Chapter 1
Introduction to Model Checking

Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith

Abstract Model checking is a computer-assisted method for the analysis of dynamical systems that can be modeled by state-transition systems. Drawing from research traditions in mathematical logic, programming languages, hardware design, and theoretical computer science, model checking is now widely used for the verification of hardware and software in industry. This chapter is an introduction and short survey of model checking. The chapter aims to motivate and link the individual chapters of the handbook, and to provide context for readers who are not familiar with model checking.

1.1 The Case for Computer-Aided Verification

> *The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. [32]*

In the ideal world of Dijkstra's Turing Award Lecture 1972, programs are intellectually manageable, and every program grows hand in hand with a mathematical proof of the program's correctness. The history of computer science has proven Dijkstra's vision limited. Manual proofs, if at all, can be found only in students' exercises, research papers on algorithms, and certain critical application areas. Although the work of McCarthy, Floyd, Hoare, and other pioneers [7, 36, 45, 62, 66] provided us with formal proof systems for program correctness, little use is made of these techniques in practice. The main challenge is scalability: real-world software systems not only include complex control and data structures, but depend on much "context" such as libraries and interfaces to other code, including lower-level systems code. As

E.M. Clarke
Carnegie Mellon University, Pittsburgh, PA, USA

T.A. Henzinger (✉)
IST Austria, Klosterneuburg, Austria
e-mail: tah@ist.ac.at

H. Veith
Technische Universität Wien, Vienna, Austria

© Springer International Publishing AG, part of Springer Nature 2018

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_1

a result, proving a software system correct requires much more effort, knowledge, training, and ingenuity than writing the software in trial-and-error style. This asymmetry between coding and verification is the main motivation for computer-aided verification, i.e., the use of computers for the verification of software and hardware.

From a 1972 perspective, the computer industry has changed the world beyond recognition. Computer programs today have millions of lines of code, they are written and maintained by globally distributed teams over decades, and they are used in diverse and complex computing environments from micro-code to cloud computing. Computer science has become pervasive in production, transportation, infrastructure, health care, science, finance, administration, defense, and entertainment. Programs are the most complex machines built by humans, and have huge responsibilities for human safety, security, health, and well-being. These developments have exacerbated the challenges and, at the same time, dramatically increased the need for correct programs and, hence, for computer-aided verification.

Starting with the work of Turing, the perspectives for automated verification did not look promising. Turing's halting problem [82] and Rice's Theorem [79] tell us that computer-aided verification is, in general, an unsolvable problem. At face value, these theorems demonstrate the undecidability of verification even for simple properties of simple programs. Technically, all that is needed for undecidability are two integer variables that, embedded into a looping control structure, can be incremented, decremented, and checked for zero. If the values of integer variables are bounded, we obtain a system with finitely many different states, and verification becomes decidable. However, complexity theory tells us that even for finite-state systems, many verification questions require a prohibitive effort.

Yet, at a time when logic in computer science was a synonym for undecidability and intractability, the invention of model checking marked a paradigm shift towards the practical use of logic for bug finding—i.e., falsification rather than verification—in the hardware and software industries. Not untypical of paradigms acquired many decades ago, the case for model checking appears simple and convincing in retrospect [22, 23, 72, 75]. In its basic classical form, the paradigm consists of the following insights:

Modeling. Finite state-transition graphs provide an adequate formalism for the description of finite-state systems such as hardware, but also for finite-state abstractions of software and of communication protocols.

Specification. Temporal logics provide a natural framework for the description of correctness properties for state-transition systems.

Algorithms. There are decision procedures for determining whether a finite state-transition structure is a model of a temporal-logic formula. Moreover, the decision procedures can produce diagnostic counterexamples when the formula is not true in the structure.

Taken together, these insights motivate the methodology that is shown in Fig. 1: the system under investigation is compiled into a state-transition graph (a.k.a. Kripke structure) K, the specification is expressed as a temporal-logic formula φ, and a decision procedure—the model checker—decides whether $K \models \varphi$, i.e., whether the

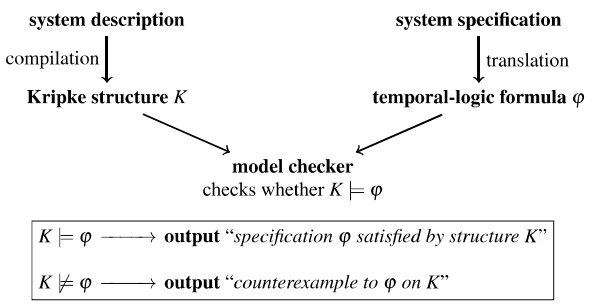

Fig. 1 Basic model-checking methodology

structure K is a model of the formula φ. If $K \not\models \varphi$, then the model checker outputs a counterexample that witnesses the violation of φ by K. The generation of counterexamples means that, in practice, falsification (the detection of bugs) can often be faster than verification (the proof of their absence).

There is of course a mismatch between the early model checkers, which were essentially graph algorithms on Kripke structures, and the complexity of modern computer systems sketched above. Research in model checking spanning more than three decades has helped to close this gap in many areas that are documented throughout this handbook. We can roughly classify the advances in model checking and the chapters of this handbook in terms of two recurrent themes that have driven much of the research agenda in model checking:

1 The algorithmic challenge: *Design model-checking algorithms that scale to real-life problems.* The main practical problem in model checking is the combinatorial explosion of states in the Kripke structure—the "state-explosion problem." Since each state represents the global system status at a given time point, a state is essentially a memory snapshot of the system under investigation, and, thus, the size of the state space is exponential in the size of the memory. Therefore, even for systems of relatively modest size, it is impossible to compute and analyze the entire corresponding Kripke structure directly. In fact, in most situations, the state space is not finite (e.g., unbounded memory, unbounded number of parallel processes), which leads to the modeling challenge. In practice, very large and infinite state-transition systems are approximated by effective abstractions such as finite-state abstractions, or decision procedures are approximated by so-called "semi-algorithms," which are aimed at finding bugs but may fail to prove correctness, or both.

2 The modeling challenge: *Extend the model-checking framework beyond Kripke structures and temporal logic.* Kripke structures are natural representations for various flavors of communicating finite-state machines. To model and specify unbounded iteration and recursion, unbounded concurrency and distribution, process creation and reconfiguration, unbounded data types, real-time and cyber-physical systems, probabilistic computation, security aspects, etc., and to abstract these features effectively, we need to extend the modeling and specification frameworks beyond Kripke structures and classical temporal logics. Some

extensions maintain decidability, often through the construction of finite-state abstractions, which ties the modeling challenge back to the algorithmic challenge. Other extensions sacrifice decidability, but maintain the ability of model checking to find bugs automatically, systematically, and early in the system design process.

As the two challenges are tied together closely, many chapters of this handbook address both. New models without algorithms and without experimental validation are not central to model-checking research.

We believe that the main strengths of model checking are threefold. First, model checking is a systematic, algorithmic methodology which can be computerized and, ideally, fully automated. Thus, model-checking tools have the goal and promise to be used during the design process by hardware and software engineers without the assistance of verification experts. Second, model checking can be applied at different stages of the design process, on abstract models as well as on implemented code. While any one model-checking tool is usually limited to particular modeling and specification languages, the methodology as such is not restricted to any level or formalism and can be applied at different levels, to incomplete systems and partial specifications, to find different kinds of bugs. Third, and perhaps most importantly, model checking deals particularly well with concurrency. The interaction between parallel processes is one of the main sources of complexity and errors in system design. Moreover, concurrency errors are especially subtle, contingent, and therefore difficult to reproduce; they are hard to find by testing the system, and their absence is hard to prove by logical arguments or program analyses because of the extremely large number of possible interactions between parallel processes. Model checking, on the other hand, which is based on the algorithmic exploration of large state spaces, is particularly well suited for finding concurrency bugs and proving their absence.

These strengths distinguish model checking from related approaches for improving system quality:

Testing is the fastest and simplest way to detect errors. It lends itself to easy automation and can often be handled with limited academic background. Testing covers a large spectrum from manual, ad hoc testing of code all the way to automated efforts and model-based testing [37, 48, 50]. Fundamentally dynamic, testing is able to detect compiler errors, hardware errors, and modeling errors that are invisible to static tools; it is integral to any comprehensive approach to safety engineering and software certification [60]. However, *"program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence"* [32], especially for concurrent systems. Model checking, on the other hand, provides a systematic approach to bug detection that can be applied to incomplete systems and, even for concurrent systems, offers in the limit—when no more errors are found—a certificate of correctness. In other words, while testing is only debugging, model checking is systematic debugging aimed at model verification. Chapter 19 will discuss the cross-fertilization of modern model-checking and white-box testing techniques.

Abstract interpretation and other program analyses are similar to model checking in that they employ algorithms to prove program properties automatically [29, 68]. Following the tradition of programming-language semantics rather than logic, static analysis frameworks such as data-flow analysis, abstract interpretation, and rich type systems are built on lattice theory. Historically, their applications have focused mainly on fast analyses for compilers, in particular on overapproximative analyses of numerical data types and assertion violations [6]. Thus, in comparison to model checking, the focus of program analyses is on efficiency rather than expressiveness, and on code rather than models. In recent research on software verification, abstract interpretation and model checking have converged to a large extent. Chapters 15 and 16 will explore the rich relationship between abstract interpretation and model checking in depth.

Higher-order theorem proving is a powerful and proven approach for the verification of complex systems. Semi-manual theorem provers [13, 69, 85] have been used to verify critical systems ranging from floating-point arithmetic [1] to microkernels [51], and even to the proof of deep mathematics such as the Kepler conjecture [40]. In comparison to model checking, the focus of higher-order theorem proving is on expressiveness rather than efficiency: it handles data manipulation precisely and typically aims at full functional correctness. Theorem proving naturally incorporates the manual decomposition of a verification problem, which is obligatory for complex systems. But even for experts, developing proofs in higher-order logic tools is a time-consuming and often non-trivial effort worthy of a research publication. Chapter 20 will explore some connections and combinations of model checking and theorem proving.

While the different verification methods have different historical starting points and different communities, ultimately they simply represent different trade-offs on the efficiency versus expressiveness (or precision) spectrum: greater expressive power tends to take us, at the cost of efficiency, from testing to abstract interpretation to model checking to theorem proving, and, over time, the differences become smaller. Through more than three decades, model checking has acquired and adapted methods from all of these research areas, but also from automata theory, process algebra, graph algorithms, game theory, hardware simulation, stochastic processes, control theory, and many other areas. At the same time, model checking has interacted with target areas for verification, most importantly computer engineering and VLSI design, software engineering and programming languages, embedded and cyber-physical systems, artificial intelligence, and even computational biology. It is thus fair to say that model checking is characterized less by purity of method than by the goal of debugging and analyzing dynamical systems that exist in the real world and can be modeled as state-transition systems.

Model checking has been covered by several monographs [8, 11, 12, 26, 46, 47, 54, 55, 67, 71] and surveys [25, 28, 33]. The early history of model checking is documented in several collections of essays [38, 63]. The rest of this chapter serves as an introduction to the handbook for readers with little familiarity with the material. In Sect. 1.2 we give a minimalist introduction to the classical setting of temporal-logic model checking over Kripke structures. Section 1.3 then revisits

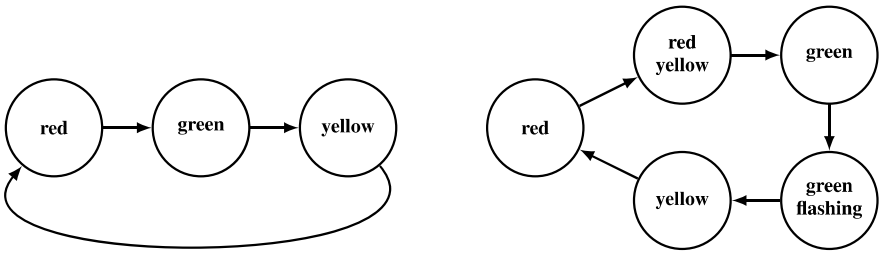

Fig. 2 American and Austrian traffic lights as Kripke structures

some of the fundamental challenges for model checking and puts the individual chapters of the handbook in perspective. In Sect. 1.4 we provide a brief outlook for the field.

1.2 Temporal-Logic Model Checking in a Nutshell

We give a brief introduction to the classical view of temporal-logic model checking.

1.2.1 Kripke Structures

Kripke structures [53] are finite directed graphs whose vertices are labeled with sets of atomic propositions. The vertices and edges of the graph are called "states" and "transitions," respectively. In our context, they are used to represent the possible configurations and configuration changes of a discrete dynamical system. For a simple illustration, consider the two Kripke structures in Fig. 2, which represent the states and state transitions of traffic lights in the USA and Austria, respectively. Formally, a *Kripke structure* over a set A of atomic propositions is a triple $K = \langle S, R, L \rangle$ where S is a finite set of states (the "state space"), $R \subseteq S \times S$ is a set of transitions (the "transition relation"), and the labeling function $L\colon S \to 2^A$ associates each state with a set of atomic propositions. For a state $s \in S$, the set $L(s)$ represents the set of atomic propositions that are true when the system is in state s, and the set $A \setminus L(s)$ contains the propositions that are false in state s. We assume that the transition relation R is total, i.e., that all states have non-zero outdegree.

The dynamic behavior of the system represented by a Kripke structure corresponds to a path through the graph. A path is a finite or infinite sequence $\pi = s_0, s_1, s_2, \ldots$ of states such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. The totality of the transition relation ensures that every finite path can be extended to an infinite path. Given an infinite path π, we write $L(\pi) = L(s_0), L(s_1), L(s_2), \ldots$ for the corresponding infinite sequence of sets of atomic propositions. Moreover, we write $\pi^i = s_i, s_{i+1}, s_{i+2}, \ldots$ for the infinite path that results from π by removing the first i states.

Related models that are based on discrete state changes are sometimes called automata, state machines, state diagrams, labeled transition systems, etc. Throughout

this book, such models are used in accordance with the traditions of the respective research areas. In most cases, algorithmic results can be easily transferred between these models. Later handbook chapters will introduce more advanced frameworks—e.g., for modeling recursive (Chap. 17), probabilistic (Chap. 28), and real-time (Chap. 29) behavior—which extend finite state-transition systems in fundamental ways and require more intricate analysis techniques.

1.2.2 The Temporal Logic CTL*

CTL* [34] is a propositional modal logic with path quantifiers, which are interpreted over states, and temporal operators, which are interpreted over paths.

Path quantifiers:

A "for every infinite path from this state"
E "there exists an infinite path from this state"

Temporal operators (for atomic propositions p and q):
$\mathbf{X}p$ "p holds at the next state"
$\mathbf{F}p$ "p holds at some state in the future"
$\mathbf{G}q$ "q holds at all states in the future"
$q\mathbf{U}p$ "p holds at some state in the future, and q holds at all states until p holds"

For instance, $\mathbf{F}p$ holds on path π iff π contains a state with label p, and $\mathbf{A}\varphi$ holds at state s iff φ holds on all infinite paths that start from state s.

Given a set of atomic propositions A, the syntax of CTL* is defined recursively as follows:

- If $p \in A$, then p is a formula of CTL*.
- If φ and ψ are formulas of CTL*, then $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$, $\mathbf{A}\varphi$, $\mathbf{E}\varphi$, $\mathbf{X}\varphi$, $\mathbf{F}\varphi$, $\mathbf{G}\psi$, and $\psi\mathbf{U}\varphi$ are formulas of CTL*.

To reflect the distinction between path quantifiers and temporal operators, we distinguish a syntactic subset of CTL* called state formulas. *State formulas* are boolean combinations of atomic propositions and CTL* formulas whose outermost operator is a path quantifier, i.e., they start with **A** or **E**. As in the example of $\mathbf{A}\varphi$ above, the truth value of a state formula can be asserted over a state in a Kripke structure. For all other formulas of CTL*, we need a path to determine the truth value. The formal semantics of CTL* is based on this syntactic distinction, and presented in Table 1.

An easy exercise on the semantics defined in Table 1 shows that the syntactic restriction of CTL* to one of the path quantifiers and the two temporal operators **X** and **U** yields the full expressive power of CTL*, i.e., all other operators can be defined from these three operators.

Given a Kripke structure K, state s, and state formula f, a model-checking algorithm is a decision procedure for $K, s \models f$. In case of $K, s \not\models f$, many model-checking algorithms provide evidence of the violation of the satisfaction relation,

Table 1 Semantics of CTL*. Here, K is a Kripke structure, π is a path, s is a state, p is an atomic proposition, f and g are state formulas, and φ and ψ are CTL* formulas

$K, s \models p$	iff	$p \in L(s)$
$K, s \models \neg f$	iff	$K, s \not\models f$
$K, s \models f \vee g$	iff	$K, s \models f$ or $K, s \models g$
$K, s \models f \wedge g$	iff	$K, s \models f$ and $K, s \models g$
$K, s \models \mathbf{E}\varphi$	iff	there is an infinite path π starting from s such that $K, \pi \models \varphi$
$K, s \models \mathbf{A}\varphi$	iff	for every infinite path π starting from s we have $K, \pi \models \varphi$
$K, \pi \models f$	iff	$K, s \models f$ for the first state s of π
$K, \pi \models \neg\varphi$	iff	$K, \pi \not\models \varphi$
$K, \pi \models \varphi \vee \psi$	iff	$K, \pi \models \varphi$ or $K, \pi \models \psi$
$K, \pi \models \varphi \wedge \psi$	iff	$K, \pi \models \varphi$ and $K, \pi \models \psi$
$K, \pi \models \mathbf{X}\varphi$	iff	$K, \pi^1 \models \varphi$
$K, \pi \models \mathbf{F}\varphi$	iff	there exists an $i \geq 0$ such that $K, \pi^i \models \varphi$
$K, \pi \models \mathbf{G}\psi$	iff	for all $j \geq 0$ we have $K, \pi^j \models \psi$
$K, \pi \models \psi \mathbf{U} \varphi$	iff	there exists an $i \geq 0$ such that $K, \pi^i \models \varphi$ and for all $0 \leq j < i$ we have $K, \pi^j \models \psi$

when possible in the form of a counterexample. If f has the form $\mathbf{A}\varphi$, then a finite or infinite path of K violating φ serves as a counterexample.

Table 2 gives common examples of CTL* formulas. For each formula, the table describes the semantics of the formula and the structure of possible counterexamples, which are illustrated in Fig. 3. Note that a counterexample is a witness for the negated formula. For instance, a counterexample for $\mathbf{AG}p$ is a witness for the satisfaction of $\neg\mathbf{AG}p$, and, thus, for $\mathbf{EF}\neg p$. As line 3 in Table 2 illustrates, a model checker may not be able to give practical counterexamples for formulas with unnegated $\mathbf{E}$ quantifiers. The situation is better for ACTL*, the fragment of CTL* where $\mathbf{E}$ does not occur and negation is restricted to atomic propositions: ACTL* has tree-like counterexamples [27] and plays an important role in abstraction; cf. Chap. 13.

When the specification is satisfied by the structure, the situation is dual: for formulas involving only unnegated $\mathbf{E}$, a model checker can output a witness for the satisfaction of the formula, but for most formulas with unnegated $\mathbf{A}$ this is unrealistic. In the latter situation, vacuity detection can be used as a "sanity check," to check whether the positive verification result may be based on a faulty specification [10]. A classical example of vacuity is antecedent failure, where a specification $\mathbf{A}(\mathbf{G}p \rightarrow \mathbf{F}q)$ holds because $\mathbf{AG}\neg p$ is true.

The example formulas in Table 2 belong to the temporal logics CTL or LTL, which are useful syntactic fragments of CTL*. Intuitively, CTL is a logic based on state formulas, and LTL is a logic avoiding state formulas. While CTL can be model checked particularly efficiently, LTL allows the natural specification of properties of dynamic behaviors, which correspond to paths.

Table 2 Examples of CTL* formulas and respective counterexamples

	Sub-logic	Formula	Intuition	Counterexample
1	CTL, LTL	$\mathbf{AG}p$	p is an invariant	finite path leading to $\neg p$
2	CTL, LTL	$\mathbf{AF}p$	p must eventually hold	infinite path (lasso-shaped) without p
3	CTL, (negated) LTL	$\mathbf{EF}\neg p$	$\neg p$ is reachable	substructure with all reachable states, all containing p
4	CTL, LTL	$\mathbf{AG}(p \vee \mathbf{X}p) = \mathbf{AG}(p \vee \mathbf{AX}p)$	p holds at least every other state	finite path leading to $\neg p$ twice in a row
5	CTL, LTL	$\mathbf{AGF}p = \mathbf{AGAF}p$	p holds infinitely often	infinite path (lasso) on which p occurs only finitely often
6	CTL, LTL	$\mathbf{AG}(p \rightarrow \mathbf{F}q) = \mathbf{AG}(p \rightarrow \mathbf{AF}q)$	every p is eventually followed by q	finite path leading to p, but no q now nor on the infinite path (lasso) afterwards
7	CTL, (boolean combination of) LTL	$(\mathbf{AGF}p) \wedge \mathbf{EF}\neg p$	both 3 and 5 hold	either counterexample for $\mathbf{AGF}p$ or for $\mathbf{EF}\neg p$
8	CTL only	$\mathbf{AGEX}p$	reachability of p in one step is an invariant	finite path leading to a state whose successors all have $\neg p$
9	CTL only	$\mathbf{AG}(p \vee \mathbf{AXAG}q \vee \mathbf{AXAG}\neg q)$	once p does not hold, either q or $\neg q$ become invariant in one step	finite path leading to $\neg p$ from which two finite extensions reach q and $\neg q$
10	LTL only	$\mathbf{AFG}p$	p must eventually become an invariant	infinite path (lasso) on which $\neg p$ occurs infinitely often
11	LTL only	$\mathbf{A}(\mathbf{GF}p \rightarrow \mathbf{GF}q)$	if p holds infinitely often, so does q	infinite path (lasso) on which p occurs infinitely often, but q does not