Volume 2, Issue 11, November 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Study of Software Model Checking Techniques

Prof. Rekha Lathi[1]
Department of Computer Engineering
Pillai Institute of Information Technology
Mumbai University, India

Mrs. Rupali Kale[2]
Department of Computer Engineering
Shah & Anchor Kuttchi College of Engineering
Mumbai University, India

Abstract — For improving Software quality the standard method is testing, conventional testing methods often fail to detect faults. Concolic testing attempts to remedy this by automatically generating test cases to explore execution paths in a program under test, helping testers achieve greater coverage of program behaviour in a more automated fashion. Often, programmers spend more time fixing bugs in existing code than they do writing new one: it is estimated that more than half of the software-development costs of a typical project are spent on identifying and fixing defects. Simulation and testing are the prevailing approaches when it comes to validating behaviours of computer systems in general and software code in particular. Testing is essential for quality assurance of database applications. Achieving high code coverage of the database application is important in testing. In practice there may exist a copy of live databases that can be used for database application testing. Concolic testing generates test cases that can achieve high coverage in an automated fashion. Core idea behind concolic testing is to obtain symbolic path formulas from concrete executions and solve them to generate test cases by using constraint solvers.

Keywords— Testing, Concolic testing, conventional testing, bugs, quality assurance.

I. INTRODUCTION

In computer science, model checking refers to the following problem: Given a model of a system, exhaustively and automatically check whether this model meets a given specification. Typically, the systems one has in mind are hardware or software systems, and the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Model checking is a technique for automatically verifying correctness properties of finite-state systems [1].

In order to solve such a problem algorithmically, both the model of the system and the specification are formulated in some precise mathematical language: To this end, it is formulated as a task in logic, namely to check whether a given structure satisfies a given logical formula. The concept is general and applies to all kinds of logics and suitable structures. A simple model-checking problem is verifying whether a given formula in the propositional logic is satisfied by a given structure. Software model checking is the algorithmic analysis of programs to prove properties of their executions. While focus here is on analyzing the behaviour of a program relative to given correctness specifications, the development of specification mechanisms happened in parallel, and merits a different survey. More recently, software model checking has been influenced by three parallel but somewhat distinct developments. First, development of program logics and associated decision procedures provided a framework and basic algorithmic tools to reason about infinite state spaces. Second, automatic model checking techniques for temporal logics provided basic algorithmic tools for state-space exploration. Third, compiler analysis formalized by abstract interpretation, provided connections between the logical world of infinite state spaces and the algorithmic world of finite representations.

Software model checking is recognized as a breakthrough in software verification and resulted, for example, in tools shipped with the Windows Driver Development Kit. The original idea of Model Checking (whose authors obtained the 2007 Turing Award) is to build a model of the system to be verified, and to automatically analyse it in order to check whether all behaviours respect a property typically expressed in temporal logic. As a result, the approach states whether the model satisfies the property and, may be more importantly, exhibits a counter example in case it does not. This counter example can be of tremendous help to the human designer or programmer in order to track the error and to fix it. Applying directly the idea to modern computer systems is intractable due to the huge number, practically infinite, of possible states and behaviours the underlying system might exhibit. This is known as the state explosion problem, and several approaches or techniques like partial order reduction, symbolic representations, bounded model checking, abstraction, interpolation, and SAT and SMT solvers have been developed to combat it. As a result, software model checking is the combination of the original automatic model checking framework.

II. PRINCIPLES OF MODEL CHECKING

Principles of Model Checking offers a comprehensive introduction to model checking that is not only a text suitable for classroom use but also a valuable reference for researchers and practitioners in the field.

A. Model Checking Preliminaries

The formalism commonly used to describe model checking and to define the semantics of temporal logics is the Kripke structure.

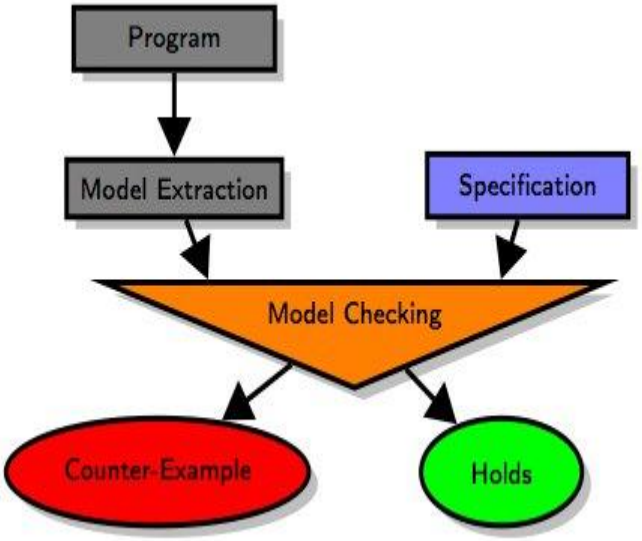

Fig. 1 Block diagram for Model checking technique

Definition 1(Kripke Structure): A Kripke structure K is a tuple K = (S, S0, T, L):

• S is a set of states.

• S0 ⊆ S is an initial state set.

• T ⊆ S×S is a total transition relation, that is, for every s ∈ S there is a s′ ∈ S such that (s, s′) ∈ T.

• L : S → 2AP is a labelling function that maps each state to a set of atomic propositions that hold in this state. AP is a countable set of atomic propositions.

Definition 2(Path): A path p := (s0, s1, ...) of Kripke structure K is an infinite sequence such that ∀ i ⩾0 : (si, si+1) ∈ T for K.

Let Paths (K, s) denote the set of paths of Kripke structure K that start in state s. We use Paths (K) as an abbreviation to denote {Paths (K, s) | s ∈ S0}.

As infinite paths are not usable in practice, model checking uses finite sequences, commonly referred to as traces. If necessary, we can interpret a finite sequence as an infinite sequence where the final state is repeated infinitely.

Definition 3 (Trace) A trace t := (s0, ...sn) of Kripke structure K is a finite sequence such that ∀ 0 ⩽ i < n : (si, si+1) ∈ T for K. There can be a dedicated state si such that si = sn and i ≠ n, which is a loopback state, and [s0, . . . si−1, (si . . . sn) ω] is a path of K. A trace t is either a finite prefix of an infinite path or a path that contains a loop, if a loopback state is given. The latter is called a lasso-shaped sequence, and has the form t := t1(t2)ω, where t1 and t2 are finite sequences. The sequence t2 is repeated infinitely often, denoted with ω, the infinite version of the Kleene star operator used for ω-languages. Temporal logics are modal logics with special operators for time. Time can either be interpreted to be linear or branching. The most common logics are the linear time logic LTL [2] (Linear Temporal Logic), and the branching time logic CTL [3] (Computation Tree Logic). CTL*, introduced by Emerson and Halpern [4], is the superset of these logics. Most current model checkers support either LTL or CTL, or sometimes both. Other temporal logics that are used in model checking are Hennessy-Milner Logic [5] (HML), Modal μ- calculus, and different flavours of CTL such as timed, fair, or action CTL.

An LTL formula consists of atomic propositions, Boolean operators and temporal operators. The operator"O" refers to the next state. E.g.,"O a" expresses that has to be true in the next state. "U" is until operator, where"a U b" means that has to hold from the current state up to a state where b is true. "□" is the always operator, stating that a condition has to hold at all states of a path, and"◊" is the eventually operator that requires a certain condition to eventually hold at some time in the future. The syntax of LTL is given as follows, where AP denotes the set of atomic propositions:

Definition 4(LTL Syntax): The BNF definition of LTL formulas is given below:

ø ::= true | false | a ∈ AP | ¯ ø| ø1 ∧ ø2 | ø1 ∨ ø2 |

ø 1 →ø 2 | ø 1 ≡ø 2 | ø 1 U ø 2 | O ø | □ ø | ◊ ø

The temporal logic CTL was introduced by Clarke and Emerson [3]. It can be viewed as a subset of CTL*, introduced by Emerson and Halpern [4]. CTL* formulas consist of atomic propositions, logical operators, temporal operators (F, G, U, R, X) and path quantifiers (A, E). The operator F ("finally") corresponds to the eventually operator ◊ in LTL, G ("globally") corresponds to.. U ("until") corresponds to U, and R ("release") is the logical dual of U. X ("next") corresponds to the next operator O □. The path quantifiers A ("all") and E ("some") require formulas to hold on all or some paths, respectively. CTL* includes all possible combinations of temporal operators with formulas, where the temporal operators do not have to be preceded by path quantifiers. As CTL* model checking is complex, most model checkers use either CTL or LTL in practice. Consequently, we do not consider CTL* in detail. CTL is the subset of

CTL* obtained by requiring that each temporal operator is immediately preceded by a path quantifier. Consequently, the syntax of CTL can be defined as follows:

Definition 6(CTL Syntax): The BNF definition of CTL formulas is given below:

ø := a ∈ AP | ø1 ∨ ø2 | ø1 ∧ ø2 | ¯ø |
AX ø | AF ø | AG ø | A ø1U ø2 | A ø1R ø2 |
EX ø | EF ø | EG ø | E ø1U ø2 | E ø1R ø2

As all temporal operators are preceded by a path quantifier in CTL, the semantics of CTL can be expressed by satisfaction relations for state formulas. K, s |= Á denotes a state formula Á that is satisfied in state s of Kripke structure K.

Definition 7(CTL Semantics): Satisfaction of CTL formulas by a state s ∈ S of a Kripke Structure K =(S, S0, T, L) is inductively defined as follows, where a ∈ AP:

K, s |= a iff a ∈ L(s) ∧ s ∈ S
K, s |= ¯ø iff ¯ (K, s |= ø)
K, s |= ø1 ∨ ø2 iff (K, s |= ø1) ∨ (K, s |= ø2)
K, s |= ø1 ∧ ø2 iff (K, s |= ø1) ∧ (K, s |= ø2)
K, s |= AX ø iff ∀ ¼ ∈ Paths (K, s) : K, ¼1 |= ø
K, s |= AF ø iff ∀ ¼ ∈ Paths (K, s) : ∃ i : K, ¼i |= ø
K, s |= AG ø iff ∀ ¼ ∈ Paths (K, s) : ∀ i : K, ¼i |= ø

Commonly, three different types of verifiable properties are distinguished:

1. ***Safety Property:*** A safety property describes a behaviour that may not occur on any path ("Something bad may not happen"). To verify a safety property, all execution paths have to be checked exhaustively. Safety properties are of the type □ ¯ø or AG ¯ø, where ø is a propositional formula.

2. ***Invariance Property:*** An invariance property describes a behaviour that is required to hold on all execution paths. It is logically complementary to a safety property. Invariance properties are of the type □ø or AG ø, where ø is a propositional formula.

3. ***Liveness Property:*** A liveness property describes that "something good eventually happens". With linear time logics, this means that a certain state will always be reached. For example, □ø1 →◊ ø2 and AG ø1 → AF ø2 are liveness properties.

The aim of model checking is to determine whether a given model fulfils a given property. Several different algorithms have been successfully used for this task, using different temporal logics and data structures. Once property violation or satisfaction is determined, a model checker can return an example of how this violation or satisfaction occurs. This is illustrated with a counterexample or witness, respectively. Satisfaction of LTL properties is defined using linear sequences. Consequently, witnesses and counterexamples for LTL [2] formulas are also linear sequences. In contrast, CTL [3] properties are state formulas. Therefore, the CTL model checking problem [6] is to find the set of states that satisfy a given formula in a given Kripke structure. Special algorithms are used to derive trace examples for witness or counterexample states [7].

The first successful model checking approach is explicit model checking. There are different approaches based on LTL and CTL properties. In all approaches, the state space is represented explicitly, and searched by forward exploration until a violation of a property is found. For example, in LTL model checking the negation of a property is represented as an automaton that accepts infinite words (B¨uchi automaton). If the synchronous product of model and B¨uchi automaton contains any accepting path, then this path proves property violation (the path shows that the negation of the property is accepted by the model automaton, and therefore the property itself is violated). The counterexample is simply the path back to the initial state. The search algorithm can either be depth- or breadth-first search; recently heuristic search has also been considered. Breadth-first search always finds the shortest possible counterexamples, but the memory demands are significantly higher than for depth-first search. In CTL model checking, all states satisfying a given property are determined by recursively calculating the satisfied sub-formulas for each state. If all states are visited and no violation is detected, then the property is consistent with the model. Directed model checking [8] extends explicit model checking with heuristic search to increase the speed with which errors are found and counterexamples are generated. Such a technique is applicable if the aim of model checking is not a proof of correctness, but the generation of counterexample. As such, this idea is well suited for test case generation.

Symbolic model checking [9], the second generation of model checking, uses ordered binary decision diagrams (BDDs [10]) to represent states and function relations on these states efficiently. This allows the representation of significantly larger state spaces, but a large number of BDD variables has a negative impact on the performance, and the ordering of the BDD variables has a significant impact on the overall size. There are different heuristic approaches of how to order variables, as determining the optimal order is NP-complete [11].

Bounded Model Checking [12], the third generation of model checking, reformulates the model checking problem as a constraint satisfaction problem (CSP). This allows the use of propositional satisfiability (SAT) solvers to calculate counterexamples up to a certain upper bound. As long as the boundary is not too big, this approach is very efficient.

There are also approaches to extend bounded model checking to infinite state systems. Bounded model checking has been successfully applied to systems where traditional model checking fails. At the same time, there are many settings where a bounded model checker fails while a symbolic model checker is efficient. Therefore, bounded model checkers do not replace but supplement traditional model checking techniques. The most commonly used model checkers in the context of testing are the explicit state model checker SPIN (Simple Promela Interpreter), the Symbolic Analysis Laboratory SAL [13], which supports both symbolic and bounded model checking, the symbolic model checker SMV as well as its derivative NuSMV, which supports symbolic and bounded model checking. Other popular model checkers include Mur ø the process algebra based FDR2, or COSPAN; some of these have also been used for testing. Many current model checkers such as NuSMV or SAL [13] support CTL [3] model checking in addition to or instead of LTL model checking. In CTL model checking, special algorithms are applied to construct linear traces from an initial state to explain a violating state. However, only certain restricted subsets of branching time temporal logics such as ACTLdet or LIN always result in linear counterexamples. When using full CTL, linear counterexamples are not always sufficient as evidence for property violation or satisfaction. Most work on testing with model checkers only considers the linear subset when using CTL for properties. Therefore, we use the term counterexample to describe a linear trace that either shows an LTL property violation or violation of a CTL property that can be violated by a linear trace. Recently, an algorithm to create tree-like counterexamples has been proposed by Clarke et al. In a related work, define a formal relation between test cases and counterexamples for full CTL.

III. PROPERTIES

The main goal of software model checking is to prove properties of program computations. Examples of properties are simple assertions, that state that a predicate on program variables holds whenever the computation reaches a particular control location (e.g., "the variable x is positive whenever control reaches"), or global invariants, that state that certain predicates hold on every reachable state (e.g.," each array access is within bounds"), or termination properties (e.g., "the program terminates on all inputs"). Broadly, properties are classified as safety and liveness. Informally, safety properties stipulate that "bad things" do not happen during program computations, and liveness properties stipulate that "good things" do eventually happen.

IV. ISSUES IN TESTING WITH MODEL CHECKERS

Testing with model checkers is an active area of research, and as such there are many issues that still need to be solved. The main showstopper for industry acceptance of model checker based testing is probably the limited performance. A main cause of this problem is the state explosion problem, but there are other issues contributing to a potentially bad performance. Even if the performance is acceptable, the results of the test case generation might not be as good as possible. Some application scenarios, like regression testing, need special treatment. Nondeterministic models or properties that require nonlinear counterexamples are further examples of issues with model checker based testing.

The main cause for performance problems with model checkers is the state explosion, which signifies the large or intractable state spaces that can easily result from complex models. Especially software model checking is susceptible to the state explosion problem. Abstraction is a popular method to overcome the state explosion problem. Abstraction is an active area of research, and many abstraction techniques have been presented in recent years. This has made it possible to verify properties on very large models. In general, abstraction methods are tailored towards verification, and therefore are not always useful in the context of testing. A full survey of available techniques is out of the scope of this document; as an example technique, we mention counterexample guided abstraction refinement (CEGAR), which refines an abstract model until no more spurious counterexamples are generated when verifying a property. This method ensures soundness, which means that a property that holds on the abstract model also holds on the concrete model. In contrast, when generating test cases with a model checker, the objective is different: Properties that are violated by a concrete model should also be violated by the abstract model.

V. CONCLUSION

Research on model checkers is progressing, and the size of models that can be handled constantly increases. There is a need to adapt model checking techniques to faster counterexample creation. Directed model checking is an example of such a technique. At the same time it is not sufficient to blame the performance of model checkers. Even if model checkers could handle models of deliberate size, many of the currently examined testing techniques would result in unfeasibly large test suites. Therefore, research on model abstraction is essential, both for performance and for scalability reasons. There is a lack of documented empirical experience with testing with model checkers. Most work revolves around a set of small, well known example applications. The only available case study that evaluated the scalability showed promising results, but later studies showed that the considered example application has some peculiarities that make it questionable, whether the results are really representative.

Even if all performance problems were resolved, there is still one intrinsic problem to all model based testing approaches: Where does the model come from? In most work on model based testing, the existence of a suitable formal model is assumed. The model creation, however, is one of the most difficult parts of the whole development process. Creating models manually is a complicated task, and two specifies writing a model for an application will probably come up with different models. Different models, however, will most likely result in different test suites.

VI. ACKNOWLEDGEMENT

Thanks to all publishers and researchers to provide so much information about Model checkers and at the same time Model Checking techniques, which will be helpful to develop new model checker testing tool.

REFERENCES

[1] J. Hatcliff, M. B. Dwyer, and H. Zheng, "Slicing software for Model construction," Higher-Order and Symbolic Computation, vol. 13, no. 4, pp. 315–353, December 2000.

[2] Amir Pnueli. The temporal logic of programs. In 18th Annual Symposium on Foundations of Computer Science, 31 October-2 November, Providence, Rhode Island, USA, pages 46–57. IEEE, 1977.

[3] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Logic of Programs, Workshop, pages 52–71, London, UK, 1982.Springer-Verlag. ISBN 3-540-11212-X.

[4] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporallogic of branching time. In STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing.

[5] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. J. ACM, 32(1):137–161, 1985.

[6] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. Model Checking. MIT Press, Cambridge, MA, 1 edition, 2001. 3rd printing.

[7] Edmund M. Clarke, Orna Grumberg, Kenneth L. McMillan, and Xudong Zhao. Efficient generationof counterexamples and witnesses in symbolic model checking, 1995.

[8] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking withHSF-SPIN. In SPIN '01: Proceedings of the 8th international SPIN workshop on Model checkingof software, pages 57–79, New York, NY, USA, 2001.

[9] Kenneth L.McMillan. Symbolic Model Checking. Kluwer Academic Publishers, Norwell, MA, USA, 1993. ISBN 0792393805.

[10] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput., 1986.

[11] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. ACMComput. Surv, 1992.

[12] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, UK, 1999.

[13] Leonardo deMoura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari.SAL 2. In Rajeev Alur and Doron Peled, editors, Computer-Aided Verification, CAV 2004.

