An Introduction to Model Checking

Pierre Wolper
University of Liège*

March 29, 1995

1 Introduction

Verifying a program is proving, in a formal mathematical way, that the program satisfies a specification written in a logical language. Though verification has often been claimed to be a promising approach for ensuring the correctness of software, almost 30 years of research, starting with the work of Floyd [Flo67] and Hoare [Hoa69], have failed to deliver a widely usable technology. There are at least two reasons for this.

First, one should remember that verification is comparing two descriptions of what the program should do: the program itself and a logical specification, which is in principle shorter and more abstract. For a complex system, the logical specification can nevertheless be very large and difficult to obtain. Its correctness is thus often nonobvious, which renders hypothetical the certainty that verification appears to offer. Second, the verification process requires very long proofs. These can require far more time than writing the program itself, and are very prone to error. Indeed, they are too specific and boring to be checked reliably by humans, and theorem-proving technology requires too much expertise and is not yet sufficiently powerful to be widely usable.

Model checking [CES86] is often thought of as a program verification approach, but differs from traditional program verification in several crucial aspects. First, it does not aim at being fully general. It was initially restricted to finite-state systems and, though this limitation is no longer absolute, model checking is still only applicable to systems whose states have short and easily manipulated descriptions. Typical systems in this category are those whose intricacy resides more in the control than in the data, for instance hardware, concurrent protocols, process control systems, and more generally what are referred to as *reactive* systems [Pnu85]. These are systems whose role is more readily described by their possible interaction sequences with their environment than by the transformation they apply to complex data.

*Address: Institut Montefiore, B28; Université de Liège; 4000 liège Sart-Tilman; Belgium. Email: pw@montefiore.ulg.ac.be

A second aspect on which model checking diverges from traditional verification is that it is fully algorithmic and of low computational complexity. From the start, model-checking has been conceived as an automatic machine-implemented approach. As such, it requires no intervention of the software developer and, furthermore, provides explicit information about what could be wrong when it fails to check a property. Finally, since model checking can be used easily and repeatedly, it can be applied to a variety of properties and provides useful information even if one does not have an exhaustive specification of the system being developed. One could view model checking as an elaborate debugging tool that, rather than observing a few possible executions of the system, checks reliably that all possible executions satisfy a given property.

2 The Principle

In one sentence, model checking is a nontraditional use of a nontraditional logic. Logic is usually thought of as a formal framework for deductive reasoning. One starts with formulas describing the domain of interest (an axiomatization of the domain) and tries to derive the formula to be proved with the help of precisely defined deduction rules. For program verification, this means that one needs to translate (or incorporate) the program into formulas, to axiomatize the objects manipulated by the program (integers, strings, ...), and to go on about doing the proof.

Model checking is based on a much simpler problem in formal logic. This problem is to check that a given formula is satisfied on a specific finite domain, i.e. one *checks* that the finite domain is a *model* of the formula. For instance, rather than being asked to axiomatize groups and to prove that some property is true of all groups, one is asked to check that a property is true of a specific finite group. This is easy to do because, since the specific group is finite, the formula can simply be evaluated on this group. As another example, in the case of Boolean logic, model checking amounts to evaluating a formula after substituting specific *true/false* values for the variables.

To use model checking in the context of programs, one needs to find a class of programs and a logic such that the formulas of the logic can be directly evaluated on the programs. This is where the nontraditional logic appears: model checking uses *propositional temporal logic*. Classical propositional logic talks about a world in which a truth value is given to propositions (Boolean variables): given values for the propositions, one can compute a truth value for a formula. Propositional *temporal* logic talks about a world where there are several possible states, and where a truth value is given to each proposition in each state. Furthermore, the possible moves from state to state are also specified. In the classical interpretation of temporal logic, the states are meant to correspond to time instants, and one can move from state A to state B if B temporally follows A. The formulas of temporal logic then include, in addition to the Boolean operators, temporal operators that can specify in which states properties should be true. One can, for instance, express that a formula should be true in some future state (eventually), or in all future states (always). One of the advantages of temporal logic is that it can simply and directly represent qualitative progress properties such as: if the scheduler allows each process to move infinitely often, then the concurrent program will terminate.

The link to programs is then quite direct. The states are the possible program states, and moving from a state to another corresponds to executing one program step. In other words, one represents the program by a finite-state transition system. To check that the program satisfies a temporal logic property, one then simply evaluates the formula on the transition system. This can be done quite efficiently: the complexity is linear in the size of the transition system and also linear in the size of the formula.

The transition systems used for model checking are global models of the system to be analyzed. They are thus inputless and nondeterministic. The non-determinism in the transition system comes either from the modeling of concurrency by interleaving, or from an absence of information about the behavior of some component of the system or of its environment. For example, an arbitrary value produced by the environment can be modeled by a nondeterministic choice of this value.

3 Linear or Branching?

Since the transition systems modeling programs are nondeterministic, an unavoidable issue is how the multiplicity of successors of states should be handled within the logic. This can be done either explicitly (the branching approach) or implicitly (the linear approach). In the branching approach (the one used in the initial paper on model checking [CES86]) the temporal operators introduced in the logic specify whether one considers *all* successors, or *some* successors of states. For instance, one can say that "in some next state" a property will be true, or that "in all reachable states" a condition will hold.

In the linear approach [LP85, VW86], the logic is designed to describe linear sequences of states (each state has only one successor). The model-checking problem is then defined slightly differently: one checks that all linear sequences that can be generated by the transition system (all possible executions of the program) satisfy the linear temporal logic formula. The linear approach is natural when the properties to be checked are expressed in terms of the executions of the program. The branching approach is well adapted when the properties are thought of in terms of the structure of the program. Also, the model-checking algorithm has a much lower complexity with respect to the size of the formula in the branching case. However, the complexity in the size of the program is the same, and this is usually the limiting factor (see next Section). For surveys, see [Wol89, Eme90].

4 Limiting Factors, Improvements and Extensions

It is not surprising that the limiting factor in model checking is the excessive size of the transition system representing the program: it often has far too many states to be constructed, or can even be infinite. This is often referred to as the *state-explosion problem*. Fortunately, several effective techniques have been introduced to cope with state explosion.

One type of technique attempts to build only part of the state space of the program, while still maintaining the ability to check the properties of interest. In this category, one finds "partial-order techniques" that attempt to avoid the wasteful representation of concurrency by interleaving [GW93], and "abstraction techniques" that replace the system to be checked by a simpler one in which the details irrelevant for the property to be checked have been suppressed [Kur94, CGL92, GL93]. Also related are "on the fly" approaches. These tailor the part of the state-space that is explored to the particular formula being checked [CVWY92].

A different direction is the "symbolic approach" to model checking. There, the idea is to represent implicitly rather than explicitly the states and transitions modeling the program [BCM+92]. The usual implicit representation is an efficient encoding of Boolean formulas known as Binary Decision Diagrams (BDDs) [Bry92]. The important idea is that the temporal formulas can be model-checked directly on the BDD representation, without ever building an explicit representation of the state space. This completely changes the limiting factor: the issue is no longer the size of the state space, but the size of its BDD representation. As a consequence, systems with astronomical number of states can sometimes be checked.

Among extensions to model checking, is the use of more powerful logics, for instance μ-calculus based logics [EL86]. A fairly recent research direction is the application of model checking to real-time systems. This required extending the type of model used for representing the programs as well as the logics used to express the properties. A common model for real-time systems is "timed automata" [AD94]. These are transition systems extended with a number of clock variables that can be tested and reset. The temporal logic is similarly extended to include quantitative timing conditions [AH93]. One interesting aspect of these developments is that one can still obtain model checking algorithms, even though the state space of the system to be checked is no longer necessarily finite [ACD93].

References

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–236, 1994.

[AH93] R. Alur and T. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35–77, May 1993.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 343–354, Albuquerque, New Mexico, January 1992.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[EL86] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the proposoitional mu-calculus. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 267–278, Cambridge, June 1986.

[Eme90] E.A. Emerson. Temporal and modal logic. *Handbook of theoretical computer science*, pages 997–1072, 1990.

[Flo67] R.W. Floyd. Assigning meaning to programs. In *Proceedings Symposium on Applied Mathematics*, volume 19, 1967.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Computer Aided Verification, Proc. 5th Int. Workshop*, volume 697, pages 71–84, Elounda, Crete, June 1993. Lecture Notes in Computer Science, Springer-Verlag.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[Hoa69] C.A.R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, New Jersey, 1994.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[Pnu85] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Proc. Advanced School on Current Trends in Concurrency*, pages 510–584, Berlin, 1985. Volume 224, LNCS, Springer-Verlag.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[Wol89] P. Wolper. On the relation of programs and computations to models of temporal logic. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Temporal Logic in Specification*, volume 398, pages 75–123. Lecture Notes in Computer Science, Springer-Verlag, 1989.