Journal of Universal Computer Science, vol. 13, no. 5 (2007), 639-649
submitted: 7/5/07, accepted: 25/5/07, appeared: 28/5/07 © *J.UCS*

Model Checking: Software and Beyond

Edmund M. Clarke, Flavio Lerda
(Computer Science Department, Carnegie Mellon University
Pittsburgh, PA 15213, USA
{emc+,flerda+}@cs.cmu.edu)

Abstract: This paper introduces model checking, originally conceived for checking finite state systems. It surveys its evolution to encompass finitely checkable properties of systems with unbounded state spaces, and its application to software and other systems.
Key Words: Formal Methods, Model Checking
Category: D.2.4, F.3.1, B.2

1 Introduction

Temporal logic model checking, first developed by Clarke and Emerson [Clarke and Emerson (1981)] and independently discovered by Queille and Sifakis [Queille and Sifakis (1982)], is an automated technique for the verification of finite-state systems. The specification is expressed as a logical formula. Model checking aims at establishing whether a system is a model for a given formula, i.e., if it is satisfies its specification. Model checking has had a big impact on formal verification over the past twenty five years [Clarke and Wing (1996), Clarke (2007)].

Section 2 describes the basic algorithm for temporal logic model checking, as well as some of the breakthroughs in this area. Section 3 introduces some recent developments and ideas for future research in this area.

2 Model Checking

The aim of formal methods is to prove that a given system satisfies its specification by formal means, e.g., mathematical proofs. In order to do so, the system and the specification need to be formally described. In model checking, the semantics of a system is usually given by means of a Kripke structure, a labeled graph that represents the possible states of a system and the transitions between them. The specification, instead, is expressed using temporal logic [Pnueli (1977)], an extension of propositional logic that allows reasoning about the relative timing of events. In the following, we will describe Kripke structures and temporal logics.

2.1 Kripke Structures

A Kripke structure is a directed graph where vertices are labeled by sets of atomic propositions. Vertices are called *states* and edges are called *transitions*. A subset of the

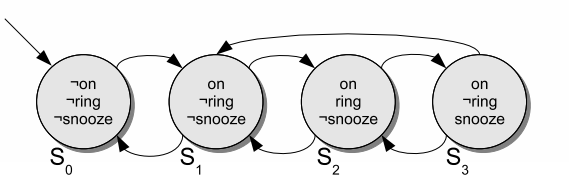

Figure 1: The Kripke structure of an alarm clock

states is designated as initial. An example of a Kripke structure, modeling the behavior of an alarm clock, is shown in Figure 1.

Given a set of *atomic proposition* AP, a **Kripke structure** is a tuple $M = (S, R, I, L)$ where S is a finite *set of states*, $R \subseteq S \times S$ is a *set of transitions* between states, $I \subseteq S$ is a *set of initial states*, and $L : S \rightarrow 2^{AP}$ is a *labeling function*. A *path* of a Kripke structure M is an infinite sequence of states $\pi = s_0, s_1, \ldots$ such that $s_0 \in I$ and, for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. Given a path π, π^i indicates the infinite path $s_i, s_{i+1}, \ldots$ and $\pi(i)$ indicates s_i. We assume that the transition relation R is total, i.e., every state has a valid successor and each finite path is a prefix of some infinite path.

The states of a Kripke structure represent the different states of a system. For example, the Kripke structure in Figure 1 has four states S_0, S_1, S_2, and S_3, corresponding to the alarm clock being *off*, *on*, *ringing*, and *snoozed* respectively. Transitions of a Kripke structure represent the possible evolutions of a system. For example, the Kripke structure in Figure 1 has a transition from state S_2 to state S_3, meaning that it is possible for the alarm clock to go from the state where it is ringing to the state where the alarm has been snoozed. However, there is no transition from state S_1 to state S_3, because the alarm cannot be snoozed if it is not ringing.

2.2 Temporal Logic

Temporal logic is a formalism for reasoning about time without introducing time explicitly. It is an extension of propositional logic where temporal operators are introduced to reason about the timing of events. In model checking, two alternative temporal logics are commonly used: Computation Tree Logic (**CTL**) [Ben-Ari et al. (1983)] and Linear-Time Temporal Logic (**LTL**) [Pnueli (1981)]. The former uses a branching notion of time and can reason about multiple paths at once. The latter uses a linear notion of time and considers a single path at a time. Both **CTL** and **LTL** can be expressed in terms of the temporal logic $\mathbf{CTL}^*$ [Emerson and Halpern (1986)]. **CTL** and **LTL** formulas represent two overlapping, but different subsets of $\mathbf{CTL}^*$ formulas.

The temporal logic $\mathbf{CTL}^*$ defines two *path quantifiers* (the universal path quantifier **A** and the existential path quantifier **E**) and four *temporal operators* (next **X**, eventually **F**, globally **G**, and until **U**). There are two types of formulas: *state formulas* and *path*

formulas. State formulas are defined as $f ::= p \mid f \vee f \mid f \wedge f \mid \neg f \mid \mathbf{A}\, g \mid \mathbf{E}\, g$ and path formulas are defined as $g ::= f \mid g \vee g \mid g \wedge g \mid \neg g \mid \mathbf{X}\, g \mid \mathbf{F}\, g \mid \mathbf{G}\, g \mid g\, \mathbf{U}\, g$ where $p \in AP$ is an atomic proposition. $\mathbf{CTL}^*$ formulas are state formulas according to the above definition.

A state s of M satisfies the formula p if s is labeled by p; it satisfies $\mathbf{A}\, g$ if every path π of M starting at s satisfies g; and it satisfies $\mathbf{E}\, g$ if there exists a path π of M starting at s that satisfies g. A path π of M satisfies the state formula f if $\pi(0)$ satisfies f; it satisfies $\mathbf{X}\, g$ if π^1 satisfies g; it satisfies $\mathbf{F}\, g$ if there exists an $i \geq 0$ such that π^i satisfies g; it satisfies $\mathbf{G}\, g$ if for every $i \geq 0$ path π^i satisfies g; and it satisfies $g_1\, \mathbf{U}\, g_2$ if there exists an $i \geq 0$ such that π^i satisfies g_2 and for every $0 \leq j < i$ path π^j satisfies g_1. Given a Kripke structure M, a state s, and a path π, we write $M, s \models f$ if state s of M satisfies state formula f, and $M, \pi \models g$ if path π of M satisfies path formulas g.

Consider again the Kripke structure of Figure 1. The initial state satisfies the formula $\mathbf{AG}\, \neg(ring \wedge snooze)$, since no state is both *ring*ing and *snooz*ing. On the other hand, it does not satisfy the formula $\mathbf{E}[\neg ring\, \mathbf{U}\, snooze]$, since every path that reaches a state labeled by *snooze* has to go through a state labeled by *ring*.

CTL and **LTL** are sub-logics of $\mathbf{CTL}^*$. **CTL** is obtained by requiring every temporal operator to be immediately preceded by a path quantifier and vice-versa. An **LTL** formula is a $\mathbf{CTL}^*$ formula of the form $\mathbf{A}\, g$ where no path quantifiers appear in g.

2.3 Model Checking Algorithm

The *model checking problem* can be stated as follows: given a Kripke structure M and a temporal logic formula f, determine if $M, s \models f$ holds for every initial state $s \in I$. Different model checking algorithms use specific techniques to answer this question in an efficient way. In this section, we present two model checking algorithms, one for specifications expressed using **CTL** and one for specifications expressed using **LTL**.

CTL Model Checking. The algorithm [Clarke and Emerson (1981), Clarke et al. (1986)] assumes that the specification f is a **CTL** formula. Every **CTL** formula can be rewritten in terms of only **EX**, **EG**, **EU**, $\neg$, and $\wedge$. The algorithm labels each state with the sub-formulas of f that hold at that state. It is applied recursively on the structure of the formula. The base case corresponds to atomic propositions: states that satisfy an atomic proposition are labeled by it. If the formula is of the form $\neg f$, all states that are not labeled by f are labeled by $\neg f$. If the formula is of the form $f_1 \wedge f_2$, all states that are labeled by f_1 and f_2 are labeled by $f_1 \wedge f_2$. If the formula is of the form $\mathbf{EX}\, f$, each predecessor of a state labeled by f is labeled by $\mathbf{EX}\, f$. If the formula is of the form $\mathbf{E}[f_1\, \mathbf{U}\, f_2]$, then all states that are labeled by f_2 and all predecessor of states labeled by f_2 or by $\mathbf{E}[f_1\, \mathbf{U}\, f_2]$ that are themselves labeled by f_1 are labeled by $\mathbf{E}[f_1\, \mathbf{U}\, f_2]$. In order to determine the states that satisfy $\mathbf{EG}\, f$, the algorithm considers the sub-graph G^f made of the states of M labeled by f. Each state in a non-trivial strongly connected component of G_f is labeled by $\mathbf{EG}\, f$. Moreover, every predecessor of a state labeled by $\mathbf{EG}\, f$ that is itself labeled by f is also labeled by $\mathbf{EG}\, f$.

LTL Model Checking. The algorithm [Vardi and Wolper et al. (1986)] assumes that the specification is an **LTL** formula, i.e., of the form $\mathbf{A}\, g$. Checking that the system satisfies the formula $\mathbf{A}\, g$ is equivalent to checking that it satisfies $\neg\mathbf{E}\neg g$. The algorithm constructs an automaton $B_{\neg g}$ that accepts the traces that satisfy $\neg g$. The automaton is composed with an automaton that accepts the traces of M. If the composition is empty, then M satisfies the specification $A\, g$. Otherwise, any of the traces of the composition is a counterexample.

The **LTL** model checking algorithm constructs a *Büchi automaton* [Thomas (1990)] $B_{\neg g}$ that accepts the traces that satisfy $\neg g$. Different algorithms for doing this have been proposed [Vardi and Wolper et al. (1986), Gerth et al. (1995)]. Given the Kripke structure M corresponding to the system, an equivalent *Büchi automaton* B_M is constructed. The model checking algorithm constructs, on-the-fly, the synchronous composition of the two automata and checks for the presence of accepting runs. In order to detect accepting runs, the algorithm performs a depth first search (DFS) on the graph corresponding to the composition. Every time an accepting state q is reached, a second DFS is started from that state to determine if state q can be reached from itself. If this is the case, the run, made of the states needed to reach state q from the initial state followed by an infinite number of repetitions of the states needed to reach state q from itself, is accepting. Therefore, it corresponds to a counterexample to the specification. If no accepting runs exist, we can conclude that every initial state satisfies the specification.

2.4 Symbolic Model Checking

One of the main limitations of model checking is the size of systems that can be verified. If the system is too large or the specification too complex, model checking might not terminate due to insufficient resources, e.g., running time or memory. This is known as the *state explosion problem*. One of the major breakthroughs in model checking has been the development of **symbolic model checking**, a technique that uses binary decision diagrams (BDDs) to represent sets of states and transitions. Model checking is performed directly on the BDD representations.

Reduced Ordered Binary Decision Diagrams, or ROBDDs, are an efficient data structure to represent Boolean functions [Bryant (1986)]. Given a set of variables $V = \{x_1, \ldots, x_n\}$, a *binary decision diagram* is a directed acyclic graph where each non-terminal vertex v has two successors $true(v)$ and $false(v)$ and is labeled by a variable $var(v)$, each terminal vertex is label by either $true$ or $false$, and a vertex r is designed as the root of the binary decision diagram. Given a variable order $\prec$, a total order over the variables in V, a binary decision diagram is ordered if, given any two nonterminal vertices u and v such that v is a successor of u, we have that $var(u) \prec var(v)$. An ordered binary decision diagram is reduced if: *(i)* there is only a single terminal vertex for each label; *(ii)* there exists no two nonterminal vertices u and v such that $var(u) = var(v)$, $false(u) = false(v)$, and $true(u) = true(v)$; and *(iii)* for every

nonterminal vertex v, $false(v) \neq true(v)$. In the following, we will use the term BDDs to refer to ROBDDs unless otherwise noted. BDDs provide a canonical representation of Boolean functions: given a set of variables $V = \{x_1, \ldots, x_n\}$, a BDD represents a Boolean function $f(x_1, \ldots, x_n)$. To determine the value of f for a given value of the inputs, start from the root node r. When at a nonterminal vertex v, if $var(v)$ is assigned the value $true$, move to the vertex corresponding to $true(v)$ otherwise move to $false(v)$. Proceed until a terminal vertex u is reached: the value of the function is the label of u. Boolean operations, e.g., $\wedge$, $\neg$, and $\exists$, can be applied directly to BDDs.

Boolean functions can be used to represent Kripke structures. Given a Kripke structure M, we can define a boolean encoding of its states using a finite set of variables $V = \{x_1, \ldots, x_n\}$. A set of states S can then be defined by its characteristic function $S(s)$ that evaluates to $true$ if state $s \in S$ and $false$ otherwise. We define a set of variables $V' = \{x'_1, \ldots, x'_n\}$, called next-state variables. The transition relation of M can then be represented by the Boolean function $R(s, s')$ over $V \cup V'$ that evaluates to $true$ if a transition from s to s' is possible.

A **CTL** model checking algorithm that operates on sets of states represented as BDDs was proposed in [Burch et al. (1990), McMillan (1992)]. As before, we need to consider only **EX**, **EG**, **EU**, $\wedge$, and $\neg$, as formulas can be rewritten to contain only these operators and atomic propositions. The set of states labeled by an atomic proposition can be represented as a BDD. Boolean operators can be represented by the corresponding Boolean operations on BDDs. Given a BDD representing the set of states satisfying the formula f, the BDD corresponding to $\mathbf{EX}\, f$ can be obtained by computing $ex(s) = \exists\, s' : f(s') \wedge R(s, s')$, where each operation can be performed directly on BDDs. The **EG** and **EU** operators cannot be computed directly. However, $\mathbf{EG}\, f$ can be defined as the greatest fixed point of $eg(Z) = f \wedge \mathbf{EX}\, Z$ and $\mathbf{E}[f_1\, \mathbf{U}\, f_2]$ as the least fixed point of $eu(Z) = f_2 \vee (f_1 \wedge \mathbf{EX}\, Z)$. Since the set of states is finite, and $eg(Z)$ and $eu(Z)$ are monotonic functions, the least and greatest fixed point of these functions can be computed iteratively. For instance, for computing $\mathbf{E}[f_1\, \mathbf{U}\, f_2]$, we start with Q being equal to the empty set of states. At each iteration, the states that satisfy f_2 and the states that have a successor belonging to Q and also satisfy f_1 are added to Q. This process continues until no new states can be added to Q. The set Q at the last iteration is equal to the set of states that satisfy $\mathbf{E}[f_1\, \mathbf{U}\, f_2]$. If the BDDs representing the set of states satisfying f_1 and f_2 are given, all operations can be performed directly using BDDs.

The main advantage of symbolic model checking is that the BDD representing a set of states can be much smaller than the set it represents. BDD-based symbolic model checkers have been used to check systems with a number of states that is beyond the reach of ordinary model checkers [Burch et al. (1990)]. However, BDD-based symbolic model checking is not the definitive solution to the state explosion problem: while on average BDDs provide a compact representation of Boolean functions, there are Boolean functions whose representations as BDDs are exponential in the number of

variables, which may make symbolic model checking impractical.

2.5 Bounded Model Checking

Boolean satisfiability (SAT), the typical example of an NP-complete problem, has been the focus of a lot of attention in recent years. SAT solvers have been developed that are able to handle problems with a large number of Boolean variables [Moskewicz et al. (2001)]. Most of modern SAT solvers take as input a Boolean formula in conjunctive normal form (CNF) and are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [Davis and Putnam (1960), Davis et al. (1962)]. If the formula is satisfiable, they produce a satisfying assignment, if it is not, they produce a proof of unsatisfiability.

In the context of model checking, SAT solvers have become popular since the introduction of bounded model checking (BMC) [Biere et al. (2003)]. The main idea of BMC is to encode the possible counterexamples to a given specification as a SAT formula. Since the formula must be finite, the counterexample needs to be finite as well. Specifically, BMC generates a formula that encodes the existence of a counterexample of a given length k.

For simplicity, we will consider only safety specifications of the form $\mathbf{AG}\, p$, where p is an atomic proposition. Extensions that are able to handle more complex specifications are present in the literature [Biere et al. (2003)]. Given an encoding of the states, we define three boolean predicates: $I(s)$, $R(s, s')$, and $F(s)$. Predicate $I(s)$ is $true$ if s is an initial state. Predicate $R(s, s')$ is $true$ if s' is a successor of s. Predicate $F(s)$ is $true$ if state s is labeled by atomic proposition p. The formula corresponding to a counterexample of length k is:

$$I(s_0) \wedge R(s_0, s_1) \wedge \ldots \wedge R(s_{k-1}, s_k) \wedge F(s_k) \tag{1}$$

which is satisfiable if and only if there is a counterexample of length k. By using a SAT solver it is possible to check if a counterexample exists. Moreover a counterexample can be obtained by analyzing the satisfying assignment produced by the SAT solver. If the formula is unsatisfiable, then there are no counterexamples of length k.

If no counterexamples of length k are present, it is necessary to repeat the check for a greater length. The procedure continues until either a counterexample is found or the completeness threshold is reached. The completeness threshold [Biere et al. (1999)] is the minimum length such that, if the specification is violated, there exists a counterexample shorter than that length. Bounds on the completeness threshold of various classes of specifications have been given in the literature [Biere et al. (2003), Kroening and Strichman (2003), Clarke et al. (2004)]. However, in practice, the computed bounds are often quite large. In this case, the verification terminates when the problem becomes intractable, without being able to prove that the system satisfies the specification.

The approach as presented is, therefore, inherently incomplete, unless tight bounds on the completeness threshold can be determined. However, complete methods based

on bounded model checking that rely on alternative methods to determine termination have been proposed [Sheeran et al. (2000), McMillan (2003), Prasad et al. (2005)].

3 Software and Beyond

The previous section described some of the seminal work in model checking. One of the application domains in which model checking has seen most successes is hardware: model checkers are currently used by many semiconductor manufacturers [Clarke and Wing (1996)]. One of the reasons is that hardware designs are well suited for model checking: they are defined by using Boolean gates and their semantics is straightforward. At the same time, historic events have provided the circumstances for technology transfer. For instance, soon after Intel had to recall a large number of Pentium processors because of a design bug, researchers were able to show that the bug could have been detected by using formal verification [Clarke et al. (1996)]. Since then, many chip design companies have been using model checking and other formal verification techniques.

Software Model Checking. More recently, software has been the focus of much effort in the model checking community. Tools like SLAM [Ball and Rajamani (2000)], BLAST [Henzinger et al. (2002)], MAGIC [Chaki et al. (2003)], and CBMC [Clarke et al. (2003)], just to cite a few, have been developed that are aimed at the verification of software. Some of the techniques for software model checking that have been very successful are *predicate abstraction* [Graf and Saïdi (1997)] and *counterexample-guided abstraction refinement* [Kurshan (1994), Clarke et al. (2000)]. These techniques have made model checking of software feasible.

Infinite-State Systems. While the state space of a software may be very large, under certain conditions, it is still finite. This is not true of all types of systems. In recent years, efforts have been made to address formal verification of *infinite state systems*. In general, model checking cannot be applied to an infinite state system directly, because model checking, in its most basic form, would enumerate the states of the system and therefore never terminate. Techniques have been developed that allow model checking infinite state systems. In the following, we will briefly describe two important examples: *(i) timed systems*; and *(ii) hybrid systems*.

Timed Systems. In some cases, correctness of a system depends on the *exact timing of events*. As a consequence, models must include the time at which events occur. A commonly used formalism to model and reason about timed systems is *timed automata* [Alur et al. (1990), Alur and Dill (1994)]. Timed automata are an extension of finite state automata that define a set of *real-valued clock variables*. The state space of a timed automaton can be infinite as the clocks assume values from the reals. Specialized algorithms and data structures have been developed that enable model checking of timed automata [Yovine (1997), Wang (2001), Larsen et al. (1995)].

Hybrid Systems. Another example of infinite state systems, *hybrid systems* are characterized by the presence of *discrete and continuous components*. The continuous components are usually defined using differential equations. Most techniques for model checking hybrid systems represent sets of continuous states using specialized data structures. Polyhedra, defined as the intersection of a set of half-spaces, are a typical example of such data structures. Operations on these data structures are usually quite expensive and they introduce over-approximations in order to guarantee that the sets that are obtained can be represented by the chosen data structure.

Timed and Hybrid Software. Most of the existing approaches for model checking of timed and hybrid systems focus on the infinite-state components. However, when looking at a complex piece of software that includes this type of component, software model checking techniques are needed to make verification feasible. We believe that techniques that apply existing software model checking to models that include time and continuous dynamics are crucial in making model checking scale to large, complex systems.

In the following, we will present two approaches, one for timed systems and one for hybrid systems, that allow applying software verification techniques to systems that include time or continuous dynamics.

3.1 Timed Automata Verification

While real-time systems are, in general, infinite-state systems because time is a continuous variable, under certain assumptions, it is possible to reduce the problem to finite-state model checking. In recent work [Clarke et al. (2007)], we have investigated techniques that, by using a well known mapping from infinite-state timed automata to finite-state *region automata*, can leverage the recent advances in model checking. In particular, we have developed abstraction and optimization techniques to reduce the size of the state space that needs to be visited by the model checker. We introduced a new abstraction technique, called GOABSTRACTION, which aims at reducing the size of the state space while preserving the validity of interesting specifications.

As for future work, we are developing technique to handle some of the other sources of the state explosion problem for this class of systems. We are currently looking at techniques that deal with large constants in the constraints and at ways of applying counterexample-guided abstraction refinement [Clarke et al. (2000)] to find the best abstraction that is able to prove the specification at hand.

3.2 Hybrid Systems Verification

Another area of active research is the verification of hybrid systems, i.e., systems in which discrete and continuous components coexist. One of the challenges is that the continuous components give rise to an infinite set of possible states.

The approach we proposed in [Scherer et al. (2005)] focuses on control software, a particular kind of software that interacts with a continuous environment. Very often such software is made of a set of periodic tasks, that are executed on a fixed schedule. In our approach, instead of looking at sets of continuous states, we look at continuous states individually. Given the amount of time during which the continuous system is evolving, the period of the tasks, the continuous state will evolve following the differential equations. If the differential equations are deterministic and time invariant, we can use numerical simulation algorithms to compute the continuous state that is reached after time elapses. One of the advantages is that numerical simulation algorithms are quite efficient and can be applied to a large class of differential equations. It is then possible to alternate continuous transitions —implemented by numerical simulation— and discrete transitions —corresponding to statements in the software— to perform model checking. Experimental results on this technique are promising. This approach, however, introduces approximations in two places: the numerical simulation introduces some numerical error; and, in order to guarantee termination, it is necessary to introduce a maximum precision that is used to compare the continuous parts of states.

As for future work, we would like to determine in which cases we are able to guarantee correctness, i.e., avoid the approximation needed to guarantee termination. We would like to be able to evaluate the amount of imprecision introduced, as well as define a refinement step that increases the precision where it is most useful.

Acknowledgments

This research was sponsored by the National Science Foundation under grant nos. CNS-0411152, CCF-0429120, CCR-0121547, and CCR-0098072, the US Army Research Office under grant no. DAAD19-01-1-0485, the Office of Naval Research under grant no. N00014-01-1-0796, the Defense Advanced Research Projects Agency under subcontract no. SA423679952, the General Motors Corporation, and the Semiconductor Research Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

References

[Alur et al. (1990)] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-Checking for Real-Time Systems. In *Proc. of the 5th Annual IEEE Symposium on Logic in Computer Science*, 1990.

[Alur and Dill (1994)] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[Ball and Rajamani (2000)] Thomas Ball and Sriram K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *Proc. of the 7th International SPIN Workshop*, 2000.

[Ben-Ari et al. (1983)] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The Temporal Logic of Branching Time. *Acta Informatica*, 20:207–226, 1983.

[Biere et al. (2003)] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. *Advances in Computers*, 58:118–149, 2003.

[Biere et al. (1999)] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proc. of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 1999.

[Bryant (1986)] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):667–691, 1986.

[Burch et al. (1990)] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^20 States and Beyond. In *Proc. of the 5th Annual Symposium on Logic in Computer Science*, 1990.

[Chaki et al. (2003)] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular Verification of Software Components in C. In *Proc. of the 25th International Conference on Software Engineering*, 2003.

[Clarke et al. (1986)] Edmund Clarke, E Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[Clarke (2007)] Edmund M. Clarke. The Birth of Model Checking. Technical Report CMU-CS-TR-110, Carnegie Mellon University, 2007.

[Clarke and Emerson (1981)] Edmund M. Clarke and E. Allen Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Proc. of Workshop on Logic of Programs*, 1981.

[Clarke et al. (1996)] Edmund M. Clarke, Steven M. German, and Xudong Zhao. Verifying the SRT Division Algorithm using Theorem Proving Techniques. In *Proc. of the the 8th International Conference on Computer Aided Verification*, 1996.

[Clarke et al. (2000)] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of the 12th International Conference Computer Aided Verification*, 2000.

[Clarke et al. (2004)] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and Complexity of Bounded Model Checking. In *Proc. of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2004.

[Clarke et al. (2003)] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral Consistency of C and Verilog Programs using Bounded Model Checking. In *Proc. of the 40th Design Automation Conference*, 2003.

[Clarke et al. (2007)] Edmund M. Clarke, Flavio Lerda, and Muralidhar Talupur. An Abstraction Technique for Real-Time Verification. In *Proc. of the GM R&D Workshop on Next Generation Design and Verification Methodologies for Distributed Embedded Control System*, 2007.

[Clarke and Wing (1996)] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[Davis et al. (1962)] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[Davis and Putnam (1960)] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[Emerson and Halpern (1986)] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, 1986.

[Gerth et al. (1995)] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. In *Proc. of the 15th International Symposium on Protocol Specification, Testing and Verification*, 1995.

[Graf and Saïdi (1997)] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Proc. of the 9th International Conference on Computer Aided Verification*, 1997.

[Henzinger et al. (2002)] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy Abstraction. In *Proc. of the 29th Symposium on Principles of Programming Languages*, 2002.

[Kroening and Strichman (2003)] Daniel Kroening and Ofer Strichman. Efficient Computation of Recurrence Diameters. In *Proc. of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2003.

[Kurshan (1994)] Robert P. Kurshan. *Computer-Aided Verification of coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[Larsen et al. (1995)] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, 1995.

[McMillan (1992)] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[McMillan (2003)] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In *Proc. of the 15th International Conference Computer Aided Verification*, 2003.

[Moskewicz et al. (2001)] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the 38th Design Automation Conference*, 2001.

[Pnueli (1977)] Amir Pnueli. The Temporal Logic of Programs. In *Proc. of the 18th Annual Symposium on Foundations of Computer Science*, 1977.

[Pnueli (1981)] Amir Pnueli. The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Prasad et al. (2005)] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A Survey of Recent Advances in SAT-based Formal Verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[Queille and Sifakis (1982)] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, 1982.

[Scherer et al. (2005)] Sebastian Scherer, Flavio Lerda, and Edmund Clarke. Model Checking of Robotic Control Systems. In *Proc. of the 8th International symposium on Artificial Intelligence, Robotics and Automation in Space*, 2005.

[Sheeran et al. (2000)] Mary Sheeran, Satnam Singh, and Gunnar Stalmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Proc. of 3rd International Conference on Formal Methods in Computer-Aided Design*, 2000.

[Thomas (1990)] Wolfgang Thomas. Automata on Infinite Objects. *Handbook of Theoretical Computer Science*, B:133–191, 1990.

[Vardi and Wolper et al. (1986)] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proc. of the 1st Symposium on Logic in Computer Science*, 1986.

[Wang (2001)] Farn Wang. RED: Model-Checker for Timed Automata with Clock-Restriction Diagram. In *Proc. of Workshop on Real-Time Tools*, 2001.

[Yovine (1997)] Sergio Yovine. KRONOS: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, December 1997.