Model checking and test generation: towards a combined approach to software verification

Mihály Dobos-Kovács*, András Vörös*†
*Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
†MTA-BME Lendület Research Group on Cyber-Physical Systems

***Abstract*—Ensuring the correctness of safety-critical systems is a key aspect of the development process. Various approaches exist to find software bugs: (1) model checking examines the mathematical model of the software and proves the logical correctness, while (2) testing is an efficient and practical technique to find bugs. Model checking is a computationally expensive task, as it explores all the possible states of the software, and despite the technological advances of the last decade, software that cannot be formally verified still exists. On the other hand, testing is computationally cheaper than model checking, and widely used by the industry, but providing an efficient test suite for a given program is still under heavy research.**

The goal of my work is to combine model checking and testing to exploit the advantages of both approaches. I introduce a new algorithm that uses an abstraction-based model checking technique to explore the behavior of the software. In case the model checking algorithm proves the properties of the software, the procedure terminates. If the algorithm can not reach a conclusion, test generation is applied, exploiting the information gathered during state space traversal of model checking.

I. Introduction

Ensuring the correct behaviour of safety-critical systems is an important task during system development. Various approaches exist to find software bugs with their own advantages and disadvantages.

Model checking is one of such approaches that examines the mathematical model of the software and proves the absence of bugs, or provides a counterexample to correctness. Model checking is a computationally expensive task, as it usually explores all the possible states of the program. Despite the technological advances of the last decade, we are still unable to formally verify industrial software systems. There are two barriers: on the one hand, software model checking is an algorithmically undecidable task. On the other hand, when the analysis is restricted to programs with finite memory and finite data structures, state space explosion still prevents successful verification.

Testing [8] is an effective technique to find the flaws in the source code of software systems. Testing is a standard step in every development process, and it is also prescribed by certain standards. The challenge of testing is usually finding a proper test suite, that covers the behavior of the program while still having a feasible size. In the literature, several approaches were published that increased the efficiency of test suits [9] [10] [11] [12].

The goal of this work is to combine model checking with testing to exploit the advantages of both approaches. Model checking explores the state space to find errors or prove the correctness of the software. However, when the verification algorithm reaches a resource limitation, our new approach tries to use up the information gathered during the verification and generate a test set targeting the unexplored part of the program. The approach can focus testing to the critical parts of the program, and we hope that fewer test cases will lead to the more rigorous testing of software systems.

II. Background

Model checking is a common name of algorithms based on the rigor of mathematics. Model checking takes a formal model and a formal requirement, and verifies if the requirement holds on the model. To utilize model checking on computer programs, they need to be formalized. One of these formalizations is *Control Flow Automata* (CFA) [2], that consists of control locations and operations represented as edges.

One of the model checking algorithms is the so-called *CounterExample-Guided Abstraction Refinement* (or CEGAR for short) [2] [3] [4], that is used in our framework for exploring the behaviour of the software under analysis. The input of the CEGAR algorithm is a formal model (CFA in the case) and a formal requirement. The algorithm either proves that the given requirement holds on the given model, or proves otherwise, by giving a counterexample. Certain locations of the CFA are marked as error locations, and the given formal requirement is that a given error location is unreachable. Such error locations can be created from assertions in the program.

The state space of even a simple program can be huge if not infinite. To tackle this problem, CEGAR uses abstraction, such as explicit value abstraction [2] or predicate abstraction [5] [6]. In our framework, we chose predicate abstraction that follows a set of predicates (Boolean formulas over the set of program variables) instead of the concrete values of the variables. Henceforth an abstract state of a program is a set of (concrete) states that share the same control locations, and a set of predicates describing them.

The core of the algorithm is the so-called CEGAR-loop that consists of two distinct phases: (1) the abstraction and (2) the refinement phase. The task of the abstraction is to build the state space in the form of an abstract reachability

tree with the given set of predicates. If an erroneous state is encountered during the building phase, it is the task of the refinement to determine whether that state is reachable in the concrete state space as well. If an error location is reachable, then the program is unsafe, if it is not then more predicates need to be used [7], and the abstraction continues to build the state space. If the abstract state space contains no error locations, then the concrete state space does not either, as the abstract state space is an over-approximation of the possible state space of the program, so the program is safe.

III. Overview of approach

In our work, we aim to combine model checking and testing to analyze the safety of software (illustrated on Fig. 1). As a result of limited resources (time, memory, etc.), *formal verification* cannot always succeed. Should that happen, the verification task needs to be terminated, and *test generation* is applied. The test generation method can use the output of the model checking procedure: the *Abstract Reachability Tree*. The role of applying model checking is to decide the correctness of the software (safe depicted as a tick, unsafe depicted as a cross). However, when verification fails to reach a conclusion, then *test running* can still find bugs. If *testing* finds no errors either, then the safety of the program is undecided with the given resources (depicted as a question mark).

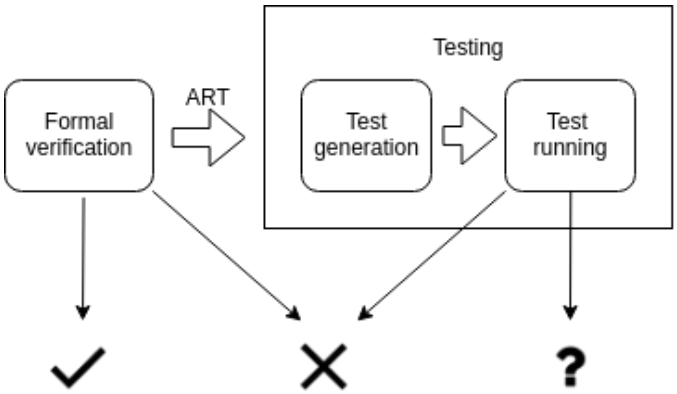

Fig. 1. Combining model checking and testing

Once the CEGAR algorithm terminates, the information gained during the traversal needs to be extracted. The algorithm stores this information in the so-called *Abstract Reachibility Tree*. Each node in the tree corresponds to an abstract state, while the children of the node denote the abstract states reachable via an operation from the parent node. Each node of the tree has one of the following four types:

- *Unreachable*: Nodes whose abstract state is part of the state space, but no input exists that drives the program to these states. These nodes can be removed from the tree.
- *Covered*: If a node A in the tree shares the same control location as a node B, and the predicates of A imply the predicates of B , then B is covered (by A).
- *Expanded*: A node is expanded if the tree contains the nodes that are reachable from that node via an operation.
- *Incomplete*: All nodes that are not unreachable, covered or expanded are incomplete.

Incomplete nodes represent the ”doorway” to the untraversed part of the state space, as all the possibly reachable states are reachable through them. (All the error states that are reachable from a covered node are reachable from the node that covers it, while all the error states that are reachable from an expanded node are reachable from one of its children.) The goal of test generation is to guide the program through these doorways, which can be achieved by creating an SMT problem [1] from the operations and guards on the path from the root to the incomplete node, and solving the problem for the input variable (note, that this method does not provide any coverage guarantee). This procedure will be detailed in the followings.

For example, a part of an Abstract Reachability Tree is depicted on Fig. 2. The node with l_3 is unavailable as no such x exists that satisfies $3 \leq x < 3$. The node with l_1 in the bottom left corner is covered by the node labelled l_1 in the center. The node with l_1 in the center and the node with l_0 are expanded, while the node labelled l_2 is incomplete.

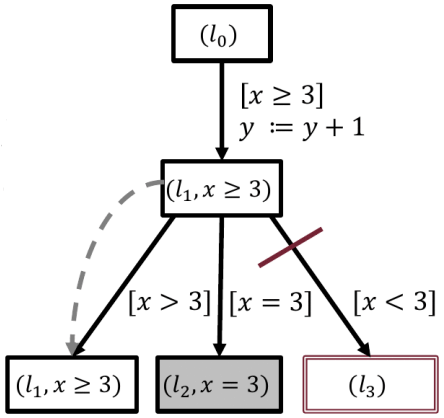

Fig. 2. (Part of an) Abstract Reachibility Tree.

IV. Generating test cases

Using the information extracted from CEGAR, test cases can be generated utilizing more approaches.

A. Boundary value analysis of input variables

Boundary value analysis is a black box testing technique, that assumes, that errors happen more frequently at the extreme/boundary values of variables. It is similar to testing based on equivalence partitioning, however, it focuses rather on the corner cases (and does not build equivalence classes explicitly). In our setting, we do local boundary analysis, that is motivated by traditional boundary analysis techniques, but focuses the boundary values by the unexplored part of the state space.

To do boundary value analysis, for each input variable the possible maximum and minimum values should be found. As mentioned earlier, an SMT problem can be constructed out of the operations on the path to an incomplete node.

Solving this problem gives one possible combination of many for the input variables. By giving the solver an optimization constraint, such as the value of one of the variables should be minimal/maximal, such a solution can be found, where the given variable is on one of its boundary values. Some solvers can solve the optimization problem [13].

The solution of the optimization problem is a combination of input values that guides the execution to the given incomplete node, while one of the input values is minimal/maximal. This minimal/maximal value can differ for the same variable if an other incomplete node is reached. These minimal or maximal values are local: the program can accept lower/higher input values than these boundary values, but on the given path, and on the state space that is reachable from the given incomplete node, these are the local minimal/maximal values. The computed values focus onto the unexplored part of the software.

To apply boundary value analysis systematically, the SMT problem should be solved twice for each input variable: for the first time the optimization constraint should be to minimize the current variable, for the second time to maximize it. Out of each solution of the SMT problem, a test case can be generated that tests the software for the minimal/maximal value of one input variable.

B. Robustness testing

The philosophy behind robustness testing is similar to boundary value analysis. The difference is that by robustness testing the errors are assumed to happen on the extremes of arithmetic conditional expressions.

The process of finding the necessary input values is similar to the method described in the previous subsection. The SMT problem with an optimization constraint needs to be solved, and using the solution, a test case can be generated. However the optimization constraint is not to minimize/maximize one of the input variables rather to minimize/maximize one of the variables that happen to be a result of an arithmetic expression.

For example, let us assume that $(z \leq 5)$ arithmetic conditional expression is given in a guard, where z is a positive integer. According to robustness testing the errors are more frequent on the extremes, so the possible minimal and maximal value of variable z should be determined, and used in the test cases. As the value of z might depend on the value of other variables, an SMT problem needs to be constructed and solved, as described earlier.

To apply robustness testing systematically, again the SMT problem should be solved twice for each variable in an arithmetic expression: for the first time the optimization constraint should be the minimization of the variable, for the second time the maximization. These test cases test the software for errors that happen in arithmetic conditions, for an input or computed variables on boundary values.

C. Finding number representation errors

In a computer program, every variable is stored on a finite number of bits. As a result, the range of every variable is a finite set (all the integer have a minimal and a maximal value, the floating-point variables are stored using the exponent and mantissa, etc.). A number representation error occurs when such a value is reached during the running of the software, that cannot be represented using the type of the variable.

For example. Let x, y, z be 4 bit unsigned integers (meaning, that the finite range of these variables is $\{0, 1, ..., 15\}$. Let $x = 8$ and $y = 8$ hold. If $z = x + y$, then z should be 16, which is not part of the domain of the variable, so it cannot be represented. This kind of error is called overflow/underflow, and a common problem in embedded systems.

Formal methods should take into consideration these characteristics of real-life program variables. Model checking does logical analysis (in our specific use-case; other model checkers can do bit-precise verification as well), so the range of every integer variable is the set of integers ($\mathbb{Z}$), while the range of every floating point variables is the set of real numbers ($\mathbb{R}$). As a result, formal methods may miss some software bugs that are related to the representation of numbers.

Finding these kinds of errors is different from the earlier methods: as it does not aim the untraversed part of the state space, rather the traversed one. The aim is to find errors, that model checking might have missed. To identify these errors, those variables should be found first, whose value might be unrepresentable. These variables are those that store the result of an arithmetic operation (such as adding, multiplying, etc. variables). This information can be extracted from the source code.

```
1. unsigned x; scanf("%u", &x);
2. unsigned y; scanf("%u", &y);
3. unsinged z = x + y;
4. if(z <= 5) {
5.     ...
6. } else {
7.     ...
8. }
```

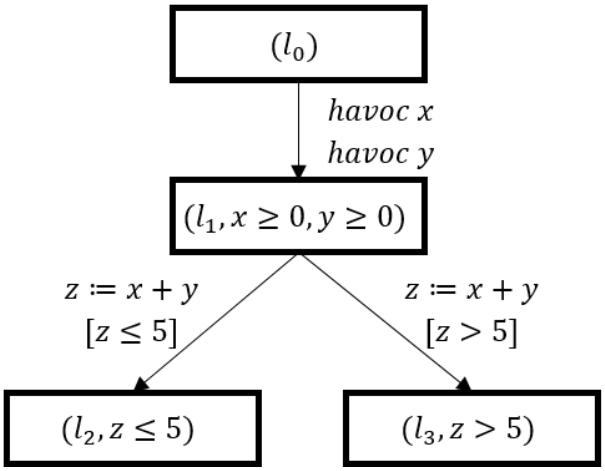

Fig. 3. A C code, and the corresponding ART (fraction)

Numerous SMT problems can be constructed from the operations and guard expressions on the path to the leafs of the ART. However, new constrains must be added, that state the possible range of each input variable. The optimization constraint should be the minimization/maximization of the variables under analysis (whose value might be unrepresentable). If in the solution of the SMT problem the value of the variable is outside the representable range, then an overflow/underflow occurred. Again, using the solution a test case can be generated that reproduces the error.

There are other kinds of number representation errors as well, but they are not discussed here.

V. Case study

On Fig. 3 a simple C program is depicted, that receives two input numbers and behaves differently based on their sum. A part of the source code is depicted below, with a fraction of the abstract reachability tree corresponding to the code. The root of the fraction is l_0 while the incomplete nodes are l_2 and l_3. The three methods described earlier will be presented using the paths from l_0 to l_2 and l_3. Furthermore let us assume that all variables are unsigned integers.

To apply boundary value analysis on the left hand side (from l_0 to l_2), the input variables should be found first. These variables are x and y. Therefore the optimization constraints and the solutions of the SMT problem will be the following:

- $max(x)$: $\{(x = 5), (y = 0)\}$
- $min(x)$: $\{(x = 0), (y = 0)\}^*$
- $max(y)$: $\{(x = 0), (y = 5)\}$
- $min(y)$: $\{(x = 0), (y = 0)\}^*$

To apply robustness testing on the left hand side (from l_0 to l_2), the variables in arithmetic conditions should be found first. The only variable is z, because of the $[z \leq 5]$ condition. Therefore the optimization constraints and the solutions of the SMT problem will be the following:

- $max(z)$: $\{(x = 5), (y = 0)\}^*$
- $min(z)$: $\{(x = 0), (y = 0)\}$

To find errors of number representations on the right hand side (from l_0 to l_3), the variables that can overflow/underflow need to be identified first. The only variable is z, because the it is the only variable that stores the result of an arithmetic operation ($x + y$). Let us assume, that the range of x, y, z is the integers between 0 and 15 (4 bit unsigned integer). The optimization constraints and the solutions of the SMT problem will be the following:

- $max(z)$: $\{(x = 15), (y = 15)\}^*$
- $min(z)$: $\{(x = 3), (y = 3)\}^*$

The first case, when both x and y are 15, the value of z is 30, so an overflow occurred.

VI. Conclusion

Ensuring correctness is a key aspect of the development process in the safety-critical domain. However, it is not a trivial task. Existing approaches, such as model checking and testing both have their advantages and disadvantages: model checking can prove the correctness for the price of heavy computations, while testing can efficiently find bugs in software.

By combining them, it is possible to exploit the advantages of both worlds. If the verification has enough resources to complete the task, then the correctness can be decided. If it is aborted as the resources are not sufficient, the information gathered during the state space traversal can be used to generate test cases focusing on the unverified part of the state space.

The novelty of the presented approach is that by using the information provided by the verification algorithm, the targeted test suite can be generated that results in fewer test cases.

A. Future Work

There is much work left. In the following, we introduce some important directions:

- In the future, further test generating methods (eg. based on equivalence classes) need to be developed to cover a greater part of the state space.
- More CEGAR abstraction methods are needed to be analyzed to extend our method.
- A common pattern in software is input inside a cycle, which often breaks abstraction. Methods need to be devised to provide values for such inputs during test case generation. That is the main weakness of our approach.

*One out of many possibilities

References

[1] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," Communications of the ACM, vol. 54, no. 9, pp. 69-77, 2011.

[2] D. Beyer and S. Löwe, Explicit-State Software Model Checking Based on CEGAR and Interpolation, Lecture Notes in Computer Science, vol. 7793, pp. 146-162, 2013.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith, Counterexample-guided Abstraction Refinement for Symbolic Model Checking, J. ACM, vol. 50, no. 5 pp. 752-794, 2003.

[4] Á. Hajdu, T. Tóth, A. Vörös and I. Majzik, A configurable CEGAR framework with interpolation-based refinements, Lecture Notes in Computer Science, vol. 9688, pp. 158-174, 2016.

[5] S. Graf s H. Saidi, Construction of abstract state graphs with PVS, Lecture Notes in Computer Science, vol. 1254, pp. 72-83, 1997.

[6] D. Beyer and M. Dangl, SMT-based Software Model Checking: An Experimental Comparison of Four Algorithms, Lecture Notes in Computer Science, vol. 9971, 2016.

[7] K. L. McMillan, Applications of Craig interpolants in model checking., Lecture Notes in Computer Science, vol. 3440, pp. 1-12, 2005.

[8] I. S. T. Q. Board, Certified Tester Foundation Level Syllabus, 2018.

[9] N. Tillmann, J. de Halleux and T. Xie, Pex for Fun: Engineering an Automated Testing Tool for Serious Games in Computer Science, 2018.

[10] J. de Halleux and N. Tillmann, Moles: Tool-Assisted Environment Isolation with Closures, Lecture Notes in Computer Science, vol. 6141, pp. 253-270, 2010.

[11] C. Cadar, D. Dunbar and D. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008.

[12] G. LiIndradeep, G. Sreeranga and P. Rajan, KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs, Lecture Notes in Computer Science, vol. 6806, pp. 609-615, 2011.

[13] L. M. d. Moura s N. Bjørner, Z3: An Efficient SMT Solver, TACAS, 2008.