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

I. INTRODUCTION

Reports on software economics mention that testing plays a significant role in software development, since more than 50 percent of the total cost of a software project is often expended in testing. Although, software testing can show the presence of bugs, it is inadequate for showing their absence and requires highly skilled engineers. Traditionally, software testing and model checking are dealt as separate verification and validation activities. However, recent works invest on the potential of model checking towards reducing the cost of software testing [1, 2]. The common denominator of these techniques is the development of an appropriate set of linear temporal logic (LTL) formulae that when applied to the program model produce a set of test cases for a given coverage criterion.

Specifically, in [1] the authors try to reduce the cost for test case generation by employing heuristic algorithms in order to minimize the set of LTL formulae in the test suite. The syntax of the used LTL formulae is relatively straightforward for a human being, but their production is not easily automated [2].

In this article we focus on the automated extraction of a model program from the source, in an attempt to reduce the cost for the generation of test cases for unit testing. The created model 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. By abstracting this information at the level of the model program we eliminate the need to specify it in the LTL formulae. In this way, the needed LTL formulae are kept as simple as possible and therefore we can support end-to-end automation from the source program to the generation of the expected test cases.

II. MODEL CHECKING FOR GENERATION OF TEST SUITES IN SOFTWARE UNIT TESTING

The proposed method for automatic generation of test cases in software unit testing relies on a Kripke structure representation of the model program and is performed in two phases as shown in Figure 1.

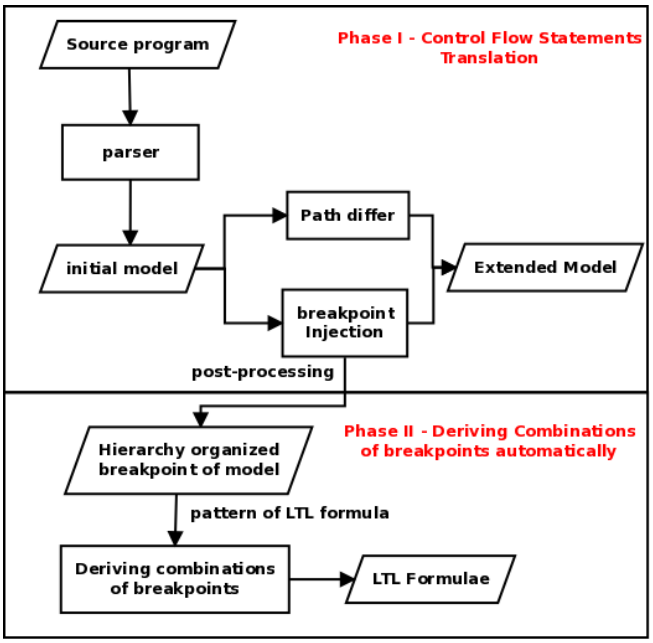

Figure 1. Proposed Method Process

During the first phase, the software unit is parsed and abstracted into an initial model program. The initial model program is enhanced with breakpoints and possible execution paths are thus differentiated during the automated translation of the flow statements to the control flow constructs of the extended model program. The model program is expressed in PROMELA, which is the input language of the SPIN model checker [3]. In the second phase, hierarchically organized breakpoints of the extended model program are selected based on the chosen coverage criterion (post-processing) that was determined during the path differentiation process of the previous phase. By automatically deriving combinations of breakpoints an appropriate LTL formula is created. The counterexamples obtained from model checking the generated LTL formulae, form the test suite for the coverage criterion at hand. The prototype tool support that is currently available implements the multiple condition coverage criterion [4], 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.

A. Control Flow Statements Translation

Program execution is affected by the control flow statements (if-then, if-then-else, switch-case and condition controlled loops) of a software unit. Program execution paths are determined by the decisions made at program locations with conditional expressions, where one or more predicates are combined with Boolean operators. The fundamental idea behind our approach is a path differentiation process during the automated translation of the control flow statements to PROMELA control flow constructs. More precisely:

- all edges of the program's control flow graph are covered and
- basic paths are built for each different combination of decisions in a conditional expression.

In the post-processing phase, breakpoints are injected at the end of each basic block and are used later to construct the LTL formulas. 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 point to the breakpoints that do not dominate any other breakpoint of the hierarchy.

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

Automatic construction of LTL formulas is based on the following pattern: every program execution path starts from an entry point, then accesses some of the basic blocks that exist in the body of the program and ends in an exit point. All aforementioned program locations are represented by breakpoints injected during the model construction and both the entry and exit points are kept in separate structures.

C. Deriving Combinations of breakpoints automatically

Given the model, the pattern on which LTL formulae will be based 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 method for automatic generation of test cases in software unit testing relies on the utilization of information that is available in the model and the non-dominant breakpoints that exist within the control flow statements. We select breakpoint combinations corresponding to paths that access multiple non-nested control statements in one execution. This may lead to an explosion in the number of test cases, but many of these combinations are not valid. By combining more than one tree structure between the entry and exit points additional information is obtained for execution paths that initially were not evident.

Alternatively, it is also possible to take all the non-dominant breakpoints within a control flow statement and create all combinations between these breakpoints and the possible entry and exit points of the program unit. This method is powerful enough if we are interested to test all the commands into a single unit, because test cases are built for all possible paths between an entry point and an exit point, paths that access basic blocks of control flow statements in the maximum nesting level.

The developed algorithm for combining breakpoints can be applied for both aforementioned approaches. For each tree structure of breakpoints, we choose the leaves that represent the non-dominant breakpoints of the model. Each set of non-dominant breakpoints is stored in a vector, which is then passed to a routine that recursively returns the Cartesian product of all sets of non-dominant breakpoints inside the vector. The obtained combinations are used to form the LTL formulae that will generate the test cases.

III. CONLUSION

We described a method for the 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 the preferable coverage criterion.

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, 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 data flow coverage criteria.

REFERENCES

[1] 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.

[2] 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.

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

[4] G. J. Myers, The Art of Software Testing, Second Edition, 2nd ed. Wiley, June 2004.