Given the model, the pattern based on which LTL formulae can be constructed and the set of breakpoints organized in a tree structure, it is possible to combine these breakpoints under the restrictions of the LTL pattern, in order to automatically produce LTL formulae that will generate the test cases.

The simplest method to combine the breakpoints, is a naive combination without discriminating them. This method, has as result, the production of a large number of formulas. Also, some of which will be invalid, because of the wrong execution order of the combined breakpoints. Another problem is the lack of information for the path lengths.

A more intelligent method, utilizes information available in the model, in order to minimize the breakpoint combinations. Based on this information, it is possible to derive an order for the breakpoints involved in the LTL formulae, since code is executed sequentially. Ley us assume that apart from the sets SP and EP, of breakpoints, our analysis also yields the sets IF_1 and IF_2. In this case, the set of possible combinations is: $C = SP \times IF_1 \times IF_2 \times EP$. In the general case, for a program with n control statements the set of possible breakpoint combinations is $C = SP \times \prod_{i=1}^{n} IF_n \times EP$.

Sequential ordering results in a restricted set of breakpoint combinations ,that however still includes redundant paths. The reason is that some breakpoints dominate other breakpoint meaning that the paths for the dominated breakpoints also contain other breakpoints. By filtering sets of breakpoints in order to keep only the dominated ones, we further reduce the length of LTL formulae that are used for test case generation. Let us denote by FIF_i the filtered set IF_i of breakpoints. Then, the set

$$C = SP \times \prod_{i=1}^{n} FIF_n \times EP$$

will correspond to test cases that represent distinct paths.

In algorithm in Figure 1, we see the main process of the combination of non-dominant breakpoints. The function *getNonDominantsBreakpointsList*, is not described in depth, because its purpose is to retrieve the leafs of the given trees. That is happened because the tree is structured in such way that leaf nodes are not dominating parent nodes.

For the sake of completeness, we have to mention that some of the paths that are generated, may be invalid. That happens, because sometimes control statements neutralize each other. Another approach to overcome this difficulty is to create LTL formulas, is to create a test case for one control statement at time. This will lead us to create the follow set:

$$C = \{\forall i \in [1, n], i \in \mathbb{Z} | SP \times IF_i \times EP\}$$

Once the LTL formulas are created, the remaining process contains the execution of them, in a model checker. We use the SPIN model checker [6], to check the properties in a model coded in PROMELA.

Algorithm 1 Generate LTL formulas

```
Procedure $constructLTL(breakpointTrees)$
1: for $\forall breakTree \in breakpointTrees$ do
2:     nonDominantVector ← getNonDominantsBreakpointsList($breakTree$)
3: end for
4: findAllCombinations(nonDominantVector)

Procedure $findAllCombinations(nonDominantVector)$
1: if nonDominantVector.size $> 1$ then
2:     nonDominands ← nonDominantVector.pop()
3:     nextTreesCombinations ← findAllCombinations(nonDominantVector)
4:     for $\forall nonDominant \in nonDominands$ do
5:         for $\forall nextTreesCombination \in nextTreesCombinations$ do
6:             combinationList.add(nonDominant,nextTreesCombination)
7:         end for
8:     end for
9:     return combinationList
10: else
11:     return nonDominantVector
12: end if
```

V. Example

In this section, we will demonstrate the principle of our theory, with a simple example. We will apply the method, to the program in Figure 1. With this small example, we concentrate in the process of automatic creation of the model and LTL formulas. In other words, we skip some of model extraction drawbacks, to illustrate the contribution of this article.

First, we have available only the source code of method. Given this source code we have to apply some transformation to produce the corresponding model. This transformation is the application of multiple condition coverage criterion. This means that for a control statement, we have to create a number of alternative paths, equal to the possible combinations of the predicates in control statement. For example, the follow control statement which is taken from Figure 1 (line 4), will be transformed as:

```
//In source
if(b>=20 && a<5) d=50;
else d=0;
//In model
if(b>=20 && a<5) d=50;
else if(b>=20 && !(a<5)) d=0;
else if(!(b>=20) && a<5) d=0;
else d=0;
```

With this transformation, we are able to force the model checker to reach the specific paths. The next step is to insert the breakpoints in order to discriminate those paths. For our convenience, we represent the transformation in Java code, because of the excessive length of the PROMELA code. Finally, the program from Figure 1, will be transformed as:

```
int method(int a, int b, int c) {
    short breakpoint=0;
        int d = 0;

    if(a<10) {
        if(b>=20 && a<5) {
            d=50; breakpoint=5;
        } else if(b>=20 && !(a<5)) {
            d=0; breakpoint=6;
        } else if(!(b>=20) && a<5) {
            d=0; breakpoint=7;
        } if((!b>=20) && !(a<5)) {
            d=0; breakpoint=8;
        } else {
            d=0; breakpoint=9;
        }
        breakpoint=1;
    } else {
        if(b<50) {
            d=30; breakpoint=10;
        } else {
            d=10; breakpoint=11;
        }
        breakpoint=2;
    }

    if(c==3) {
        d=40; breakpoint=3;
    } else {
        breakpoint=4;
    }

    breakpoint=12;

    return d;
}
```

Note, that in the PROMELA model, the if statement transformed in an if-else statement, in order to separate the branches for test case generation.

From this transformed code, we can identify four different breakpoint trees: entry point tree, first control command tree, second control command tree and exit point tree. Only the control command trees are true trees, because entry and exit point trees contain only one root and one child. The total sets of breakpoints we take from the program are:

$SP = \{b_0\}$
$IF_1 = \{b_1, b_2, b_5, \ldots, b_{11}\}$
$IF_2 = \{b_3, b_4\}$
$EP = \{b_{12}\}$

After the breakpoint reduction in the previous sets we keep only the non-dominant breakpoints. In this example, the only set that is affected by this process, is the set of the first control statement. This set became:

$$IF_1 = \{b_5, \ldots, b_{11}\}$$

We observe that from the previous set, the process excludes only breakpoints with number 1 and 2.

The last step is to combine the filtered sets in order to obtain the final paths. The final set that include all combination of filtered sets is the set:

$$C = \{b_0\} \times \{b_1, b_2, b_5, \ldots, b_{11}\} \times \{b_3, b_4\} \times \{b_{12}\}$$

After this step, we have available the total LTL formulas and the method's model. The last step is to perform the check, by applying those LTL formulas in model, with a model checker. After the model checker execution, we take the actual test cases in a form that inform us about the initial value of inputs and the values that variable has at the exit of the method. The implementation of this attribute, is hard-coded in the model.

VI. Conclusion

We introduced a method for generation of test cases in coverage-base unit testing. The method relies on the transformation of the source code to a SPIN model with injected breakpoints. The breakpoints differentiate the program's basic blocks and at the same time provide sufficient information for the generation of test cases by model checking simple LTL formulae. Automation has been achieved not only in the program to model transformation, but also in post-processing the breakpoints of the model program for the production of a test suite for some preferable coverage criterion. We provided results obtained from the prototype tool support.

Our work demonstrates the feasibility of automated generation of test suites by the use of model checking. Many of the problems encountered in related work have been successfully addressed, in an attempt to provide a higher degree of automation in unit testing. We experienced problems like for example the need for fine tuning the source to model abstraction in a way that eliminates the possibility of state space explosion, while at the same time the model retains the necessary fidelity for the generation of input values representing complete definitions of test cases. The program's execution environment and its role in the source to model transformation is an additional problem that has to be properly addressed. Last but not least, an interesting future research prospect is the automated generation of test cases for selected program variables, based on the well-known dataflow coverage criteria.

References

[1] G. J. Myers, *The Art of Software Testing, Second Edition*, 2nd ed. Wiley, June 2004. [Online]. Available: http://www.worldcat.org/isbn/0471469122

[2] O.-J. Dahl, E. W. Dijkstra, and T. Hoare, *Structured Programming*. London: Academic Press, 1972.

[3] P. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *ICFEM*, 1998, pp. 46–.

[4] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Software Eng.*, vol. 11, no. 4, pp. 367–375, 1985.

[5] J. Jacky, M. Veanes, C. Campbell, and W. Schulte, *Model-Based Software Testing and Analysis with C#*, 1st ed. Cambridge University Press, 2007.

[6] G. J. Holzmann, "The model checker spin," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[7] P. Ammann, P. E. Black, and W. Ding, "Model checkers in software testing," NIST-IR 6777, National Institute of Standards and Technology, Tech. Rep., 2002.

[8] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural, "A temporal logic based theory of test coverage and generation," in *TACAS*, ser. Lecture Notes in Computer Science, J.-P. Katoen and P. Stevens, Eds., vol. 2280. Springer, 2002, pp. 327–341.

[9] H. S. Hong and H. Ural, "Dependence testing: Extending data flow testing with control dependence," in *TestCom*, ser. Lecture Notes in Computer Science, F. Khendek and R. Dssouli, Eds., vol. 3502. Springer, 2005, pp. 23–39.

[10] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural, "Data flow testing as model checking," in *ICSE*. IEEE Computer Society, 2003, pp. 232–243.

[11] S. Rayadurgam and M. P. E. Heimdahl, "Coverage based test-case generation using model checkers," *Engineering of Computer-Based Systems, IEEE International Conference on the*, vol. 0, p. 0083, 2001.

[12] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[13] P. E. Black, V. Okun, and Y. Yesha, "Mutation operators for specifications," in *ASE*, 2000, pp. 81–.

[14] G. J. Holzmann, "Logic verification of ansi-c code with spin," in *SPIN*, ser. Lecture Notes in Computer Science, K. Havelund, J. Penix, and W. Visser, Eds., vol. 1885. Springer, 2000, pp. 131–147.

[15] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.

[16] G. J. Holzmann and M. H. Smith, "Software model checking," in *FORTE*, ser. IFIP Conference Proceedings, J. Wu, S. T. Chanson, and Q. Gao, Eds., vol. 156. Kluwer, 1999, pp. 481–497.

[17] E. M. Clarke, "The birth of model checking," in *25 Years of Model Checking*, ser. Lecture Notes in Computer Science, O. Grumberg and H. Veith, Eds., vol. 5000. Springer, 2008, pp. 1–26.

[18] T. Jéron and P. Morel, "Test generation derived from model-checking," in *CAV*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds., vol. 1633. Springer, 1999, pp. 108–121.

[19] A. Engels, L. M. G. Feijs, and S. Mauw, "Test generation for intelligent networks using model checking," in *TACAS*, ser. Lecture Notes in Computer Science, E. Brinksma, Ed., vol. 1217. Springer, 1997, pp. 384–398.

[20] H. S. Hong and H. Ural, "Using model checking for reducing the cost of test generation," in *FATES*, ser. Lecture Notes in Computer Science, J. Grabowski and B. Nielsen, Eds., vol. 3395. Springer, 2004, pp. 110–124.