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

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