Review of: **Model Checking** by Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled

Vicky Weissman
Department of Computer Science
Cornell University

September 1, 2001

Overview

The goal of model checking is to determine if a given property holds in a particular system. For example, we may want to know that a server never sends sensitive data to the wrong client. Model checking has been used effectively in practice for many finite-state systems, including real-time applications, and for some infinite families of finite systems. Theorem proving and testing are other approaches for system verification. A key advantage of model checking over theorem proving is that it's verification procedure can be fully automated. As for testing, the results from a model checker, as with any formal method, are obviously more conclusive. There are three main components to model checking; describing the system, stating the property, and verifying that the property holds in the system.

The system is typically described by either a finite Kripke structure, $M = (S, R, L)$, or a slightly more general model, $M_{gen} = (S, T, L)$, where S is a finite set of states, R is a binary relation on states, T is a set of binary relations on states, and L is a function from states to atomic propositions. Often, a subset of S will be designated as the start states. For simplicity, we require that there is at least one transition out of each state (i.e. $\forall s \in S \exists s'.(s, s') \in R$) and define a path to be an infinite sequence of states, $s_1, s_2, \ldots$ in which $(s_i, s_i + 1)$ is in R. In symbolic model checking, the state transition graphs are viewed as boolean formulas and represented using ordered binary decision diagrams (OBDD), because there are very efficient algorithms to manipulate OBDDs. A binary decision diagram (BDD)is a graph in which each terminal node is associated with a true or false value and any path from a designated root node to a terminal node corresponds to a set of variable assignments that make the encoded formula have the terminal node's value. An OBDD is a BDD in canonical form; all redundant nodes and edges are removed and there is an ordering on the nodes such that the i^{th} node in any root-to-leaf path is associated with the same variable in the encoded formula.

The most common logics to express system properties are Computational Tree Logic (CTL), Linear Temporal Logic (LTL), CTL* , and propositional μ-calculus. CTL and LTL are sub-logics of CTL* . Any CTL formula can also be expressed in the propositional μ-calculus. Although more complicated than the others, the propositional μ-calculus is particularly interesting, because many temporal and program logics can be encoded into it and a number of efficient model checking algorithms exist for it.

Formulas in CTL* are composed of atomic propositions, propositional logic symbols ($\wedge$, $\vee$, and $\neg$), path quantifiers (**A**and **E**), and temporal operators (**X**, **F**, **G**, **U**, and **R**) as follows:

- Every atomic proposition is a formula.
- If f and g are formulas, then $f \wedge g$, $f \vee g$, $\neg f$, $\mathbf{A}f$, $\mathbf{E}f$, $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\mathbf{U}g$, and $f\mathbf{R}g$ are formulas.

Any CTL* formula can be expressed using only atomic propositions, $\wedge$, $\neg$, **X**, **U**, and **E**. We can define inductively what it means for a formula f to hold in a state s or a path $\pi = s_1, s_2, \ldots$ in a model, M, (written $M, s \models f$ or $M, \pi \models f$) as follows (omitting the standard logical symbols $\wedge$, $\vee$, and $\neg$):

- $M, s \models \mathbf{A}f$ if f holds on every path that begins at state s.
- $M, s \models \mathbf{E}f$ if f holds on at least one path that begins at state s.
- $M, \pi \models \mathbf{X}f$ if $M, s_2 \models f$
- $M, \pi \models \mathbf{F}f$ if f holds in some state in π.
- $M, \pi \models \mathbf{G}f$ if f holds in every state in π.
- $M, \pi \models f\mathbf{U}g$ if g holds in some state in π and, in all previous states, f holds.
- $M, \pi \models f\mathbf{R}g$ if g holds in either every state in π or until, and including the state where, f holds.

A CTL* formula is true in a model if it is true in all the start states (or in every state if no initial states are specified). CTL restricts the set of CTL* formulas to those in which each temporal operator is immediately preceeded by a path quantifier. The CTL semantics are often modified slightly to express properties along 'fair paths', ones in which a state from each designated subset appears infinitely often, rather than all the paths in a system. An LTL formula is a CTL* formula of the form **A**f where f does not contain a path quantifier.

Given a model, $M = (S, T, L)$, and a set of relational variables $Var = \{Q, Q_1, Q_2, \ldots\}$ where each Q_i ranges over sets of states, the μ-calculus formulas are constructed as follows:

- Every atomic proposition is a formula.
- Every relational variable is a formula.
- If f is a formula and a is a binary relation in T, then $[a]f$ and $\langle a \rangle f$ are formulas.
- If f is a formula and Q is a variable then $\mu Q.f$ and $\nu Q.f$ are formulas, provided that all occurrences of Q in f fall under an even number of negations.
- If f and g are formulas, then $\neg f$ and $f \wedge g$ are also formulas.

Given an environment e that maps random variables to sets of states, we can define inductively (omitting standard symbols $\neg$ and $\wedge$) what it means for a formula f to hold in a state s in M (written $M, s, e \models f$):

- $M, s, e \models p$ if p is in $L(s)$.
- $M, s, e \models Q$ if s is in $e(Q)$.
- $M, s, e \models [a]f$ if for every state s' such that $(s, s') \in T(a)$, $M, s', e \models f$.
- $M, s, e \models \langle a \rangle f$ if there is a state s' such that $(s, s') \in T(a)$ and $M, s', e \models f$.
- $M, s, e \models \mu Q.f (\nu Q.f)$ if the least (greatest) fixed point holds.

The naive model checking algorithms for the languages are fairly straight-forward. The key to CTL formulas is to evaluate which states satisfy the smallest sub-formulas first, then use these results to find the states that satisfy the next largest and then the next until the entire formula has been evaluated. A CTL* checker basically uses the same approach, except that some of the sub-formulas are LTL and must be evaluated using LTL checking techniques. The interesting part of the μ-calculus checker is the evaluation of fixed points. It turns out that, because S is finite and any μ-calculus formula is monotonic, the fixed points can be found through a simple, iterative process. Specifically, the least fixed point, $\mu Q.f$, can be determined through a series of approximations, $Q_1, Q_2, \ldots$, in which Q_1 is the empty set, Q_{i+1} is the evaluation of f when $Q = Q_i$, and the least fixed point is the first approximation that equals its predecessor. The greatest fix point can be found through the same procedure with $Q_1 = S$. The LTL checkers take a different approach, either by using automata theory or a tableau construction. For brevity, I'll only discuss the tableau strategy here.

For an LTL formula $\mathbf{A}f$ that is logically equivalent to a formula $\neg\mathbf{E}\neg f = \neg\mathbf{E}g$, the tableau approach constructs a graph to determine if $\mathbf{E}g$ is satisfied and then uses this information to decide if $\mathbf{A}f$ holds. More specifically, the checker first constructs the closure of g, $cl(g)$, which is the set of formulas whose truth value effects the truth of g. Then, the checker associates with each state, s, a set k_s that contains $L(s)$ and the maximal consistent set of formulas in $cl(g)$ that is also consistent with $L(s)$. A new graph, G, is constructed where the nodes are the (s, k_s) pairs and the edges correspond to the transitions (s, s') in R such that if a formula $\mathbf{X}h$ is in k_s then h is in $k_{s'}$. Finally, a state s satisfies $\mathbf{E}g$ iff g is in k_s and there exists a path in G from (s, k_s) to a self-fulfilling strongly connected component (a single node with a self-loop or a set of nodes where every node is reachable from every other node and if any node has a k that contains a formula $h_1 U h_2$ then some node in the component has a k that contains h_2).

The efficiency of the basic model checking algorithms can be improved in a number of ways. As previously stated, the computation time can be decreased by changing the algorithms to use OBDDs. One can also try to find an equivalent, smaller model. This can be done by exploiting any symmetry in the system and removing data that is not relevant to the checked property. It may also be possible to replace actual system data by abstract values and collapse the model accordingly. For example, a model of a traffic light may make a distinction between a yellow and red light, where the property may not need to differentiate between the two 'stop' signals. Often concurrent systems are represented by a model in which every possible interleaving is considered. If the property does not distinguish between some or all of the different orderings, then only one 'representative' ordering must be checked. In addition to finding a smaller model, a natural strategy is to split the model into independent parts. For example, we may be able to verify that a server is operating correctly without considering the network or the client. When interdependencies between sections of the model exist, we may make assumptions about one component, that we later verify, to check a property of another.

Contents

Chapter 1, **Introduction**, is a general overview of what model checking is, how it compares to other verification techniques, and how it can be optimized.

Chapter 2, **Modeling Systems**, explains how systems can be represented by formulas or Kripke structures and reminds us that a poor match between the model and the real system can lead to incorrect conclusions.

Chapter 3, **Temporal Logics**, defines the CTL, LTL, and CTL* logics, as well as the modifications to CTL for handling fair paths.

Chapter 4, **Model Checking**, presents basic algorithms for checking formulas in CTL, CTL with fairness, LTL, and CTL* .

Chapter 5, **Binary Decision Diagram**, defines BDDs, gives an algorithm for obtaining an OBDD from a BDD, and explains how Kripke structures can be expressed using OBDDs. The size of an OBDD can depend critically on the variable ordering used to construct it. Although some heuristics are mentioned (e.g. try to keep variables in the same subcircuit close together), simply checking that an ordering is optimal is NP-complete.

Chapter 6, **Symbolic Model Checking**, gives symbolic model checking algorithms (ones based on the manipulation of boolean formulas/OBDDs) for CTL, CTL with fair paths, and LTL. The CTL algorithm very roughly corresponds to converting the CTL formula to μ-calculus and doing the basic evaluation technique for that logic (as given in the **Overview** section). The LTL approach presented here still constructs a tableau, but fewer formulas are used. For example, the entire closure is not taken. An algorithm for finding witnesses and counterexamples (a path that proves a property is or is not satisfied) is presented. In addition, strategies are discussed for improving computational efficiency by handling the OBDDs 'in chunks', rather than as single, large structures.

Chapter 7, **Model Checking for the μ-calculus**, defines the propositional μ-calculus and presents a model checking algorithm for it.

Chapter 8, **Model Checking in Practice**, gives a brief overview of the Symbolic Model Verifier (SMV) tool. It then discusses how SMV was used to uncover errors and ambiguities in the cache coherency protocol for the IEEE Futerbus+ standard.

Chapter 9, **Model Checking and Automata Theory**, explains how both the model and the negation of an LTL formula can be represented as generalized Buchi automata (finite automata over infinite words), how two generalized Buchi automata (gba) can be combined to create a third that accepts their intersection, and how to determine if a gba does not accept any input. Putting these pieces together, an LTL formula holds in a model if taking the intersection of the model and the negation of the formula produces a gba that accepts the empty set. By using the property gba to guide the construction of the model gba, a counter-example (showing that the intersection is non-empty) can often be found before the entire model is constructed.

Chapter 10, **Partial Order Reduction**, shows that an LTL formula can be expressed without using the next time operator, **X**, iff the property cannot distinguish between two stuttering equivalent models (ones that only differ in their sequences of identically labeled states). Using this fact, the chapter presents algorithms that reduce the time needed for verification, by reducing the size of the model.

Chapter 11, **Equivalences and Preorders between Structures**, gives algorithms for determining if two structures are bisimulation equivalent (same CTL* formulas hold in both) or if one is a simulation (abstraction/generalization) of the other. If a model M' is a simulation of a model M, then any ACTL* formula (CTL* restricted to universal quantification) that holds in M', holds in M. This fact is used as the foundation for an ACTL* model checking algorithm which is in the same spirit as the LTL checker given in this review and refined in Chapter 6.

Chapter 12, **Compositional Reasoning**, shows how to decompose a model into inter-dependent parts. Each component is checked separately, based on assumptions about the rest of the structure. Once these assumptions are verified, we can infer if the property holds in the entire model without ever having to construct the (large) global, state-transition graph.

Chapter 13, **Abstraction**, presents two approaches to reducing the model's size. The first determines which variables effect the truth of the property and removes the others from the model. The second finds a mapping from the actual data to a smaller set of abstract values (e.g. map x = 7 and y = 9 to x = y = odd).

Chapter 14, **Symmetry**, uses group theory to derive an equivalence relation on the model's state space. This relation is used to construct a bisimulation equivalent model, called a quotient model, that is usually smaller, and hence easier to check, than the original.

Chapter 15, **Infinite Families of Finite-State Systems**, looks at the problem of verifying a property for an infinite set of finite systems. Although the problem is, in general, undecidable, a number of techniques have been developed for particular families. Most of these rely on finding an invariant such as a model that is bisimulation equivalent to all the models in the family.

Chapter 16, **Discrete Real-Time and Quantitative Temporal Analysis**, presents an extension of CTL, called RTCL, which associates the operators **EU**and **EG**with time intervals. Also, naive algorithms are given for determining the minimum and maximum delays in a system. The chapter concludes with a discussion on how these algorithms and the model checking techniques discussed thus far were used to verify the correctness, both functional and timing, of an aircraft controller.

Chapter 17, **Continuous Real Time**, shows how continuous real time systems can be represented as timed automaton and references a number of papers that propose model checking algorithms for these structures. (A timed automaton is a finite automaton augmented with a set of real-valued clocks. Each state has a set of clock constraints that must be met for the automaton to be in that state. Time can elapse 'in a state', but transitions occur instantaneously. Finally, some, or all, of the clocks can be reset when a transition occurs.)

Chapter 18, **Conclusion**, gives several directions for future research including making model checking easier for engineers to use, finding more concise representations for the models and properties, exploring probabilistic verification, determining performance measurements rather than only checking if a property holds, and combining model checking techniques with theorem proving to get the best of both worlds.

Opinion

The book was designed to be both an introductory text and a reference for researchers. To accommodate the novice, no previous knowledge about model checking is assumed. In fact, foundational ideas such as automaton and group theory are briefly reviewed. If the reader is unfamiliar with an area, however, the review will not be enough to fully understand the relevant chapter(s). For the theoretician, the book contains most of the theorems and proofs needed to convince him/her that the various approaches are correct. The systems researcher and the practitioner are not ignored either. There is a significant amount of discussion on the protocols used in model-checkers and the results of applying these to practical, verification problems. When a proof is omitted or the details of a topic are not fully discussed, the book recommends additional reading, namely the original research papers. The reference section, which contains 253 entries, makes this book an excellent 'first-stop' for people who want a thorough understanding of a specific aspect of model checking.

I have only two recommendations for future editions. First, although examples are worked-out in the text, adding exercises at the end of the chapters would help students test their understanding of the presented techniques. Second, the 'flow' of the book is not smooth. Specifically, some of the chapters cover much more information in much more depth than others. An attempt is given to have every chapter begin with an overview, but for some chapters this is a paragraph and for others it is several pages including definitions and proofs. Also, the book continually jumps from one logic to another and back again, rather than discussing all the techniques for one logic and then switching to another.

I strongly recommend this book to researchers looking to begin or further their understanding of model checking.