Temporal Logic Model Checking

Edmund Clarke,[1] Ansgar Fehnker,[2] Sumit Kumar Jha[1], and Helmut Veith[3]

[1] School of Computer Science, Carnegie Mellon University, PA 15213, Pittsburgh, U.S.A.
{emc,jha+}@cs.cmu.edu

[2] National ICT Australia, University of New South Wales, Australia
ansgar@cse.unsw.edu.au

[3] Institut für Informatik, Technische Universität München, Munich, Germany
veith@in.tum.de

1 Introduction and Overview

Errors in safety-critical systems such as embedded controllers may have drastic consequences and can even endanger human life. It is therefore crucially important to verify the correctness of such systems in a logically precise manner during system design itself. This chapter is an introduction to model checking—an automated and practically successful approach for the formal verification of the correctness of hardware and software systems.

The origins of model checking date back to the early 1980s, when Clarke and Emerson [8] and, independently, Queille and Sifakis [26] introduced a new algorithmic approach for the verification of computer systems. Their approach amounts to checking the satisfaction of a logical specification over a system model which is represented by an annotated directed graph; hence, the term *model checking*. Prior to that, the use of temporal logic for the analysis and specification of computer systems had been advocated by Pnueli [25], and model checking has in fact been employing variants of temporal logic as the predominant specification language ever since. Experiments with early model checkers quickly made clear that the size of the model represents the crucial technical barrier for realizing the full potential of model checking in practically relevant verification tasks. In turn, the *state explosion problem* is the key to appreciating the technical achievements in model checking during the last decades. At the time of this writing, model checking techniques have achieved practical significance for the hardware and software industries, routinely analyzing digital circuit designs and programs, with more than 10^{100} system states in some cases.

The aim of this chapter is to introduce those important lines of research which transformed model checking from a method of primarily theoretical interest into a powerful tool for the analysis of computer hardware and software. We shall focus in particular on those subjects which have shaped our thinking about model checking in the verification group of Carnegie Mellon University, most notably symbolic model checking and abstraction. The development of *symbolic model checker* [6, 24] was arguably a turning point in the formal methods field. Employing a combination of binary decision diagrams and fixed-point algorithms, the symbolic model verifier (SMV) became the first model checker to verify models with hundreds of Boolean variables and a tool to benchmark new ideas for more than a decade. Thus, after a brief theoretical introduction into logical foundations of model checking in Section 2, we will describe the methodology behind SMV in Section 3.1; we also cover bounded model checking, a more recent orthogonal symbolic model checking paradigm which is based on SAT solvers. Sections 3.2 and 3.3 finally are devoted to abstraction, the key principle underlying the big advances in software verification during the last few years. The focus in these sections will be on counterexample-guided abstraction refinement as well as predicate abstraction, both of which constitute key features of modern software verification tools.

Most of the material included in this chapter is self-contained, requiring only a general mathematical maturity. However, we explicitly advise the reader that the space restrictions imposed by the handbook format render it impossible to provide either a comprehensive coverage of the subject or even an extended bibliography which gives full credit for all presented concepts impossible. The papers and books cited here should serve mainly as entry points to the literature, with an emphasis on newer papers not yet listed in the standard literature. The most evident omission in this chapter is an extensive treatment of the automata-theoretic approach to model checking [30]; just like temporal logic model checking, the alternative automata-theoretic approach has also produced powerful model checkers such as SPIN. A more comprehensive survey on model checking including extensive citations can be found in [12], more detailed accounts on logical questions in [13, 15], an introduction to SPIN and the automata-theoretic method in [21], and an easily accessible primer on logic and verification in [22].

2 Fundamentals of Model Checking

A model checker is an algorithm which determines whether a system K satisfies a specification ϕ, formally $\mathsf{K} \models \phi$. In contrast to stochastic methods such as testing, a positive result of the model checker provides a logically precise assertion of system correctness—albeit not in terms of a step-by-step-proof, but by virtue of the construction and correctness of the model checking algorithm. If the system K is found to violate the specification ϕ, i.e., $\mathsf{K} \not\models \phi$, then most model checkers will compute a diagnostic counterexample C which helps to localize the source of the error.

The fundamental notion of model checking has been adopted to diverse application areas and formal methods even beyond verification; these areas cannot all be treated in detail within the scope of this chapter. We shall therefore concentrate on the classical model checking framework where the system K is given as a *Kripke structure*, and the specification ϕ is a *temporal logic formula.*

Kripke structures

The notion of a *state* is at the center of model checking. A state is a momentary description of a system at a given point in time, similar to a point in a physical phase space. When a system has only a finite number of possible states, we speak of a finite state system.

A Kripke structure describes the dynamics of a finite state system by a finite directed graph whose vertices denote the states and whose edges denote transitions between the states. The states are labelled by atomic propositions which denote the properties of each state. For example, the states of the Kripke structure can be taken to describe the different states of a microprocessor, and the labels describe the values of the registers associated with a state. In the simple examples used in this chapter, the atomic propositions will typically just have the form of Boolean variables; in practice, atomic formulas often describe properties of the internal variables of the real-life system we model, e.g., "`counter == 5`" or "`x == -1`".

Formally, a Kripke structure is a tuple $\mathsf{K} = (S, S_0, R, L, \mathrm{AP})$ where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation, AP is the set of atomic properties, and $L : S \to 2^{\mathrm{AP}}$ is the labelling function. R is required to be total, i.e., for each $s \in S$ there exists $t \in S$ such that $(s, t) \in R$. When the context is understood, S_0 and AP are often omitted. For simplicity, we will assume in this chapter that $S_0 = \{s_0\}$ contains a single initial state.

A path π is an *infinite* sequence of states $\pi = p_0, p_1, \ldots$ such that for all i, $(p_i, p_{i+1}) \in R$. For $i \geq 0$, we define $\pi^i = p_i, p_{i+1}, \ldots$ to be the path starting at p_i, and $\pi[i] = p_i$. Note that the totality of R guarantees that all finite paths can be extended infinitely.

Example 1. The Kripke structure M in Fig. 1 shows a simple example of mutual exclusion between two processes A and B. Label C_X denotes that process X is in the critical state, and label T_X denotes that X is trying to enter its critical section. By visual inspection, the reader will easily verify that mutual exclusion is guaranteed, i.e., that no state can be reached where both C_A and C_b hold. Note that this example of mutual exclusion is very simplified: no fairness guarantees are given and a process may stay in the critical section forever.

Several remarks about the use of Kripke structures in model checking are in place here:

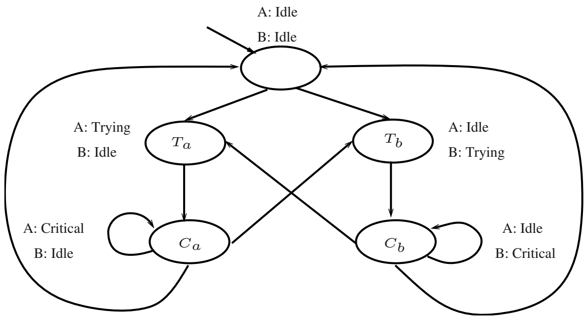

Fig. 1. A Kripke structure modelling a trivial mutual exclusion protocol

(a) Kripke structures are a versatile *mathematical model* we consider in model checking. Practical model checkers use specific programming languages and not Kripke structures as their *input language*. The complexity issues arising from direct compilation of a program into a Kripke structure constitute the central algorithmic challenge of model checking (the "state explosion problem") and will be addressed in Section 3.1.

(b) Kripke structures are in general *non-deterministic*, i.e., from a given state s there will be more than one outgoing transition. Non-determinism is a natural way to describe the effects of external input to a system. In Section 3.2 we will see that non-determinism also arises when we approximate the behavior of large systems by relatively small Kripke structures.

(c) Kripke structures are closely related to finite automata, finite transition systems, Moore machines, process algebraic expressions and similar concepts whose differences are to a large extent rooted in pragmatic aspects and tradition. In principle, every finite state system can be represented by a Kripke structure. In Section 3.3 we will investigate extensions of model checking to deal with infinite state systems.

Temporal logic

Given the atomic propositions of the Kripke structure, we can use Boolean logic to describe compound properties of *single* states. For example, we write $\mathsf{K}, s \models f \wedge \neg g$ to denote that state s of system K has label f, but not label g. However, simple Boolean logic does not account for the temporal dynamics of systems: In Boolean logic we cannot express properties such as "f is an invariant", "f always precludes g", or "f will persist up to the time when g occurs". This is where the temporal logics LTL, CTL and CTL* come into play.

We will first describe the linear time logic (LTL) which is defined over paths of the system. LTL extends Boolean logic by two operators U and X. Given a path $\pi = p_0, p_1, \ldots$, the Boolean and temporal operators have the following recursive semantics:

$\mathsf{K}, \pi \models f$	iff $f \in L(\pi[0])$ where $f \in \mathrm{AP}$
$\mathsf{K}, \pi \models \phi \wedge \psi$	iff $\mathsf{K}, \pi \models \phi$ and $\mathsf{K}, \pi \models \psi$
$\mathsf{K}, \pi \models \neg\phi$	iff $\mathsf{K}, \pi \not\models \phi$
$\mathsf{K}, \pi \models \mathsf{X}\phi$	iff $\mathsf{K}, \pi^1 \models \phi$ *" ϕ is true in the next state"*
$\mathsf{K}, \pi \models \phi \mathsf{U} \psi$	iff there is an $i \geq 0$ such that $\mathsf{K}, \pi^i \models \psi$ and for all j with $0 \leq j < i$ we have $\mathsf{K}, \pi^j \models \phi$ *" ϕ is true until ψ becomes true"*
$\mathsf{K} \models \phi$	iff for all paths π starting in s_0, we have $\mathsf{K}, \pi \models \phi$.

Disjunction $\phi \vee \psi$ and implication $\phi \rightarrow \psi$ are as usual defined by $\neg(\neg\phi \wedge \neg\psi)$ and $\neg(\phi \wedge \neg\psi)$, respectively. Moreover, $\mathsf{F}\psi)$ and $\neg(\phi \wedge \neg\psi)$, respectively. Moreover, $\mathsf{F}\phi$ is defined $\textbf{true}\mathsf{U}\phi$, and $\mathsf{G}\phi$ is defined $\neg\mathsf{F}\neg\phi$. With these definitions, $\mathsf{F}\phi$ intuitively means *"ϕ will be true at some time in the future"* and $\mathsf{G}\phi$ means *"ϕ will always be true in the future."*

Note that LTL specifications describe properties of paths; an LTL specification holds true on a Kripke structure, if it holds true on *every path* starting at the initial state s_0. Thus, an LTL specification contains an implicit universal quantification over all paths starting at s_0.

We will now introduce the computational tree logics CTL* and CTL which enable us to quantify over paths explicitly. CTL* extends LTL by an operator A as follows:

$\mathsf{K}, \pi \models \mathsf{A}\phi$	iff for all paths σ starting in state $\pi[0]$, we have $\mathsf{K}, \sigma \models \phi$.

Existential path quantification $\mathsf{E}\phi$ is defined as an abbreviation for $\neg\mathsf{A}\neg\phi$. The important specification logic CTL is the syntactic fragment of CTL* which uses the LTL operators and the CTL operators only pairwise, i.e., CTL contains exactly the following temporal operators: $\mathsf{AX}, \mathsf{EX}, \mathsf{AU}, \mathsf{EU}, \mathsf{AF}, \mathsf{EF}, \mathsf{AG}, \mathsf{EG}$. For example, the CTL* formula $\mathsf{AXX}f$ is not in CTL, since the second occurrence of X is not preceded by E or A. Finally, ACTL (ACTL*) is the fragment of CTL (CTL*) where negation is restricted to atomic formulas, and only the path quantifier A is allowed.

CTL and LTL specifications

Since CTL specifications can quantify repeatedly over paths, CTL and CTL* are examples of *branching time logics*, while LTL is a *linear time logic*. It can be shown that the expressive power of CTL and LTL is not comparable, and both are strictly contained in CTL*. The algorithms and paradigms for CTL and LTL model checking are sufficiently different so as to provoke a controversial discussion in the literature as to which is preferable, branching time or linear time logic. Practical applications employ variants of either CTL or LTL.

Example 2. In the example Kripke structure M used previously, the mutual exclusion property is specified in CTL as $\mathsf{AG}\neg(C_a \wedge C_b)$. The *liveness property* that each of the processes enters its critical region infinitely often is given by the CTL formula $\mathsf{AG}(\mathsf{AF}(C_a)\wedge\mathsf{AF}(C_b))$. The two properties are also expressible in LTL by $\mathsf{G}\neg(C_a \wedge C_b)$ and $\mathsf{GF}(C_a) \wedge \mathsf{GF}(C_b)$, respectively.

To shed more light on the difference between CTL and LTL, let us consider the equivalence relation between Kripke structures induced by CTL and LTL specifications, i.e., we say that two Kripke structures are equivalent if there is no CTL (or LTL) specification which holds true for one, but not for the other. For LTL, this equivalence relation is known as *trace equivalence*, i.e., two Kripke structures are trace equivalent iff they have the same paths from the initial state. For CTL, in contrast, the equivalence relation is *bisimulation*, a central notion in process algebra which we describe in more detail below.

Bisimulation and simulation

Bisimulation can be defined by a combinatorial two-player game [28] where one player (the *spoiler*) attempts to show that two Kripke structures $\mathbf{A}$ and $\mathbf{B}$ are different, and the second player (the *duplicator*) attempts to show that the structures are equivalent. The game starts with two pebbles placed on the initial states of the two structures. Each round of the game proceeds as follows: (i) If the pebbles are located at states with different labels, the spoiler wins, and the game terminates. (ii) The spoiler chooses one Kripke structure, and moves the pebble along an edge in this Kripke structure. (iii) In the other Kripke structure, the duplicator moves the other pebble, and the game continues at step (i). The Kripke structures are bisimilar if the spoiler does not have a winning strategy.

If the spoiler has a winning strategy, i.e., if the Kripke structures are not bisimilar, then the strategy can already be expressed by a CTL formula which uses the temporal operators AX and EX and distinguishes $\mathbf{A}$ from $\mathbf{B}$. Conversely, a distinguishing specification gives rise to a winning strategy for the spoiler.

Closely related to bisimulation is the notion of *simulation*, which orders Kripke structures with respect to their behaviors. For two Kripke structures, $\mathbf{A}$ and $\mathbf{B}$, simulation $\mathbf{A} \preceq \mathbf{B}$ can be defined by a similar two-player game as above, with the following restriction: the spoiler always plays on Kripke structure $\mathbf{A}$, and the duplicator always plays on Kripke structure $\mathbf{B}$. If the spoiler does not have a winning strategy, then $\mathbf{A} \preceq \mathbf{B}$ holds true. Intuitively, in this case, $\mathbf{B}$ has more behavior than $\mathbf{A}$, as the duplicator can duplicate every move on $\mathbf{B}$ which the spoiler does on $\mathbf{A}$. Simulation has the important property that it preserves ACTL* specifications: If $\mathbf{A} \preceq \mathbf{B}$ and $\mathbf{B} \models \phi$ for an

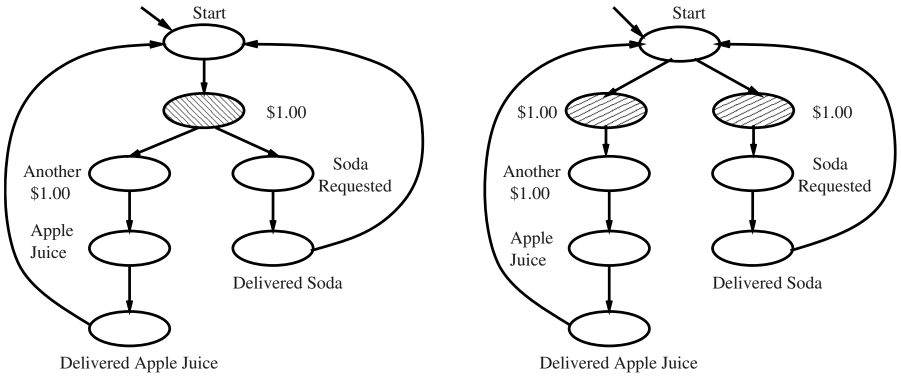

Fig. 2. Simulation versus bisimulation

ACTL* specification ϕ, then $\mathsf{A} \models \phi$. We will see that this relationship has crucial algorithmic applications in model checking.

It is easy to see that bisimulation equivalence implies trace equivalence: suppose the Kripke structure A has a path π_1 not contained in B. Then there exists a finite prefix π_1' of π_1 which is not present in B. It is now easy to construct a CTL specification of the from $\mathsf{EX}(\ldots \wedge \mathsf{EX}(\ldots \wedge \mathsf{EX}(\ldots)))$ which stipulates the existence of π_1'. Example 3 demonstrates a classical case where we have trace equivalence, but not bisimulation equivalence.

Example 3. Fig. 2 demonstrates a case where two Kripke structures are trace equivalent but not bisimilar. Both describe a simplified vending machine for soda (at \$1 per serving) and apple juice (at \$2). In the right machine, money is inserted in separate slots, and thus, the first dollar determines the purchase. It is easy to see that both structures have the same traces, i.e., they are trace equivalent, but not bisimilar. This is also reflected in the bisimulation game: When the spoiler moves the pebble to the shaded state in the left structure he keeps a future choice between juice and soda, but in the right structure, the duplicator has to decide on his tastes at this moment. It is easy to see that both structures have the same traces, i.e., they are trace-equivalent, but not bisimilar. This is also reflected in the bisimulation game: When the spoiler moves the pebble to the shaded state in the left structure, he keeps a future choice between juice and soda, but the duplicator has to decide on his tastes at this moment in the right structure.

Similar situations occur, for example, in computer security when instead of coins we enter passwords which enable us to call operating system functions. In such situations, the difference between bisimulation and trace equivalence may become crucial.

Principle algorithmic aspects of model checking

In contrast to many other applications of logic in computer science, most questions related to temporal logics and model checking are decidable, and indeed have often reasonable complexity. Quite surprisingly, the problem of deciding $\mathsf{K} \models \phi$ can be solved in linear time $O(|\phi| \times (|S| + |R|))$ for CTL. This explicit CTL model checking algorithm proceeds bottom-up on the formula structure which yields the factor $|\phi|$ above. For each subformula ϕ', the algorithm labels the states of K which satisfy ϕ' in time $O(|S| + |R|)$. For LTL and CTL*, the model checking problem is complete for the complexity class PSPACE. Deciding the *validity* of specifications, i.e., determining whether a specification is always true (independent of the Kripke structure) is EXPTIME-complete for CTL, PSPACE-complete for LTL and 2EXPTIME-complete for CTL*. In practice, however, the complexity bounds obtained from these general abstract considerations are usually not sufficient to verify industrially relevant systems. This question will be addressed in the next section.

3 State Explosion and Efficient Verification Methods

In most practical applications of model checking, the size of the state space is too large by several magnitudes as to allow naive verification algorithms which compile the input into a Kripke structure, and perform the model checking algorithm thereafter. Even an extremely simple system containing three 32-bit integer variables has a theoretical state space of $(2^{32})^3$. A quick calculation shows that for a terahertz processor which can evaluate one state per system cycle, it will take around 10^9 years to enumerate all states. Since the state space is in general exponential in the memory a program uses, it is entirely unrealistic to perform model checking on an explicit Kripke structure. This principal problem, commonly referred to as the “state explosion problem,” is the core issue in most scientific research in model checking. The practical success of a model checking technique depends most crucially on its ability to alleviate the state explosion. We distinguish several principal ways to address state explosion.

- **Symbolic verification:** In this approach, which made model checking a practical technique, the transition relation of the Kripke structure is encoded in a Boolean formalism (either plain Boolean formulas or specific data structures such as ordered binary decision diagrams (OBDDs) [5]), thereby obtaining a potentially exponential compression factor. Specific “symbolic” model checking algorithms are devised to operate on such system representations. We will describe symbolic model checking methods for CTL and LTL below.
- **State space exploration:** Alternatively to symbolic representation, a model checking algorithm may also attempt to explore the state space for specification violations on the fly, i.e., by a systematic depth-first search starting from the initial state. This method is based on the insight that LTL specifications can be transformed into trace equivalent *Büchi automata* [12] which monitor the state space exploration. While the size of

the state space sets a principal limit to this approach, it is successful in *finding errors*, in particular in combination with abstraction and reduction methods (described next).
- **Abstraction and reduction methods:** In this category we subsume more aggressive methods which are typically orthogonal to both symbolic and exploration techniques. Their common characteristic is their attempt to restrict the state space by semantic considerations, i.e., properties known about the system or derived from its description. Typical examples of such methods include *abstraction* [11], where system states are partitioned into equivalence classes, *partial order reduction* [18], which curbs the state space explosion incurred by concurrency, or *symmetry reductions* [9, 16], which employ the natural symmetry between repeated system components. In Section 3.2 we shall describe abstraction and counterexample-based abstraction refinement in more detail.

3.1 Symbolic model checking

Recall from above that the characteristic step in symbolic model checking is to represent the transition relation R of a Kripke structure $\mathsf{K} = (S, S_0, R, L, \mathrm{AP})$ in terms of a Boolean function f_R in such a way that every state $s \in S$ is described by a unique Boolean vector $\bar{s}$, and $f_R(\bar{s}, \bar{t}) = \textbf{true}$ iff $(s, t) \in R$. Since for natural binary encoding the size of the Boolean vector is logarithmic in $|S|$, the size of f_R may—in principle—also be polynomial or even linear in $\log |S|$. While there is no mathematical guarantee for this compression to occur (in fact, information theoretic counting arguments easily show that such a compression is very rare), practical systems tend to have many regularities, and often allow significant compression.

Note that in the computation of f_R, the model checker does not have access to the Kripke structure K (which is too large by assumption), but only to the input program. In this setting, choosing binary representations $\bar{s}$ for states s is usually a very natural step, since each system state s describes a program state at a given time, and thus corresponds to specific values of the program variables; these program variables themselves have natural binary representations which the model checker can reuse. In fact, a close correspondence between symbolic variables and program variables is often advantageous: knowledge about the semantic relationship between symbolic variables can facilitate both compression and abstraction.

CTL verification: verification by fixed point computation

Recall that in the specification logic CTL, every LTL operator is immediately preceded by either E or A. Since the semantical definitions of $\mathsf{K}, \pi \models \mathsf{A}\phi$ and $\mathsf{K}, \pi \models \mathsf{E}\phi$ depend only *on the first state* $\pi[0]$ of path π, formulas with a leading A or E are called state formulas. A model checking algorithm can associate each CTL state formula ϕ with the set of states $[[\phi]]$ where ϕ holds true. For

CTL formulas ϕ and Kripke structures K, the set $[[\phi]]$ can be computed by a fixed-point algorithm which can be implemented symbolically.

To illustrate the principles of symbolic model checking for CTL, let us consider the specification $\mathsf{EF}\phi$ ("a state with property ϕ is reachable"). Given the set $[[\phi]]$ of states where ϕ holds true, $[[\mathsf{EF}\phi]]$ is inductively defined as follows:

- If $s \in [[\phi]]$, then $s \in [[\mathsf{EF}\phi]]$.
- If $s \in [[\mathsf{EF}\phi]]$ and $R(t,s)$, then $t \in [[\mathsf{EF}\phi]]$.
- Nothing else is in $[[\mathsf{EF}\phi]]$.

This gives rise to the fixed-point characterization

$$\mathsf{EF}\phi \equiv \mu Y.\phi \vee \mathsf{EX}\ Y$$

where $\mu Y.f(Y)$ denotes the least fixed-point of formula $f(Y)$. The fixed-point extension of temporal logic is called the μ-calculus, and has been studied extensively, see [12] for detailed definitions and references. From the μ-calculus characterization of $\mathsf{EF}\phi$ we can derive the following fixed-point algorithm to compute $[[\mathsf{EF}\phi]]$.

```
$Y := \emptyset$
**repeat**
    $Y' := Y$;
    $Y := [[\phi]] \cup \mathbf{pre}(Y)$;
**until** $Y = Y'$
```

where $\mathbf{pre}(Y)$ denotes the *pre-image* operator

$$\mathbf{pre}(Y) := \{s \mid \exists t.(s,t) \in R \wedge t \in Y\}.$$

A closer study shows that all CTL formulas can be expressed using fixed points, propositional logic, and the temporal operator EX, i.e., pre-image computation. It remains to be shown as to how the fixed-point algorithms can be implemented symbolically.

The crucial idea is to represent not only R by f_R, but also sets of states by Boolean functions. A set Y of states is represented by its characteristic Boolean function $Y(\bar{z}) := \bigvee_{s \in Y} \bar{z} \equiv \bar{s}$ which is **true** iff $\bar{z}$ is the binary representation of a state in Y. For two sets Y and Z, its union $Y \cup Z$ is represented by $Y(\bar{z}) \vee Z(\bar{z})$, and similarly for other set operations. Pre-image computation can also be expressed easily in this framework:

$$\mathbf{pre}(Y(\bar{s})) = \exists \bar{t}.f_R(\bar{s},\bar{t}) \wedge Y(\bar{t}).$$

Since Boolean quantification can be eliminated, the result of $\mathbf{pre}(Y(\bar{s}))$ is again a Boolean function. We conclude that all operations in the fixed-point algorithm can be computed symbolically.

For a practical implementation, it is necessary to have a data structure for Boolean functions which (i) facilitates good compression capabilities, but

(ii) makes it easy to recognize that a fixed point has been reached. Ordinary Boolean functions have the disadvantage that deciding the termination condition $Y(\bar{z}) \equiv Y'(\bar{z})$ of the fixed-point algorithm is coNP-complete, and thus a computationally hard problem. A successful trade-off is achieved by *OBDDs* [5]. OBDDs are compact (although somewhat less so than Boolean functions), and, importantly, they have canonical representations which makes it easy to decide $Y(\bar{z}) \equiv Y'(\bar{z})$ efficiently.

OBDDs

Let AP be the set of propositional variables, and $<$ a linear order on AP[1]. An OBDD O over AP is an acyclic graph (V, E) whose non-terminal vertices (*nodes*) are labelled by variables from A, and whose edges and terminal nodes are labelled by 0, 1. Each non-terminal node v has out-degree 2, such that one of its outgoing edges is labelled 0 (the *low edge* or *else-edge*), and the other is labelled 1 (the *high edge* or *then-edge*). If v has label a_i and the successors of v are labelled a_j, a_k, then $a_i < a_j$ and $a_i < a_k$. In other words, for each path, the sequence of labels along the path is strictly increasing with respect to $<$.

Each OBDD node v represents a Boolean function O_v. The terminal nodes of O represent the constant functions given by their labels. A non-terminal node v with label a_i whose successors at the high and low edges are u and w, respectively, defines the function $\mathsf{O}_v := (a_i \wedge \mathsf{O}_u) \vee (\neg a_i \wedge \mathsf{O}_w)$. For every variable order $<$ and Boolean function f there exists a *canonical minimal* OBDD O over AP which represents the Boolean function f. Given any OBDD for f which respects $<$, the canonical OBDD O can be computed in polynomial time. Thus, with OBDDs, set operations including equality testing of sets can be efficiently complemented. Pre-image computation, however, is much harder; in fact, it is one of the major bottlenecks in verification. Another problem with OBDDs is the prevalence of state explosion: for certain functions, the size of the minimal OBDD may be exponential in AP, and moreover the size crucially depends on the variable order $<$. A simple example of a binary decision diagram (BDD) is given in Fig. 3.

LTL verification: bounded model checking.

Bounded model checking [3] is a new method which leverages the surprising power of recent SAT solvers, i.e., algorithms which on input of a Boolean formula search for a satisfying assignment[2]. Recall that LTL specifications express properties which have to hold over *all* paths; consequently, a counterexample for an LTL property is given by a single path which violates the

[1]In this section we assume for simplicity that all states in the Kripke structure are uniquely identified by their labels, i.e., that the labelling function L is injective.

[2]Note that Boolean satisfiability is a prototype NP-complete problem, and thus we cannot expect a SAT solver to scale polynomially for all inputs. However, state-of-the-art SAT-solvers are remarkably successful at a large portion of those formulas which occur in practical verification tasks.

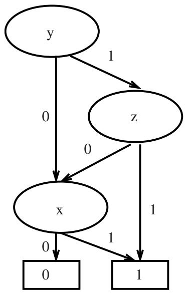

Fig. 3. A BDD for $(y \wedge z) \vee (y \wedge \neg z \wedge x)$. Note that the size of the diagram in this case is linear in the number of Boolean variables.

specification. Consider for example a specification of the form $\mathsf{G}b$, i.e., "always b". Then a counterexample for $\mathsf{G}b$ is a path where at some point $\neg b$ holds. Suppose that for a state s, $b(\bar{s})$ is the Boolean formula expressing that b holds at state s. Then the formula

$$f_R(\bar{s}_0, \bar{s}_1) \wedge f_R(\bar{s}_1, \bar{s}_2) \wedge \ \cdots \ \wedge f_R(\bar{s}_{k-1}, \bar{s}_k) \wedge \bigvee_{0 \le i \le k} \neg b(\bar{s}_i)$$

is satisfiable if and only if a counterexample of size $\le k$ exists; consequently, a SAT solver can be used to decide the existence of a counterexample. Similarly, it can be shown that for any fixed counterexample length k, any LTL specification can be translated into a SAT instance. Moreover, the satisfying assignment computed by the SAT solver can be used easily to compute the counterexample.

Bounded model checking has been used successfully for both hardware and software, and has outperformed BDD-based methods on various examples. The evident drawback of bounded model checking, however, is its inherent incompleteness: unless the bound k is chosen to be significantly larger than $|S|$, bounded model checking provides no assertion about the total absence of counterexamples. Consequently, bounded model checking is mainly considered a method to find errors rather than a complete verification tool.

3.2 Counterexample-guided abstraction refinement

Abstraction reduces the state space by removing irrelevant features of a Kripke structure. Given a Kripke structure K, an abstraction is a Kripke structure $\widehat{\mathsf{K}}$ such that $\widehat{\mathsf{K}}$ is significantly smaller than K, and $\widehat{\mathsf{K}}$ *preserves* a useful class of specifications for K. Consequently, the expensive task of model checking K can be reduced to the more feasible task of model checking $\widehat{\mathsf{K}}$. We know from above that in order to preserve all CTL specifications, K and $\widehat{\mathsf{K}}$ must be bisimilar. But bisimilarity, by its very definition, expresses that K and $\widehat{\mathsf{K}}$ are behaviorally equivalent. Consequently, $\widehat{\mathsf{K}}$ still models a lot of irrelevant behavior and will therefore be quite large in general.

A more practical approach is to employ the fact explained in Section 2 that simulation preserves ACTL* formulas, i.e., $\mathsf{A} \preceq \mathsf{B}$ and $\mathsf{B} \models \phi$ imply $\mathsf{A} \models \phi$. Consequently, for an abstract system $\widehat{\mathsf{K}}$ where $\mathsf{K} \preceq \widehat{\mathsf{K}}$ holds, a successful run of the model checker over $\widehat{\mathsf{K}}$ implies correctness over the original Kripke structure K, *without model checking* K. The converse implication, however, will not hold in general: an ACTL* property which is false in $\widehat{\mathsf{K}}$ may still be true in K. In this case, the abstract counterexample obtained over $\widehat{\mathsf{K}}$ cannot be reconstructed for the concrete Kripke structure K, and is called a *spurious counterexample* [10], or a false negative.

An important instance of simulation-based abstraction is *existential abstraction* [11, 14] where the abstract states are essentially equivalence classes of concrete states; a transition between two abstract states holds if there was a transition between any two concrete member states in the corresponding equivalence classes. Formally, an abstraction function h is a surjection $h : S \to \widehat{S}$ where $\widehat{S}$ is the set of *abstract states*. The surjection h induces an equivalence relation $\equiv$ on the state space S where $d \equiv e$ iff $h(d) = h(e)$. The abstract Kripke structure $\widehat{\mathsf{K}} = (\widehat{S}, \widehat{S_0}, \widehat{R}, \widehat{L}, \mathrm{AP})$ derived from h is defined as follows:

$$\begin{aligned}
\widehat{S_0} &= \{\widehat{d} \mid \exists d \in S_0 \,.\, h(d) = \widehat{d}\} \\
\widehat{R} &= \{(\widehat{d_1}, \widehat{d_2}) \mid \exists d_1, d_2 \in S \,.\, h(d_1) = \widehat{d_1} \wedge h(d_2) = \widehat{d_2} \wedge R(d_1, d_2)\} \\
\widehat{L}(\widehat{d}) &= \bigcup_{h(d)=\widehat{d}} L(d)
\end{aligned}$$

We also write $\widehat{\mathsf{K}} = \mathsf{K}/\!\equiv$ to express the dependence of $\widehat{\mathsf{K}}$ on $\equiv$. An atomic proposition $f \in \mathrm{AP}$ *respects* an abstraction function h if for all d and d' in the domain S, $(d \equiv d') \Rightarrow (d \models f \Leftrightarrow d' \models f)$. When the specifications contain only atomic propositions respecting h, we can without loss of generality assume that in both K and $\widehat{\mathsf{K}}$, AP is restricted to the propositions occurring in the specifications. Then existential abstraction indeed guarantees $\mathsf{K} \preceq \widehat{\mathsf{K}}$ as intended.

However, determining a good abstraction function h is a difficult task: If $\widehat{K}$ is too large, then verification remains infeasible. If, on the other hand, $\widehat{K}$ is too small, then spurious counterexamples are likely to occur, as illustrated by the next example. The example also illustrates that abstraction typically introduces non-determinism.

Example 4. Fig. 4 shows two abstract Kripke structures $\widehat{\mathsf{M}_1}$ and $\widehat{\mathsf{M}_2}$ obtained from the original Kripke structure M by two different equivalence relations $\equiv_1$ and $\equiv_2$. M describes a simplified bus arbiter that controls access on two buses. The arbiter chooses one of the two buses for any request before giving a grant. It asserts A, B and C to suggest the slave to use bus 1; otherwise, it asserts D, E and F to suggest the slave to use bus 2 and then gives a grant. The slave is supposed to probe which of the lines have been asserted

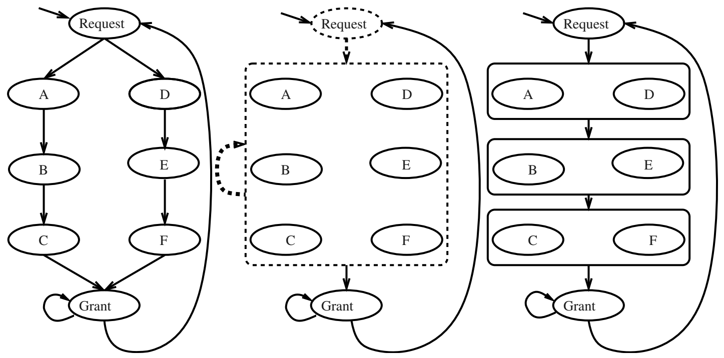

Fig. 4. The Kripke structure M on the left is existentially abstracted in two ways, yielding abstract Kripke structures $\widehat{\mathsf{M}_1}$ (center figure) and $\widehat{\mathsf{M}_2}$ (right figure). Note that $\widehat{\mathsf{M}_1}$ contains an infinite path which never reaches the state labelled Grant; being unique to the abstract model, this path is called a spurious path.

when it receives the grant and use the appropriate bus. Let us consider the specification $\mathsf{AG}(request \rightarrow \mathsf{AF}grant)$ which says that every request is finally granted. By manual inspection we see that the specification holds true for M. The first abstraction $\widehat{\mathsf{M}_1}$ of the arbiter, however, is too coarse, and does not allow us to prove correctness of the specification: we get a counterexample involving a self-loop, as indicated by the dashed lines in the figure. A finer abstraction $\widehat{\mathsf{M}_2}$ passes the specified property and hence, the property is also true for our original Kripke structure M. Note that in both cases $\mathsf{M} \preceq \widehat{\mathsf{M}_1}$ and $\mathsf{M} \preceq \widehat{\mathsf{M}_2}$, i.e., all universal properties on the abstract model are preserved. In the case of $\widehat{\mathsf{M}_1}$, however, preservation does not help, since $\widehat{\mathsf{M}_2}$ exhibits too much information loss for the specification to hold.

Counterexample-guided abstraction refinement (CEGAR) is a natural approach which resolves this situation by using an adaptive algorithm which starts with a coarse abstraction and gradually improves the abstraction function by analyzing spurious counterexamples. CEGAR-style approaches have been investigated by several researchers beginning with the *localization reduction* of Kurshan [23] where the model is abstracted/refined by removing/adding variables from the system description. The first systematic account of CEGAR for CTL model checking was given in [10]. We describe the CEGAR loop using the equivalence relation $\equiv$ induced by h in Fig. 5.

In the CEGAR loop, the abstraction is refined until the property is either verified or disproved by a non-spurious counterexample. Note that the CEGAR loop involves two crucial steps in addition to model checking: the computation of the initial relation $\equiv$, and the computation of the refined abstraction. The initial abstraction is usually obtained by static analysis of the input program (cf. also Section 3.3), and the refinement is achieved by projecting $\widehat{\mathsf{C}}$ back onto K, determining where the spurious behavior occurs, and

Counterexample Guided Abstraction Refinement

```
$R_\equiv$ := initial state equivalence;
result := empty;
**repeat**
    $\widehat{\mathsf{K}} := \mathsf{K}/R_\equiv$
    call model checker for $\widehat{\mathsf{K}} \models \phi$
    **if** $\widehat{\mathsf{K}} \models \phi$
        **then** result := "specification true";
        **else** compute counterexample $\widehat{\mathsf{C}}$;
            **if** $\widehat{\mathsf{C}}$ is spurious
                **then** $R_\equiv$ := **refine**$(\mathsf{K}, \widehat{\mathsf{C}}, R_\equiv)$;
                **else** result := $\widehat{\mathsf{C}}$;
**until** result not empty;
```

Fig. 5. General scheme for CEGAR. For better readability, the relation $\equiv$ is written $R_\equiv$ in the program text.

locally refining $\equiv$ to eliminate $\widehat{\mathsf{C}}$. Since $\widehat{\mathsf{C}}$ typically is much smaller than $\widehat{\mathsf{K}}$, the projection of $\widehat{\mathsf{C}}$ back onto K involves only a small portion of the state space of K, and is therefore feasible in many practical cases. CEGAR frameworks have become a widely used paradigm in verification, and are routinely used for both hardware and software.

3.3 Model checking for infinite state systems

Model checking was originally designed for the verification of finite state systems. Although the first practically useful applications of model checking were oriented towards hardware verification, where the finite state restriction comes naturally, the method was conceived of as an approach to software verification as well. The early papers on model checking clearly drew their motivations from the software area, focusing in particular on concurrency properties to be verified over the synchronization skeleton of a program, i.e., a finite abstract model which preserves the relevant behavior for interprocess communication. It is still the case that abstraction is one of the key methods to be used for software verification; in fact, model checking and abstract interpretation [14] share many common techniques which deserve further exploration. In this section, we will concentrate on predicate abstraction [19], a particularly important abstraction method which underlies the recent advancements in software verification exemplified by tools such as BLAST, SLAM and MAGIC [2,7,20]. We present a variant of predicate abstraction as it is used in MAGIC.

Predicate abstraction

For the analysis of software, the simplest abstraction which arguably represents the program behavior in a meaningful way is the control flow graph

(CFG) which can be viewed as a Kripke structure where the states are program counter positions, and the transitions denote non-deterministic changes of the control flow. Note that the CFG can be seen as an existential abstract model where the abstraction function h abstracts away everything except the program counter information. It is evident that for many properties of interest, the CFG does not contain sufficient information for verification.

The technique of predicate abstraction [19] is based on the observation that what often counts in the analysis of a program is not so much the actual values of the variables, but rather their relation to each other. Important relations between variables are expressed, for example, in the control conditions which occur in if-statements and in loop headers. Thus, instead of keeping 64 bits for two integer variables x, y in our global state, we may have a single bit which keeps the truth value of the *predicate* $x > y$ if this is the property of interest. In predicate abstraction, we define k such predicates, and extend the CFG by the different evaluations of the predicates. The state space of the new system is 2^k times the size of the CFG, and by choosing the number k of predicates, we can obtain a trade-off between preciseness and state explosion. Thus, in our model, each state of the extended CFG can be described by a formula Ψ of the form

$$(\text{ProgramCounter} = \text{i}) \ \wedge \bigwedge_{1 \le i \le k} \psi_i,$$

where the ψ_i are predicates or negated predicates ranging over the variables of the program. Such a formula Ψ can be identified with an abstract state representing all concrete states (i.e., memory contents) which satisfy Ψ. However, the tricky part in predicate abstraction is to define the transition relation: Suppose that we have a transition in the CFG between program counter positions i and j through a simple statement `statement`, and 2^k abstract states each for i and j, i.e., $\Psi_{i,1}, \ldots, \Psi_{i,2^k}$ and $\Psi_{j,1}, \ldots, \Psi_{j,2^k}$. Potentially, the single transition in the CFG gives rise to up to $(2^k)^2$ transitions in the extended CFG. We actually include a transition from $\Psi_{i,a}$ to $\Psi_{j,b}$ if the weakest precondition required for $\Psi_{j,b}$ to hold after execution of `statement` *is consistent* with $\Psi_{i,a}$, i.e., if

$$\Psi_{i,a} \wedge \text{WP}[\texttt{statement}, \Psi_{j,b}]$$

is logically satisfiable. Deciding satisfiability is in general a hard question, and is often delegated to an automated theorem prover or a decision procedure.

It is not hard to prove that the model obtained by predicate abstraction fits into the simulation-based approach to abstraction; in fact, it is not even necessary for the theorem prover to always terminate. When a theorem prover does not produce a definite answer in due time, we can overapproximate the result by pretending that the theorem prover asserted consistency. It can be shown that the resulting model is still a sound abstract model in this case; if this happens too often, however, the quality of the model will deteriorate towards a very coarse model with extensive spurious behavior. Importantly, predicate abstraction provides a natural and clean interface between model checking and theorem proving, playing to the strengths of both methods. Predicate abstraction is very successful when verifying control-intensive software such as device drivers or many embedded programs. For complicated data structures, in particular dynamic data structures, predicate abstraction is potentially applicable, but the logics and corresponding decision procedures are in most cases still beyond the state of the art.

Other approaches to infinite systems

Let us finally discuss the question of verification of infinite systems on a broader scale. Precisely speaking, most of the systems we consider are not infinite, but rather parameterized: they are described by a finite program text, and the actual size of the state space depends on a parameter (e.g., the memory size) which is not known in advance, and is often assumed to be arbitrarily large. Besides memory size, other sources of unbounded behavior include recursion depth, the preciseness of floating-point operations, dynamic thread creation, protocols with unlimited number of participants and hybrid systems where parts of the system or the environment are modeled by differential equations as in control theory. The different flavors of infinity we encounter in applications clearly need to be matched by a correspondingly rich set of tools, each tailored for a specific source of infinity. However, in contrast to finite state verification, we cannot realistically expect to find methods which apply uniformly to all kinds of infinite state systems. Due to space limitations, we will just briefly mention some of the promising current approaches.

Classically, Petri nets have been used to model concurrent processes using a single transition graph. More recently, verification methods for pushdown systems have been described [1, 17] which enable the direct modelling of unbounded calling stacks. An approach mainly geared at parameterized systems is regular model checking [4] where the infinite transition relation is described by finite automata, rendering many reachability properties decidable. Important progress in modelling dynamic data structures has been made in a three-valued framework [27]. A promising special form of predicate abstraction for hybrid systems has recently been proposed by Tiwari [29] who suggested the use of predicates describing analytical properties of functions such as $\frac{\mathrm{d}g}{\mathrm{d}x} > 0$.

4 Conclusion

In the twenty-five years since its invention, model checking has developed into a highly active research area of its own right which combines algorithms, logic, (discrete) mathematics and of course application knowledge. Although model checking is usually simpler to apply than theorem proving, it is still not always easy for engineers with the right application knowledge but without formal training in verification to use model checking to its full capability. As expressed in Rushby's notion of "disappearing formal methods," we expect that for many settings model checking will finally become a push-button technology similar to compilers in which the trade-off between the preciseness and the computational cost of the correctness analysis can be controlled by a few simple parameters. Generally though, the principal undecidability of virtually all questions in software verification makes clear that there is no silver bullet for verification, and there will always be a need to design model checking methods specific to problem classes.

While verification of hardware and software systems will evidently remain a core concern of model checking, there are also exciting new avenues of research which often involve the combination of traditional model checking techniques with continuous mathematics, most notably the verification of stochastic, hybrid and biological systems.

Acknowledgments

The authors extend their gratitude to Stefan Katzenbeisser for his careful proofreading.

References

1. R. Alur, K. Etessami, and P. Madhusudan. A Temporal Logic of Nested Calls and Returns. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of LNCS, pages 467–481, 2004.
2. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proc. Model Checking Software, 8th International SPIN Workshop*, volume 2057 of LNCS, pages 103–122, 2001.
3. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th Conference on Design Automation (DAC)*, pages 317–320, 1999.
4. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 403–418, 2000.
5. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35(8), pages 677–691, 1986.
6. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *In Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *Proc. 25th Int. Conference on Software Engineering (ICSE)*, pages 385–395, 2003. Extended version in IEEE Transactions on Software Engineering, 2004.
8. E. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs: Workshop*, volume 131 of *LNCS*, pages 52–71, 1981.

9. E. Clarke, T. Filkorn, S. Jha. Exploiting Symmetry In Temporal Logic Model Checking. *Proc. Computer Aided Verification (CAV)*, volume 697 of LNCS, pages 450–462, 1996.
10. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 154–169, 2000. Extended version in *J. ACM* 50(5): 752–794, 2003.
11. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
12. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, MA, 1999.
13. E. Clarke and H. Schlingloff. Model checking. In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1367–1522. Elsevier, Amsterdam, 2000.
14. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.
15. E. Emerson. Temporal and modal logic. In J. van Leeuven, editor, *Handbook of Theoretical Computer Science, Vol. B.*, pages 995–1072. Elsevier, Amsterdam, 1990.
16. E.A. Emerson and A.P. Sistla. Symmetry and model checking. *Proc. Computer Aided Verification (CAV)*, volume 697 of LNCS, pages 463–478, 1996.
17. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 232–247, 2000.
18. P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. Computer Aided Verification (CAV)*, volume 531 of LNCS, pages 176–185, 1990.
19. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proc. Computer Aided Verification (CAV)*, volume 1254 of LNCS, pages 72–83, 1997.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proc. ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 58–70, 2002.
21. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, Reading, MA, 2003.
22. M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, London, 1999.
23. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ, 1994.
24. K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, Dordrecht, 1993.
25. A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science (FOCS)*, pages 46–67, 1977.
26. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Symposium in Programming*, volume 137 of *LNCS*, pages 337–351, 1982.